Skew Characteristics and Their Effects on Parallel Relational Query Processing

Kevin Hao Liu

Skew Characteristics and Their Effects on Parallel Relational Query Processing

Kevin Hao Liu

This thesis is presented in fulfilment of
the requirements of the degree of
Doctor of Philosophy

Department of Computer and Mathematical Sciences
Faculty of Science

Victoria University of Technology

March 1997

ABSTRACT

As queries grow increasingly complex and large data sets are becoming prevalent, database sizes grow dramatically particularly in *Decision Support Systems* (*DSS*), and *On-Line Analytic Processing Systems* (*OLAP*) which have recently emerged as important database applications. In these systems, performance is a critical issue and speeding up the system has always been an objective but the processing power of individual processors can only handle a small fraction of current applications. As a result, parallel processing is exploited to improve database systems performance. In the thesis we focus on relational database systems and study skew characteristics and their effects on parallel query processing.

A unique problem of parallelism is load imbalance that occurs when the processors have different workloads resulted from data partitioning. The main cause of load imbalance is the problem of data skew that takes place when the appearance of some attribute values is more frequent than others in the *raw* input relation. The study of skewness can be classified into empirical skew handling and theoretical skew prediction. The latter has been largely neglected in comparison with the former. This thesis establishes a systematic skew foundation which we believe provides for the first time an in depth theoretical investigation of skewness in parallel database systems. It consists of a skew taxonomy which furnishes a fundamental framework for studying the skew problem, and skew prediction models which are able to reinforce and unify the different elements of the skew taxonomy. Skew prediction of range partitioning is quantitatively described. A complete skew prediction model of hash partitioning is also provided to predict the load and operation skew, where the degree of data skew is represented by the *Uniform Distribution*, *Zipf Distribution*, and *Normal Distribution* families. The relationship between the various kinds of skewness is expressed in a closed form. The mean, maximum, and minimum load skew as well as the standard deviation of the mean load and their distribution functions are provided. The skew models are evaluated by both a detailed simulation study and an experimental implementation on a parallel client-server system, and the results exhibit close agreement with the predicted values. In addition, an analysis of the relationship among data skew, load skew, and operation skew of hash partitioning is provided, and the applications of the skew models are indicated.

Another unique problem in parallel relational query processing is concerned with resource allocation, and in the thesis we focus on the resources and processors, in a shared-nothing environment. Processor allocation deals with efficient allocating processors to operations

in such a way that the query execution time is minimised. The problem of data skew has been recognised as one of the main obstacles in preventing maximum speedup in parallel query execution. Although techniques of skew handling have received considerable attention in recent years, most of them are devoted to solving or avoiding data skew for single binary relational operation such as join (intra-operation level only). Here, we study the effect of data skew on the parallel execution of queries (intra-query and inter-query levels) that may involve a number of queries requiring multiple operations. Since allocating a large number of processors to a single relational operation does not always improve the overall execution time significantly in the presence of data skew, our approach is to identify the skewed operations and group them with other operations in each execution phase, so that the performance degradation caused by the skewness is kept to a minimal level. Both unary and binary operations cost models are provided with the data skew factor modelled by the *Zipf Distribution*. Three processor allocation strategies are presented and their performances are evaluated and compared.

Parallelising aggregate functions often involves queries of more than one relation, so that performance improvement can be achieved in both the intra-operation and inter-operation parallelism level. Carrying out join before aggregation is the conventional way for processing aggregate functions in uniprocessor systems, and parallel processing of aggregate functions has received little attention. In this thesis, the effects of the sequence of aggregation and join operations are identified in a parallel processing environment. Three parallel methods of processing aggregate functions for general queries are presented which differ in their selection of partitioning attribute; *JPM* partitions on join attribute, *APM* fragments on group-by attribute, and *HPM* adaptively partitions on both join and group-by attribute with a logical hybrid architecture. Their cost models are provided which incorporate the effect of data skew.

Depending on the size of the search space, processor allocation can be divided into phase-based approach and non phase-based approach. A formal classification of the two approaches is presented and the concepts of the optimal degree of parallelism for each operation and query are introduced. A complete query execution model is developed by incorporating the effects of skew and parallel processing overheads; the optimal time for each operation and query and the time equalisation principle are presented. To provide efficient query processing, three intra-query algorithms are developed; one is a phase-based approach, one is non-phase-based approach, and the other is a hybrid method. The hybrid method guarantees to provide a global optimal solution with a sufficient number of processors and a local phase optimisation is performed when the number of processors is insufficient. Furthermore, we present a new inter-query processor allocation algorithm aiming at enhancing the performance of multiple dependent queries by making use of activity analysis, resource scheduling, resource leveling, and decompression techniques. A comprehensive performance study on processor allocation algorithms for both intra-query and inter-query parallelism levels is conducted. In simulation at the intra-query level, five algorithms are implemented and a large number of queries are selected from five different query groups varying the number of joins and relation cardinality; at the inter-query level, three algorithms and two types of query dependency logic are implemented. The results are able to confirm that the new algorithms are superior to the existing methods.

ACKNOWLEDGMENTS

My deep gratitude first goes to my research supervisor, Prof. Clement Leung, for supporting my work and for his excellent technical and professional guidance. He has allowed me to pursue my own interests while also giving me the opportunity to participate in group projects. He has contributed immeasurably to both my professional and personal development.

I would like to thank Dr. Yi Jiang for his proofreading the early draft of this thesis and for his invaluable technical advice on query processing in distributed database systems. I have enjoyed working with all other members of the Department of Computer and Mathematical Sciences, and special thanks go to Prof. Peter Cereone, Ted Alwast, and Dr. Audrey Tam. Many thanks are also due to the members of Terabyte Database Research Group, and a special memory is given to Jane Shou who passed away after a seven-year battle with leukaemia.

I would like to thank all my fellow students, particularly, David Taniar, Ivan Justrisa, Simon So, Philip Tse, Refyul Ray Fatri, Savitri Bevinakoppa, Dwi Sutanto, and Pak-Fai Tang for their valuable help. I also would like to thank Wai-Foong Hong and He-Pu Deng for being special friends and making my life more pleasurable and interesting. Many thanks go to my college friend, Guo-Hua Sheng, and our continuous communications have always been an inspiring source in my life.

I would like to thank my parents and my sister for their constant love and encouragement. Special thanks go to my mum for her impatiently urging me to finish, for sharing my anxiety, and for never having doubted me.

Throughout my candidature I was fully supported by an Australian Postgraduate Research Award, and in 1994 I was supported by a Research Assistantship from the Collaborative Research Group. Finally, I am also grateful to the Postgraduate Committee of Department of Computer and Mathematical Sciences and the Victoria University of Technology for backing the applications that made this degree possible.

DECLARATION

The material in the thesis is the product of the author's own independent research carried out at the Department of Computer and Mathematical Sciences of Victoria University of Technology under the supervision of Professor Clement Leung.

The thesis consists of eleven chapters and three appendixes. The first two chapters contain introductory and background material, while the last chapter contains the conclusions. Appendix A presents a mathematical derivation relating to Chapter 5; Appendix B introduces the Terabyte Database Simulation Model also in Chapter 5; Appendix C explains the simulation model for intra-query processor allocation relating to Chapter 9. Some of the materials presented in Chapters 3, 4, 5, 6, 7, 8, 9, and 10 of this thesis have been published before in various refereed conference proceedings and journals (see next page) and technical reports. The thesis is less than 100,000 words in length.

Kevin Hao Liu

March 1997

List of External Refereed Publications

1. Liu K. H. and C. H. C. Leung, "Multiple queries execution using critical path scheduling in parallel databases", to appear, *Proceedings of the IEEE 3rd International Conference on Algorithms and Architectures for Parallel Processing* (ICA3PP'97), World Scientific Press, Melbourne, Australia, 1997

2. Liu K. H., Y. Jiang, and C. H. C. Leung, "An intelligent agent for adaptive processor allocation in parallel databases", to appear, *Proceedings of the IEEE 1st International Conference on Intelligent Processing Systems* (ICIPS'97), Beijing, China, 1997

3. Liu K. H., Y. Jiang, and C. H. C. Leung, "Parallel processing of aggregate functions in the presence of partition skew", to appear, *The Computer Journal*, 1997 (accepted for publication)

4. Liu K. H., "Design and evaluation of optimal processor assignment on parallel relational query processing", to appear, *Proceedings of the 11th ACM SIGARCH International Conference on Supercomputing* (ICS'97), Vienna, Austria, 1997

5. Liu K. H., "Performance evaluation of processor allocation algorithms for parallel query execution", to appear, *Proceedings of the 12th Annual ACM Symposium on Applied Computing* (SAC'97), San Jose, California, USA, 1997

6. Liu K. H., "Improving database systems performance with parallel processing", *Proceedings of the Global Information and Software Society Internet Realtime Conference* (GISSIC'96), New York, USA, 1996 (http://pride-i2.poly.edu/GISS)

7. Liu K. H., C. H. C. Leung, and Y. Jiang, "Processor allocation algorithm with minimal number of processors achieving optimal time", *Proceedings of the 3rd Australasian Conference on Parallel and Real-Time Systems* (PART'96), Brisbane, Australia, pp 51-59, 1996

8. Liu K. H. and D. Taniar, "Efficient processor allocation in parallel Object-Oriented databases", *Proceedings of the 3rd Australasian Conference on Parallel and Real-Time Systems* (PART'96), Brisbane, Australia, pp 164-172, 1996

9. Liu K. H., "Load balancing algorithms for hash partitioned unary relational operations", *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications* (PDPTA'96), Sunnyvale, California, USA, pp 671-675, 1996

10. Liu K. H., Y. Jiang, and C. H. C. Leung, "Query execution in the presence of data skew in parallel databases", *Australian Computer Science Communications*, Vol. 18, No. 2, pp 157-166, 1996

11. Liu K. H., C. H. C. Leung, and Y. Jiang, "Analysis and taxonomy of skew in parallel databases", *Proceedings of the International High Performance Computing Symposium* (HPCS'95), Montreal, Canada, pp 304-315, 1995

12. Taniar D., K. H. Liu and C. H. C. Leung, "Factors affecting performance speed-up of parallel Object-Oriented systems", to appear, *Proceedings of the 4th Australasian Conference on Parallel and Real-Time Systems* (PART'97), Springer-Verlag, Newcastle, NSW, Australia, 1997

13. Jiang Y., C. H. C. Leung, and K. H. Liu, "A comparative evaluation of phase-based and non-phase-based processor allocation for parallel query execution", *Proceedings of the 3rd Australasian Conference on Parallel and Real-Time Systems* (PART'96), Brisbane, Australia, pp 43-51, 1996

TABLE OF CONTENTS

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

1.1 Statement of the Problem

Parallelism has been proved to be effective in improving system performance in relational databases. The push comes from both application and technology. From the application points of view, queries are growing large and complex; large data sets are common, and more importantly, many relational operations are naturally independent. From the technology points of view, processors are achieving rapid improvement while the hardware, software and communication network are continually being enhanced. Therefore, the inexpensive processors may be connected to form a network in order to share the huge workload of incoming queries. As a result, we have a parallel database system where a high performance and high availability database server running on a multiple processors architecture is available at a low price [DeWi92a, Moha94, Vald93, Pira90, Hasa95].

Due to the increasing complexity in database applications such as scientific computing or engineering and decision support systems, load balancing over multiple processors is

playing an important role in such systems. Ideally, load balancing is done by evenly distributing the load into the available processors, and thus if it is perfectly done, the system throughput will improve by a factor equal to the number of parallel processors. However, this is hard to achieve because of extra overheads induced by adding processors, and more importantly the presence of skew [Laks90, Walt91, Hua91, Kell91, DeWi92b, Wolf93a, Wolf93b] which resulted from workload partitioning. In real databases, certain values (normally, non-primary key values) for a given attribute occur more frequently than others. The above non-uniformity distribution of tuples and key values gives rise to skew. In the presence of skew, an unbiased partitioning strategy (range partitioning or hash partitioning) will result in load imbalance. This worsens the response time of the algorithms since other processors have to wait for the heavily loaded processor(s) to complete. In extreme cases, one processor is doing all the work, while other processors are waiting idle with wasted CPU cycles. Hence, the system performance is even worse than that of uniprocessor because of communication overheads or network contentions associated with initiating and terminating a query on multiple processors.

In uniprocessor database systems, query processing has been an active research area for sometime since multiple queries with multiple operations introduce a number of optimisation options [Ioan95, Ioan91, Cope88, Jark84, Kim82]; in parallel database systems, the situation on query processing has been further complicated by multiple processors [Hasa95, Hasa94, Hong92, Hong93]. The unique problem in a parallel system is resource allocation because there are resources privately owned by each processing unit. Public resources do not have problem on allocation, but they do give rise to contention problem. There are three basic system architectures for parallel databases, namely, shared everything, shared nothing, and shared disk systems. In this thesis, a shared nothing architecture is adopted for query processing because of its scalability and cost efficiency. In such a system, resources such as processor, memory, and disk are all distributed over the network. When queries come in, how to allocate resources efficiently and effectively in particular in the presence of data skew so that the query can be finished in less time, is a crucial problem.

1.2 Summary of Main Results

This thesis concentrates on skewness and the work can be summarised in two parts; skew characteristics and skew effects on query processing (see Figure 1.1). To achieve the

objective on load balancing, avoiding skew and allocating resource[1] must be carried out carefully. With the task of skew avoidance, all existing works are concerned with practical skew handling algorithms. In this thesis, we investigate the skew characteristics and establish a skew foundation formed by a skew taxonomy and a complete skew model. With the task of processor allocation in query processing, we focus on the situation in the presence of data skew. Several processor allocation algorithms are developed to achieve global optimised query execution time and to minimise the skew effect. Parallel processing methods of aggregate functions are also developed.

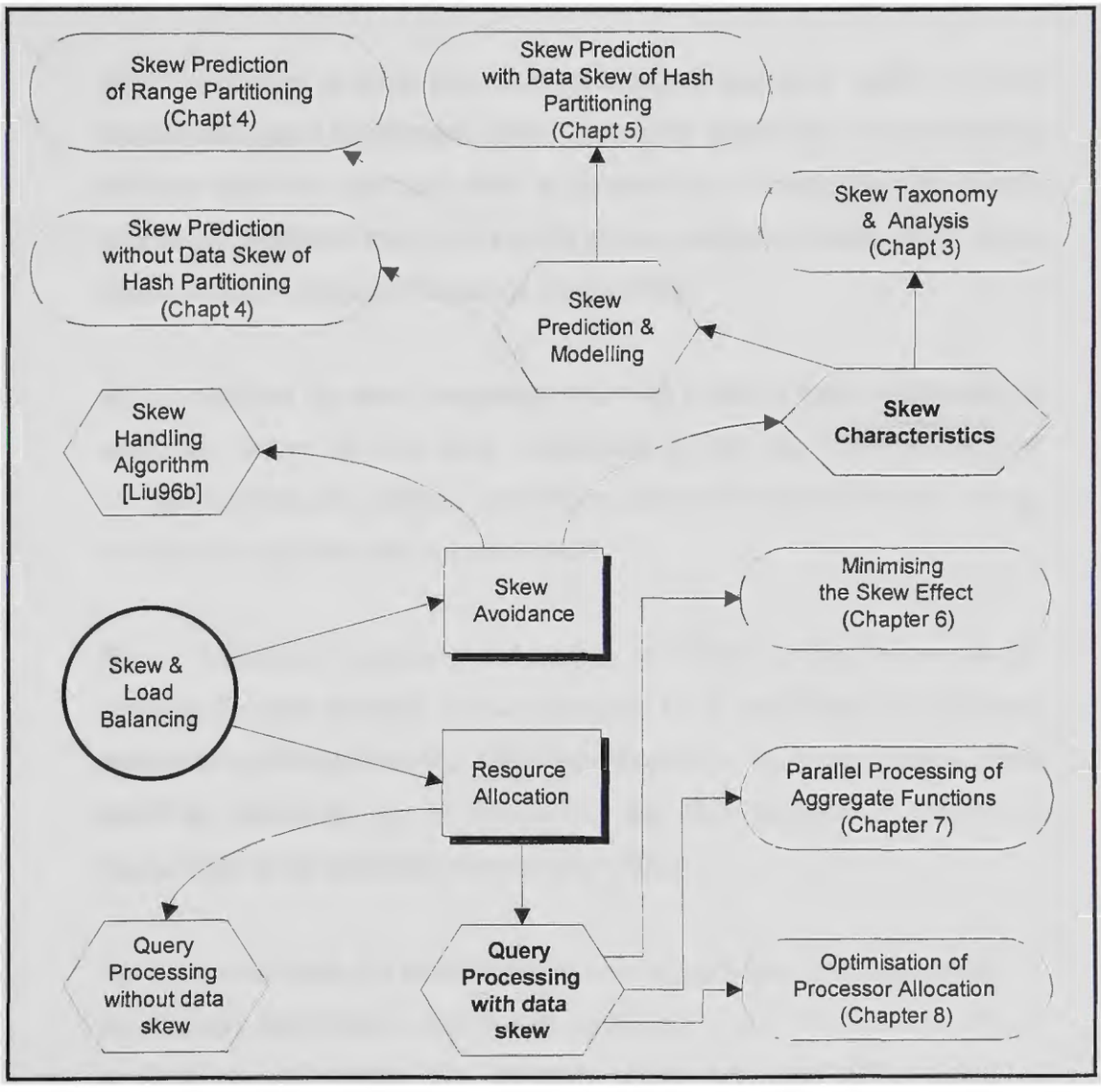


Figure 1.1: Thesis Overview

To sum up, the contributions of this thesis are as follows:

[1] In this thesis, we only consider one kind of resource, i.e. processor.

- Provides for the first time a theoretical study of skewness in parallel database systems in depth; a conceptual framework of skew is presented and a skew taxonomy is developed with three kinds of skewness, namely, data skew, load skew, and operation skew identified; load skew is further distinguished into I/O load skew, operation load skew, and result load skew; all skewness are quantitatively analysed and their expressions are provided; performance bounds on best and worst case behaviour are also presented.

- Provides a skew prediction model based on range partitioning.

- Provides a systematic skew foundation analytical model of hash partitioning based on extreme value distribution properties; the relationship between data skew and load skew is expressed in a closed form; the mean, maximum, minimum load skew as well as the standard deviation of the mean load and their distribution functions are provided.

- Applies the skew foundation analytical model of hash partitioning to study the degree of data skew represented by the *Zipf Distribution* and *Normal Distribution* families; provides an analysis of the relationship among data skew, load skew, and operation skew.

- Using skew taxonomy and analysis, provides a unified framework for studying the skew problem in load balancing, i.e. it establishes a relationship among the different elements of the skew taxonomy; allowing different skew handling algorithms to be compared, and also provides a theoretical background of the entire load balancing problem.

- Investigates the parallel processing of aggregate functions which is increasingly important in data warehousing and mining for decision support applications, and studies the important issues associated with it such as selection of partition attribute, sequence of aggregation and join operation.

- Develops parallel methods for the processing aggregate functions for general queries; their cost models are presented, and the performance of these

methods are studied.

- Develops a new processor allocation algorithm aiming at minimising the skew effect exploiting not only inter-operator parallelism but also intra-operator parallelism; quantifies the *Skew Principle* whereby allocating a large number of processors to a single operation does not improve its execution time.

- Provides a complete parallel query execution model incorporating the parallel processing overheads and the load imbalance in massive parallel systems; the concepts of optimal degree of parallelism and optimal time for operations, and time equalisation are introduced; the optimal query time and processor bounds are presented.

- Develops new intra-query processor allocation algorithms: phase-based, non phase-based, and the hybrid methods; the hybrid method guarantees to provide a global optimal solution with a large number of processors, and provides a local phase optimal solution where the number of processors is insufficient.

- Develops a new inter-query processor allocation algorithm aiming at enhancing the performance of multiple dependent queries by making use of activity analysis, resource scheduling, resource leveling, and decompression techniques.

- Presents a comprehensive performance study of processor allocation algorithms in both intra-query and inter-query parallelism levels. In the simulation at intra-query level, five algorithms are implemented; in the simulation at inter-query level, three algorithms and two kinds of queries dependency logic are implemented.

- Implements the parallel relational database processing and the skew models on a parallel client-server shared everything system; the server is a DEC Alpha 2100 system and the operating system is Digital UNIX with SMP and processor affinity; the test database is generated using the synthetic

methods and is stored in RAID; the inter-processor communication is implemented as the piped data channel by the writer and reader process; to achieve mutual exclusion, spin locks of master-slave style are selected and implemented as the multiple processor concurrency control mechanism.

1.3 Outline of the Thesis

Figure 1.1 shows an overview on thesis which can be grouped into two areas, skew characteristics, and skew effects on parallel relational query processing. The rest of this thesis is organised as follows.

Chapter 2 presents a survey on background information and topics which include parallel relational database, parallel vs. distributed DBMS, pyramid of parallel database processing, data partitioning, data processing, and parallel performance modelling.

Chapter 3 presents the skew taxonomy and analysis, and it also discusses performance degradation caused by skew. Chapters 4 and 5 provide the skew model. Using range partitioning, the prediction on the degree of load imbalance is provided in Chapter 4. Then, the chapter introduces the foundation model of hash partitioning and an urn model description. Chapter 5 describes the extension model of hash partitioning and it also presents an analysis on data skew, load skew, and operation skew. Finally, we conclude the skew modelling with a discussion of its usage.

Chapter 6 describes the system architecture for parallel query processing, the parallel query execution model, and processor allocation method to reduce skew effect. Chapter 7 introduces issues on the parallel processing of aggregate functions and presents three parallel methods followed by a sensitivity analysis. Chapter 8 introduces a complete model of parallelism with parallel processing overheads, communication delay, and optimal degree of parallelism. It presents three intra-query processor allocation algorithms together with query examples. Furthermore, the inter-query processing issues are discussed and a new decompression algorithm is presented for handling multiple dependent queries.

Chapter 9 implements five intra-query and three inter-query processor allocation algorithms using simulation. The simulation model is described first, followed by the

experimental results, and the chapter is concluded with a discussion on processor allocation. Chapter 10 describes the implementation on a shared memory parallel system, DEC Alpha 2100, and the associated results.

Chapter 11 contains the thesis conclusions where a summary of the thesis and its contributions are provided.

CHAPTER 2

BACKGROUND AND RELATIONSHIP WITH OTHER WORK

2.1 Parallel Relational Databases

In this Chapter, we describe the thesis background in Sections 2.1 and 2.2, and then we review the related work in Sections 2.3, 2.4, and 2.5. The mechanisms described in the

thesis are intended to work in an environment where a high performance and high availability relational database server is available at a low price with a multiple processors architecture, i.e. parallel relational database systems [Vald93]. The main focus is on the skewness and we shall emphasise skew characteristics and skew effects throughout this thesis. In the first part of this Chapter, we discuss why parallelism is chosen to improve the performance of relational database, and then present the relationship between parallel and distributed database systems. To clarify parallel relational query processing, we introduce a database processing pyramid in Section 2.2 which shows the boundary and the position of the thesis in the overall picture of parallel relational query processing.

There are two main categories associated with the related work on parallel relational query processing: data partitioning and data processing. As we are also interested in the analytical modelling for skew prediction, a separate section on performance modelling is also presented and reviews the theoretical treatment of parallelism in the literature.

2.1.1 Motivation

The parallel wave of computing has emerged since not only the applications are getting complex as in Decision Support Systems, Image Database Systems and Multi-media applications, but also both the processors and the network technology have improved dramatically. Parallelism has been used in both the hardware and the software levels [Lewi92, Alma94, Casa96, DeWi92a, Vald93, Kuma94]. The former development is more mature than that of the latter, and in the software level parallelisation may be applied to either data or programs. It is generally considered that the parallelisation of data offers much greater scope for concurrent operation than the parallelisation of control, since for a large data file, the degree of parallelism through horizontal table fragmentation can be orders of magnitude higher than that for control parallelisation. In addition, the codes for control parallelisation can be significantly more complex than the corresponding serial codes [Thin93]. Due to the relative ease and benefits of data parallelisation, database systems have become important targets for parallel processing. In addition, the demand for increasingly complex and flexible queries contributes to the adoption of high-performance platforms for database processing.

2.1.2 Relational Databases

The most commercialised and widely accepted database up to now is the relational database where data is stored in tables containing rows and columns. The relationship between these tables is implicitly defined via the relationship between the common attributes in both tables. A common way to construct the relationship between tables is to use the primary key of one table as a foreign key of another table. Only when we want to assess the data in multiple tables according to the query, we use SQL to create the relationship between tables and establish the navigation of the database for that relationship. [Codd70, Codd90] discussed a set of relational rules for a database to be considered as truly relational. Although there are a number of relational database products such as DB2 and SQL/DS from IBM, PDQ from Honeywell, INGRES from Relational Technology, dBASE from Ashton-Tate (now owned by Borland), R:BASE from MicroRIM, INFORMIX from Relational Database Systems, ORACLE from ORACLE running on different hardware platforms, none of the existing relational databases meets all of Codd's criteria.

Being the most widely used database, relational database has its strengths. It offers data independence so that modifying relational tables does not require changing application programs and the addition of new data to the data model rarely demands restructuring of the tables; it offers simplicity and flexibility since the relational database is easy to describe and both users and designers are familiar with the concept of tables; it offers constraints specification so that business rules can be interpreted into constraints and implemented into the system, e.g. entity integrity and referential integrity; it is equipped with 4GL, i.e. SQL so that database navigation is hidden from the programmers and the details of the access paths are taken care of by the SQL optimiser; it has a solid mathematical foundation because the concept of relation is borrowed from mathematics and some of the mathematical optimisation theories may be applied in the query optimiser to improve performance[2] [Date95, Elma94, Codd90].

Relational database is set-oriented and relational operators are independent, and thus data may be partitioned over multiple processors and processed in parallel. Ideally, the partitioning could be done in such a way that the workloads of the processors are balanced,

[2] That is also the reason why relational query optimisation has been an active research area for decades.

i.e., the processors are allocated equal sized data fragments, in which case the speed-up of query execution can be made to be approximately linearly proportional to the number of processors.

2.1.3 Forms of Parallelism

The parallelisation of a single query processing can be conducted at intra-operation level, inter-operation level or a way that combines both. The *intra-operation parallelism* attempts to, one at a time, distribute the load of each single relational operation involved in the query onto all processors and thus perform the operation in parallel; by comparison, the *inter-operation parallelism* aims at parallelising the operations of the query and allocating them to different processors for execution [DeWi92]. In addition, an important issue is how to schedule the multiple queries in processor allocation with each of the queries consisting of multiple relational operations. The result of exploiting parallelism in this upper level is *inter-query parallelism.*

2.1.4 Parallel Database System Architecture

Memory, disk and processor are the three most important resources in parallel database systems and thus the way in which they interact critically affects parallel processing. Up to now, the architecture can be classified into four categories, shared nothing, shared everything, shared disk and the hybrid approach.

Figure 2.1 shows three common architectures of parallel systems. In *a shared nothing system* (*SN*), none of the three resources is shared among the processing elements. Each processor has its own dedicated disk and private memory, and processors communicate through an interconnection network. Examples of this architecture are Volcano [Grae90], Gamma [DeWi90], Bubba [Bora90], EDS ESPRIT project: EDBS, TANDEM: NONSTOP SQL (Englert, 1989), Teradata DBC/1012 (Teradata 83, 85) [Page92]. In *a shared everything system* (*SE*), all disks and memory are public properties and shared by all candidate processors. The stored data are equally accessible from all tightly-coupled processors. Examples of this architecture are DB2 or ORACLE ported on classical multiprocessor mainframes (e.g. IBM), and ORACLE, INGRES or SYBASE on UNIX based multiprocessor (e.g. ENCORE). In *a shared disk system* (*SD*), each processor can

directly access any disks, but each processor has its own private memory (data sharing). Examples of this architecture are ORACLE on NCUBE, ORACLE on VAX CLUSTER, and IMS/VS data sharing product.

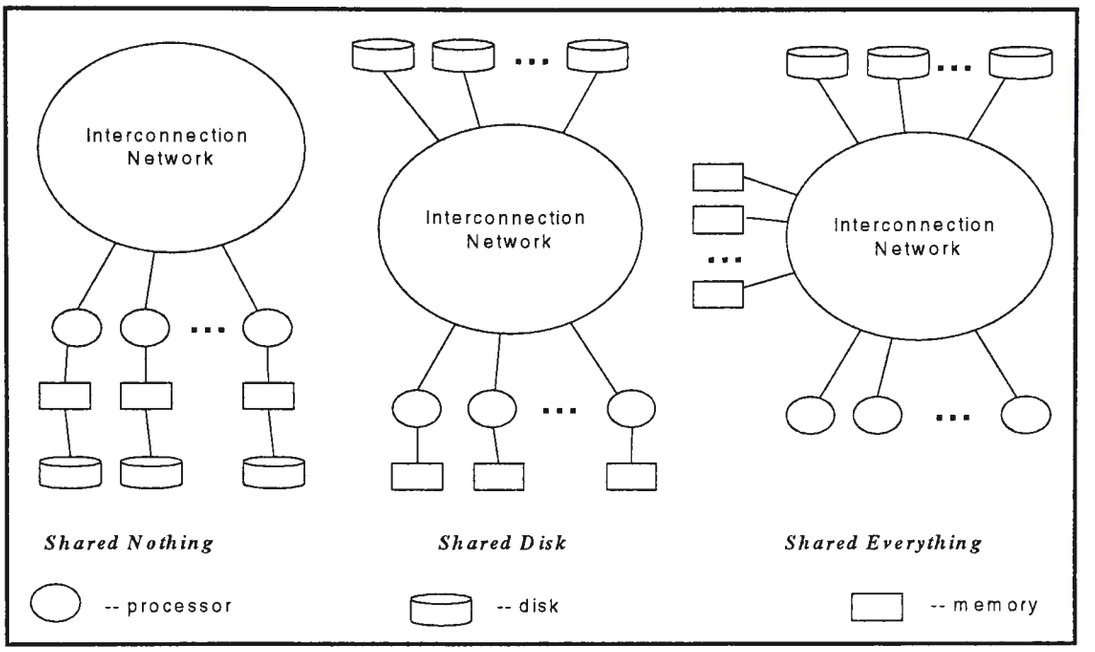


Figure 2.1: Shared Nothing, Shared Disk, and Shared Everything Architectures

A hybrid architecture can be a mixture of the above mentioned architecture types. One of the Hybrid architectures used in the Super Database Computer (*SDC*) under development at University of Tokyo is hybrid of tightly coupled multiprocessor (*SE*) and the message-passing architecture (*SN*) [Kits90, Kits92]. The *SDC* consists of several processing modules connected to each other through the message-passing interconnection network. In order to realise on the fly processing of the data stream from disk, each processing module itself is designed as a tightly coupled multiprocessor system which has considerable computational power. Another hybrid architecture is to have several *SE* systems as nodes of a *SD* system. The disks may be shared logically and distributed physically or they may be physically shared [Berg93, Rahm93, Hua90].

To summarise the fundamental features of the above architectures in parallel database systems, *SE* provides the low communication overheads and less load imbalance effect, *SD* supports good load balancing, *SN* takes the advantage of cheap hardware and provides the excellent scalability. *SD* is a compromise between *SN* and *SE*, and the hybrid architecture is another compromise among *SN*, *SE*, and *SD*.

2.1.5 Parallel vs Distributed DBMS

A database management system (DBMS) is a software package developed to access, manipulate, and manage the data in a database efficiently. A distributed database is a collection of multiple, logically interrelated databases distributed over networks [Ozsu91]. A parallel computer is a distributed system consisting of a number of nodes connected by a high speed network within a cabinet and each node comprises several components such as processor, memory and disk. The difference between a loosely coupled parallel system and a distributed system is conceptually small, but there are still a number of identifying characteristics.

The communication time in two systems are different. In distributed systems, it may involve a wide area network so that the time of data transmission becomes a dominant factor of the total time. In comparison, parallel systems mainly use local area networks where the ratio of data communication time to processing time is low.

There are only a small number of processors in a distributed system and the database is fragmented or replicated over the sites. The heterogeneity of both the software and hardware are obvious among the sites, e.g. one site may use ORACLE on UNIX and the other site uses DB2 on VMS. In contrast, parallel system mainly consists of homogeneous processors and there is one operating system.

There is always a host coordinating the parallel processing such as starting and terminating processes in parallel systems. Therefore, the query always comes to the host first and then data is partitioned and sent to different processors for processing. Results will be consolidated at the host before they are presented to the user. In contrast, every site in distributed systems is a stand alone system and hence the system reliability, availability and transparency are ensured.

The difference between the two systems also lies in the software support and transparency. Most of the parallel systems have their own parallel programming languages, so that they can provide better processor utilisation and a complete network transparency. Transparency is one of the utmost important issues in distributed systems since one of the fundamental requirements of distributed systems is that the distribution of data across sites

is not visible to the external users. There are several forms of transparency in distributed systems such as network transparency, fragmentation transparency, replication transparency and data transparency.

The current trend is that both parallel and distributed DBMS have started to become the dominant database technology in data intensive applications. These two systems combine and integrate together so that the parallel processing technique can be easily implemented in any distributed systems to improve performance. Taking an example of a distributed system of workstations[3], where application programs are running on the workstations and database systems are managed by the dedicated computers, i.e. the former introduces the concept of application servers and the latter results in database servers. When the query comes in, a certain number of processors in the system are activated to process it in parallel.

2.2 Pyramid of Parallel Relational Query Processing

Relational database processing on parallel systems can be carried out in several steps, namely, *data placement*, *data loading*, *data processing*, and *data consolidation* (see Figure 2.2). At the bottom of the pyramid is the data placement where database is fragmented and stored at each site possibly with replication. Both logical and physical design issues are considered so that the index files are established, local and global directory information are written for each site, and relations are fragmented and replicated over sites. This step may be treated as off-line operation.

When the query comes in, the processing starts with the data loading. At this step, query is examined and the relevant data are extracted through the directory files. The relevant relations may be loaded into memory at each site if they are on secondary storage and thus the main cost is due to data transmission. Data processing then takes place. Data is partitioned over multiple processors and local processing starts with the relation fragments. Here, the cost consists of local processing and data transmission among processors. Finally, each worker processor will send its processing result to the host where the data will be consolidated before presenting to the external user. The cost involves a simple unifying operation and data transmission.

---

[3] This is one of the most popular network connections nowadays and the client-server technology may be used in such an environment.

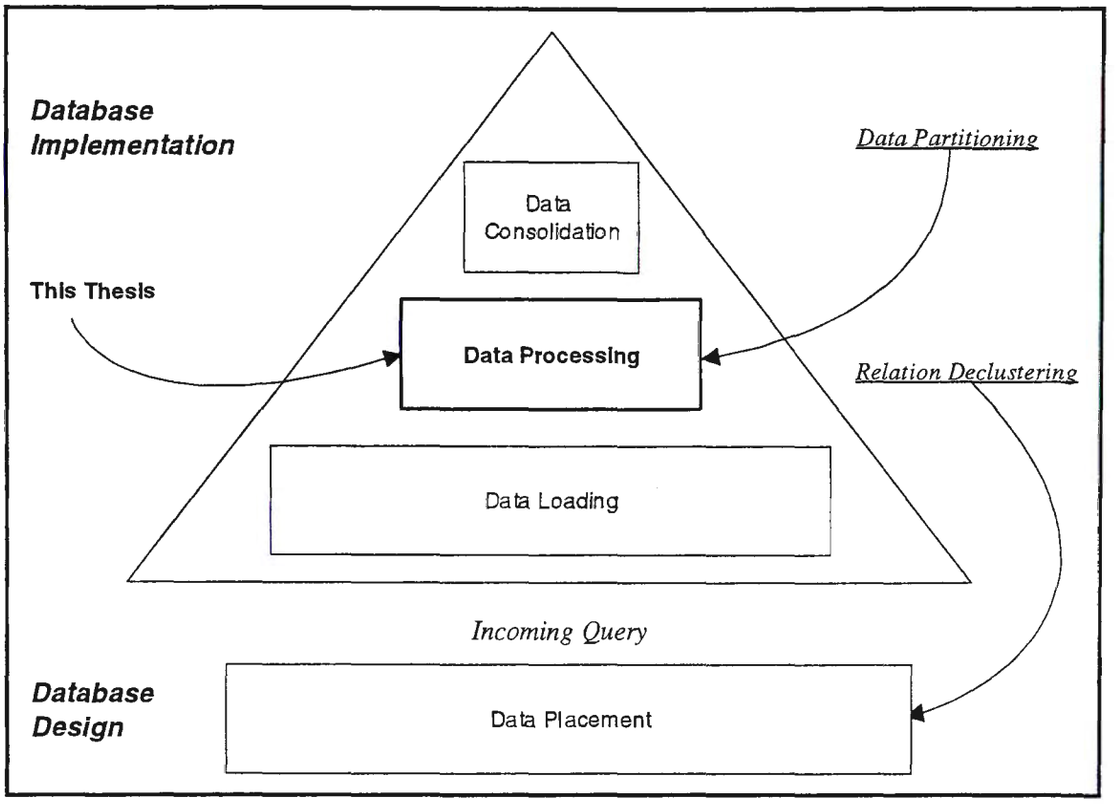


Figure 2.2: Pyramid of Parallel Relational Query Processing

A confusion may arise between data partitioning and relation declustering. The aim of data partitioning is to distribute tuples to multiple processors as evenly as possible so that linear performance speed-up can be achieved. It is an on-line operation and happens after receiving the query. The objective of relation declustering is to store the entire database over different sites and thus to increase the resource limitation at the host and to upgrade the performance by providing higher availability. This thesis only considers *database implementation* and we shall concentrate on the *data processing* of the pyramid.

2.3 Data Partitioning and Data Skew

We review the related work on data partitioning in this Section and data processing in next Section. In addition, we survey the recent parallel performance modelling work in Section 2.5.

2.3.1 Data Partitioning Methods

There are three common data partitioning methods, namely Round-Robin, Range Partitioning, and Hashing in parallel relational database and a detailed comparison of them is provided in Chapter 4. Round-Robin partitions the tuples of one relation in a round robin fashion; Range Partitioning distributes tuples to processors according to ranges (boundary values) specified by users; Hashing transfers tuples to various processors by employing a hash function that determines the tuple's destination [DeWi92a]. A mixture of the three common methods of partitioning, Hybrid-Range partitioning, is proposed in [Ghan90]. The algorithm sorts the relation, and divides the relation into fragments based on the processing capability of the system and the resource requirements of the queries that access the relation instead of the number of processors. The fragments are distributed in a round-robin fashion among the processors. The Hybrid-Range method is implemented on the Gamma database machine [Ghan90].

A multiple dimensional partitioning strategy (MAGIC) is proposed in [Ghan94]. The strategy declusters the relation according to several attributes to localise the execution of a greater variety of queries with selection operation. The Grid file is employed and both range and exact match selection can be dealt with. Another multiple dimensional strategy is developed by Copeland [Cope88] where relations are partitioned equally based on balanced heat (the access frequency of an object over some period of time) rather than size (the number of bytes in the object).

2.3.2 Data Skew and Load Balancing

a. Data Skew

Traditionally, parallelisation of relational operations has been investigated using the uniformity assumption [Chri83] which is originally employed to model the distribution of values of a single attribute in its domain. Based on the assumption, the occurrence of each attribute follows the uniform probability distribution. However, in most circumstances, it is highly likely that some attribute values appear more often than others, and the non-uniformity of the attribute values is the primary cause of data skew [Laks90]. In [Lync88] a series of selectivity estimation methods for coping with highly skewed data such as Zipf

distributions found in the textual or bibliographic databases are proposed. A new selectivity estimator is introduced and the costs of incorrect estimation are quantified.

In [Walt91], some issues of the problem of data skew are classified, and the data skew effects are identified. Furthermore, it discovered four kinds of *partition skew*, tuple placement skew, selectivity skew, redistribution skew, and join product skew. Tuple placement skew is caused by the initial distribution of tuples since the *attribute value skew* may result in different numbers of tuples over processors. Selectivity skew happens when the selectivity factors among processors after partitioning are different, and redistribution skew takes place whenever redistribution is involved to prepare the actual join and a different number of tuples is received by each processor. Finally, the join selectivity on each processor may differ, leading to various number of output tuples which is referred as join product skew. The experimental result shows that scheduling hash join algorithm effectively handles redistribution skew and increasing redistribution skew degrades the performance of hybrid hash join algorithm significantly. A more formal and general approach can be found in [Liu95] where a complete skew taxonomy for intra-operation parallelism is introduced and parallel database performance issues are identified. The detailed taxonomy and analysis will be discussed in Chapter 3.

b. Load Balancing

With multiple processors and multiple relational operations constrained by the data flow, ideally, all processors finish at almost the same time with a minimal execution time and this shall be referred as the load balancing. To some extent, load balancing in parallel database is always a critical issue and it is affected by both the application and the architecture. The load balancing can be easily achieved and efficiently handled by the run-time system with a *SE* architecture, and the cache memories may be introduced to provide fast access to the most frequently used data and code. In contrast, *SD* architecture can also provide good load balancing because the bottleneck nodes are easily removed by data replication. However, load balancing is complex and hard to achieve for *SN* architecture since the nodes are loosely coupled and the communication overheads dominate the main cost. [Hua95, Yu92, Lu90, Yu86]

2.4 Data Processing and Skew Handling

2.4.1 Skew Handling

The study of the skewness may be classified into empirical skew handling and theoretical skew prediction as shown in Figure 2.3. The former has attracted the attention of many researchers, and a number of skew handling algorithms have been proposed while the latter has been neglected. We shall introduce skew prediction in Chapters 4 and 5, and here we shall concentrate on skew handling which can further be classified into skew estimation and skew management. Skew estimation attempts to identify the existence of the skew among parallel processors and, if exist, estimate the extent of the skew. In other words, it is the process on collecting statistics, e.g. from data dictionary. The available information may be none, partial or complete. Based on this knowledge, the skew management then attempts to avoid or resolve skewed load distribution with two approaches, static and dynamic methods. The former aims at declustering the relation table evenly over parallel processors before operation processing begins, and once one fragment is allocated to one processor no migration is performed, so that the fragment remains there until the execution completes. The latter either allocates workload dynamically during operation execution or reallocates the workload from the busy processor to others with inter-processor communication [DeWi90, DeWi92b, Grae93, Hua91, Kits90, Lu92, Omie91, Schn89, Wolf93a, Wolf93b].

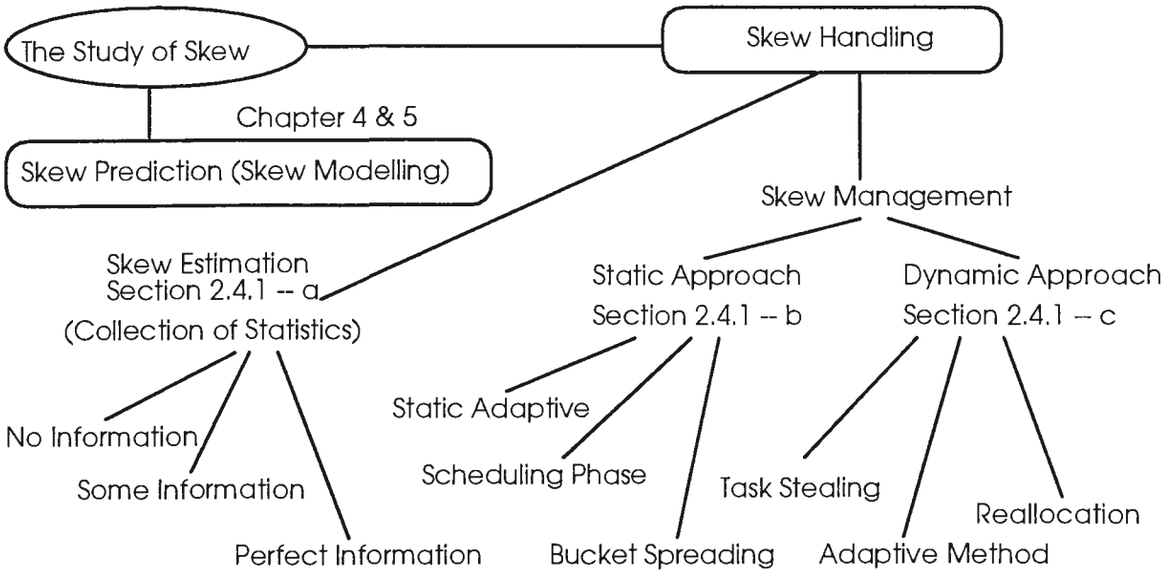


Figure 2.3: The Study of Skew

a. Skew Estimation

Within skew handling, most works have been done concerning skew management and only a relatively small number of works are related to skew estimation [Chri83, Seli79, Sesh92, Sun93]. Statistical methods and probability theory play an important role in skew estimation. Statistical methods can be classified into parametric and nonparametric methods depending on how much is known about the shape of the distribution. If there are only a few unknown parameters with a known basic distribution, the methods involved are referred as parametric methods. On the other hand, in many cases an experimenter does not know the form of the basic distribution and needs statistical techniques which are applicable regardless of the form of the density [Mura88]. These methods are known as nonparametric methods. The collected information may be stored in the data dictionary, but the critical issues are how much information should be stored, how often information should be updated, and how information of data dictionary is placed in the system, i.e. replication and fragmentation.

An important statistical method is sampling which has been widely used in query size estimation [Sesh92, Lipt90, Mann88]. In a parallel database system, it is not practical to conduct measurement based on the entire input population (all tuples, relations, and queries). Therefore, we can employ sampling by which some on-line and useful information can be collected. Moreover, sampling has the advantage that it is more accurate than pure probability and parametric methods which are entirely based on their assumptions. However, sampling introduces the following issues.

- Cost; generally, the cost of the sampling is high.

- Sample size; how many samples should we take for a certain skew level and confidence level. It goes without saying that there will be always a trade-off between sample size and cost.

- The way of taking samples; random sampling means every element in the population has the same probability being selected. Therefore, all elements (tuples) must be put in one storage (disk or memory) before sampling process. One commonly used alternative in multiprocessor systems is dividing the whole population into n subpopulations where n is the number of processors.

Then, sampling is carried out in each sub-population for each processor. The trade-off is losing precision.

- Skew in sampling; again, load balance has to be dealt with because of the skew in sampling. Ideally, the sampling workload is partitioned equally over the processors. However, this is hard to achieve because of the randomness of the sampling. [Sesh92] provides an analytical model which specifies the probability of the maximum of the number of sampling units chosen at each site. According to their predictions, having 1000 sites and taking 10000 samples, the maximum and minimum number of samples that would be chosen by the sites are 22 and 1 respectively (Page 332, [Sesh92]). In other words, one processor will be loaded as 22 times heavily as other processors.

b. Static Skew Management Approach

The characteristic of this approach is that there exist no inter processor communication and no process migration during execution.

Static Adaptive Method

An adaptive skew handling method is developed by [DeWi92b] and the idea behind the method is to employ multiple algorithms to deal with different degrees of skew. Selecting algorithms in the method fully depends on the result of the sampling process carried out at the beginning of the processing. To make the sampling feasible, sufficient, and not too expensive, stratified sampling and page-level extent map sampling are used. Hence, sampling process employed in the method not only intends to choose the appropriate algorithm among the five alternatives but also determines the splitting vectors for the individual algorithm except hybrid hash join. The five algorithms implemented in the method are *Hybrid Hash* (*HH*), *Simple Range Partitioning* (*SRP*), *Weighted Range Partitioning* (*WRP*), *Virtual Processor Partitioning -- Round Robin* (*VPPRR*), and *Virtual Processor Partitioning -- Processor Scheduling* (*VPPPS*). Among them, *HH* copes with no skew or lower skew cases; *SRP* and *WRP* deal with redistribution skew; *VPPRR* and *VPPPS* are employed to avoid join product skew. Both single skew and join product skew experiments have been carried out, and the method has been implemented on the Gamma Parallel database machine. The test data (relations) are generated with a number of integer attributes, each with various amounts of "Scalar Skew". However, sampling process has its

drawbacks as we discussed in Skew Estimation section. Furthermore, multiple algorithms in one approach may introduce more complexity both to the systems operating and to the systems maintaining. On top of this, how to handle joins in which the operand relations are of greatly different size is a difficult task.

Scheduling Phase

To avoid the skewness, an extra scheduling phase may be introduced in the conventional parallel algorithms. A new parallel sort-merge join algorithm is developed in [Wolf93a] and the scheduling phase is inserted after the conventional sort phase. The purpose of the scheduling phase is to identify the largest skewed fragment and allocate more processors. The new algorithm includes two classical optimisation algorithms. The *Longest Processing Time first* (*LPT*) algorithm is always employed first to distribute the tasks to processors whereas *Galia and Mediddo* (*GM*) algorithm is involved only if the result of *LPT* is unsatisfactory. An analytical model of the algorithm is also presented, and it illustrates that the algorithm provides good load balancing in the high skew case.

In [Wolf93b], a parallel hash join algorithm for managing data skew is developed which introduces the scheduling phase after the hash phase to balance the load for the join phase. The scheduling phase comprises two steps. First, a minimal number of additional tasks are created; then *LPT* algorithm is employed iteratively. The main activities in each iteration are splitting out the largest type 2 pair and running *LPT* to evaluate the result. Experimentation is carried out together with several existing parallel hash join methods and the results show that the algorithm works well with highly skewed data. Another contribution of the work is introducing the idea of hierarchical hashing.

Bucket Spreading

[Kits90] proposes a bucket spreading parallel hash join algorithm which can be treated as an alternative hash join method of the earlier bucket converging strategy together with introducing the problem of data skew. In bucket converging strategy, a relation is partitioned into a large number of buckets greater than the number of processors. After partitioning the operand relations, redistribution takes place to ensure the load balance among processors. The initial repartitioning is highly likely to result in partition skew, and as an alternative a bucket spreading algorithm is developed. The new algorithm differs in the initial redistribution where each bucket is horizontally partitioned over processors. A

sophisticated network, Omega network, is used to redistribute buckets to processors. The algorithm is demonstrated with parallel GRACE hash join method and implemented on the Super Database Computer with hybrid parallel architecture, high-functional interconnection network (Omega), high speed hardware sorter, and the separation of data and control passes. A simulation model is also set up to evaluate the performance of the strategy and the results show the improved performance in the presence of data skew (modelled by the Zipf distribution) compared to the bucket converging strategy.

To avoid the use of special hardware, [Omie91] designed a load balancing hash join algorithm and is implemented on a shared everything multiprocessor system. The algorithm is based on the bucket spreading method and an extra first-fit decreasing heuristic is employed to allocate buckets to processors in order to minimise the number of redistribution steps. An analytical model of the cost of the algorithm is developed and experimental results show the improved performance when compared with the basic parallel join methods in the presence of data skew.

In [Hua91], three skew management algorithms for parallel hash join, tuple interleaving parallel hash join, adaptive load balancing parallel hash join and extended adaptive load balancing parallel hash join are presented. The first algorithm is a variation of the bucket spreading algorithm with software control at each *processing node* (*PN*) instead of special hardware. The only difference is in the first phase -- Split phase. During the split phase, relations are partitioned to processors and then grouped into buckets at each processor. Next, buckets are spread over processors. The algorithm introduces unnecessary communication overhead and computation overhead when the skew condition is mild due to their load balancing process. To avoid the massive data redistribution especially in the case of mild skew, the second algorithm is developed with a more selective redistribution. A bucket partition tuning phase is inserted after the initial split phase with two stages: bucket retaining stage which employs best-fit strategy to retain buckets for each processor, and bucket relocating stage which has a coordinator to gather information of each processor size and size of the excess buckets. Then, best-fit decreasing strategy is employed again to allocate excess buckets to the under-utilised processors. The third algorithm is designed for the case of high skew by deferring the tuple transfer until the partition tuning phase, and thus may be treated as an extension of the second algorithm. In addition, it avoids disk overflow by storing each sub-bucket in the local disks and has less problem on network traffic. A performance model is developed with cost function and sensitivity analysis is

carried out varying the degree of skew, communication bandwidth, and I/O bandwidth.

c. Dynamic Skew Management Approach

Unlike static skew handling, dynamic skew handling attempts to either allocate workload dynamically during operation execution or reallocate the workload from the busy processor to others.

Task Stealing

[Lu92] proposes a dynamic load balancing algorithm to minimise the response time based on task-oriented database query on a multiprocessor system with shared-disk architecture. There are three phases in the algorithm, task generation, task acquisition and execution, and task stealing. In the first phase, a set of tasks are generated from the original query. The optimal number of tasks generated is related to the memory size of each processor and the data size that an operation works on. A directory with a sub-bucket is created to store the disk identifier and page identifier. In the second phase, a free processor acquires a task if it is available, and then has the address of all pages in the task through linking directories. The task is executed, and when each page is read the global information is updated accordingly. This process continues until all tasks are allocated. In the third phase, an idle processor chooses a donor (one of any running processors) and determines the amount of data to be transferred in such a way that donor and receiver get equal workload (half load for each processor). This process continues until the minimum completion time is achieved. A brief simulation is carried out but more work is required for the general queries. The performance gain is in a wide range of 5% to 90% depending on the data skew comparing with the basic parallel hash-based join methods.

The issues of parallel join on *shared virtual memory* (*SVM*) are studied in [Shat93] and two variants of an algorithm for parallel join are proposed with *SVM*. The idea behind the algorithm is load sharing which is the same as the task stealing of [Lu92]. A detailed simulation study is presented and the result shows that using *SVM* can improve the join performance in the presence of data skew in a *SN* system. When the data skew is absent, the performance of the algorithm is identically to that of parallel hybrid hash join algorithm.

Adaptive Method with Threshold Function

In [Kell91], an adaptive method is developed by modifying the conventional hash phase of the parallel hash join and introducing skew detection and workload redistribution. Skew is detected at run time by observing the frequencies of the join attribute values and setting a threshold function. If the skew reaches a certain limit set in the threshold function, the load in that processor will be repartitioned among the available processors. To facilitate the algorithm, local cache is provided to store the global profile of the load distribution across all processors. If the large workload of one processor has been detected by the threshold function, then all other coming traffic will be directed away. To fragment and redistribute the highly imbalanced load, three methods are adopted, *Symmetric Fragment and Replicate* to split the skewed workload, *Decentralised and Approximate Fragmentation Scheme* to determine the number of fragments and minimise the communication overhead, and *Adaptive Resizing of Fragments* to fine-tune the load distribution by dynamically resizing. The algorithm is implemented in message passing environment with single user mode and relations are partitioned horizontally. Both partitions of operand in each processor are assumed in the main memory, and no indexes are used in join. A simplified analysis has been carried out and the results show some improvement over ten or fewer processors. Increasing the number of processors, i.e. more than ten, the response time is reduced and almost linear speed-up is obtained in the single skew case. However, it is hard to measure the speed-up for the case of double skew, which requires partial replication of the input tuples.

Reallocation

[Dewa94] develops a dynamic load balancing algorithm to balance the computations of parallel hash join over heterogeneous processors in the presence of data skew and external loads. The algorithm has two stages, initial and batch processing stages. In the initial stage, the number of buckets and the maximum batch size are decided with the exchanging of information between processing sites and coordinator processor. In the batch processing stage, the batches of buckets are processed until the join is fully completed. A rescheduling phase is inserted in the batch processing stage when one processing (the fastest) site finishes its work. The objective of the rescheduling phase is to reallocate unprocessed buckets from the current batch at each site over all the sites so as to minimise the overall execution time for the current batch by using the *Weighted Longest Processing Time First* algorithm. As a result, the buckets are shuffled among the processing sites and execution continues. This

process lasts until the operation is fully completed. Cost models and predictive dynamic load balancing protocols to detect imbalance are also developed. The algorithm is implemented on a prototype of the system under various load distributions and degrees of data skew, and the results show that the algorithm presents better performance when the processing sites are heterogeneous.

d. Summary

The static skew management method approaches the problem by detecting input data distribution before executing or implementing the algorithms. This approach is more accurate and feasible in conducting simple operations such as binary join. The method will become complex with an increasing number of processors and multiple queries requiring multiple operations in each query. Generally, the static skew management method will be employed for every input original relation and every intermediate result relation. Surely, most of the existing static partitioning methods are unable to estimate precisely the intermediate coefficients growth since it relies on a large number of parameters. Taking into account only a small number of these parameters to characterise the inputs, leads to too large and often unrealistic upper bounds on the results.

The dynamic skew handling approaches the problem at run time by waiting for the skew occurrence and eliminating the skew. It does not need preprocessing and relies on no distribution assumptions. However, the performance gain of this approach is highly variable. In other words, it is not ensured that the method will improve the system throughput to a satisfactory level. Therefore, a few systems will take the risk of employing unreliable performance enhancing algorithms by adding in more complexity and cost into the existing system.

All methods have their limitations, assumptions, and usually targeted at specific situations. To sum up, most of the existing skew management algorithms cannot deal with situations in which there is no intrinsic data skew, and in general no consideration is given to determine the stochastic nature of skew behaviour which is the main focus of this thesis. It is felt that skew estimation should be emphasised along with skew management, so that the workload of the system as a whole can be properly balanced and the system performance can be effectively optimised.

2.4.2 Parallel Join

[Schn89] analyses and compares four parallel join algorithms, simple hash join algorithm (looping algorithm), sort-merge join algorithm, GRACE hash join and hybrid hash join algorithm. The issues discussed include the tuple distribution policies, the existence of bit vector filters, the sizes of main-memory for joining, and skewness. The experimental relations are extracted from the standard Wisconsin Benchmark running on the Gamma database machine. The results show that hybrid hash join is superior in particular in no skew case at all degrees of memory availability. In addition, it is found that the hash-based join algorithms are sensitive to redistribution skew in the building relation but relatively insensitive to redistribution skew in the probing relation. However, the double skew case was not considered in the paper.

The comparisons in [Schn89] did not take into account the join inputs, and a more general survey is provided in [Grae94]. The performance evaluation of sort-merge join and hybrid hash join is extended with the following issues, effectiveness and skew, graceful adaptation to memory re-allocation, read-ahead and write-behind, disk arrays, and disk caches. Moreover, the experimentation is focused on very large inputs (inputs are larger than the memory size multiplexed with a relatively small fan-in or fan-out). The results show that hybrid hash join outperforms sort-merge join in most of the cases with only two exceptions.

- the joining (merge) attributes from multiple indexes are already in the order of data values or in the order of hash values.

- the output of the query must be sorted on a join key and the cost of sorting output is more than the cost of sorting two join inputs.

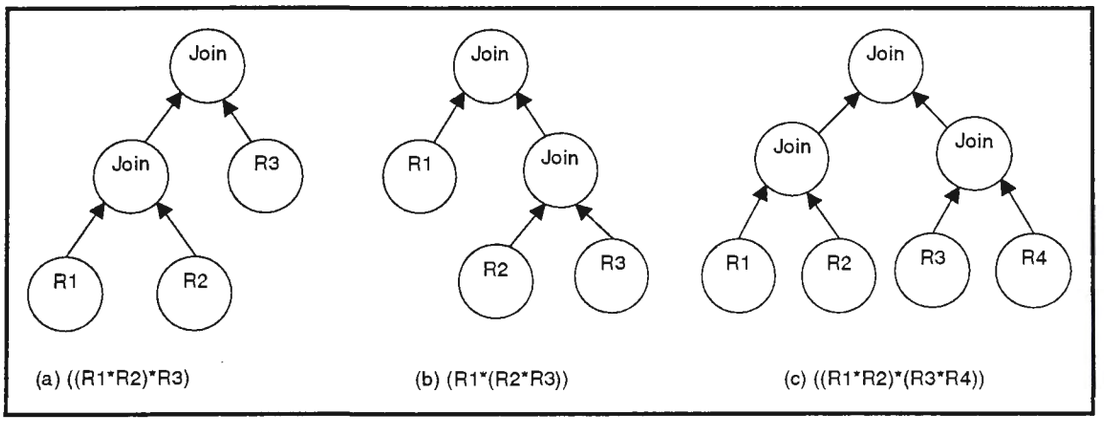
2.4.3 Parallel Query Processing

Parallelism can be classified into partitioned and pipelined parallelism. Partitioned parallelism divides the workload into fragments, and executes each fragment on one or a group of processors or several fragments on one processor according to the number of available processors. In other words, it is horizontal parallelism that aims at improving performance by increasing the degree of concurrent worker processors. As such, the

processors are most likely dealing with the same operation but different ranges of data. In contrast, the pipelined parallelism treats the operations processing into several steps and the output of the first step is fed to the input of the second step. Hence, it is vertical parallelism that speeds up operations by introducing the concept of functional processors. As a result, different processors are dealing with different operations but the same range of data. For example, with a sort-merge join operation, there are scanning processors, sorting processors, and merging processor by using pipelined parallelism. Partitioned parallelism is more versatile than pipelined parallelism as it may be used for any operator, but the pipelined parallelism can only be used between two operators connected by a pipeline edge. In addition, skewness is also a more interesting problem in partitioned parallelism since this kind of parallelism has more control of the data and it also has a greater impact on performance. As such, hereafter in the thesis, parallelism means partitioned parallelism.

a. Query Tree Representation

The execution of a single query can be denoted by a query execution tree and three different formats are used to construct the tree of operators as shown in Figure 2.4. Each tree node represents one relational operation and the tree shows the procedural choices such as the order in which the operators are evaluated. The left-deep tree and right-deep tree are also referred as the linear tree. In [Sche90], it is observed that the linear tree offers two extreme options of restricted-formatted query trees, and the bushy tree has no restrictions placed on their construction. However, it is harder to synchronise the activity of join operators within an arbitrarily complex bushy tree.



(a) Left-Deep Tree (b) Right-Deep Tree (c) Bushy Tree

Figure 2.4: Query Tree Representation

b. Query Processing

Query processing may be carried out in two phases as in Figure 2.5. Phase one is query decomposition where a high level language such as SQL is translated into low level language such as relational algebra. The second phase is query optimisation where alternative execution plans are generated and the optimal or near optimal plan is selected based on the cost model. In multiprocessor systems, the formulation of a parallel query plan deals with not only the execution sequence and the processing methods of the operations required in the plan, but also processor allocation which enables parallel processing of the operations so as to minimise query response time [Hong92, Gang92, Chek95]. Parallel multiple query processing may be performed at three levels, *intra-operation*, *inter-operation*, and *inter-query* [DeWi92a, Vald93, Moha94].

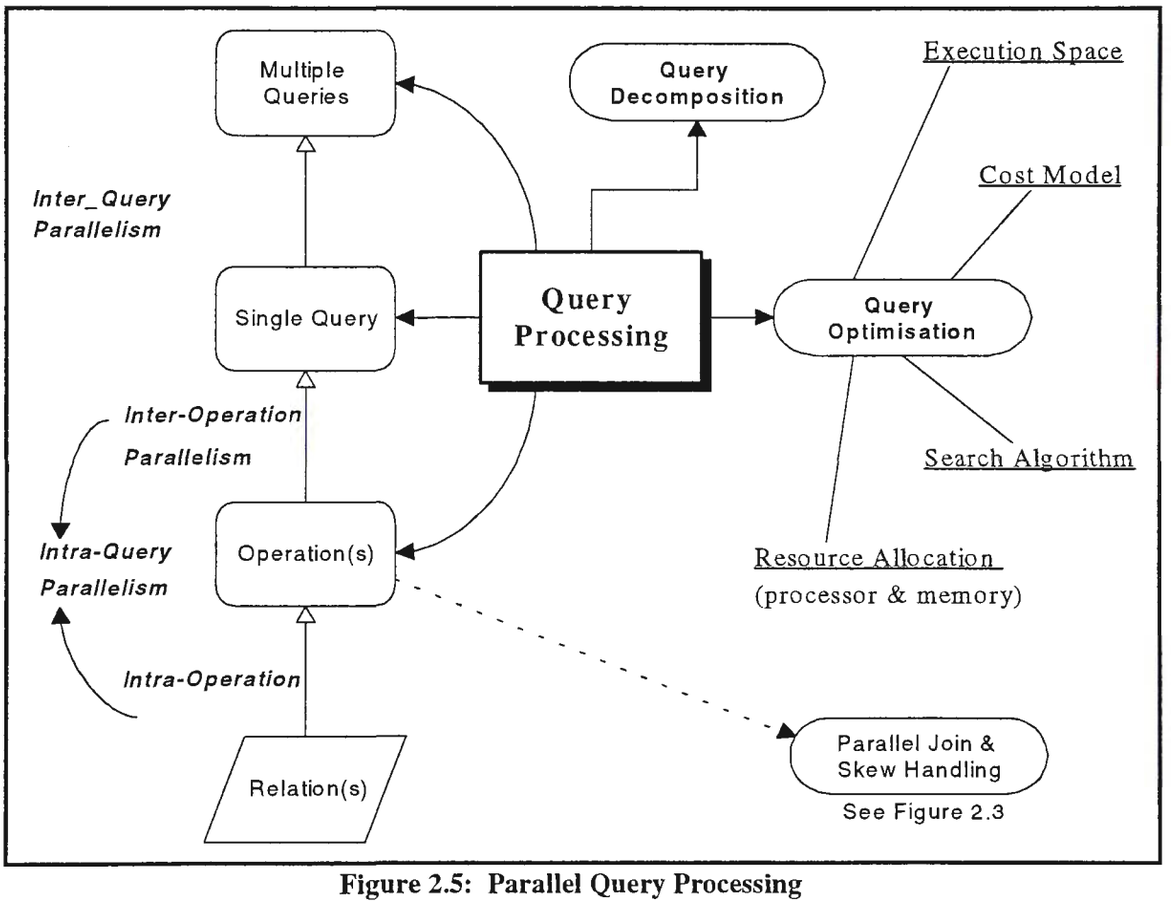


Figure 2.5: Parallel Query Processing

c. Query Optimisation

Figure 2.5 shows that query optimisation is concerned with issues on execution space, cost model, search algorithm, and resource allocation. Execution space is the entire search scope; cost model is the criteria to evaluate the alternative algorithms; search algorithm is the algorithm used to look for solution by searching the execution space; resource allocation

provides policies on sharing and distributing resources among relational operations.

Intra-query Level

[Gang92] offers a framework on query optimisation for parallel execution and addresses the problems in designing execution space, cost model, and search algorithm. They discovered that the response time metric violates a fundamental assumption in dynamic programming and the optimisation has to extend to multiple dimensions. [Sriv93] claims that a good query optimiser must take the design of a resource schedular into account in creating an apparent circular dependency in a *SN* system, and the circular dependency may be removed by an exhaustive algorithm. The issues of using the inherent parallelism in a hypercube multiprocessor to optimise large join queries are studied in [Lin94]. They propose an algorithm to evaluate a large parallel join plan and present three heuristics for generating an initial solution followed by the iterative local improvement.

[Hong93] proposes a two-phase optimisation strategy to reduce the search space and is capable of coping with run-time parameters such as available buffer size and number of free processors. During the first phase, the sequential query execution plan is examined and optimised with fixed parameters at compile time. In the second phase, the optimal sequential plan from the first phase is optimised again with multiple processors and updated parameters at run time. [Hua95] introduces the cost of load balancing as a new factor for query optimisation, and based on this factor three new optimisers are implemented on a simulation model validated by a multiprocessor system. The paper also observes that the *Load Balancing Optimisation with the Symmetric Fragment and Replicate Feature scheme* is immune from bucket skews and its performance follows a flat curve for all skewed conditions. Moreover, the new algorithms consistently provide good performance despite varying system parameters.

The inter-operator parallelism is studied in [Chen92] and the focus of the paper consists of scheduling the execution sequence of multiple joins within a query and determining the number of processors to be allocated for each join operation. One join-sequence scheduling heuristic and several processor allocation heuristics are proposed and evaluated by simulation. It is found that the best scheme is a two-phase approach which applies the join-sequence heuristic to build a bushy tree first as if under a single processor system, and then allocates processors to the internal nodes of the bushy tree in a top down manner. Multi-join

query optimisation for symmetric processors is investigated in [Shek93] and optimisation algorithms are developed by using dynamic programming and greedy heuristics. In addition, its cost model takes into account the memory resources and pipelining, and is used to compare the performance of the algorithms.

Inter-query Level

[Wolf95] devises and evaluates a number of heuristic scheduling algorithms on multiple queries with data dependency only existing within each query. One of the algorithms, the hierarchical algorithm, consists of two components, intra-query precedence-based scheduling and inter-query non-precedence-based scheduling. The former advocates that when a task is assigned to a number of processors, each of its children will be allocated to subsets of those processors, and is also referred to as tree-split algorithm. The latter will give each task a number of processors such that the total work time is minimised. By employing a nonmalleable scheduling algorithm, the number of non-utilised processors is found. The task with the largest wasted work is regarded as a bottleneck. The number of processors will be reconsidered for every operation in the task and the objective is to drive the bottleneck to zero.

Dynamic site allocation algorithms and query scheduling policies are studied in [Frie94] in a distributed memory multiprocessor system with a hypercube architecture. The site allocation does not employ a prior optimisation but select sites during the execution. When multiple queries are running concurrently, the selection of queries is made based on their resource requirements.

Parallel Task Scheduling

Parallel task system is studied in [Du89] where a task is running on one or a group of processors and there is no workload or processor migration once the number of processors assigned to a task is determined and fixed. Both the complexity of nonpreemptive and preemptive scheduling of parallel tasks are examined. Finding an optimal nonpreemptive schedule for parallel tasks with the precedence constraints consisting of chains, is strongly NP-hard when the number of processors is equal to or greater than two; for preemptive scheduling, the problem is also NP-hard for arbitrary number of processors for a set of independent tasks.

[Ture94] considers nonpreemptive scheduling of a system of independent tasks. The paper tackles the problem of finding a schedule with minimum average response time by designing a polynomial time algorithm whose worst-case performance is within a fixed multiplicative constant of the optimal. The algorithm is shelf-based where the tasks are assigned to shelves whose placement corresponds to constant time values, and all tasks assigned to a shelf have starting times equal to this value. The paper also claims that the algorithm can be extended to solve the comparable malleable minimum makespan problem in polynomial time with identical worst-case performance bounds.

d. Other Relevant Work

[McCa94] introduces the concept of *efficiency preservation* as a characteristic of processor allocation policies and argues that the measure should be taken relatively to a particular workload since all allocation policies are tied closely to the supporting workload. Two families of processor allocation policies, *Equipartition* and *Folding*, are proposed which emphasise efficiency preservation and equality of processor allocation. Folding gives better performance than that of Equipartition based on a static analysis and a simple Markovian birth-death model.

2.5 Parallel Performance Modelling

2.5.1 Parallel Computation Model

Analytical models are generally more versatile, as they can be applied in a wide range of systems. They are comparatively inexpensive and allow more effective sensitivity analysis and system tuning. A widely used analytical model on parallel systems is Amdahl's law whereby all operations are divided into either sequential or parallel processing with the running time of parallel operations improving by a factor equal to the number of processors [Amda67]. The Amdahl's law is revisited by [Gust88] and it is found that the speed-up should be measured by scaling the problem to the number of processors, not by fixing the problem size.

[Helm90] proposes a simple model of parallel computations which is capable of exploiting the speed-up greater than n on n processors. The model tries to unify the previous modelling

results by including a certain number of parameters describing the previous reports and illustrating different assumptions responsible for the apparently contradictory outcome. A software tool for measuring parallelism in large programs is presented in [Kuma90] and an important feature of this tool is to accept ordinary Fortran program as its input so that parallelism can be measured easily in large scientific applications.

2.5.2 Parallel Join and Load Balancing

In [Pate94] an analytical model is given to estimate the execution time of hybrid hash join when a single join is running on a single node. Physical database issues taken into account into the model are disk seek time, caching choices, intra-operator synchronisation and disk interference patterns, and the model is evaluated by a simulation study. In [Pete94] an analytical model for designing architectures and characterising applications is proposed. Load imbalance and the number of iterations are treated as two main factors, and the model is applied to discrete-event simulation on distributed-memory MIMD machines.

Theoretical resource allocation is studied in [Azar94] using probability theory. It is found that with n balls and n boxes, if we choose two boxes randomly (instead of one box) and put the ball into the one less full, the fullest box contains balls exponentially less than that of one box. The infinite version of the random allocation process is also studied, and an analysis is provided in a situation where a ball is chosen uniformly at random and replaced with a new ball, and the new ball is placed in the least full box among a number of possible destinations.

2.5.3 Skewed Distributions

Perhaps the most well-known non-uniform distribution is the Zipf distribution which gives a good approximation for the occurrence frequencies of words in natural text. Based on this distribution, the rth highest multiplicity m_r is given by $m_r \approx C / r^{\theta}$ where r stands for the rank [Zipf49]. In [Knut73], the Zipf distribution is employed to describe the results of hashing, and recently, it is used to describe the number of appearances of unique tuples [Wolf93a, Wolf93b].

The effect of load skew on the performance of parallel unary relational operations based on the SIMD paradigm operating in a homogeneous distributed memory configuration is examined in [Leun94a]. It is found that the horizontal partitioning of a given relation often leads to uneven distribution of workload among the participating processors. The paper shows that in most data partitioning schemes, the actual performance can depart significantly from the ideal loading condition where each processor receives the same number of data tuples for processing. It also found that as the number of processors increases, the load imbalance between the least utilised and the most utilised processors will tend to increase as $O(k \ln k)$.

In [Falo96], it is shown that the multi-fractal theory formalises and generalises the 80-20 law, and also includes the uniform case as a special case when p=0.5. The paper defines that a binomial multifractal distribution is a distribution of N records with parameters (N, p, k) if it has 2^k possible attribute values, each attracting records with the bias parameter p. Using this assumption, the paper presents a simple way to estimate the multiplicity vector which can help to extrapolate for several useful statistical quantities such as supersets of a relation. However, it does not examine the application of multifractals to other useful settings such as the join size estimation and join selectivity. In addition, the discovery does not reveal any relationship between the partitioning result and the partitioning function.

2.6 Summary

We have reviewed the works relevant to this thesis in this chapter. Parallel processing of relational databases has been shown to provide highly effective improvement on performance. The motivation for using parallelism, the features of relational databases, forms of parallelism, and parallel database systems architecture are presented. A comparison between distributed and parallel databases systems is provided. To clarify the mechanism of relational database processing, we provide a pyramid of parallel relational query processing where four main steps are identified, namely, data placement, data loading, data processing, and data consolidation. Three common data partitioning methods are listed and previous studies on data skew and load balancing are examined. A detailed survey on skew handling in parallel join is provided where a number of existing algorithms are classified into several groups and their main contributions are highlighted. The important issues on parallel query processing are examined and existing performance

models on parallel computation are surveyed.

From the literature survey in parallel relational query processing, we see that most of the existing work are concerned with practical algorithms aiming at processing relations in parallel effectively. However, a critical problem with load balancing in such an environment is the skewness where there is not a great deal amount of work have been done so far. Considering both the complexity and the importance of the skew problem, we believe that a systematic and in-depth theoretical treatment on it is needed urgently. This will also lay down a foundation by which the degree of skew can be predicted accurately and the efficiency of various algorithms may be evaluated. Taking the skew factor into the cost model of query optimisation, it is possible to produce a high-performance and cost-efficient parallel processing plan.

CHAPTER 3

SKEW TAXONOMY AND ANALYSIS

3.1 Introduction

A major obstacle to performance improvement in parallel database processing is the presence of skew, which in extreme cases can contribute to performance degradation to a level below that of uniprocessing. Skew handling algorithms for parallel join have been an active research area for some time, but most of the existing methods can not deal with the case of no data skew efficiently and in general involve considerable overheads. Moreover, the unsolved questions are what is a systematic definition of the entire problem of data skew, why there is the problem of data skew, and what are the effects of skewness on system performance. Related questions are what is the classification of the problem of data skew since it is a such complex and critical issue in parallel databases, and how to model the problem of data skew quantitatively. We expect that a formal skew foundation can be

established by answering the above questions on which the various skew handling algorithms can be evaluated and compared.

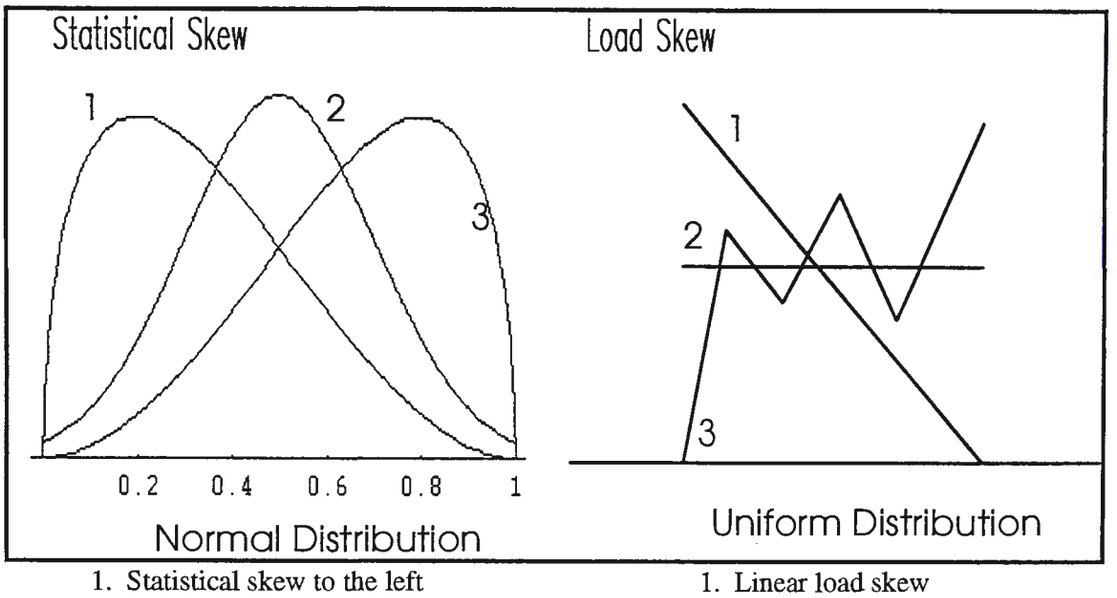
In this chapter, a new analysis of skew for parallel database system is presented with attention focused on the relational model operating in a shared-nothing distributed memory environment in SPMD mode. Three types of skew are identified: data skew, load skew, and operation skew. Load skew may further be distinguished into I/O, operation, and result load skew. These different types of skew are quantitatively analysed and expressions are provided for their evaluation. In particular, it is shown that skew effect can be significant even when the data partitioning is carried out uniformly across all processors. In addition, performance bounds on best and worst case behaviour are also presented.

The rest of this chapter is organised as follows. Different types of skewness are introduced in Section 3.2 and parallel database performance issues are discussed in Section 3.3. Finally, Section 3.4 presents a summary.

3.2 Conceptual Framework of Skew

Data skew is the phenomenon that certain values for a given attribute occur more frequently than other values and is the main reason of having imbalanced load over processors. Throughout this thesis, we shall refer to the entire load balancing problem as *the problem of data skew* and further identify three types of skewness within the problem of data skew in parallel relational database systems, *data skew, load skew,* and *operation skew.*

The word "skew" stems from statistics, but there is a difference between the definition of skew in statistics and load skew used by computer scientists as shown in Figure 3.1. The former can be defined as a data set with observations that are not symmetrically distributed, while the latter is the load imbalance over the processors. More precisely, the differences are that the former describes the departure from the symmetrical distribution and the latter describes the departure from uniformity. In terms of load balancing, equalised load distribution (i.e. uniform distribution) means no skew whereas no skew refers to symmetrical distribution (i.e. normal distribution) in statistics. There is no doubt that the latter is not sufficient to represent the former unless it is related to other measurement (e.g. measures of variation).



1. Statistical skew to the left
2. No statistical skew
3. Statistical skew to the right

1. Linear load skew
2. No load skew
3. Non-linear load skew

Figure 3.1: Statistical Skew vs Load Skew

3.3 Different Types of Skew

In general, several types of skew may be identified, these are *data skew, load skew,* and *operation skew*. Data skew is a property of the attributes without reference to the mode of processing involved. It is concerned with the clustering and replication of the underlying attribute values. Load skew is related to how the data are allocated to different processors, and is linked to the uneven spread of input workload and output across different nodes. Operation skew signifies the combined effects resulting from the skew effects of a number of input relations, which may accumulate and reinforce each another.

To provide a concrete illustration of the principles, we make use of the following example. Two relations of the database, *customer* and *sales_order*, are shown in Tables 3.1 and 3.2. For these relations, we may have a natural join on the attribute *customer_no*. The result of the join will have nine tuples. We cluster the two relations by *customer_no* and thus may present the join as shown in Table 3.3 (where next to the attribute value is the number of times the value occurs).

Customer_No	Name	Address
1000	Smith A. B.	ABC, Victoria
1001	White B. C.	BCD, N.S.W.
1002	Naugh C.D.	CDE, Queensland
1003	Young D.E.	DEF, South Australia
1004	Hua E.F.	EFG, Tasmania
1005	DeWitt F.G.	FGH, N.T.

Table 3.1: Records Listing of Relation Customer (hereafter called **relation A**)

Sales_Order_No	Customer_No	Order_Status	Date
A101	1005	O.K.	1-1-94
A102	1002	O.K.	2-2-94
A103	1002	O.K.	3-3-94
A104	1002	O.K.	3-3-94
A105	1002	O.K.	3-3-94
A106	1000	O.K.	3-3-94
B102	1002	O.K.	3-4-94
B103	1002	O.K.	3-4-94
D456	1002	O.K.	3-8-94
D478	1002	O.K.	3-9-94
E100	1005	O.K.	4-4-94
E200	1002	O.K.	4-4-94

Table 3.2: Records Listing of Relation Sales_Order (hereafter called **relation B**)

Customer Relation (A)	Sales_Order Relation (B)	Join Result (# tuples)
(1000, 1) [4]	(1000, 1)	1
(1001, 1)	(1001, 0)	0
(1002, 1)	(1002, 9)	9
(1003, 1)	(1003, 0)	0
(1004, 1)	(1004, 0)	0
(1005, 1)	(1005, 2)	2

Table 3.3: Join Result Distribution

[4] The values in the bracket, in order, represent the Customer Number and the number of the times that customer appears in the table.

3.3.1 Data Skew

Data skew relates to the intrinsic distribution and occurrences of data values of particular attributes, and its presence is unrelated to whether a single processor or multiprocessor are used to process the data. It is caused by the non-uniform distribution and multiple occurrence of attribute values. When a relational operation is applied to such an attribute, the number of result tuples may vary significantly, as compared to a value distribution without data skew. In the above example, data skew is present in attribute *customer_no* of the relation B, where the tuple value 1002 occurs more frequently than that of 1001. The data skew is inherent in the data set and solely depends on the database applications. The knowledge about the data skew is essential to study the skew behaviour in parallel databases and to develop truly efficient algorithms for database operations. Data skew could be given *a priori* based on certain distributional assumptions, or by carrying out sampling of actual data values. The skewness of the tuples within the relation can be described in terms of coloured balls in an urn, where the colours correspond to the domain values. The question on how to estimate the number of colours present in the urn on the basis of sampling and knowledge of the total number of balls in the urn has been studied in [Good49]. The parent distribution of known size may be subdivided into an unknown number of mutually exclusive classes. In taking a random sample of n elements without replacement from the population, we can estimate the total number of classes. In other words, the number of domain values of the relation can be worked out from sampling and its cardinality. Therefore, data skew can be described in terms of duplicate values of the partitioning attribute.

Denoting by r the number of tuples in the relation R, and n the sample size. We let x_i signify the number of tuples for each domain value in the samples, and y the estimated number of domain values of relation R. Unlike the load and operation skew, data skew can be described with mean T_{DS} and its deviation V_{DS} since the individual domain value does not determine the distribution results

$$T_{DS} = \frac{r}{y}, \qquad V_{DS} = \sqrt{\frac{\sum_{i=1}^{n} (x_i - T_{DS})^2}{n}}. \tag{3-1}$$

The larger the variation from mean, the higher is the data skew present in the relation for that particular attribute.

The estimated number of domain values y is given by

$$y = \sum_{i=1}^{n} A_i x_i \tag{3-2}$$

where

$$A_i = 1 - (-1)^i \frac{(r-n+i-1)^{(i)}}{n^{(i)}}, \tag{3-3}$$

with the notation $m^{(i)} = m(m-1)(m-2)\ldots(m-i+1)$, so that

$$T_{DS} = \frac{r}{\left\{\sum_{i=1}^{n}\left[1-(-1)^i \frac{(r-n+i-1)^{(i)}}{n^{(i)}}\right]x_i\right\}}. \tag{3-4}$$

Generally, more samples will result in better accuracy, but the extra cost of sampling will need to be taken into account. Sampling issues can be found in [Sesh92]. In addition, we note that there is no data skew if the partitioning attribute is the primary key of the relation since $y \approx r$, $T_{DS} \approx 1$, and $V_{DS} = 0$.

3.3.2 Load Skew

While data skew is unrelated to the processing mode of relational operations, load skew is directly related to parallel processing and does not exist in the case of conventional uniprocessor operation. When a relational operation is allocated to more than one processor, load skew may occur because of the uneven load distribution. Again consider the join example given earlier and assume three processors participating in the join. Let the join domain be *customer_no*, and tuples of the two input relations are allocated according to range partitioning with two values assigned to each processor. Then three tuples (with values 1000 and 1001) will go to the first processor, eleven tuples (with 1002 and 1003)

will go to the second, and four tuples (with 1004 and 1005) will go to the third. All three processors receive different volume of tuples with a consequent effect on processing time. This non-uniform load distribution is referred to as load skew. Clearly, the load skew limits processor utilisation and the speedup of the operations by parallelism.

Since a relational operation involves several processing steps, the load skew may be further classified as follows based on its physical operations:

- *IO load skew (IOLS)*, which is caused by uneven I/O load distribution imposed on the processors to read input relations into their local memory. The I/O cost of each processor is mainly determined by the size of the fragment(s) of the input relation(s) that is allocated to it. In addition, the access paths of the relations may also affect the I/O cost. A relation with an index usually involves less I/O costs than that without an index since only the blocks that contain the required tuples will be retrieved.

- *operation load skew (OLS)*, which is determined by non-uniform CPU costs for searching tuples that belong to the result relation among the processors. Therefore, *OLS* depends not only on the partitioning of the input relations, but also on the processing algorithms that are used at different processors. For example, when one of the input relations is small or has index, the nested-loop join involves the smallest cost; otherwise the hash join may perform better than the nested-loop join.

- *result load skew (RLS)*, which refers to the skewed loads for the processors generating the result of the operation. Some processors may generate a large number of result tuples while others may only have a few, leading to different CPU costs. *RLS* depends on the selectivity factor of the operation over the fragments allocated to the processors.

Figure 3.2 shows the three different load skews over some common relational operations and associated processing methods. The following assumptions are made in Figure 3.2.

- The input relations are R and S, with $r = |R|$, $s = |S|$, and $r < s$;
- The length of predicates and the number of columns to be projected are one;

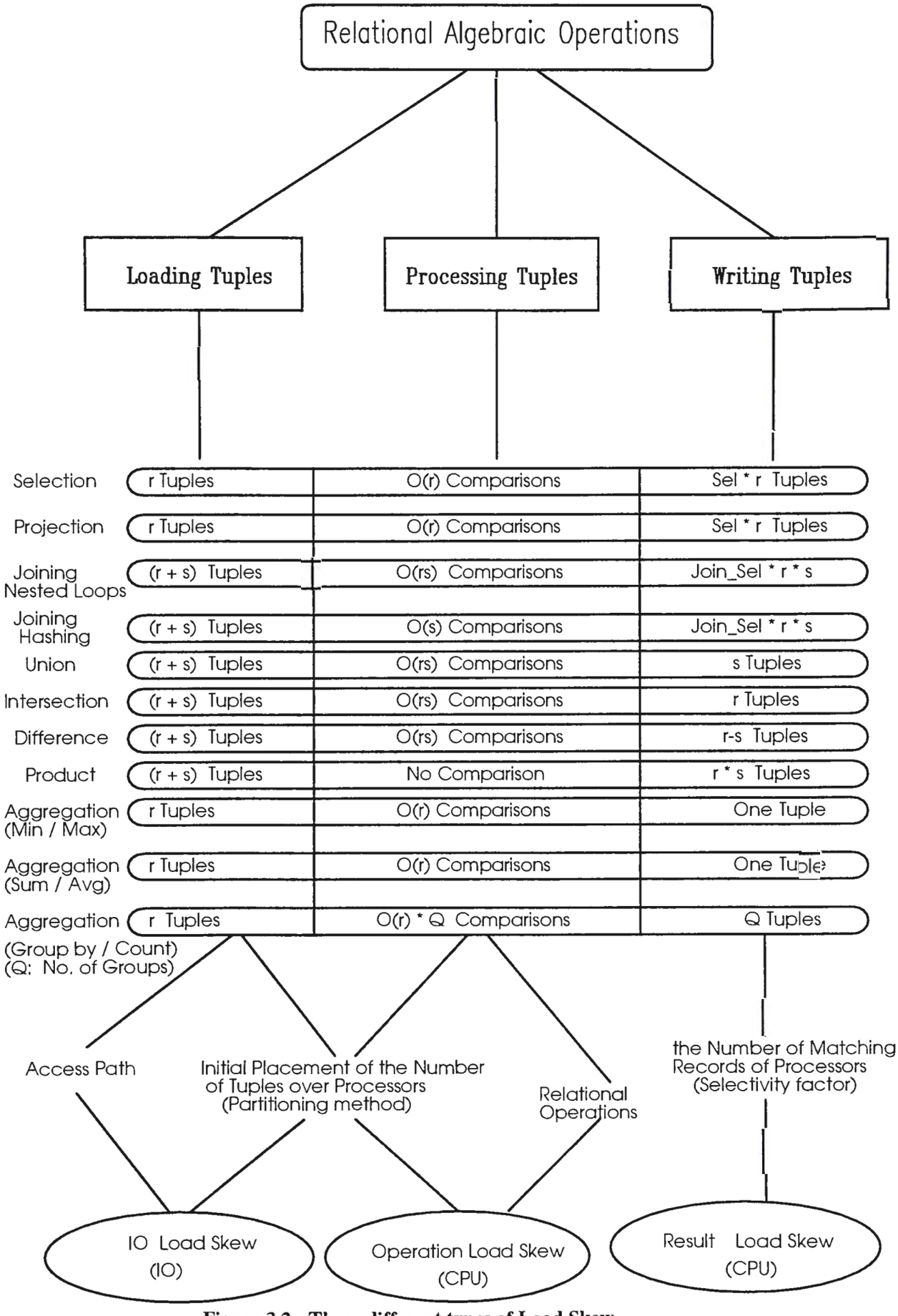


Figure 3.2: Three different types of Load Skew

- *Sel* and *Join_Sel* are the selectivity for Selection and Joining;
- To save memory, hash-based join algorithms build hash table with the smaller input relation;
- Memory size is large enough to hold entirely either R or S;
- For set theoretic operations (Union, Intersection, and Difference), the two input relations (R and S) are compatible with $R \subset S$.

We primarily focus on a shared-nothing distributed memory architecture. In such an architecture, the total load skew may be obtained as follows. We let N denote the number of nodes, where a node includes the I/O as well as processor components. Here, we assume that there is a single processor to each node. In the general case, a node may consist of a collection of heterogeneous processors operating on a shared memory basis. The following notations are used.

r_i: the number of tuples in the ith node after the partitioning of relation R

$r \geq r_i \geq 0, \ i = 1, ..., N;$

σ_R: selectivity factor of the relation R of a given operation;

σ_i: selectivity factor of the ith fragment in the relation R of a given operation after partitioning ;

R_i: total load of the ith node of relation R;

R_{Skew}: the maximum imbalanced load associated with relation R;

T_{LS}: load skew factor $(0 \leq T_{LS})$.

The total output from an operation is

$$r\sigma_R = \sum_{i=1}^{N} (r_i \sigma_i) = r_1\sigma_1 + r_2\sigma_2 + ... + r_N\sigma_N \ .$$

Now, the workload of each node consists of three components

W_1: loading cost for each tuple (including disk access time and transfer time),

W_2: processing cost for each tuple (mainly comparison and computation time as reading time from RAM should be comparatively small),

W_3: writing cost for each tuple.

A high *IOLS* does not necessarily mean high total load skew as both processing and writing cost (*OLS* and *RLS*) need to be taken into account. The total load is the weighted arithmetic mean

$$R_i = \frac{W_1 r_i + W_2 r_i + W_3 r_i \sigma_i}{W_1 + W_2 + W_3}, \tag{3-5}$$

and the total load imbalance as represented by the deviation from the perfectly balanced situation is given by

$$R_{Skew} = \max(R_1, \ldots, R_N) - \frac{1}{N}\left(\sum_{i=1}^{N} R_i\right). \tag{3-6}$$

The load skew factor is given by

$$T_{LS} = \frac{\max(R_1, \ldots, R_N)}{\frac{1}{N}\left(\sum_{i=1}^{N} R_i\right)}, \tag{3-7}$$

where $1 \le T_{LS} \le N$, and the larger this factor, the higher the load skew occurs over the nodes.

Equal partitioning has been mistakenly regarded as the aim of parallel processing. Figures 3.3 and 3.4 show that there still exists load skew even when the relation is perfectly partitioned. This is because selectivity factor of each fragment may also contribute to load skew. Assuming $W_1 = W_2 = W_3 = t$, where t is a standard cost unit, and $r_i = \frac{r}{N}$, where the relation is evenly partitioned to all processors, we have $R_i = \frac{r_i(2+\sigma_i)}{3}$, $T_{LS} = \frac{N(2+\sigma_{\max})}{2N+1}$, and the load skew factor against the number of processors is plotted in Figure 3.3.

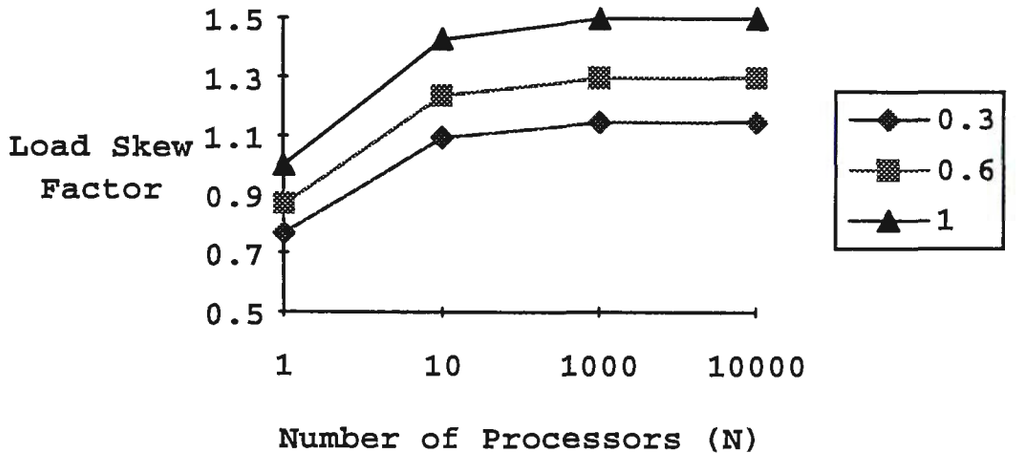


Figure 3.3: Load Skew with Maximum Partition Selectivity

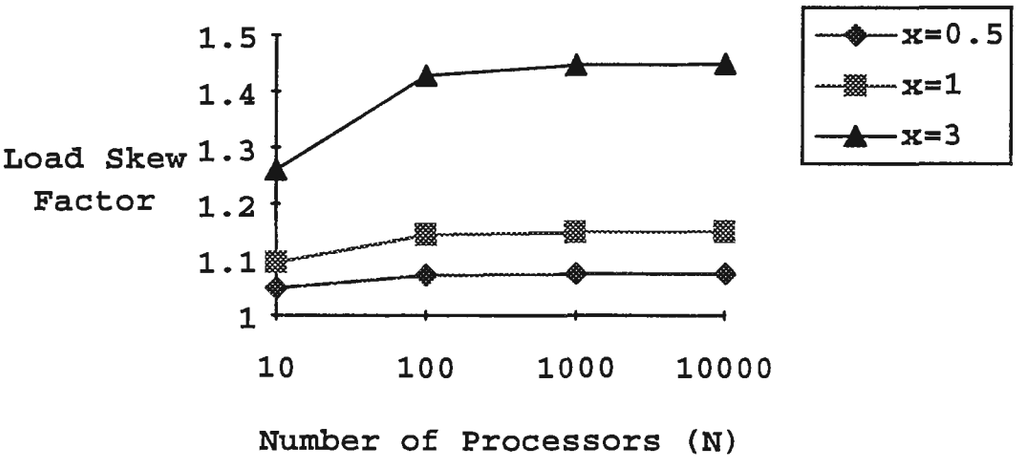


Figure 3.4: Load Skew with Writing Cost

The data in the legend of Figure 3.3 refer to the maximum partition selectivity; that is 0.3 means the highest selectivity factor among all partitions is 0.3. As we increase the number of parallel processors, the load skew rises and approaches a constant level. Moreover, decreasing the maximum partition selectivity factor will reduce the load skew. Therefore, from Figure 3.3, we can conclude that load skew still exists in evenly partitioned relation over processors because of the varying selectivity factor(s) among the partitions.

Assuming $W_1 = W_2 = t$, $r_i = \frac{r}{N}$, $\sigma_{max} = 0.3$, and $W_3 = xt$ where x is the cost ratio of writing over processing tuples, we have $R_i = \frac{r_i(2 + x\sigma_i)}{2 + x}$ and $T_{LS} = \frac{N(2 + x\sigma_{max})}{2N + x}$.

The cost ratio of writing over processing tuples is introduced and its effect on load skew is shown in Figure 3.4. Increasing either the number of processors or the cost ratio will lead to higher load skew. From Figure 3.4, we see that the cost ratio of writing and processing tuples, working in conjunction with the partition selectivity factor, has a significant influence on load skew.

For a shared-nothing architecture, since load skew is *mainly* caused by non-uniform placement of data over processors, the methods which partition the input relation(s) onto the nodes will significantly contribute to the load skew. However, a partitioning method may not be the sole cause of all the above load skews. For example, the round-robin partitioning for a unary operation such as selection, does not involve *IOLS* and *OLS* since the tuples of the input relation(s) are evenly spread over the processors. However, hash join partitioning would involve all types of load skews. In summary, *IOLS* would be caused by allocating the tuples with the identical join attribute values to the same processors in the presence of data skew. The various relational algebraic operations and the existence of the data skew create *OLS*. In comparison, *RLS* should only occur when the number of matching records of processors are different, and depends on selectivity factor.

3.3.3 Operation Skew

Given a relational operation, skew may also be classified into no operation skew (*NOS*), single operation skew (*SOS*), and double operation skew (*DOS*) depending on how many input relations are skewed (i.e. the *skew dimension*) after partitioning. Multiple operations can always be regarded as a series of binary operations. If the tuples of both operand are evenly allocated over the processors, it is *NOS*. *SOS* indicates that one input relation has load skew in the attribute(s) related to the operation, whereas *DOS* relates to the load skew on both input relations for binary operations, such as join. Evidently, unary relational operations, such as selection and projection, may have either *SOS* or *NOS*, while binary operations are likely to have any one of them. In the earlier example, there is only a *SOS* (due to relation *B*) in the join operation described above.

It is worth noting that *DOS* is complex but *normally* only appears in the operations which use the intermediate results of other operations. For the base relations, it is pointed out in

[Elma94] that when a binary 1:1 or 1:*N* relationship type is involved, a single join operation is usually needed; For a binary *M*:*N* relationship type, two join operations are needed. For example, joining *Sales_Order* and *Product* is actually doing two joins, *Sales_Order* and *Order_Product* (join attribute is *Sales_Order_No*), and *Order-Product* and *Product* (join attribute is *Product_No*). Therefore, it is not a join of *DOS* but two joins of *SOS*. Furthermore, duplicate attributes (except foreign keys) are avoided in the database design because they cause anomalies (deletion, insertion, and update). Therefore, the operations on the normalised database relations do not often involve *DOS*.

If we adopt the following notations

s_i: the total load of the *i*th processor after partitioning relation *S*,

u_R: the average load associated with relation *R* over *N* nodes

$$u_R = \frac{1}{N}\left(\sum_{i=1}^{N} R_i\right),$$

u_S: the average load associated with relation *S* over *N* nodes

$$u_S = \frac{1}{N}\left(\sum_{i=1}^{N} S_i\right),$$

R_{Skew}: the maximum imbalanced load associated with relation *R*,

T_{OS}: operation skew factor,

then for unary operations such as Selection (relation *R*), we have

NOS: $T_{OS} = 1$

SOS: $T_{OS} = T_{LS}$ (same as load skew);

and for binary operations such as Nested-Loops Join (relation *R* and *S*), we have

NOS: $T_{OS} = 1$

SOS: $T_{OS} = T_{LS} \times 1$ (same as load skew)

DOS [5]: $\max(R_1S_1,\ldots,R_NS_N)/(u_R\ u_S)$.

That is,

[5] Double skewed relational operation over processors may also produce "No Operation Skew" because the two load skewed relations can cancel out the effect of each other.

$$T_{OS} = \begin{cases} \dfrac{\max(R_i)}{u_R}, & \text{unary operations} \\ \dfrac{\max(R_i S_i)}{u_R u_S}, & \text{binary operations} \end{cases} \qquad (3\text{-}8)$$

where $T_{OS} \geq 1$, and the larger this factor, the higher is the operation skew over the nodes.

3.4 Performance Degradation

Data skew, load skew, and operation skew are not isolated but closely related. Their relationships are presented in Figure 3.5. Data skew cannot be changed, whereas load skew and operation skew could be avoided using various skew handling approaches (see Chapters 4 and 5, and [Liu96b]).

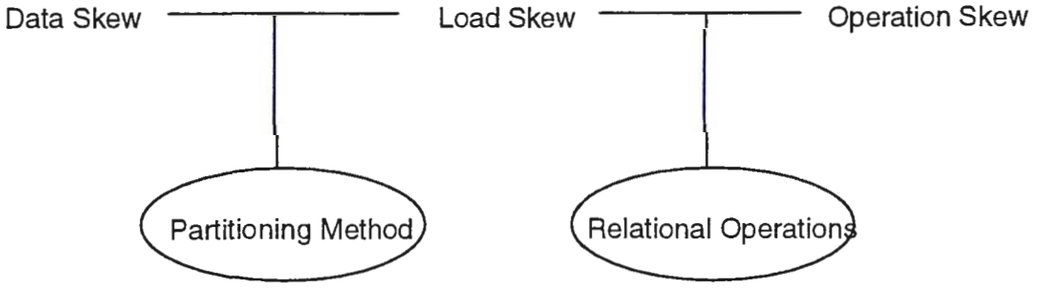


Figure 3.5: Data Skew, Load Skew, and Operation Skew

In equation (3-5), the parameter r_i is the tuple allocation over processors, and clearly data skew has significant influence on r_i. Supposing there is *no data skew,* load skew can still occur because of uneven partitioning and variation of the selectivity factor. Assuming a perfect hash function is employed in a situation where data skew is absent, we obtain the following bounds for the load skew

Lower Bound: $\sigma_i r_i = \dfrac{r\sigma_R}{N}$

$R_{Skew} = 0$

Upper Bound: $\sigma_i r_i = r_i$

$$R_{Skew} = \frac{r_i(W_1 + W_2 + W_3\sigma_i)}{W_1 + W_2 + W_3} - \frac{1}{N}\left(\sum_{i=1}^{N} R_i\right)$$

$$= \frac{r}{N} - \frac{1}{N}\left(\sum_{i=1}^{N} R_i\right),$$

where the last equality is obtained by noting that $r_i = r / N$. The question arise is: "If there is *data skew*, what effect does data skew have on load skew?". Our answer is data skew affects load skew through partitioning method. Round-robin, range partitioning and hashing are the three most common partitioning strategies. Among them, in terms of load balancing, round-robin is the best policy as it destroys the effect of data skew entirely if it exists. Unfortunately, round-robin policy can only be feasibly adopted with unary operations and compulsory sequential reading. On the other hand, data skew has influence on range partitioning and hashing, and in most cases, data skew exacerbates load skew except where proper splitting function is employed in which case data skew can offset the impact of load skew.

Skew degrades the system performance in a number of ways. It causes bucket overflow in parallel processors with shared-nothing or shared-disk architecture. If a highly skewed relation is partitioned into N parts without any load balancing mechanism, it is possible for one processor get all the work while all other processors have nothing in the extreme case. The individual bucket (memory such as RAM) of the processors may not have enough space to store the entire relation, resulting in bucket overflow, in which case it may be necessary to re-partition the relation.

Skew also causes load imbalance by directing different tuple volumes to different processors. The most lightly loaded processor has to wait until the heaviest loaded one finishes. Thus scaleup and speedup cannot achieve the expected linear results, and it is possible that system performance may be even worse than that of the uniprocessor case since the use of multiple processors involves extra overheads.

Skew may also cause problem on the network and connecting processors. If one processor is over utilised, the corresponding I/O operations and the CPU time for the processor outweigh others. Clearly, disk I/O is a particular problem in database systems because of its slow access time, typically 10,000 times of main memory access time. In addition,

various processors must exchange information such as synchronisation data, concurrency control messages and some of the intermediate results. Eventually, the heavily loaded processor becomes the bottleneck (hot spot) of the system and can even cause congestion of the communication network.

3.5 Summary and Concluding Remarks

We have provided a new taxonomy of skew for parallel database systems, with attention focused on the relational model operating in a shared-nothing distributed memory architecture in SPMD mode. Three types of skew are identified: data skew, load skew, and operation skew. Data skew is an intrinsic property of the attributes related to the naturally occurring replication and clustering of the underlying data values irrespective of how the data are processed. Load skew is related to how the data are allocated to different processing nodes, and is caused by the non-uniform spread of input workload and output across different processors. Operation skew is caused by the combined effects of a number of input relations, which may reinforce or neutralise each another. Load skew may further be distinguished into I/O skew, operation skew, and result skew. These different types of skew have been quantitatively analysed and expressions are provided for their evaluation. In particular, it is shown that skew effect can be significant even when the data partitioning is carried out uniformly for all processors. In addition, performance bounds on best and worst case behaviour are also derived.

The skew taxonomy further clarifies the problem of load balancing in parallel database systems, identifies the main issues causing skewness, and provides alternatives to solving the problem. While most of the existing works try to partition the relation evenly to all processors, in the light of our taxonomy, it becomes clear that this is not enough since the total load consists of three components, loading, processing, and writing. Equal sized fragments for each processor only mean equal loading and processing cost among processors; but some processor(s) might do fruitless work because their fragment's selectivity is zero which is likely in some situations (e.g. Select a tuple from a relation).

Skew handling involves two basic steps: skew estimation and skew management (see Figure 3.6). Skew estimation attempts to identify the existence of the data skew or load skew among parallel processors and, if exist, estimate the degree of the corresponding skew.

Based on the knowledge of the first step, the skew management procedure then attempts to avoid or solve skewed load distribution. In our opinion, skew estimation has been largely neglected compared with skew management.

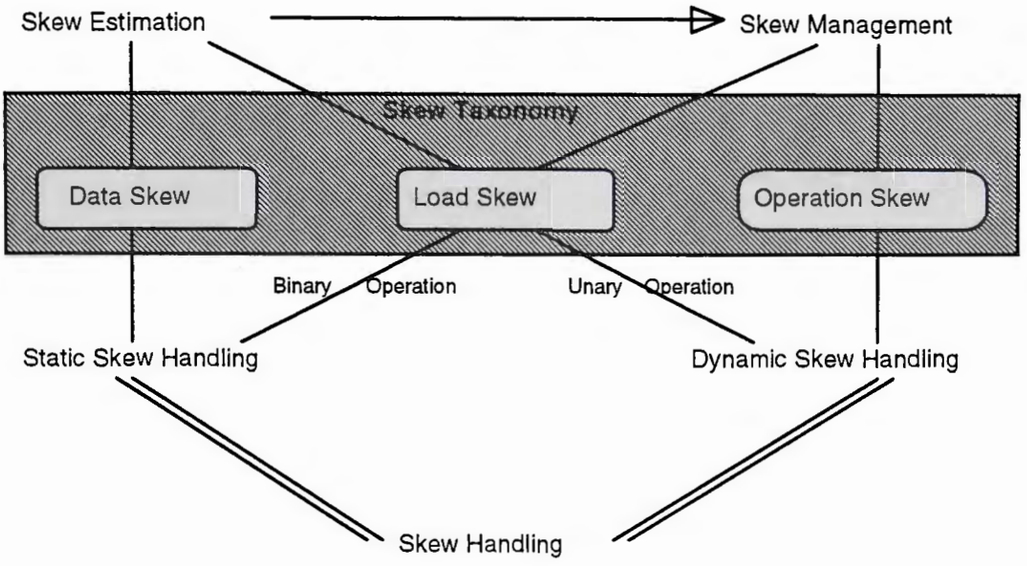


Figure 3.6: The Usage of Skew Taxonomy

Another way to look at the skew taxonomy is in static and dynamic skew handling methods (Figure 3.6). The former approaches the problem before dividing the total workload and the main focus is on data skew (or load skew for binary relational operations). It is normally feasible for simple operations such as unary operation and binary join. However, it will become highly complex as the number of processors increases and in situations where multiple queries with multiple operations are involved. The static handling method will need to perform estimation for every input relations and intermediate result relations. As it is not always possible to estimate precisely the host of parameters, it often leads to unrealistic bounds on the results. The latter approaches the problem at run time by waiting for operation skew (or load skew for unary relational operations) occurrence and resolving the skew. Generally, it does not require preprocessing and relies on no distribution assumptions. However, taking overheads into account, the extent of performance gain of this approach is not always certain with the result that few systems will take the risk of adding in extra complexity and cost to the existing system through the deployment of these algorithms. Moreover, the efficient and effective overall objective function in terms of both cost and response time is hard to obtain.

CHAPTER 4

SKEW PREDICTION AND MODELLING

4.1 Introduction

Parallel processing provides a useful solution in very large databases where, in theory, a near linear speedup of database operations is thought to be possible [DeWi92a, Moha94, Vald93]. However, linear speedup is not achievable in practice not only because of high communication overheads (initiation and termination) and low resource utilisation, but also because of the non-uniform load distribution over the processors -- the problem of skew (see Chapter 3). The study of skewness may be classified into empirical skew handling and

theoretical skew prediction. The former has attracted the attention of many researchers, and a number of skew handling algorithms have been proposed [Liu96b, DeWi92b, Hua91, Kits90, Lu92, Omie91, Wolf93a Wolf93b], while the latter has been largely neglected [Falo96].

With the increasing complexity of parallel databases, it is difficult to accurately predict the execution time since it requires a communication model and a computation model. The communication model is characterised by parameters such as the inter-processor communication, the ratio of network bandwidth to local memory or local cache bandwidth, the contention for the destination, and the computation overheads of transmitting and receiving messages which are independent of transmission latency between processors [Dris95]. On the other hand, the computation model is concerned with different relational operations carried out at each processor and the methods employed for processing these operations. Exploiting parallelism in parallel databases has been under investigation for many years [Bitt83, Mish92, Pang93, Pira90, Qada88, Echn89], but the main barrier in formulating a precise computation model is the lack of adequate skewness description. Here, we present a complete analytical foundation model for the study of skewness based on the extreme value distribution theory, and Chapter 5 will expand the foundation model by describing data skew using Zipf distribution and Normal distribution. Two main data partitioning methods, range and hashing, are analysed and their influences on load balancing are predicted quantitatively.

An analysis and taxonomy of skew in parallel database processing is presented in Chapter 3 and three kinds of skewness, namely, data skew, load skew, and operation skew are identified. Figure 3.5 shows the relationships among data skew, load skew, and operation skew. It goes without saying that skew exists not only in intra-operation level but also in inter-operation and inter-query levels. However, we believe that the upper level parallelism can not proceed efficiently and effectively without the solving the lower level skewness.

This Chapter focuses on skew prediction of range partitioning and hash partitioning in the absence of data skew at the intra-operation level. In a given query, the data skew presented in the relevant base relations are fixed, and thus the optimal system throughput mainly depends on the data partitioning methods. A detailed skew analysis is presented based on range partitioning. In addition, a complete analytical model of load skew under hash partitioning is developed, and it shows that even when data skew is absent, load skew may

still be present. Moreover, as data skew increases, load skew will be magnified supra-linearly. In both situations, the effects of both maximum and minimum load skew are quantified, and the standard deviation of the mean load and the distribution functions for all extreme values (minimum and maximum) are provided in the foundation model based on the number of processors and the cardinality of the relation.

The remainder of the Chapter is organised as follows. Section 4.2 provides a comprehensive discussion of the data partitioning methods. The skew problem with range partitioning is discussed in Section 4.3 and a detailed skew analysis of range partitioning is provided in Section 4.4. The skew problem with hash partitioning is described using urn models in Section 4.5. Section 4.6 introduces the load skew foundation model and section 4.7 presents the operation skew foundation model under hash partitioning. Finally, a summary is included in section 4.8.

4.2 Partitioning Methods

To process database operations in parallel, three common data partitioning methods are often used: round-robin, hashing, and range partitioning. Round-Robin distributes the tuples of one relation in a round-robin fashion; range partitioning sends tuples to processors according to ranges (boundary values); hash partitioning transfers tuples to various processors using a hash function to determine the destinations. These three partitioning strategies have already been widely used in real database systems such as Teradata (hashing), Tandem (range partitioning), Bubba (hashing and range partitioning), and Gamma (all three strategies).

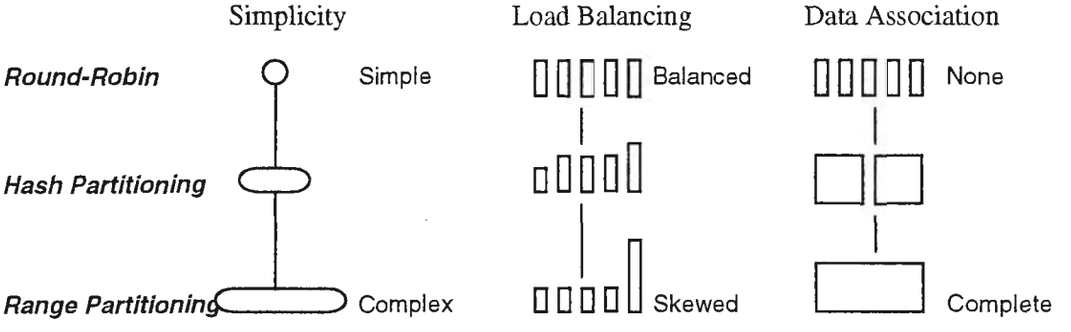


Figure 4.1: Comparison of Three Partitioning Methods

In terms of simplicity, round-robin is the best choice, followed by hashing which needs to

apply a hash function, and range partitioning which involves a number of comparisons and a range table. From the load balancing point of view, range partitioning is likely to have higher degree of skewness; hashing is better than range partitioning because it involves some degree of randomisation; round-robin is the only policy providing complete evenness, cancelling the effect of data skew entirely. However, range partitioning maintains the association of data and provides better control of outgoing tuples, facilitating not only exact match retrieval but also range processing, e.g. activating only relevant processors by applying predicate with boundary values. By comparison, round-robin and hashing must direct the range predicate to all processors. In addition, round-robin demolishes the data association, and so would only be useful in situations where a complete sequential scan is needed. Figure 4.1 gives a summary of partitioning methods. Round-Robin gains load balancing with its complete evenness by sacrificing data association; at the other extreme, range partitioning sustains the association of data but has poor load balancing; hashing is a compromise of round-robin and range partitioning, and may be viewed as either an enumerated range partitioning method or a round-robin policy with distinct values.

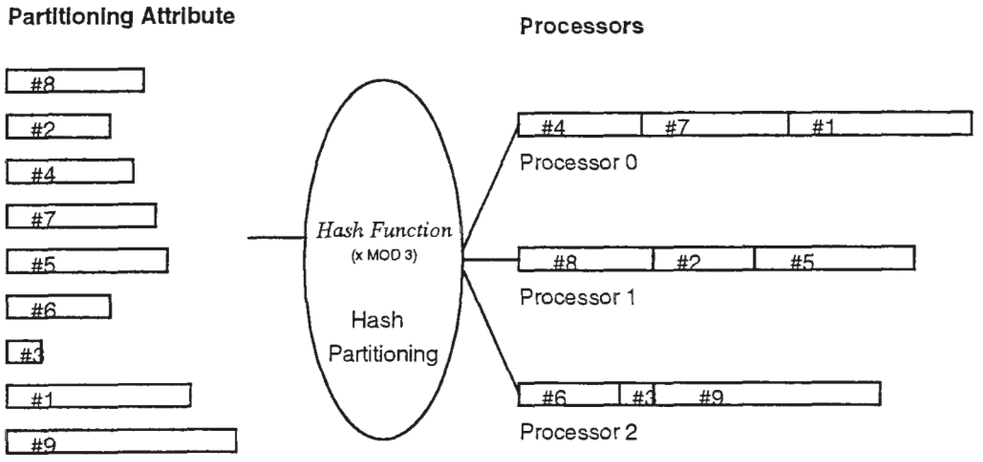


Figure 4.2: Hash Partitioning

Hash partitioning strategy has been widely used for both unary and binary relational algebra operations such as selection, projection and join. Consider an example of hash partitioning on a single relation as shown in Figure 4.2, where the length of the horizontal bars on the left side of the figure represents the number of occurrences of the attribute values before partitioning. The right side of the figure is the distribution results after partitioning. This example shows that data skew (left) affects load skew (right) through hashing, and the prediction on the influences of hash partitioning strategy on load skew is provided from Section 4.5 to Section 4.8.

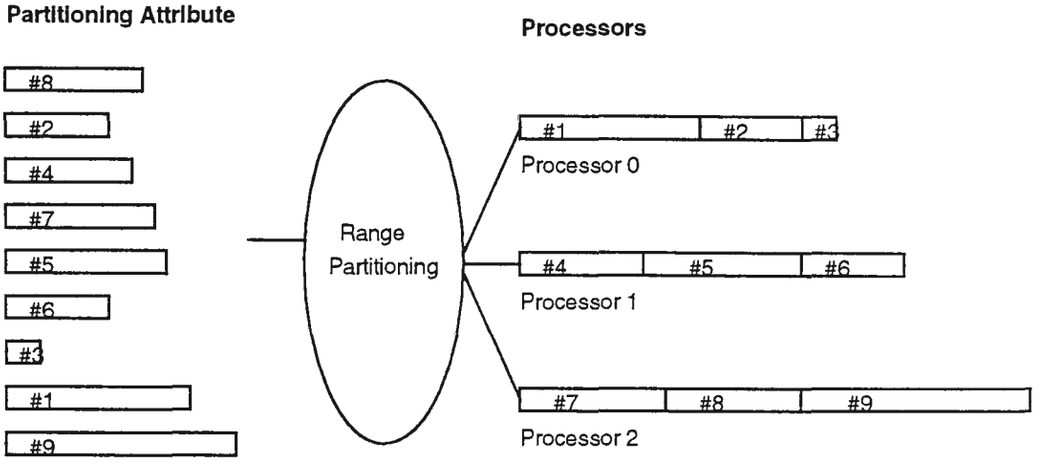


Figure 4.3: Range Partitioning

An example of range partitioning is shown in Figure 4.3 where tuples are distributed according to boundary values, i.e. #1 to #3 for Processor 0, #4 to #6 for Processor 1, and #7 to #9 for Processor 2. At the left hand side of the figure, the length of each bar represents the number of tuples of one specific domain value in a relation. We can see from the figure that the distribution result on the right hand side is different from that of using hash partitioning but this method also results in load skew over processors. Skewness analysis of binary operations using range partitioning is introduced in Sections 4.3 and 4.4.

4.3 A Case Study of Using Range Partitioning for Parallel Processing

By range partitioning, relations are horizontally distributed to multiple processors. The method maintains the data association and has better control of outgoing tuples in a unary operation, but it may involve a high degree of skew in binary relational operations. Here, we show query examples to illustrate the skewness caused by data partitioning. A detailed analysis is provided in Section 4.4 and both of Section 4.3 and Section 4.4 assume a range partitioning strategy.

The following is a legal court management system of relational database design. Within the system, each office has a number of lawyers; each case is heard in only one court session and presided over by one judge; each client in a case is represented by one lawyer; all documents have unique internal numbers within the case.

CLIENT (**client#, client-name, sex, age, address, city, lawyer#**)
LAWYER (**lawyer#, lawyer-name, phone, office#**)
CASE (**case#, date, judge**)
COURT_CASE (**client#, lawyer#, case#, time, court-room, address, city**)
DOCUMENT (**document#, type, date-prepared, case#**)
OFFICE (**office#, area, level, person-in-charge**)

Query 4.1: **SELECT** client#, client-name, age
FROM client
WHERE sex='M' and age>40 and city='MELBOURNE'

Query 4.2: **SELECT** document#, type, date-prepared, case#
FROM document
WHERE document#=123456

Query 4.3: **SELECT** client#, client-name, client.city, court_case.time,
court_case.court-room
FROM client, court_case
WHERE client.city=court_case.city and court_case.court-room='n110'

Query 4.4: **SELECT** lawyer#, lawyer-name, office#, level, person-in-charge
FROM lawyer, office
WHERE lawyer.office#=office.office# and person-in-charge='SMITH NEWS'

Parallel processing of unary operations using range partitioning does not involve load skew since relations can always be partitioned evenly. In most cases, the cardinality of the relations is known and a range table is constructed for distribution purpose. Therefore, each processor receives an equal number of tuples and the boundary values for each processor are also recorded in the range table. The selection criteria of incoming query is applied to the range table and sent to only relevant processors for processing.

In Query 4.1, we want to list the male clients who lives in Melbourne and their age is over 40. The client# is selected as partitioning attribute and the predicates are forwarded to all processors since the predicate attribute is different from the partitioning attribute. Assuming there are 100 tuples in the client table and 10 processors, we send the first 10 tuples to the

first processor and record the boundary values in the range table. In such a way, unary operation can always be evenly partitioned based on range partitioning. In Query 4.2, the partitioning attribute, document#, is also the predicate attribute, so that the predicate is only forwarded to a set of processors in accordance with the range table. Again, equal partitioning is assured.

For a binary operation, the same technique may be applied to the first relation and then the second relation is partitioned based on the constructed range table. As a result, there is little control on the outgoing tuples of the second relation, which may cause load imbalance.

In Query 4.3, we can select either an unsorted attribute, e.g. client-city, or a sorted attribute, e.g. client#, but the entire second relation has to be broadcasted to every processor. In this case, there is no load skew. However, the principle of relational database is that data are stored in simple tables and all tables are linked using duplicate attributes (i.e. the foreign key). Hence, most of the binary join operations make use of the foreign key attribute which is sorted in the referenced table [Ullm89]. An example of this is shown in Query 4.4 and clearly the partitioning attribute is office.office#. The office table is evenly partitioned first and the range table is constructed. Applying the range table to the second table, we distribute the fragments of different sizes over processors. Consequently, load skew may occur due to less control and in Section 4.4 we will analysis and quantify the degree of load skew caused in this situation, i.e. parallel processing of binary operations using range partitioning on a sorted attribute.

4.4 Skewness Analysis of Range Partitioning

In this section, we provide a theoretical treatment of skewness in binary operation using range partitioning for parallel processing, together with useful closed-form formula for the operation performance prediction in the presence of data skew. Parallel processing of binary operation with range partitioning consists of:

Step 1: equal partitioning based on the area covered for the first operation. Then, obtain the boundary values a_1, a_2, ..., a_i, a_{k-1} assuming there are k processors.

Step 2: apply these boundary values to the second relation and thus partition the second relation into k partitions. The size of each partition is represented by the area covered by the function curve and boundary intervals. e.g. the size of the first partition in Figure 4.4 is the shaded area[6] bounded by R_0 and a_1 under the curve $f(z)$.

Let the attribute value z be distributed over the interval $I:[R_0, R]$, where $R > R_0$, and has density function

$$\begin{cases} f(z) & R_0 \le z \le R \\ 0 & z > R \text{ or } z < R_0 \end{cases}.$$

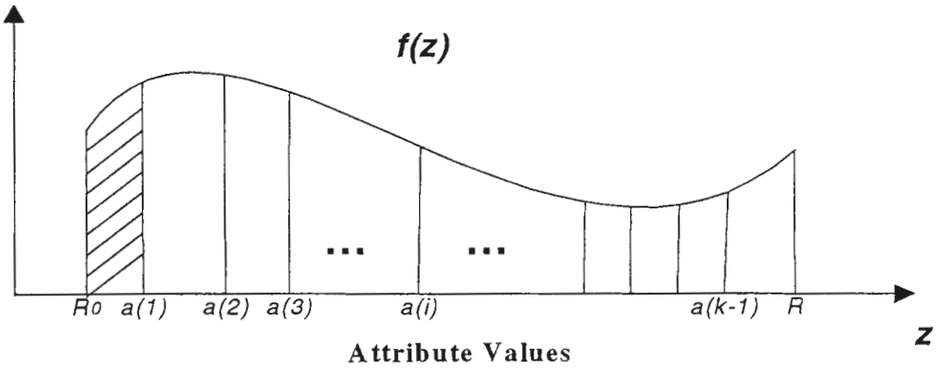


Figure 4.4: Partitioning the Second Relation using Range Partitioning

With k processors, we divide I into k unequal length subintervals $\{I_i\}_{i=1}^{k}$, i.e.

$$I_1 = [R_0, a_1],\ I_2 = [a_1, a_2],\ ...,\ I_i = [a_{i-1}, a_i],\ ...,\ I_k = [a_{k-1}, R].$$

Let there be a total of x tuples distributed over I, then the jth processor will have a workload of

$$\int_{I_j} xf(z)dz = x\left[F(a_j) - F(a_{j-1})\right]$$

[6] This area can be calculated using mathematical software such as *Mathematica* or a simulation software assuming $f(z)$ is reasonable simple and its integration interval is easy to define.

tuples, where $F(.)$ is the cumulative distribution function. The most heavily loaded processor is the j^* corresponding to

$$\max_{1\le j\le k}\left[F(a_j)-F(a_{j-1})\right]$$

with p_{max} given by

$$p_{\max} = F(a_{j^*})-F(a_{j^*-1}).$$

4.4.1 Unimodel Distribution

For a unimodel distribution, the k quantities in the above equations need not all be evaluated and j^* may be taken to be the processor where the mode occurs. The mode m may be located using the conditions $f'(m)=0$ and $f''(m)<0$.

Although this may not always correspond to the most heavily loaded processor, it should provide a close approximation to system performance. Figure 4.5 shows an example of this situation where the area <1> with the mode m is less than the area <2> covered by its neighbour subinterval.

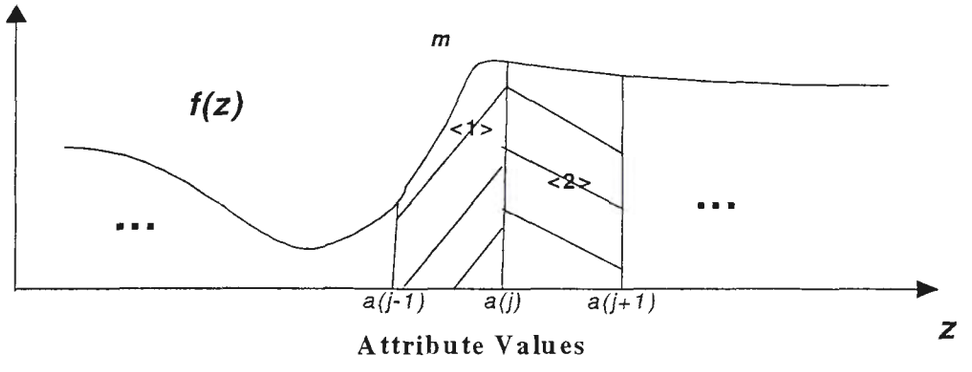


Figure 4.5: A Special Case

4.4.2 Multimodel Distribution

For multimodel distribution with modes m_1, m_2, ..., m_q, the evaluation of the k quantities in the above equations may similarly be simplified by confining attention to

$$\max_{j \in s}\left[F(a_j) - F(a_{j-1})\right],$$

where s if the set of q subintervals containing $m_1, m_2, \ldots, m_q$ as shown in Figure 4.6.

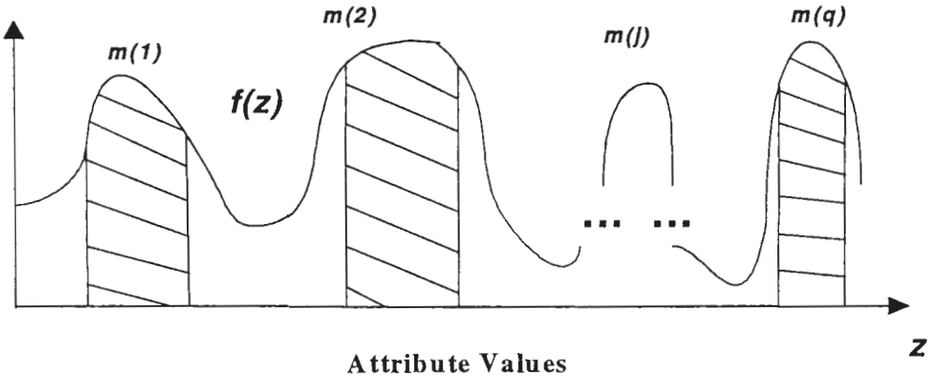


Figure 4.6: Multimodel Distribution

4.4.3 Erlang Distribution

Supposing the attribute values are distributed according to the *Erlang Distribution* which can be used to model a wide variety of characteristics

$$\eta(z) = \frac{\rho^r z^{r-1} e^{-\rho z}}{(r-1)!}, \qquad z>0$$

with the mean r / ρ and the standard deviation $\sqrt{r} / \rho$.

When confined to a finite interval $[0, R]$, then it becomes the truncated density

$$f(z) = \frac{c\rho^r z^{r-1} e^{-\rho z}}{(r-1)!}, \qquad 0 \le z \le R$$

where $c = \left[\int_0^R \frac{\rho^r z^{r-1} e^{-\rho z}}{(r-1)!} dz\right]^{-1}$.

For $r>1$, the mode of $f(z)$ is found from

$$f'(z) = \frac{c\rho^r (r-1) z^{r-2} e^{-\rho z}}{(r-1)!} + \frac{c\rho^r z^{r-1} (-\rho) e^{-\rho z}}{(r-1)!} = 0$$

$$\Rightarrow (r-1) z^{r-2} - \rho z^{r-1} = 0$$

$$\Rightarrow (r-1) = \rho z \quad \text{or} \quad z = \frac{r-1}{\rho}.$$

Thus, we obtain

$$j^* = \left\lceil \frac{r-1}{k\rho} \right\rceil .$$

Hence p_{max} may be taken to be

$$p_{\max} = F\left(a_{\left\lceil \frac{r-1}{k\rho} \right\rceil} \right) - F\left(a_{\left\lfloor \frac{r-1}{k\rho} \right\rfloor} \right) .$$

Quadrature is normally required for the evaluation of $F(.)$. However, for $R>>1$, $F(.)$ may be approximated by [Cox62]

$$F(z) = 1 - \sum_{i=0}^{r-1} \frac{e^{-\rho z} (\rho z)^i}{i!}$$

This gives

$$p_{\max} = \sum_{i=0}^{r-1} \frac{\exp\left(-\rho a_{\left\lfloor \frac{r-1}{k\rho} \right\rfloor} \right) \left\{ \rho a_{\left\lfloor \frac{r-1}{k\rho} \right\rfloor} \right\}^i}{i!} - \sum_{i=0}^{r-1} \frac{\exp\left(-\rho a_{\left\lceil \frac{r-1}{k\rho} \right\rceil} \right) \left\{ \rho a_{\left\lceil \frac{r-1}{k\rho} \right\rceil} \right\}^i}{i!},$$

i.e.

$$p_{\max} = \sum_{i=0}^{r-1} \frac{1}{i!} \left\{ \exp\left(-\rho a_{\left\lfloor \frac{r-1}{k\rho} \right\rfloor} \right) \left\{ \rho a_{\left\lfloor \frac{r-1}{k\rho} \right\rfloor} \right\}^i - \exp\left(-\rho a_{\left\lceil \frac{r-1}{k\rho} \right\rceil} \right) \left\{ \rho a_{\left\lceil \frac{r-1}{k\rho} \right\rceil} \right\}^i \right\} .$$

When $r = 1$, $\eta(z)$ yields the exponential distribution $\eta(z) = \rho e^{-\rho z}$, and thus we have

$$f(z) = \frac{\rho e^{-\rho z}}{1 - e^{-\rho R}}, \qquad 0 \le z \le R .$$

As $f(z)$ is a decreasing function, then evidently the most heavily loaded processor is the first one, i.e.

$$p_{\max} = \int_0^{R/k} f(z)dz = \frac{1 - e^{-\rho R/k}}{1 - e^{-\rho R}} .$$

4.5 Problem Description and Urn Models for Hash Partitioning

As in [Azar94], the skew problem of hash partitioning may be described by the urn model of coloured balls, where each of the colours refers to one unique domain value with each ball representing an individual tuple, and each urn corresponding to a processor. Hence, a relation consists of coloured balls. During the partitioning, the balls are assigned to urns according to their colours, i.e. balls of the same colour will be sent to one and only one urn based on hashing. Figure 4.7 shows the data skew, load skew, and their relationship in parallel database systems. The number of appearances of each value before processing relates to data skew whereas the number of tuples in each processor after partitioning relates to load skew. The load skew can be represented as a function of data skew and the number of processors, and the relationship between data skew and load skew is reflected by the transforming hash function. Hereafter, the skew model is based on hash partitioning method which can be regarded as randomly distributing balls to urns.

There are two relations, R and S, in the case of a binary operation. Therefore, the balls of S are distributed to the same number of urns randomly after the completion of the distribution

of R. This may result in different load skew distribution of operand relations as shown in Figure 4.8. In this case, the most heavily loaded urn of R matches the lowest loaded urn of S and thus reduces the imbalance. The number of balls in each urn after distributing balls of R and S gives the degree of operation skew.

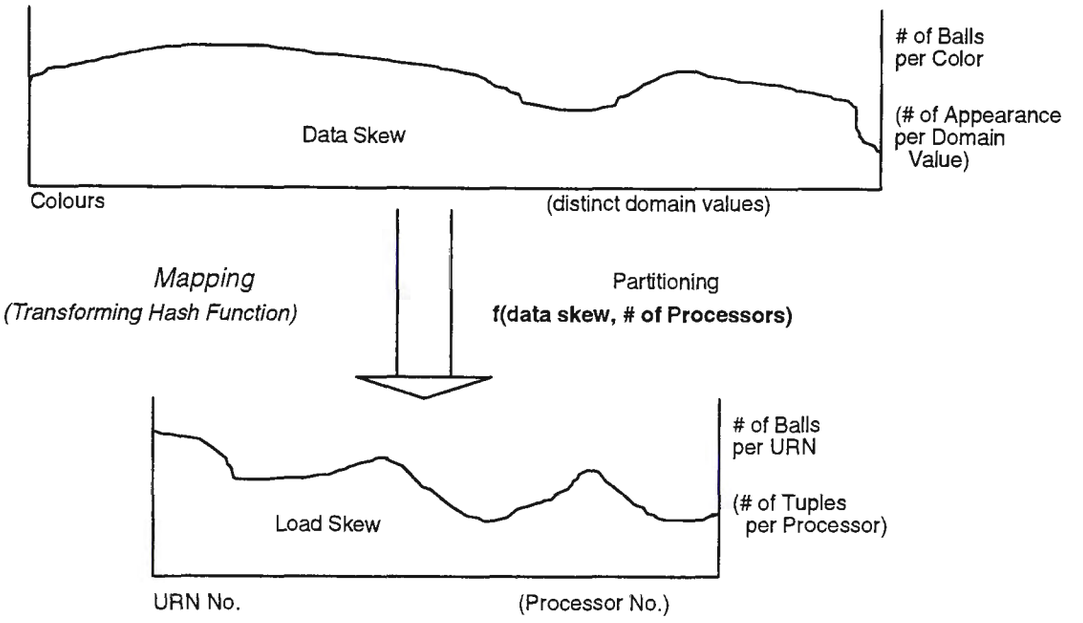


Figure 4.7: Data Skew vs Load Skew

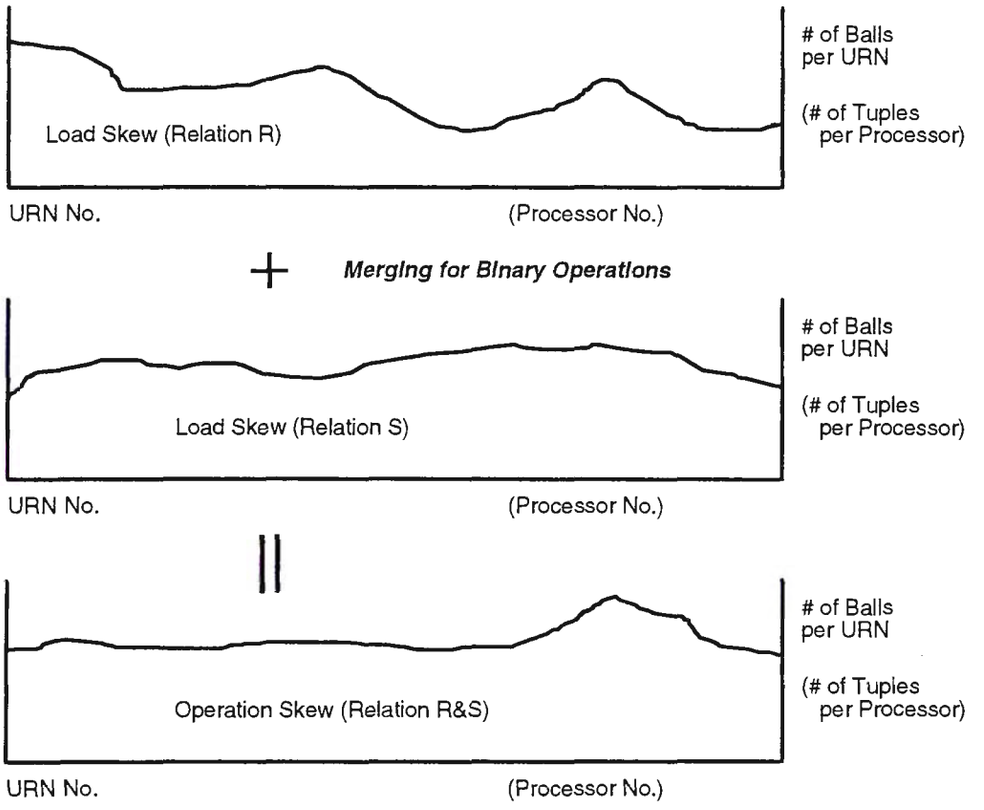


Figure 4.8: Load Skew vs Operation Skew

4.6 Load Skew Foundation Model of Hash Partitioning

Let x be the number of tuples of relation R and k, the number of processors. The aim of the model is to find the average number of tuples in the most heavily loaded processor and in the least heavily loaded processor. With traditional urn model, their limiting distributions of maximal and minimal occupancy of boxes are based on asymptotic methods of probability theory. The only assumption made is that we are allocating x balls to k urns randomly and independently, with the probability of any one ball getting into a given urn equal to $1/k$. Therefore, supposing tuples are randomly allocated (e.g. hashed) to each available processor with the same probability $1/k$, then the joint distribution of the occupancy numbers $x_1,...,x_k$ across all processors (i.e. x_i gives the number of tuples in processor i) follows a multinomial distribution, and we need to find

$$L = \max(x_1, \ldots, x_k) \quad \text{and} \quad L^* = \min(x_1, \ldots, x_k),$$

the largest and the smallest multinomial occupancy numbers respectively in this situation. From [John60], it is shown that, for large x, the standardised variables

$$w_i = \frac{x_i - \frac{x}{k}}{\sqrt{x\left[\frac{1}{k}\left(1-\frac{1}{k}\right)\right]}} = \frac{kx_i - x}{\sqrt{(k-1)x}} \qquad i = 1,2,\ldots,k\,,$$

follow a multivariate normal distribution, with zero mean, unit variances, and each covariance equal to $\frac{-1}{(k-1)}$.

We denote $\max(w_1, \ldots, w_k)$ by L_k and $\min(w_1, \ldots, w_k)$ by L_k^*. In [Gala87], it is shown that the maximum extreme value limiting distribution function for

$$L_k' = \frac{L_k - a_k}{b_k},$$

where

$$a_k = \frac{1}{b_k} - \frac{b_k}{2}(\ln\ln k + \ln 4\pi), \quad b_k = \frac{1}{\sqrt{2\ln k}}, \tag{4-0}$$

is given by

$$H(t) = \exp\ (-e^{-t}),$$

which is the Gumbel distribution with mean equalled to Euler's Constant $\gamma = 0.577216$, and variance of $\pi^2 / 6$. Again form [Gala87], the minimum extreme value limiting distribution function for

$$L_k^{'} * = \frac{L_k * - c_k}{d_k},$$

where $c_k = -a_k$ and $d_k = b_k$, is given by

$$H*(t) = 1 - \exp(-e^t).$$

In order to measure load skew, the maximum number of tuples in an urn is often used, i.e. by means of an extreme value. There are three types of extreme value distribution in order statistics and namely, the Frechet Type, Weibull Type, and Gumbel Type [Gala87]. Among them together with their corresponding preconditions, the Gumbel (third) type distribution is adopted in the skew model.

4.6.1 Mean of Maximum and Minimum Load

a. Mean of Maximum Load

From the properties of the Gumbel distribution [Gala87], we have

$$E(L_k^{'}) \approx \frac{E(L_k) - a_k}{b_k} \approx \gamma,$$

$$E(L_k) \approx \gamma \times b_k + a_k$$

In this particular situation, the parameters a_k and b_k are given in (4-0). Thus, we get, on simplification,

$$E(L_k) \approx \frac{\gamma}{\sqrt{2\ln k}} + \sqrt{2\ln k} - \frac{1}{2\sqrt{2\ln k}}(\ln\ln k + \ln 4\pi)$$

$$\approx \frac{1}{\sqrt{2\ln k}}\left(2\ln k - \frac{\ln\ln k}{2} - 0.69\right) .$$

Thus,

$$E(L) \approx \frac{x}{k} + \frac{E(L_k)\sqrt{x(k-1)}}{k}$$

$$\approx \frac{x}{k}\left[1 + \sqrt{\frac{k-1}{2x\ln k}}\left(2\ln k - \frac{\ln\ln k}{2} - 0.69\right)\right].$$

After the above transformation and simplification, we obtain for the maximum average load

$$\overline{L} \approx \frac{x}{k}\left[1 + \sqrt{\frac{k-1}{2x\ln k}}\left(2\ln k - \frac{\ln\ln k}{2} - 0.69\right)\right] . \qquad (4\text{-}1)$$

b. Mean of Minimum Load

From the properties of the Gumbel distribution [Gala87], we have

$$E(L_k^{'}*) \approx \frac{E(L_k*) - c_k}{d_k} \approx -\gamma ,$$

$$E(L_k*) \approx -\gamma \times b_k - a_k$$

Again from (4-0) and on simplification, we have

$$E(L_K *) \approx -\frac{\gamma}{\sqrt{2\ln k}} - \sqrt{2\ln k} + \frac{1}{2\sqrt{2\ln k}}(\ln\ln k + \ln 4\pi)$$

$$\approx \frac{1}{\sqrt{2\ln k}}\left(\frac{\ln\ln k}{2} + 0.69 - 2\ln k\right).$$

Thus,

$$E(L*) \approx \frac{x}{k} + \frac{E(L_k *)\sqrt{x(k-1)}}{k}$$

$$\approx \frac{x}{k}\left[1 - \sqrt{\frac{k-1}{2x\ln k}}\left(2\ln k - \frac{\ln\ln k}{2} - 0.69\right)\right].$$

After the above transformation and simplification, the minimum average load is given by

$$\bar{L}* \approx \frac{x}{k}\left[1 - \sqrt{\frac{k-1}{2x\ln k}}\left(2\ln k - \frac{\ln\ln k}{2} - 0.69\right)\right] . \qquad (4\text{-}2)$$

By introducing a *load skew factor* φ ,

$$\varphi = \sqrt{\frac{k-1}{2x\ln k}}\left(2\ln k - \frac{\ln\ln k}{2} - 0.69\right) \qquad (4\text{-}3)$$

equations (4-1) and (4-2) can be rewritten as

$$\bar{L} = \frac{x}{k}(1+\varphi) \qquad (4\text{-}4)$$

and

$$\overline{L}^* = \frac{x}{k}(1-\varphi). \tag{4-5}$$

Figure 4.9 shows the 3D representation of maximum average load in varying parameters k and x. With a small number of processors, maximum average load drops sharply but the curve in the figure becomes smooth when the number of processors is further increased. Moreover, in heavy workload, having more processors means reduced maximum load, whereas with light workload, maximum load is kept to a minimal level. The 3D representation of minimum average load is shown in Figure 4.10. In general, the trend is close to that of maximum average load. The load reduction is significant when k is small, but becomes marginal when k is large. A relatively light workload is insensitive to the increase of the number of processors. Finally, in both figures, we can see that a large number of processors do not reduce the maximum average load or increase the minimum average load significantly.

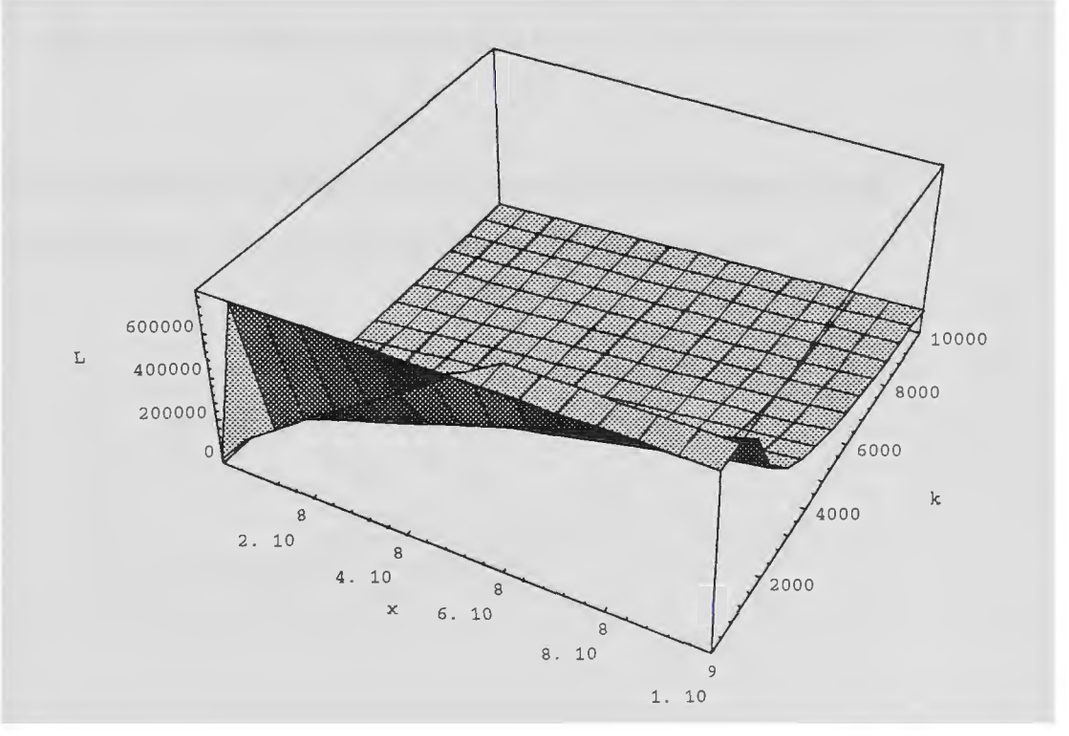


Figure 4.9: Maximum Avg Load when k is in $\left[2,\ 10^5\right]$ and x is in $\left[10^5,\ 10^9\right]$

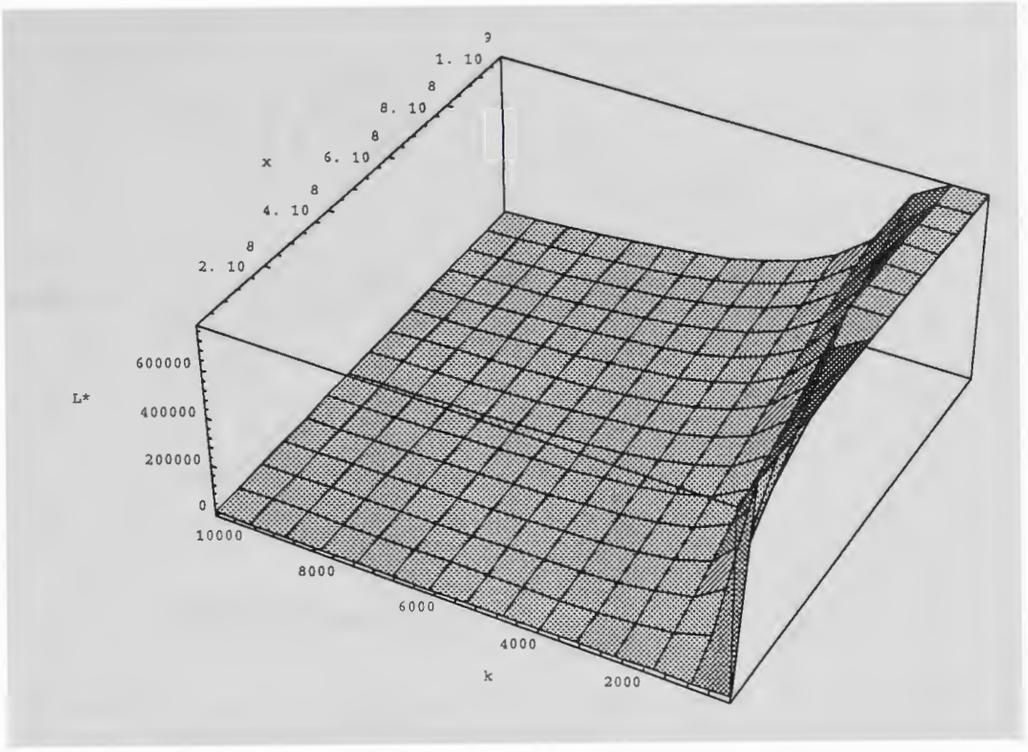


Figure 4.10: Minimum Avg Load when k is in $\left[2,\ 10^5\right]$ and x is in $\left[10^5,\ 10^9\right]$

4.6.2 Standard Deviation of Maximum and Minimum Load

From the properties of the Gumbel distribution [Gala87], we have

$$Var\left(L_k^{'}\right) \approx \frac{1}{b_k^{\,2}} \times Var\left(L_k\right) \approx \frac{\pi^2}{6} \quad ,$$

$$Var\left(L_k\right) \approx \frac{\pi^2 \times b_k^{\,2}}{6} \approx \frac{\pi^2}{12 \ln k} \quad .$$

Using (4-0), we obtain

$$Var(L) \approx \frac{x(k-1)}{k^2} \times Var\left(L_k\right)$$

$$\approx \frac{x(k-1)}{k^2} \times \frac{\pi^2}{12 \ln k}$$

$$\approx \frac{\pi^2 x(k-1)}{12k^2 \ln k},$$

$$\sigma_L \approx \frac{\pi}{k}\sqrt{\frac{x(k-1)}{12\ln k}}.$$

In addition,

$$Var(L_k^{'}*) \approx \frac{1}{b_k^{\,2}} \times Var(L_k *) \approx \frac{\pi^2}{6},$$

$$Var(L_k *) == \frac{\pi^2 \times b_k^{\,2}}{6} \approx \frac{\pi^2}{12\ln k},$$

$$Var(L*) \approx \frac{x(k-1)}{k^2} \times Var(L_k *)$$

$$\approx \frac{x(k-1)}{k^2} \times \frac{\pi^2}{12\ln k}$$

$$\approx \frac{\pi^2 x(k-1)}{12k^2 \ln k},$$

$$\sigma_{L^*} \approx \frac{\pi}{k}\sqrt{\frac{x(k-1)}{12\ln k}}.$$

Therefore, the standard deviations of L and L^* may both be approximated by

$$\frac{\pi}{k}\sqrt{\frac{x(k-1)}{12\ln k}}. \qquad (4\text{-}6)$$

The graph representations of standard deviations of L and L^* are shown in Figure 4.11 where parameters k and x are in the same range as their mean load values. In the figure, the standard deviations are large when the workload is heavy and the number of processors is small. Increasing the number of processors reduces the standard deviations.

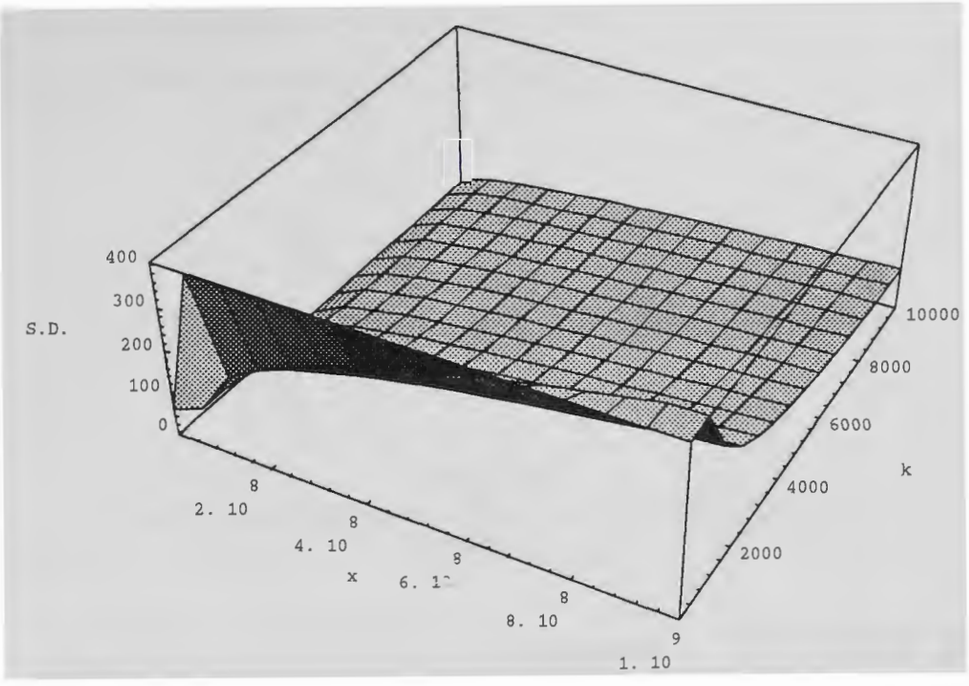


Figure 4.11: Standard Deviations when k is in $\left[2,\ 10^5\right]$ and x is in $\left[10^5,\ 10^9\right]$

4.6.3 Distribution Function of Maximum and Minimum Load

a. Distribution Function of Maximum Load

Here,

$$L_k' = \frac{L_k - a_k}{b_k} \quad \text{and} \quad L_k = \left(L_k' + \frac{a_k}{b_k}\right) b_k \ ,$$

so that from [Gala87],

$$\Pr(L \le t) \approx \Pr\left[\left(\frac{L_k}{k}\sqrt{x(k-1)} + \frac{x}{k}\right) \le t\right] \approx \Pr\left[\left(\frac{b_k L_k' + a_k}{k}\sqrt{x(k-1)} + \frac{x}{k}\right) \le t\right]$$

$$\approx \Pr\left\{\left[\frac{k\left(t-\frac{x}{k}\right)}{\sqrt{x(k-1)}}-a_k\right] / b_k \geq L_k^{'}\right\}$$

$$\approx \Pr\left\{L_k^{'} \leq \left[\frac{k\left(t-\frac{x}{k}\right)\sqrt{2\ln k}}{\sqrt{x(k-1)}}-2\ln k+\frac{\ln\ln k}{2}+\frac{\ln 4\pi}{2}\right]\right\}$$

$$\approx \Pr\left\{L_k^{'} \leq \left((kt-x)\sqrt{\frac{2\ln k}{x(k-1)}}-2\ln k+\frac{\ln(4\pi\ln k)}{2}\right)\right\}$$

$$\Rightarrow \Pr(L \leq t) \approx \exp\left\{-\exp\left[2\ln k-\frac{\ln(4\pi\ln k)}{2}-(kt-x)\sqrt{\frac{2\ln k}{x(k-1)}}\right]\right\}.$$

Therefore, the maximum load distribution function may be approximated by

$$\Pr(L \leq t) \approx \exp\left\{-\exp\left[2\ln k-\frac{\ln(4\pi\ln k)}{2}-(kt-x)\sqrt{\frac{2\ln k}{x(k-1)}}\right]\right\} \quad , \text{(4-7)}$$

from which the percentiles may be determined.

b. Distribution Function of Minimum Load

Similar to the above, we have

$$L_k^{*} = \left(L_k^{'}{}^{*}+\frac{c_k}{d_k}\right)d_k,$$

so that,

$$\Pr(L^* \geq t) \approx \Pr\left[\left(\frac{L_k^*}{k}\sqrt{x(k-1)} + \frac{x}{k}\right) \geq t\right]$$

$$\approx \Pr\left[\left(\frac{b_k L_k^{'} * - a_k}{k}\sqrt{x(k-1)} + \frac{x}{k}\right) \geq t\right]$$

$$\approx \Pr\left\{\left[\frac{k\left(t - \frac{x}{k}\right)}{\sqrt{x(k-1)}} + a_k\right] / b_k \leq L_k^{'} *\right\}$$

$$\approx \Pr\left\{L_k^{'} * \geq \left[\frac{k\left(t - \frac{x}{k}\right)\sqrt{2\ln k}}{\sqrt{x(k-1)}} + 2\ln k - \frac{\ln\ln k}{2} - \frac{\ln 4\pi}{2}\right]\right\}$$

$$\approx \Pr\left\{L_k^{'} * \geq \left((kt - x)\sqrt{\frac{2\ln k}{x(k-1)}} + 2\ln k - \frac{\ln(4\pi\ln k)}{2}\right)\right\} .$$

Therefore,

$$\Pr(L^* \geq t) \approx 1 - \exp\left\{-\exp\left[2\ln k - \frac{\ln(4\pi\ln k)}{2} + (kt - x)\sqrt{\frac{2\ln k}{x(k-1)}}\right]\right\},$$

$$\Pr(L^* \leq t) \approx \exp\left\{-\exp\left[2\ln k - \frac{\ln(4\pi\ln k)}{2} + (kt - x)\sqrt{\frac{2\ln k}{x(k-1)}}\right]\right\}.$$

Thus, the minimum load distribution function is given by

$$\Pr(L^* \leq t) \approx \exp\left\{-\exp\left[2\ln k - \frac{\ln(4\pi\ln k)}{2} + (kt - x)\sqrt{\frac{2\ln k}{x(k-1)}}\right]\right\} . \quad (4\text{-}8)$$

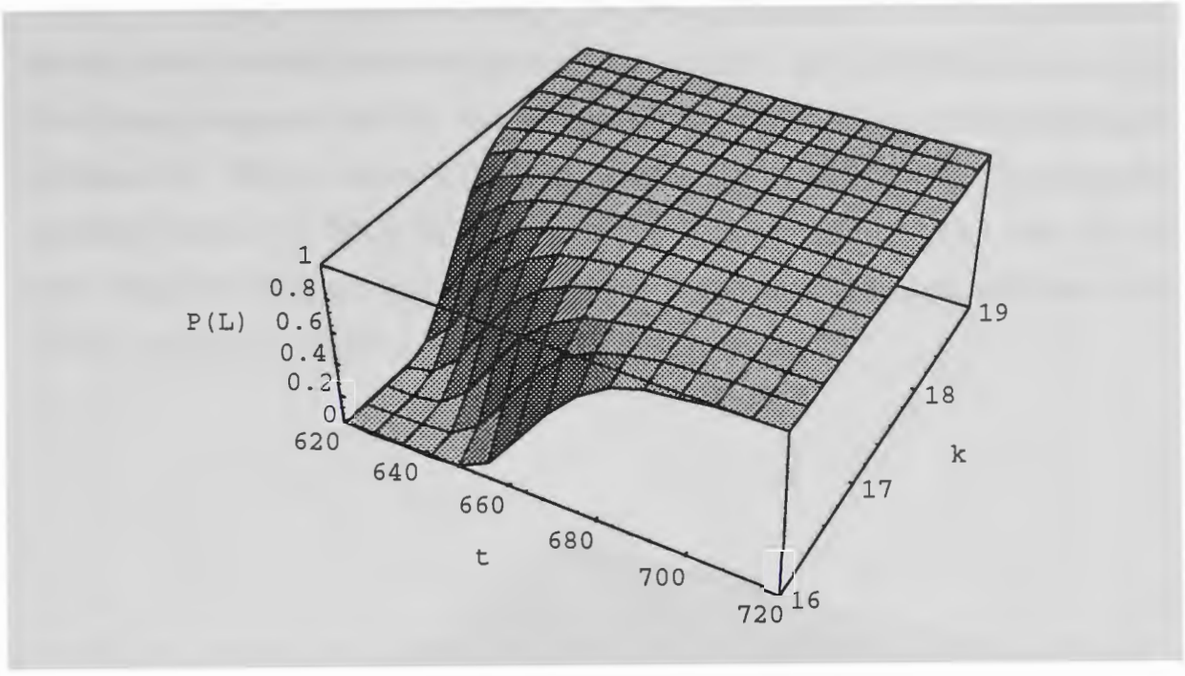


Figure 4.12: Maximum Load Distribution when k is in $[16, 19]$, t is in $[600, 720]$ and x=10000

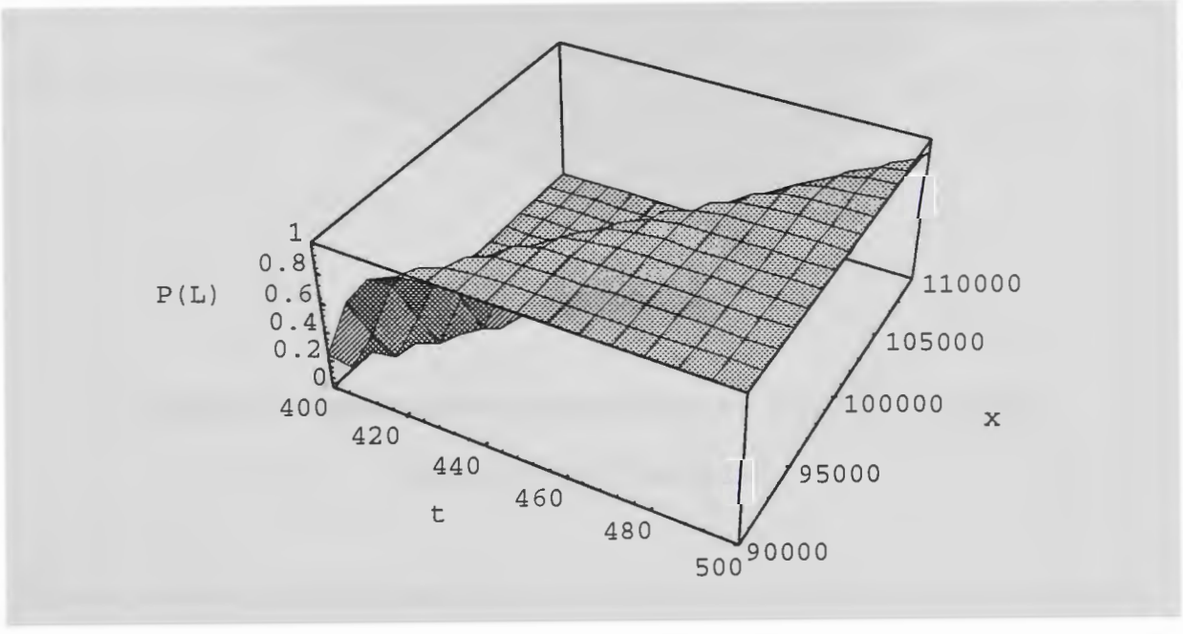


Figure 4.13: Maximum Load Distribution when x is in $[90000, 110000]$, t is in $[400, 500]$ and k=256

Setting the value of x and varying the value of k in a short range, the cumulative distribution function is displayed in Figure 4.12. With 17 processors, it is almost certain that the maximum average load is less than 700 tuples. When the maximum load is 660 tuples, the

chance of it being the maximum load is 10% with 16 processors, and the probability steadily grows when the number of processors is increased. Again for the maximum load distribution, setting the value of k and varying the value of x and t give the result displayed in Figure 4.13. With a x value of 90000, the probability of a t value of 400 tuples being the maximum load is 20%. When the workload is raised, the probability of the t value of 400 tuples being the maximum load is decreased. In other words, increasing workloads will raise the maximum average load with a fixed number of processors.

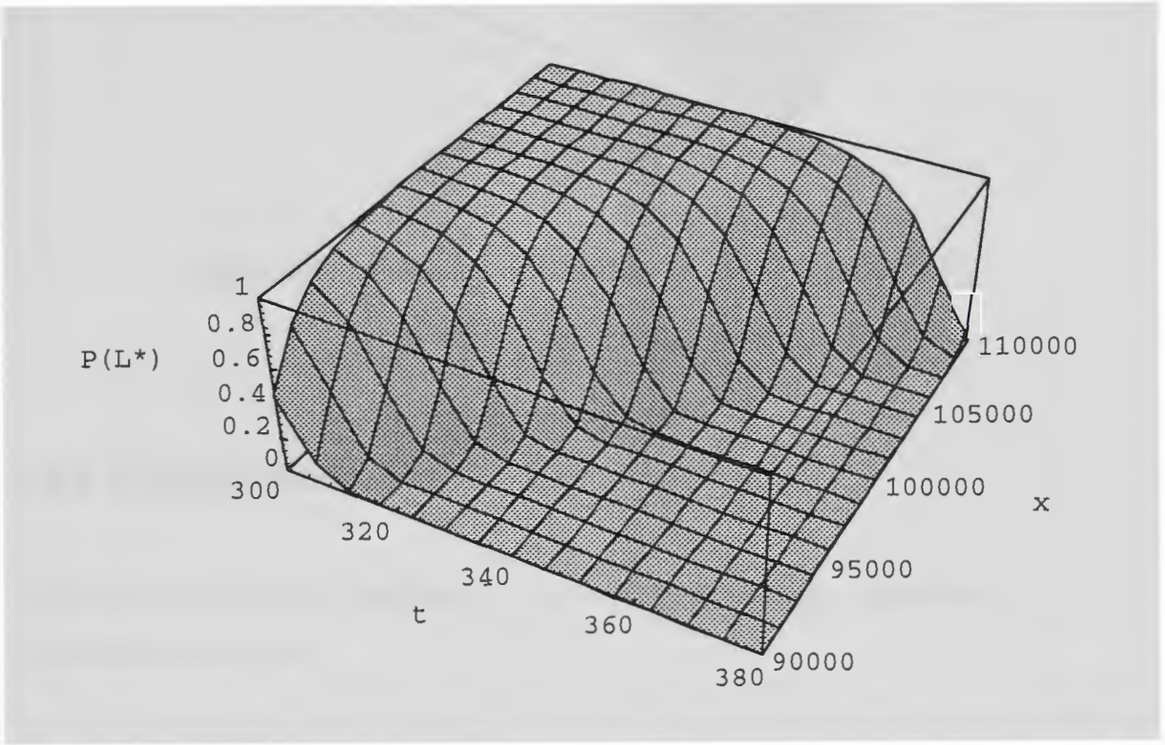


Figure 4.14: Minimum Load Distribution when x is in $[90000, 110000]$, t is in $[300, 380]$ and k=256

Figure 4.14 shows the 3D representation of the minimum load distribution when the number of processors is set to 256. When the value of t is increased, the likelihood of it being the minimum load is decreased. When the value of x is 100000, it is sure that the minimum load is less than 300 tuples, reducing workloads to 90000 will lessen the chance of the 300 tuples being the minimum load. Figure 4.15 shows the 3D representation of the minimum load distribution when the workload is set to 10000 tuples. With 16 processors, increasing the t value will lower the chance of it being the minimum load, and increasing the number of processors will reduce the chance of the t value being the minimum load.

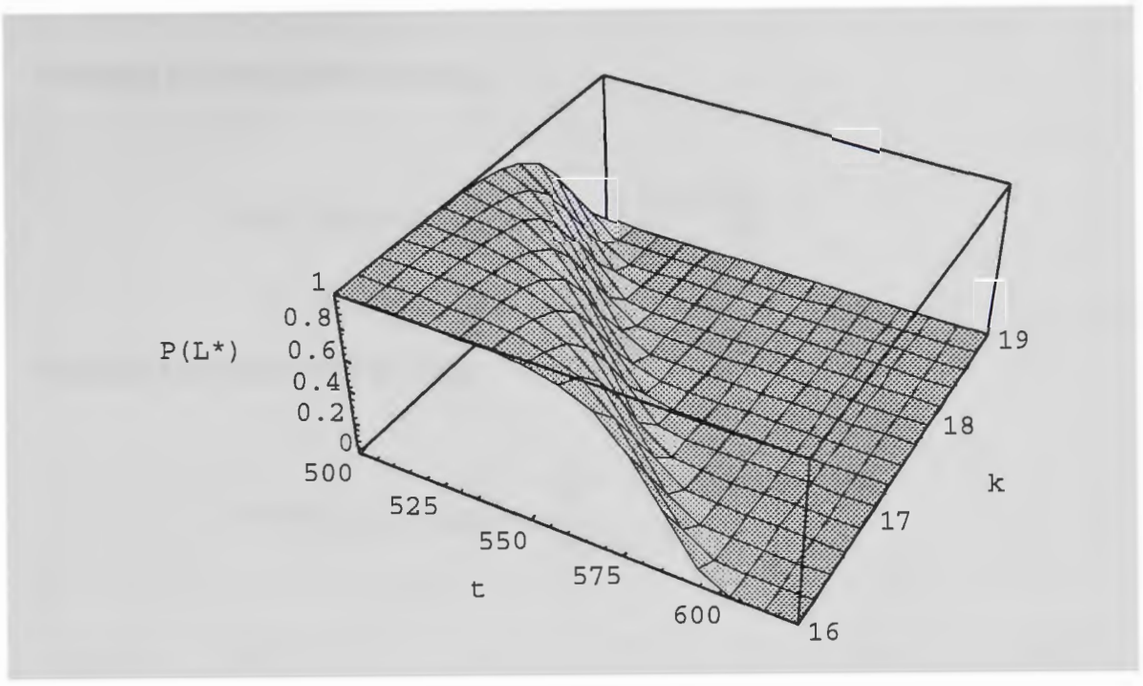


Figure 4.15: Minimum Load Distribution when k is in $[16, 19]$, t is in $[500, 620]$ and x=10000

4.6.4 Model Simplification

For k>>1, we obtain from equations (4-1) - (4-8) the following approximations, maximum average load:

$$\overline{L} \approx \frac{x}{k}\left[1+\sqrt{\frac{2k\ln k}{x}}\right],$$

minimum average load:

$$\overline{L}^{*} \approx \frac{x}{k}\left[1-\sqrt{\frac{2k\ln k}{x}}\right],$$

standard deviation of maximum and minimum average load:

$$\pi\sqrt{\frac{x}{12k\ln k}},$$

maximum load distribution function:

$$\Pr(L \le t) \approx \exp\left\{-\exp\left[\left(\frac{x}{k} - t\right)\sqrt{\frac{2k \ln k}{x}}\right]\right\} ,$$

minimum load distribution function:

$$\Pr(L^* \ge t) \approx 1 - \exp\left\{-\exp\left[\left(t - \frac{x}{k}\right)\sqrt{\frac{2k \ln k}{x}}\right]\right\} .$$

4.7 Operation Skew Foundation Model of Hash Partitioning

Recalling in Figure 3.5, operation skew due to hash partitioning is based on load skew, the relational operation and its processing method. With a unary operation, the operation skew is the same as load skew because there is only one operand relation. With a binary operation, operation skew is the combined effects of load skew with two input relations since the load skew in one relation may not match the load skew in another relation. In other words, load skew in both operand relations may reinforce or cancel out each another. The operation processing method also has influence on operation skew since it determines the local processing complexity.

With operation skew in binary operations, we concentrate on the join operation. Being one of the most expensive and widely used relational algebra operations, join combines two relations with a common domain into a single relation. Looking for efficient join algorithm has been an active research area for decades, hash based join, sort merge join and nested loops join are generally considered as common join algorithms. Accurately modelling the parallel join is always desirable because it not only predicts execution time precisely but also provides more effective and efficient parallel query processing.

Skew modelling focuses on load imbalance over multiple processors after data partitioning, and therefore to emphasise and isolate the skew problem with various fragments sizes, only

the number of tuples after partitioning is considered. It is recognised that memory, I/O, and communication contentions also contribute to the total time, but the skew model is independent of the off-line operations such as the operating system, the data placement and the hardware platform. It only depends on on-line operations such as the degree of data skew in the incoming queries and the data partitioning strategies. In this section, we introduce an operation skew factor, compares it to the load skew factor of the last section, and then integrate the factor into three common parallel join processing methods.

With the binary join operation, the same distribution procedure is carried out for both operand relations. Hence, after partitioning the two relations, the final result is only affected by the total number of tuples allocated to the processor irrespective of their owner (belonging to either relation R or S), so we can derive the following *operation skew factor* ψ by replacing x with $(x+y)$ of equation (4-3).

$$\psi = \sqrt{\frac{k-1}{2(x+y)\ln k}}\left(2\ln k - \frac{\ln\ln k}{2} - 0.69\right) \tag{4-9}$$

Hash based join, sort merge join and nested loops join are three common join methods and their processing complexity are well known in uniprocessor system. Taking into account of skewness, the complexity of these three methods are given below.

4.7.1 Parallel Hash Based Join

The hash based join is the most widely used join algorithms. With multiple processors, two relations are partitioned with the same hash function, and at each processor, one fragment is used to build the hash table and the other fragment is applied to probe the table. If there is a match, output the result. In such a simple parallel hash-based join, the maximum and minimum load over k processors are given as

$$L = \frac{x+y}{k}(1+\psi) \tag{4-10}$$

and

$$L^* = \frac{x+y}{k}(1-\psi). \tag{4-11}$$

4.7.2 Parallel Nested Loops Join

One of the simplest join methods is the nested loops join. For every tuple in relation R (outer relation) the entire relation S (inner relation) is scanned and the matches are found. The advantages of the method are its simplicity and that it requires no extra space. In parallel nested loops join, both operand relations are partitioned with the same hash function. At each processor, for every tuple of one fragment, all tuples of another fragment are compared and matching results output. Therefore, the maximum and minimum load over k processors are given respectively as

$$L = \frac{x \times y}{k^2}(1+\psi)^2 \tag{4-12}$$

and

$$L^* = \frac{x \times y}{k^2}(1-\psi)^2. \tag{4-13}$$

4.7.3 Parallel Sort Merge Join

The sort merge join consists of two phases, the sort phase and the merge phase. In the first phase, the relations are sorted on the join key. Then, the input relations are merged in the order of join key. The parallel version of sort merge join has three phases, sort, transfer, and merge, namely. After the relations are sorted, the fragments of the relations are transferred to various processors in the transfer phase. Finally, each processor works independently on the sorted range tuples. The tuples are merged in the individual processor. The sort merge join is sensitive to the initial order of its input since the ordered input relations omit the sorting phase of the algorithm. In addition, sort merge join uses extra space proportional to the size of the input relations. Again here, we only consider the complexity of operation skew so that we do not include the time for sorting and data hashing. Therefore, we have

$$L = \frac{x(1+\psi)}{k} \times \log_2\left(\frac{y(1+\psi)}{k}\right) \tag{4-14}$$

and

$$L^* = \frac{x(1-\psi)}{k} \times \log_2\left(\frac{y(1-\psi)}{k}\right). \tag{4-15}$$

4.8 Summary

In this chapter we presented a comparison of common partitioning methods emphasising their strengths and weaknesses. Two main partitioning methods are used in parallel processing, namely, range and hash partitioning. Skewness analysis of range partitioning is introduced using Unimodel, Multimodel, and Erlang distributions. With a hash function, the load skew resulted depends on the data skew in the base relations and the hash function. We provided the skew prediction foundation model in the absence of data skew using the extreme value distribution theory of order statistics. Both maximum and minimum average load values are provided based on a complete analytical skew model. Standard deviations of the mean load and the extreme values distributions are also provided for the load skew. In addition, an operation skew foundation model is developed for the join operation.

CHAPTER 5

SKEW PREDICTION OF HASH PARTITIONING WITH DATA SKEW

5.1 Introduction

In the last chapter, we assume that before hash partitioning the data values are inherently uniformly distributed and present a skew foundation model in Section 4.6 which gives the accurate prediction in the absence of data skew. The problems remain are that what

happen if there is data skew, and going a bit further what is the relationship between data skew and load and operation skew. These are the main tasks of this chapter. Section 5.2 discusses the skew problem with data skew and its relationship with the mathematical urn model. The load skew extension model is presented in Section 5.3 and operation skew extension is provided in Section 5.4. All the models are evaluated on the Terabyte Database Simulation Model and the results are reported in Section 5.5. We offer an analysis on data skew, load skew and operation skew in Section 5.6. Finally, we give a discussion of the skew model usage and concluding remarks on skew modelling in Section 5.7.

5.2 Problem Description and Urn Model of Hash Partitioning with Data Skew

If data skew exists, a given attribute value of a relation may appear more frequently than other values, and this might result in an uneven number of tuples spread across processors after partitioning. Therefore, the situation can be described as a distribution with bias. Using the telephone directory example, generally, there are more entries starting with letter "c" and "d" than that starting with letter "y" and "z". Again in the context of the urn model [John77], urns correspond to different classes with different probabilities, and the balls are likely to go to the most favoured urn. In other words, the maximum load potentially happens in the most favoured urn and minimum load occurs in the least favoured urn. When all the urn's classes have the same probability $1/k$ of receiving balls, we obtain the situation studied in the last chapter, i.e. there is no data skew.

Figure 5.1 shows the phenomenon represented by the urn models in the presence of data skew. There are four urns (processors) with different capacities and the third urn is the most favoured urn. The balls (tuples) are treated equally as the single distribution union but the distribution rule is biased in the sense that urns are associated with different receiving probabilities as shown in the figure: 10%, 30%, 50%, and 20%. Certainly, the final distribution result will reveal that number of balls in each urn is proportional to the urn probability.

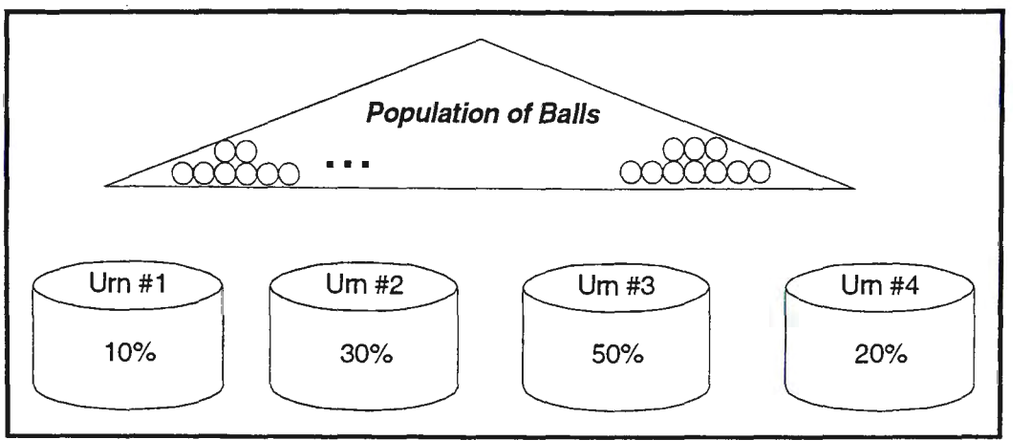


Figure 5.1: Urn Models with Different Probabilities

5.3 Load Skew Extension Model

In this section, we shall examine the influence of data skew on the load skew. Supposing the processors are now chosen with non-uniform probabilities $p_1,\ldots,p_k$ caused by data skew, then letting $p_{\max} = \max(p_1,\ldots,p_k)$, and replacing $1/k$ by $p_{\max}$ in equation (4-1), (4-6), and (4-7), the maximum load skew now becomes

$$\overline{L} \approx xp_{\max}\left[1+\sqrt{\frac{(1/p_{\max}-1)}{2x\ln(1/p_{\max})}}\left(2\ln(1/p_{\max})-\frac{\ln\ln(1/p_{\max})}{2}-0.69\right)\right], \quad (5\text{-}1)$$

with standard deviation

$$\sigma_L \approx p_{\max}\pi\sqrt{\frac{x(1/p_{\max}-1)}{12\ln(1/p_{\max})}}, \quad (5\text{-}2)$$

and distribution function given by

$$\Pr(L\le t)\approx\exp\left\{-\exp\left[2\ln(1/p_{\max})-\frac{\ln(4\pi\ln(1/p_{\max}))}{2}-\left(\frac{t}{p_{\max}}-x\right)\sqrt{\frac{2\ln(1/p_{\max})}{x(1/p_{\max}-1)}}\right]\right\}. \quad (5\text{-}3)$$

Likewise, we have $p_{\min} = \min(p_1,\ldots,p_k)$ and replacing $1/k$ by $p_{\min}$ in equation (4-2), (4-6), and (4-8), the minimum value may be obtained

$$\overline{L}^* \approx xp_{\min}\left[1-\sqrt{\frac{(1/p_{\min}-1)}{2x\ln(1/p_{\min})}}\left(2\ln(1/p_{\min})-\frac{\ln\ln(1/p_{\min})}{2}-0.69\right)\right], \quad (5\text{-}4)$$

with standard deviation

$$\sigma_{L^*} \approx p_{\min}\pi\sqrt{\frac{x(1/p_{\min}-1)}{12\ln(1/p_{\min})}}, \quad (5\text{-}5)$$

and distribution function given by

$$\Pr(L^* \le t) \approx \exp\left\{-\exp\left[2\ln(1/p_{\min})-\frac{\ln(4\pi\ln(1/p_{\min}))}{2}+\left(\frac{t}{p_{\min}}-x\right)\sqrt{\frac{2\ln(1/p_{\min})}{x(1/p_{\min}-1)}}\right]\right\}. \quad (5\text{-}6)$$

Although the distribution function and the standard deviations are useful, we shall primarily focus on the mean maximum and minimum load in the rest of the thesis as these provide the simplest yet reliable summary of skew behaviour.

5.3.1 Representing Data Skew with Zipf Distribution

In many textual or bibliographic databases, the data distribution tends to follow the Zipf law [Zipf49] which has been found to give a reasonable approximation to the number of appearances of each domain value of the relation. The Zipf distribution has been widely used in skew handling algorithms [Lync88, Kell91, Wolf93a, Wolf93b]. Here, we take the probabilities of processors receiving tuples as following a Zipf distribution and, to make it more general, we introduce a data skew factor θ in the probability mass function as follows

$$P_i = \frac{1}{i^\theta \times \sum_{j=1}^{k}\frac{1}{j^\theta}}, \qquad 1 \le i \le k, \quad (5\text{-}7)$$

where $0 \le \theta \le 1$ and i is the ith processor. When $\theta = 0$, it yields a discrete uniform

distribution (i.e. no data skew), thus subsuming the representation in Chapter 4 as a special case as shown in Figure 5.2. One of the most commonly used distributions for describing skew is the pure Zipf distribution ($\theta = 1$) where the ith common word in natural language text seems to occur with a frequency proportional to i [Knut73, Zipf49] and its probability mass function is

$$P_i = \frac{1}{i \times \sum_{j=1}^{k} \frac{1}{j}} = \frac{1}{i \times H_k} \approx \frac{1}{i \times (\gamma + \ln k)}, \tag{5-8}$$

where $\gamma = 0.577216$ and H_k are respectively the Euler's constant and the kth Harmonic Number [Knut73]. Figure 5.3 shows the probability density of Zipf distribution with 10 processors with the light coloured columns indicating the maximum and the minimum probability.

In Figure 5.3, the maximum probability is

$$p_{\max} = p_1 \approx \frac{1}{H_k} \approx \frac{1}{\gamma + \ln k} \tag{5-9}$$

and the minimum probability is

$$p_{\min} = p_k \approx \frac{1}{k \times H_k} \approx \frac{1}{k(\gamma + \ln k)}. \tag{5-10}$$

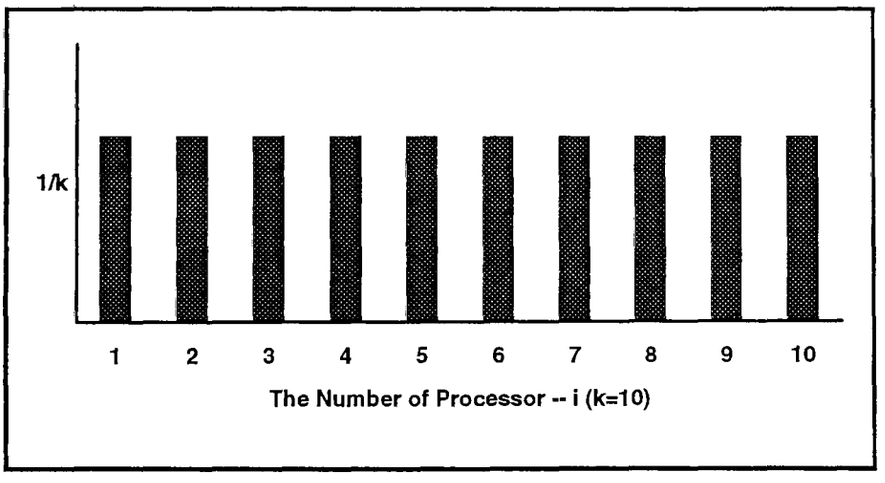


Figure 5.2: Density of Discrete Uniform Distribution with 10 Processors

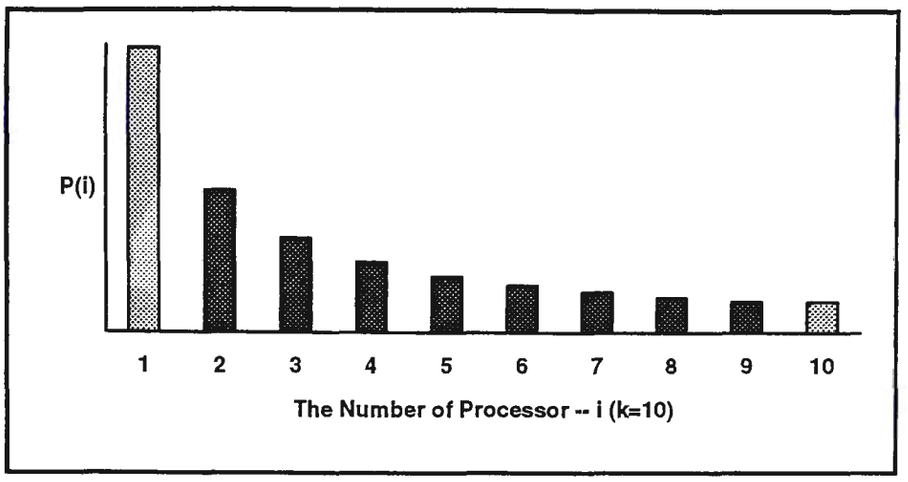


Figure 5.3: Density of Zipf Distribution with 10 Processors

Thus, from equations (5-1) and (5-4) the maximum and the minimum load skew for the Zipf law are:

$$\bar{L} \approx \frac{x}{\gamma + \ln k}\left[1 + \sqrt{\frac{(\gamma + \ln k - 1)}{2x\ln(\gamma + \ln k)}}\left(2\ln(\gamma + \ln k) - \frac{\ln\ln(\gamma + \ln k)}{2} - 0.69\right)\right] \qquad (5\text{-}11)$$

and

$$\bar{L}^* = \frac{x}{k(\gamma + \ln k)}\left[1 - \sqrt{\frac{(k(\gamma + \ln k) - 1)}{2x\ln(k(\gamma + \ln k))}}\left(2\ln(k(\gamma + \ln k)) - \frac{\ln\ln(k(\gamma + \ln k))}{2} - 0.69\right)\right] . \qquad (5\text{-}12)$$

5.3.2 Representing Data Skew with Normal Distribution

Given a certain degree of data skew θ, the estimate from the previous section will be shown to be fairly accurate in Section 5.5 and Chapter 10. However, discovering the degree of data skew has been proven to be a difficult task since it involves either expensive sampling or periodic collection of statistics from the data dictionary [Liu95, Pira90]. Moreover, both methods require complex procedures for sampling or updating the data dictionary and to obtain a satisfactory confidence level often results in extremely high cost.

The focus of our study is on very large parallel databases which consist of relations close

to terabyte size and it would seem appropriate to model the data values using the continuous normal distribution, which provides a useful description of many phenomena. In addition, the normal distribution is the most useful distribution in modelling in general because of the following reasons [Gray80].

- The sum of n independently and identically distributed random variables tends to be normally distributed as n tends to infinity. This result is the central limit theorem.

- If the random variable is distributed binomially, then as n tends toward infinity, the random variable tends to be normally distributed with mean $\mu = nq$, and variance $\sigma^2 = npq$.

- If the random variable is distributed according to the Poisson distribution with parameter λ, then as λ gets large, the random variable tends to be normally distributed with mean $\mu = \lambda$.

The normal density with mean μ and standard deviation σ, is given by

$$\psi = P(i;\ \mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(i-\mu)^2}{2\sigma^2} \right),$$

where the parameters μ and σ satisfy $-\infty < \mu < +\infty$ and $\sigma > 0$. The distribution is Bell-Shape and symmetrical, and by suitably adjusting μ and σ a variety of skewness characteristics may be captured.

To adapt the normal density function for our purpose, we split it into k intervals where k is the number of processors. Each interval has length $\frac{6\sigma}{k}$, where σ is the standard deviation of the normal distribution, which in total takes up 99.73% of the mass (since $\mu \pm 3\sigma$ accounts for 99.73% of the probabilities). When k is an even number, the boundary values of the largest and smallest probability intervals are

$$\left[\mu-\frac{6\sigma}{k},\ \mu\right] \text{ and } \left[\mu-3\sigma,\ \mu-3\sigma+\frac{6\sigma}{k}\right].$$

Therefore,

$$\Pr\left\{\left(\mu-\frac{6\sigma}{k}\right)<i<\mu\right\}=\Phi\left(\frac{\mu-\mu}{\sigma}\right)-\Phi\left(\frac{\mu-\frac{6\sigma}{k}-\mu}{\sigma}\right)$$

$$=\Phi(0)-\Phi\left(-\frac{6}{k}\right)=\Phi(0)-\left[1-\Phi\left(\frac{6}{k}\right)\right]=\Phi(6/k)-0.5$$

and

$$\Pr\left\{(\mu-3\sigma)<i<\left(\mu-3\sigma+\frac{6\sigma}{k}\right)\right\}$$

$$=\Phi\left(\frac{\mu-3\sigma+\frac{6\sigma}{k}-\mu}{\sigma}\right)-\Phi\left(\frac{\mu-3\sigma-\mu}{\sigma}\right)$$

$$=\Phi\left(-3+\frac{6}{k}\right)-\Phi(-3)=1-\Phi\left(3-\frac{6}{k}\right)-\left[1-\Phi(3)\right]$$

$$=\Phi(3)-\Phi(3-6/k).$$

When k is an odd number, the boundary values of the largest and smallest probability interval are

$$\left[\mu-\frac{6\sigma}{2k},\ \mu+\frac{6\sigma}{2k}\right] \text{ and } \left[\mu-3\sigma,\ \mu-3\sigma+\frac{6\sigma}{k}\right].$$

Therefore,

$$\Pr\left\{\left(\mu-\frac{6\sigma}{2k}\right)<i<\left(\mu+\frac{6\sigma}{2k}\right)\right\}=\Phi\left(\frac{\mu+\frac{6\sigma}{2k}-\mu}{\sigma}\right)-\Phi\left(\frac{\mu-\frac{6\sigma}{2k}-\mu}{\sigma}\right)$$

$$= \Phi\left(\frac{3}{k}\right) - \Phi\left(-\frac{3}{k}\right) = \Phi\left(\frac{3}{k}\right) - \left(1 - \Phi\left(\frac{3}{k}\right)\right)$$
$$= 2\Phi(3/k) - 1$$

and

$$\Pr\left\{(\mu - 3\sigma) < i < \left(\mu - 3\sigma + \frac{6\sigma}{k}\right)\right\}$$
$$= \Phi\left(\frac{\mu - 3\sigma + \frac{6\sigma}{k} - \mu}{\sigma}\right) - \Phi\left(\frac{\mu - 3\sigma - \mu}{\sigma}\right)$$
$$= \Phi\left(-3 + \frac{6}{k}\right) - \Phi(-3) = 1 - \Phi\left(3 - \frac{6}{k}\right) - (1 - \Phi(3))$$
$$= \Phi(3) - \Phi(3 - 6/k).$$

However, the 6σ only covers 99.73% of the mass in normal distribution and the rest covers probabilities beyond 3σ on either side of μ which may be significant when the number of processors k is large, and we assume that this is spread evenly across all processors with $\frac{0.27\%}{k}$ for each processor.

Figure 5.4 shows the discretised mass function of the normal distribution, with the light coloured columns representing the minimum and the maximum probabilities. Whether the number of processors is even or odd affects the maximum load, and their effects are shown in Figure 5.4(a) and Figure 5.4(b).

It is shown that if $\Phi(C)$ is the standard normal distribution function evaluated at C,

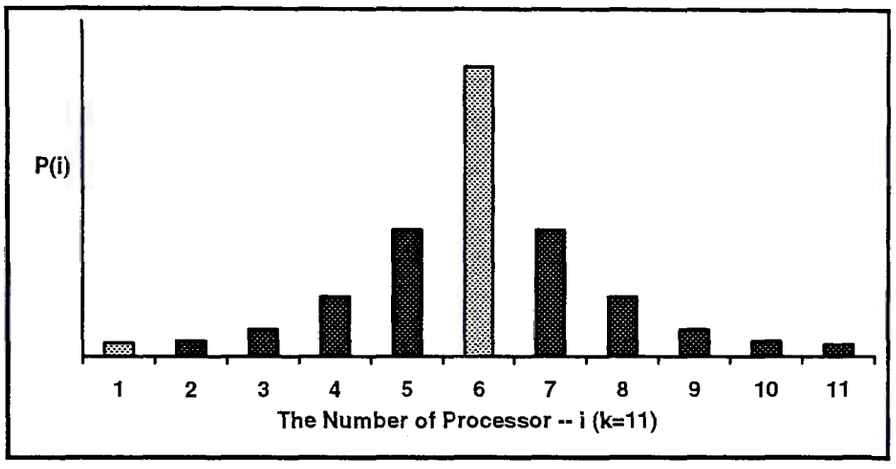
$$p_{\max} \approx \begin{cases} \Phi(6/k) - 0.5 + A & \text{for } k \text{ even} \\ 2\Phi(3/k) - 1 + A & \text{for } k \text{ odd} \end{cases} \quad (5\text{-}13)$$

and

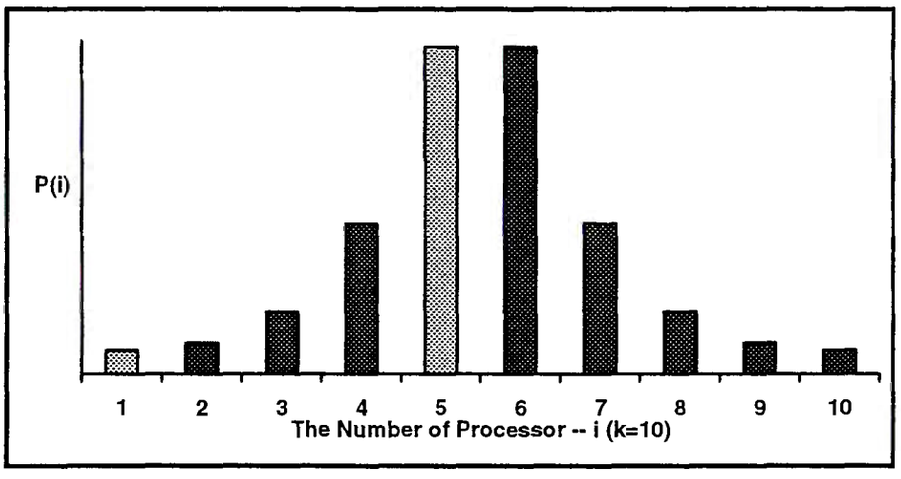
$$p_{\min} \approx 1 - \Phi(3 - 6/k) + A \quad (5\text{-}14)$$

where $A = \frac{0.27\%}{k}$. It is also shown in Appendix A that for all k,

$$\left(2\Phi(3/k)-1+A\right) \geq \left(\Phi(6/k)-0.5+A\right).$$



(a). 11 Processors



(b). 10 Processors

Figure 5.4: Density of Normal Distribution

Thus, we may take

$$p_{max} \approx 2\Phi(3/k)-1+A$$

for all k, with the understanding that this is exact for odd k but represents a slight over-estimation for k even.

We note that neither p_{max} nor p_{min} directly depends on the mean and variance of the underlying normal distribution. The number of processors alone is sufficient to characterise p_{max} and p_{min}. This is so because

- where the peak (mode) of the distribution occurs is unimportant as long as all the processors are identical,

- the spread of the distribution (variance) is already incorporated through the number of processors since nearly all the tuples $(\mu \pm 3\sigma)$ must be allocated to those processors.

Although a closed-form representation for $\Phi(x)$ is not available, an excellent approximation is given by [Borj79]

$$\Phi(x) \approx 1 - \left[\frac{1}{0.661x + 0.339\sqrt{x^2 + 5.51}}\right]\frac{e^{-x^2/2}}{\sqrt{2\pi}}, \qquad x \geq 0. \tag{5-15}$$

Assuming the number of processors is even, from equation (5-1), the maximum load skew is given by

$$\begin{aligned}\overline{L} \approx\ & x \times \left(\Phi(6/k) - 0.5 + A\right) \times \\ & \left\{1 + \left[\left(\left(\Phi(6/k) - 0.5 + A\right)^{-1} - 1\right) / \left(2x\ln\left(\Phi(6/k) - 0.5 + A\right)^{-1}\right)\right]^{1/2}\right. \\ & \left.\left(2\ln\left(\Phi(6/k) - 0.5 + A\right)^{-1} - \ln\ln\left(\Phi(6/k) - 0.5 + A\right)^{-1} / 2 - 0.69\right)\right\}\end{aligned} \tag{5-16}$$

and the minimum value, from equation (5-4), is

$$\begin{aligned}\overline{L}^* \approx\ & x \times \left(\Phi(3) - \Phi(3 - 6/k) + A\right) \times \\ & \left\{1 - \left[\left(\left(\Phi(3) - \Phi(3 - 6/k) + A\right)^{-1} - 1\right) / 2x\ln\left(\Phi(3) - \Phi(3 - 6/k) + A\right)^{-1}\right]^{1/2}\right. \\ & \left.\left(2\ln\left(\Phi(3) - \Phi(3 - 6/k) + A\right)^{-1} - \ln\ln\left(\Phi(3) - \Phi(3 - 6/k) + A\right)^{-1} / 2 - 0.69\right)\right\}\end{aligned} \tag{5-17}$$

For k large, values such as A, 0.69 and 0.5 may be ignored in equations (5-16) and (5-17), and constants 0.661 and 0.339 may be approximated by 2/3 and 1/3 in equation (5-15). By applying these simplifications, we can obtain a closed form result as follows:

$$\begin{aligned}
\bar{L} \approx x \times & \left(1-\left(3e^{-18/k^2} / \left(12/k+\left(36/k^2+5.51\right)^{1/2}\right)\right)/(2\pi)^{1/2}\right) \times \\
& \left\{1+\left[\left(\left(1-\left(3e^{-18/k^2} / \left(12/k+\left(36/k^2+5.51\right)^{1/2}\right)\right)/(2\pi)^{1/2}\right)^{-1}-1\right)\right. \\
& \left./\left(2x\ln\left(1-\left(3e^{-18/k^2} / \left(12/k+\left(36/k^2+5.51\right)^{1/2}\right)\right)/(2\pi)^{1/2}\right)^{-1}\right)\right]^{1/2} \\
& \left[2\ln\left(1-\left(3e^{-18/k^2} / \left(12/k+\left(36/k^2+5.51\right)^{1/2}\right)\right)/(2\pi)^{1/2}\right)^{-1}\right. \\
& \left.\left.-\ln\ln\left(1-\left(3e^{-18/k^2} / \left(12/k+\left(36/k^2+5.51\right)^{1/2}\right)\right)/(2\pi)^{1/2}\right)^{-1}/2\right]\right\}
\end{aligned} \tag{5-18}$$

and

$$\begin{aligned}
\bar{L}^* \approx x \times & \left(\left(3e^{-(3-6/k)^2/2} / \left(2(3-6/k)+\left((3-6/k)^2+5.51\right)^{1/2}\right)\right)/(2\pi)^{1/2}\right) \times \\
& \left\{1-\left[\left(\left(\left(3e^{-(3-6/k)^2/2} / \left(2(3-6/k)+\left((3-6/k)^2+5.51\right)^{1/2}\right)\right)/(2\pi)^{1/2}\right)^{-1}-1\right)\right. \\
& \left./2x\ln\left(\left(3e^{-(3-6/k)^2/2} / \left(2(3-6/k)+\left((3-6/k)^2+5.51\right)^{1/2}\right)\right)/(2\pi)^{1/2}\right)^{-1}\right]^{1/2} \\
& \left(2\ln\left(\left(3e^{-(3-6/k)^2/2} / \left(2(3-6/k)+\left((3-6/k)^2+5.51\right)^{1/2}\right)\right)/(2\pi)^{1/2}\right)^{-1}\right. \\
& \left.\left.-\ln\ln\left(\left(3e^{-(3-6/k)^2/2} / \left(2(3-6/k)+\left((3-6/k)^2+5.51\right)^{1/2}\right)\right)/(2\pi)^{1/2}\right)^{-1}/2\right)\right\}
\end{aligned} \tag{5-19}$$

5.4 Operation Skew Extension Model

With a unary operation, operation skew is the same as load skew, but with a binary operation, operation skew is the combined effects of load skew in the two operand relations. The propagation of skew in multiway join is how the skew has developed after several joins. Accurately predicting the skew propagation will give a more precise query cost model, but it can not be carried out without an operation skew model presented in the thesis. Given a relational operation, if there are data skew in the input relations, it will affect the operation skew in the end. Depending on how many input relations are skewed, i.e. skew dimension, the operation skew can be referred as no data skew, single data skew, and double data skew. Again, we only consider binary join operation skew although the skew model can be easily implemented to other binary relational operations.

The case of operation skew without data skew is discussed in the last chapter. This section shall focus on operation skew with single and double data skew relations. The inter-attribute correlations also have a potentially important effect on skew, performance, and estimation of join sizes [Bell89]. For the operations with single skewed relation, the chance of the maximum load of the skewed relation matching the maximum load of the non skewed relation is $1/k$, whereas in other cases the average load of the non skewed relation and the maximum load of the skewed relation are used to represent the distribution results; for the operations with double skewed relations, assume the skewed partition from one relation always matches the skewed partition from the other relation and thus the distribution results can be represented by maximum load in each relation. The maximum load of both operand relations also provide a bound for worst case scenario and the assumption is also true with an unbiased range partitioning.

5.4.1 Operation Skew with Single Data Skew

Relational operation of one skewed input relation is extremely common in any real relational databases. For an instance, there are two tables, *Supplier* and *Part*, and they have a one-to-many relationship.

Table: *Supplier*

Supplier-No	Supplier-Name	Status	City	Comment

Table: *Part*

Part-No	Part-Name	Colour	Description	Weight	City	Supplier-No

One common binary operation is joining two relations through the Supplier-No attribute. To process it on multiple processors, both relations are partitioned on the Supplier-No attribute. Table *Supplier* can be horizontally fragmented evenly, but Table *Part* is skewed since there may be several parts supplied by the same vendor. In the following subsections on operation skew with single data skew, we assume that Table R with x tuples is always the skewed relation.

a. Nested Loops Join

After data partitioning on both input relations, the nested loops join method is employed at each processor for local join processing. The maximum and minimum time complexity of this join can then be given by

$$\left(L_{x\max}(\theta=1)\times\frac{y}{k}\right)\times\left(\frac{k-1}{k}\right)+\left(L_{x\max}(\theta=1)\times L_{y\max}(\theta=0)\right)\times\frac{1}{k}$$

and

$$\left(L_{x\min}(\theta=1)\times\frac{y}{k}\right)\times\left(\frac{k-1}{k}\right)+\left(L_{x\min}(\theta=1)\times L_{y\min}(\theta=0)\right)\times\frac{1}{k},$$

where L is the load value, x and y are the relational cardinality for R and S, $\theta=0$ indicating a non skewed relation, and $\theta=1$, a skewed relation. Thus, after simplification, we have for the maximum and minimum complexity

$$\left(\frac{L_{x\max}(\theta=1)}{k}\right)\times\left(\frac{y(k-1)}{k}+L_{y\max}(\theta=0)\right)$$

and

$$\left(\frac{L_{x\min}(\theta=1)}{k}\right)\times\left(\frac{y(k-1)}{k}+L_{y\min}(\theta=0)\right).$$

b. Hash Based Join

With hash based local join, one relation R is used to set up the hash table and another relation S is applied to probe the hash table. The matched results are written to the output buffer. The complexity of parallel hash based join can be given by

$$\left(L_{x\max}(\theta=1)+\frac{y}{k}\right)\times\left(\frac{k-1}{k}\right)+\left(L_{x\max}(\theta=1)+L_{y\max}(\theta=0)\right)\times\frac{1}{k}$$

and

$$\left(L_{x\min}(\theta=1)+\frac{y}{k}\right)\times\left(\frac{k-1}{k}\right)+\left(L_{x\min}(\theta=1)+L_{y\min}(\theta=0)\right)\times\frac{1}{k}.$$

c. Sort Merge Join

Assuming both relations are already sorted, the merging takes place right after partitioning. Therefore, the complexity fully depends on the number of tuples allocated to each processor, i.e.

$$\left(\frac{L_{x\max}(\theta=1)}{k}\right)\times\left((k-1)\times\log_2 y+\log_2\left(L_{y\max}(\theta=0)\right)\right)$$

and

$$\left(\frac{L_{x\min}(\theta=1)}{k}\right)\times\left((k-1)\times\log_2 y+\log_2\left(L_{y\min}(\theta=0)\right)\right).$$

5.4.2 Operation Skew with Double Data Skew

A binary join of many-to-many relationship tables does not result in double data skew since there is always an intersection entity introduced in this situation and it is replaced with two binary joins of one-to-many relationship tables. However, when two relations have one common and non-key attribute, double data skew occurs if this attribute serves as the partitioning attribute.

Table: *Project*

Project-No	**Project-Name**	**City**	**Manager-ID**	**Sponsor-Name**

Table: *Supplier-Project*

Supplier-No	**Project-No**	**Qty**	**Date**

Based on the above *Supplier* and *Supplier-Project* table, and *Part* and *Supplier* table in section 5.4.1, if information on *Project* and *Supplier* are required, links must be established through the *Supplier-Project* table. Therefore, they may only involve multiple single data skew but not double data skew. However, if we try to collect some statistics based on the city location from table *Supplier* and *Project*, we may partition on the city attribute because it reduces the aggregation cost. This causes double data skew. The problem is more complex when both relations are deformed and follow some unknown discrete distributions. However, we believe that in real databases there is always some degrees of correlation between two attributes from two relations. For example, if they are the same attribute they will have the common domain, and one skewed value in one relation is likely to associate with other skewness in another relation. Therefore in the following subsection on double data skew, assume there is strong correlation between the two relations, i.e. the skewed fragment of relation R matches the skewed fragment of relation S.

Therefore, the maximum and minimum complexity of parallel nested loops join can be given as

$$L_{x\max}(\theta=1)\times L_{y\max}(\theta=1) \quad \text{and} \quad L_{x\min}(\theta=1)\times L_{y\min}(\theta=1).$$

Likewise, we can derive those of parallel hash join

$$L_{x\max}(\theta=1)+L_{y\max}(\theta=1) \quad \text{and} \quad L_{x\min}(\theta=1)+L_{y\min}(\theta=1),$$

and parallel sort merge join

$$L_{x\max}(\theta=1)\times \log_2\left[L_{y\max}(\theta=1)\right] \quad \text{and} \quad L_{x\min}(\theta=1)\times \log_2\left[L_{y\min}(\theta=1)\right].$$

5.5 Simulation Experimentation

5.5.1 Simulation Model

A simulation study has been conducted with a simulator written in C++ programming language running on a Sun workstation. The skew model is hardware independent since it predicts the partitioned load among processors together with its distribution, and does not specify how communication is modelled[7]. To simplify the implementation, we assume a parallel architecture shown in Figure 5.5, where data reside at the host site at the beginning of the processing and are distributed to processors randomly or according to specific distributions.

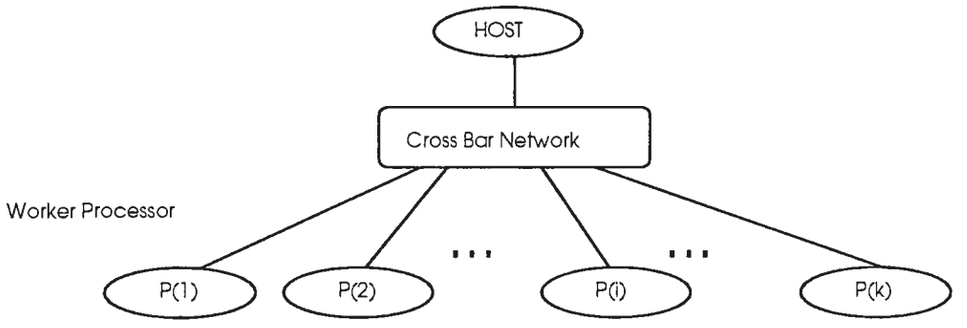


Figure 5.5: Parallel Database Architecture

The input of the simulation model include the relation cardinality, the number of processors, and the number of experiments per run, and the output from the simulation model can be the extreme loads (minimum and maximum) for each experiment, the statistics collected for each run (mean maximum and minimum as well as their standard deviations), ideal loading, and the graph representation of each experiment and each run. The details of the simulation model can be found in [Liu94] and the parameters used in the simulation are listed in Table 5.1 (see Appendix B).

There are three steps in the simulation and they are tuple generation, tuple allocation, and statistics collection. At the end of the simulation, a graph result representation is also provided to visualise the skew distribution. Skew prediction with range partitioning in

[7] Inserting a few parameters characterising communication, our model may become a complete parallel execution time prediction model and we will discuss this at the end of this chapter.

Section 4.4 is straight forward by calculating the integration of the input distribution but skew prediction with hash partitioning in Sections 4.6, 4.7, 5.3 and 5.4 involve a considerable amount of randomness. Therefore, we will evaluate the skew model of hash partitioning against our terabyte database simulation model [Liu94].

Parameters	**Meaning**
x	the number of tuples of the relation (cardinality)
k	the number of processors
Ideal	ideal loading = *x/k*
L	the number of tuples in the most heavily loaded processor
*L**	the number of tuples in the most lightly loaded processor
t	the number of tuples in the fullest or least fullest processors used in distribution function
Pr	the probability of *t* being the extreme value (either maximum or minimum)
Relative Error (%)	*(Predicted - Experimental) / Experimental*

Table 5.1: Parameters Listing for Evaluation

a. Tuple Generation

Tuples are generated randomly and the random number generator uses the linear congruential method. To reduce the variation two hundred runs are conducted for the same experiment, and for the initialisation purpose the seed is obtained by function "srand((int) time(NULL))".

```
unsigned temp;
for (unsigned long i=0; i<size; i++)
        {
        temp=rand( );                          /* generate tuples */
        /* each processor has the same probability receiving tuples i.e. no data skew */
        for (unsigned j=1; j<=size1; j++)              /* allocate tuples to processors */
                if (temp<(double(j) * Tup_Range / (double(size1))))
                        { ps[j-1]=ps[j-1]+1; break; }
        /* collect statistics for this experiment and store them in a structure */
        ...
        }
```

Figure 5.6: Algorithm for Uniform Skewness Generation

b. Tuple Allocation

In each run of the experimentation, when each tuple is generated, a random number is produced. The random number gives the tuple destination after comparing with the boundary values of each processor. Throughout the simulation, a tuple is treated as one

unit and distributed to one processor. Assuming each processor having the same probability receiving tuples, the algorithm for tuple allocation is showing in Figure 5.6.

c. Statistics Collection and Result Representation

Both the maximum and minimum values are collected in each run and two hundred runs are conducted for each experiment. To visualise the skew distribution, graph representation can be selected as shown in Figures 5.7 and 5.8 (directly output from the running program). In Figure 5.7, each 3D-bar represents one processor; the length of the 3D-bar represents the number of tuples in that processor; two vertical lines with coordinates on them provide scale of workload, i.e. 60 means 60 tuples. In Figure 5.8, two points with the minimum and maximum value for each run of the experiment are plotted; the relative positions of the points are determined by their maximum and minimum values and standard deviation from the mean load.

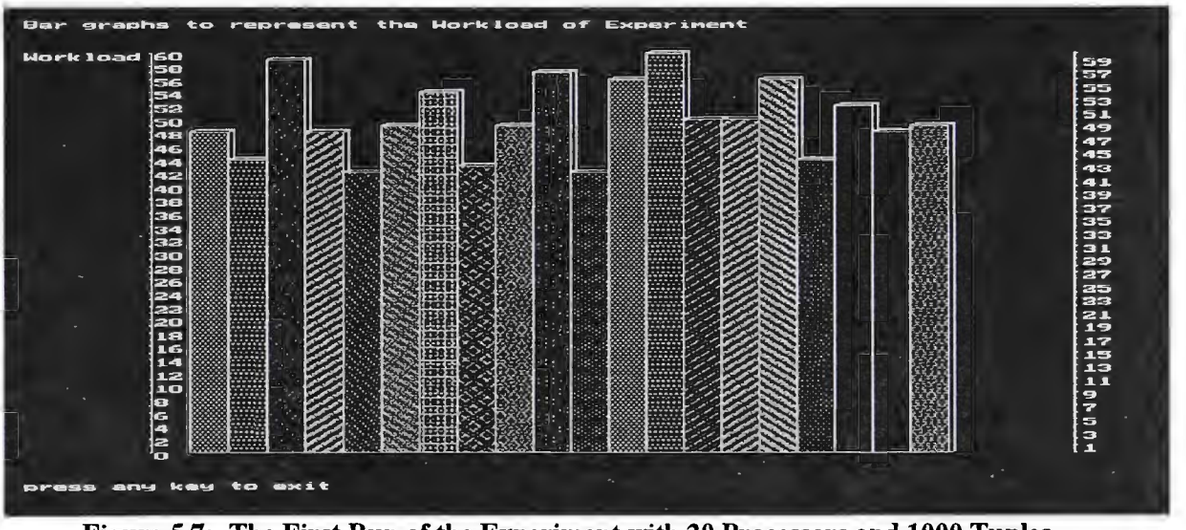


Figure 5.7: The First Run of the Experiment with 20 Processors and 1000 Tuples

5.5.2 Validation of Load Skew Foundation Model

In this subsection, we evaluate the load skew foundation model of Section 4.6 and this may be regarded as the heart of skew prediction because the operation skew model in Section 4.7 and the Zipf and Normal distribution in Chapter 5 can be viewed as extensions of the fundamental model. Consequently, we validate not only mean load of the extreme values but also their standard deviations and distribution functions. To reduce the variation introduced by the random numbers, a large number and a wide range of experiments are carried out for each run.

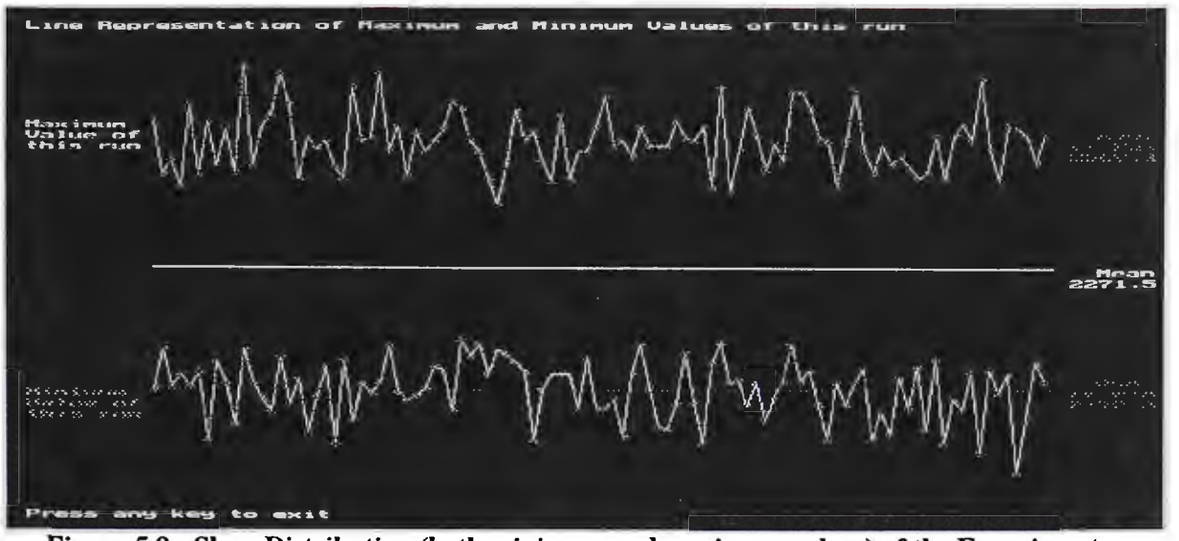


Figure 5.8: Skew Distribution (both minimum and maximum values) of the Experiment with 100 Runs Where Each Run Consists of 25 Processors and 56,789 Tuples

No. of Tuples	Comparison		Relative Error
(16 Processors)	Predicted	Experimental	(%)
5000	344.08	345.54	-0.42
6000	409.60	409.76	-0.04
7000	474.87	474.5	+0.08
8000	539.95	540.02	-0.01
9000	604.87	603.86	+0.17
10000	669.67	671.40	-0.26
20000	1313.17	1313.11	0
30000	1952.37	1952.56	-0.01
40000	2589.33	2598.47	-0.35
50000	3224.88	3223.72	+0.04
60000	3859.41	3858.61	+0.02
70000	4493.18	4495.54	-0.05
80000	5126.34	5125.87	+0.01
90000	5759.00	5761.66	-0.05
100000	6391.25	6392.90	-0.04
500000	31565.84	31573.36	-0.02
1000000	62946.67	62956.04	-0.01

Table 5.2: Skew Foundation Model Evaluation of Maximum Load with 16 Processors

a. Mean of Maximum and Minimum Load

To the best of our knowledge, there are few papers on skew prediction, and traditionally skewness in parallel database is either neglected or is assumed the ideal loading with equal partitioning. Figure 5.9 shows the comparison between the predicted values of skew model given by equation (4-1) together with experimental results obtained from simulation. Not only the maximum load but also the minimum load can be predicted based on the equation (4-2) and the comparison is shown in Figure 5.10. In Figures 5.9 and 5.10, the notation *4Pred* signifies the predicted result based on 4 processors, and *4Expe* signifies the corresponding observed experimental result. We observed that the predicted values are in close agreement with the experimental results. The number of tuples in these experiments range from 5000 to 1,000,000, and the number of processors ranges from 4 to 1024. Altogether, a total of two hundred experiments for each run have been performed and the comparisons are summarised in Tables 5.2 and 5.3.

No. of Tuples	Comparison		Relative Error
(16 Processors)	Predicted	Experimental	(%)
5000	280.916	280.85	+0.02
6000	340.401	341.99	-0.47
7000	400.129	401.58	-0.36
8000	460.049	462.21	-0.47
9000	520.125	520.71	-0.11
10000	580.333	582.82	-0.43
20000	1186.83	1191.03	-0.35
30000	1797.63	1798.4	-0.04
40000	2410.67	2413.64	-0.12
50000	3025.12	3029.3	-0.14
60000	3640.59	3645.93	-0.15
70000	4256.82	4256.37	+0.01
80000	4873.66	4876.58	-0.06
90000	5491	5493.8	-0.05
100000	6108.75	6113.06	-0.07
500000	30934.2	30873.36	+0.20
1000000	62053.3	62043.92	+0.02

Table 5.3: Skew Foundation Model Evaluation of Minimum Load with 16 Processors

b. Standard Deviation of Maximum and Minimum Mean Load

Due to the stochastic character of skew behaviour, the standard deviations of the mean load are often useful. A comparison of predicted values given by equation (4-3) with experimental results is shown in Table 5.4. The relative error is larger than their corresponding mean load. More experiments per run tend to produce more accurate results but will lead to longer simulation time. Nevertheless, these results suggest a relative error of no more than 5%.

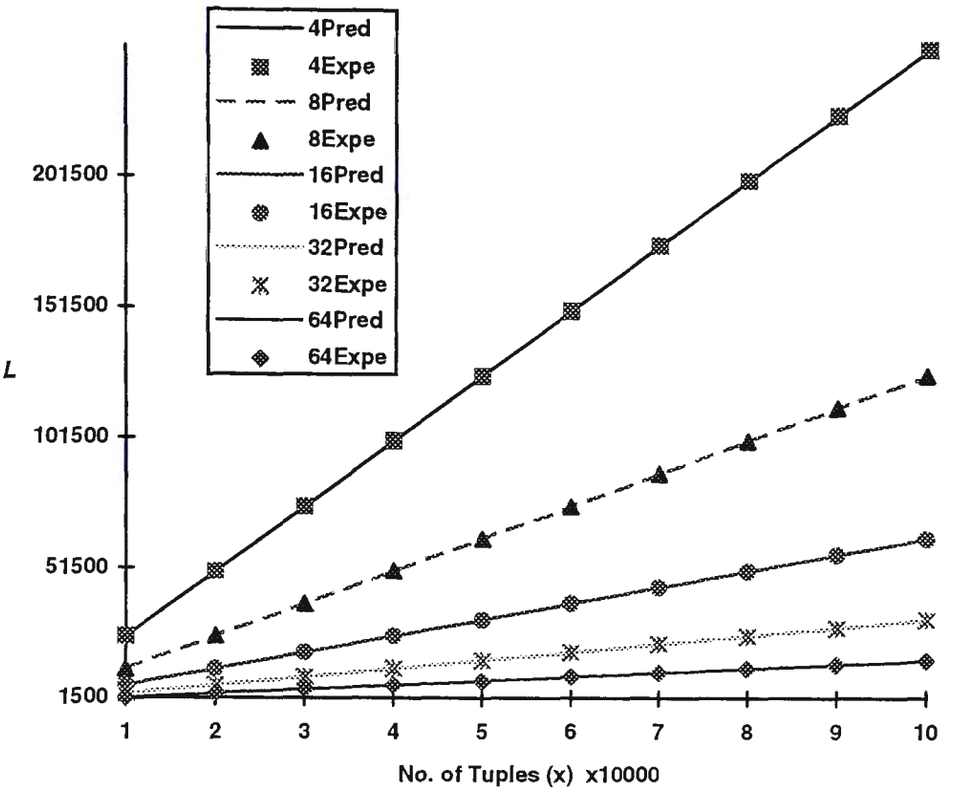
No. of Tuples	Comparison			Relative Error (%)	
(16 Processors)	Predicted	Experimental (Maximum, Minimum)		(Maximum, Minimum)	
1000	4.17	4.29	3.98	-2.92	+4.77
5000	9.32	9.10	8.88	+2.42	+4.95
10000	13.18	13.62	13.09	-3.23	+0.69
20000	18.64	19.16	18.09	-2.71	+3.04
30000	22.84	22.51	22.44	1.46	+1.77
40000	26.37	26.12	25.14	0.93	+4.87
50000	29.48	30.50	28.10	-3.34	+4.91
60000	32.29	30.79	31.47	+4.87	+2.63
70000	34.88	33.27	34.50	+4.82	+1.10
80000	37.29	35.90	36.17	+3.86	+3.10
90000	39.55	38.90	37.99	+1.67	+4.11
100000	41.67	43.08	41.50	-3.23	+0.41
500000	93.22	92.30	93.11	+1.00	+0.12
1000000	131.84	130.08	132.09	+1.35	-0.19

Table 5.4: Skew Foundation Model Evaluation of Standard Deviation of Mean Maximum Load and Mean Minimum Load with 16 Processors

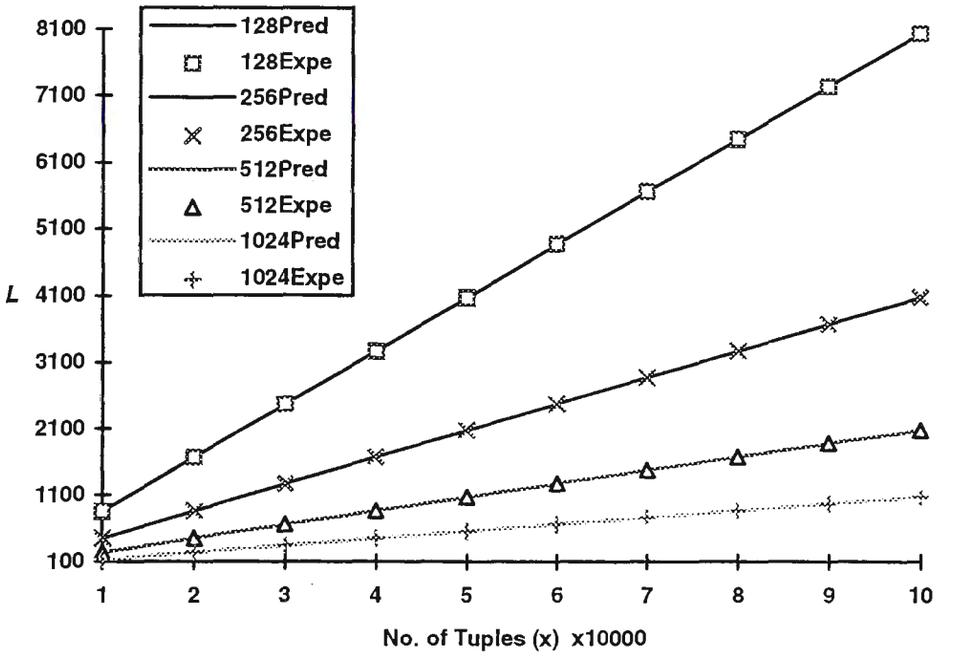
c. Distribution Function of Maximum and Minimum Load

The predicted maximum load distribution function is given by equation (4-4) and comparison with the simulation results is displayed in Figure 5.11. In Figure 5.11(a), there are 16 processors and 10000 tuples, and the ideal loading is 625 tuples per processor. From Figure 5.11(a), we see that the chance of the maximum load less than or equal to 640 tuples is zero, indicating that the ideal loading is far from accurate. Likewise, the situation with 256 processors and 100,000 tuples is shown in Figure 5.11(b). The

minimum load distribution function is given by equation (4-5) and the comparison with



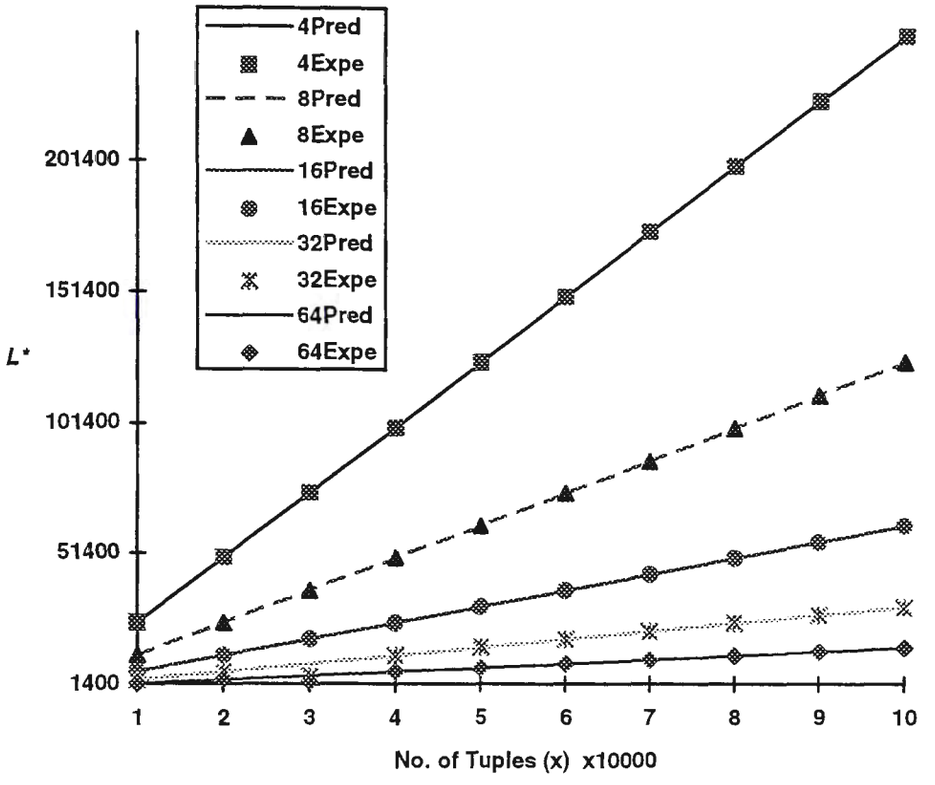
(a) 4-64 processors



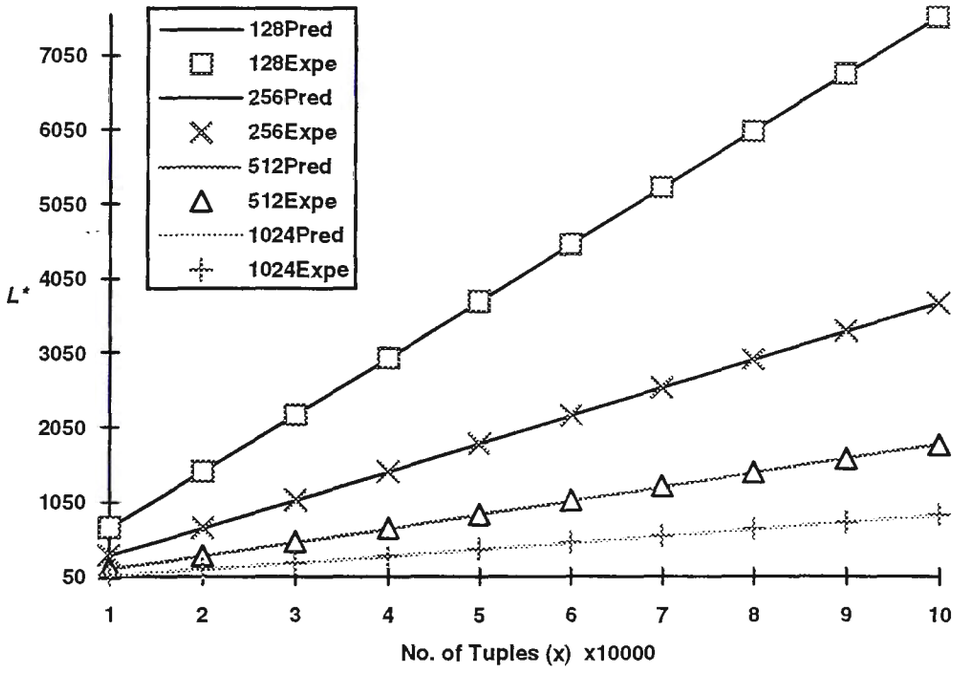
(b) 128-1024 processors

Figure 5.9: Validation of the Mean of Maximum Load of Skew Foundation Model

the simulation results is shown in Figure 5.12. In Figure 5.12(a), the chance of minimum load less than or equal to 604 tuples is 100%, and of course, this is much less than the ideal loading. Figure 5.12(b) shows the performance of 256 processors and 100,000 tuples.

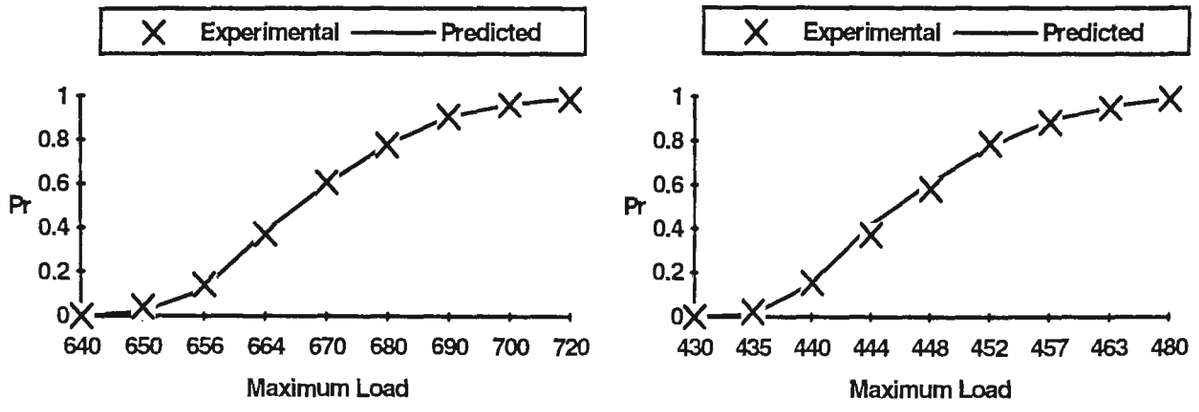


(a) 4-64 processors



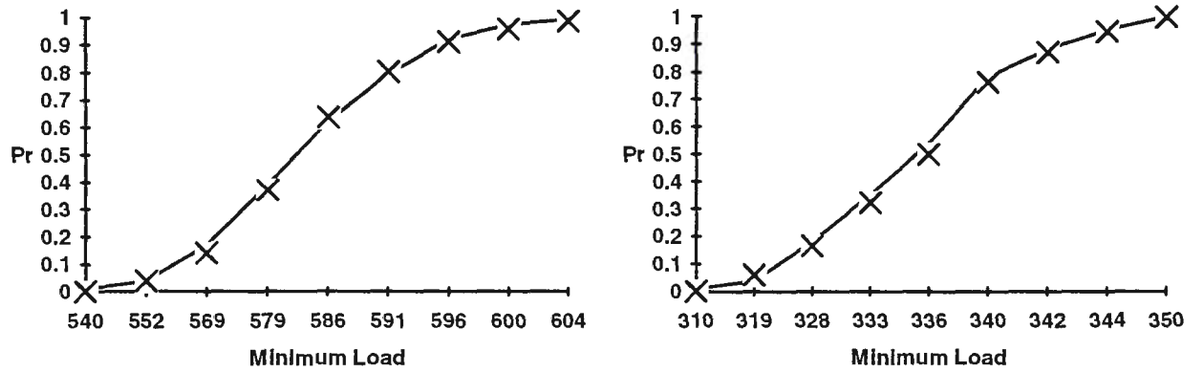
(b) 128-1024 processors

Figure 5.10: Validation of the Mean of Minimum Load of Skew Foundation Model



(a) 16 Processors and 10,000 Tuples (b) 256 Processors and 100,000 Tuples

Figure 5.11: Validation of the Maximum Load Distribution of Skew Foundation Model



(a) 16 Processors and 10,000 Tuples (b) 256 Processors and 100,000 Tuples

Figure 5.12: Validation of the Minimum Load Distribution of Skew Foundation Model

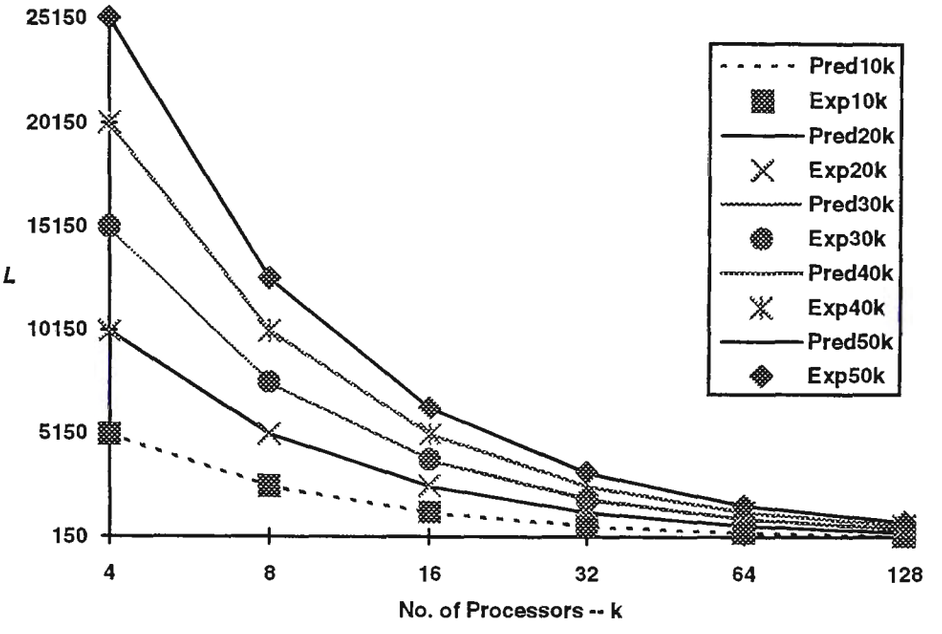
5.5.3 Validation of Operation Skew Foundation Model

In this section, we will evaluate the operation skew foundation model presented in section 4.7. Again, a number of experiments are conducted for each run and both the mean extreme load values and their standard deviations are obtained from the simulation. Considering that skew estimation only requires the mean workloads, we plot the following figures and present the discussion based on the mean load.

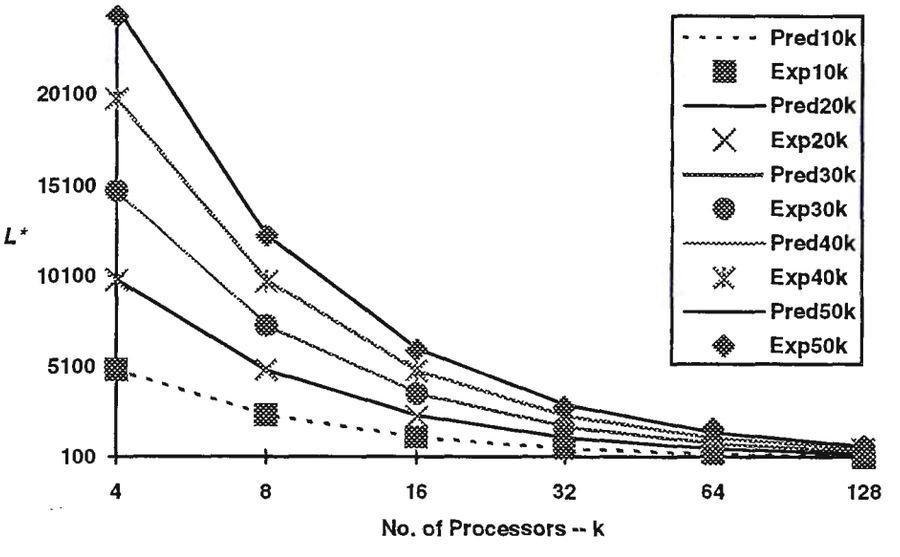
a. Parallel Hash Join

Figure 5.13 shows the comparison between the predicted operation skew values given by (4-10) and (4-11) and the simulation values. Increasing the number of processors will reduce both the maximum and minimum workload. There are five different groups workload ranging from 10000 to 50000 tuples. From the Figures 5.13(a) and 5.13(b), we

can conclude that the analytical model fits properly based on the simulation results.



(a) Maximum Load Prediction

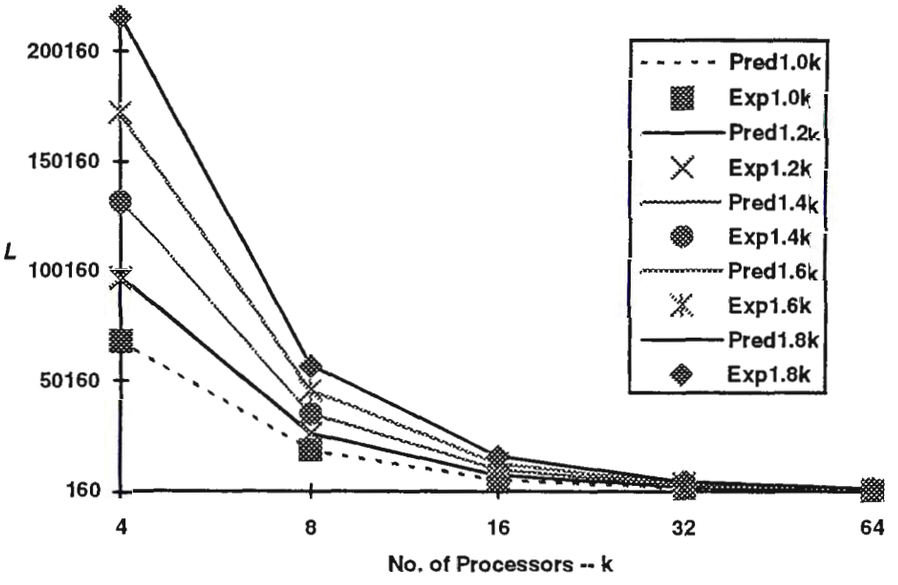


(b) Minimum Load Prediction

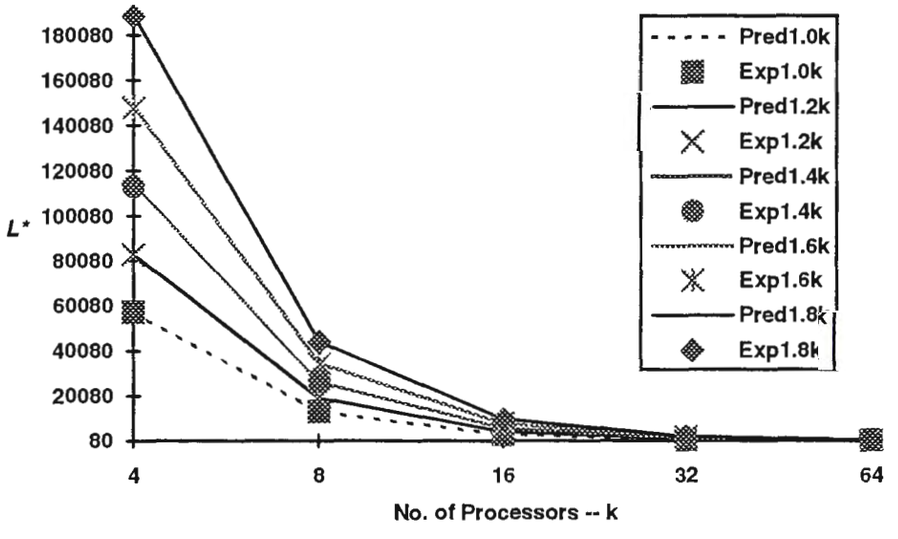
Figure 5.13: Validation of Operation Skew of Parallel Hash Join

b. Parallel Nested Loops Join

Figure 5.14 shows the model evaluation of parallel nested loops join. There are five groups of workloads ranging from 1000 to 1800 tuples. Both maximum and minimum load prediction based on the formula provided in Section 4.7 match the experimentation values.



(a) Maximum Load Prediction

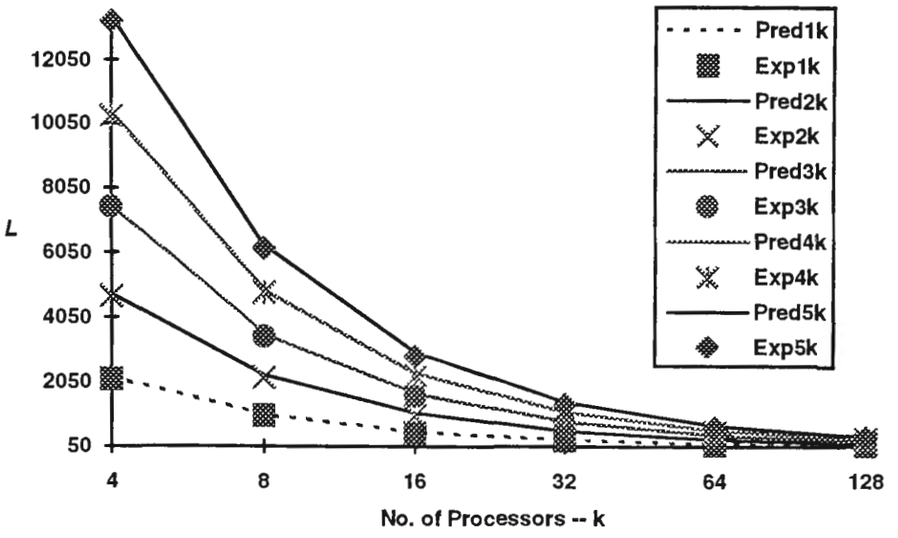


(b) Minimum Load Prediction

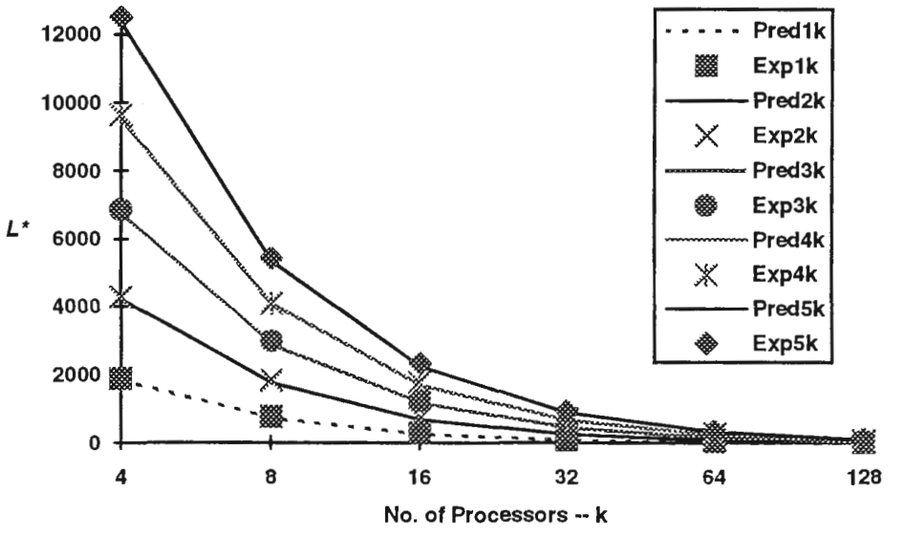
Figure 5.14: Validation of Operation Skew of Parallel Nested Loops Join

c. Parallel Sort Merge Join

The evaluation of parallel sort merge join is shown in Figure 5.15. With a large number of processors, the maximum load is reduced when the workload is set. There are five groups of workload ranging from 1000 to 5000 tuples. All predicted values agree with the experimental values collected.



(a) Maximum Load Prediction



(b) Minimum Load Prediction

Figure 5.15: Validation of Operation Skew of Parallel Sort Merge Join

5.5.4 Load Skew and Operation Skew Generation

a. Load Skew Generation

With the data skew modelled by discrete uniform distribution (i.e. no data skew), load skew is still possible because of the selection of partitioning function, e.g. when hash partitioning is used, a perfect hash function is hard to find. The algorithm of generating load skew with the uniformly distributed data skew is shown in Figure 5.6. When the data skew is modelled by a pure Zipf distribution (modified version introduces a parameter θ),

the probabilities of processors receiving tuples are changed correspondingly and the algorithm generating the skewness is listed below in Figure 5.16.

```
unsigned temp;
for (unsigned long i=0; i<size; i++)
        {
        temp=rand( );                       /* generate tuples */
        double temp_sum=0.0;
        /* processors probabilities of receiving tuples follow a Zipf distribution */
        for (unsigned j=1; j<=size1; j++)    /* allocate tuples to processors */
                if (temp<(temp_sum+(Tup_Range / (double(j) * Harmo))))
                        { ps[j-1]=ps[j-1]+1;
                          break; }
                else
                        {temp_sum=(Tup_Range / (double(j) * Harmo) + temp_sum;}
        /* collect statistics for this experiment and store them in a structure */
        ...
        }
```

Note: *Harmo* is the function to calculate the Harmonic number of *size1* which is the number of processors

Figure 5.16: Algorithm for Zipf Skewness Generation

```
unsigned temp;
double probabi, SND_Value;
for (unsigned long i=0; i<size; i++)
        {
        temp=rand( );                       /* generate tuples */
        /* processors probabilities of receiving tuples follow Normal distribution */
        for (unsigned j=0; j<size1+2; j++)          /* allocate tuples to processors */
                {
                SND_Value=double(j) * 6.0 / double(size1) - 3;
                if (SND_Value > 0)
                        probabi=1-S_Nor_Dis(SND_Value);
                else
                        probabi=1-(1-S_Nor_Dis(abs(SND_Value)));
                if (temp<(Tup_Range * probabi))
                        { ps[j-1]=ps[j-1]+1; break; }
                }
        /* collect statistics for this experiment and store them in a structure */
        ...
        }
```

Note: *S_Nor_Dis(SND_Value)* is the function to calculate the Standard Normal Distribution of value *SND_Value*.

Figure 5.17: Algorithm for Normal Distribution Skewness Generation

When the data skew is modelled by Normal Distribution, i.e. the deformity on the number of appearances of attribute values in the incoming query follows a bell-shape symmetrical normal distribution, Figure 5.17 presents the simulation algorithm. It is also worth pointing out that numerical integration is required in the calculation of standard normal distribution so that it slows down the simulation speed significantly.

b. Operation Skew Generation

For unary relational operation, certainly, load skew is equivalent to operation skew. However, for binary relational operations such as join, both join methods and data skew have influence on operation skew. If the operand relations are all uniformly distributed, the algorithm generating skewness is listed in Figure 5.18.

```
unsigned temp;
for (unsigned long i=0; i<size; i++)          /* deal with tuples in Relation R   */
        {
        temp=rand( );                         /* generate tuples                  */
        /*each processor has the same probability receiving tuples i.e. no data skew*/
        for (unsigned j=1; j<=size1; j++)     /* allocate tuples to processors    */
                if (temp<(double(j) * Tup_Range / (double(size1))))
                        { ps[j-1]=ps[j-1]+1; break; }
        }
for ( i=0; i<size0; i++)                      /* deal with tuples in Relation S   */
        {
        temp=rand( );                         /* generate tuples                  */
        /*each processor has the same probability receiving tuples i.e. no data skew*/
        for ( j=1; j<=size1; j++)             /* allocate tuples to processors    */
                if (temp<(double(j) * Tup_Range / (double(size1))))
                        { pr[j-1]=pr[j-1]+1; break; }
        }
for (j=0; j<size1; j++)
        {
        if (flag="HASH")                      /* join method selection            */
                ps[j]=ps[j]*pr[j];
        else if (flag="NESTED_LOOPS")
                ps[j]=ps[j]+pr[j];
        }
delete pr;
/* collect statistics for this experiment and store them in a structure        */
...
```

Figure 5.18: Algorithm for Uniform Distribution Operation Skewness Generation

Figure 5.19 shows the situation when both input relations follow Zipf distribution and the corresponding operation skew comprises three steps. First, relation R is partitioned to processors according to the Zipf distribution, and then relation S is fragmented over the same number of processors still based on the Zipf distribution but into a temporary array. In the last step, fragments are randomly allocated to processors. The algorithm is shown in Figure 5.20.

Figure 5.21 shows the situation when both operand relations follow normal distribution and a random merging takes place at the end. The algorithm doing so is very close to that of Zipf distribution which is shown in Figure 5.20.

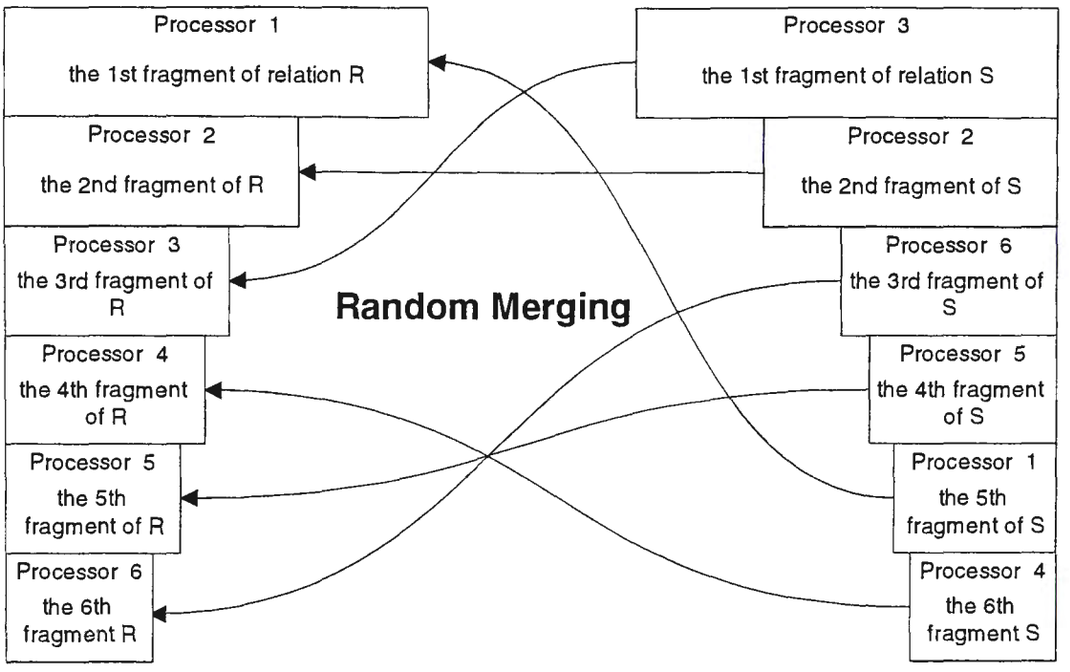


Figure 5.19: Generating Operation Skew with Zipf Distributed Relations

```
/* initialise all elements of the dynamic array pt[ ] to -1 */
for (j=0; j<size1; j++)
        {
        temp=rand( ) % size1;
        for (unsigned k=0; k<=j; k++)                    /* check its uniqueness      */
                while (temp=pt[k]) do
                        temp=rand( ) % size1;
        if (flag="HASH")                                 /* join method selection     */
                ps[j]=ps[j]*pr[temp];
        else if (flag="NESTED_LOOPS")
                ps[j]=ps[j]+pr[temp];
        pt[j]=temp;
        }
```

Figure 5.20: Algorithm for Random Merging on Operation Skewness Generation

5.5.5 Validation of Load Skew Extension Model

a. Load Skew Prediction for the Zipf Data Skew Distribution

In this section, we evaluate the load skew prediction when the degree of data skew is represented by a pure Zipf distribution. In equations (5-11) and (5-12), we analytically approximate the harmonic number H_k by $(\gamma + \ln k)$, and thus we expect the relative error will be slightly larger than that without data skew, especially when the number of processors k is small. Figure 22(a) shows the comparison between the predicted load skew given by equation (5-11) and the simulation results, and Figure 22(b) offers the contrast

between predicted minimum load by equation (5-12) and that by simulation. Despite the approximation, both of the relative errors of minimum and maximum load prediction are less than 10%, and when k is greater than or equal to 8, the relative error of maximum load prediction is less than 4%.

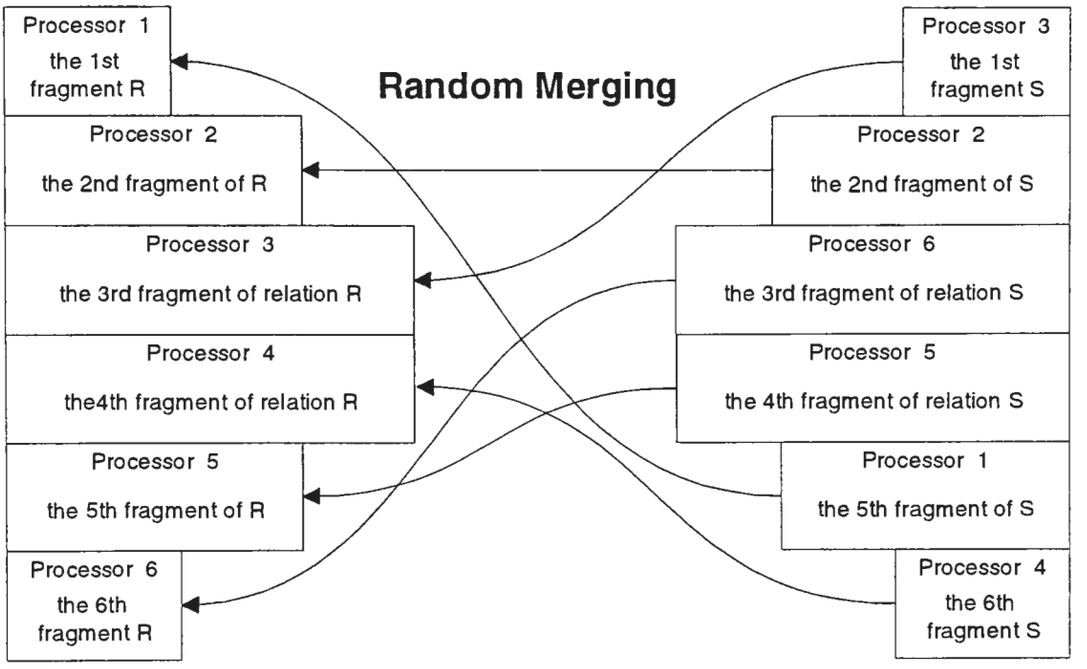
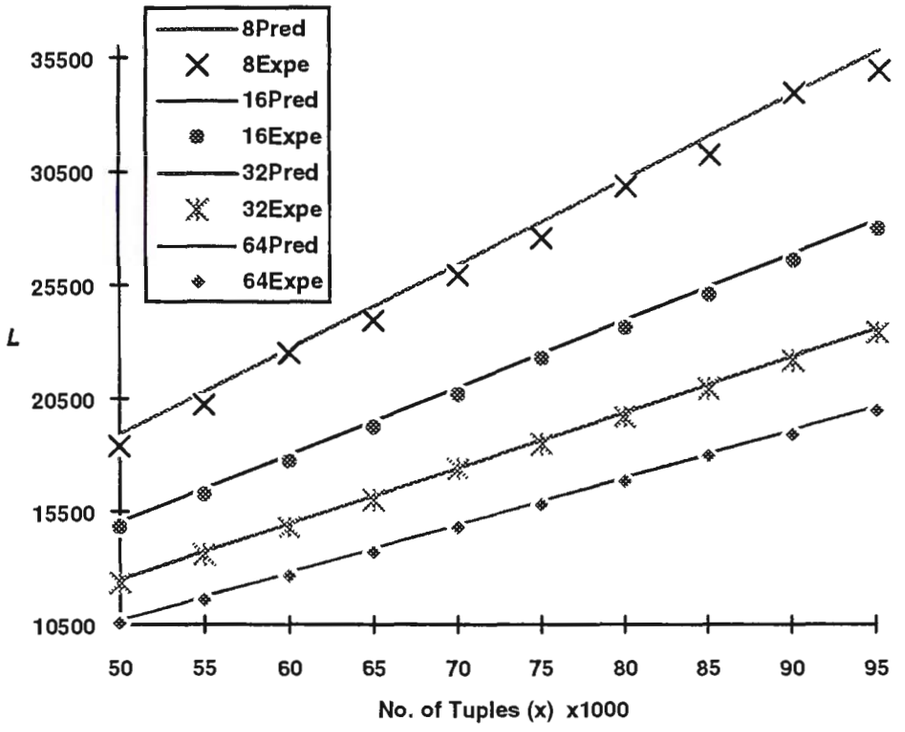


Figure 5.21: Generating Operation Skew with Normal Distributed Relations

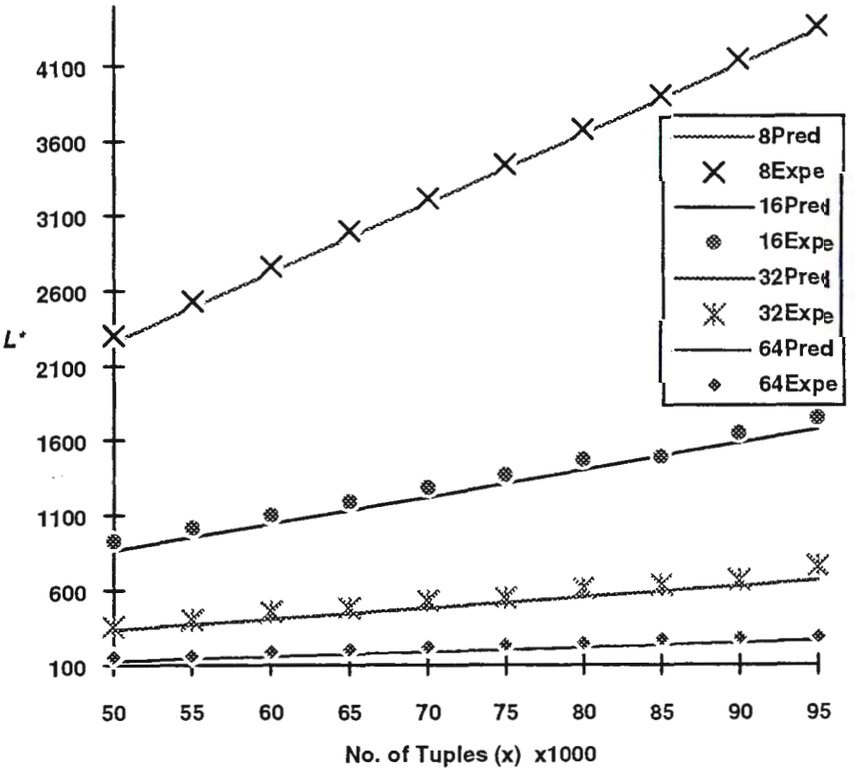
b. Load Skew Prediction for the Normal Data Skew Distribution

With the data skew modelled by the normal distribution, the degree of skewness is less than that of pure Zipf distribution, and the prediction results are given by equations (5-16) and (5-17). In the simulation program, to avoid the complex integration which not only slows down the simulation speed but also reduces the accuracy because of dynamic setting the integration interval, the approximation from equation (5-15) is also employed to distribute the tuples. Figure 5.23 shows the comparison of the maximum and minimum load for the normal distribution. The relative error is less than 3% for maximum load prediction and 5% for minimum load prediction.



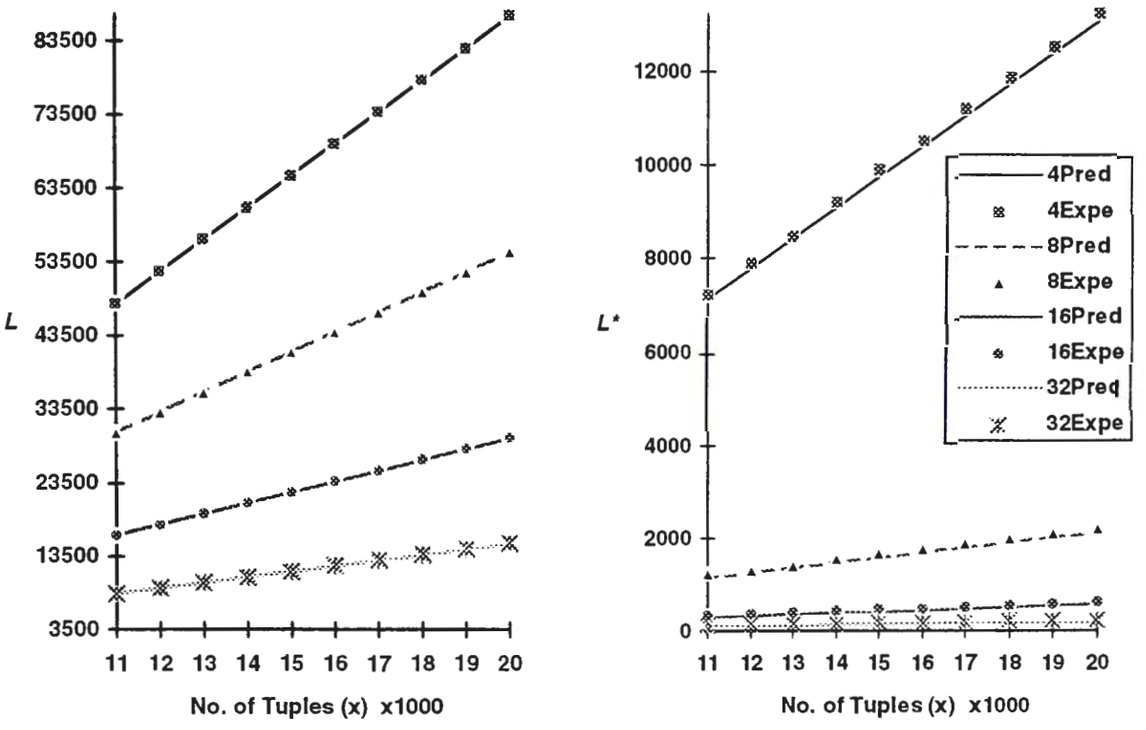
Maximum load skew

Figure 5.22(a): Load Skew Prediction with High Data Skew (pure Zipf Distribution)



Minimum load skew

Figure 5.22(b): Load Skew Prediction with High Data Skew (pure Zipf Distribution)



(a) Maximum Load (b) Minimum Load

Figure 5.23: Validation of the Load Skew Prediction with Data Skew Modelled by Normal Distribution

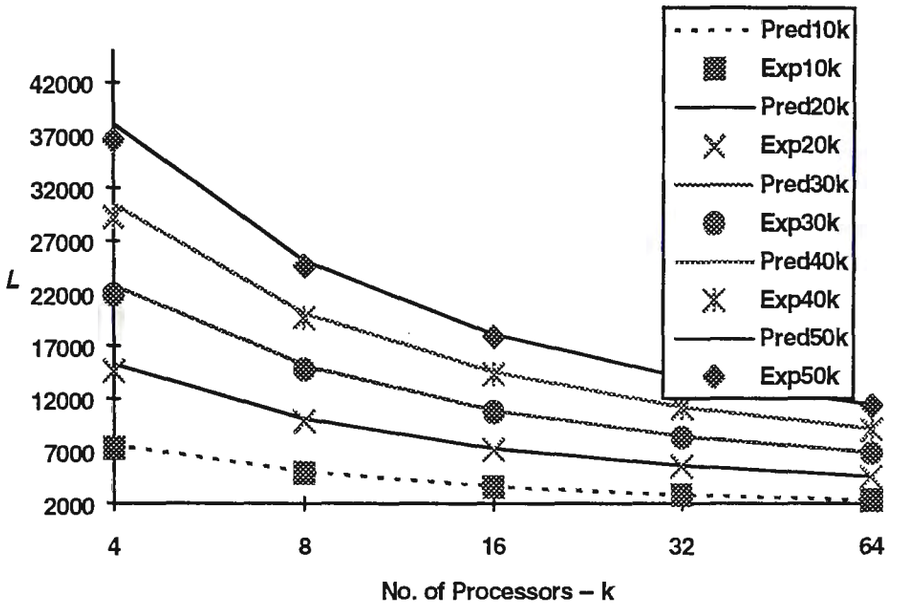
5.5.6 Validation of Operation Skew Extension Model

In this section, the operation skew extension model in section 5.4 is evaluated. Based on the skew dimension, the case of no data skew is discussed in Chapter 4 and we shall focus on the evaluation of single and double data skew here. We only provide the experimentation result with parallel hash join since all three foundation models are already verified in section 5.5.3. Therefore, our primary concern is on the operation skew factor. Once again, to reduce variation a large number of experiments are conducted for each run and both the mean maximum and minimum extreme load values are collected from the simulation.

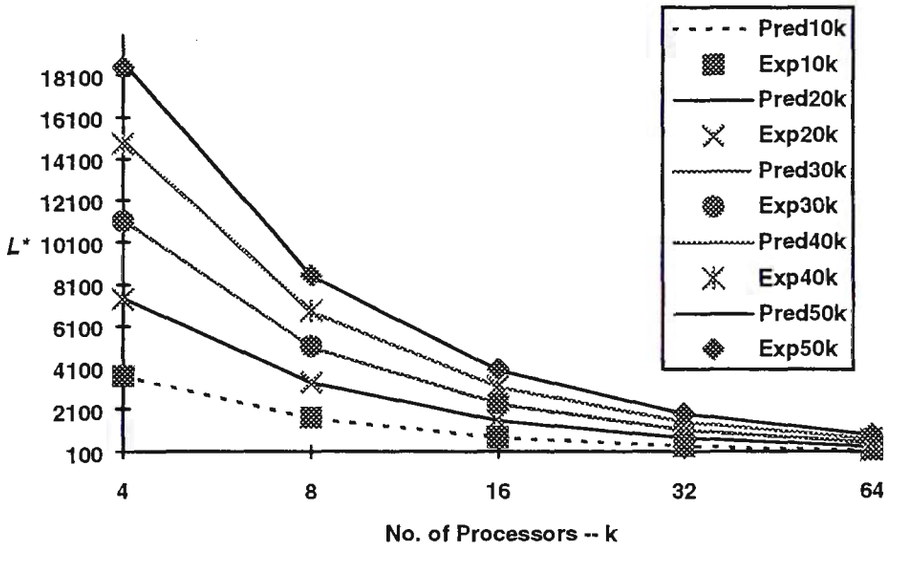
a. Parallel Hash Join with Single Data Skew

With parallel hash based join methods, the complexity at each processor is simply related to the number of tuples from each relation distributed to the processor. Figure 5.24 shows the comparison between the predicted operation skew values and the simulation results. In the figure, increasing the number of processors will reduce both the maximum and minimum workload despite the skewness in one relation, and the operation skew load is

increased when the size of relations is raised. Based on five different groups workload ranging from 10000 to 50000 tuples, we see that the analytical model provides good agreement with experimental results.



(a) Maximum Load Prediction



(b) Minimum Load Prediction

Figure 5.24: Validation of Operation Skew of Parallel Hash Join

b. Double Data Skew

With double skew operations, based on our assumption on strong correlation, skew fragments in one relation will match skew fragments in another relation. Therefore, the situation may be simplified by distributing two relations to the same number of processors with exactly the same probability of receiving tuples. Certainly, this will give the same result as distributing one large relation to processors. Thus, it presents the same trend as that of load skew validation with Zipf or Normal data skew.

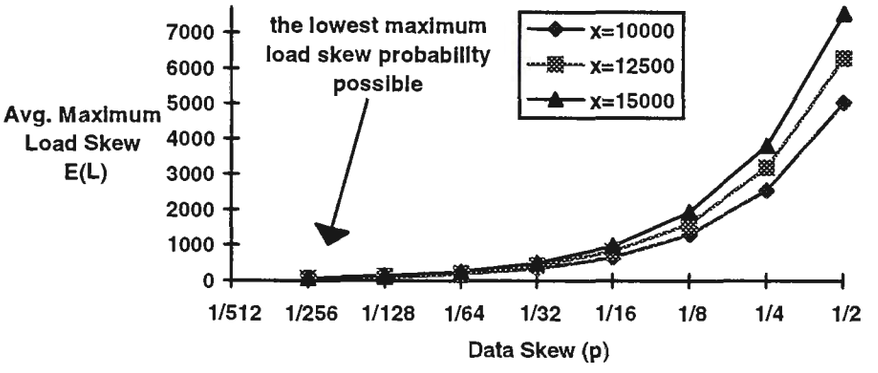


Figure 5.25(a): Maximum Load Skew with 256 Processors

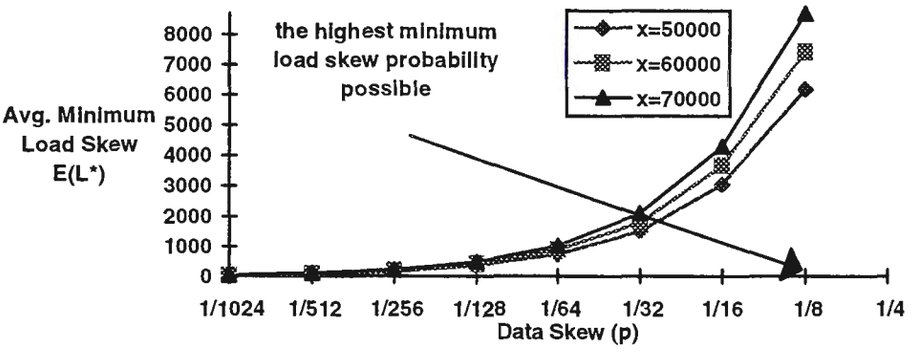


Figure 5.25(b): Minimum Load Skew with 8 Processors

5.6 Data Skew, Load Skew and Operation Skew

Maximum load skew is the number of tuples in the fullest processor, and when we increase either the relation cardinality or the degree of data skew, the maximum load skew steadily increases as shown in Figure 5.25(a) which is based on equation (5-1). With 256 processors, the maximum probability of receiving tuples among processors is $1/k$ = 1/256 without data skew, but using $1/k$ = 1/128 means that the degree of data skew is increased by a factor of 2. We see from the figure that maximum load skew magnifies supra-linearly

with increasing data skew. Minimum load skew is the number of tuples in the least loaded processor, and when we increase either the cardinality of the relation or the degree of data skew, minimum load skew also grows as shown in Figure 5.25(b) with 8 processors, which is based on equation (5-4).

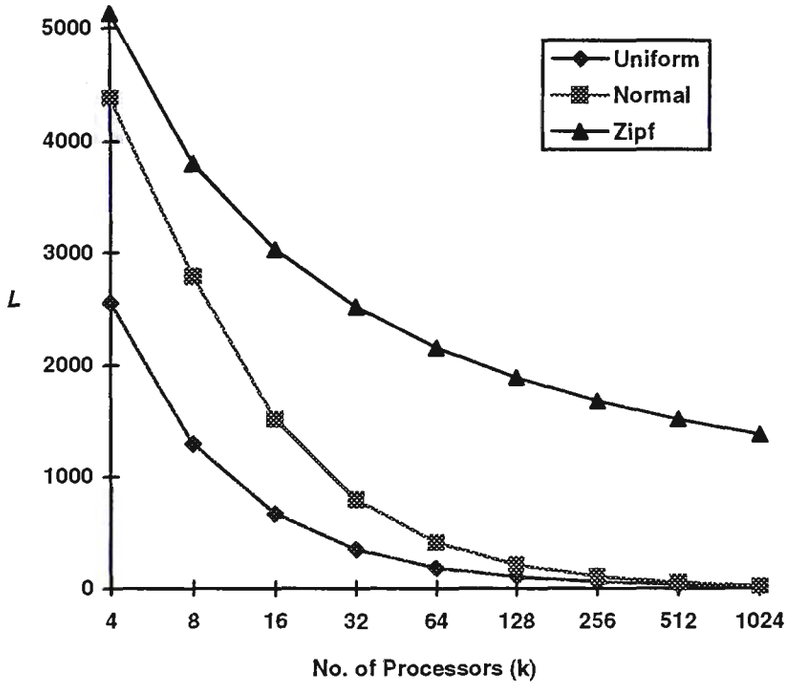


Figure 5.26(a): Maximum Load with 10000 Tuples

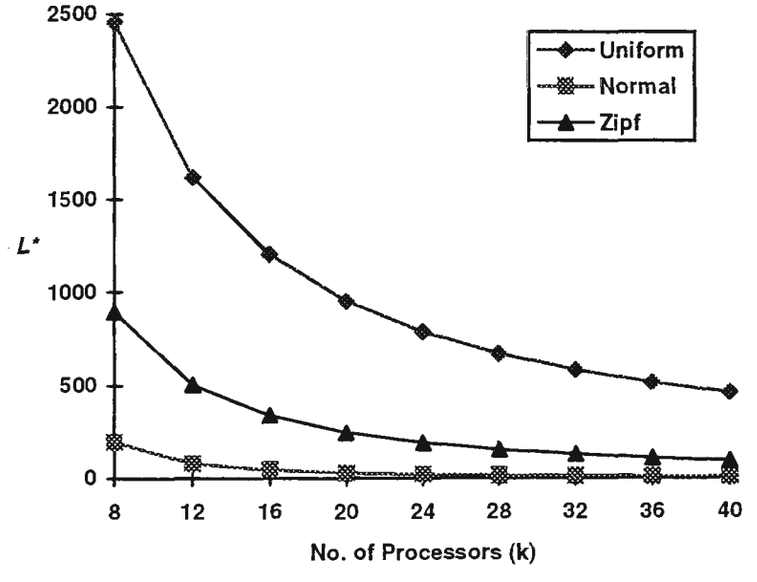


Figure 5.26(b): Minimum Load with 20000 Tuples

Figures 5.26(a) and 5.26(b) show different kinds of data skew model and their influences on load skew with a fixed relation cardinality and an increasing number of processors.

The pure Zipf distribution tends to yield high maximum load skew especially when the number of processors is large. The maximum load prediction of the Normal Distribution always lies between those of the Uniform and Zipf distributions, and thus may be used to model the situations when the degree of data skew is moderate. In addition, with the Normal Distribution, maximum load skew increases much slower than that of Zipf distribution when the number of processors increases.

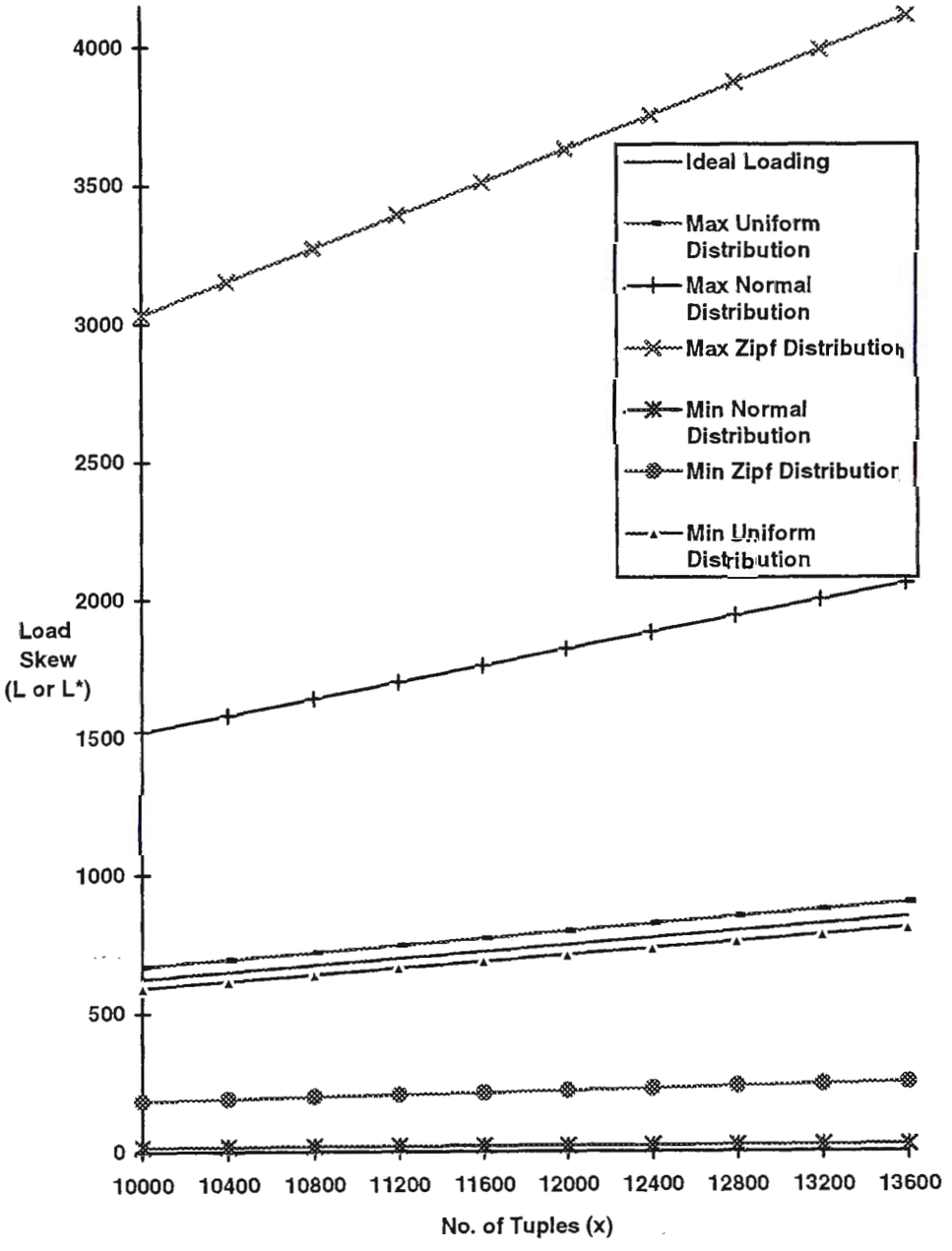


Figure 5.27: Varying the Number of Tuples with 16 Processors

Figure 5.27 shows the different kinds of data skew models and their influences on load skew for different relation cardinality. This suggests that the Uniform distribution of values leads to low load skew, Normal Distribution of data values leads to moderate load

skew, and pure Zipf distribution of data values results in high load skew. Ideal loading always underestimates maximum load and overestimates minimum load even when data skew is absent. In addition, the minimum load skew of the Normal Distribution gives values less than those of the Zipf distribution since the Normal Distribution density spreads over a shorter range (the probabilities associated with 2σ, 4σ, and 6σ are 68%, 95%, and 99.73% respectively) and the Zipf distribution has a long flat tail. However, comparing both the range (difference between the maximum and the minimum) and the maximum load skew, the Zipf distribution still tends to generate a larger degree of skew.

In Figures 5.28, 5.29, and 5.30, the impact of data and load skew on operation skew is studied and the relational cardinality for R and S are 15,000 and 5,000 tuples. In the absence of data skew, the relationship between load skew and operation skew over the number of processors is displayed in Figure 5.28. Both maximum and minimum load for each relation are shown and their corresponding operation skew is also presented in the figure. The larger relation tends to produce larger skewness and operation skew is always higher that either of the load skew of the relation operand.

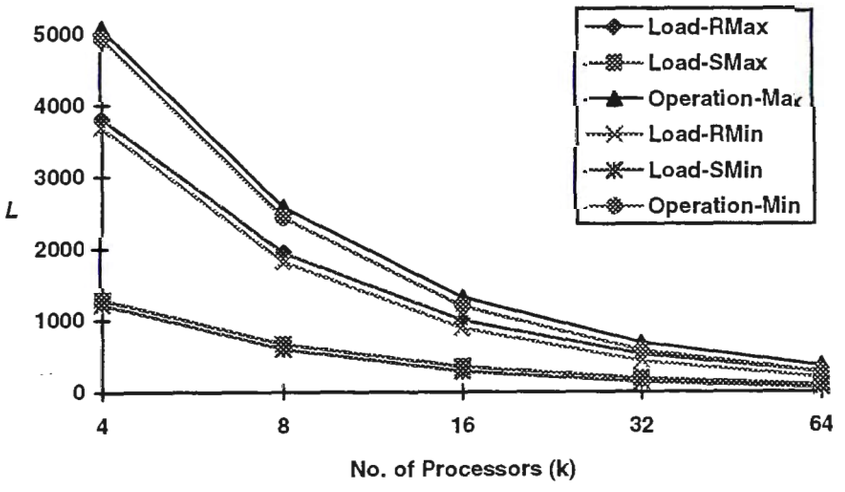


Figure 5.28: Load Skew and Operation Skew in Hash Join without Data Skew

When one relation is skewed and another relation is distributed uniformly, the operation skew performance is shown in Figure 5.29. An interesting point is that there is a cross over between the maximum load of relation R and S. This is caused by the Zipf skewness in relation S. Comparing R and S, S gives less load with a small number of processors

because of its small cardinality and S results in more load with a large number of processors because the skewness threatens the benefit of parallel processing.

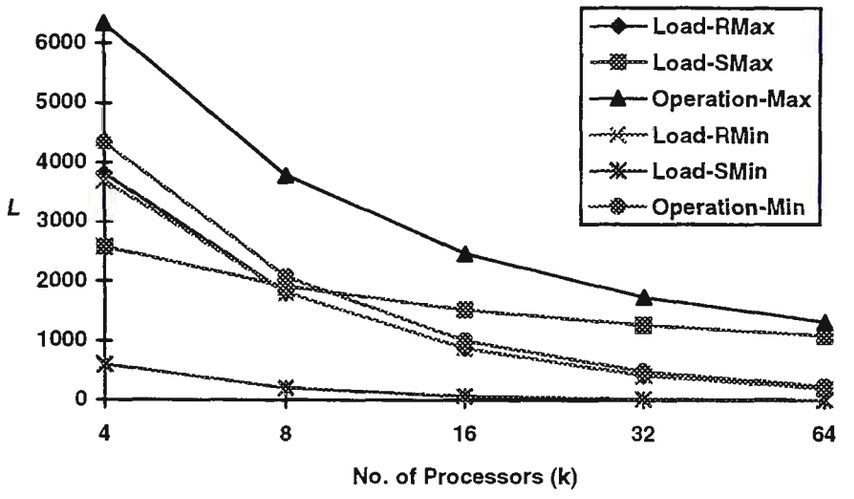


Figure 5.29: Load Skew and Operation Skew in Hash Join with Single Zipf Data Skew

When both of the input relations of the binary relational operation are skewed, the maximum load increased and the minimum load decreased sharply as shown in Figure 5.30. Operation Skew is a combined effects of both load skew and thus it is higher than any of them. Assuming both load skew following Zipf distribution, the relation with larger cardinality gives higher load skewness.

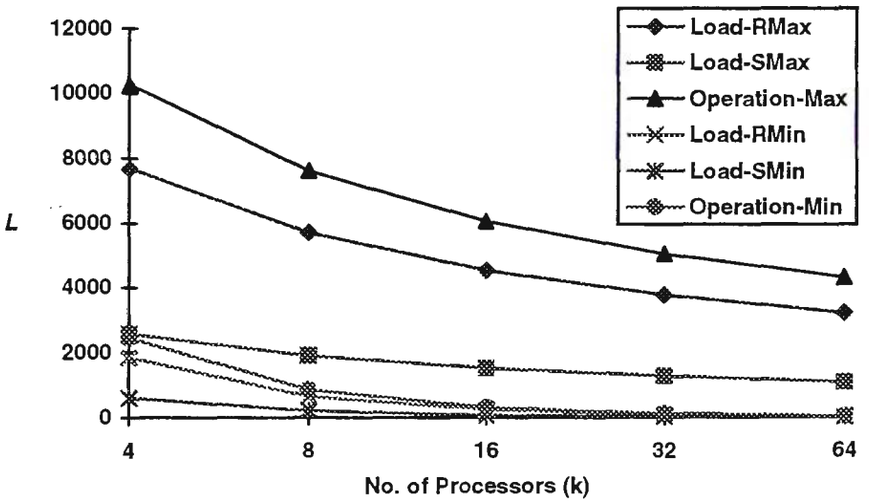


Figure 5.30: Load Skew and Operation Skew in Hash Join with Double Zipf Data Skew

5.7 Concluding Remarks on Skew Modelling and Prediction

In Chapter 3 a skew taxonomy is established, and in Chapters 4 and 5 the skew models are developed. Thus, together, the skew foundation is laid down with a detailed systematic study. The problem of data skew under range partitioning is analysed and the degree of data skew is quantified. A complete analytical model for load skew under hash partitioning is presented and consists of the foundation model and extension model. Not only the mean maximum and minimum load skew, but also their standard deviations and distribution functions are provided in the skew foundation model. In addition, the model is extended to include data skew described by the Zipf distribution and the Normal distribution. The skew model is verified on the Terabyte Database Simulation Model and experimental results exhibit close agreement with the predicted values. An analysis of the relationship between data skew and load skew is also provided based on the proposed model, and the results show that it is possible to have load skew even when there is no data skew, and as data skew increases, load skew will grow supra-linearly.

The usage of the performance model is to predict the execution time for the average case. In the thesis, both the foundation and extension models are given based on the average case. If the degree of data skew can be detected, the Zipf skew model and the corresponding extension skew model can provide a meaningful prediction with the parameter θ varying from 0 to 1, matching a wide range degrees of data skew. However, if the data skew is unknown or is too costly to estimate, the skew model can still be usefully employed by adopting the Normal distribution for describing data skew, since the Normal distribution lies between the Zipf and Uniform extremes. Furthermore, the distribution of a large number of tuples in large databases tends to the Normal distribution by virtue of the Central Limit Theorem.

We summarise the applications of the skew model as follows

- Predict execution time.

  Not only in parallel databases but also in parallel programming, the skew model can be used for time prediction because Amdahl's Law with ideal loading is always an

under-estimation, and the Gustafson function (the parallel component is a function of parallel processors i.e. $f(k)$) does not include overhead and does not give the details of the function. The presented skew model and the parallel processing methods of the operations can form a computation model which can be further combined with architecture characteristics, thus giving a complete time prediction performance model. In addition, the skew model could be extended to model the propagation of skew in multiway join, and the error rate on this propagation could also be quantitatively analysed.

- Predict system utilisation.

  Given the minimum load skew of the skew model, we can predict when the first processor finishes its work in multiprocessor systems. Dividing minimum load skew into ideal load or the mean load, we can have a system utilisation factor varying from 0 to 1. Based on the model, we may conclude that 100% utilisation is hard to obtain because of the selection of the transforming function.

- Provide alternatives to resolve skew problems.

  Applying our model, we can determine the precise degree of load skew, so that we may avoid cases of no load skew or low load skew in which the benefits of most of existing skew handling methods are minimal. In the light of the skew model, we can acquire a deeper understand of skew behaviour in parallel relational database. Using the minimum and maximum load skew prediction, we can design dynamic algorithms with precise threshold functions or complete cost models to tackle certain degrees of load imbalance without involving excessive complexity.

- Analysis of Sensitivity.

  Skew prediction may provide a theoretical foundation by which the various skew handling algorithms may be compared. In addition, it facilitates database systems tuning so as to provide the most effective means of optimising system efficiency.

- Provide more effective and efficient parallel processing.

  Other benefits gained from skew prediction are data placement efficiency and query optimisation. When data are placed using a hashing function, by calculating the skew distribution, we can predict the maximum number of tuples allocated among processors. Taking into account the skew and load imbalance factor, we can have

more accurate cost functions for relation operations for various types of queries. Therefore, it improves both precision and efficiency in the optimisation of query execution.

CHAPTER 6

MINIMISING THE SKEW EFFECT ON PARALLEL QUERY PROCESSING

6.1 Introduction

Query execution in parallel databases involves translating a high-level query into an efficient low-level execution plan followed by the execution of the plan. The formulation of a parallel query plan deals not only with the execution sequence and the processing methods of the operations required in the plan, but also the processor allocation which enables

parallel processing of the operations so as to minimise query response time [Hong92, Gang92, Chek95].

This chapter investigates the effect of data skew on the parallel execution of queries that involve a number of relational operations, and explores the processor allocation strategies that may reduce the negative effect of the data skew. In particular, three parallel query processing strategies are presented: *intra-parallel processor allocation (IPA)* applies sole intra-operation parallelism for query execution; *phase-based processor allocation (PPA)* adopts a phase-oriented paradigm to cluster the operations of the query plan into several execution phases; and *modified phase-based processor allocation (MPA)* attempts to improve the *PPA* strategy when some of the operations are expected to involve a certain degree of data skew. The performance of these three processor allocation strategies are evaluated by a simulation study under various settings. Two join partitioning methods, *simple range partitioned join* and *hash partitioned join*, are presented and their performance in the presence of data skew is investigated.

The remainder of the chapter is organised as follows. Section 6.2 introduces the system architecture. Section 6.3 presents the query cost models for the selection, projection and join operations. The processor allocation methods are presented in Section 6.4, followed by the performance evaluation in Section 6.5. The chapter is concluded in Section 6.6.

6.2 System Architecture for Parallel Query Processing

This thesis may be viewed as two parts, skew modelling and skew effects on query processing. Although skew modelling is implemented on a shared memory parallel client-server system and will be presented in Chapter 10, the skew model itself is hardware independent. For query processing in Chapters 6 to 9, we assume a *shared nothing* parallel database architecture *SN* which consists of multiple processors connected by a cross bar network as shown in Figure 6.1 [Leun93]. This is analogous to the Transputer System and the implementation of databases on a such system can be found in [Leun93]. The cross bar is capable of connecting n inputs and n outputs in a one-to-all fashion so that a *broadcast* operation can be carried out in one single step. By employing this kind of dynamic

interconnection networks[8], messages can be passed to destination processor from source processor without hopping through any intermediate processors.

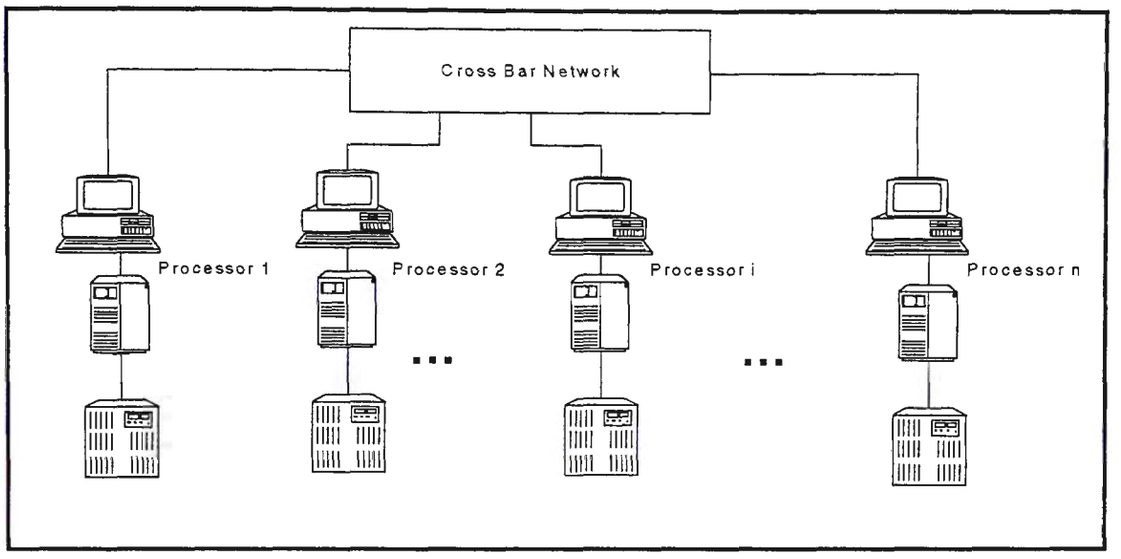


Figure 6.1: A Parallel Database Architecture

Hereafter, we shall separate the word *contention* and *overhead*. The former is related to resources such as memory and disk while the latter is associated with communication. The best example to explain contention is by a queuing model [Leun88] where the number of servers (processors) is limited so that customers (tasks or workloads) have to wait in a queue for service available. The resource contention problem occurs only when the resources are shared among processing elements, i.e. it is not an important issue in *SN*. The overhead is hardware-connection dependent and is directly caused by inter processor communication. In other words, the communication overhead may be reduced by adding complexity in connection, e.g. in a fully connected network.

In addition, we define the word *fine grain* and *coarse grain* in parallel systems as follows. With a fine grained parallel system it is expected to have a large number of less powerful processors whereas with a coarse grained parallel system, generally, the number of processors is small but each processor tends to be more powerful, e.g. having a 64 bit word size. In the thesis, we shall refer the number of processors as *degree of parallelism*. When the degree of parallelism is high we call it *massively parallel*. In this situation contention is a significant cost factor in *SE* or *SD* whereas overhead plays an important role in *SN*.

[8] Dynamic network is capable of establishing a connection between two or more processors on the fly as messages are routed along the links.

We focus on *SN* as system architecture not only because it has been widely used both for commercial and research purposes, e.g. Gamma, Bubba, Volcano, TANDEM: NONSTOP SQL (Englert, 1989), Teradata DBC/1012, but also because it has the following salient features.

First, *SN* provides the scalability and has little contention problem. In *SN*, the system is easy to scale up to a large number of processors, e.g. there are 16,385 processors in CM5, because the contention in *SN* environment is less important or negligible. By comparison, in *SE*, when the number of processors is increased, the memory contention is also increased so that the overall system performance may even decrease with the addition of processors. In *SD*, the public disk is the bottleneck and the contention problem still exists even with the aid of RAID technology.

Second, *SN* can provide high performance at low cost. By exploiting parallelism, we are aiming at improving performance without extra cost or with low cost so that the improvement is made based on the existing hardware, e.g., we may connect the existing systems together to offer a parallel system with a common user interface such as a network of workstations. With socket level network programming, all stand alone systems are connected. At any time, if one system is idle, i.e. there is no local job, the system is regarded as a candidate worker processor to share the workload of other busy processors. All jobs are assigned a priority and local jobs always have the highest priority at local processors. Moreover, the manufacturing cost of *SN* is also low comparing with *SD* and *SE* since *SD* requires specialised disk controllers and inter connection networks to enable inter processor communication, and *SE* needs specialised hardware to implement tight coupling, e.g. cache coherency.

However, *SN* does have its own problem and the most important issue in *SN* is load balancing since processors are loosely coupled and data are stored locally. Therefore, it is crucial to ensure that workload is partitioned evenly over processors during processing. Inter process communication is also expensive because the processes are running on different sites. *SN* also complicates loading and updating data since data are stored at local sites possibly with replication or fragmentation. It is highly likely that the time for accessing data at one site is much more expensive than those at other sites.

Another drawback with *SN* is the complexity of resource allocation. With the private owned memory, disk and processor in *SN*, algorithm designer must be careful to provide high resource utilisation. In the rest of this thesis, we only consider processors and our objective is to allocate processors to operations in the query in such a way so that the overall query execution time is minimised.

For data placement, assume that the database is distributed over the processors possibly with fragmentation and replication. In parallel query processing, the host accepts queries from users and distributes each query together with the required base relations to the processors for execution. The processors perform the query in parallel with possibly intermediate data transmission between each other through the network, and finally send the result of the query to the host for consolidation.

In Chapters 6 and 7, the degree of parallelism is small and thus communication overhead is ignored. In Chapters 8 and 9, the number of processors is assumed to be large and thus the optimal degree of parallelism can be derived as the combined effects of communicating overhead and computing in parallel.

6.3 Parallel Query Execution Model

In previous chapters, we have focused on modelling and predicting skewness at operation level which is the theoretical foundation of skew and load balancing. In contrast, from Chapters 6 to 9 we will concentrate on query level and consider parallel processing methods to improve performance. A query is specified by a parallel execution plan that may involve a number of relational operations. The plan can be represented by a query tree $Q=(P, A)$, where P is a set of nodes and A is a set of arcs. Each node in P represents an operation, and each arc between two nodes specifies the execution order of the operations and the arrow indicates the data flow. For example, an arc from node p_i to p_j indicates that the operation p_j will use the intermediate result of p_i as an operand and thus it can be processed only after p_i is processed. Node p_i is also called a predecessor of p_j and p_j is a successor of p_i. A node in P may have several incoming arcs and one outgoing arc except the root node which has no outgoing arc and gives the final result of the query. For simplicity, the query plan is

assumed to involve selections, projections and joins (S-P-J query) and an example of such plans is shown in Figure 6.2 with 8 processors.

Associated with each operation node is its allocated processors, processing method used and the execution cost estimated. The execution cost is composed of local processing cost and the communication cost for transferring operand data from the processors where the data are resident to the processing processor. The local processing cost is assumed to be proportional to the sizes of the operand relations while the data transmission cost is proportional to the amount of data transferred.

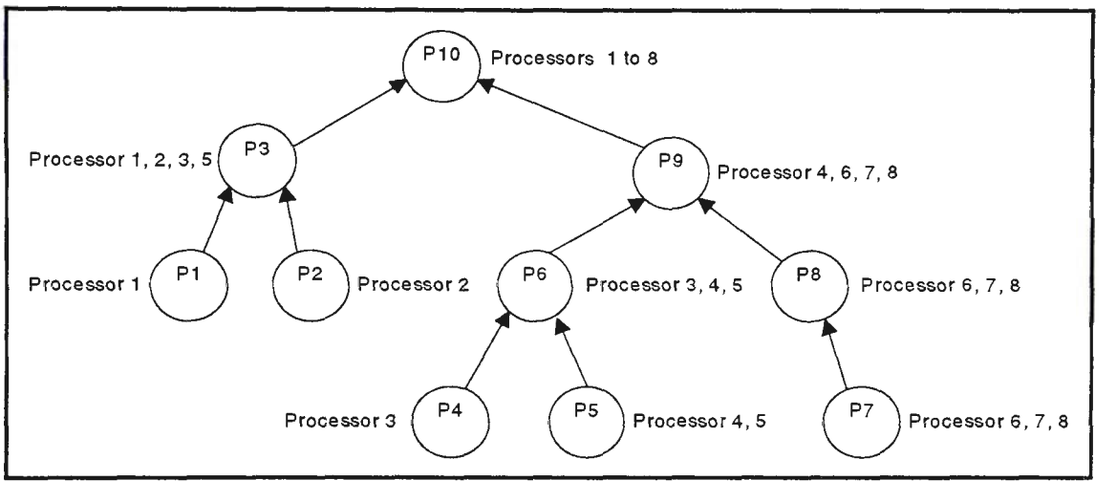


Figure 6.2: An Example of Parallel Query Execution Plan

6.3.1 Skewed Data Partitioning

Using the Zipf distribution, given the number of processors n and a data skew factor θ for an operand relation, the size of the ith fragment may be represented by

$$r_i = \frac{r}{i^{\theta}\sum_{j=1}^{n}\frac{1}{j^{\theta}}}. \qquad (\theta \geq 0) \qquad (6\text{-}1)$$

Clearly, when $\theta = 0$, i.e. no data skew in the relation, the fragment sizes follow a discrete uniform distribution and thus we have $r_i = \frac{r}{n}$. In contrast, when $\theta = 1$ indicating a high data skew, the fragment sizes follow a pure Zipf distribution and, therefore, we have

$$r_i = \frac{r}{i \times \sum_{j=1}^{n} \frac{1}{j}} = \frac{r}{i \times H_n} \approx \frac{r}{i \times (\gamma + \ln n)},$$

where $\gamma = 0.57721$ (Euler Constant) and H_n is the Harmonic number which can be approximated by $(\gamma + \ln n)$. In the case of $\theta > 0$, the first fragment r_1 is always the largest in size whereas the last one r_n is the smallest. (Note that fragment i is not necessarily allocated to processor i.) Therefore, the size of the maximum fragment is

$$\max(r_i) = r_1 = \frac{r}{\sum_{j=1}^{n} \frac{1}{j}} \approx \frac{r}{\gamma + \ln n} \quad .$$

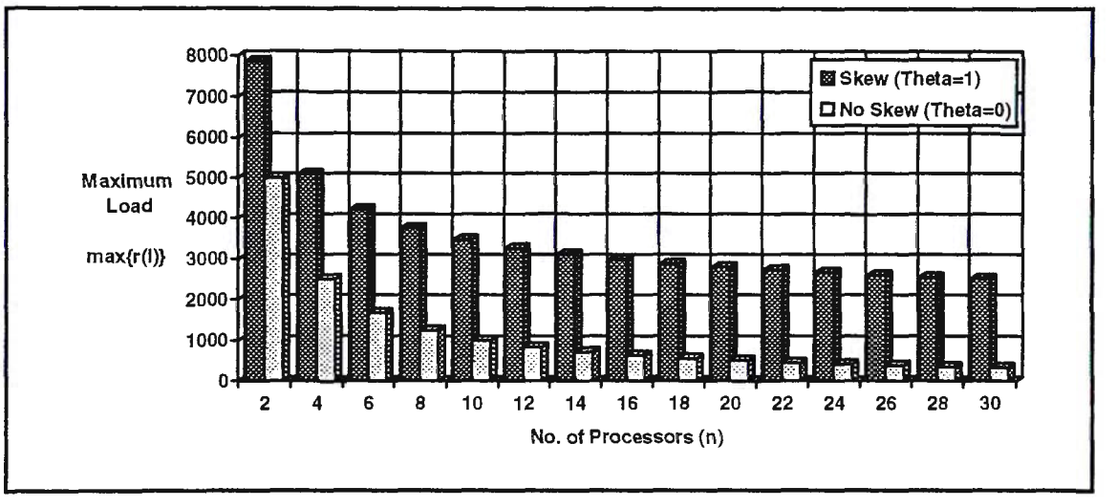


Figure 6.3: Influence of Skew on Maximum Processor Load

Figure 6.3 shows the effect of data skew on the maximum processor load which is measured by the number of tuples allocated. The cardinality of the relation is assumed to be 10,000. It is clear that the maximum processor load in the case of data skew is much higher than that without skew. For example, when the number of processors is 10, the maximum loads for the two cases are 3500 and 1000, respectively. Another interesting point is that the load reduction appears to be significant when the number of processors increases from 2 to 10, and becomes marginal with further increase in the number of processors. This observation indicates that allocating large number of processors to a single operation may not be beneficial, particularly when data skew is involved. This shall be referred to as the *Skew Principle* in parallel query processing.

6.3.2 Parallel Execution of Selection/Projection

The parallel execution of a unary operation is carried out in two steps. First, the operand relation R is partitioned into a number of equal-sized fragments and each processor (possibly with a partial copy) is assigned a fragment. The processors with the fragments then conduct the operation in parallel. The execution time of the operation therefore only consists of the time for initialising the operation and the time of local processing since the partitioning involves simply dividing the relation equally into a given number of fragments whose time is negligible and there is no load skew among the processors. Using the notation in Table 6.1, the execution time can then be expressed as

$$T_{unary} = T_{init} + \max(L_{R(i)}) = T_{init} + \frac{r}{n}(W_1 + W_2 + W_3\sigma_i). \qquad (6\text{-}2)$$

Parameters	Meaning
r, s	the cardinality of relation R and S
n	the number of processors
r_i	the number of tuples in the ith processor after the partitioning of relation R
σ_i	selectivity factor of the ith fragment of the relation R after partitioning
W_1	loading time for each tuple (including disk access time and transfer time)
W_2	processing time for each tuple (mainly comparison and computation time)
W_3	writing time for each tuple
θ	the data skew factor $(\theta \geq 0)$
L	the local operation processing time
T_{init}	time cost for initialising the operation on multiple processors (fixed cost)
T_{hash}	time cost for hashing one tuple
σ_{Join_Sel}	the join selectivity factor
T_{data}	time cost for transferring one tuple

Table 6.1: Notations

6.3.3 Parallel Execution of Joins

We present two partitioning methods for parallel join execution, *hash partitioned join* and *simple range partitioned join*. In the hash partitioned join method, an identical hash function is first used to partition the tuples of each of the join relations into a sequence of fragments based on join attribute values. The hash partitioning ensures that the tuples in a fragment of one relation may only be joined with those in the corresponding fragment of the other relation. Each pair of the corresponding fragments are then allocated to a processor, and the processors allocated with join fragments conduct the join in parallel. Therefore, the execution time of the hash partitioned join includes initialisation time, hash partitioning time, data transmission time as well as local join processing time, i.e.

$$T_{hash\_join} = T_{init} + \beta \max\left(T^i_{hash}\right) + \gamma \sum T^i_{data} + L_{hash\_join}$$

where the parameters β and γ are determined by the overlapped execution time among hashing, data transmission and local join processing. When the join relations originally reside in more than one processor, hash partitioning may be performed by those processors in parallel. The hash partitioning time can then be given as $\max(T^i_{hash}) = \max(r_i + s_i) \times T_{hash}$. The data transmission time is required for the hash partitioning processor(s) to transfer all tuples belonging to the fragments of other processors. The *average* total transmission time is then expressed as $\sum T^i_{data} = \frac{n-1}{n}(r+s)T_{data}$. The local join time is determined by the join method used, such as nested loops join, sort merge join and hash join. The costs of the join methods vary with the index available on the join attributes and the cardinality of join relations, and thus none of them always outperforms the others. Nevertheless, the hash join method is often used in parallel processing since it usually performs best when no index but only the join relations are transferred across the processors. Assuming that the hash join is used, the local processing time involves the time for building a hash table for one relation, probing the table for the other relation and writing the joined tuples into the buffer. Note that because of load skew and operation skew, both the sizes of the fragments allocated to the processors and the sizes of the join results vary with the skew factor θ. Given the equation (6-1) of skewed data partitioning, the largest pair of the fragments allocated to a single processor is given by

$(r+s)/\sum_{j=1}^{n}\frac{1}{j^{\theta}}$ and the corresponding join result size will be $(r\times s\times \sigma_{Join\_Sel})/\sum_{j=1}^{n}\frac{1}{j^{\theta}}$. Therefore, the local join time can be rewritten as

$$L_{hash\_join}=\frac{r+s}{\sum_{j=1}^{n}\frac{1}{j^{\theta}}}\times(W_1+W_2)+\frac{r\times s}{\sum_{j=1}^{n}\frac{1}{j^{\theta}}}\times W_3\times\sigma_{Join\_Sel}.$$

The total execution time of the hash partitioned join may therefore be expressed as

$$\begin{aligned}T_{hash\_join}=&T_{init}+\beta\max(r_i+s_i)T_{hash}+\gamma\frac{n-1}{n}(r+s)T_{data}\\&+\frac{r+s}{\sum_{j=1}^{n}\frac{1}{j^{\theta}}}(W_1+W_2)+\frac{r\times s}{\sum_{j=1}^{n}\frac{1}{j^{\theta}}}W_3\sigma_{Join\_Sel}\end{aligned}. \qquad (6\text{-}3)$$

The simple range partitioned join method is different from the hash join method in that it equally partitions the first join relation into a set of fragments, each assigned to a processor, and broadcasts the entire second relation to every processor. In this method, the time for data partitioning is negligible since it only involves dividing one relation into equal-sized fragments. Moreover, since the join relations may reside fully or partially at more than one processor, the data transmission is needed only for the processors which do not have the fragments assigned to them. In other words, data transmission time depends on how many fragments of the partitioned relation need to be transferred and whether the other relation needs broadcast, and the time cost is given by $\sum_i T^i_{data}=(\sum_{R_i \text{ not in } s_i}|R_i|+\alpha|S|)T_{data}$, where α is in the range (0, 1) based on data stored at local processors. For each processor that conducts the join, the local processing time involves joining the tuples of one fragment of a relation with all tuples in the other relation. Given the notation and assumption in Table 6.1, the local join time can be expressed as

$$L_{hash\_join}=\max(T^i_{local})=(\frac{r}{n}+s)(W_1+W_2)+\frac{r\times s}{n}W_3\sigma_{Join\_Sel}.$$

Therefore, the total execution time for the simple range partitioned join is given by

$$T_{range\_join} = T_{init} + \gamma \sum T^{i}_{data} + L_{hash\_join}$$

$$= T_{init} + \gamma \left(\sum_{r_i \text{ not in } s_i} r_i + \alpha s \right) T_{data} + \left(\frac{r}{n} + s \right) (W_1 + W_2) + \frac{r \times s}{n} W_3 \sigma_{Join\_Sel} \quad \text{(6-4)}$$

On the surface, the execution time of the simple range partitioned method appears larger than that of the hash partitioned method mainly because of its much larger local join time. However, since the fragments in the simple range partitioned method are always equally partitioned, there is no load skew among the processors. In other words, the execution time is not affected by the data skew factor and hence would be smaller as compared to that of the hash partitioned method when the data skew is high.

6.4 Processor Allocation for Parallel Query Processing

Processor allocation deals with efficient assignment of the processors to the operations involved in a query execution plan such that the query response time is minimised. The minimisation of the query response time is achieved by exploiting a number of parallel processing strategies according to different database architectures and the characteristics of the queries. We first present in this section two simple processor allocation methods, *intra-parallel processor allocation (IPA)* and *phase-based processor allocation (PPA)*. A new method that attempts processor allocation adaptively in accordance with the load skew is then presented.

6.4.1 Intra-Parallel Processor Allocation (IPA)

In the *IPA* method, the operations in a given query plan are carried out one after another, starting from the leaf operations that need only base relations to the root operation that produces the query result. For each of the operations, parallel processing is exploited by partitioning and distributing operand relation(s) over all available processors, followed by execution of the operation in parallel.

When the operand relations of each operation are uniformly distributed to the processors, i.e. no load skew, maximum speedup of the operation is achieved since no processors are idle when others are busy working. However, if load skew occurs, some processors may have heavier load than the others and require more time to complete the part of the operation assigned. The completion time of the whole operation therefore would be much higher than expected since it is determined by the time required for the heaviest loaded processor. Moreover, as shown in Figure 6.3, in the case of high load skew, the heaviest load over the processors reduces only marginally when the number of the processors is large, indicating that allocating a large number of processors does not help the reduction of execution time of the operation.

6.4.2 Phase-based Processor Allocation (PPA)

The *PPA* method follows a phase-oriented paradigm by which the operations of a query plan are performed by several execution phases. The first phase involves the operations that require only base relations and thus are ready to process. The next phase may then contain the operations that become ready to process after completion of the first phase, and so on. The last phase produces the result of the query. Within an execution phase, each of the operations is allocated to one or more processors such that all operations in the phase are processed in parallel and are expected to complete at about the same time. The details of this method is given in [Leun93].

As compared with the *IPA* method, the *PPA* method would achieve larger speedup of the query processing in the existence of the load skew. For example, consider two operations that have the identical processing cost and the data skew factor θ=1. If the two operations can be processed in the same execution phase with 10 processors each, the maximum processor load as shown in Figure 6.3 is equivalent to processing about 3500 tuples. In contrast, if the two operations are processed one after another applying pure intra-operation parallelism, the total maximum processor loads increase up to 2800×2 tuples assuming 20 processors used for each operation. Nevertheless, a drawback of the phase-based processor allocation method is that the processors allocated with different operations within an execution phase may not always complete the execution at about the same time, reducing the processor utilisation even though there is no load skew. The low processor utilisation

affects the speedup of the query execution particularly when the number of processors is limited [Jian95].

It may also be noted that the *PPA* method often involves less data transmission cost than the *IPA* method, especially when the base relations are replicated over the processors. For example, we consider two join operations that have the same cost and can be processed in parallel. By applying both inter-operation and intra-operation parallelism, each operation may be allocated to half of the processors, denoted by m. Assuming hash partitioned join method used, the portion of the total join relations to be transferred will be $1-\frac{1}{m}$ since one out of m fragments in each operation need not be transferred. However, if only intra-operation parallelism is used, the two join will be processed one after another by $2m$ processors. The portion of the join relations to be transferred is then increased by $\frac{1}{2m}$. Such an increase is not negligible when the number of processors is not extremely large and the communication cost over the network is comparable to the local processing cost.

6.4.3 Modified Phase-based Processor Allocation (MPA)

The *MPA* method attempts to improve the efficiency of parallel query processing in the presence of load skew. As noted from the description of the *IPA* and *PPA* methods, the intra-operation parallelism performs well when there is no load skew and the data transmission cost is small as compared to local join cost; otherwise the *PPA* method may outperform the *IPA* method. Although naturally the *PPA* method is less affected by the skewness, neither of the methods pay special attention to the skew problem. The idea behind the *MPA* method is that if an operation is expected to involve load skew, it should be processed together with some other operations in one execution phase since the inter-operation parallelism may reduce the negative influence of the load skew on the performance improvement. The heuristic implemented in the *MPA* method is neighbourhood search and the method is a phase-based approach aiming at local improvement, i.e. in each execution phase.

For instance, consider the simple query with 5 join operations and 6 relations with query tree shown in Figure 6.4. In the figure, the numbers after the base relations refer to the

relations cardinality and the numbers beside the operation nodes are related to the number of tuples output from this join operation. Certainly, the output tuples are affected by the join selectivity and it is reasonable to assume that the number of tuples output from a join operation is proportional to the maximum relation cardinality of the operand relations. With the *IPA* method, the execution result is shown in Table 6.2, and as we predicted that there are 5 execution phases and 12 processors for each join operation. With the *PPA* method, there are only three execution phases and P1, P2, and P4 will be grouped into the first execution phase. The third column of Table 6.3 shows the number of processors allocated for each operation and 2, 3, 7 means the allocation result in this phase is P1 with 2 processors, P2 with 3 processors, and P4 with 7 processors. From the fourth column, we can see that although the total execution time for the phase is 85.1, the first operation is finished much early with a execution time of 50.8.

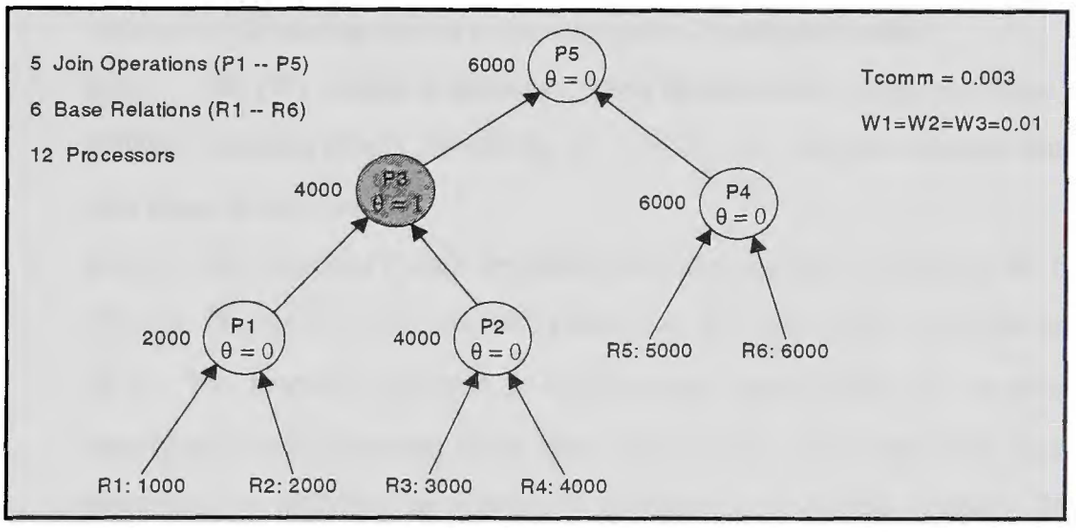


Figure 6.4: An Example of Query Tree in the Presence of Skew

Phase Number	Operation Number	No. of Processors	Execution Time
1	P1	12	17.4
2	P2	12	40.1
3	P3	12	91.4
4	P4	12	62.8
5	P5	12	61.1
	Total Execution Time		272.8

Table 6.2: Execution Time for *IPA*

In addition, with the two partitioning methods, the simple range partitioned join method that is insensitive to the load skew should be attempted for the highly skewed operations, hopefully leading to less execution times than that of using hash partitioned join.

Phase Number	Operation Number	No. of Processors	Execution Time
1	P1, P2, P4	2, 3, 7	85.1 (50.8, 85.1, 79.4)
2	P3	12	91.4
3	P5	12	61.1
	Total Execution Time		237.6

Table 6.3: Execution Time for *PPA*

The *MPA* method is outlined as follows:

- Step 1. Given a query execution plan, the load skew for each of the operations in the plan is estimated according to the equation (6-1) presented earlier.
- Step 2. The *PPA* method is applied to cluster the operations of the query plan into different execution phases, denoted by H_i, i=1,2,.....,m, and the execution time of each phase is calculated.
- Step 3. The execution phases are checked one after another in reverse order from H_{m-1} to H_2 (there is only one root operation in H_m and all leaf operations are in H_1). The heuristic employed is neighbourhood search since the number of operations in each execution phase after Step 2 is in a decreasing order and the objective is to distribute the number of operations in each phase evenly. If the current phase, say H_i, has at least one skewed operation, consider re-clustering the operations in the current phase H_i and its precedent phase H_{i-1} into new execution phases, that is:

  ⇒ a. Find all possible groups of the operations from H_i and H_{i-1} that can be executed in parallel.

  ⇒ b. For each possible group H_i', calculate the new phase execution time $T(H_i')$.

  ⇒ c. Given H_i', re-calculate the execution times for the rest of the operations in the phases H_{i-1} and H_i, i.e. $T(H_{i-1} - H_i')$ and $T(H_i - H_i')$. (Note that $H_{i-1} - H_i'$ and $H_i - H_i'$ might be empty groups.)

$\Rightarrow$ d. If $\left[T(H_i')+T(H_{i-1}-H_i')+T(H_i-H_i')\right] < \left[T(H_i)+T(H_{i-1})\right]$, i.e. the new phase partitioning leads to less execution time than that of the existing phases, the existing phases are replaced by the new one.

It may be noted that in step 3 the execution phases from H_{m-1} down to H_2 are examined since no reformulation of the execution phases is possible for the first and the last phases. Moreover, the modification is carried out in the reverse order of the phases because the late phases are likely to have less number of operations as the query plan in the form of tree, and thus have less feasibility to be improved if some of their operations are highly load skewed. In addition, the *MPA* method attempts to improve the query execution time by altering only the adjacent execution phases. The minimum query execution time would be found but is not practical because of the time complexity of the exhaustive search.

From the query in Figure 6.4, using the proposed method *MPA*, the execution results are shown in Table 6.4. We notice the number of operations in the first and second phases is changed with two operations in each phase and the reason is P3 is a skewed operation and is not preferred to stay in an execution phase itself. Therefore, based on *MPA*, we will do a checking and then find out that grouping P3 and P4 gives better results. Consequently, the execution phase is reformed with P3 and P4 in the second execution phase and P1 and P2 in the first phase[9]. Moreover, comparing each phase's execution time to each operation's execution time within the phase, we can conclude that the new algorithm offers much better processor utilisation.

Phase Number	Operation Number	No. of Processors	Execution Time
1	P1, P2	3, 9	45.1 (37.4, 45.1)
2	P3, P4	6, 6	92.8 (92.8, 86.1)
3	P5	12	61.1
	Total Execution Time		199.0

Table 6.4: Execution Time for *MPA*

[9] Sometimes, the number of total execution phases for one query may also be changed with the possibility of eliminating or inserting the execution phase.

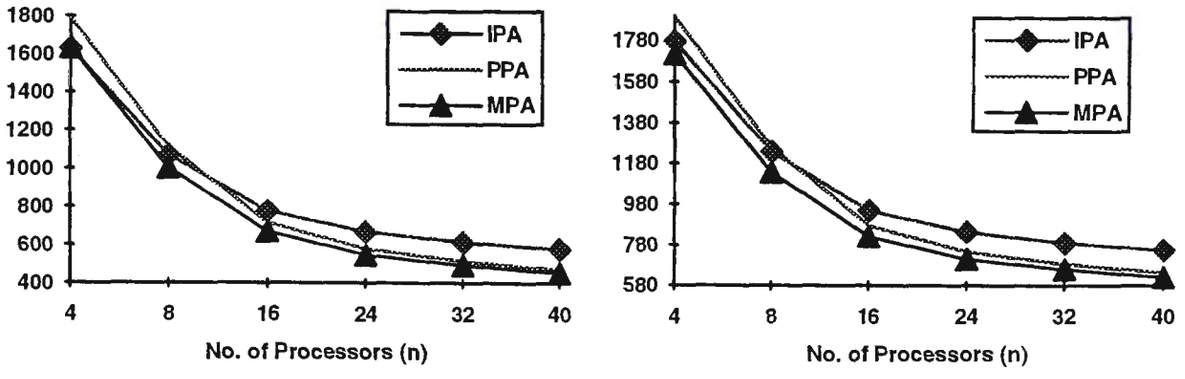
6.5 Performance Evaluation

A simulation study has been conducted to evaluate the performance of the above processor allocation methods. In the simulation, the processors are assumed to be identical in processing capacities and are connected by a cross-bar network. For the sake of simplicity, the database relations are assumed to be stored over the processors with either non replication or full replication. The former allows a relation to be available at only one processor, while the latter assumes that the relation is available at every processor.

Since joins are the most time-consuming operations and form the main target of parallelisation, we have used a set of join queries in the simulation. Each query involves 5 to 10 joins and is performed on the base relations with cardinality varying from 1000 to 10,000 tuples. The execution times of the queries are calculated according to the cost equations given in Section 6.3 and the default parameter settings are given in Table 6.5 [Liu96a]. The simulation is implemented using C language and runs on a Sun workstation. A number of test runs were conducted and the results of the simulation are presented below.

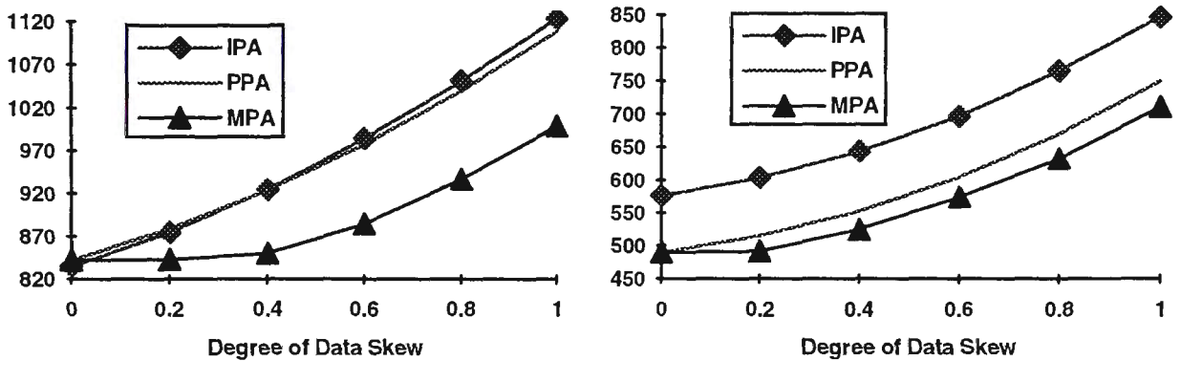
Parameters	Values
number of processors n	4 to 40
Number of base relations	6 - 12
Data replication	full, optionally no
T_{data}	0.003 (time unit per tuple)
T_{hash}	0.01 (time unit per tuple)
W_1, W_2, W_3	0.01 (time unit per tuple)
T_{init}	0.1 (time unit per tuple)
Overlapping factors β, γ	1.0
Number of test queries	10
Number of joins per query	5 - 11
Cardinality of relations (r, s)	1000 - 10,000
Data skew factor θ	0 - 1.0
Percentage of joins with data skew	20% -30%
Partitioning Methods	hash partitioning, optionally simple range partitioning

Table 6.5: Default Parameter Settings



(a) $\theta = 0.5$ (b) $\theta = 1.0$

Figure 6.5: Query Execution Time vs Number of Processors



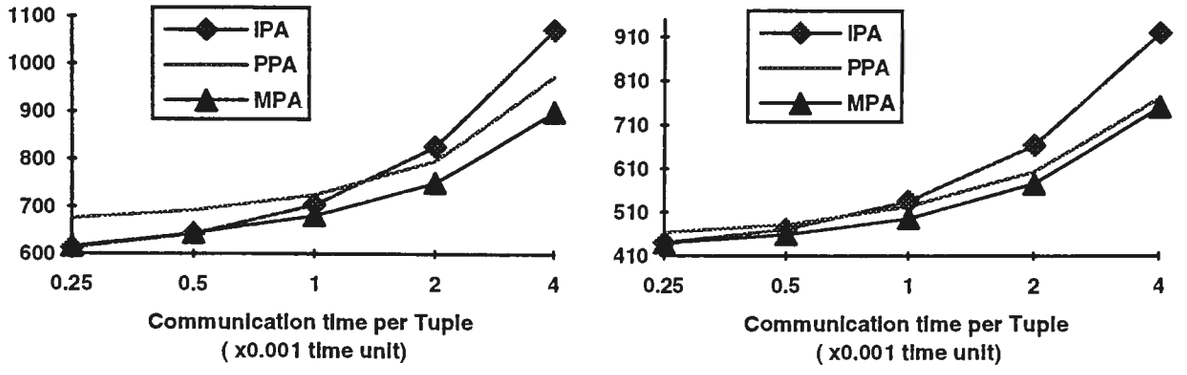
(a) 10 processors (b) 24 processors

Figure 6.6: Query Execution Time vs Data Skew Factor

6.5.1 Query Execution Time vs Number of Processors

Figure 6.5 shows the average query execution times of the three processor allocation methods, *IPA*, *PPA* and *MPA*, when the number of processors varies from 4 to 40. A full data replication was assumed and the data skew factor took values of 0.5 and 1.0. Firstly, it is shown that for all three methods the query execution times are reduced significantly along with the increase of processors, indicating performance gain from parallelisation. Secondly, when the degree of data skew is moderate and the number of processors is small, the *IPA* method appears to perform better than the *PPA* method. With the increase in the processors and/or the data skew factor, however, the *PPA* method becomes outperforming the *IPA* method. This is not surprising because it is noted early that the *IPA* method is sensitive to the degree of load skew and often involves large data transmission cost as compared to the *PPA* method especially when the queries are distributed over many processors. Finally,

Figure 6.5 demonstrates that the *MPA* method consistently performs better than the other two methods no matter if the data skew is low or high. This supports our claim that assigning more operations into the execution phases with high data skew may reduce the negative skew effect on the overall query execution time.



(a) 16 processors (b) 32 processors

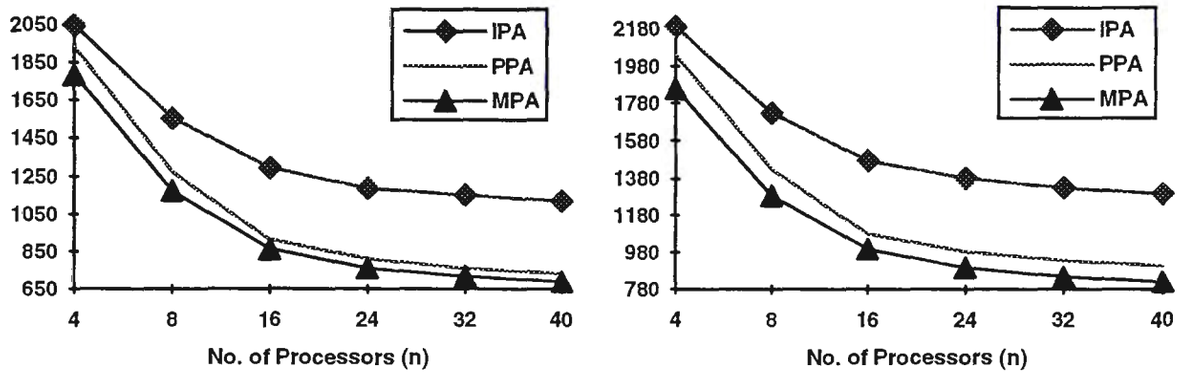
Figure 6.7: Effect of Increasing Communication Time

6.5.2 Query Execution Time vs Data Skew Factor

Figure 6.6 shows how the query execution time changes with the data skew factor θ when 10 and 24 processors were used. Clearly, a large value of θ and thus high data skew leads to significant increase in the query execution time. For example, the query execution times shown in Figure 6.6(a) are increased by about 34%, 29% and 17% for the *IPA*, *PPA* and *MPA* methods when θ changes from to 0 to 1.0. In addition, as a comparison of the three methods, a similar trend to Figure 6.5 is shown here, i.e. the *PPA* method outperforms the *IPA* method except when the data skew is low and the number of processors is small, whereas the new *MPA* method is always superior. It is also worth noting that in Figure 6.6(b) the *IPA* method appears to lead much large query execution time as compared to the *PPA* method. This is again because the sole intra-operation parallelism attempts to distribute every operation of the query to all processors and thus involves larger data transmission time than the *PPA* method. When there are a number of processors, the local join times for both methods are reduced significantly and hence the data transmission time becomes dominant in the total query execution time. Therefore, the large time difference between the two methods occurs.

6.5.3 Effect of Increasing Communication Time

The performance of the processor allocation methods presented may vary with different hardware platforms. One important hardware parameter is the ratio of the communication bandwidth to the processor speed. We have conducted test runs by varying the communication time such that the data transmission time took different portion in the total query time. The results are shown in Figure 6.7 and indicate that no matter what ratio was assumed, the trend of the performance of the three methods remains unchanged. The *MPA* method always performs best while there is crossover point for the *IPA* and *PPA* methods.

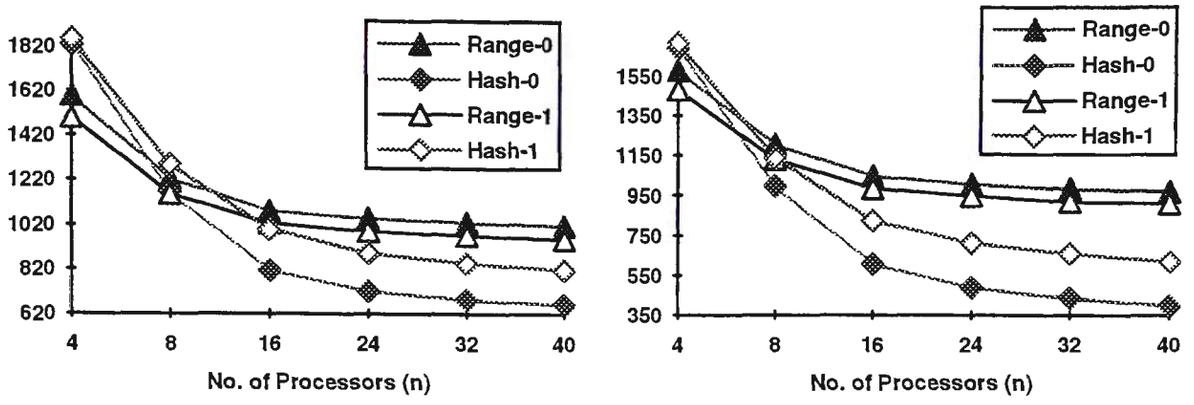


(a) $\theta = 0.5$ (b) $\theta = 1.0$

Figure 6.8: Query Execution Time with No Database Replication

6.5.4 Non Data Replication vs Full Data Replication

Test runs were also conducted assuming non replication of base relations and the results are shown in Figure 6.8. With non replication, each relation is stored at one and only one processor and thus, the data transmission time tends to increase and dominate the total query execution time. As a consequence, the *IPA* method shows poor performance in all cases. In contrast, the impact of data replication appears small to the *PPA* and the *MPA* methods. In other words, these two methods have steady performance over a wide range of data replication.



(a) Non replication of database (b) Full replication of database

Note: In the legend, 0 means $\theta = 0$ and 1 refers to $\theta = 1.0$

Figure 6.9: Comparison of Hash Partitioned Join Method and Simple Range Partitioned Method

6.5.5 Hash Partitioned Join vs Simple Range Partitioned Join

In the previous test runs, we have used hash partitioning as underlying join processing method since it is expected to perform well in most of cases. However, another partitioning method described in Section 3, simple range partitioning, may also perform well when there is a high skew in the query processing. We have compared the two partitioning methods in the simulation and the results shown in Figure 6.9 validate our expectation. Generally, the simple range partitioned method only works well when the degree of the data skew is high and/or the number of processors is small. However, an interesting point is that the improvement of the simple range partitioned method is, as expected, insensitive to the data skew factor. This indicates that the simple range partitioned method may be a good choice when the load skew is unknown but highly possible. Another point is that the hash partitioned method tends to gain more improvement as the number of processors increases, while the simple range partitioned method only gives marginal improvement. The main reason is due to the fact that the hash method partitions both join relations into fragments but the range method partitions only one relation and broadcasts the other over all processors. Since allocating more processors does not reduce the size of the broadcast relation, the improvement obtained by the range method approaches a limit no matter how many more processors are allocated.

6.6 Summary

In this chapter, we analyse the performance of parallel query processing in the presence of data skew. The effects of data skew on processor allocation are studied and it is found that having a large number of processors does not improve the overall execution time significantly in the presence of data skew (the *Skew Principle*). Cost models for both unary and binary operations are also presented with the data skew factor modelled by the Zipf distribution under different data partitioning methods. Two basic processor allocation algorithms, namely *IPA* and *PPA*, are studied and compared, one using sole intra-operation parallelism and the other using phase-oriented inter-operation parallelism. A new efficient algorithm, *MPA*, which is found to outperform the two basic ones, especially in the cases of high data skew, is presented.

The performance of the processor allocation algorithms as well as the data partitioning methods are evaluated by detailed simulation experiments. It is shown that the new algorithm *MPA* always performs best for a wide range degree of data skew. The *IPA* method appears to be better than the *PPA* method only when the data skew is low and the number of processors is limited, otherwise the *PPA* would be a better choice. The simple range partitioning method only works well when the number of processors is small despite its insensitivity to skewness. It is also found that the effects of communication time and idle time (closely associated with data skew) are critical in limiting the extent of performance improvement in parallel query execution.

CHAPTER 7

PARALLEL PROCESSING OF AGGREGATE FUNCTIONS IN THE PRESENCE OF SKEW

7.1 Introduction

In recent years, a new approach to improve the performance of database systems is data warehouse which is a repository of integrated information, available for querying and analysis [Inmo96]. With the growing number of large data warehouses for decision support applications, efficiently executing aggregate functions is becoming increasingly important. In data warehouse, large historical tables are usually joined with other tables and

aggregated, so better optimisation of aggregate functions has the potential to result in huge performance gains.

The parallelisation of query processing can be conducted at intra-operation level, inter-operation level or a combination of both, and Section 2.1.3 provides a discussion on forms of parallelism. Although parallel processing of major relational operations (e.g. selection, projection and join) has been studied extensively, parallel aggregation receives much less attention in spite of the fact that it is critical to the performance of database applications such as decision support, quality control, and statistical databases. Aggregation may be classified into *scalar aggregate* and *aggregate function* [Bitt83, Grae93]. The former refers to the simple aggregation that produces a single value from one relation such as counting the number of tuples or summing the quantities of a given attribute; while the latter refers to those that cluster the tuples of the relation(s) into groups and produce one value for each group. The queries with aggregate functions often involve more than one relation, and thus require join operations. The issues on parallel processing scalar aggregate has been studied in [Shat94] for locally distributed databases.

Join before aggregation is the conventional way for processing aggregate functions in uni-processor systems and parallel processing of aggregate functions has received little attention. This chapter concentrates on the issues of aggregate functions and investigates efficient parallel processing methods for queries involving aggregations and joins[10]. Three methods, namely, *join-partition method (JPM), aggregation-partition method (APM)* and *hybrid-partition method (HPM),* are presented. *JPM* and *APM* mainly differ in the selection of partitioning attribute for distributing workload over the processors and *HPM* is an adaptive method based on *APM* and *JPM* with a logical hybrid architecture. Furthermore, all methods take into account the problem of data skew since the skewed load distribution may affect the query execution time significantly. The performance of the parallel aggregation methods are compared under various queries and different environments with a simulation study, and the results are also presented.

In the next section, we discuss the critical issues on parallelising aggregate functions namely, the selection of partitioning attribute, the sequence of aggregation and join

---

[10] Hereafter, aggregation means simple aggregation operation (e.g. AVG and SUM on an attribute in one relation) while aggregate function (or aggregation query) consists of aggregation and join operation.

operation, and the skewness. The parallel processing methods and their cost models are introduced in Section 7.3 followed by a sensitivity analysis in Section 7.4.

7.2 Parallelising Aggregate Functions

For simplicity of description and without loss of generality, we consider queries that involve only one aggregation function and a single join. The example queries given below arise from a *Suppliers-Parts-Projects* database. The first query clusters the part shipment by their city locations and selects the cities with average quantity of shipment between 500 and 1000. The second query retrieves the project number, name and the total quantity of shipment for each project.

SUPPLIER (**s#, sname, status, city**)
PARTS (**p#, pname, colour, weight, price, city**)
PROJECT (**j#, jname, city, budget**)
SHIPMENT (**s#, p#, j#, qty**)

Query 7.1: **SELECT** parts.city, AVG(qty)
FROM parts, shipment
WHERE parts.p#=shipment.p#
GROUP BY parts.city
HAVING AVG(qty)>500 AND AVG(qty)<1000;

Query 7.2: **SELECT** project.j#, project.jname, SUM(qty)
FROM project, shipment
WHERE project.j#=shipment.j#
GROUP BY project.j#, project.name
HAVING SUM(qty)>1000;

Based on the architecture presented in Chapter 6, an aggregation query is carried out in three phases:

- *Data partitioning*, the operand relations of the query are partitioned and the fragments are distributed to each processor;

- *Parallel processing,* the query is executed in parallel by all processors and the intermediate results are produced;
- *Data consolidation*, the final result of the query is obtained by consolidating the intermediate results from the processors.

7.2.1 Selection of Partition Attribute

Choosing proper partition attribute is a key issue in the above procedure. Although in general any attributes of the operand relations may be chosen, two particular attributes, i.e. join attribute and group-by attribute, are usually considered (e.g. *p#* and *city* in the first example query). If the join attribute is chosen, both relations can be partitioned into N fragments using either range partitioning or hash partitioning strategy, where N is the number of processors. The cost for parallel join operation can therefore be reduced by a factor of N^2 as compared with a single processor system. However, after join and local aggregation at each processor, a global aggregation is required at the data consolidation phase since the local aggregation is performed on a subset of the group-by attribute. In contrast, if the group-by attribute is used for data partitioning, the relation with the group-by attribute can be partitioned into N fragments while the other relation needs to be broadcast to all processors in order to perform the join, leading to a reduction in join cost by a factor of merely N. Although, in the second method, the join cost is not reduced as much as in the first method, no global aggregation is required after local join and aggregation at each processor because the tuples with identical values of the group-by attribute have been allocated to the same processor. Assuming that there are indexes on the join attribute and 4 processors, and the execution time is given in terms of the number of tuples processed in the absence of data skew, Figure 7.1 and Figure 7.2 illustrate the execution time of two types of partitioning strategies on the first example query.

7.2.2 Sequence of Aggregation and Join Operation

When the join attribute and group-by attribute are the same as shown in the second example query (i.e. *j#*), the selection of partitioning attribute becomes obvious. Instead of performing join first, the aggregation would be carried out first followed by the join since the join is more expensive in cost and it would be beneficial to reduce the join relation sizes by

applying aggregation first. Generally, aggregation should always precede join whenever it is possible with the exception that the size reduction gained from aggregation is marginal or the join selectivity factor is extremely small. Figure 7.3 shows that by applying aggregation first, the execution time of the second example query is much lower than that of join operation first as shown in Figures 7.1 and 7.2. However, aggregation before join may not always be possible, and the semantic issues on aggregation and join and the conditions under which the aggregation would be performed before join can be found in [Kim82, Daya87, Bult87, Yan94]. In the following sections, we assume the more general case where aggregation can not be executed before join. Since earlier aggregation reduces execution time, we process aggregation before join if it is possible.

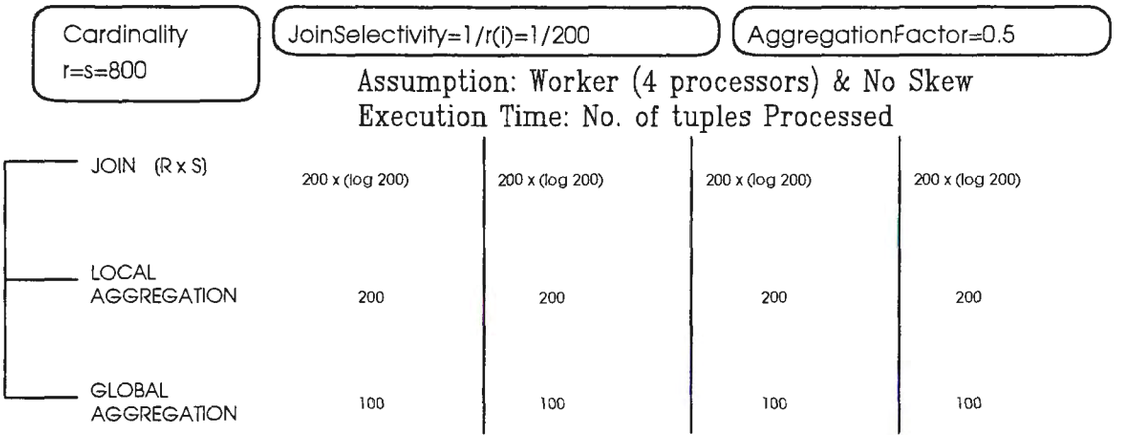


Figure 7.1: Join-Partition method

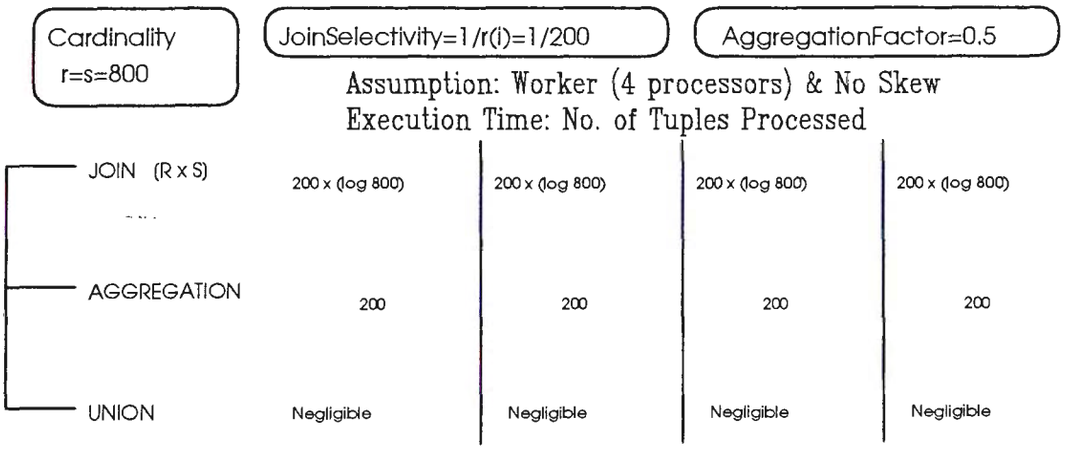


Figure 7.2: Aggregation-Partition method

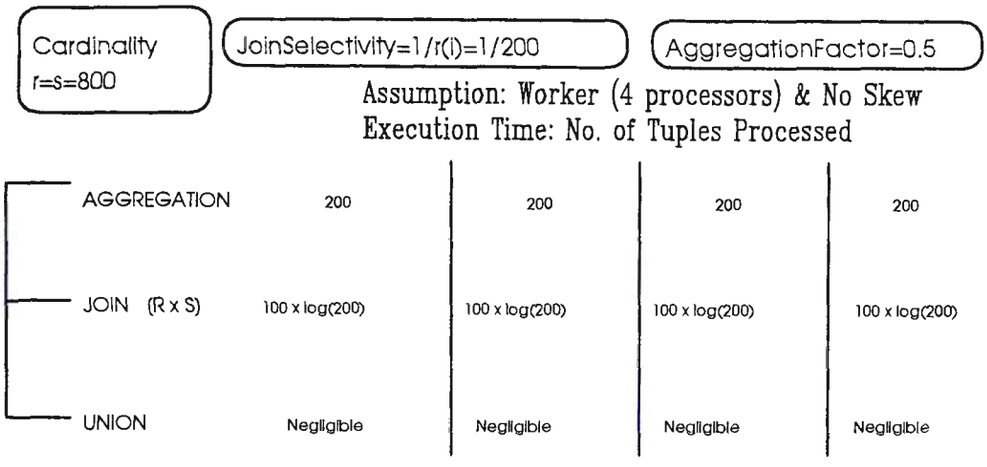


Figure 7.3: Aggregation Before Join

7.3 Three Parallel Processing Methods

We present in this section three parallel processing methods for queries that involve joins and aggregation functions. The notations used in the description of the methods and in the subsequent performance evaluation are given in Table 7.1.

Parameters	Meaning
N	the total number of processors
m	the number of processor clusters
n	the number of processor in each cluster ($N = m \times n$)
r, s	the number of tuples in base relations R and S
r_i, s_i	the number of tuples of fragments of relations R and S at processor i
$Sel(i)$	join selectivity factor for fragment i
$Agg(i)$	aggregation factor for fragment i
θ	reduction factor after performing *Having* clause
α	data partitioning skew factor
β	data processing skew factor
T_{comm}	the average data transmission time for each message
T_{join}	the average join time for each tuple
T_{agg}	the average aggregation time for each tuple
z	message size in terms of the number of tuples

Table 7.1: Parameters Listing of Parallel Processing of Aggregate Functions

7.3.1 Join Partitioning Method (JPM)

The *JPM* can be stated as follows. First, the relations R and S are partitioned into N fragments based on the join attribute, i.e. the tuples with the same join attribute values in the two relations fall into a pair of fragments. Each pair of the fragments will be sent to one processor for execution. Upon receipt of the fragments, in parallel, the processors perform the join operation and then the local aggregation operation on the fragments allocated. After that, a global aggregation operation is carried out by re-distributing the local aggregation results across the processors such that the result tuples with identical values of group-by attribute are allocated to the same processors. Then, each processor performs a N-way merging with the local aggregation results, followed by doing a restriction operation for the *Having* clause if exists at local processors. Finally, the host simply consolidates the partial results from the processors by a union operation, and produces the query result.

The execution time for the *JPM* method can be expressed as follows

$$\begin{aligned} JPM = \; & T_{comm} \times \left(\max\left(r_i + s_i\right)\right) + T_{join} \times \left(\max\left(r_i \log s_i\right)\right) + T_{agg} \times \left(\max\left(r_i \times s_i \times Sel(i)\right)\right) \\ & + T_{comm} \times \left(\max\left(r_i \times s_i \times Sel(i) \times Agg(i)\right)\right) \\ & + T_{agg} \times \left(\max\left(r_i \times s_i \times Sel(i) \times Agg(i)\right)\right) \times (1+1). \end{aligned} \tag{7-1}$$

The maximum execution time for each of the components in the above equation varies with the degree of skewness, and could be far from the average execution time. Therefore, we introduce two skew factors α and β to the above cost equation, and α describes the data partitioning skew while β represents the data processing skew. Assume that α follows the Zipf distribution as in equation (6-1). Recall that the first element p_1 always gives the highest probability and the last element p_N gives the lowest. Considering both operand relations R and S use the same number of processors and follow the Zipf distribution, the data partitioning skew factor α thus can be represented as

$$\alpha = \alpha_r = \alpha_s = \frac{1}{H_N} = \frac{1}{\gamma + \ln N},$$

where $\gamma = 0.57721$ is Euler's Constant, and N is the number of processors.

The other skew factor β for data processing skew is affected by the data partitioning skew factors in both operand relations since the join/aggregation results rely on the operand fragments. Therefore, the range of β falls in $[\alpha_r \times \alpha_s,\ 1]$. However, the actual value of β is difficult to estimate because the largest fragments from the two relations are usually not allocated to the same processor, resulting the β much less than the product of α_r and α_s. We assume in this chapter $\beta = (\alpha_r \times \alpha_s + 1)/2 = (\alpha^2 + 1)/2$.

Applying the skew factors to the above cost equation, we also make the following assumptions and simplifications:

- $J_i = r_i \times s_i \times Sel(i) = J$ i.e. in the absence of the skewness,
- $Agg(i) = Agg$,
- $T_{join} = T_{agg} = T_{proc}$,
- data transmission is carried out by message passing with a size z.

The cost equation (7-1) can then be re-written below

$$
\begin{aligned}
JPM = \; & T_{comm}\left\{\left[\alpha(r+s)+\frac{J\times Agg}{\beta}\right]/z\right\} \\
& +T_{proc}\left[\alpha r\times\log(\alpha s)+\frac{J}{\beta}+\frac{(1+1)\times J\times Agg}{\beta}\right] \\
= \; & T_{comm}\left[\left(\frac{r+s}{\gamma+\ln N}+\frac{2\times(\gamma+\ln N)^2}{1+(\gamma+\ln N)^2}\times J\times Agg\right)/z\right] \\
& +T_{proc}\left(\frac{r}{\gamma+\ln N}\log\left(\frac{s}{\gamma+\ln N}\right)+\frac{2\times(\gamma+\ln N)^2}{1+(\gamma+\ln N)^2}\times J\times(1+2\times Agg)\right).
\end{aligned}
$$

7.3.2 Aggregation Partitioning Method (APM)

In the *APM* method, the relation with the group-by attribute, say *R*, is partitioned into *N* fragments in terms of the group-by attribute, i.e. the tuples with identical attribute values

will be allocated to the same processor. The other relation S needs to be broadcast to all processors in order to perform the binary join. After data distribution, each processor first conducts the joining one fragment of R with the entire relation S, followed by the group-by operation and *having* restriction if exists on the join result. Since the relation R is partitioned on group-by attribute, the final aggregation result can be simply obtained by an union of the local aggregation results from the processors, i.e. the step of merging of the local results used in *JPM* method is not required. Consequently, the cost of the *APM* is given by

$$\begin{aligned} APM = \; & T_{comm} \times \left(\max\left(r_j + s\right)\right) + T_{join} \times \left(\max\left(r_j \log s\right)\right) \\ & + T_{agg} \times \left(\max\left(r_j \times s \times Sel(j)\right) \times \left(1 + Agg(j)\right)\right) \\ & + T_{comm} \times \left(\max\left(r_j \times s \times Sel(j) \times Agg(j) \times \theta\right)\right). \end{aligned} \tag{7-2}$$

The skew factors α and β can be added to the above equation in the same way for the *JPM* method. For the purpose of comparison of the two methods, we assume that $J_j = r_j \times s \times Sel(j) = J$ and $Agg(j) = \frac{1}{N} Agg$. The time of *APM* method can then be expressed as

$$\begin{aligned} APM &= T_{comm}\left\lceil\left(\alpha r + s + \frac{J \times Agg \times \theta}{\beta \times N}\right)/z\right\rceil + T_{proc}\left\lceil(\alpha r) \times \log s + \frac{J}{\beta} \times \left(1 + \frac{Agg}{N}\right)\right\rceil \\ &= T_{comm}\left\lceil\left(\frac{r}{\gamma + \ln N} + s + \frac{2(\gamma + \ln N)^2}{1 + (\gamma + \ln N)^2} \times \frac{J \times Agg \times \theta}{N}\right)/z\right\rceil \\ &\quad + T_{proc}\left(\frac{r}{\gamma + \ln N} \log s + \frac{2(\gamma + \ln N)^2}{1 + (\gamma + \ln N)^2} \times J \times \left(1 + \frac{Agg}{N}\right)\right). \end{aligned}$$

7.3.3 Hybrid Partitioning Method (HPM)

The *HPM* method is a combination of the *JPM* and *APM* methods. In the *HPM*, the processors are divided into m clusters each of which has N/m processors as shown in Figure 7.4. Based on the proposed logical architecture, the data partitioning phase is carried out in

two steps. First, the relation with group-by attributes is partitioned into processor clusters in the same way as the *APM*, i.e. partitioning on the group-by attribute and the other relation is broadcast to the cluster. Second, within each cluster, the fragments of the first relation and the entire broadcast relation is further partitioned by the join attributes as the *JPM* does. Depending on the parameters such as the cardinality of the relations and the skew factors, a proper value of m will be chosen such that the minimum query execution time is achieved.

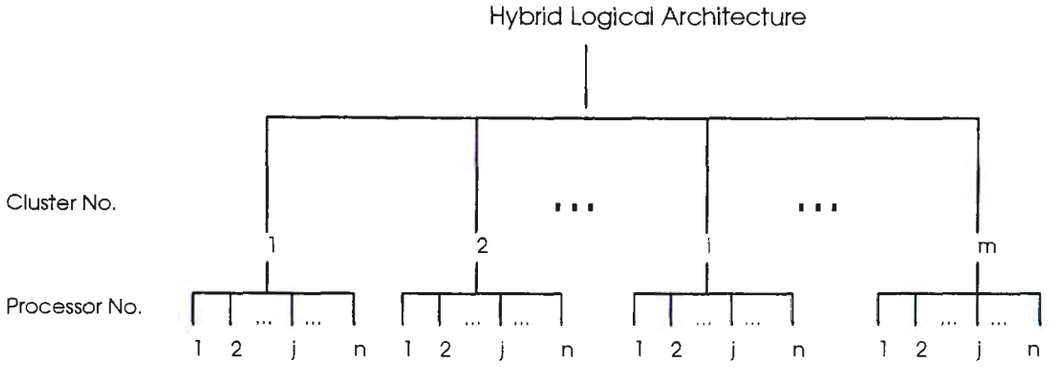


Figure 7.4: Logical Architecture for *HPM*

The detailed *HPM* method is described below:

Step 1 Partition the relation R on group-by attribute to m clusters, denoted by r_i. Within each cluster, further partition the fragment r_i and the other relation S on join attribute to $n=N/m$ processors, denoted by r_{ij} and s_j. Therefore, the total data transmission time is given by

$$T_{comm} \times \left(\max\left(r_{ij} + s_j\right)\right)$$

where i is in the range of $[1,m]$ and j is in the range of $[1,n]$.

Step 2 Carry out join at each processor and the maximum processing time is expressed as

$$T_{join} \times \left(\max\left(r_{ij} \log s_j\right)\right)$$

Step 3 Perform local aggregation at each processor with the execution time

$$T_{agg} \times \left(\max\left(r_{ij} \times s_j \times Sel(j)\right)\right)$$

Step 4 Redistribute the local aggregation results to the processors within each cluster by partitioning the results on the group-by attribute. The transmission time is given as

$$T_{comm} \times \left(\max\left(r_{ij} \times s_j \times Sel(j) \times Agg(j)\right)\right)$$

Step 5 Merge the local aggregation results within each cluster and this requires the time

$$T_{agg} \times \left(\max\left(r_{ij} \times s_j \times Sel(j) \times Agg(j)\right)\right)$$

Step 6 Perform the *Having* predicate in each cluster with the processing time

$$T_{agg} \times \left(\max\left(r_{ij} \times s_j \times Sel(j) \times Agg(j)\right)\right)$$

Step 7 Transfer the results from the clusters to the host. The time for data consolidation in the host is small and thus only data transmission cost is counted, i.e.

$$T_{comm} \times \left(\max\left(r_{ij} \times s_j \times Sel(j) \times Agg(j) \times \theta\right)\right)$$

The total execution time of the *HPM* is the sum of the time of the above steps and is

$$\begin{aligned} &HPM \\ &= T_{comm} \times \left(\max\left(r_{ij} + s_j\right)\right) + T_{join} \times \left(\max\left(r_{ij} \log s_j\right)\right) + T_{agg} \times \left(\max\left(r_{ij} \times s_j \times Sel(j)\right)\right) \\ &\quad + T_{comm} \times \left(\max\left(r_{ij} \times s_j \times Sel(j) \times Agg(j)\right)\right) \\ &\quad + T_{agg} \times \left(\max\left(r_{ij} \times s_j \times Sel(j) \times Agg(j)\right)\right) \times 2 \\ &\quad + T_{comm} \times \left(\max\left(r_{ij} \times s_j \times Sel(j) \times Agg(j) \times \theta\right)\right). \end{aligned} \tag{7-3}$$

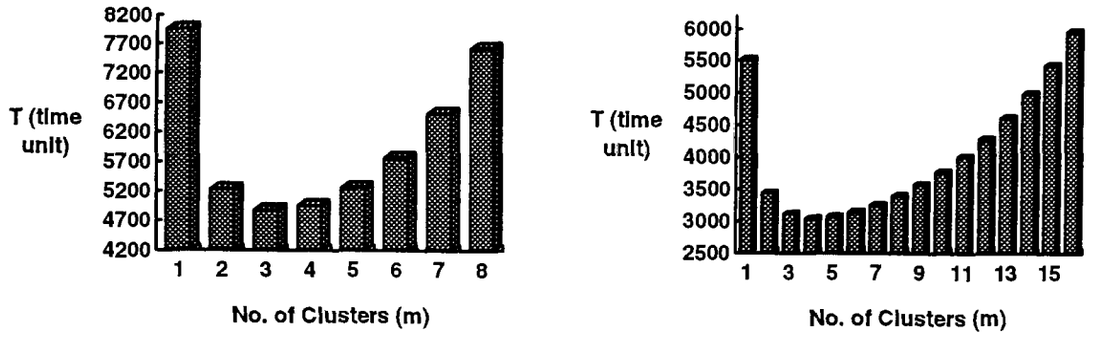
By applying the same simplification assumptions in the previous methods and $Agg(j) = \frac{1}{m} Agg$, equation (7-3) can be re-written as

$$HPM = T_{comm}\left[\alpha_n \times \alpha_m \times r + \alpha_n \times s + \frac{(1+\theta)}{\beta_n} \times J \times \frac{Agg}{m}\right] / z$$

$$+ T_{proc}\left[\alpha_n \times \alpha_m \times r \times \log(\alpha_n \times s) + \frac{J}{\beta_n} \times \left(1 + 2 \times \frac{Agg}{m}\right)\right],$$

where $J = r_{ij} \times s_j \times Sel(j)$, $\alpha_n = \frac{1}{\gamma + \ln n}$, $\alpha_m = \frac{1}{\gamma + \ln m}$,

and $\beta_n = \frac{1 + (\gamma + \ln n)^2}{2(\gamma + \ln n)^2}$.



(a) 8 processors (b) 16 processors

Figure 7.5: Cost vs No. of Clusters in *HPM*

It can be seen from the above execution time equation that the number of clusters m has strong influence on the performance of *HPM*. Figures 7.5(a) and 7.5(b) show the changes of query execution time when increasing the number of clusters in the *HPM*. It appears that a value of $m \approx \lceil \sqrt{N} \rceil$ approximately gives the optimal cost of the query[11] although the precise value of m may be worked out by finding the minimum value by differentiating equation (7-3).

7.4 Sensitivity Analysis

The performance of the three parallel processing methods presented may vary with a number of parameters listed in Table 7.1. In this section, we analysis among them the effect of the aggregation factor, the join selectivity factor, the degree of skewness (α and β), the relation cardinality, and the ratio of T_{comm} / T_{proc}. The default parameter values are given in Table 7.2.

7.4.1 Varying the Aggregation Factor

The aggregation factor β is defined by the ratio of result size after aggregation to the size of the base relation and its impact on three methods is shown in Figure 7.6. Not surprisingly, *APM* is insensitive to the aggregation factor. The reasons include little data to

[11] The approximation can also show the robustness of the adaptive method (*HPM*).

transmit after joining and processing the *Having* predicate comparing with data partitioning at beginning and broadcasting relation *S*, and little data to select with the group-by condition (*Having*) as the aggregation factor is reduced by a factor of the number of processors. Generally, the larger the aggregation factor, the more the running time is needed as shown in Figure 7.6(a) with 16 processors. Moreover, increasing the number of processors will reduce the running time despite data partitioning and processing skew, and the performance of *JPM* is better than that of *APM* except when the aggregation factor is very large as shown in Figure 7.6(b) with 32 processors. In both Figures 7.6(a) and 7.6(b), *HPM* offers the best performance.

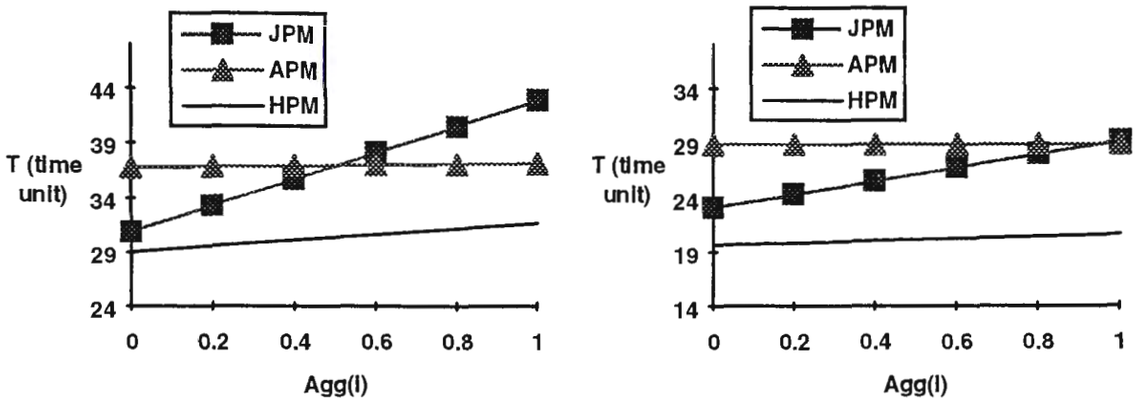
Parameters	Values
N	16
m	$\sqrt{16}=4$
r	1000 tuples
s	1000 tuples
$Sel(i)$	$5(N/r) = 0.08$
$Agg(i)$	0.5
θ	0.5
α	0.2985
β	0.5446
α_n	0.5093
α_m	0.5093
β_n	0.6297
T_{comm}	0.1 standard time unit per message
T_{proc}	0.01 standard time unit per tuple
z	100 tuples per message

Table 7.2: Default Values Listing of Parallel Processing of Aggregate Functions

7.4.2 Varying the Relation Cardinality

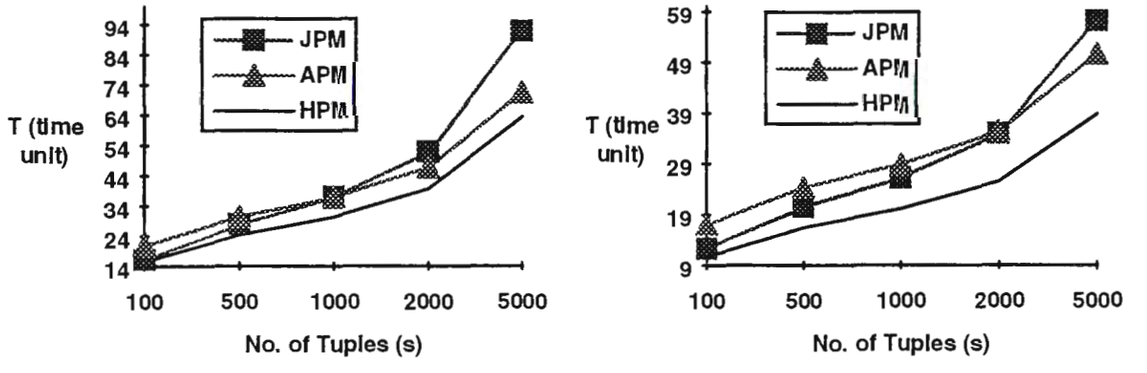
The cardinality of the operand relations are assumed to be the same elsewhere in the sensitivity analysis and their influences on performance are investigated in this subsection. Figure 7.7 shows the query execution time when we fix the cardinality of one relation and increase the cardinality of another relation. *JPM* appears to be better than *APM* only when

the varied relation size is small while the *HPM* again outperforms *APM* and *JPM* in all situations. Comparing Figure 7.7(a) to Figure 7.7(b), increasing processors will raise the cross-over point of *JPM* and *APM*.



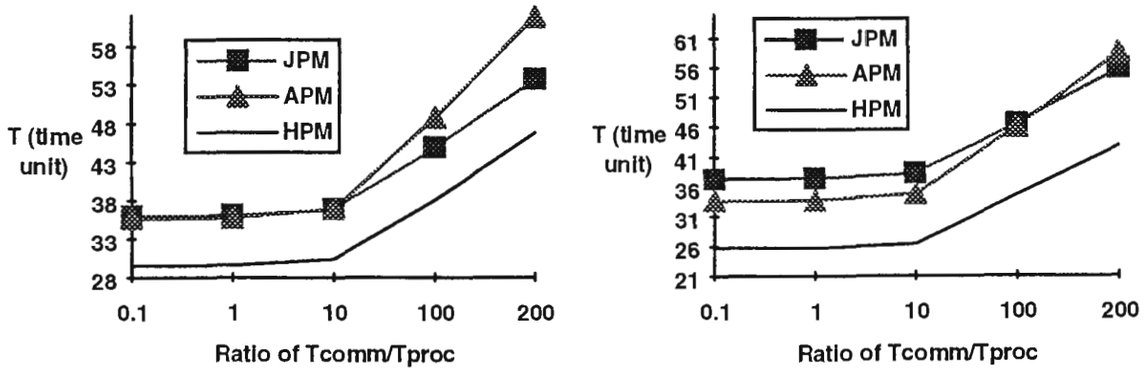
(a) 16 processors (b) 32 processors

Figure 7.6: Varying Aggregation Factor



(a) 16 processors (b) 32 processors

Figure 7.7: Varying the Cardinality of Relation *S*



(a) 16 processors (b) 32 processors

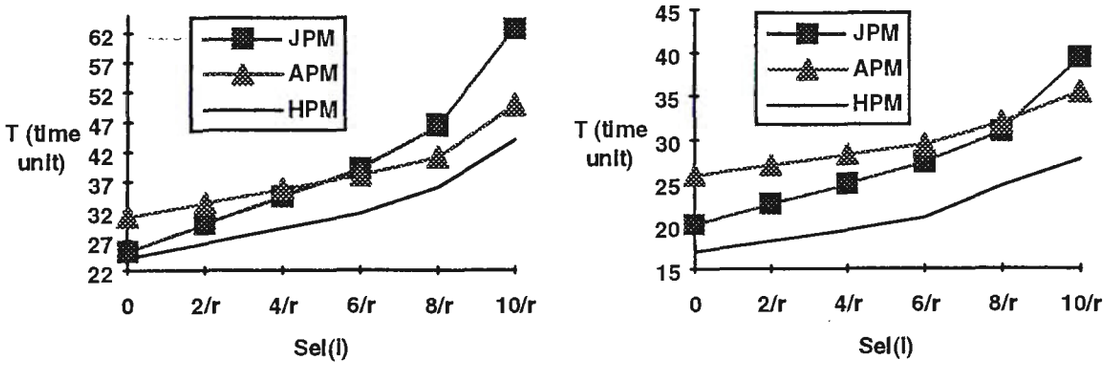
Figure 7.8: Varying the Ratio of T_{comm} / T_{proc}

7.4.3 Varying the Ratio of T_{comm} / T_{proc}

The ratio of T_{comm} / T_{proc} reflects the characteristic of the network used in the parallel architecture. Primarily, data communication is not a critical issue any more in parallel database systems comparing with distributed database systems [Alma94]. As we increase the ratio shown in Figure 7.8, system performance decreases since we treat T_{proc} as a standard time unit and magnify the communication cost, i.e. higher ratio means more expensive communication. Being a parallel database system, the ratio tends to stay small and *APM* is the most sensitive to the communication cost. Nevertheless, *HPM* will always perform better than either *JPM* or *APM*.

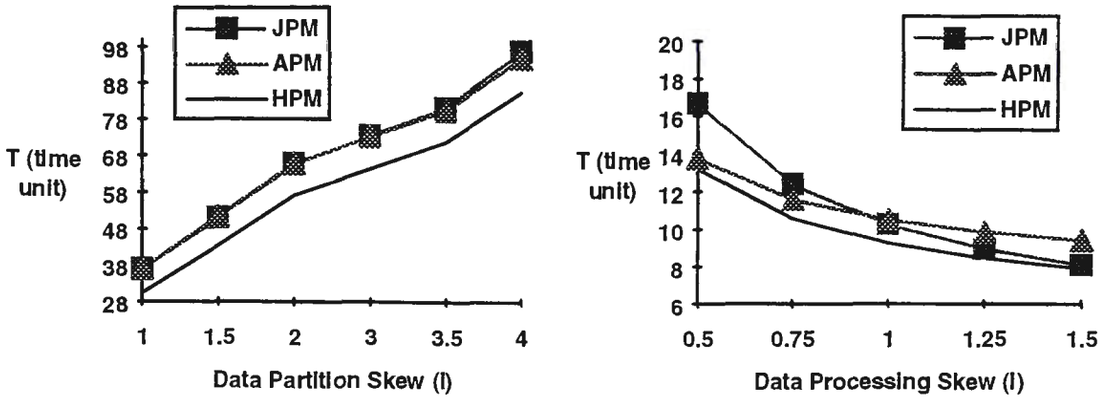
7.4.4 Varying the Join Selectivity Factor

The join selectivity factor has significant influence on parallel aggregation processing as it determines the number of intermediate tuples resulting from join intermediately. After that, those tuples are processed for aggregation and evaluated with the predicates. Eventually, the qualified tuples are unioned to form the query result. Lower selectivity factor involves less aggregation processing time and transferring time, and thus favours *JPM* as displayed in Figure 7.9. Less processors will reduce the impact of the entire second relation (both communication and processing) on running time so it favours *APM*.



(a) 16 processors (b) 32 processors

Figure 7.9: Varying the Selectivity Factor

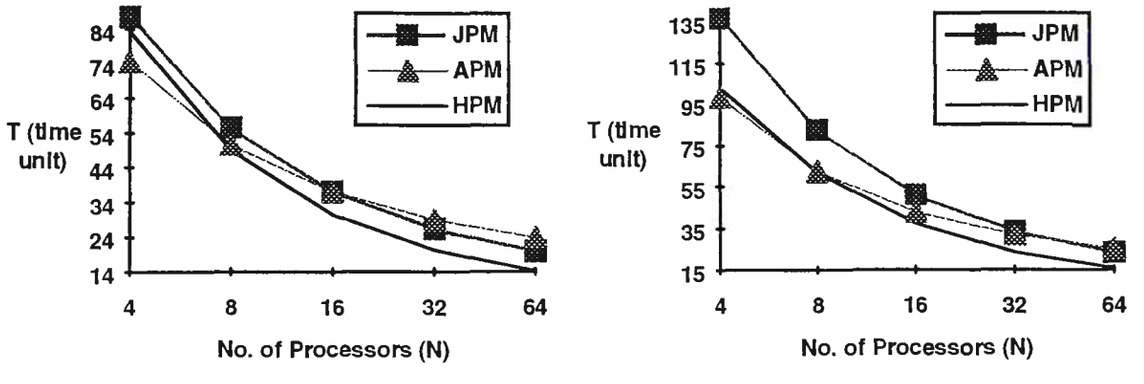


(a) Data partition skew (b) Data processing skew

Figure 7.10: Varying the Skewness with 16 Processors

7.4.5 Varying the Degree of Skewness

Figure 7.10(a) indicates the tendency of the performance when the data processing skew changes accordingly with the data partitioning skew whereas Figure 7.10(b) provides the comparison when we ignore the data partitioning skew, i.e. $\alpha = 1/N$ and alter the data processing skew. The values on the horizontal axis of both figures represent the expanding skewness factor which then is multiplied by the basic unit given by the Zipf distribution. Unlike the α, the β is inversely proportional to data processing skew and the larger the factor β, the less the data processing skew is. We conclude from Figure 7.10 that either of partitioning skew or processing skew degrades the performance of parallel processing, *HPM* outperforms *APM* and *JPM* even in the presence of skewness, and *APM* is less affected by the skewness compared with *JPM*.



(a) *Agg(i)*=0.5, *Sel(i)*=5(*N*/*r*) (b) *Agg(i)*=0.6, *Sel(i)*=10(*N*/*r*)

Figure 7.11: Varying the Number of Processors

7.4.6 Varying the Number of Processors

One of the desired goals of parallel processing is to have a linear scale up which can be achieved when twice as much processors perform twice as large a task in the same time. When the number of processors is increased, as we expected, performance is improved in spite of the skewness shown in Figure 7.11. *APM* performs extremely well when the number of processors is small, and it is even better than *HPM* because the number of clusters in *HPM* may not be optimised and less processors make the communication cost insignificant which favours *APM*. However, when the database system scales up, both *HPM* and *JPM* perform better than *APM*.

7.5 Summary

Traditionally, join operation is processed before aggregation operation and relations are partitioned on join attribute. In this chapter, we demonstrate that group-by attribute may also be chosen as the partition attribute and present three parallel methods for aggregation queries, *JPM*, *APM*, and *HPM*. These methods differ in their way of distributing query relations, i.e. partitioning on the join attribute, on the group-by attribute, or on a combination of both; consequently, they give rise to different query execution costs. In addition, the problem of data skew has been taken into account in the proposed methods as it may adversely affect the performance advantage of parallel processing. A performance comparison of these methods has been provided under various circumstances of queries and processors. The results show that when the join selectivity factor is small and the degree of skewness is low, *JPM* leads to less cost; otherwise *APM* is desirable. Nevertheless, the hybrid method (*HPM*) is always superior to the other two methods since the data partitioning is adaptively done on both join attribute and group-by attribute. In addition, it is found that the partitioning on group-by attribute method is insensitive to the aggregation factor and thus the method will simplify algorithm design and implementation.

CHAPTER 8

OPTIMAL PROCESSOR ALLOCATION FOR PARALLEL QUERY EXECUTION

8.1 Introduction

The objective of the parallel query processing is to translate a high-level query into an efficient low-level execution plan and allocate processors to each operation in such a way that the overall execution time is minimised. Following [Hong92, Gang92, Shek93, Hasa94, Chek95], we divide the processing into two phases, query decomposition and parallel plan formulation. The former includes query normalisation, query analysis, elimination of redundancy, and rewriting while the latter is looking for parallel optimal plans to execute the query; this chapter is devoted to the latter and considers a single query from Section 8.3 to Section 8.7, and multiple queries in Section 8.8 and Section 8.9.

In Chapter 6, intra-query parallelism is studied and the emphasis is on skew effects and skew principle by which allocating a large number of processors to a single operation may not necessarily be beneficial. In this chapter, we study the processor allocation strategies for parallel relational databases and the focus is on both intra-query parallelism possibly with massive parallelism and multiple dependent queries of inter-query parallelism. Global optimisation issues on processor allocation are investigated and the degree of parallelism may be large. A complete query cost model is developed taking into account of communication overheads in multicasting and the load skew of data partitioning. The concepts of *optimal degree of parallelism* for each operation and *time equalisation* are introduced. Two new processor allocation algorithms are presented. *DPAA* is a dynamic method based on non phase-based approach, and *MPPPA* tries to group operations evenly into execution phases based on the heuristic of the merge-point evaluation. Optimising processor allocation issues are discussed and processor bounds achieving optimal time are given. The optimal time of a single query can be easily derived with the proposed clustering method, *subtree-grouping method* (*SGM*).

Processor allocation in parallel system is an NP complete problem and many heuristic methods have been proposed [Kris86]. However, the heuristics though invaluable in certain circumstances, hardly ever take into account the global picture. An optimised processor allocation strategy, *OPA*, is presented by making use of the processor bounds for optimal time. *OPA* guarantees to provide a global optimal solution when the number of processors is sufficient and a local phase optimal solution when the number of processors is insufficient. Query examples are given to illustrate how the new algorithms perform.

How to schedule multiple queries in processor allocation with each query consisting of multiple relational operations is also an important issue. The result of exploiting parallelism at this upper level is *inter-query parallelism*. In contrast to intra-query parallelism, inter-query parallelism has received little attention particularly for multiple dependent queries. When there is query dependency, the complexity of the problem grows dramatically since the queries can only be represented using a directed graph which is no longer a tree structure. We extend *SGM* to deal with multiple independent queries represented by a logical hyper-tree structure. Finally, we focus on multiple dependent queries. Their applications are discussed, and query processing is represented using the *critical path scheduling* (*CPS*). A decompression algorithm is developed to optimise the execution time by making use of the activity analysis of critical path analysis, and resource scheduling and leveling in project management.

The remainder of the chapter is organised as follows. Section 8.2 describes the data communication model, the skewness in computation and the query cost model. The processor allocation policy classification is discussed in Section 8.3. The new processor allocation policies are presented in Section 8.4 and time equalisation technique is proposed in Section 8.5. Optimisation issues on processor allocation such as optimal query time and processors bounds are introduced, and an adaptive processor allocation algorithm is developed in Section 8.6. Intra-query examples are given in Section 8.7, and the issues of multiple queries and their applications are described in Section 8.8. Multiple query execution with query dependency is introduced in Section 8.9. Finally, the chapter is concluded in Section 8.10.

8.2 Optimal Degree of Parallelism with Overheads

In Chapters 6 and 7, we assume that the number of processors is small so that the communication mainly deals with data transmission, and communication overheads are ignored. However, in this chapter, we shall introduce the concepts of optimal degree of parallelism and sufficient number of processors. Depending on the individual operation cost model, the optimal degree of parallelism may require a large number of processors, i.e. massive parallelism. In such a situation, communication overheads will play a significant role in the overall query cost model.

With the hardware architecture model shown in Figure 6.1, the software model consists of one process serving as the coordinator and other processes running on worker processors. Due to database fragmentation, processors may deal with their partitions simultaneously.

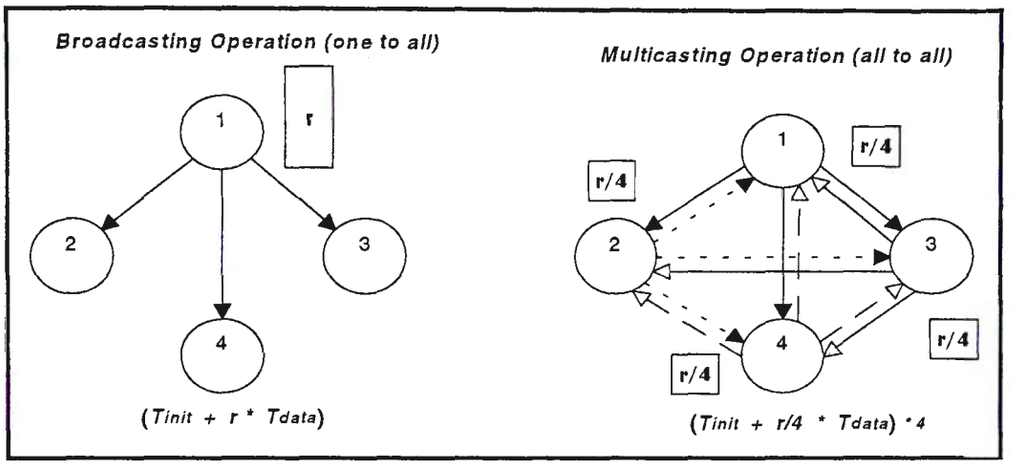


Figure 8.1: Data Communication

8.2.1 Data Communication Overheads

The broadcast one-to-all operation can be carried out in one single step without hopping through any intermediate processors but the multicasting all-to-all operation is expected to involve communication delay. Figure 8.1 shows the data communication operation for broadcasting and multicasting. There is little problem with broadcasting one-to-all operation, but the multicasting all-to-all operation does involve some degree of optimisation. If communication includes an identifiable preparation phase such as packaging and transmission, multiple sites may take advantage of the preparation time so that channels are better utilised used. However, exactly how to model the preparation and transmission time is beyond the scope of this thesis, and the communication modelling issues can be found in [Saad89, Bhat93, Alma94]. In the thesis, we make use of the worst case scenario for multicasting operation where each site carries out the broadcast operation in sequence. In Figure 8.1, there are four sites and one site has a relation of size r for the broadcasting operation; for multicasting there are 4 sites and each site has a relation of size $r/4$ due to replication and fragmentation. Hence from the figure, the data transmission time for multicasting operation is the same as that of broadcasting operation but the multicasting operation does introduce more overheads, e.g. synchronisation and hand-shaking, are required before every individual transmission.

Parameters	Meaning
r, s	the cardinality of relation R and S
n	the number of processors
r_i	the number of tuples in the ith processor after the partitioning of relation R
σ_i	selectivity factor of the ith fragment of the relation R after partitioning
W_1	loading time for each tuple (including disk access time and transfer time)
W_2	processing time for each tuple (mainly comparison and computation time)
W_3	writing time for each tuple
θ	the data skew factor $(\theta \geq 0)$
L	the local operation processing time
T_{init}	time cost for initialising the operation on multiple processors (fixed cost)
T_{hash}	time cost for hashing one tuple
σ_{Join_Sel}	the join selectivity factor
T_{con}	the communication time for data gathering at the end of the processing
T_{data_par}	the communication time for data partitioning at the beginning of the processing
T_{data}	time cost for transferring one tuple

Table 8.1: Parameters Listing of Optimal Processor Allocation

8.2.2 Load Imbalance Description

Due to the non-uniform occurrences of relation domain values (the data skew), the data fragments over processors after unbiased partitioning may result in uneven sizes. In this chapter, the load skew is measured in terms of the different sizes of fragments that are allocated to the processors. Given the notations in Table 8.1, the size of the ith fragment[12] is described by a Zipf Distribution as shown in equation (6-1). In the case of $\theta > 0$, it is observed that the first fragment r_1 is always the largest in size whereas the last one r_n is the smallest. Therefore,

$$\max(r_i) = r_1 = \frac{r}{\sum_{j=1}^{n} \frac{1}{j^{\theta}}}. \qquad (8\text{-}1)$$

[12] The fragment i is not necessarily allocated to the processor i.

8.2.3 Query Cost Model with Overheads

a. Selection / Projection

The unary operations are processed locally without data partitioning because of their simplicity. As such, each processor will load the data fragment at that site into memory, process it, and output the result to the buffer. The time for selection or projection operation is given by

$$\begin{aligned} T_{unary} &= T_{init} + \max(L_{R(i)}) \\ &= T_{init} + \max\left(r_i \times (W_1 + W_2 + W_3\sigma_i)\right). \end{aligned} \tag{8-2}$$

b. Binary Join (Hash Partitioning Strategy and Local Hash Join)

When the tuples of operand relations are read in at each processor, a hash function is employed to determine these tuples destinations. This works as a multicasting phase in which each processor sends a data stream to every other processors, and if there is a perfect hash function and no data skew in data partitioning, the sizes of the transmitted data stream will be all the same. Local join processing is then carried out in individual processor. The costs of join methods vary with the index available on the join attributes and the distribution of cardinality of join relations, and thus none of the join methods always outperforms others. Nevertheless, the hash join method is often used in parallel processing since it usually performs best when no index but only the join relations are transferred across the processors. Assuming that the hash join is used, the local processing time involves the time for building a hash table for one relation, probing the table by the other relation and writing the joined tuples into the buffer. Therefore, the time for binary join is given by

$$T_{hash\_join} = T_{init} + \beta \max\left(T^i_{hash}\right) + \gamma T_{data\_par} + L_{hash\_join}, \tag{8-3}$$

where the parameters β and γ are determined by the overlapped execution time among hashing, data transmission and local join processing.

Assuming that binary join requires two base relations R and S, we obtain

$$\max(T_{hash}^{i}) = \max(r_i + s_i) \times T_{hash}$$

and

$$T_{data\_par} = \left(\frac{n-1}{n} \max(r_i + s_i) \times T_{data} + T_{init} \right) \times n .$$

Moreover, the local hash join has three components: loading into memory, processing, and writing; thus its time can be described by

$$L_{hash\_join} = \max(r_i + s_i) \times (W_1 + W_2) + \max(r_i \times s_i \times \sigma_{Join\_Sel}) \times W_3 .$$

Therefore, the total execution time of hash partitioned join is given by

$$\begin{aligned} T_{hash\_join} &= T_{init} + \beta \max(r_i + s_i) \times T_{hash} \\ &\quad + \gamma \times \left(\frac{n-1}{n} \max(r_i + s_i) \times T_{data} + T_{init} \right) \times n \\ &\quad + \max(r_i + s_i) \times (W_1 + W_2) + \max(r_i \times s_i \times \sigma_{Join\_Sel}) \times W_3 \end{aligned} \quad . \quad (8\text{-}4)$$

To simplify the above equation, when $\theta = 0$ and $\beta = \gamma = 1$, the time for binary join in the absence of skew can be rewritten as

$$\begin{aligned} T_{hash\_join} &= T_{init}(n+1) + \frac{r+s}{n} T_{hash} + (n-1) \frac{r+s}{n} T_{data} \\ &\quad + \frac{r+s}{n} (W_1 + W_2) + \frac{r \times s}{n} \sigma_{Join\_Sel} \times W_3 \end{aligned} \quad ; \quad (8\text{-}5)$$

when $0 \le \theta \le 1$ and $\beta = \gamma = 1$, the time for binary join in the presence of skew is

$$\begin{aligned} T_{hash\_join} &= T_{init}(n+1) + \frac{r+s}{\sum_{j=1}^{n} \frac{1}{j^{\theta}}} T_{hash} + (n-1) \frac{r+s}{\sum_{j=1}^{n} \frac{1}{j^{\theta}}} T_{data} \\ &\quad + \frac{r+s}{\sum_{j=1}^{n} \frac{1}{j^{\theta}}} (W_1 + W_2) + \frac{r \times s}{\sum_{j=1}^{n} \frac{1}{j^{\theta}}} \sigma_{Join\_Sel} \times W_3 \end{aligned} \quad . \quad (8\text{-}6)$$

8.2.4 Optimal Degree of Parallelism for One Operation

The degree of parallelism is the number of processors used to execute the operation, and in the cost model equation (8-4) increasing the number of processors raises the parallel processing overheads and decreases the data processing time without skewness. Therefore, for every operation in the query, there is an optimum number of processors that minimises the overall execution time. Beyond the turning point, adding extra processors will degrade the performance. The issues on optimal degree of parallelism for parallelisation are also discussed in [Wils92, Alma94].

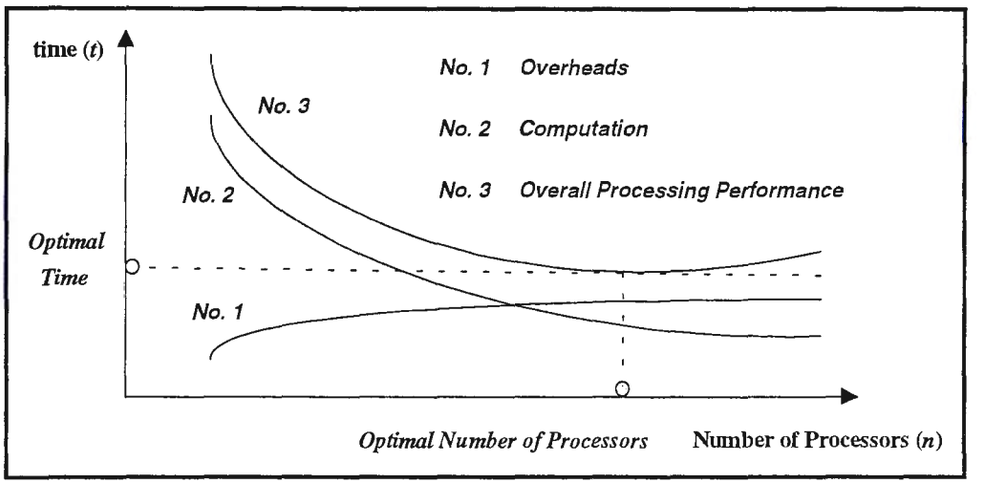


Figure 8.2: Optimal Number of Processors and Optimal Time

From the practical points of view, the degree of parallelism is affected by the communication overhead that includes software overhead, hardware latency, and message delays caused by network and memory contention. Generally, this overhead depends on such factors as the message length, number of nodes involved, traffic conditions, network bandwidth, and messaging algorithms used. Letting t_i be the time of processing the ith operation and n_i be the number of processors required, Figure 8.2 shows that the execution time for each operation is given by $t_i = T_{overheads} + T_{computation} = f(n_i)$. Consequently, the optimal degree of parallelism $n_{op\text{-}i}$ can be obtained from $dt_i/dn_i(n_{op\text{-}i}) = 0$, and the optimal time $t_{op\text{-}i}$ is equal to $f(n_{op\text{-}i})$.

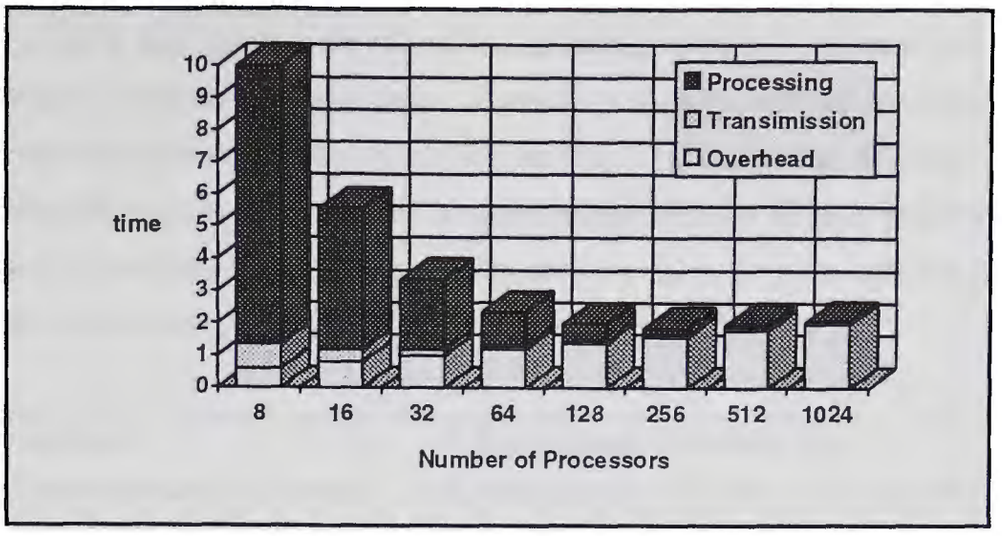
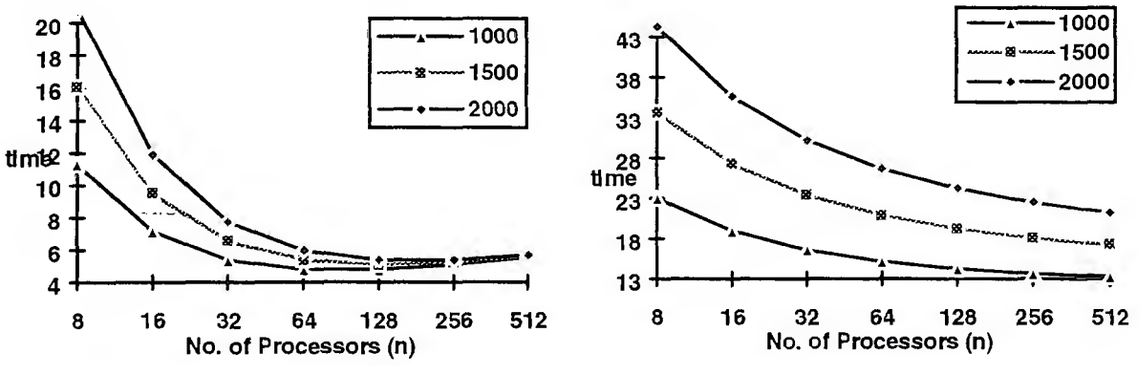


Figure 8.3: Optimal Degree of Parallelism r=s=1000 tuples, T_{data}=0.003 time unit, T_{hash}=W_1=W_2=W_3=0.01 time unit and T_{init}=0.2 time unit

Figure 8.3 shows the optimal degree of parallelism of a binary join operation without skewness where we have assumed r=s and σ_{Join_Sel}=1/r. In the figure, the time consists of the processing time, data transmission time and parallel processing overheads. With a small number of processors, the CPU processing time dominates the join time function and the overhead is a small component of join time. However, increasing the number of processors gives rise to large overhead, e.g. in Figure 8.3, when the number of processors reaches 1024, the join time can even be approximated by the overheads only.



(a) without data skew $\theta = 0$ (b) with data skew $\theta = 1$

Figure 8.4: Optimal Time for Each Operation

The skewness degrades the system performance but it delays the turning point as shown in Figure 8.4. With the same workload of relation cardinality 1000, 1500, 2000 tuples, Figure 8.4(a) shows the turning points are 64, 128, and 256 processors in the absence of data skew whereas Figure 8.4(b) does not show any turning points in the presence of data skew. The

main reason is that although the skewness has influence both on communication and computation, it does not have any impact on parallel processing overheads associated with the number of processors. To find the optimal number of processors that minimises t_i, we set the derivative $dt_i/dn_i = 0$. However, considering that it may be difficult to differentiate the cost model and rounding a non-integer value may lose the truly optimal solution, a short algorithm is provided in Figure 8.5 to look for the turning point[13].

```
int flag=1;                          /* flag to indicate the turning */
float miny=cost(1, r, s, skew);      /* variable to store the minimal execution time */
float y;
int n=2;
DO {    y=cost(n, r, s, skew);       /* look for the turning point */
        IF (y<miny) {miny=y; n++;}
        ELSE flag=0; }
WHILE (flag && (n<num_site));
return (miny && n);                  /* return the turning point and the time */
```

Figure 8.5: Algorithm For Optimal Time

8.3 Processor Allocation Policy Classification

In intra-query parallelism, decisions have to be made on allocating the available processors among a number of competing database operations running in parallel and the overall objective is to minimise the query execution time. In other words, there are n processors, N operations, and operation sequences are listed in the query tree (an example can be found in [Liu96a]). We are looking for an optimal processor allocation rule represented by the numbers $\{n_1, n_2, \ldots, n_i, \ldots, n_N\}$ and the optimal clustered execution phases, time given by $\{p_1, p_2, \ldots, p_j, \ldots, p_m\}$, with the constraint $\sum_{\text{each phase}} n_i \le n$, (i.e. the total number of processors available at one time is limited), so that $\min\left(\sum_{j=1}^{m} p_j\right)$ is achieved.

Based on the searching scope, processor allocation can be divided into phase-based

[13] A numerical solution may be worked out using the *Mathematica*.

approach and non phase-based approach. The former groups the operations into several blocks based on the data flow and starts from the outside block with all ready operations. Blocks of operations are either directly saved or indirectly stored (modified first) into phases and the operations are executed phase by phase. The advantage of the approach is its simplicity but it may only achieve the local phase optimisation. The latter globally approaches the entire query tree considering all operations, and ideally conducts the operations in such a way that the query execution time is minimised under the processors constraint and data dependency. In both approaches, heuristic methods may be employed to avoid the exhaustive search for optimal solution.

8.4 Two Intra-query Processor Allocation Algorithms

Based on above discussion, a non phase-based approach Dynamic Processor Allocation Algorithm and a phase-based approach Merge-Point Phase Partitioning Algorithm using the heuristic of merge point evaluation, are presented below.

8.4.1 Dynamic Processor Allocation Algorithm (*DPAA*)

With *DPAA*, all ready operations in the query tree start executing simultaneously by allocating the optimal number of processors to each operation. Here, the optimal numbers of processors are allocated to one operation after another until no more operations or no more processors. Therefore, we may have surplus processors in free processor pool when there are a large number of processors in the system, or we may have operations waiting and an empty processor pool when the number of processors in the system is small. During execution if one operation finishes, its parent operation is checked and is labelled as ready-to-run if all its children finish their jobs. Otherwise, the parent operation has to wait for its children to complete. The algorithm is presented in Figure 8.6. The advantage of the algorithm is its simplicity and flexibility and the limitation of the algorithm is the number of processors must be sufficiently large to provide a satisfactory performance.

Notation	n	the total number of processors available
	n_i	the optimal number of processors for the ith operation
	t_i	the optimal execution time of the ith operation
	Q	the entire query tree or the query plan
	m	the total number of phases if phase-oriented approach
	k	the phase number in parallel plan of phase-oriented approach ($1 \leq k \leq m$)
	$a[k]$	the sub total number of processors in each phase
	sum	the total number of processors needed for the parallel execution plan

```
Dynamic Processor Allocation Algorithm (DPAA)
BEGIN
    traverse Q to find all ready operations;
    allocate the optimal number of processors to each operation
        until no more processors or no more operations;
    execute operations with processors allocated in parallel;
    WHILE there is unfinished operations DO        {
        IF one operation finishes THEN    {
                check all of its sibling;
                IF all of them are ready i.e. dependency allowance THEN
                        proceed to execute their parent;
                ELSE {
                        parent operation waits;
                        WHILE there are unprocessed children operations
                                allocate  MIN(optimal no. processors, available no. processors)
                                        to them[14];
                } /* end inner else */
        } /* end outer if */
    } /* end while */
END
```

Figure 8.6: Dynamic Processor Allocation Algorithm

[14] If there are multiple unprocessed children operations and the available number of processors can not satisfy the children optimal number of processors, children operations have to be executed one by one. This situation is really one of the worst case scenarios of the algorithm.

```
Merge-Point Phase Partitioning (MPPPA)
BEGIN
    traverse Q to find all ready operations and store in phase 1;
    k=1;                                        /* start from the first execution phase */
    a[k] = Σ (one phase) n_i ;
    REPEAT    {
        IF ( n >= a[k] ) THEN    {
            allocate the optimal # processors to each operation within the phase;
            parallel execution;                 /* local (phase) optimisation is achieved */
            phase execution time depends on the local heaviest operation; }
        ELSE    {
            FOR each operation in this phase
                IF not all its sibling are in this phase THEN
                        put this operation in the next execution phase;
                /* if this operation's parent's all children are in the current phase,
                 we call its parent a merging point; if its parent is not a merging
                point, we delete it from the current phase and re-schedule it to the
                next phase */
            apply TEM to all operations within the phase;
                /* time equalisation used in each phase */
        }       /* end else */
        k=k+1;                                  /* proceed to the next execution phase */
        find all updated ready operations in Q and store in phase k;
        a[k] = Σ (one phase) n_i ;
        }       /* end repeat loop */
    UNTIL reach the root;
END
/* Note: Time Equalisation Method (TEM) will be discussed in the following section */
```

Figure 8.7: Merge-Point Phase Partitioning Algorithm

8.4.2 Merge-Point Phase Partitioning Algorithm (MPPPA)

With *MPPPA*, all ready operations of the query tree are grouped into the first execution phase, and then an evaluation is conducted within the phase. If one operation's parent is a merge point, this operation can remain in current phase; otherwise this operation will be left

in the next execution phase. An operation is a merge point if and only if all its children are in previous execution phase. The heuristic employed here tries to distribute the number of operations in each execution phase evenly, and the *time equalisation* technique is also employed in each phase so that local phase optimisation is achieved. The algorithm is shown in Figure 8.7.

Notation t the number of operations in the phase

```
int flag=1;
IF (n >= t) THEN allocate each operation with one processor first;
ELSE { IF (n >= t/2) THEN
                        /* when # processors is small and # operations is relatively large */
            { flag=2;
              break the current execution phase into two phases with each phase
              having equal number of processors;}
       ELSE
            { flag=0;
              execute operations one by one with n processors; }
       } /* end outer else */
IF (flag!=0) THEN {
       WHILE there is available processors
            {compare the time among operations;
            record the longest execution time and the operation number;
            allocate one more processor to the operation with the longest time;
            decrement the available number of processors; }
       in parallel, execute the operations with allocated processors;
       flag--; } /* end if */
```

Figure 8.8: Time Equalisation Method

8.5 Time Equalisation Technique

It is not impossible to have several operations starting in one execution phase with an insufficient number of processors. The situation may be described as t operations of one phase and n processors available. We are looking for a set of $\{n_1, n_2, \ldots, n_t\}$ such that $\min(\max(f_i(n_i)))$. The individual functions for all operations are known

$f_1(n_1), f_2(n_2),\ldots, f_t(n_t)$ and the constraint is $n_1 + n_2 + \ldots + n_t \leq n$. The *minmax* objective gives rise to the following time equalisation criterion:

$$f_1(n_1) = f_2(n_2) = \ldots = f_t(n_t) = y.$$

The problem can be solved algebraically with the following

$$n = n_1 + n_2 + \ldots + n_t = n_1 + f_2^{-1}(y) + \ldots + f_t^{-1}(y).$$

However, sometimes, it may be difficult to work out the inverse function of $f_i(n_i)$. Therefore, we provide an algorithm listed in Figure 8.8 to complete the time equalisation.

8.6 Optimal Processor Allocation of A Single Query

Processor allocation is a classical resource scheduling problem and the resource constraint is the number of processors available. In this section, we shall illustrate that it has a global optimised solution when the number of processors is sufficient, and becomes an NP complete problem when the number of processors is insufficient. Hence, global optimal query execution time is discussed first followed by processor bounds derivation achieving global optimal time. The bounds also give the boundary between sufficient and insufficient number of processors. Finally, we propose an optimised processor allocation algorithm that is an adaptive method based on *MPPPA*, *DPAA*, and processors lower bound derivation. The algorithm presents a global optimal solution when the number of processors is sufficiently large and a locally optimal solution when the number of processors is insufficient.

8.6.1 Optimal Time

In Section 8.2.4, the optimal number of processors and optimal time for each operation are introduced and the task of this section is to provide the optimal time and number of processors bounds for the entire query. The query tree is decomposed into the simplest units with at most three operations in one single unit. A unit is regarded as an operation of

the superunit which in turn is an operation of the next higher level superunit. With a postorder tree traversal, a query tree may be divided into several units. We name this *subtree-grouping method (SGM)*.

In Figure 8.9, there are 4 units and they consist of P1,P2,P3 in unit one, P4,P5,P6 in unit two, P456,P7,P8 in unit three, P123, P45678, P9 in unit four. Associated with each operation are the optimal number of processors n_i for producing the optimal time t_i. The numbers angled brackets are the optimal number of processors and the optimal time for each unit.

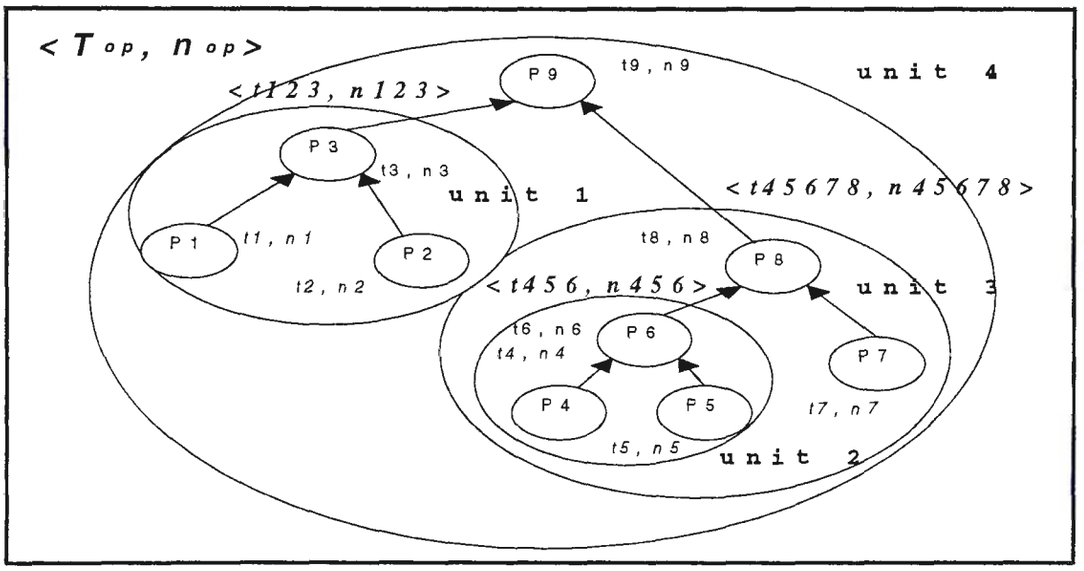


Figure 8.9: An Example of Optimal Time T_{op} and Number of Processors n_{op}

The general steps obtaining the query optimal time are as follows,

Step 1. Unit Division

Step 2. Postorder tree traversal, within each unit the unit optimal time is worked out by

$$T_{op} = T_{parent} + \max\left(T_{child-left},\ T_{child-right}\right).$$

In the example shown in Figure 8.9, there are 4 units and the query optimal time T_{op} can be worked out by starting with unit 1 and ending with the unit 4 including the root operation. Thus, we have

$$T_{op} = t_9 + \max\left(t_{123},\ t_{45678}\right)$$

$$= t_9 + \max\Big(\big(t_3 + \max(t_1, t_2)\big), \big(t_8 + \max(t_{456}, t_7)\big)\Big)$$

$$= t_9 + \max\Big(\big(t_3 + \max(t_1, t_2)\big), \big(t_8 + \max\big(\big(t_6 + \max(t_4, t_5)\big), t_7\big)\big)\Big).$$

8.6.2 Processor Bounds

Still based on the unit concept, the upper bound on number of processors for optimal time (*UBNP*) n_{op} is obtained by following steps,

Step 1. Unit Division

Step 2. Postorder tree traversal, within each unit the upper bound of the unit optimal number of processors is worked out by

$$n_{op} = \max\Big(n_{parent},\ \big(n_{child-left} + n_{child-right}\big)\Big).$$

Still based on the example shown in Figure 8.9, the upper bound of the optimal number of processors n_{op} can be worked out by starting with unit 1 and ending with the unit 4 with the root operation. Thus, we have

$$n_{op} = \max\big(n_9,\ (n_{123} + n_{45678})\big)$$

$$= \max\Big(n_9,\ \big(\max\big(n_3, (n_1 + n_2)\big) + \max\big(n_8, (n_{456} + n_7)\big)\big)\Big)$$

$$= \max\Big(n_9,\ \big(\max\big(n_3, (n_1 + n_2)\big) + \max\big(n_8, \big(\max\big(n_6, (n_4 + n_5)\big) + n_7\big)\big)\big)\Big).$$

The lower bound on number of processors for optimal time (*LBNP*) n'_{op} can be worked out with the following algorithm.

Step 1. Traverse the query tree, and work out the optimal number of processors and optimal time for each operation; group operations into units and calculate the unit optimal time.

Step 2. Starting from the highest level unit, i.e. unit with the root operation, reduce the slack time on the non-critical operation (or unit) to zero within the unit. If the non-critical operation is itself a unit, decision has to be made on decreasing the number of processors either from descendant operations or parent operations.

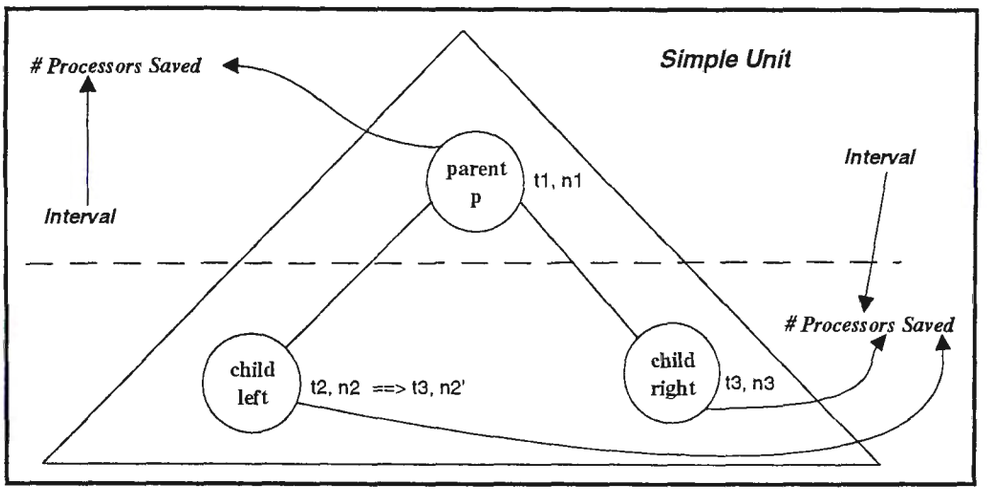


Figure 8.10: Searching the Number of Processors in One Simple Unit

```
if (t2 == t3);                          /* reduce the local slack time to zero */
else if (t2 > t3)   {
        t3 = t2; using the new t3 to work out n3'; }
else   { t2 = t3; using the new t2 to work out n2'; }
while (t > (t1+max(t2,t3))) {           /* compare # processors saved by allocating a */
        t1' = t1 + interval;            /* time interval to Parent and children       */
        using the t1' to work out n1';
        t3' = t2' = t2 + interval;
        using the t2' and t3' to work out n2' and n3';
        if ((n2'+n3') < n1')    {
                n2 = n2';   n3 = n3';   t2 = t3 = t2';  }
        else    { n1 = n1';   t1 = t1';   }
        t = t - interval;
        }
```

Figure 8.11: Algorithm on Searching the Number of Processors in One Simple Unit

The most fundamental issue of the algorithm is that given a time t, work out the minimal number of processors in one simple unit as shown in Figure 8.10. In the figure, the right child is the critical operation and thus we make $t2$ equal to $t3$ by reducing the number of processors from $n2$ to $n2'$. Next, based on the extra time unit *interval*, we compare the processors' reduction on parent and children. The time unit *interval* is allocated to the operation so that the number of processors required in parent and children operations are close. The algorithm is shown in Figure 8.11.

Optimised Processor Allocation Algorithm *(OPA)*

BEGIN

traverse Q to work out t_i and n_i for each operation;

in post tree order, divide Q into units;

within each unit, sum = $\max\left(n_{parent}, \left(n_{child-left} + n_{child-right}\right)\right)$ until the unit with root is reached;

/* *calculate total optimal number of processors for Q* */

IF (*sum* <= *n*) THEN *DPAA;* /* *global optimisation is achieved* */

ELSE *MPPPA*; /* *local optimisation is achieved* */

END

Figure 8.12: Optimised Processor Allocation Algorithm (*OPA*)

8.6.3 Optimised Processor Allocation Algorithm (*OPA*)

The remaining questions are how to achieve the global optimal query execution time with a sufficient number of processors and how to obtain the local phase optimal time with an insufficient number of processors in one algorithm. This gives rise to the proposed *OPA* that consists of two algorithms, *DPAA* and *MPPPA*, and their usage depends on the number of processors available. When the number of processors available is relatively large, i.e. sufficient, *DPAA* is used, and when the number of processors is small, i.e. insufficient, *MPPPA* is employed with the time equalisation within each phase. The detailed algorithm is shown in Figure 8.12 and the method is summarised in following four steps.

Step 1. Work out the optimal number of processors and the optimal execution time for each operation based on the cost model with travelling the whole query tree.

Step 2. Decompose the query tree into simple units, and calculate the optimal number of processors and optimal execution time for the unit; then, treat that entire unit as one operation and work out the optimal number of processors and optimal time for the super unit; in such a way, detect the query optimal execution time as well as the total number of processors required m.

Step 3. If m is not greater than n i.e. the number of processors is sufficient, *DPAA* is adopted. With *DPAA*, first, all *ready* operations[15] in the query tree are executed with the optimal number of processors. Then, whenever one operation finishes, a checking is involved. If all the operation's siblings are ready, proceed to process their parent operation, otherwise wait. This process continues till no more unprocessed operations are in the query tree and the program stops with a guaranteed global optimisation.

Step 4. If the condition in Step 3 cannot be satisfied, form the execution phases with *MPPPA* approach. With the *minmax* execution time objective, the local optimisation is obtained by applying time equalisation in each phase. Here the heuristic employed is merge-point phase partitioning, and an operation in the query tree is defined as a merge point if *all* its children are in the previous execution phase. The aim of the heuristic is distributing the number of operations evenly in each execution phase.

8.7 Examples of Intra-query Parallelisation

For comparison purpose, two other algorithms are also considered and implemented in the simulation. One algorithm, the intra-parallel processor allocation (*Intra*), executes operations one after another starting from the leaf nodes in the query tree. The operand relations of each operation are partitioned and distributed over the *optimal* number of processors or the available number of processors (the smaller of above two). Another algorithm, the phase-oriented processor allocation (*Phase*), clusters the query tree into several execution phases based on the data flow. The first phase involves the operations that require only base relations and thus are ready to process. The next phase may then contain the operations that become ready after completion of the first phase, and so on. Within each phase, the number of processors allocated to the operation is always less than the optimal number of processors despite the possibility of existing idle processors. The details of above two algorithms without introducing the degree of parallelism are discussed in [Leun93, Jian95].

[15] Ready operations are those whose operand relations are all available and thus the first group of ready operations only requires base relations.

Algorithm Name	Phase Number	Execution Time	Operation I.D.
Intra	1	2.8	P1
	2	3.5	P2
	3	3.9	P3
	4	3.4	P4
	5	4.2	P5
	6	3.4	P6
	7	3.8	P7
	8	3.5	P8
	9	3.7	P9
	Query execution time	32.2	
Phase	1	4.2	P1,P2,P4,P5,P7
	2	3.9	P3,P6
	3	3.5	P8
	4	3.7	P9
	Query execution time	15.3	
MPPPA	1	4.2	P1,P2,P4,P5,P7
	2	3.9	P3,P6
	3	3.5	P8
	4	3.7	P9
	Query execution time	15.3	
DPAA	non-phase approach		
	Query execution time	14.8	
OPA	non-phase approach		
	Query execution time	14.8	

Table 8.2: An Example of Query Execution Time with A Sufficient No. of Processors (256)

To illustrate how *OPA* works, we take an example query with 9 operations. First, we traverse the query tree and work out the optimal number of processors as well as the optimal execution time for every operation as listed in Figure 8.13. Then, we intentionally decompose them into the simplest unit with three operations. The unit of operations 1, 2, 3 requires 63 processors to have the minimal execution time -- 7.4 since the number of processors needed is the maximum of two phases, i.e. the maximum of phase 1 with operations 1 and 2 and phase 2 with operation 3, and the execution time is the summation of two phases, i.e. the totalling of phase 1 with operation 2 and phase 2 with operation 3. Consequently, the global optimal execution time 14.8 is obtained as well as the optimal number of processors required 241. Assuming a system with 256 processors, the query can

be processed with the globally optimised execution time of 14.8. According to *DPAA*, operations P1,P2,P4,P5,P7 are executed in parallel with 241 processors in use. When P4 is finished, one checking is conducted and it is found that P4 has to wait because P5 is still in progress; when P5 is completed, another checking is carried out and it is found that P6 can be processed. This dynamic processing continues until no more operations in the tree. The experimental results from simulation upon four algorithms are shown in Table 8.2 assuming a system with 256 processors.

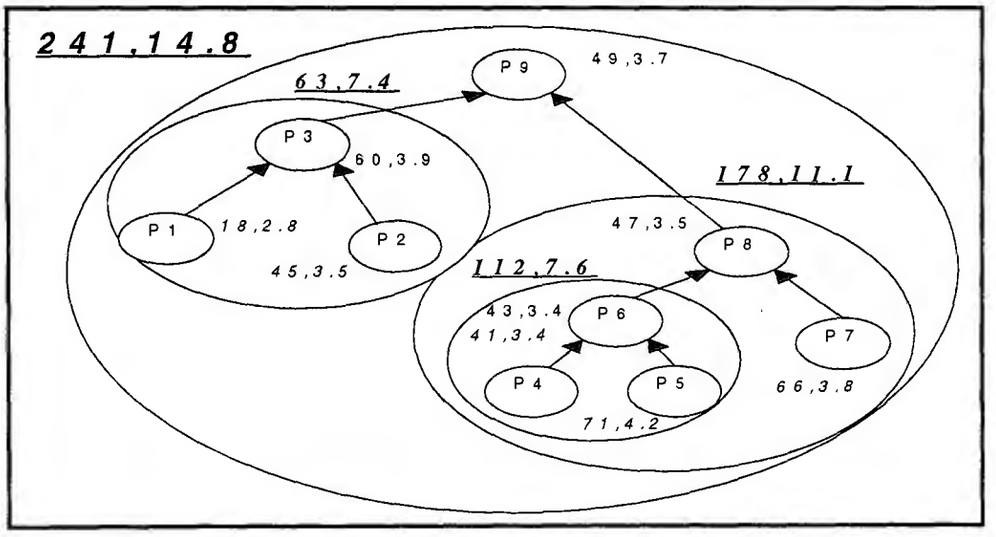


Figure 8.13: A Query Example with Sufficient Number of Processors

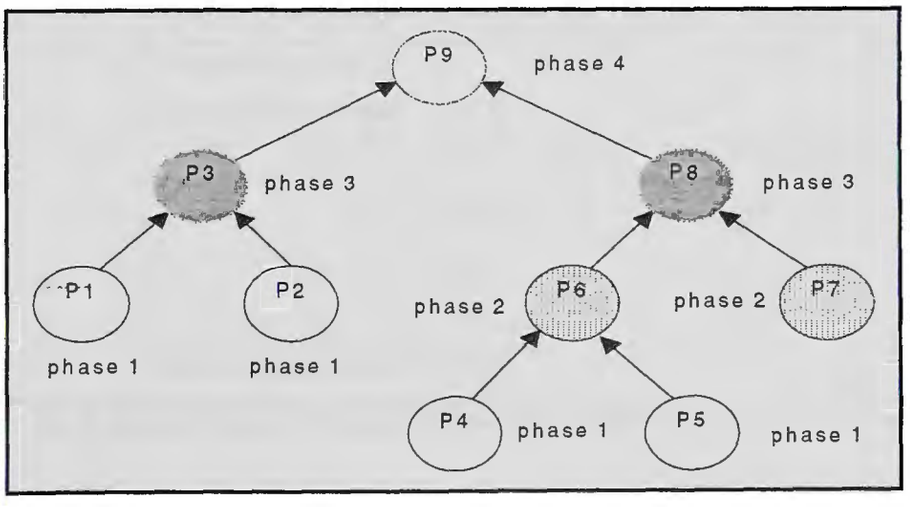


Figure 8.14: A Query Example with An Insufficient Number of Processors

Algorithm Name	Phase Number	Execution Time	Operation I.D.
Intra	1	3.1	P1
	2	5.5	P2
	3	6.1	P3
	4	5.1	P4
	5	6.4	P5
	6	5.3	P6
	7	7.3	P7
	8	5.7	P8
	9	6.0	P9
	Query execution time	50.5	
Phase	1	33.2	P1,P2,P4,P5,P7
	2	8.2	P3,P6
	3	5.7	P8
	4	6.0	P9
	Query execution time	53.1	
MPPPA	1	13.2	P1,P2,P4,P5
	2	10.4	P6,P7
	3	9.3	P3,P8
	4	6.0	P9
	Query execution time	38.9	
DPAA	non-phase approach		
	Query execution time	49.2	
OPA	1	13.2	P1,P2,P4,P5
	2	10.4	P6,P7
	3	9.3	P3,P8
	4	6.0	P9
	Query execution time	38.9	

Table 8.3: An Example of Query Execution Time with An Insufficient No. of Processors (8)

If the number of processors in the system is insufficient e.g. 8 processors, the operations are grouped into execution phases based on *MPPPA* as shown in Figure 8.14. First, all ready operations P1,P2,P4,P5,P7 in the query tree are collected into the first phase and a merge point evaluation is then accomplished to assure that all current phase operations' parent operations are in the next execution phase. As a result, P7 is left to the second phase with P6 and P3. Furthermore, P3 has to stay in phase 3 with P8. At last, P9 is in phase 4. The

operations in the tree are executed phase by phase and time equalisation technique is implemented within each phase. The experimental results of four algorithms from simulation are shown in Table 8.3 assuming a system with 8 processors.

8.8 Multiple Queries Execution

In a multi-user environment, it is common for a system receiving multiple queries at the same time. As a result, several queries are running on different processors in parallel. Multiple queries execution can be classified into two categories based on query dependency, multiple dependent and independent queries. Examples of these two categories of queries and their applications are given below. We extend the *SGM* method to cope with multiple independent queries. The decompression algorithm for tackling query dependency will be introduced in Section 8.9.

8.8.1 No Query Dependency

Figure 8.15 shows that there are m independent queries arriving at the system at the same time. To represent this with a hyper-tree structure, we make use of a logical root node, i.e. dummy node, and several logical arcs, i.e. pseudo-dependency. Associated with each query node, the numbers in angled brackets give the optimal time and optimal number of processors of the query respectively.

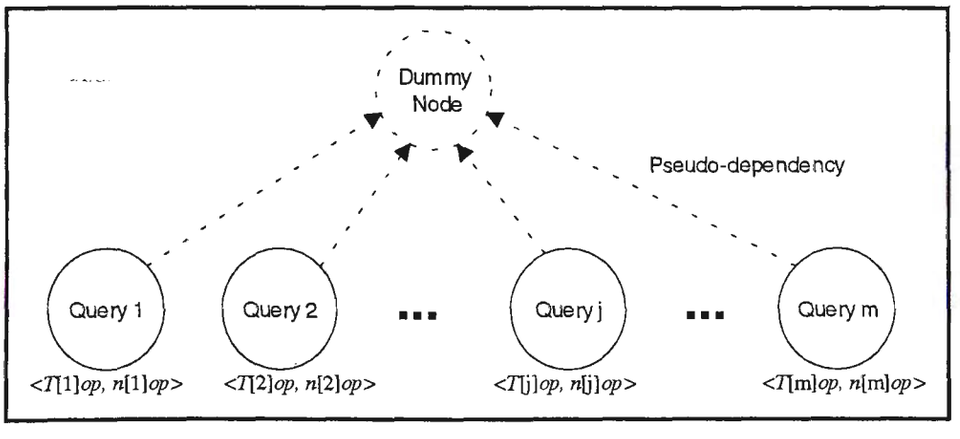


Figure 8.15: Multiple Queries without Query Dependency

The processors bounds and *SGM* provided in Section 8.6 clearly draw the line of demarcation where the global optimal solution of a single query is achievable. Here, we

extend *SGM* to cover multiple queries and present the steps of deriving the global optimal time and the optimal processors bounds of multiple queries as follows:

Step 1. Isolate each query and work out the individual query optimal time and number of processors, $T[i]_{op}$ and $n[i]_{op}$, based on the method provided in Section 8.6.

Step 2. Each query is treated as a composite unit; the optimal time and the optimal number of processors for multiple queries are given by

$$T_{op} = \max_i \left(T[i]_{op} \right)$$

and

$$n_{op} = \sum_i \left(n[i]_{op} \right).$$

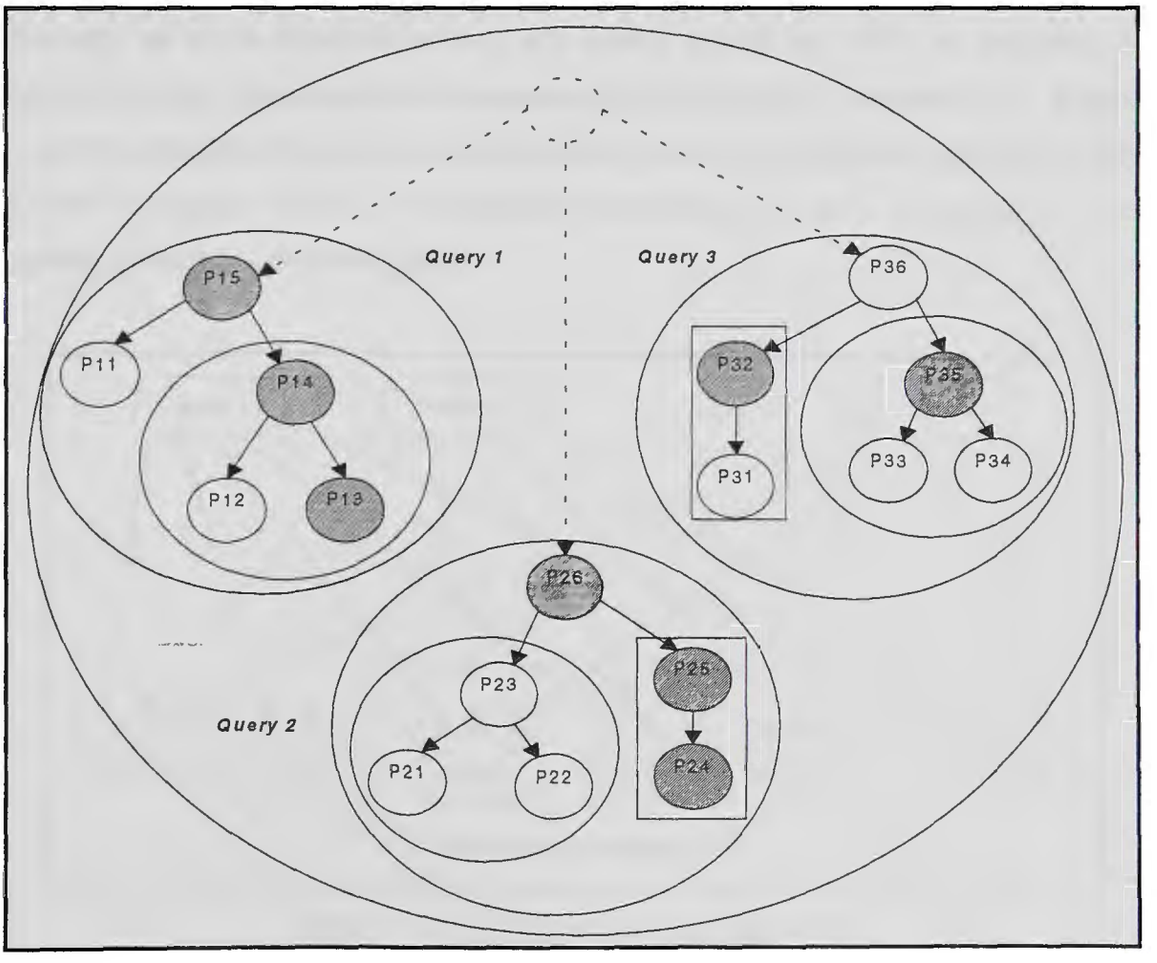


Figure 8.16: An Example of Three Queries without Data Dependency

Figure 8.16 shows an example of three queries without data dependency. The shaded circles indicate the skew operation, and the number of operations in queries are five, six, and six, respectively. A logical task with the three queries is constructed, and the logical root and its data flow are represented in dotted lines in the figure. As such, the optimal time and the optimal number of processors for the logical task can be easily derived.

When the number of processors is relatively small, the global optimal solution is unreachable, and the complexity in processor allocation and operation ordering is much higher than that of single query. The relevant issues on multiple independent queries optimisation can be found in [Wolf95, Ture94, Frie94, Du89].

8.8.2 Query Dependency

Recently, an active database research area is data mining, by which the extraction of information from large amounts of data accumulated and used for other purposes. A good example is the airline reservation system analysing the travellers pattern to keep planes fully booked. During the analysis, it is found that the result of one query is required by other queries, i.e. multiple dependent queries.

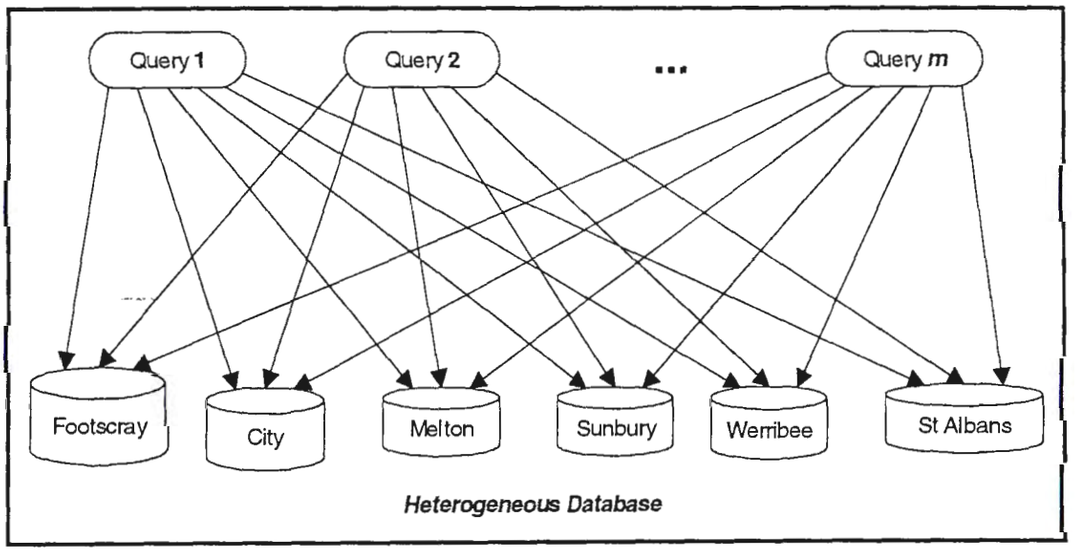


Figure 8.17: Traditional Database Approach

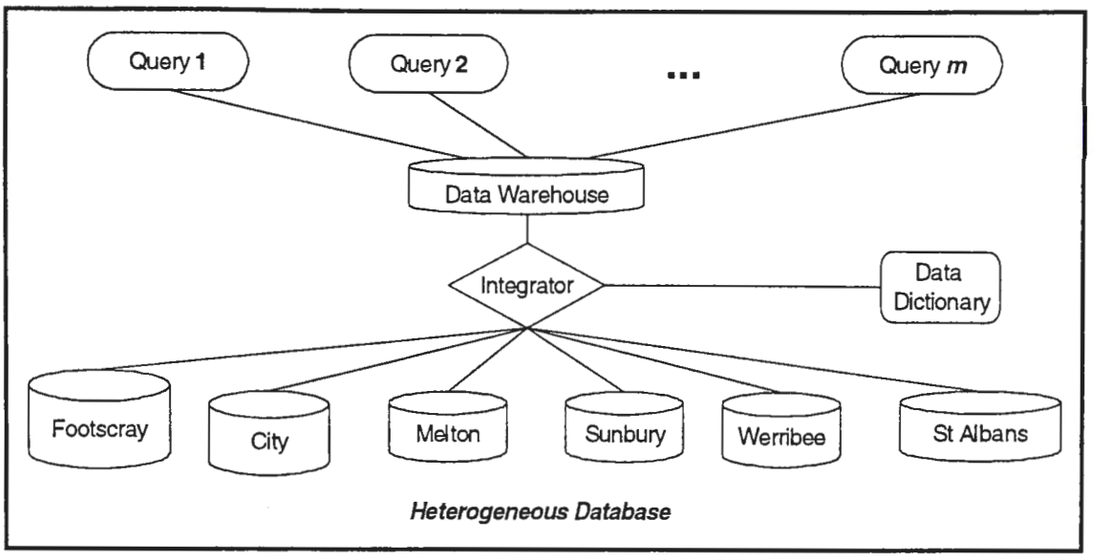
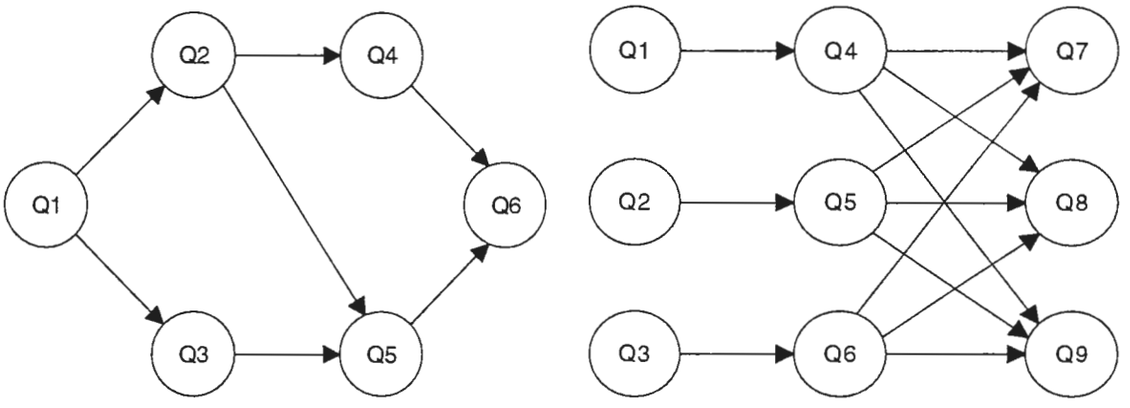


Figure 8.18: Data Warehouse Approach



(a) Example I with 6 Queries (b) Example II with 9 Queries

Figure 8.19: Examples of Multiple Dependent Queries

Another example of the existing query dependency is in heterogeneous databases where data are fragmented and possibly replicated at different sites with both hardware and software heterogeneity. Figure 8.17 shows an example of distributed databases located at six locations and there are *m* queries in the system. Assuming every query requires information from all sites and each site provides the same information to all queries, the most efficient way to process these queries is to identify the common components of different queries and rearrange multiple queries using query dependency. A more advanced approach is to integrate information in advance and store them in a data warehouse for direct querying and *OLAP* (see Figure 8.18). During integration, we can rearrange queries by isolating common query components based on the data dictionary so that the data transmission is minimised and the high reference of locality is obtained.

Figure 8.19 shows multiple queries with query dependency. A cyclic query dependency is shown in Figure 8.19(a) hereafter known as example I, and an acyclic query dependency is shown in Figure 8.19(b) hereafter known as example II.

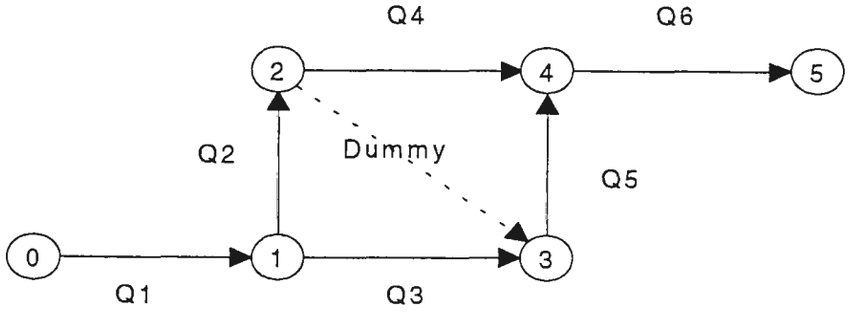


Figure 8.20(a): Activity-Oriented Representation of Example I

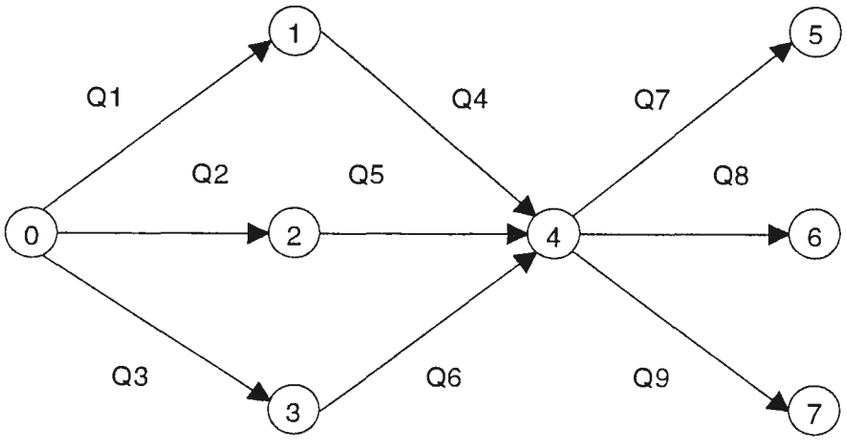


Figure 8.20(b): Activity-Oriented Representation of Example II

8.9 Multiple Queries Execution with Query Dependency

As shown in Figure 8.19, multiple queries with query dependency produce a much more complicated situation where queries are stored in a directed graph structure. A popular tool in Operations Research to manage and control project is the *critical path scheduling* (*CPS*) which is a representation of a project plan by a schematic diagram or network that depicts the sequence and interrelation of all the component parts of the project, and the logical analysis and manipulation of this network in determining the best overall program of operation [Batt64]. There is considerable similarity between project plan and control and query execution plan as we will discuss them later. Other reasons for using *CPS* are that it

has a very strong mathematical foundation, and based on the optimum time required for each activity. Using a logical mathematical model of the project, the most economical use of available resources (processors) can be obtained. In addition, we believe that *CPS* can be applied in the complete process of parallel query optimisation in intra-operation, inter-operation, and inter-query level. In other words, query execution without query dependency can be treated as a special case of that with query dependency. However, *SGM* of Section 8.6 is comparatively simple and thus it will be employed whenever it is applicable. *SGM* has its limitation since it is a tree structure optimisation. By contrast, multiple dependent queries are represented by a graph structure.

8.9.1 Problem Description with *CPS*

If multiple queries execution is regarded as a project, the project duration is the query execution time. Then the query execution plan may be formulated using *CPS*. As such, the resources and activities in the project are the processors and queries in execution. The objective function is the query cost model and the aim is to minimise the multiple queries execution time. The query representation in Figure 8.19 may be described as an event-driven network where the nodes are the jobs (i.e. queries) and the arrows indicate the dependency. To make use of *CPS*, we intentionally convert them to an activity-driven network. Figure 8.20 shows the translation results of Figure 8.19, and here the arrows are the queries, the nodes are events, and a dummy activity is introduced for synchronisation purpose.

8.9.2 Activity Analysis and Critical Path

With the examples of Figure 8.20, we conduct an activity analysis and identify the critical path based on t_{op} and n_{op} of each query. Using the notations listed in Table 8.4, the analysis results of Figure 8.20(a) with 6 queries are shown in Table 8.5. Likewise, the activity analyses of Figure 8.20(b) with 9 queries are shown in Table 8.6.

8.9.3 Resource Leveling and Resource Scheduling

If the number of processors available is infinite, we shall allocate processors according to the activity analysis table, i.e. n_{op} for each query and $n_{op\text{-}i}$ for the operation within the

Parameters	Meaning
EST	earliest start time
LST	latest start time
EFT	earliest finish time
LFT	latest finish time
TF	Total Float: the maximum additional time that can be made available to complete a particular activity and cannot be exceeded without delaying the project *TF=LFT-EFT*
FF	Free Float: the additional time available to complete an activity, assuming all other activities commence and finish as early as possible *FF*=the *EST* of its following activity -*EFT*
IF	Interfering Float: the difference between *TF* and *FF* for any activity *IF=TF-FF*

Table 8.4: Notations of Activity Analysis

Activity	Arrow	t_{op}	n_{op}	*EST*	*LST*	*EFT*	*LFT*	*TF*	*FF*	*IF*	Remarks
Q1	0-1	5.6	24	0	0	5.6	5.6	0	0	0	critical
Q2	1-2	4.8	19	5.6	5.6	10.4	10.4	0	0	0	critical
Q3	1-3	3.7	20	5.6	6.7	9.3	10.4	1.1	1.1	0	---
Dummy	1-4	0	0	10.4	10.4	10.4	10.4	0	0	0	critical
Q4	2-4	4.8	14	10.4	10.9	15.2	15.7	0.5	0.5	0	---
Q5	3-4	5.3	15	10.4	10.4	15.7	15.7	0	0	0	critical
Q6	4-5	4.5	28	15.7	15.7	20.2	20.2	0	0	0	critical

Table 8.5: Activity Analysis of Example I

Activity	Arrow	t_{op}	n_{op}	*EST*	*LST*	*EFT*	*LFT*	*TF*	*FF*	*IF*	Remarks
Q1	0-1	8.2	29	0	1.5	8.2	9.7	1.5	0	1.5	---
Q2	0-2	9.6	32	0	0	9.6	9.6	0	0	0	critical
Q3	0-3	6.8	22	0	0.7	6.8	7.5	0.7	0	0.7	---
Q4	1-4	6.3	20	8.2	9.7	14.5	16.0	1.5	1.5	0	---
Q5	2-4	6.4	18	9.6	9.6	16.0	16.0	0	0	0	critical
Q6	3-4	8.5	19	6.8	7.5	15.3	16.0	0.7	0.7	0	---
Q7	4-5	3.4	10	16.0	16.9	19.4	20.3	0.9	0.9	0	---
Q8	4-6	2.9	8	16.0	17.4	18.9	20.3	1.4	1.4	0	---
Q9	4-7	4.3	12	16.0	16.0	20.3	20.3	0	0	0	critical

Table 8.6: Activity Analysis of Example II

query. The multiple queries execution time can be derived using the critical path and a global optimal solution is achieved. However, it is often the case that the number of processors in the system is limited in the above *CPS* process. This is analogous to real world projects where we separate planning from scheduling to ensure an initial concentration on the development of the construction logic and the deferment of project resource considerations.

When there is not enough processors at a given stage for concurrent operations, we refer to this situation as resource constraint. Historically, there are two approaches, resource leveling and resource scheduling towards resource constraint. The former is to remove or ease the peak requirements of limited resource or minimise the number of certain specific limited resource types; the latter seeks the minimal time extension of the project duration based on the resource availability. Both approaches maintain the construction logic. In this thesis, we assume that the number of processors in the system is known and only consider resource scheduling.

8.9.4 Decompression Algorithm

When the resources are limited, the critical path of *CPS* may be changed and thus the query execution time may be lengthened. This process can be carried out by network decompression that is based on continuous optimal increase of the project duration from the shortest time to the longest time possible. We summarise the decompression algorithm as follows.

Step 1. Construct the *CPS* network and complete the activity analysis table.

Step 2. Phase classification based on the critical path activities and their interfering float (*IF*).

Step 3. While travelling along the critical path, check the available resource at each phase. If it is greater than or equal to the total number of processors of all current activities, execute queries in parallel and terminate the program. There is

a logical synchronisation point at the end of each phase so early completed queries have to wait due to query dependency.

Step 4. With processors constraint, decompression starts by eliminating as much total float (*TF*) as possible from non-critical activities one by one. When the *IF* of a non-critical activity is not zero, there is an option since the float time is shared by multiple activities. If the project cost-curve can be represented or can be approximated by a linear function, the cheapest way of increasing the non-critical path is determined by using the activity with the steepest cost slope. To fulfil this first level decompression, we provide a simple algorithm in Figure 8.21. This process carries out for all phases in the *CPS* and stops only when the resource constraint is released. As a result, all activities are critical at the end of this step. We will continue to Step 5 if the number of processors available is still insufficient. Up to this step, the project duration is unchanged.

Step 5. Further decomposition by reducing the number of processors for all concurrent activities. In fact, there is an option of serial execution of all paths which will also reduce the number of processors required. However, we assume activities are intermittent, so parallel execution of all paths gives better results since the number of concurrent activities is limited and the number of processors is reasonably large. As a result, parallel execution tends to distribute operations more evenly than that of serial execution in particular in the presence of load imbalance (see Chapter 6). Therefore, we are aiming at finishing all paths in the phase at the same time but the project duration is lengthened. Comparing the reduction on the number of processors in each phase (including all paths) by increasing the same amount of time, the maximum gain is selected from the alternatives. The reduction is continued until the resource constraint is relaxed. During the reduction within each phase, all paths are increased by the same time and the best activity for decompression is selected by the first level decompression algorithm shown in Figure 8.21.

However, in step 5, the selection of time unit is a proven difficult task since a small time unit may take long time to release the resource constraint and a larger time unit may miss the optimal point. Therefore, a good heuristic is reducing the number of processors and comparing the time shortened at each path.

8.9.5 Multiple Dependent Queries Examples

To show how the decompression algorithm works, we go through the two examples shown in Figures 8.19(a) and 8.19(b). The results of the first step are already presented in Tables 8.5 and 8.6. According to the phase classification in Step 2, we obtain several phases displayed in Table 8.7. When the number of processors available is infinite, we allocate processors based on Tables 8.5 and 8.6. Both examples are terminated with global minimal execution time, 20.2 and 20.3 time unit respectively.

```
FOR all non-critical paths DO
    WHILE (TF > 0) DO    {
        reduce a processor from the first query in the non-critical path;
        min_time_reduction=query new time - query old time;
        FOR all queries in this non-critical path DO    {
            reduce a processor from the ith query;
            temp_time_reduction=query new time - query old time;
            IF (min_time_reduction < temp_time_reduction)    {
                min_time_reduction = temp_time_reduction;
                allocate one less processor to this query;
                }
            }
        recalculate TF;
};
```

Figure 8.21: First Level Decompression Algorithm

If the number of processors available is small, we conduct first level decompression in Step 4. In example I, Q3 and Q4 are decompressed by increasing their completion time to 4.8 and 5.3 time unit, and decreasing the number of processors by 6 and 4. In example II, similar action can be taken in the second phase. However, there are options when we try to decompress the non-critical paths, Q1 and Q4, and Q3 and Q6. For the path with Q1 and Q4, the problem can be described as reducing a maximum number of processors by increasing the given time unit. As such, we can reduce processors from Q1, Q4, or Q1 and Q4. Based on the algorithm in Figure 8.21, a full comparison is conducted and the best plan is selected. We provide the same treatment on the path with Q3 and Q6.

If the number of processors available is still not enough after the decompression in Step 4, we continue Step 5. In example I, we increase the project duration by a certain time. Then, a decision is made on which phase, i.e. phases 1, 2, 3, 4, the time should be allocated. Within each phase, all queries finish at the same time, and again, a decision has to be made on which query the time should be allocated.

Phase No	Example I (Queries Contained)	Phase No	Example II (Queries Contained)
1	Q1	1	Q1, Q4
2	Q2, Q3, Dummy		Q2, Q5
3	Q4, Q5		Q3, Q6
4	Q6	2	Q7, Q8, Q9

Table 8.7: Phase Classification

This is really an extension to Step 4 with introducing multiples phases in one project. For example II, the same procedure is applied.

8.10 Summary

We have presented several intra-query processor allocation algorithms, namely, one phase-based, one non phase-based, and an adaptive algorithm in this chapter. The previous query cost model on a shared nothing parallel architecture is refined by incorporating the effects of data communication overheads. *MPPPA* makes use of the heuristic of merge-point evaluation so that the number of operations in each execution phase is distributed evenly. *DPAA* is a dynamic approach and its performance is sensitive to the number of processors available. The concept of optimal degree of parallelism for each operation is introduced, and achieving time equalisation is highlighted and implemented in *MPPPA*. The global optimal time and the optimal number of processors bounds are derived for each query, and they also provide a boundary where global optimal solution is achievable. Another usage of the bounds and the optimal time is providing a criterion by which other processor allocation algorithms can compare and their relative efficiency may be defined. *OPA* is an adaptive method based on *MPPPA* and *DPAA* by making use of the optimal processors bounds, and

the method always presents a global optimal solution with a sufficient number of processors and a local phase optimal solution in all other situations.

We have also presented a decompression algorithm in the chapter for multiple dependent queries and activity analysis based on *CPS* to deal with the optimisation issues in multiple query execution in parallel databases. The algorithm makes use of resource scheduling and resource leveling in project management, and the queries representation is converted to the activity-oriented network diagram. Examples are given to illustrate the operation of the algorithm. *SGM* has been extended to cope with multiple independent queries.

CHAPTER 9

PERFORMANCE STUDY OF PROCESSOR ALLOCATION ALGORITHMS

9.1 Introduction

In this chapter, we conduct a performance study on the intra-query and the multiple dependent queries processor allocation algorithms presented in Chapter 8. Five intra-query processor allocation algorithms, namely, the intra-parallel processor allocation (*Intra*), the phase-oriented processor allocation (*Phase*), the dynamic processor allocation (*DPAA*), the merge-point phase partitioning (*MPPPA*), and the optimised processor

allocation (*OPA*), are implemented, and their performance are evaluated on a simulation model. A large number of queries in five different query groups are selected for experimentation, and the algorithms performance are presented in the cases of both having a sufficient and an insufficient number of available processors. The results show that the proposed algorithms *DPAA*, *MPPPA*, and *OPA* always provide better performance than those of the traditional methods, *Intra* and *Phase*, and the performance improvements are significant. The adaptive method, *OPA*, guarantees to provide a global optimal solution with a sufficient number of processors. When the system is relatively small, the problem of processor allocation is identified as an NP complete problem. However, even in this situation *OPA* still provides a near global optimal solution by offering local phase optimisation.

Three multiple queries processing algorithms, the intra-query-only parallel processing (*IQO*), the phase-inter-query parallel processing (*PIQ*), and the critical path scheduling (*CPS*), are selected for experimentation dealing with a number of dependent queries, and their dependency is represented by a directed graph. In this upper level of query processing, the query execution time of a single query is conducted using *OPA* intra-query algorithm. The query dependency logic follows that of examples I and II in Figure 8.19, and more experiments are carried out using example II by varying the input parameters of each query. The experimental results show that *CPS* provides a global minimal execution time when there is a sufficient number of processors, and it always outperforms two existing algorithms, *PIQ* and *IQO*.

The remainder of the chapter is organised as follows. Section 9.2 describes the simulation model and Section 9.3 provides an overview of the intra-query allocation algorithms. Experimental design issues are discussed in Section 9.4 and the simulation results are presented in Sections 9.5 and 9.6 with a sufficient and an insufficient number of processors. Multiple dependent queries experimentation is introduced in Section 9.7. The chapter is concluded in Section 9.8.

9.2 Simulation Model

A simulation model has been constructed to evaluate the performance of the processor allocation algorithms. The processors in the simulation are assumed to be identical in

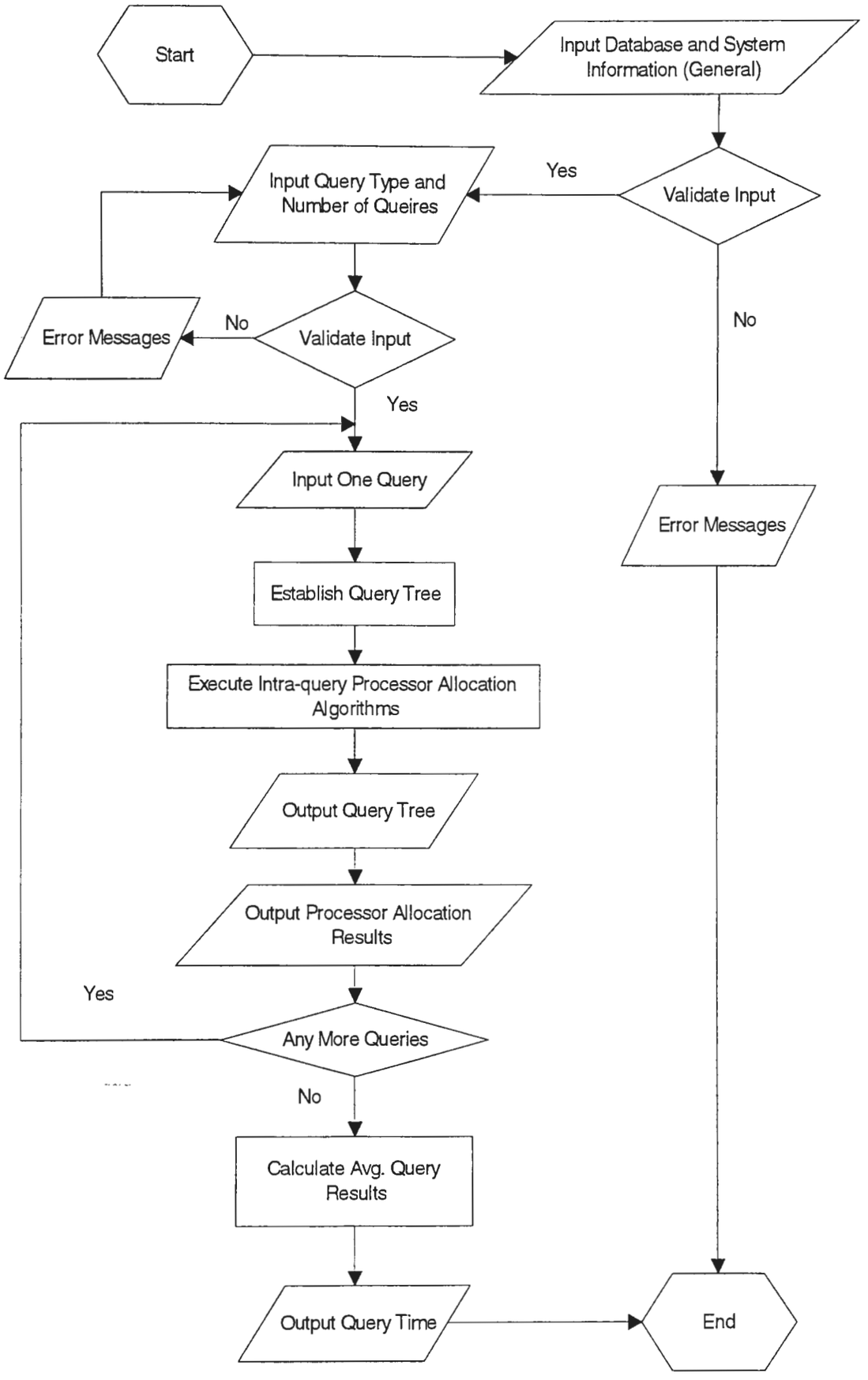


Figure 9.1: Simulation Model for Intra-query Processor Allocation

processing capacities and connected by a dynamic cross bar as shown in Figure 6.1. The issues on data placement and system architecture are discussed in Section 6.2.

Parameters	**Values**
Cardinality of relations	1,000 to 16,000
Number of processors n	4 to 256
Number of base relations	6 to 14
Number of joins per query	5 to 13
W_1, W_2, W_3	0.01×10^{-2}
Number of group queries	5
Number of queries in each group	3 to 10
Degree of data skew θ	0 to 1.0
T_{init}	0.1×10^{-2}
T_{hash}	0.01×10^{-2}
Overlapping factors β, γ	1.0
Percentage of joins with data skew	0% to 60%
T_{data}	0.003×10^{-2}

Table 9.1: Default Parameters Settings of Optimal Processor Allocation

The simulator is written in C++ programming language and runs on a Sun workstation. For intra-query parallelism the input to the simulation is a set of base relations and queries, and the output consists of the global and local phase execution times as well as the average query execution time of all queries in each query mix. The flow chart of the simulation model is shown in Figure 9.1 (see Appendix C). In the simulation, queries are stored in a tree structure and tree traversal always follows a post tree order. The maximum number of operation is 24 and the maximum number of phases in phase-based processor allocation approach is also 24. The default parameters' settings are listed in the Table 9.1 [Leun93].

For multiple query execution, the simulation is an extension of that of the intra-query processing and again the simulator is coded in C++ programming language running under Sun workstation. The input includes multiple queries and their dependency; the intra-query processing simulator can provide on-line information on databases and systems, and individual query. However, here, the queries are stored in the graph structure instead of the tree structure and query dependency is represented using a two-dimension matrix. The flow chart of the simulation model for multiple dependent queries processor allocation is

shown in Figure 9.2. The assumptions are that each query cost model is known, i.e. the query processing time is a function of number of processors employed, and the intra-query parallel processing has been conducted using *OPA*. The output is the processor assignments and the multiple queries execution time.

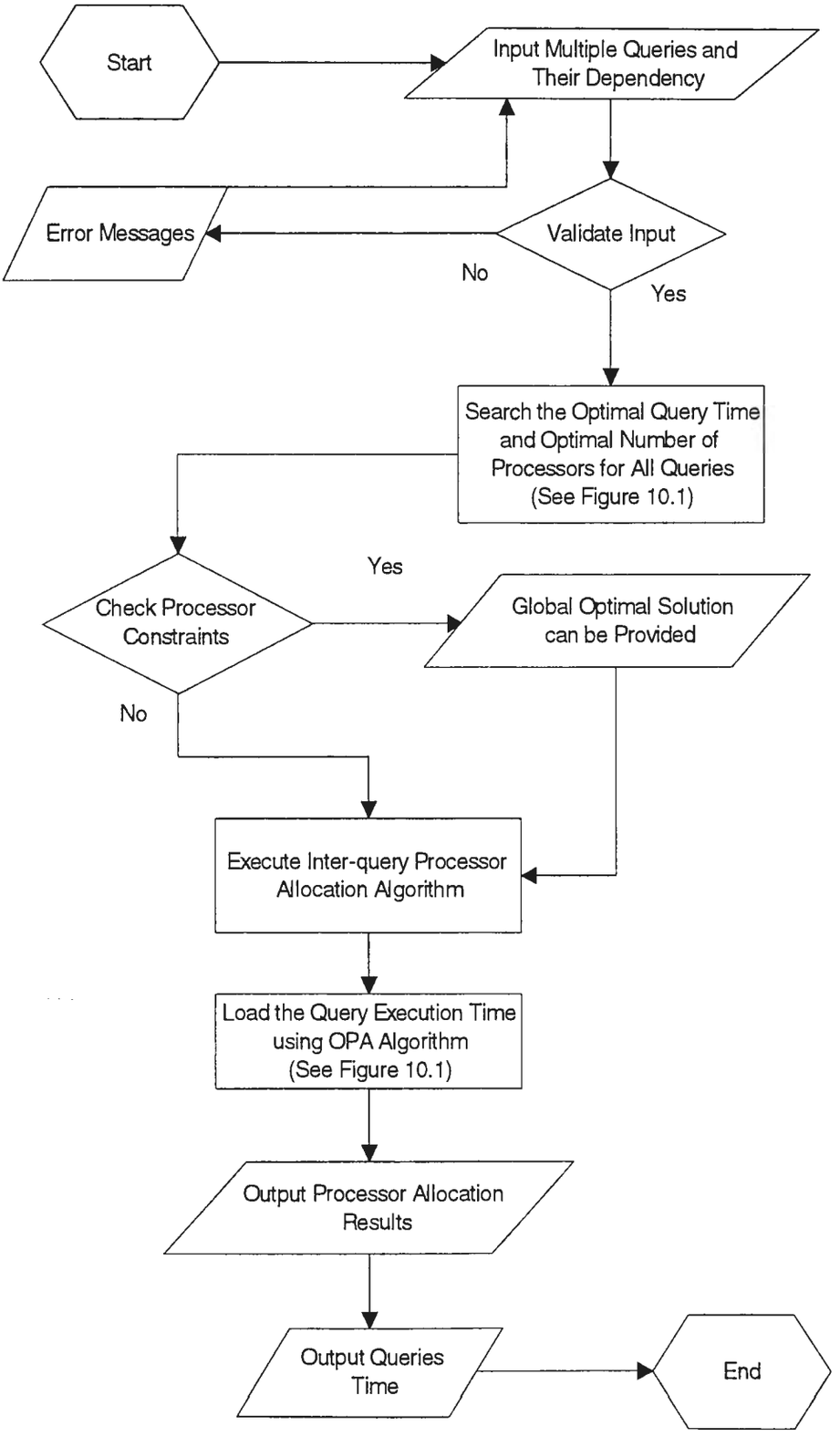


Figure 9.2: Simulation Model for Multiple Dependent Queries Processor Allocation

9.3 Intra-query Processor Allocation Algorithms Overview

For evaluation purposes, five algorithms are implemented in the simulation and they are *Intra*, *Phase*, *DPAA*, *MPPPA*, and *OPA*. All five algorithms have been discussed in Chapter 8. Among them, time equalisation is employed in the phase-based approaches and the degree of parallelism is implemented in all algorithms. *Intra* and *DPAA* are non-phase-based approach whereas *Phase* and *MPPPA* are phase-based approach. *OPA* is a hybrid approach based on *DPAA* and *MPPPA* making use of the processors bounds on achieving query optimal time.

9.4 Intra-query Experimentation Design

A large number of queries are selected and they are categorised into five data sets with the relation cardinality varying from 1000 to 16000 tuples. Data set 1 comprises three left-deep tree queries with the number of joins varied from 5 to 9. Data set 2 includes three right-deep tree queries with the number of joins varied from 5 to 9. Data set 3 contains balanced-bushy-tree queries with the number of joins varied from 7 to 13. Data set 4 is made up of 3 unbalanced-bushy-tree queries with the number of joins varied from 9 to 11. Finally, data set 5 constitutes ten mixed queries with the number of joins varied from 5 to 13. The examples of data sets are shown in Figure 9.3 and the fifth data set type is a combination of the above four kinds of data sets.

For two relations *R* and *S*, the corresponding estimate for the size of the join is the product of the relation sizes divided by the product of the domains for each of the variables that are arguments of both *R* and *S*. The domain of a variable can frequently be treated as the largest of the domains of the arguments in which that variable appears [Ullm89]. Frequently, we express the estimate of the join size as the product of the sizes of the relations being joined, times a parameter of the join, called the join selectivity factor. Figure 9.4 shows a query example with 10 join operations and 11 input relations, and the join selectivity factor is 1 divided by the summation of the operand relation sizes[16]. For

[16] This assumption has been widely used and accepted especially in the fields of query processing and join size estimation [Ullm89].

simplicity, all join selectivity factors in the query are assumed to be the same, and once one of them changes all other selectivity factors change correspondingly in the simulation.

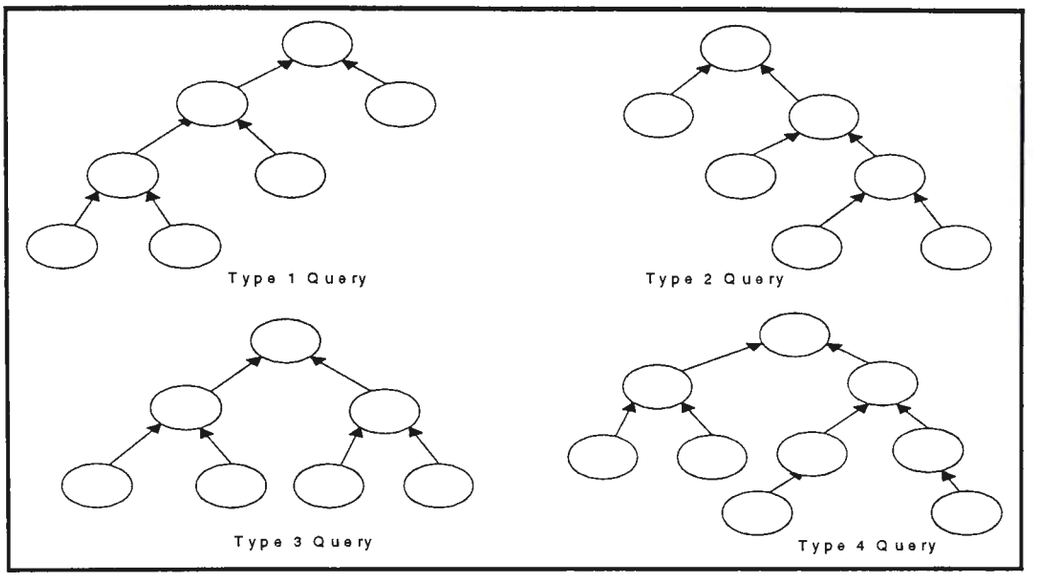


Figure 9.3: Examples of Data Sets

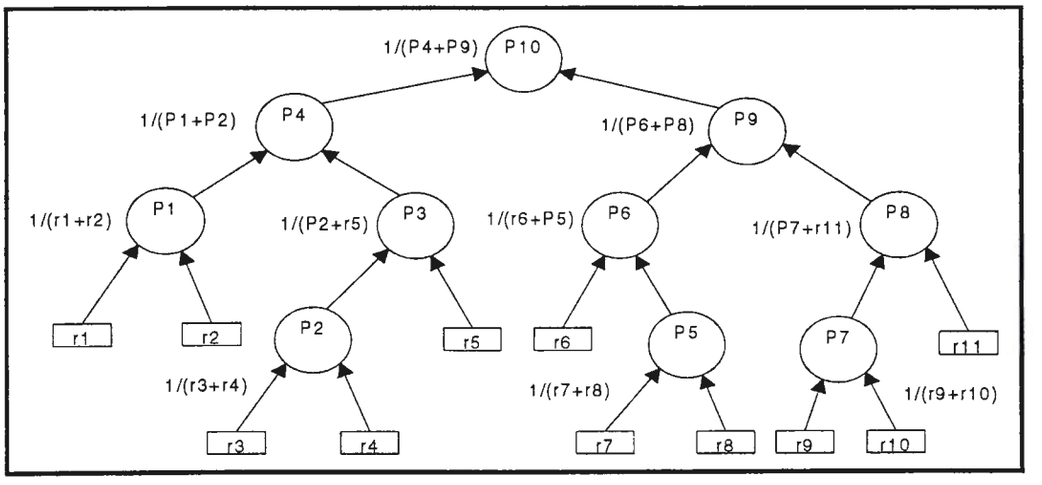


Figure 9.4: The Selectivity Factor in A Query Tree

To implement the degree of skewness, we make use of the skew factor θ which is a floating point value among 0 and 1. An example of skewed query is shown in Figure 9.5 with 12 operations where the shaded circles represent skewed operations and the number associated with the operation node is the degree of skewness.

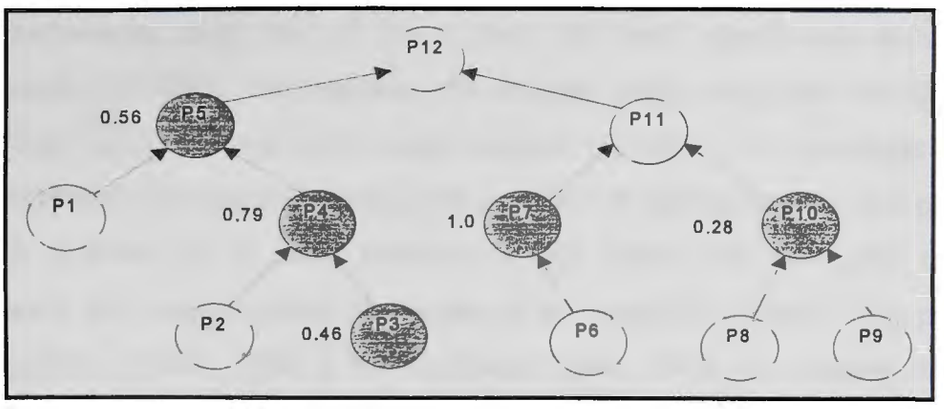


Figure 9.5: An Example of Skewed Query

9.5 Sufficient Number of Processors

Sufficient number of processors means that the optimal number of processors for each operation can always be supplied by the system processor pool according to the data flow of the query tree graph. Assume that in the simulation the sufficient number of processors is 256. With a sufficient number of processors, a global optimal solution for a query is achievable, and thus the non phase-based approaches provide minimal query execution time.

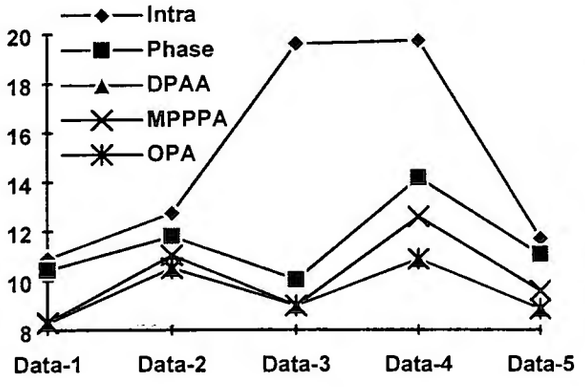


Figure 9.6: Different Query Groups

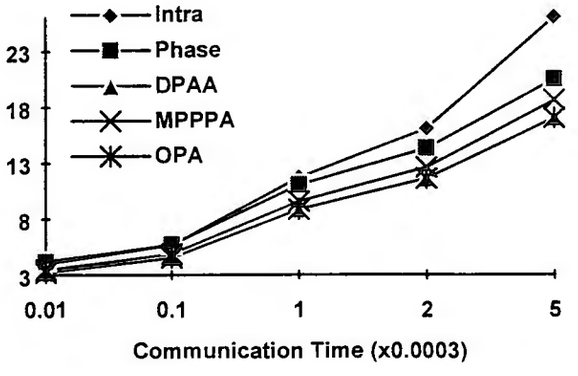


Figure 9.7: Effect of Communication Time

9.5.1 Different Query Groups

The five algorithms' performances are shown in Figure 9.6 with five different data sets. As we can see from the figure, *Intra* always has the poorest performance especially with data sets 3 and 4 because the number of operations in the phase-oriented approaches is nearly even-distributed in the bushy-tree parallelism. In addition, *MPPPA* constantly provides

better performance than that of *Phase* since the *time equalisation* technique is implemented in *MPPPA*. The improvements are quite stable except that for data set 1, where *Phase* gives a worse performance because the number of operations in each execution phase is distributed unevenly with a number of operations in the first phase and only one operation in all other execution phases. *DPAA* and *OPA* give the same performance, and as such, when the number of processors is sufficient they provide a global optimal solution. With a balanced-bushy query, there is a slightly difference between *MPPPA* and *DPAA*. For all groups of queries, experimental results match our prediction that *OPA* provides the minimal query execution time.

9.5.2 Communication Time

More experiments have been conducted using data set 5 since it consists of more queries and mixes four kinds of query types. Figure 9.7 shows the effect of increasing communication time. When the communication time is comparatively low, *Intra* gives better performance than that of *Phase* because *Intra* involves heavy inter-processor communication time. Again, *OPA* and *DPAA* offer the best performance despite the changing of communication time cost, and the new algorithms, *OPA*, *DPAA*, and *MPPPA* outperform the traditional algorithms, *Intra* and *Phase*.

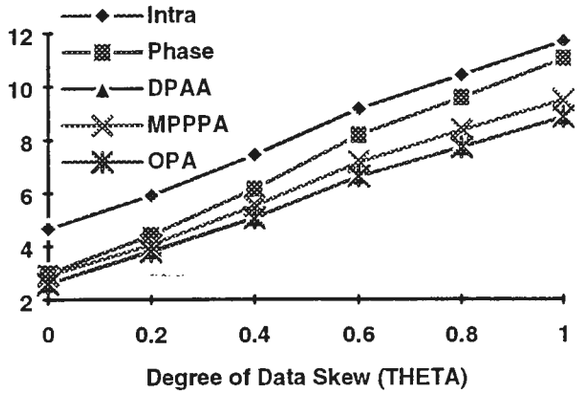


Figure 9.8: Degree of Data Skew

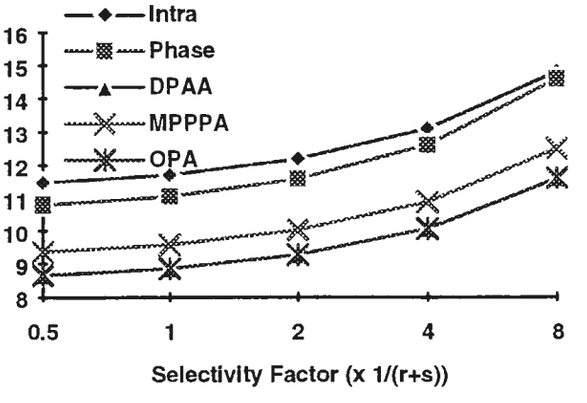


Figure 9.9: Selectivity Factor

9.5.3 Degree of Data Skew

To measure and model the load imbalance over multiple processors, the degree of data skew is introduced and its effect on performance is shown in Figure 9.8. In the figure, when the degree of data skew is raised, the query execution time of all algorithms are

increased. *Intra* gives the worst performance since it tends to involve a large number of processors while *Phase* provides a good performance in particular in the absence of data skew. When the degree of data skew is high, *Phase*'s performance is closing to *Intra*'s because it suffers from long idle time, i.e. low processor utilisation within each execution phase. *OPA* and *DPAA* give the best performance for all degrees of data skew.

9.5.4 Selectivity Factor

A given query consists of a number of relational operations such as selection or join and the *raw* input relation sizes are known. However, the intermediate result size is a run-time dynamic factor and fully depends on the operation selectivity which may have a significant impact on other inner operations in the query. In other words, the operation selectivity factor affects the query time by changing operand relation sizes. Figure 9.9 shows the algorithm's performance under the variation of selectivity factor. When the selectivity is high, i.e. the workload is heavy, the performance of *Phase* deteriorates more quickly comparing with that of *Intra*. The reason is that the heavy workload makes the ratio of computation to communication high and thus it is in favour of *Intra*. With a sufficient number of processors, *OPA* and the non phase-based approach *DPAA* outperform the phase-based approach *MPPPA*.

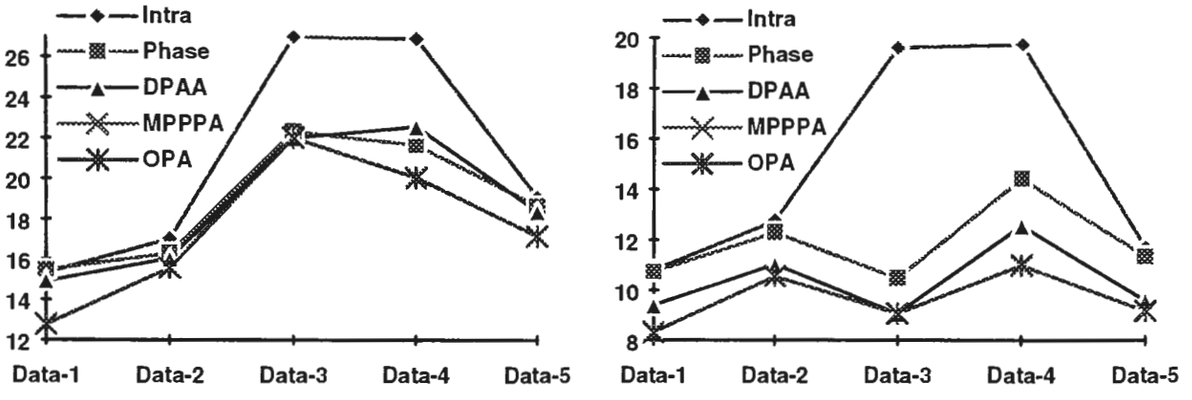
9.6 Insufficient Number of Processors

When the number of processors is limited, the optimal number of processors of the operations in the query tree can not be endowed. This situation is referred as the insufficient number of processors. In other words, hereafter, the resource is constrained but the objective of query time minimisation remains.

9.6.1 Different Query Groups

Figure 9.10 shows five algorithms comparison on five different data sets. When the number of processors is relatively small, e.g. 4 in Figure 9.10(a), *OPA* and *MPPPA* always outperform others. For data set 1, *Intra* produces a better performance than that of *Phase* since *Phase* does not provide an effective phase clustering and may not supply an efficient processor allocation strategy within each phase. *Phase* even performs better than *DPAA*

based on the experimentation on data set 4 with a small number of processors. With the balanced-bushy queries, all algorithms except *Intra* provide nearly the same performance.



(a) 4 processors (b) 64 processors

Figure 9.10: Different Query Groups

When the number of processors is relatively large, e.g. 64 in Figure 9.10(b), the performance starting from the best is in order of *OPA* and *MPPPA*, *DPAA*, *Phase*, and *Intra*. In the figure, clearly the improvements of *OPA* and *MPPPA* over other algorithms are in a wide range and the most amelioration gains in the fourth data set. Using the third data set, *MPPPA* presents the same performance as that of *DPAA* since the balanced-bushy-tree queries are highly likely to have the same operation execution sequence for both phase and non phase based approaches.

9.6.2 Effect of Number of Processors

Figure 9.11 shows that increasing the number of processors reduces the query execution time based on the fifth data set. *Phase* performs better than *Intra* when the number of processors is either very small or relatively large, i.e. there are two cross-over points. *OPA*, *MPPPA*, and *DPAA* outperform *Intra* and *Phase* despite the number of processors available. When the number of processors gets large the performance difference between *DPAA* and *MPPPA* is reduced, and thus it infers that a large number of processors is in favour of *DPAA*. Another interesting point is that the load reduction is large with a small number of processors but becomes marginal as the number of processors increases to a large number.

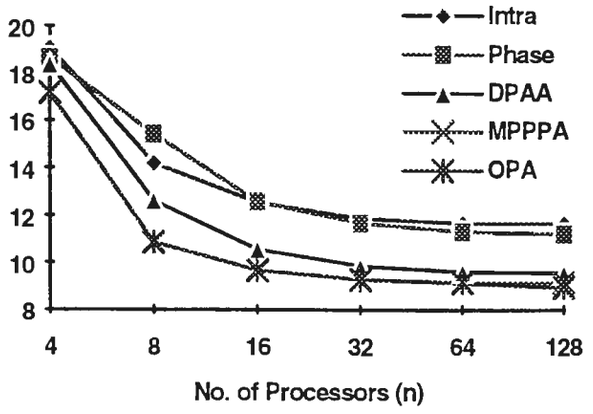


Figure 9.11: Effect of Number of Processors

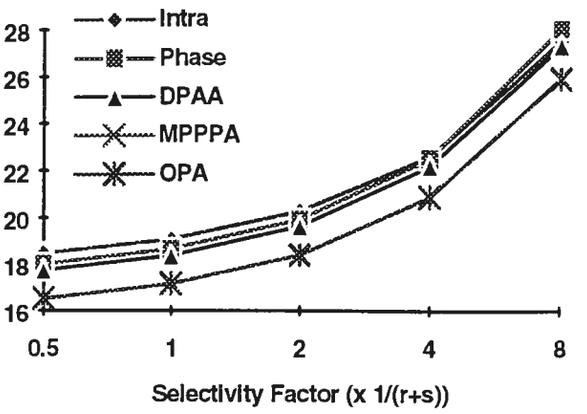


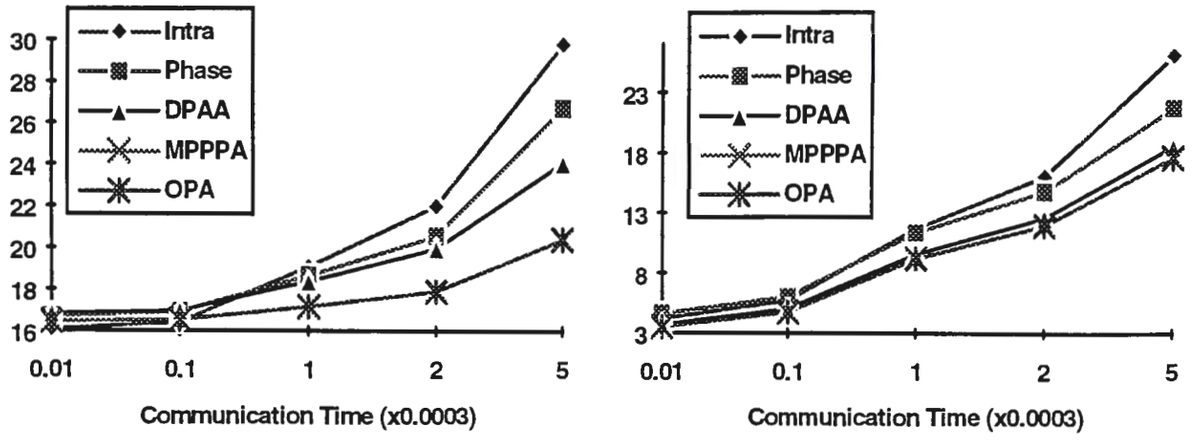
Figure 9.12: Selectivity Factor

9.6.3 Selectivity Factor

When the number of processors is 4, the effect of selectivity factor on performance is shown in Figure 9.12. In all cases, *OPA* and *MPPPA* provide the best results, and the improvement over traditional methods *Intra* and *Phase* is constantly above 10 percent. There is a cross-over point between *Intra* and *Phase* when the selectivity factor closes to $4/(r+s)$. *Intra* performs better when the selectivity factor is high whereas *Phase* provides better performance when the selectivity factor is low.

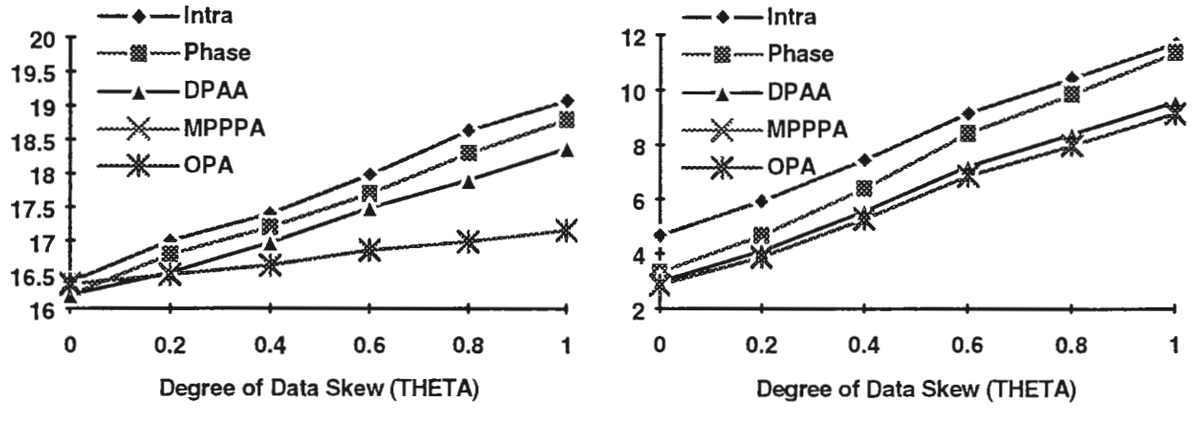
9.6.4 Communication Time

Increasing the data communication time encourages the local processing and inhibits inter processors communication. Figure 9.13 shows that increasing data communication time degrades the performance for all algorithms. With a small number of processors (see Figure 9.13(a)), *Intra* gives the best performance when the communication time is very cheap, and responses more quickly to the communication time than any other algorithms. *OPA* and *MPPPA* outperform *Phase* and *DPAA* in spite of the change of communication time cost. When the number of processors is increased to 64 in Figure 9.13(b), *OPA* and *MPPPA* outperform others but there is a cross-over point for *Intra* and *Phase*. High communication cost tends to bias in favour of *Phase,* whereas low communication cost tends to bias in favour of *Intra.*



(a) 4 processors (b) 64 processors

Figure 9.13: Query Execution Time vs Communication Time



(a) 4 processors (b) 64 processors

Figure 9.14: Query Execution Time vs Degree of Data Skew

9.6.5 Degree of Data Skew

Data skew causes the load imbalance over processors and thus it slows down the system as shown in Figure 9.14. Figure 9.14(a) shows the effects of data skew on query execution time with 4 processors. In the absence of data skew, *DPAA* and *Phase* have the best performance. In the presence of data skew in particular with a high degree of data skew, clearly, *MPPPA* and *OPA* provide better performance than other algorithms, and the improvement increases as the degree of data skew grows. Figure 9.14(b) shows the performance of the algorithms with 64 processors and *Phase* always performs better than *Intra*. As shown in both Figures 9.14(a) and 9.14(b), the changing of the degree of skewness has the same impact on *Intra* and *Phase*. When the number of processors is small, *DPAA* responses more quickly to the increasing degree of data skew than *MPPPA* and *OPA*.

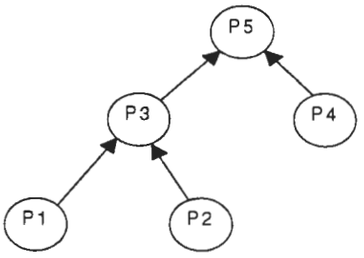
9.7 Multiple Dependent Queries

For experimentation in the inter-query level with data dependency, we implement two other existing multiple dependent queries processing algorithms. One is pure *intra-query-only* parallel processing (*IQO*) which executes queries sequentially with all available processors, and another is *phase-inter-query* parallel processing (*PIQ*) which group queries into execution phases based on the query dependency before processing. All algorithms assume that single query time is already optimised by exploiting intra-query parallelism.

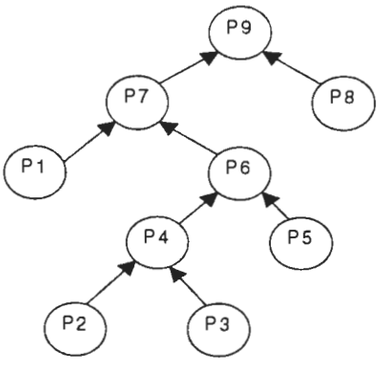
To simplify the implementation of multiple queries, we store the individual query execution results in files and make use of the algorithm shown in Figure 9.15 to extract the query response time on-line from query files (directly written by the intra-query simulator). This method provides a considerable simulation performance gain comparing to combining intra-query and inter-query processor allocation algorithms.

```
int k, temp_pro;
for (k=0; k<Max_Num_Query; k++) {
        if ( ... ) {
                ...;
                char buffer[20];
                sprintf(buffer,"queryfile%d.dat", k);
                a[k].fs.open(buffer,ios::in);
                a[k].fs>>temp_pro>>a[k].time;
                while ((temp_pro!=a[k].num_pro) && (temp_pro!=-1)) {
                        a[k].fs>>temp_pro>>a[k].time;
                        } //end of inner while
                a[k].fs.close();
                a[k].num_pro=temp_pro;
        } // end of if
        cout<<endl<<a[k].time<<", "<<k;
        ...;
} // end of for
```

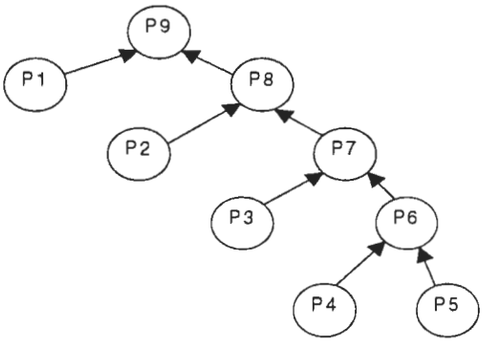
Figure 9.15: Dynamic Files Linking



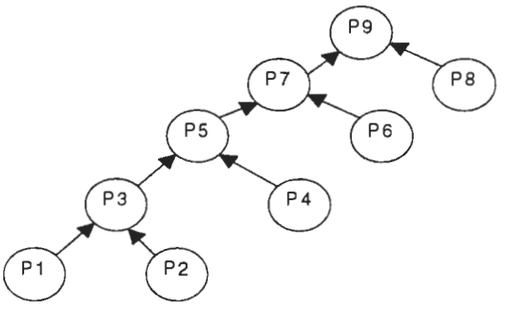
<1.87, 55>
Type 1 Query



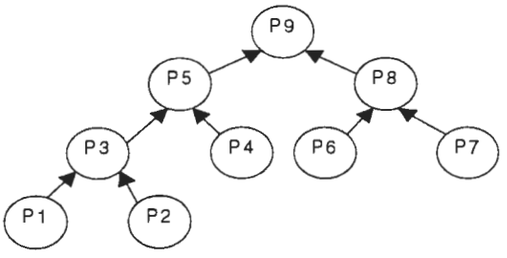
<12.85, 78>
Type 2 Query



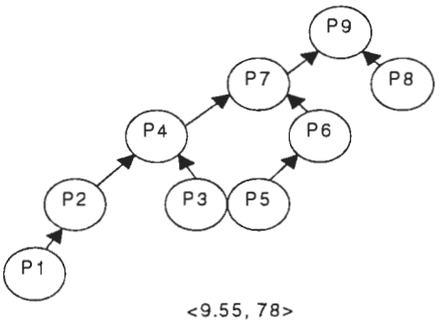
<10.49, 83>
Type 3 Query



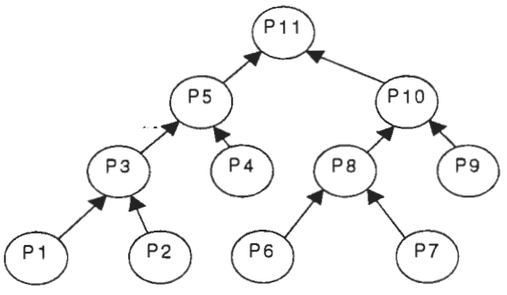
<12.71, 78>
Type 4 Query



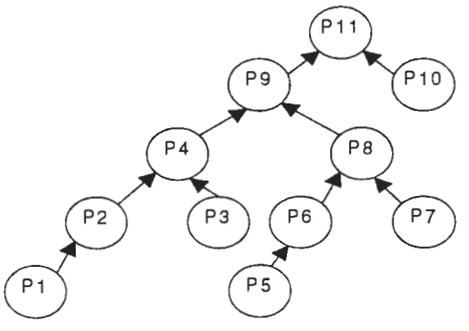
<9.24, 83>
Type 5 Query



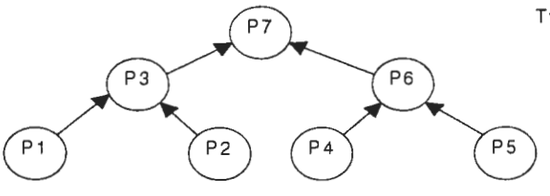
<9.55, 78>
Type 6 Query



<9.45, 36>
Type 7 Query



<11.86, 70>
Type 8 Query



<10.23, 14>
Type 9 Query

Figure 9.16: Multiple Queries for Experimentation

Based on examples I and II provided in Figure 8.19, the multiple queries execution time are displayed in Table 9.2. *CPS* provides a global minimal execution time, and *PIQ* gives better performance than that of *IQO* when the number of processors is sufficient. Going a bit further, we implement the three algorithms based on the query logic presented in the example II of Figure 8.19(b). The forming queries of example II are shown in Figure 9.16 where the angled numbers under each query graph are their optimal query execution time and optimal number of processors to provide the optimal time. As we can see that the 9 different types of queries are selected and they cover left-deep tree, right-deep tree, and bushy tree parallelism. The initial setting of the parameters is the same as that of Table 9.1.

	CPS	*PIQ*	*IQO*
Example I -- Type I Dependency	20.2	20.2	28.7
Example II -- Type II Dependency	20.3	22.4	54.4

Table 9.2: Three Methods Comparison with Sufficient Number of Processors

	CPS	*PIQ*	*IQO*
Sufficient No of Processors	33.95	88.26	38.56

Table 9.3: Experimentation Result of Example II with Sufficient Number of Processors

Assuming that the number of processors is sufficient, the experimental result is shown in Table 9.3. When a certain number of processors is involved, *CPS* performs as we discussed in Chapter 8. In the simulation, we output three tables, Tables 9.4, 9.5, 9.6 with 32 processors. In the Table 9.4, the activity analysis of the input is provided; in the Table 9.5, with the processor constraint the first level decompression is conducted and the result is that all paths are finished closely; in Table 9.6, due to the further processors shortage, the second level decompression is carried out until the number of available processors is sufficient.

Still based on the dependency logic of examples I and II, more experiments are conducted. Figure 9.17 shows the results when there is a sufficient number of processors. *CPS* always gives the best performance despite varying the types of dependency logic. The performance of *IQO* is extremely poor when queries are loosely inter related, i.e. acyclic query dependency, and most of the queries are able to proceed concurrently. This is also because *PIQ* is a phase based approach where in this situation it can group queries into phases evenly. Figure 9.18 shows the performance comparison by varying the number of

processors when the number of processors available is insufficient. When the number of processors is small, *IQO* performs better than *PIQ*; when the number of processors is relatively large, *PIQ* gives better performance than that of *IQO*. *CPS* outperforms the other two despite the varying number of processors. Another interesting point observed is that the algorithms performance also depend on the query dependency logic. In example I, in general *IQO* provides a much better performance than that of *PIQ*; however, in example II, *PIQ* constantly performs well especially with a large number of processors because there are several queries in each execution phase and the number of queries in each phase is evenly distributed.

Activity	**Arrow**	t_{op}	n_{op}	*EST*	*LST*	*EFT*	*LFT*	*TF*	*FF*	*IF*	**Remarks**
Q1	0-1	1.87	55	0	7.51	1.87	9.38	7.51	0	7.51	---
Q2	0-2	12.85	78	0	0	12.85	12.85	0	0	0	critical
Q3	0-3	10.49	83	0	2.05	10.49	12.54	2.05	0	2.05	---
Q4	1-4	12.71	78	1.87	9.38	14.58	22.09	7.51	7.51	0	---
Q5	2-4	9.24	83	12.85	12.85	22.09	22.09	0	0	0	critical
Q6	3-4	9.55	78	10.49	12.54	20.04	22.09	2.05	2.05	0	---
Q7	4-5	9.45	36	22.09	24.5	31.54	33.95	2.41	2.41	0	---
Q8	4-6	11.86	70	22.09	22.09	33.95	33.95	0	0	0	critical
Q9	4-7	10.23	14	22.09	23.72	32.32	33.95	1.63	1.63	0	---
number of processors			364			optimal execution time				33.95	

Table 9.4: Activity Analysis

Activity	**Arrow**	t_{op}	n_{op}	*EST*	*LST*	*EFT*	*LFT*	*TF*	*FF*	*IF*	**Remarks**
Q1	0-1	2.61	8	0	0.08	2.61	2.69	0.08	0	0.08	---
Q2	0-2	12.85	78	0	0	12.85	12.85	0	0	0	critical
Q3	0-3	11.56	11	0	0.08	11.56	11.64	0.08	0	0.08	---
Q4	1-4	19.4	5	2.61	2.69	22.01	22.09	0.08	0.08	0	---
Q5	2-4	9.24	83	12.85	12.85	22.09	22.09	0	0	0	critical
Q6	3-4	10.45	14	11.56	11.64	22.01	22.09	0.08	0.08	0	---
Q7	4-5	11.62	10	22.09	22.33	33.71	33.95	0.24	0.24	0	---
Q8	4-6	11.86	70	22.09	22.09	33.95	33.95	0	0	0	critical
Q9	4-7	11.32	7	22.09	22.63	33.41	33.95	0.54	0.54	0	---
number of processors			192			optimal execution time				33.95	

Table 9.5: The First Level Decompression

Activity	Arrow	t_{op}	n_{op}	*EST*	*LST*	*EFT*	*LFT*	*TF*	*FF*	*IF*	Remarks
Q1	0-1	8.75	1	0	1.97	8.75	10.72	1.97	0	1.97	---
Q2	0-2	13.12	22	0	2.15	13.12	15.27	2.15	0	2.15	---
Q3	0-3	14.63	5	0	0	14.63	14.63	0	0	0	critical
Q4	1-4	19.4	5	8.75	10.72	28.15	30.12	1.97	1.97	0	---
Q5	2-4	14.85	6	13.12	15.27	27.97	30.12	2.15	2.15	0	---
Q6	3-4	15.49	5	14.63	14.63	30.12	30.12	0	0	0	critical
Q7	4-5	12.41	9	30.12	31.26	42.53	43.67	1.14	1.14	0	---
Q8	4-6	12.26	17	30.12	31.41	42.38	43.67	1.29	1.29	0	---
Q9	4-7	13.55	6	30.12	30.12	43.67	43.67	0	0	0	critical
number of processors			32			execution time				43.67	

Table 9.6: Second Level Decompression of 32 Processors

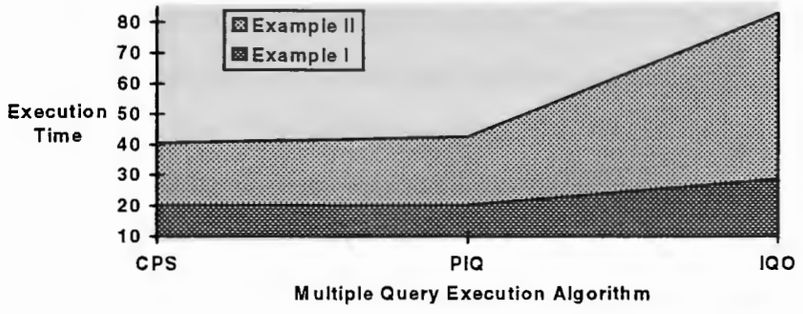


Figure 9.17: Sufficient Number of Processors

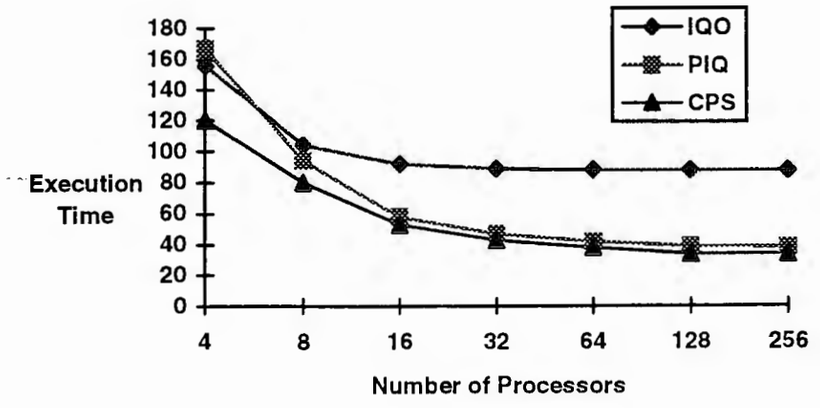


Figure 9.18: Varying Number of Processors

9.8 Summary

In this chapter, we have evaluated the performance of the processor allocation algorithms. Here, the main issues are the number of processors, communication time, selectivity factor, and degree of data skew. At the intra-query level, the proposed methods *OPA*, *DPAA* and *MPPPA* provide better performance than the traditional methods *Intra* and *Phase* based on the experimentation on a large number of queries in five different query groups. The phase-based approaches *Phase* and *MPPPA* perform well when there is a sufficient number of processors, and the non phase-based approaches *Intra* and *DPAA* provide better performance when there is an insufficient number of processors. *OPA* is an adaptive method based on *MPPPA* and *DPAA*, and always gives the best performance.

We have also presented a performance study on three multiple query processing algorithms focusing particularly on multiple dependent queries. The simulation based on two query dependency logic has been carried out and the results show that *CPS* is superior to *PIQ* and *IQO* in all cases. The intermittent result tables of *CPS* algorithm are also presented to illustrate how it performs. *PIQ* provides better performance than that of *IQO* when the number of available processors in the system is large or when there are a large number of concurrent queries due to the dependency in the queries graph. In contrast, *IQO* involves the least complexity and offers a reasonable performance only when there are a small number of concurrent queries and a small number of available processors.

In addition, the experimentation further demonstrates that the processor bound provided in Section 8.6.2 is able to indicate when a global optimal solution is achievable.

CHAPTER 10

EMPIRICAL STUDY OF SKEW MODEL ON PARALLEL SYSTEM

10.1 Introduction

In this chapter, we implement the skew model presented in Chapters 4 and 5 on a parallel client-server system with synthetically generated databases. First, we highlight the characteristics of the DEC Alpha server architecture and its operating environment for parallel processing in Section 10.2. The issues on how to justify the configuration of Alpha system and how to simulate a large number of processors in such a system are dealt

with in Section 10.3. The generation of synthetic databases is described in Section 10.4. Performance analysis of the skew model is presented in Section 10.5, and load skew and operation skew prediction are evaluated with data skew modelled by the Discrete Uniform, Zipf, and Normal Distributions. Finally, we conclude the chapter in Section 10.6.

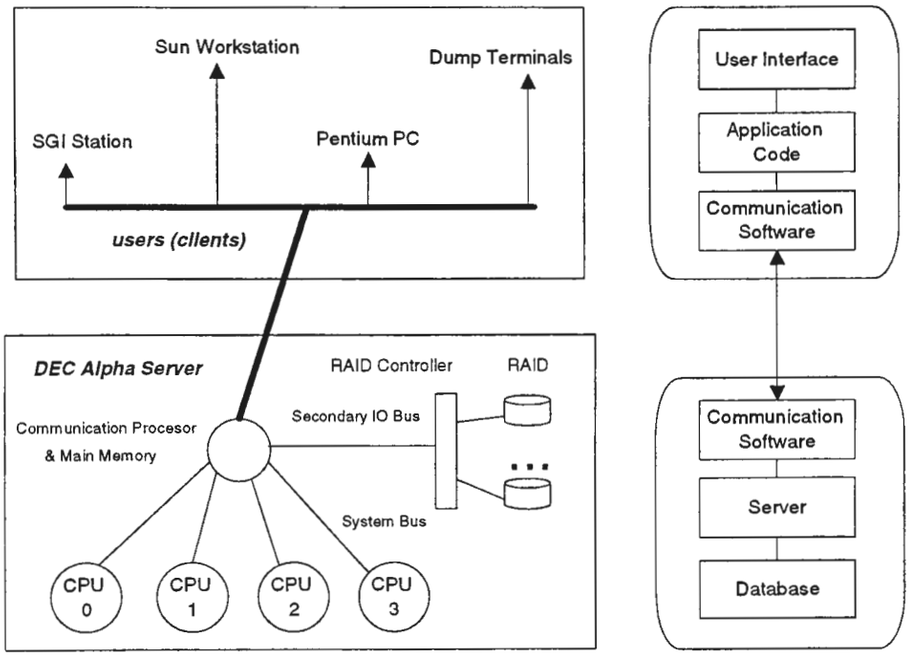


Figure 10.1: Client-Server Parallel System

10.2 Parallel System Experimentation Environment

Within the system, most of the operating system functions are implemented in user processes so that to request a service such as reading a block of a file, a client process sends the request to a server process which carries out the work and transfers back the information (see Figure 10.1). Therefore, with the shifting the application code in clients, the kernel only handles the communication between clients and server. The advantages of this approach are enhanced data sharing, centralised management, integrated servicing, data interchangeability and interoperability, and location independence of data and processing. The database is stored in Redundant Array of Inexpensive Disks (RAID) and the testing database is generated using the synthetic methods [Gray94].

The Alpha server consists of four Alpha 64-bit processors and each of them has a CPU clock speed of 190 Mhz. The server offers SMP, industry-standard PCI and EISA I/O, 256

MB RAM, 600 MB CD ROM Drive and 8 GB capacity 4mm DAT Tape Drive.

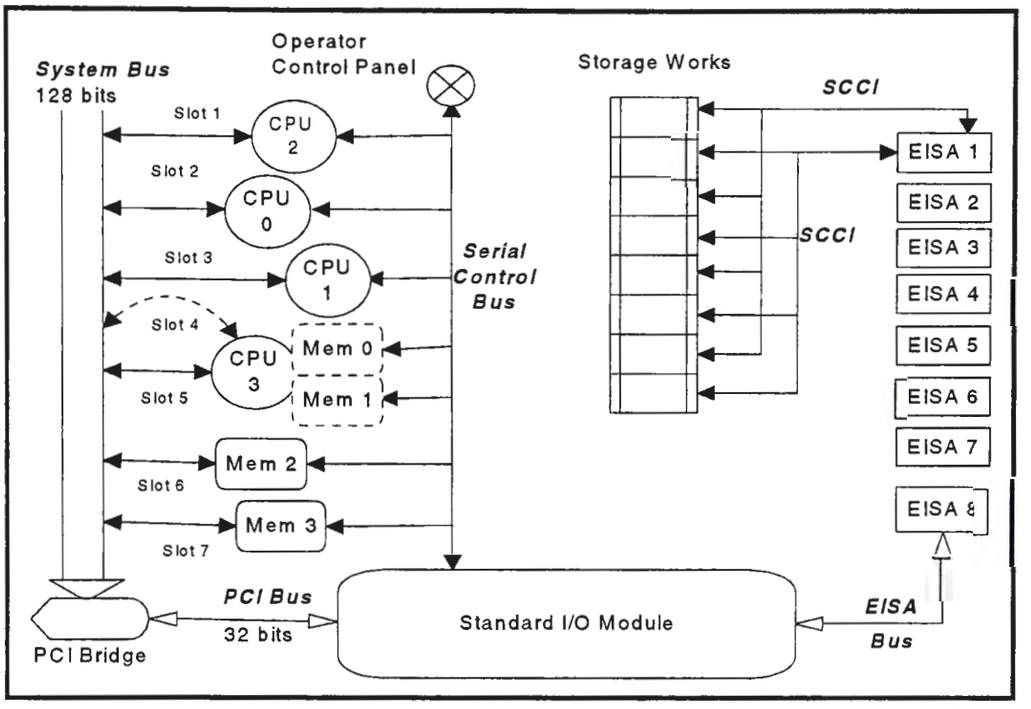


Figure 10.2: Alpha System Architecture

Figure 10.2 shows the system architecture of Alpha Server. The Alpha microprocessor is one of the leading RISC 64-bit processors and provides the floating-point unit supporting both DEC and IEEE floating point data types. All instructions are 32-bit long and have a regular instruction format. Each processor module has an interface to the system bus which is the primary interconnect among CPU, memory and I/O subsystems. The system bus is a limited-length, nonpended, synchronous, 128-bit wide multiplexed address and data bus with central arbitration, and is capable of delivering a peak data transfer bandwidth of 667 MB/Sec. A PCI I/O bus with three slots supports 32-bit PCI options and EISA options (via a PCI to EISA bridge) giving a bandwidth of 132 MB/Sec; an EISA secondary I/O bus with eight slots operates at 8.33 MHz giving a bandwidth of 32 MB/Sec [Digi95b]. The system has one internal BA35E Storage Works shelf with the hardware RAID Disk controller.

The memory subsystem is interleaved and can support up to four memory modules for a total of 256 MB of memory. A minimum of one memory module is required and the memory is available in 64 MB and 128 MB. In the implementation, we adopt the shared memory architecture so the memory is addressed uniformly.

On the Alpha server, Digital UNIX serves as the operating system. Previously known as DEC OSF/1, Digital UNIX is a 64-bit advanced kernel architecture based on CMU's Mach V2.5 kernel design with components from BSD 4.3 and 4.4, UNIX System V. The current version of Digital UNIX (Ver 3.2C) provides the symmetric multiprocessing (SMP) which enables systems containing two or more processors to execute the same copy of the operating system, access common memory, and execute instructions simultaneously [Digi94, Digi95a].

Closely related to our focus, the operating system also offers processor affinity and unattended reboot, and provides multiple threads from the same or different tasks running concurrently on different processors. In addition, it is able to stop and start a specified nonboot processor and there are no architectural limits on the number of CPUs supported. There are five SMP modes which can be configured at system boot time and during the implementation the optimised SMP mode is set.

10.3 Experimental Design

One way to implement parallel processing is through parallel compiler where separate programs need to be created for the host, the coordinator processor, and the worker processors. As a result, different programs running on different CPUs and final result has to be consolidated at the host. For instance, the Shiva system, originally designed by *Defence Science and Technology Organisation* (*DSTO*), is an Intel i860 based system consisting of one single *master processor unit* (*MPU*) and a number of *slave processors* (*SP*). The host of the system is a SUN Sparc IPX Station and the communication is through the SBus which provides communication bandwidth of 80 MB/Sec. For processing, application programs are cross compiled on the host and downloaded to *MPU*. Then, based on the partitioning information, *MPU* sends various parts of the binary code to the local memories of *SP*s.

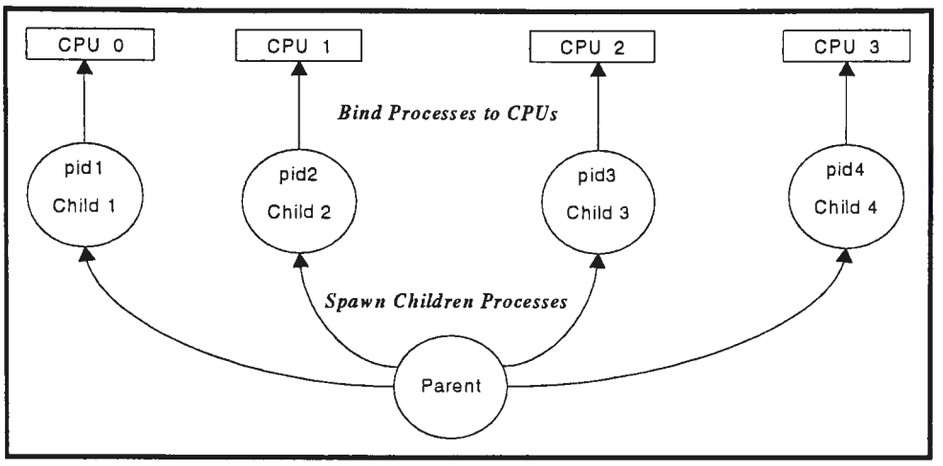


Figure 10.3: Parallel Processing on Multiple Processors

On the DEC Alpha system, Digital UNIX provides support for multithreaded applications where multiple threads are created in one single program. On top of this, it provides system calls to bind processes to CPUs. In the implementation, we employ higher level system calls[17], and the implementation language is DEC C for Digital UNIX Systems. As shown in Figure 10.3, the parent process in the main program creates four children processes using the *fork*( ) system call. Then, the children processes are bound to CPUs and subsequently interprocess communication channels are established between the parent process and all children processes.

10.3.1 Inter-processor Communication with Pipes

There are several possible *inter-process communication* (*IPC*) solutions on Digital UNIX such as files, shared file pointers, FIFOs, messages, pipes, semaphores, signals, shared memory, and process tracing, and they differ in the type and amount of information communicated between processes, ease of communication, reliability of communication, and efficiency. In general each method has some applications for which it is well-suited, and in the implementation pipes have been used because of their simplicity and elegancy. The idea is that the output of one process is sent directly to the input of the other process through the file descriptors.

---

[17] In fact, Digital UNIX can be treated as a parallel operating system with the option of processor affinity.

```
#include <stdio.h>
#include <sys/signal.h>
#include <sys/types.h>
#include <sys/resource.h>
#define CPU_3 0x8
#define CPU_2 0x4
#define CPU_1 0x2
#define CPU_0 0x1

main( int argc, char *argv[])
{
    int pfd[2], nread, i;
    char fdstr[10];              /* character string for file descriptor */
    char strg1[512];             /* storage fpr messages */
    int loop;                    /* message size */
    int bytes;                   /* number of messages for transimission */
    int pid;
    loop = atoi(argv[1]);        /* convert argument to number */
    bytes = atoi(argv[2]);       /* convert argument to number */
    if (pipe(pfd) == -1)
            fputs("Error in Pipe",stderr);
    switch (pid=fork())
    {
            case -1:
                    fputs("Error in fork",stderr);
                    exit(1);
            case 0:
                    if (close(pfd[1] == -1))
                            fputs("Error in Close",stderr);
                            /* convert number to char */
                    sprintf(fdstr, "%d", pfd[0]);
                    if (execlp("./childp","childp",fdstr,argv[1],NULL)==-1)
                            fputs("Error in Exec",stderr);
                    break;
            default:
                    printf("\nPlease Wait -- We are doing the binding now\n");
                    if (bind_to_cpu(pid, CPU_2, BIND_NO_INHERIT)) {
                            kill (pid, SIGKILL); exit(1); }
                            /* creating message */
                    for (i=0; i<bytes; i++)
                            strg1[i]='a';
                    /* sending message */
                    for (i=0; i<loop; i++)
                            if (write(pfd[1],strg1,sizeof(strg1))==-1)
                                    fputs("Error in Write",stderr);
                    printf("\nI am at the end of piping\n");
    } /* end of switch */
} /* end of main   */
```

Figure 10.4: *IPC* file Main_pipe.c

```
#include <stdio.h>
#include <sys/resource.h>
#include <sys/sysinfo.h>

main( int argc, char *argv[])
{
    int fd, nread, i;
    int loop;
    char s[512];
    long cpu_num;
    printf("\nWe are the Children -- How are you");
    getsysinfo(GSI_CURRENT_CPU, &cpu_num, 0L, 0L, 0L);
    printf("\nThis child running on CPU %d\n", cpu_num);
    fd=atoi(argv[1]); /* convert file desc to an int */
    loop=atoi(argv[2]);          /* convert num_loops to an int */
    for (i=0; i<loop; i++)
    {
            switch(nread=read(fd, s, sizeof(s)))
            {
                    case -1:
                            fputs("Error in Read",stderr);
                            exit(1);
                    default:
                            printf("read %d bytes: %s\n",nread, s);
            } /* end of switch */
    } /* end of for loop */
    return;
} /* end of main */
```

Figure 10.5: *IPC* file Childp.c

The limitation of the method is that two piped processes must be related so that they can share a file descriptor. In a shared memory architecture children processes only need to communicate with their parent process and thus one-way communication channel is sufficient for passing information. The example codes for the reader and writer are listed in the Figures 10.4 and 10.5, and it is assumed that a parent process writes the data to a child process which reads it at the end of the pipe.

10.3.2 Mutual Exclusion (*ME*)

Mutual Exclusion (*ME*) is designed for the critical section problem which occurs when two or more concurrent processes access the common resource or data. The solution is to allow only one process to use the shared resource while all other processes must wait until it becomes available. As such, any systems making use of multiple threads of control, need to consider mutual exclusion between the threads. However, to achieve high performance in a multiple processors system, system calls and kernel activities are allowed to occur on every processor and the kernel workload can be distributed throughout the system. Therefore, the uniprocessor *ME* solutions on avoiding race

conditions fail to work properly mainly because multiple processors executing in the kernel simultaneously violate the assumption on supporting short-term *ME*. In addition, *ME* with interrupt handlers may not function as we desired since the interrupt handlers only affect the processes priority on the processors that they are executed on, and do not affect interrupts delivered to other processors.

```
typedef int lock_unit;
int test_and_set (int *addres)
{   int old_value;
    old_value=swap_atomic(addres, 1);
    /* swap_atomic is the most basic single atomic read-modify-write instruction and
    such an instruction swaps a value stored in a register with a value in memory */
    if (old_value==0)
            return 0;
    return 1;
}

void intlock (lock_unit *lock_status)
{   *lock_status=0;
}

void lock (lock_unit *lock_status)      /* this is an automatic locking routine */
{   while (test_and_set (lock_status)==1)
    ;
}

void unlock (lock_unit *lock_status)
{   *lock_status=0;
}

/* Implementing a critical section with spin lock which only allows one processor at
 a time to change the lock status from 0 to 1 */
lock (&spin_lock);
critical section code
unlock(&spin_lock);
```

Figure 10.6: Algorithm for Mutual Exclusion

There are many complex *ME* methods on *SMP*, and here we implement a simple short-term *ME* solution. This method reinstates the uniprocessor assumption on multiple processors system by requiring that all kernel execution occur on one physical processor. This physical coordinator processor is referred as the master and all other processors in the systems are named as the slaves. The method is known as *Master-Slave ME* method (*MSME*). As a result, in user mode, processes may execute on any processor in the system but they will be switched to the master processor whenever they involve system calls. Moreover, all interrupt handlers and device drivers run only on the master processor. The algorithm for *MSME* is listed in Figure 10.6, and it does not provide a good performance if long-term *ME* is expected because it makes use of spin locks, and processors waiting for

the lock do not perform any useful work while spinning. However, in the implementation, skew modelling considers the distribution result with the synthetic relations so the performance is not the utmost important issue.

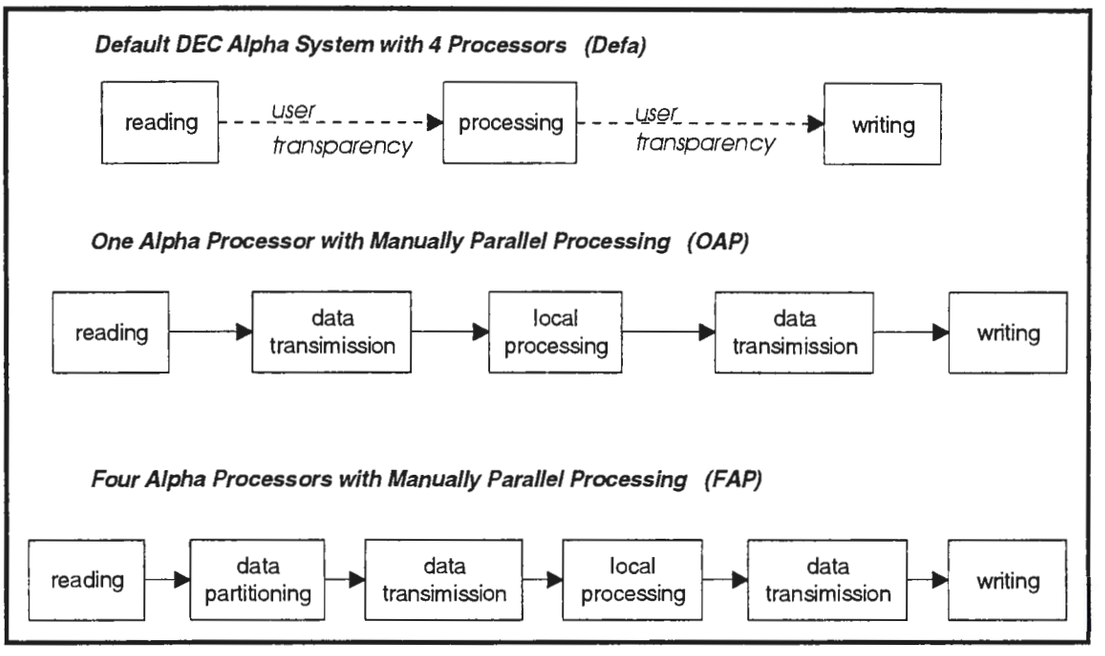


Figure 10.7: Processing Procedures on DEC Alpha

10.3.3 Database Processing Procedure

Using the DEC Alpha system, we conduct experiments based on three different methods involving CPUs as shown in Figure 10.7. Method I carries out processing on default DEC Alpha system with four CPUs so that the files are read into memory for processing and are written back to disks for storing. From the user point of view, the operating system is really a parallel operating system which takes care of parallel processing and provides full user transparency. Method II involves only one CPU at a time, i.e. sequential processing. At the beginning of processing, files are still read into memory and then transmitted to one CPU through the communication channel. Processing is conducted locally at that CPU and the processing result is transferred back to main memory before writing to disks. Method III is processing in parallel on four CPUs and does it manually with full user control. Files are read into memory and sent to CPUs according to a partitioning function. After local processing at each processor in parallel, the processing results are transferred back to main memory. At the end, the results are output from memory to disks.

Table 10.1 shows the elapsed time comparison for reading and writing based on the above

three methods. In the experimentation, the database file has 2500 tuples with 80 bytes/tuple and the file size is 208 Kbytes. Assume that *the file size for writing is one fourth of that for reading*, and the size of each communication message is 512 bytes. To test reading and writing, high level system calls such as *fscanf*( ) and *fprintf*( ) are employed since in commercial database systems these operating system calls are also used for loading and storing data. To reduce the variation in on-line environment, each operation has been conducted 100 times and the average is reported. Based on the results shown in Table 10.1, all three methods have the same reading and writing time and this agrees with our prediction since the server is a shared memory system.

	Defa	*OAP*				*FAP*
CPU No.	unknown	0	1	2	3	all 4 CPUs
reading	1.6112	1.6046	1.6201	1.6129	1.6023	1.6109
writing	0.4741	0.4753	0.4713	0.4723	0.4743	0.4751

Table 10.1: Reading and Writing Time Comparisons (time units)

Accurately timing processing data on CPU has been proven a difficult task because a simple comparison such as IF-ELSE consumes a very short time and is far quicker than that of reading and writing. The reason is Alpha microprocessor has a powerful floating point unit and the system memory has 256 MB. The results in Table 10.2 are based on processing simple floating point operations, and the selectivity factor at each processor is 0.25, i.e. only one fourth of the forwarded data are transferred back to main memory from processors and one fourth of the read data are written back to disks. Furthermore we assume that there is no skew after data partitioning in *FAP*, and the collected data processing time is shown in Table 10.2. *Defa* gives 0.3044 for processing and transmission, and the detail of the processing procedure is transparent to users, e.g. which CPU has been used for which portion of the file, how many CPUs has been used for parallel processing, how the interprocessor communication is implemented. As indicated in the result, *Defa*'s execution time lies in between that of *OAP* and *FAP*. The sequential method *OAP* presents the longest elapsed time because it involves data transmission and only one CPU has been used for processing. *FAP* offers the best performance because it fully utilises all four CPUs in the system.

	Defa	*OAP*				*FAP*
CPU No.	unknown	0	1	2	3	all 4 CPUs
data part.	0.3044	--	--	--	--	0.0241
forward trans.		0.108	0.1101	0.1098	0.1291	0.0282
processing		0.3532	0.3672	0.3612	0.3498	0.0892 (0.0886,0.0892, 0.0883,0.0869)
backward trans.		0.0284	0.0261	0.0298	0.0290	0.0060
Total Time	0.3044	0.4896	0.5034	0.5008	0.5079	0.1475

Table 10.2: Processing Time Comparisons (time units)

A problem remains is how to simulate more than four processors on the existing DEC Alpha system. This can be solved with the idea of processes stacks where a processes stack is set up for each CPU and a number of processes can be put into one stack. The stack follows a *first in last out* order (*FILO*) and each CPU will execute one process at one time until the stack is empty. To avoid data sharing at each CPU with multiple running processes[18], the *wait*( ) system call is used to ensure only one active process at one time on each CPU. The penalty of this approach is of course longer execution time and there is also a limit of the number of processes which can be created at one time in memory. However, the objective of the implementation is to verify the skew model in which we only consider the number of tuples in each process but not the overall execution time nor the concurrency problem caused by data sharing. Figure 10.8 shows an experiment of simulating 16 processors with four CPUs and in the figure, there are four processes stacks with four processes on each stack.

10.4 Synthetic Database Generation

Synthetic database is a group of files or relations filled with dummy information but having certain statistical properties [Gray94]. In the implementation the synthetic databases are generated, and subsequently partitioned and sent to the processes for

[18] This introduces the problem of concurrency where semaphores must be implemented to protect data sharing.

processing. In Table 10.3, a synthetic database is generated with seven relations[19] and each relation consists of different number of tuples of various tuple sizes. In the table, the column of No. of Bytes is the file content size, i.e. how many characters are in the file, and the column of No. of Blocks is the actual file size when the file is created in Digital UNIX. The size of the No. of Blocks may be slightly larger than that of the No. of Bytes since the operating system requires extra blocks to store file information such as file *inode*. Reading and writing time are presented in Table 10.3 based on the average of 100 runs.

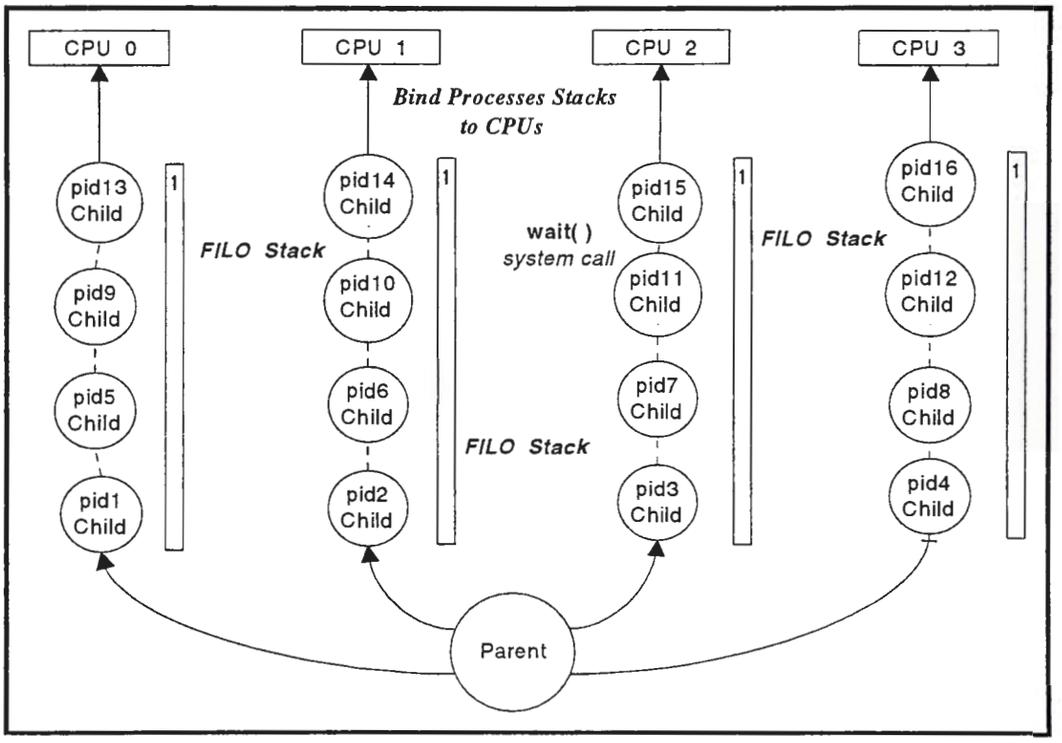


Figure 10.8: Using Processes Stacks to Simulate 16 Processors with 4 CPUs

The attributes of relation TenK are listed in Table 10.4, and on Digital UNIX, an integer is 4 bytes, a float is 8 bytes, a character is 1 byte, and a date type is 8 bytes. Hence each record requires 80 bytes. The column of cardinality is the number of unique domain values, i.e. 20 means that there are 20 distinct values out of 10000 values, and the column of range refers to the range of the domain values, e.g. with column 2 there are only two distinct values and they are 0 and 1. Unique1 is the primary key of the relation and this column is sorted.

[19] These synthetic relations simulate the example database EMP - DEPT of ORACLE 7.1.3 running on SUN SPARC server.

Relation	No. of Tuples	No. of Bytes	No. of Blocks	Reading Time	Writing Time
OneK	1,000	80,000	79 K	0.9131	0.9955
TwoK	2,000	160,000	168 K	1.3284	2.0314
FiveK	5,000	340,000	344 K	2.7570	3.7311
TenK	10,000	800,000	792 K	6.5152	8.794
TwentyK	20,000	1,600,000	1344 K	15.6784	31.5116
FiftyK	50,000	3,400,000	3336 K	28.1734	29.6847
HundK	100,000	6,800,000	6656 K	55.9047	77.2801

Table 10.3: A Synthetic Database with 7 Relations

Name	Type	Cardinality	Range	Order	Comment
unique1 (4)	integer	10000	0-9999	sorted	primary key
unique2 (20)	character	10000	--	random	candidate key
string1 (20)	character	10000	--	random	--
twenty (4)	integer	20	0-19	random	0,1...19,0,1...
date (8)	date	2000	--	random	e.g.14021996
thousand (8)	float	1000	0-999	random	0,1...999,0,1...
two (4)	integer	2	0-1	random	0,1,0,1,0,1...
string2 (18)	character	5000	--	random	--
hundred (4)	integer	100	0-99	random	foreign key

Table 10.4: Attributes Listing of Relation TenK

10.5 Performance Evaluation of Skew Model

To implement the skew model, the synthetic database has been generated and is stored on secondary disks. First, the relation is read into main memory and children processes are created in the parent process of the main program. The number of processes is related to the number of processors and subsequently children processes are bound to CPUs. Then, the communication channels between parent process and all children processes are established. The relation is partitioned on single attribute in the parent process and tuples are sent to the corresponding CPUs. The number of tuples are counted at each process and the result is reported back to the parent process. Children processes are terminated and the result is written to disk in the main program. In the following subsections, we describe the implementation of both unary and binary relational operations and data skew is modelled using Discrete Uniform Distribution, Zipf Distribution, and Normal

Distribution. Experimental results on operation skew are also collected for the parallel hash based join.

10.5.1 Load Skew Prediction without Data Skew

When the data skew follows discrete uniform distribution, we refer to this situation as no data skew. In the implementation, the degree of data skew is reflected by the selection of

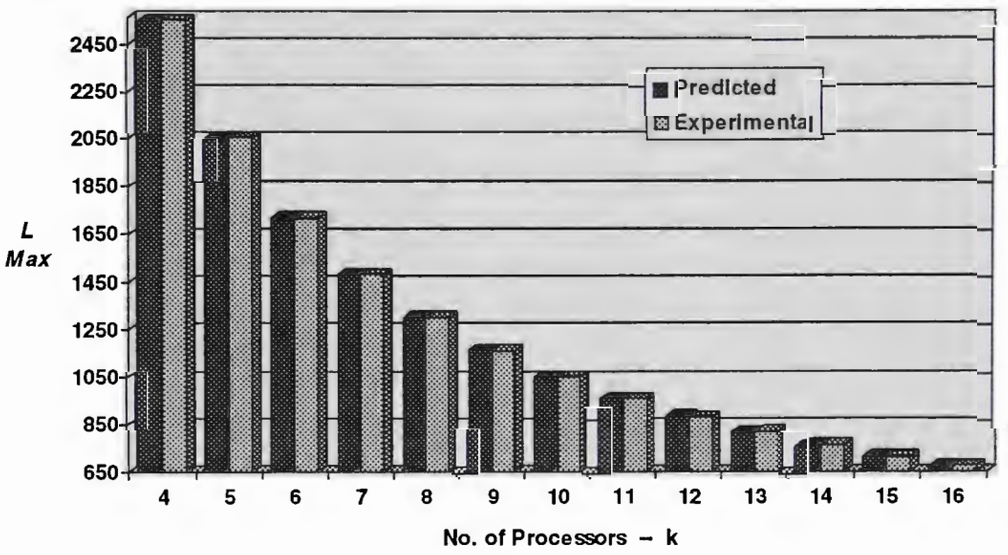


Figure 10.9: Maximum Load Prediction without Data Skew

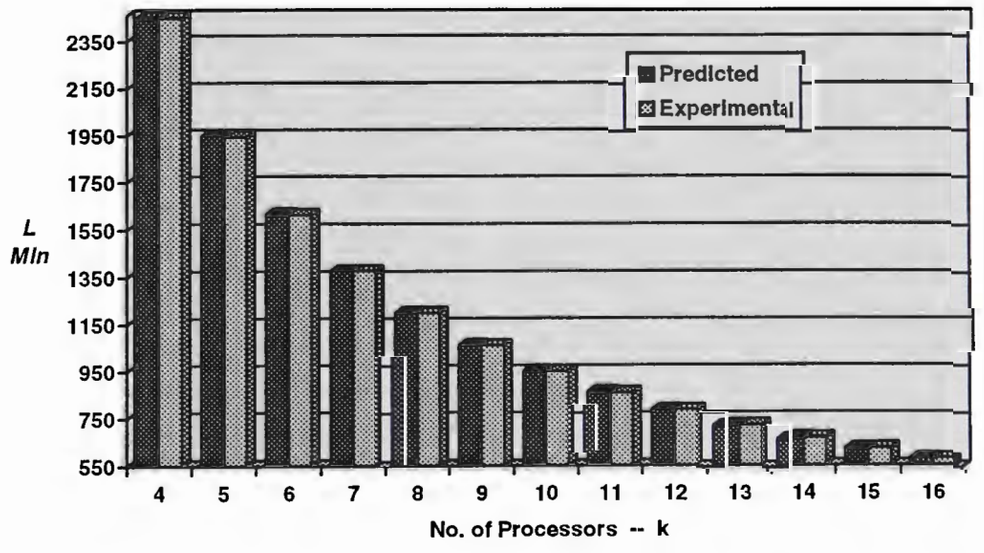


Figure 10.10: Minimum Load Prediction without Data Skew

4 processors	Maximum Load Prediction			Minimum Load Prediction		
	Predicted	**Experiment**	**RE-%**	**Predicted**	**Experiment**	**RE-%**
1000	265.78	268.19	0.90	234.22	234.65	0.18
2000	522.32	523.02	0.13	477.68	478.07	0.08
3000	777.34	777.86	0.07	722.66	722.37	0.04
4000	1031.57	1030.50	0.10	968.43	969.78	0.14
5000	1285.29	1286.42	0.09	1214.71	1213.64	0.09
6000	1538.66	1540.19	0.10	1461.34	1458.84	0.17
7000	1791.76	1791.21	0.03	1708.24	1710.11	0.11
8000	2044.64	2046.37	0.08	1955.36	1955.03	0.02
9000	2297.35	2300.41	0.13	2202.65	2195.10	0.34
10000	2549.91	2551.18	0.05	2450.09	2446.94	0.13
20000	5070.58	5076.78	0.12	4929.42	4927.25	0.04
30000	7586.45	7584.09	0.03	7413.55	7418.20	0.06
40000	10099.82	10101.58	0.02	9900.18	9901.69	0.02
50000	12611.60	12617.70	0.05	12388.40	12384.95	0.03
60000	15122.26	15113.55	0.06	14877.74	14874.73	0.02
70000	17632.05	17630.92	0.01	17367.94	17365.16	0.02
80000	20141.17	20144.01	0.01	19858.83	19852.44	0.03
90000	22649.73	22651.64	0.01	22350.27	22348.38	0.01
100000	25157.83	25156.74	0.00	24842.17	24842.58	0.00
200000	50223.21	50223.05	0.00	49776.79	49778.86	0.00
300000	75273.37	75250.78	0.03	74726.63	74728.96	0.00
400000	100315.66	100311.03	0.00	99684.34	99680.74	0.00
500000	125352.92	125352.42	0.00	124647.08	124660.63	0.01
600000	150386.61	150387.59	0.00	149613.39	149605.53	0.01
700000	175417.58	175418.77	0.00	174582.42	174558.77	0.01
800000	200446.42	200457.56	0.01	199553.58	199524.30	0.01
900000	225473.50	225477.72	0.00	224526.50	224506.20	0.01
1000000	250499.11	250539.36	0.02	249500.89	249478.41	0.01
2000000	500705.85	500710.63	0.00	499294.16	499259.72	0.01
3000000	750864.48	750919.19	0.01	749135.52	749122.31	0.00
4000000	1000998.22	1001008.75	0.00	999001.78	998963.25	0.00
5000000	1251116.03	1251174.50	0.00	1248883.96	1248884.75	0.00
6000000	1501222.56	1501280.38	0.00	1498777.44	1498752.38	0.00
7000000	1751320.52	1751342.50	0.00	1748679.49	1748625.00	0.00
8000000	2001411.69	2001469.13	0.00	1998588.31	1998527.50	0.00
9000000	2251497.32	2251509.75	0.00	2248502.68	2248491.25	0.00
10000000	2501578.32	2501664.75	0.00	2498421.68	2498378.75	0.00

Table 10.5: Load Skew Prediction without Data Skew on 4 Processors

partitioning attribute. In the case of no data skew, the primary key is chosen since it is the only unique identification of the relation, and it is guaranteed that there is no repeated values. Using 4 CPUs for the processors, the experimental results are shown in Table 10.5. Both maximum and minimum load values are collected in the implementation and their values are compared with the theoretical prediction based on the skew model. The first column of the table is the number of tuples in the relation, i.e. the relation cardinality, and relative errors are also calculated in columns 4 and 7. According to implementation, the skew model gives precise prediction on both maximum and minimum load in particular when the relation size is large.

Using processes stacks introduced in Section 10.4, we simulate the number of processors which varies from 4 to 16. The maximum load prediction without data skew is shown in Figure 10.9 and the corresponding minimum load prediction is shown in Figure 10.10. In both figures, the experimental values agree with the skew model, and as increasing the number of processors, both the maximum and the minimum load over processors are reduced.

10.5.2 Load Skew Prediction with Zipf Data Skew

With the Zipf data skew, we only implement the situation when the number of appearances of distinct domain values follows a pure Zipf distribution, and thus we pick up a non-key attribute in the synthetic relation as the partitioning attribute. The experimental results with 4 processors are listed in Table 10.6. All relative errors are less than 8.6% and larger than those of the case of without data skew. The reason is that the approximation on the Harmonic Number H_k by $\gamma + \ln k$ for simplicity used in the skew model. The experimental maximum values are always less than the predicted values while the experimental minimum values are greater than the predicted minimum values. This further agrees with our forecast because by simplifying Harmonic Number, the maximum load is increased and the minimum load is decreased in the skew model.

Varying the number of processors from 4 to 16, the maximum and minimum load comparisons are shown in Figures 10.11 and 10.12. The general trend is still the same with increasing number of processors reducing the skew values but the curves tend to more smooth or the load reduction is slower comparing those of Figures 10.9 and 10.10.

4 processors	Maximum Load Prediction			Minimum Load Prediction		
	Predicted	**Experiment**	**RE-%**	**Predicted**	**Experiment**	**RE-%**
1000	520.94	479.90	8.55	111.38	119.65	6.91
2000	1035.06	957.50	8.10	232.10	243.25	4.58
3000	1548.06	1438.96	7.58	354.36	362.68	2.29
4000	2060.48	1917.03	7.48	477.41	480.76	0.70
5000	2572.52	2394.20	7.45	600.97	601.67	0.12
6000	3084.30	2878.22	7.16	724.89	722.22	0.37
7000	3595.88	3365.42	6.85	849.09	838.70	1.24
8000	4107.30	3847.94	6.74	973.50	956.91	1.73
9000	4618.60	4323.85	6.82	1098.09	1076.98	1.96
10000	5129.78	4799.54	6.88	1222.82	1198.43	2.04
20000	10237.98	9586.68	6.79	2475.18	2403.50	2.98
30000	15342.62	14386.62	6.65	3732.39	3599.34	3.70
40000	20445.43	19202.61	6.47	4992.11	4800.68	3.99
50000	25547.06	24007.97	6.41	6253.45	5997.38	4.27
60000	30647.86	28816.67	6.35	7515.92	7186.33	4.59
70000	35748.03	33579.35	6.46	8779.26	8394.24	4.59
80000	40847.69	38390.61	6.40	10043.29	9605.72	4.56
90000	45946.95	43213.61	6.33	11307.87	10804.41	4.66
100000	51045.86	47996.16	6.35	12572.92	12022.79	4.58
200000	102023.48	95977.96	6.30	25239.23	23986.18	5.22
300000	152989.86	144048.69	6.21	37920.90	35977.92	5.40
400000	203950.43	191988.84	6.23	50610.52	48017.27	5.40
500000	254907.28	240005.08	6.21	63305.23	59983.32	5.54
600000	305861.50	288034.72	6.19	76003.54	72018.50	5.53
700000	356813.72	336060.69	6.18	88704.59	83978.19	5.63
800000	407764.35	383930.28	6.21	101407.81	96009.21	5.62
900000	458713.69	432106.63	6.16	114112.80	107996.44	5.66
1000000	509661.95	479939.09	6.19	126819.26	120022.22	5.66
2000000	1019108.06	959926.25	6.17	253933.83	240010.92	5.80
3000000	1528518.66	1439945.88	6.15	381096.98	360100.16	5.83
4000000	2037910.89	1920310.25	6.12	508285.27	479882.49	5.92
5000000	2547291.36	2400076.75	6.13	635489.64	599896.38	5.93
6000000	3056663.49	2880312.00	6.12	762705.43	719853.81	5.95
7000000	3566029.30	3360275.50	6.12	889929.87	839875.75	5.96
8000000	4075390.11	3839804.50	6.14	1017161.15	960060.75	5.95
9000000	4584746.82	4319890.50	6.13	1144398.03	1079979.63	5.96
10000000	5094100.11	4800244.00	6.12	1271639.60	1199912.88	5.98

Table 10.6: Load Skew Prediction with Zipf Data Skew on 4 Processors

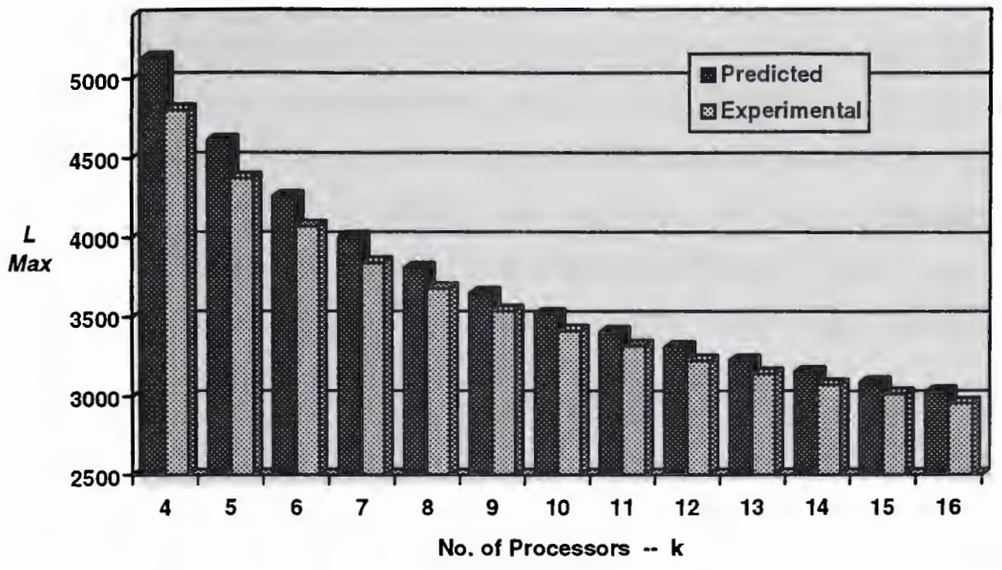


Figure 10.11: Maximum Load Prediction with Zipf Data Skew

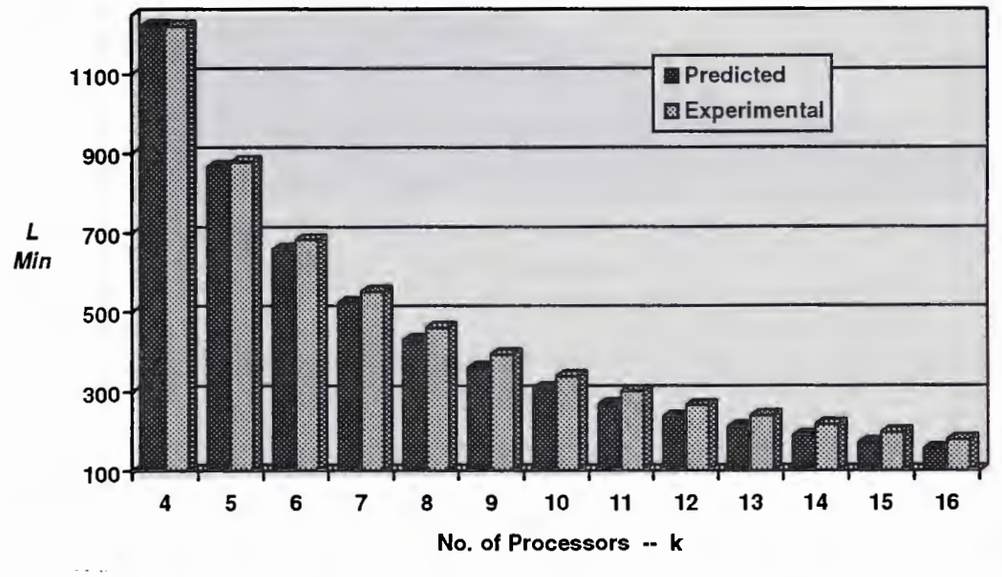


Figure 10.12: Minimum Load Prediction with Zipf Data Skew

10.5.3 Load Skew Prediction with Normal Distribution Data Skew

With the Normal Distribution data skew, the appearances of distinct values of attributes can be described using symmetrical Normal Distribution. Again, a non-key attribute has to be selected for partitioning within the synthetic relation since there are two valleys and a peak in the distribution. Using 4 CPUs for the processors, the maximum and minimum load comparisons are displayed in Table 10.7. The relative error on maximum load

prediction is bounded by 0.4% while the relative error on minimum load is less than 12.3%. There are two explanations on the high mean error for minimum load prediction. One is caused by the approximation on the standard normal distribution for generating skewness in the synthetic relation in the implementation, since integration in the standard normal distribution has been a complex and expensive task in any simulation or implementation and inappropriate selection of integration interval not only slows down the implementation but also drops the preciseness. Another reason is that the minimum load of normal distribution tends to

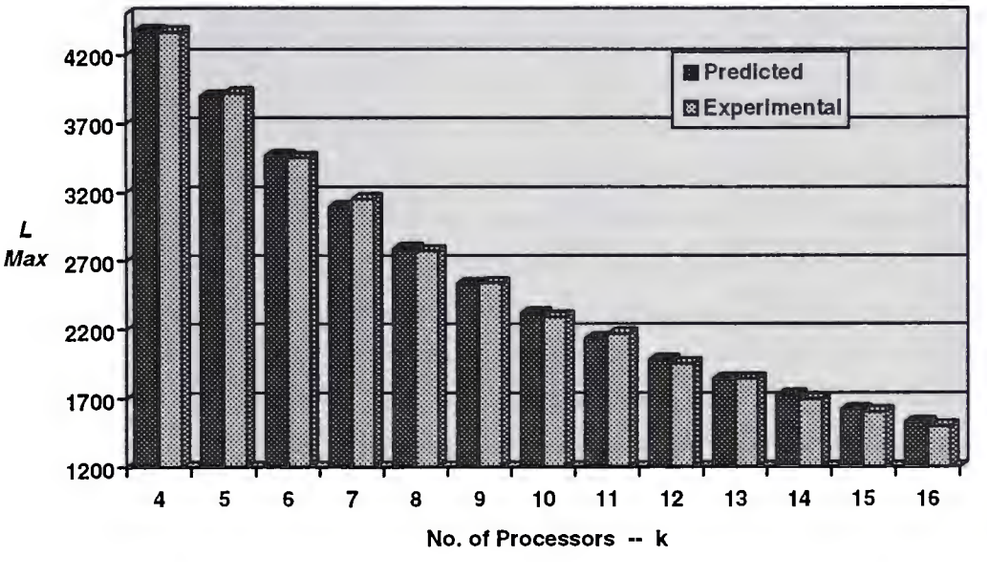


Figure 10.13: Maximum Load Prediction with Normal Distribution Data Skew

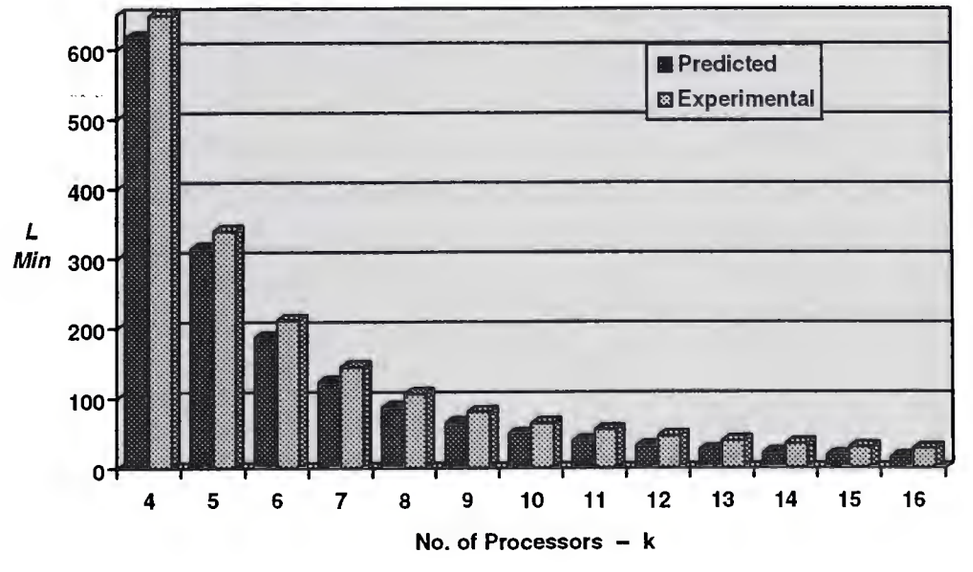


Figure 10.14: Minimum Load Prediction with Normal Distribution Data Skew

4 processors	Maximum Load Prediction			Minimum Load Prediction		
	Predicted	Experiment	RE-%	Predicted	Experiment	RE-%
1000	446.73	446.71	0.00	51.94	59.18	12.23
2000	885.85	882.59	0.37	112.27	123.90	9.39
3000	1323.72	1320.12	0.27	173.98	188.89	7.89
4000	1760.94	1757.78	0.18	236.40	257.78	8.29
5000	2197.75	2194.01	0.17	299.27	321.60	6.94
6000	2634.27	2629.29	0.19	362.47	385.65	6.01
7000	3070.56	3069.31	0.04	425.92	451.04	5.57
8000	3506.68	3500.41	0.18	489.56	517.76	5.45
9000	3942.65	3938.19	0.11	553.36	584.12	5.27
10000	4378.50	4372.96	0.13	617.29	647.40	4.65
20000	8732.95	8735.31	0.03	1261.10	1303.59	3.26
30000	13083.45	13099.09	0.12	1909.26	1963.91	2.78
40000	17431.90	17435.41	0.02	2559.69	2621.22	2.35
50000	21779.04	21786.96	0.04	3211.55	3283.04	2.18
60000	26125.25	26155.01	0.11	3864.44	3929.79	1.66
70000	30470.76	30479.73	0.03	4518.11	4605.72	1.90
80000	34815.71	34855.77	0.11	5172.39	5259.23	1.65
90000	39160.20	39186.18	0.07	5827.18	5909.19	1.39
100000	43504.31	43565.02	0.14	6482.38	6573.61	1.39
200000	86932.59	87065.19	0.15	13048.60	13175.78	0.97
300000	130348.35	130547.97	0.15	19628.61	19781.84	0.77
400000	173757.64	174096.31	0.19	26215.76	26390.28	0.66
500000	217162.79	217620.27	0.21	32807.47	33010.17	0.61
600000	260565.00	261035.88	0.18	39402.42	39637.97	0.59
700000	303964.98	304578.91	0.20	45999.83	46239.49	0.52
800000	347363.20	348105.03	0.21	52599.18	52836.14	0.45
900000	390759.97	391551.41	0.20	59200.12	59496.38	0.50
1000000	434155.54	435086.63	0.21	65802.40	66092.18	0.44
2000000	868070.62	870104.81	0.23	131869.89	132237.59	0.28
3000000	1301946.14	1305168.13	0.25	197981.00	198403.64	0.21
4000000	1735801.19	1740553.88	0.27	264114.69	264583.63	0.18
5000000	2169643.14	2175441.50	0.27	330262.81	330915.19	0.20
6000000	2603475.80	2610366.50	0.26	396421.18	397090.09	0.17
7000000	3037301.41	3030275.50	0.23	462587.31	463194.97	0.13
8000000	3471121.45	3481019.75	0.28	528759.59	529393.50	0.12
9000000	3904936.93	3915571.50	0.27	594936.89	595730.63	0.13
10000000	4338748.59	4350963.50	0.28	661118.40	661842.88	0.11

Table 10.7: Load Skew Prediction with Normal Distribution Data Skew on 4 Processors

produce small values. In other words, normal distribution has two deep valleys and the small number always tends to generate a large relative error.

Using 4 CPUs for 4 - 16 processors, the maximum and minimum load comparisons are shown in Figures 10.13 and 10.14. Both figures show a close agreement between the experimentation and skew model. In addition, the maximum load is nearly linear to the number of processors and the minimum load reduces sharply with increasing the number of processors.

10.5.4 Operation Skew Prediction in Parallel Hash Join

With operation skew in binary join, two synthetic relations are read into memory and they are partitioned one after another. The fragments are sent to children processes from the parent process. Hash based join will be conducted at each processor so that the complexity of the operation is directly related to the fragment size of operand relations, i.e. the operation skew is a linear summation operation of the load skew in input relations. However, whether the relation is skewed will affect the operation skew significantly. As such, we will implement the operation skew of parallel hash based join. The output from the implementation is the parallel join complexity in terms of the maximum and minimum number of tuples allocated over processors. Throughout this section, we assume that two relations have the same cardinality R=S, and we only consider the comparison at each processor not the IO bottleneck. In addition, we assume that if there is data skew in input relation it always follows a pure Zipf distribution.

When there is no data skew in both input relations, two synthetic relations of *one-to-one* relationship are loaded into memory. Both of them are partitioned using the primary key. Table 10.8 shows the implementation results of 4 processors. Experimental results closely agree with the predicted values in the skew model, and the relative mean error is less than 1.7% for maximum load and less than 2.8% for minimum load. Varying number of processors 4 - 16, the load comparisons between predicted and experimental are shown in Figures 10.15 and 10.16.

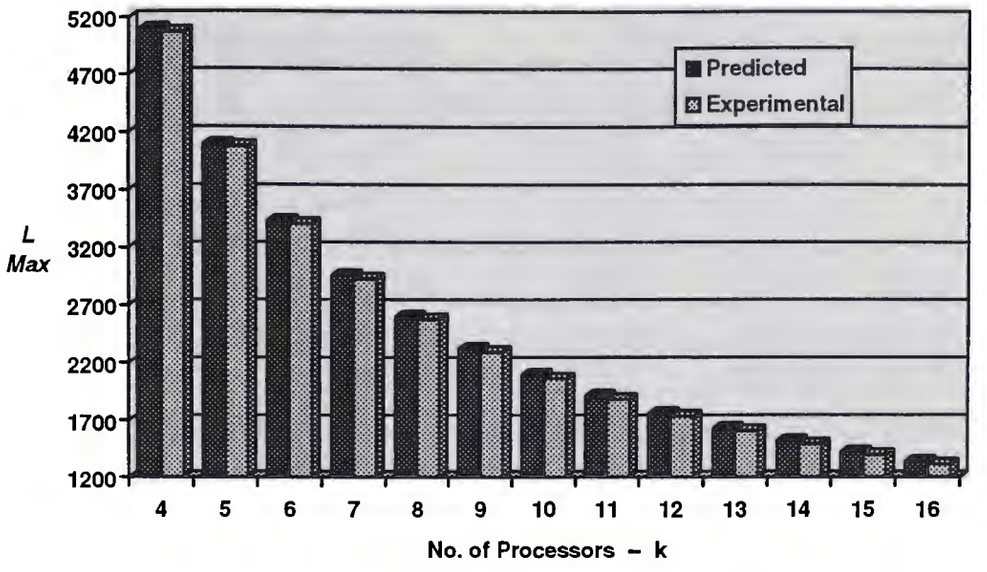


Figure 10.15: Maximum Operation Skew Prediction without Data Skew

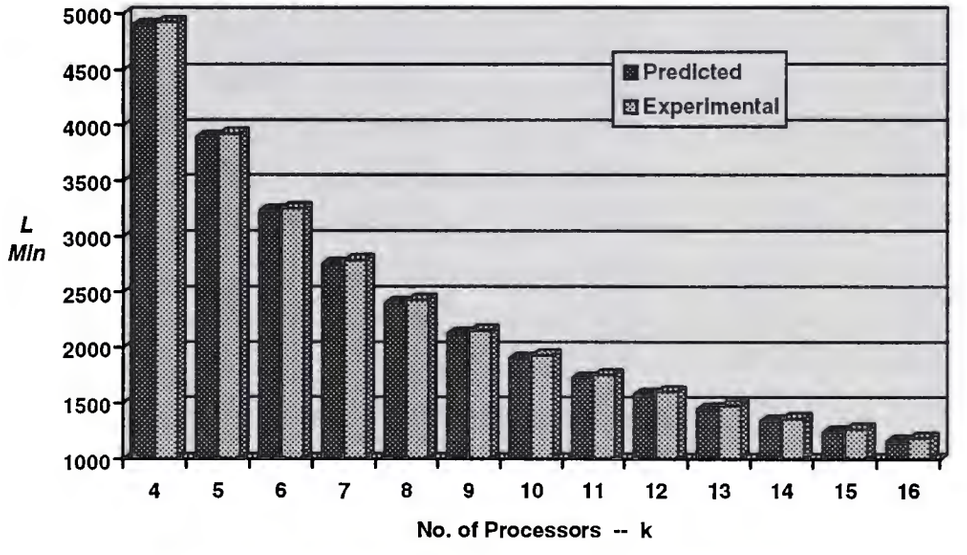


Figure 10.16: Minimum Operation Skew Prediction without Data Skew

To implement the situation of single data skew, the appearances of number of distinct domain values in one synthetic relation follow a pure Zipf distribution and those of another synthetic relation follow a discrete uniform distribution. Therefore, two synthetic relations of *one-to-many* relationship are selected, and one relation is partitioned on primary key and

4 processors	Maximum Load Prediction			Minimum Load Prediction		
R=S	Predicted	Experiment	RE-%	Predicted	Experiment	RE-%
1000	531.57	522.81	1.68	468.43	476.64	2.71
2000	1044.64	1029.47	1.47	955.36	967.53	1.82
3000	1554.67	1537.93	1.09	1445.33	1457.25	0.86
4000	2063.13	2046.03	0.84	1936.87	1954.16	0.84
5000	2570.58	2548.55	0.86	2429.42	2450.16	0.58
6000	3077.32	3055.83	0.70	2922.68	2941.72	0.15
7000	3583.52	3562.76	0.58	3416.48	3437.25	0.07
8000	4089.28	4062.83	0.65	3910.72	3938.77	0.08
9000	4594.70	4570.00	0.54	4405.30	4434.65	0.15
10000	5099.82	5076.38	0.46	4900.18	4925.56	0.22
20000	10141.17	10101.62	0.39	9858.83	9894.52	0.87
30000	15172.90	15123.51	0.33	14827.10	14878.15	0.93
40000	20199.64	20144.25	0.27	19800.36	19852.16	1.09
50000	25223.21	25160.14	0.25	24776.79	24846.45	1.14
60000	30244.51	30171.12	0.24	29755.49	29821.61	1.25
70000	35264.10	35201.23	0.18	34735.90	34793.52	1.41
80000	40282.34	40200.96	0.20	39717.66	39793.52	1.37
90000	45299.46	45233.72	0.15	44700.54	44772.82	1.45
100000	50315.66	50218.90	0.19	49684.34	49755.01	1.39
200000	100446.42	100345.78	0.10	99553.58	99645.70	1.55
300000	150546.75	150402.25	0.10	149453.25	149597.22	1.66
400000	200631.33	200470.09	0.08	199368.67	199521.08	1.75
500000	250705.85	250539.42	0.07	249294.15	249436.08	1.73
600000	300773.21	300585.03	0.06	299226.79	299430.13	1.76
700000	350835.17	350643.75	0.05	349164.83	349318.41	1.74
800000	400892.83	400684.44	0.05	399107.17	399320.09	1.79
900000	450946.99	450721.53	0.05	449053.01	449278.63	1.81
1000000	500998.22	500756.09	0.05	499001.78	499232.13	1.80
2000000	1001411.69	1000981.38	0.04	998588.31	998983.06	1.87
3000000	1501728.96	1501208.13	0.03	1498271.04	1498678.88	1.88
4000000	2001996.43	2001476.38	0.03	1998003.57	1998482.75	1.90
5000000	2502232.08	2501663.50	0.02	2497767.92	2498386.25	1.91
6000000	3002445.12	3001765.00	0.02	2997554.88	2998218.25	1.90
7000000	3502641.03	3501906.25	0.02	3497358.97	3497930.00	1.92
8000000	4002823.38	4002038.00	0.02	3997176.62	3997965.01	1.93
9000000	4502994.65	4502190.12	0.02	4497005.35	4497760.23	1.92
10000000	5003156.63	5002235.00	0.02	4996843.37	4997703.92	1.92

Table 10.8: Operation Skew Prediction of Parallel Hash Join without Data Skew on 4 Processors

4 processors	Maximum Load Prediction			Minimum Load Prediction		
R=S	Predicted	Experiment	RE-%	Predicted	Experiment	RE-%
1000	774.89	732.04	5.85	357.44	367.39	2.71
2000	1540.64	1462.68	5.33	726.52	740.01	1.82
3000	2304.90	2193.95	5.06	1097.52	1106.99	0.86
4000	3068.37	2924.52	4.92	1469.52	1481.94	0.84
5000	3831.34	3643.53	5.15	1842.15	1852.94	0.58
6000	4593.97	4380.03	4.88	2215.23	2218.52	0.15
7000	5356.32	5118.29	4.65	2588.65	2590.43	0.07
8000	6118.46	5835.34	4.85	2962.34	2960.06	0.08
9000	6880.43	6558.56	4.91	3336.25	3341.38	0.15
10000	7642.26	7293.59	4.78	3710.34	3702.09	0.22
20000	15255.62	14588.73	4.57	7457.53	7393.55	0.87
30000	22864.23	21884.48	4.48	11210.77	11106.95	0.93
40000	30470.39	29193.00	4.38	14967.16	14805.96	1.09
50000	38074.96	36500.15	4.31	18725.55	18515.30	1.14
60000	45678.43	43805.48	4.28	22485.36	22207.52	1.25
70000	53281.04	51102.40	4.26	26246.25	25881.56	1.41
80000	60882.99	58427.91	4.20	30007.99	29602.21	1.37
90000	68484.38	65703.48	4.23	33770.44	33288.51	1.45
100000	76085.32	73003.02	4.22	37533.47	37017.42	1.39
200000	152079.28	145976.81	4.18	75183.43	74032.34	1.55
300000	228058.20	218977.59	4.15	112852.56	111009.11	1.66
400000	304029.34	292018.38	4.11	150531.61	147943.78	1.75
500000	379995.51	365019.72	4.10	188217.00	185009.88	1.73
600000	455958.15	438047.97	4.09	225906.89	221991.13	1.76
700000	531918.11	510949.31	4.10	263600.20	259084.98	1.74
800000	607875.95	584018.19	4.09	301296.20	295986.69	1.79
900000	683832.06	657000.94	4.08	338994.42	332964.00	1.81
1000000	759786.72	730057.38	4.07	376694.49	370028.31	1.80
2000000	1519284.52	1459981.63	4.06	753757.37	739921.81	1.87
3000000	2278734.78	2190033.00	4.05	1130880.86	1110010.63	1.88
4000000	3038160.44	2920176.00	4.04	1508035.72	1479929.25	1.90
5000000	3797570.37	3650194.00	4.04	1885210.63	1849961.25	1.91
6000000	4556969.13	4379974.51	4.04	2262399.79	2220307.25	1.90
7000000	5316359.43	5110225.09	4.03	2639599.74	2589959.32	1.92
8000000	6075743.03	5840288.12	4.03	3016808.23	2959810.18	1.93
9000000	6835121.15	6569983.48	4.04	3394023.70	3329988.37	1.92
10000000	7594494.68	7300155.49	4.03	3771245.02	3700110.91	1.92

Table 10.9: Operation Skew Prediction of Parallel Hash Join with Single Data Skew on 4 Processors

another is fragmented on foreign key. Using 4 CPUs simulating 4 processors, the results are displayed in Table 10.9. Again, we approximate the Harmonic Number in one relation in the skew model and thus expect the relative error is slightly larger than that of operation skew without data skew. Increasing the number of processors from 4 to 16, the load comparisons are shown in Figures 10.17 and 10.18. Not surprisingly, all values provide a similar trend as in load skew prediction with Zipf data skew.

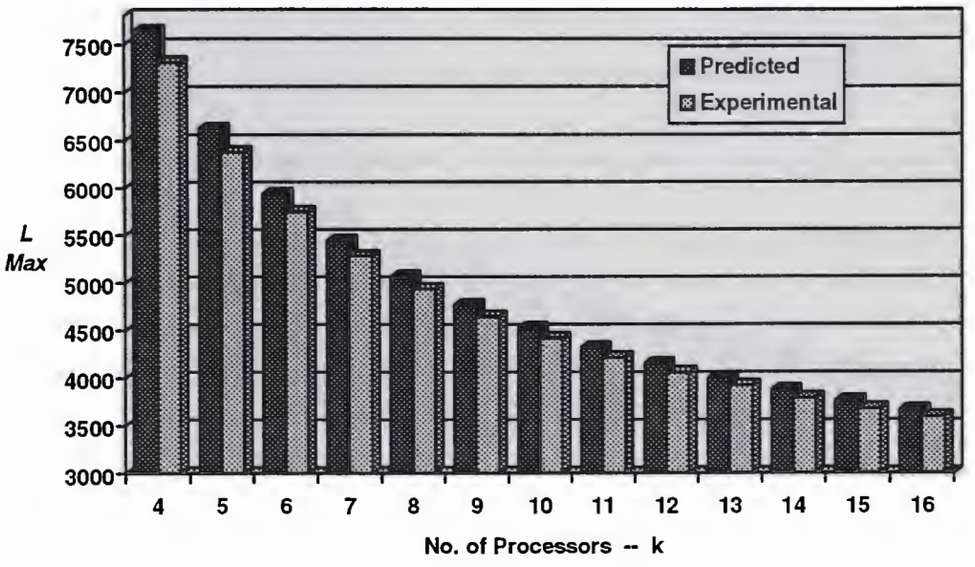


Figure 10.17: Maximum Operation Skew Prediction with Single Data Skew

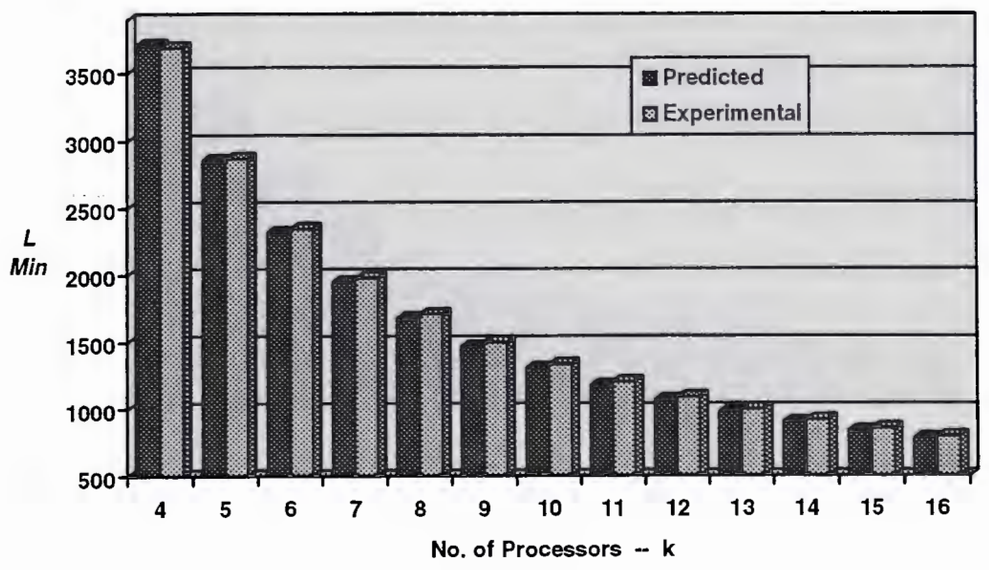


Figure 10.18: Minimum Operation Skew Prediction with Single Data Skew

Finally, we implement data skew in both input synthetic relations and the appearances of distinct values in both relations follow a pure Zipf distribution. With the assumption on strong correlation between the two input relations, two synthetic relations are loaded and they are partitioned on a common non-key attribute. With 4 processors, the experiment results and the relative error are listed in Table 10.10.

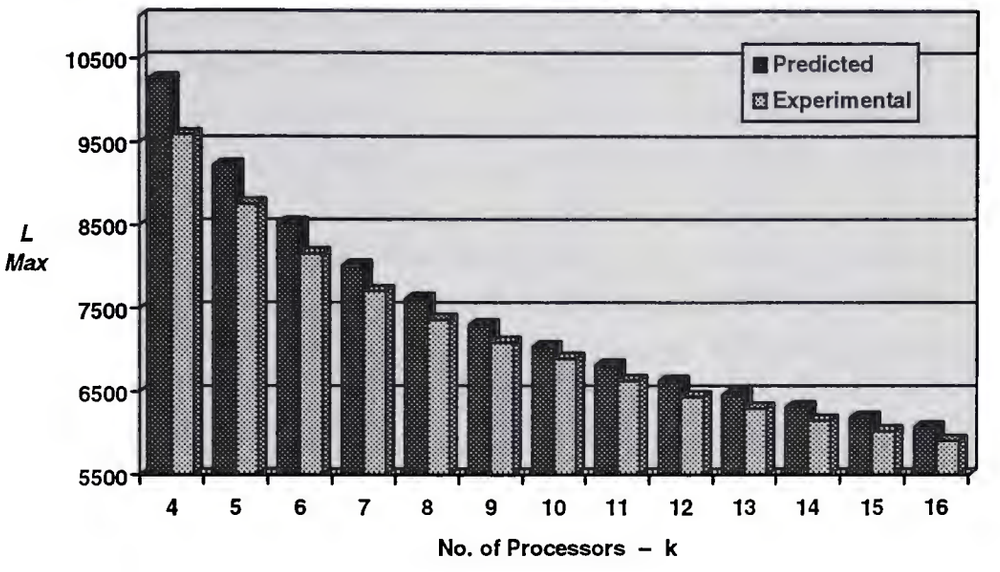


Figure 10.19: Maximum Operation Skew Prediction with Double Data Skew

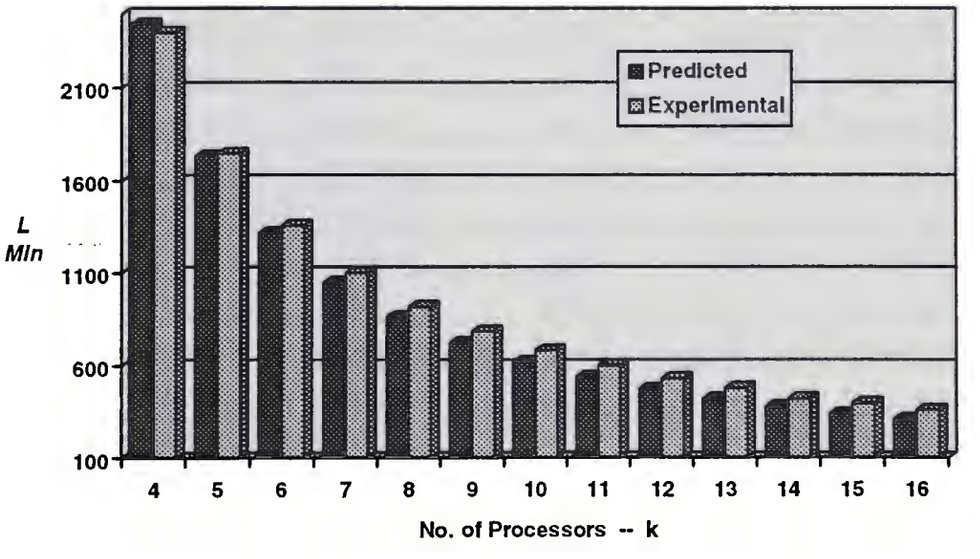


Figure 10.20: Minimum Operation Skew Prediction with Double Data Skew

4 processors	Maximum Load Prediction			Minimum Load Prediction		
R=S	Predicted	Experiment	RE-%	Predicted	Experiment	RE-%
1000	1041.89	959.10	8.63	222.76	238.54	6.62
2000	2070.13	1916.75	8.00	464.20	480.96	3.48
3000	3096.12	2877.13	7.61	708.72	721.13	1.72
4000	4120.95	3836.08	7.43	954.82	955.70	0.09
5000	5145.04	4801.96	7.14	1201.94	1202.05	0.01
6000	6168.60	5762.36	7.05	1449.78	1439.60	0.71
7000	7191.76	6720.25	7.02	1698.17	1682.10	0.96
8000	8214.61	7680.66	6.95	1947.00	1920.08	1.40
9000	9237.19	8635.70	6.97	2196.17	2159.72	1.69
10000	10259.56	9605.83	6.81	2445.65	2397.58	2.00
20000	20475.95	19199.45	6.65	4950.35	4798.93	3.16
30000	30685.25	28795.32	6.56	7464.77	7201.60	3.65
40000	40890.86	38377.59	6.55	9984.23	9595.78	4.05
50000	51094.13	47998.60	6.45	12506.89	11996.23	4.26
60000	61295.72	57629.56	6.36	15031.84	14382.29	4.52
70000	71496.06	67207.41	6.38	17558.52	16802.92	4.50
80000	81695.39	76816.86	6.35	20086.57	19207.74	4.58
90000	91893.90	86381.13	6.38	22615.74	21598.04	4.71
100000	102091.73	95967.22	6.38	25145.85	23995.00	4.80
200000	204046.95	192017.80	6.26	50478.46	47995.69	5.17
300000	305979.72	288027.53	6.23	75841.80	72004.66	5.33
400000	407900.86	384032.72	6.22	101221.04	95985.38	5.45
500000	509814.57	480089.28	6.19	126610.45	120009.42	5.50
600000	611723.00	576070.81	6.19	152007.08	143982.45	5.57
700000	713627.43	672026.38	6.19	177409.18	168017.06	5.59
800000	815528.70	767917.69	6.20	202815.62	192016.95	5.62
900000	917427.38	863916.94	6.19	228225.59	216007.00	5.66
1000000	1019323.90	960015.94	6.18	253638.53	240059.93	5.66
2000000	2038216.12	1920265.50	6.14	507867.66	479847.31	5.84
3000000	3057037.33	2879818.25	6.15	762193.95	720084.00	5.85
4000000	4075821.77	3839964.75	6.14	1016570.55	959973.69	5.90
5000000	5094582.73	4800040.50	6.14	1270979.29	1199891.63	5.92
6000000	6113326.99	5760330.00	6.13	1525410.86	1439908.50	5.94
7000000	7132058.61	6720379.50	6.13	1779859.73	1679788.00	5.96
8000000	8150780.21	7680397.50	6.12	2034322.30	1919822.13	5.96
9000000	9169493.64	8640177.00	6.13	2288796.07	2159937.25	5.97
10000000	10188200.21	9600266.01	6.12	2543279.20	2399898.12	5.97

Table 10.10: Operation Skew Prediction of Parallel Hash Join with Double Data Skew on 4 Processors

Varying the number of processors from 4 to 16, the comparisons of both minimum and maximum between the skew model and the implementation results are shown in Figures 10.19 and 10.20.

10.6 Conclusion

We have implemented the relational database processing and the skew model on a parallel client server system with a shared memory architecture. The server is a DEC Alpha 2100 system and the database is stored in RAID and generated using synthetic methods. From the experiments, we conclude that overheads due to parallel processing such as data transmission and partitioning play an important role in the overall processing time. IO is clearly the bottleneck in parallel processing on DEC Alpha Server and constantly consumes more than 60% of the total elapsed time. The default setting on DEC Alpha Server makes use of parallel processing but the solution provided is not optimal. With the proper partitioning of data, the processing time is linearly speeded up relative to the number of processors as in the *FAP* method. However, when the number of processors is scaled up, the parallel processing overheads may dominate the total time in a shared memory system.

During the skew model implementation, we program and address it as a shared memory system but we believe that implementing the skew model on shared nothing systems will give the same result because of the architecture independence of the skew model. Data skew is modelled using Discrete Uniform Distribution, Zipf Distribution, and Normal Distribution. The experimental results collected from the implementation further confirm the validity of the analytical model in predicting load and operation skewness.

CHAPTER 11

CONCLUSIONS

11.1 Summary of the Thesis

With new data demands and advanced requirements for flexible data correlation, systems performance is a critical issue, especially with the advent of *Decision Support Systems* (*DSS*) and *On-Line Analytic Processing Systems* (*OLAP*) applications. Since the processing power of conventional computer systems can only handle a small fraction of current applications, parallel processing with multiple processors is widely acknowledged as the only viable solution for very large relational databases.

However, parallelism does bring with it considerable problems. One of these problem is that of skewness since it is highly likely that processors are assigned different workloads so that the earlier finished processors have to wait until the most heavily loaded processors to complete. The causes of such load imbalance include the data partitioning function and the phenomenon that some attributes appear more often than others in input relations. Another problem is effective query optimisation over multiple processors in the presence of the skewness. This thesis has focused on skew taxonomy, modelling, and prediction, and skew effects on query processing.

A skew taxonomy has been developed followed by a skew analysis. We believe that this is the first time the skew problem has been systematically studied. With the taxonomy, we developed a skew foundation model by identifying three types of skew, namely, data skew, load skew, and operation skew. Load skew is further classified into I/O load skew, operation load skew, and result load skew. These different types of skew have been quantitatively analysed and evaluated. In the light of skew taxonomy, we discover that skew effect may be significant even when data are partitioned evenly over the processors.

To strengthen the skew framework, a skew model has been developed based on extreme value properties. The skew model consists of two parts, foundation model and extension model. In the foundation model, not only the mean maximum and minimum load but also their standard deviations and distribution functions are obtained. All extreme value predictions are evaluated in the simulation study. Furthermore, we implemented the skew model on a parallel system, DEC Alpha 2100. The test database is generated using synthetic methods and the results show a close agreement between experimental and predicted values.

Although skew handling has received considerable attention in recent years, most of proposed methods are devoted to solving or avoiding data skew for single binary join operation. We have extended the current work to a higher level which deals with multiple binary operations in one query. We also find that allocating a large number of processors does not always improve the performance significantly in the presence of data skew. Based on this, we approach the skew problem by identifying the skewed operations and group them with other operations based on a phase based processor allocation approach so that the performance degradation caused by skewness is kept to a minimal level. The heuristic employed in our new algorithm is based on neighbourhood search.

We have identified and evaluated three critical issues in the processing of aggregate functions, namely, the selection of partitioning attribute, the sequence of aggregation and join operation, and the skewness. Three parallel methods of processing aggregate functions are introduced and they differ in their selection of partitioning attributes. Cost models for these methods incorporating the effect of data skew have been developed.

We have also developed three new intra-query processor allocation algorithms. The first uses a dynamic non phase based approach, a second one uses a phase based approach and the third is a hybrid method based on the system size. The optimal degree of parallelism is introduced for each operation together with the operation optimal time. Using network analysis, the optimal query time and processors bound are derived. A time equalisation algorithm is also developed to achieve local optimisation in the phase based approach. Furthermore, we have developed a new inter-query processor allocation algorithm which focuses on multiple dependent queries and makes use of the activity analysis, resource scheduling, resource leveling, and decompression techniques.

To sum up, the key contributions of this thesis include the following.

- Provided for the first time a theoretical study of skewness in parallel database systems in depth, where the extent of skewness are quantitatively analysed and performance bounds were obtained.
- Provided a skew prediction model based on range partitioning.
- Provided a systematic skew foundation analytical model of hash partitioning where the relationship between data skew and load skew as well as load characteristics are expressed in closed form.
- Developed parallel methods for the processing aggregate functions for general database queries.
- Developed a new processor allocation algorithm which minimises the skew effect through inter-operator parallelism and intra-operator parallelism.
- Provided a complete parallel query execution model incorporating the parallel processing overheads and load imbalance in massive parallel systems.
- Developed new intra-query processor allocation algorithms.
- Developed a new inter-query processor allocation algorithm for enhancing the performance of multiple dependent queries.

11.2 Future Work and Limitations

Skew modelling can be extended in several ways. Although operation skew for join operation is presented in the thesis, the operation mainly covers the method of sort merge join, hash join, and nested loops join. As such, we can extend the skew model in a

practical environment taking into account more detailed system characteristics, e.g. hybrid hash join in shared nothing systems incorporating the effect of buffer size. Moreover, based on the skew model, the propagation of skew in multiway join can be quantitatively analysed and the error rate on skew propagation can also be provided in multiple operations.

It is also true that extremely severe skew is rarely found because the skew in one relation may not reinforce the skew in another relation, and in fact, they might even cancel out each another in a binary operation.

From the user points of view, an interface of the skew model is needed for it to be widely useable, and it may be built as a kind of parallel performance tools. It may be implemented using component architectures, such as Javabeans or ActiveX Control, so that the model can be used, manipulated, or customised visually.

Our results primarily focus on relational databases but they may be extended to object-oriented databases [Kim90, Leun95, Liu96c]. Skewness occurs whenever multiple processors are involved, and thus the research techniques in parallel relational databases can be directly applied or slightly modified in parallel object-oriented databases, both of which deal with data partitioning functions and are concerned with the distribution of tuples or objects. The skew taxonomy and model may be modified to reflect the skew behaviour in parallel object-oriented databases. In Object-Oriented systems, we have objects and they are linked through pointers. Therefore, they seldom have multiple explicit joins and several objects in one path expression. The issues here are skew propagation and constrained processor allocation.

In addition, we can develop novel processor allocation methods through introducing the concept of processor allocation efficiency because adding one additional processor to different operations may have different impact both locally and globally. This is particularly meaningful when the number of processors is relatively small and there is no global optimal solution. Moreover, the techniques from classical optimisation theory such as *dynamic programming* and *network analysis*, and heuristics such as *simulated annealing* and *tabu search* may also be incorporated into new processor allocation methods to achieve better performance.

The cost model used in query processing plays a crucial role in the overall optimisation process. Therefore, it should always be adjusted based on the real environment parameters before actual query processing implementation.

BIBLIOGRAPHY

[Alma94] Almasi G. S. and A. Gottlieb, *Highly Parallel Computing*, The Benjamin/Cummings Publishing Company, Inc., 1994

[Amda67] Amdahl G., "Validity of the single processor approach to achieving large scale computing capabilities", *Proceedings of the AFIPS Computing Conference*, Atlantic City, N. J., USA, Vol. 30, pp 483-485, April 1967

[Aper83] Apers P. M. G., A. R. Hevner, and S. B. Yao, "Optimization algorithms for distributed queries", *IEEE Transactions on Software Engineering*, Vol. 9, No. 1, pp 57-68, 1983

[Azar94] Azar Y., A. Z. Broder, A. R. Karlin, and E. Upfal, "Balanced allocations", *Proceedings of the 26th Annual ACM Symposium on the Theory of Computing*, Canada, May 1994

[Barl94] Barlos F. and O. Frieder, "A join ordering approach for multicomputer relational databases with highly skewed data", *Proceedings of the 27th Annual Hawaii International Conference on System Sciences*, IEEE CS Press, Los Alamitos, CA, pp 253-262, 1994

[Bell89] Bell D. A., D. H. O. Ling, and S. McClean S., "Pragmatic estimation of join sizes and attribute correlations", *Proceedings of 5th IEEE Data Engineering Conference*, Los Angeles, pp 76-84, 1989

[Bell93] Bell D. A., "From data properties to evidence", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 5, No. 6, pp 965-969, 1993

[Bern81] Bernstein P. A., N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie, "Query processing in a system for distributed databases (SDD-1)", *ACM Transactions on Database Systems*, Vol. 6, No. 4, pp 602-625, 1981

[Bhat93] Bhatt S. N., A. Ranade, and A. L. Rosenberg, "Scattering and gathering messages in networks of processors", *IEEE Transactions on Computers*, Vol. 42, No. 8, August 1993

[Bic89] Bic L. and R. L. Hartmann, "AGM: a dataflow database machine", *ACM Transactions on Database Systems*, Vol. 14, No. 1, pp 114-146, March 1989

[Bitt83] Bitton D., H. Boral, D. J. DeWitt, and W. K. Wilkinson, "Parallel algorithms for the execution of relational database operations", *ACM Transactions on Database Systems*, pp 324-353, September 1983

[Bitt86] Bitton D. and C. Turbyfill, "Performance evaluation of main memory database systems", Technical Report TR 86-731, Department of Computer Science, Cornell University, January 1986

[Bora88] Boral H., "Parallelism in Bubba", *Proceedings of the 1st International Symposium Databases in Parallel and Distributed Systems*, Los Alamitos, CA, pp 68-71, 1988

[Bora90] Boral H., W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez, "Prototyping Bubba, a highly parallel database system", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No, 1, pp 4-24, 1990

[Borj79] Borjesson P. O. and C. E. W. Sundberg, "Simple approximations of the error function Q(x) for communications applications", *IEEE Transactions on Communications*, Vol. COM-27, No.3, pp 639-643, March 1979

[Brin96] Brinkhoff T., H. Kriegel, and B. Seeger, "Parallel processing of spartial joins using R-trees", *Proceedings of the 12th International Conference on Data Engineering*, New Orleans, pp 258-265, 1996

[Bult87] Bultzingsloewen G., "Translating and optimizing SQL queries having aggregate", *Proceedings of the 13th International Conference on Very Large Data Bases*, Brighton, U.K., 1987

[Catt94] Cattel R. G. G., *Object data management: object-oriented and extended relational database systems*, Addison-Wesley Publishing Company, 1994

[Chek95] Chekuri C., W. Hasan, and R. Motwani, "Scheduling problems in parallel query optimization", *Proceedings of the ACM Symposium on Principles of Database Systems*, San Jose, U.S.A. 1995

[Chen92] Chen M. S., P. S. Yu, and K. L. Wu, "Scheduling and processor allocation for parallel execution of multi-join queries", *Proceedings of the 8th International Conference on Data Engineering*, Los Alamitos, CA, U.S.A., pp 58-67, February 1992

[Chen93] Chen M. S. and P. S. Yu, "Combining join and semi-join operations for distributed query processing", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 5, No. 3, pp 534-542, 1993

[Chri83] Christodoulakis S., "Estimating record selectivities", *Information Systems*, Vol.8, No.2, pp 105-115, 1983

[Codd70] Codd E. F., "A relational model of data for large shared data banks", *Communications of the ACM*, Vol. 13, No. 6, pp 377-387, 1970

[Codd90] Codd E. F., *The Relational Model for Database Management*, Version 2, Addison-Wesley Publishing Company, 1990

[Cope88] Copeland G., W. Alexander, E. Boughten, and T. Keller, "Data placement in Bubba", *Proceedings SIGMOD International Conference on Management of Data*, Chicago Illinois, pp 99-108, June 1988

[Cox62] Cox D. R., *Renewal Theory*, Methuen, 1962

[Date95] Date C. J., *An Introduction to Database Systems*, Volume 1, Fifth Edition, Addison-Wesley Publishing Company, 1995

[Daya87] Dayal U., "Of nests and trees: a unified approach to processing queries that contain nested subqueries, aggregates, and quantitiers", *Proceedings of the 13th International Conference on Very Large Data Bases*, Brighton, U.K., 1987

[Dewa94] Dewan H., *Runtime Reorganization of Parallel and Distributed Expert Database Systems*, Ph.D. Thesis, Department of Computer Science, Columbia University, New York, 1994

[DeWi90] DeWitt D. J., S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen, "The Gamma database machine project", *IEEE Transactions On Knowledge and Data Engineering*, Vol. 2, No. 1, March 1990

[DeWi92a] DeWitt D.J. and J. Gray, "Parallel database systems", *Communications of the ACM*, Vol. 35, No. 6, pp 85-98, 1992

[DeWi92b] DeWitt D.J., J. F. Naughton, D. A. Schneider, and S. Seshadri., "Practical skew handling in parallel joins", *Proceedings of the 18th International Conference on Very Large Data Bases*, Vancouver, British Columbia, Canada 1992

[Digi94] Digital Equipment Corporation, *DEC OSF/1: Technical Overview*, August 1994

[Digi95a] Digital Equipment Corporation, "UNIX software strategy", *A Digital*

Equipment Corporation White Paper, September 1995

[Digi95b] Digital Equipment Corporation, *Digital 2100 Server: Owner's Guide*, 1995

[Dris95] Driscoll M. A. and W. R. Daasch, "Accurate predictions of parallel program execution time", *Journal of Parallel and Distributed Computing*, Vol. 25, pp 16-30, 1995

[Du89] Du J. Z. and J. Y. T. Leung, "Complexity of scheduling parallel task systems", *SIAM Journal of Discrete Mathematics*, Vol. 2, No. 4, pp 473-487, 1988

[Elma92] Elmagarmid A. K. (Ed.), *Transaction Models for Advanced Database Applications*, Morgan-Kaufmann, San Mateo, CA, 1992

[Elma94] Elmasri R. and S. B. Navathe, *Fundamentals of Database Systems*, 2nd Ed., Benjamin/Cummings, 1994

[Falo96] Faloutsos C., Y. Matias, and A. Silberschatz, "Modeling skewed distributions using multifractals and the '80-20 law' ", *Proceedings of the 22nd VLDB Conference*, Mumbai, India, pp 307-317, 1996

[Frie90] Frieder O., "Parallelism: using the right tool for the right job", *Proceedings of the International Conference on Databases, Parallel Architectures, and their Applications*, Miami Beach, Florida, pp 537-538, March 1990

[Frie94] Frieder O. and C. K. Baru, "Site and query scheduling policies in multicomputer database systems", *IEEE Transactions on Knowledge and Data Engineering*, Vol.6, No.4, pp 609-619, August 1994

[Gala87] Galambos, J. *The Asymptotic Theory of Extreme Order Statistics*, 2nd Ed., Kreiger Publishing Co., 1987

[Gang92] Ganguly S., W. Hasan, and R. Krishnamurthy, "Query optimization for parallel execution", *Proceedings of ACM SIGMOD International Conference on Management of Data*, June 1992

[Ghan90] Ghandeharizadeh S. and D. J. DeWitt, "Hybrid-range partitioning strategy: a new declustering strategy for multiprocessor database machines", *Proceedings of the 16th VLDB Conference*, Brisbane, Australia, pp 481-492, 1990

[Ghan94] Ghandeharizadeh S. and D. J. DeWitt, "MAGIC: a multiattribute declustering mechanism for multiprocessor database machines", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 5, 1994

[Good49] Goodman B. L. A., "On the estimating of the number of classes in a population", *Annals of Mathematical Statistics*, Volume XX, pp 572-579, 1949

[Grae90] Graefe G., "Encapsulation of parallelism in the Volcano query processing system", *Proceedings of International Conference on ACM SIGMOD*, 1990

[Grae93] Graefe G., "Query evaluation techniques for large databases", *ACM Computing Surveys*, Vol. 25, No. 2, June 1993

[Grah69] Graham R., "Bounds on multiprocessing timing anomalies", *SIAM Journal of Computing*, Vol. 1, No. 7, pp 416-429, 1969

[Gray80] Graybeal W. and U. W. Pooch, *Simulation: Principles and Methods*, Winthrop Publishers, Inc., 1980

[Gray91] Gray J., *The benchmark handbook for database and transaction processing systems*, Morgan Kaufmann Publishers, Inc., 1991

[Gray94] Gray J., Sundaresan P., Englert S., Baclawski K., and P. Weinberger, "Quickly generating billion-record synthetic databases", *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis, Minnesota, U.S.A., 1994

[Gust88] Gustafson J. L., "Reevaluating amdahl's law", *Communications of the ACM*, Vol.31, No.5, pp 532-533, May 1988

[Hagm86] Hagmann R. B. and D. Ferrari, "Performance analysis of several bach-end database architectures", *ACM Transactions on Database Systems*, Vol. 11, No. 1, pp 1-26, March 1986

[Hasa94] Hasan W. and R. Motwani, "Optimization algorithms for exploiting the parallelism-communication tradeoff in pipelined parallelism", *Proceedings of the 20th VLDB Conference*, Santiago, Chile, pp 36-47, 1994

[Hasa95] Hasan W., D. Florescu, and P. Valduriez, "Open issues in parallel query optimisation", *SIGMOD Record*, 1995

[Helm90] Helmbold D. P. and C. E. McDowell, "Modeling speedup (n) greater than n", *IEEE Transactions on Parallel and Distributed Systems*, Vol.1, No. 2, pp 250-256, April 1990

[Hong92] Hong W., *Parallel Query Processing using Shared Memory Multiprocessors and Disk Arrays*, Ph.D. Thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, August 1992

[Hong93] Hong W. and M. Stonebraker, "Optimization of parallel query execution plans in XPRS", *Distributed and Parallel Databases*, Vol. 1, No. 1, Kluwer Academic Publishers, pp 218-225, 1993

[Hua91] Hua K. A. and C. Lee, "Handling data skew in multiprocessor database computers using partition tuning", *Proceedings of the 17th International Conference on Very Large Data Bases*, Barcelona, pp 525-535, September 1991

[Hua95] Hua K. A., Y. L. Lo and H. C. Young, "Optimizer-assisted load balancing techniques for multicomputer database management systems", *Journal of Parallel and Distributed Computing*, Vol. 25, Academic Press, pp 42-57, 1995

[Hwan93] Hwang K., *Advanced Computer Architecture*, McGraw-Hill, New York, 1993

[Inmo96] Inmon W. H., *Building The Data Warehouse*, 2nd Edition, Wiley, 1996

[Ioan91] Ioannidis Y. E. and S. Christodoulakis, "On the propagation of errors in the size of join results", *Proceedings of the 1991 International Conference on ACM SIGMOD*, pp 268-277, 1991

[Ioan95] Ioannidis Y. E. and V. Poosala, "Balancing histogram optimality and practicality for query result size estimation", *Proceedings of the 1995 International Conference on ACM SIGMOD*, San Jose, CA, pp 233-244, May 1995

[Jark84] Jarke M. and J. Koch, "Query optimization in database sytems", *ACM Computing Surveys*, Vol 16, No 2, pp 111-152, 1984

[Jian95] Jiang Y. and C. H. C. Leung, "Site allocation for parallel query execution in locally distributed databases", *Proceedings of the 7th IASTED International Conference on Parallel and Distributed Computing and Systems*, Washington D. C., pp 37-41, October 1995

[Jian96] Jiang Y., C. H. C. Leung, and K. H. Liu, "A comparative evaluation of phase-based and non-phase-based processor allocation for parallel query execution", *Proceedings of the 3rd Australian Conference on Parallel and Real-Time Systems*, Brisbane, Australia, pp 43-51, September 1996

[John60] Johnson, N. L. and D. H. Young, "Some applications of two approximations to the multinomial distribution," *Biometrika*, Vol. 47, 1960

[John77] Johnson N. L. and S. Kotz, *Urn Models and Their Application,* John Wiley & Sons, 1977

[Kell91] Keller A. M. and S. Roy, "Adaptive parallel hash join in main-memory databases", *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems*, December 1991

[Kim90] Kim K. C., "Parallelism in Object-Oriented query processing", *Proceedings of the 6th International Conference on Data Engineering*, pp 209-217, 1990

[Kim82] Kim W., "On optimizing an SQL-like nested query", *ACM Transactions on Database Systems*, Vol.7, No.3, September 1982

[Kim89] Kim W., "A Model of Queries for Object-Oriented Databases", *Proceedings of the 15th International Conference on Very Large Data Bases*, Amsterdam, pp 423-432, 1989

[Kits90] Kitsuregawa M. and Y. Ogawa, "A new parallel hash join method with robustness for data skew in super database computer (SDC)", *Proceedings of the 16th International Conference on Very Large Data Bases*, pp 210-221, 1990

[Knut73] Knuth D.E., *The Art of Computer Programming*, Volume 3: Sorting and Searching, Addison-Wesley Publishing, pp 397-399, 1973

[Kris86] Krishnamurthy R., H. Boral, and C. Zaniolo, "Optimization of nonrecursive queries", *Proceedings of the 12th International Conference on Very Large Data Bases*, Kyoto Japan, pp 128-137, August 1986

[Kuma90] Kumar M., "Measuring parallelism in computation-intensive scientific/engineering applications", *IEEE Transactions on Computers*, Vol.37, No.9, pp 1088-1098, September 1988

[Kuma94] Kumar V., A. Grama, A. Gupta and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, The Benjamin/Cummings Publishing Company, Inc., 1994

[Laks90] Lakshmi M. S. and P. S. Yu, "Effectiveness of parallel joins", *IEEE Transactions On Knowledge and Data Engineering*, Vol. 2, No. 4, December 1990

[Lee93] Lee E. K., *Performance Modeling and Analysis of Disk Arrays*, Ph.D. Thesis, Department Computer Science, University of California, Berkeley, 1993

[Leun85] Leung C. H. C. and K. S. Wong, "File processing efficiency on the content addressable filestore," *Proceedings of the 11th International Conference on Very Large Data Bases,* Stockholm, pp 282-291, August 1985

[Leun86] Leung C. H. C., "Dynamic storage fragmentation and file deterioration", *IEEE Transactions on Software Engineering*, Vol. 12, No. 3, pp 436-441, 1986

[Leun88] Leung C. H. C., *Quantitative Analysis of Computer Systems*, John Wiley & Sons, 1988

[Leun93] Leung C. H. C. and H. T. Ghogomu, "A high-performance parallel database architecture", *Proceedings of the 7th ACM International Conference on Supercomputing,* Tokyo, pp 377-386, July 1993

[Leun94a] Leung C. H. C., "Parallel paradigms for query evaluation and processing", *Proceedings of the Australasian Workshop on Parallel and Real-Time Systems*, Melbourne, Australia, pp 1-10, July 1994

[Leun94b] Leung C. H. C. and K. H. Liu, "Limits to database speedup for data

partitioned in unary relational operations", Technical Report, Department of Computer and Mathematical Sciences, Victoria University of Technology, December 1994

[Leun95] Leung C. H. C. and K. H. Liu, "Skewness analysis of parallel join execution", Technical Report, Department of Computer and Mathematical Sciences, Victoria University of Technology, December 1995

[Leun96] Leung C. H. C. and K. H. Liu, "Load skew prediction of parallel relational database operations under hash partitioned strategy", Technical Report, Department of Computer and Mathematical Sciences, Victoria University of Technology, April 1996

[Lewi92] Lewis T. G. and H. El-rewini, *Introduction to Parallel Computing*, Prentice-Hall International, Inc., 1992

[Liu82] Liu A. C. and S. K. Chang, "Site selection in distributed query processing", *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pp 7-12, 1982

[Liu94] Liu K. H. and C. H. C. Leung, "TDB Simulation model user's manual", *Proceedings of the Terabyte Database Workshop'94*, Melbourne, Victoria, Australia, December 1994

[Liu95] Liu K. H., C. H. C. Leung, and Y. Jiang, "Analysis and taxonomy of skew in parallel databases", *Proceedings of International High Performance Computing Symposium (HPCS'95)*, Montreal, Canada, pp 304-315, July 1995

[Liu96a] Liu K. H., Y. Jiang, and C. H. C. Leung, "Query execution in the presence of data skew in parallel databases", *Australian Computer Science Communications*, Vol. 18, No. 2, pp157-166, 1996

[Liu96b] Liu K. H., "Load balancing algorithms for hash partitioned unary relational operations", *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*, Sunnyvale, California, USA, pp 671-675, August 1996

[Liu96c] Liu K. H. and D. Taniar, "Efficient processor allocation in parallel Object-Oriented databases", *Proceedings of the 3rd Australian Conference on Parallel and Real-Time Systems (PARTS'96)*, Brisbane, Australia, pp 164-172, September 1996

[Lu90] Lu H. J. and K. Tan, "Buffer and load balancing in locally distributed database systems", *Proceedings of the 6th International Conference on Data Engineering*, pp 545-552, 1990

[Lu91] Lu H. J., M. C. Shan and K. L. Tan, "Optimization of multi-way join queries for parallel execution", *Proceedings of the 17th VLDB Conference*, Barcelona, pp 549-560, September 1991

[Lu92] Lu H. J. and Tan K. L., "Dynamic and load-balanced task-oriented database query processing in parallel systems", *Advances in Database Technology -- EDBT'92, 3rd International Conference on Extending Database Technology*, Vienna, Austria, March, 1992

[Lync88] Lynch C. A., "Selectivity estimation and query optimization in large databases with highly skewed distributions of column values", *Proceedings of the fourteenth International Conference on Very Large Data Bases*, pp 240-251, 1988

[Mann88] Mannino M. V., P. Chu, and T. Sager, "Statistical profile estimation in database systems", *ACM Computing Surveys*, Vol. 20, No. 3, pp 191-221, 1988

[McCa94] McCann C. and J. Zahorjan, "Processor allocation for message-passing parallel computers", *Proceedings of the ACM Sigmetrics Conference*, May 1994

[Mish92] Mishra P. and M. H. Eich, "Join processing in relational databases", *ACM Computing Surveys*, Vol. 24, No. 1, March 1992

[Moha94] Mohan C., H. Pirahesh, W. G. Tang, and Y. Wang, "Parallelism in relational database management systems", *IBM Systems Journal*, Vol.33, No.2, pp 349-371, 1994

[Mood74] Mood A.M., F. A. Graybill, and D. C. Boes, *Introduction to the Theory of Statistics*, third edition, McGraw-Hill, 1974

[Murp93] Murphy M. C. and D. Rotem, "Multiprocessor join scheduling", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 5, No.2, pp 322-338, 1993

[Nels95] Nelson R., *Probability, Stochastic Processes, and Queueing Theory: The Mathematics of Computer Performance Modeling*, Springer-Verlag, 1995

[Omie91] Omiecinski E., "Performance analysis of a load balancing relational hash-join algorithm for a main-memory databases", *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems*, pp 58-67, December 1991

[Ozsu91] Ozsu T. M. and P. Valduriez, *Principles of Distributed Database Systems*, Prentice Hall, 1991

[Page92] Page J., "A study of a parallel database machine and its performance: the NCR/Teradata DBC/1012", *Proceedings of the 10th British National Conference on Databases*, Aberdeen, UK, pp 115-137, July 1992

[Pang93] Pang H.H., M J. Carey., and M. Livny, "Partially preemptible hash joins", *Proceedings of the International Conference on ACM SIGMOD*, Washington D. C., USA, May 1993

[Pate94] Patel J. M., M. J. Carey, and M. K. Vernon, "Accurate modeling of the hybrid hash join algorithm", *Proceedings of ACM SIGMETRICS Conference*, Santa Clara, CA, U.S.A., May 1994

[Patt88] Patterson D. A., G. Gibson and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)", *Proceedings of the International Conference on ACM SIGMOD*, Chicago, May 1988

[Pete94] Peterson G. D. and R. D. Chamberlain, "Beyond execution time: expanding the use of performance models", *IEEE Parallel & Distributed Technology*, Summer 1994

[Pira90] Pirahesh H., C. Mohan, J. Cheng, T. S. Liu, and P. Selinger, "Parallelism in relational data base systems: architectural issues and design approaches", *Proceedings of the 2nd International Symposium on Databases in Parallel and Distributed Systems*, Ireland, pp 1-29, July 1990

[Qada85] Qadah G. Z. and K. B. Irani, "A database machine for very large relational databases", *IEEE Transactions on Computers*, Vol. C-34, No. 1, pp 1015-1025, November 1985

[Qada88] Qadah G. Z. and K. B. Irani, "The join algorithms on a shared-memory multiprocessor database machine", *IEEE Transactions on Software Engineering*, pp 1668-1683, November 1988

[Rich87] Richardson J. P., H. Lu, and K. Mikkilineni, "Design and evaluation of parallel pipelined join algorithms", *Proceedings of International Conference on ACM SIGMOD*, pp 399-409, 1987

[Saad89] Saad Y. and M. H. Schultz, "Data communication in parallel architectures", *Parallel Computing*, Vol. 11, No. 2, pp 131-150, August 1989

[Salz89] Salza S. and M. Terranova, "Evaluating the size of queries on relational databases with non uniform distribution and stochastic dependence", *Proceedings of the International Conference on ACM SIGMOD*, pp 510-516, 1989

[Schn89] Schneider D.A. and D. J. DeWitt, "A performance of four parallel join algorithms in a shared-nothing multiprocessor environment", *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, Portland, Oregon, Vol.18, No.2, pp 110-121, June 1989

[Schn90] Schneider D. A. and D. J. DeWitt, "Tradeoffs in processing complex join queries via hashing in multiprocessor database machines", *Proceedings of the 16th VLDB Conference*, Brisbane, Australia, pp 469-480, August 1990

[Seli79] Selinger P. G., N. M. Astrashan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system", *Proceedings of the International Conference on ACM SIGMOD*, Boston, Mass., pp 23-34, June 1979

[Sesh92] Seshadri S. and J. F. Naughton, "Sampling issues in parallel database systems", *Proceedings of the 3rd International Conference on Extending Database Technology -- EDBT'92*, Vienna, Austria, March 1992

[Seve90] Severance C., S. Pramanik, and P. Wolberg, "Distributed linear hashing and parallel projection in main memory databases", *Proceedings of the 16th VLDB Conference*, Brisbane, Australia, 1990

[Shas91] Shasha D. and T. L. Wang, "Optimizing equijoin queries in distributed databases where relations are hash partitioned", *ACM Transactions on Database Systems*, Vol. 16, No. 2, pp 279-308, 1991

[Shat93] Shatdal A. and J. F. Naughton, "Using shared virtual memory for parallel join processing", *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Washington D. C., U. S. A., May 1993

[Shat94] Shatdal A. and J. F. Naughton, "Processing aggregates in parallel database systems", *Computer Sciences Technical Report # 1233*, Computer Sciences Department, University of Wisconsin-Madison, June 1994

[Shek93] Shekita E. J., H. C. Young, and K. L. Tan, "Multi-join optimization for symmetric multiprocessors", *Proceedings of the 19th VLDB Conference*, Dublin, Ireland, pp 479 - 492, 1993

[Silb96] Silberschatz A., H. F. Korth, and S. Sudarshan, *Database System Concepts*, 3rd Edition, McGraw Hill, New York, 1996

[Ston83] Stonebraker M., J. Woodfill, J. Ranstorm, M. C. Murphy, M. Meyer, and E. Allman, "Performance enhancements to a relational database system", *ACM Transactions on Database Systems*, Vol. 8, No. 2, pp 167-185, 1983

[Ston88] Stonebraker M., R. Katz, D. Patterson, and J. Ousterhout, "The design of XPRS", *Proceedings of the 14th VLDB Conference*, Los Angeles, California, USA, pp 318-330, 1988

[Ston91] Stonebraker M. and G. Kemnitz, "The Postgres next generation database management system", *Communications of the ACM*, Vol. 34, No. 10, pp 78-92, 1991

[Su88] Su S. Y. W., *Database Computers: Principles, Architectures, and Techniques*, McGraw-Hill, New York, 1988

[Sun93] Sun W., Y. B. Ling, N. Rishe, and Y. Deng, "An instant and accurate size estimation method for joins and selection in a retrieval-intensive environment", *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Washington D. C., May 1993

[Swam88] Swami A. and A. Gupta, "Optimization of large join queries", *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, Chicago, Illinois, USA, pp 8-17, June 1988

[Thin93] Thinking Machines Corporation, "Connection machine CM-5 technical summary", 1993

[Ture94] Turek J., U. Schwiegelshohn, J. L. Wolf, and P. S. Yu, "Scheduling parallel tasks to minimize average response time", *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, Arlington, Philadelphia, pp 112-121, January 1994

[Tvrd96] Tvrdik P. and F. Plasil, *Parallel Computers: Theory and Practice*, IEEE Computer Society Press, 1996

[Ullm89] Ullman J. D., *Principles of Database and Knowledge-Base Systems*, Volume I and II, Computer Science Press, 1989

[Vald84] Valduriez P. and G. Gardarin, "Join and semijoin algorithms for a multiprocessor database machine", *ACM Transactions on Database Systems*, Vol. 9, No. 1, pp 133-161, 1984

[Vald93] Valduriez P., "Parallel database systems: open problems and new issues", *Journal of Distributed and Parallel Databases*, Kluwer Academic Publishers, pp 137-165, 1993

[Wang85] Wang Y. T. and R. J. T. Morris, "Load sharing in distributed systems", *IEEE Transactions on Computers*, Vol. 34, No. 3, pp 204-217, 1985

[Wang92] Wang Q. Z. and K. H. Cheng, "A heuristic of scheduling parallel tasks and its analysis", *SIAM Journal of Computing*, Vol. 21, No. 2, pp. 281-294, April 1992

[Walt91] Walton B., Dale A., and R. Jenevein, "A taxonomy and performance model of data skew effects in parallel joins", *Proceedings of the 17th International Conference on Very Large Data Bases*, Barcelona, pp 537-548, September 1991

[Wils92] Wilschut A. N., J. Flokstra, and P. M. Apers, "Parallelism in a main-memory DBMS: the performance of PRISMA/DB", *Proceedings of the 18th VLDB Conference*, Vancouver, British Colombia, Canada, pp 521-531, 1992

[Wils95] Wilschut A. N., J. Flokstra, and P. M. Apers, "Parallel evaluation of multi-join queries", *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, San Jose, CA, 1995

[Witk93] Witkowski A., F. Carino, and P. Kostamaa, "NCR 3700 -- the next generation industrial database computer", *Proceedings of the 19th VLDB Conference*, Dublin, Ireland, pp 230-243, 1993

[Wolf93a] Wolf J. L., D. M. Dias, and P. S. Yu, "A parallel sort-merge join algorithm for managing data skew", *IEEE Transactions On Parallel And Distributed Systems*, Vol. 4, No. 1, January 1993

[Wolf93b] Wolf J. L., P. S. Yu, J. Turek and D. M. Dias, "A parallel hash join algorithm for managing data skew", *IEEE Transactions On Parallel and Distributed Systems*, Vol.4, No. 12, December 1993

[Wolf95] Wolf J. L., J. Turek, M. S. Chen and P. S. Yu, "A hierarchical approach to parallel multiquery scheduling", *IEEE Transactions on Parallel and Distributed Systems*, Vol.6, No.6, pp 578-589, June 1995

[Yan94] Yan W. P. and P. Larson, "Performing group-by before join", *Proceedings of the International Conference on Data Engineering*, 1994

[Yu86] Yu C. T. and C. C. Chang, "Distributed query processing", *Computing Surveys*, Vol. 16, No.4, pp 399-433, 1984

[Yu89] Yu C. T., K. C. Guh, D. Brill, and A. Chen, "Partition strategy for distributed query processing in fast local networks", *IEEE Transactions on Software Engineering*, Vol. 15, No. 6, pp 780-792, 1989

[Yu92] Yu P. S., M. S. Chen, H. Heiss, and S. H. Lee, "On workload characterization of relational database environments", *IEEE Transactions on Software Engineering*, Vol. 18, No. 1, 1992

[Zipf49] Zipf G.K., *Human Behaviour and the Principle of Least Effort*, Addison-Wesley, Reading, Mass, 1949

APPENDIX A

To show

$$\left(2\Phi(3/k)-1+A\right)\geq\left(\Phi(6/k)-0.5+A\right),$$

the inequality is equivalent to

$$2\Phi(3/k)\geq\Phi(6/k)+0.5$$

$$\Leftrightarrow 2\Phi(3/k)+\Phi(0)-\Phi(0)\geq\Phi(6/k)+0.5$$

since $\Phi(0)=0.5$, we have

$$\Phi(3/k)-\Phi(0)+\Phi(3/k)\geq\Phi(6/k)-\Phi(3/k)+\Phi(3/k)$$

or

$$\frac{1}{\sqrt{2\pi}}\int_0^{3/k} e^{\frac{-t^2}{2}}dt\geq\frac{1}{\sqrt{2\pi}}\int_{3/k}^{6/k} e^{\frac{-t^2}{2}}dt\,.$$

Letting $y = t - 3/k$ on the right, we have

$$\frac{1}{\sqrt{2\pi}}\int_0^{3/k} e^{\frac{-t^2}{2}}dt\geq\frac{1}{\sqrt{2\pi}}\int_0^{3/k} e^{\frac{-(y+3/k)^2}{2}}dy\,.$$

which is true since the integrand on the right hand side is always bounded by that on the left.

APPENDIX B
TERABYTE DATABASE SIMULATION MODEL

B.1 Simulation Model Overview

The Terabyte Database Simulation Model (TDBSM) is designated for providing a user friendly environment which is able to facilitate users doing research in parallel database system in particular in query optimisation, data mining, and parallel architecture design. The model presents the TDBSM platform, i.e. an integrated development environment, which provides

- online help, and temporarily Operating System exit,
- built-in calendar, calculator, and ASCII table,
- multiple, movable, and resizable windows with fully mouse support, dialogue boxes, put down menus, and online status options,
- simulation setting to initiate or update database sites and query workload parameters,
- a list of unary relational algebraic operations such as selection, projection (with and without duplicate removal), and aggregation (minimum or maximum),
- a list of binary relational algebraic operations such as join (nested-loops join, sort-merge join, and hash-based join), and aggregate functions,
- online clock and heap view,

- library primitives including objects such as *TView*, *TMenu*, and *TDialog*, and applications such as skew generation in parallel database and path expression operation in Object-Oriented database.

TDBSM employs a hybrid architecture where nodes are loosely coupled and each node may have a number of processors with possible both hardware and software heterogeneity. The parameters are classified into systems, database sites, and workload categories. The default values of all parameters are interfacing users through dialog boxes and these values can be updated by users at any time. The TDBSM simulator is written in C++ and compiled using Borland C++ with the framework of Turbo Vision. The model is an event driven simulation which consists of four main types of events, keyboard events, mouse events, message events, and nothing event, respectively.

B.2 Source File Names Listing

Header Files

ascii.h
bgi.h
bgii.h
calculator.h
calendar.h
cost51.h
dialog.h
fileopen.h
hc_view.h
graphapp.h
mouse.h
gpuzzle.h
sk_simu.h
sksim1.h
tdbhelp.h
tdbsimu.h
tvcmds.h

Standard Include Header Files

borlandc\include
borlandc\tvision\include

Program Files

ascii.cpp	// ascii table
calculator.cpp	// calculator
calendar.cpp	// calendar

cost51.cpp	// path expression operation
fileopen.cpp	// open file
hc_view.cpp	// clock and heap
mouse.dlg	// mouse selection
gpuzzle.cpp	// game
sk_simu.cpp	// skewness
tdbsimu1.cpp	// main program one
tdbsimu2.cpp	// main program two
tdbsimu3.cpp	// main program three
tdbsimu4.cpp	// main program four
utls.cpp	// graphics utilities

Libraries

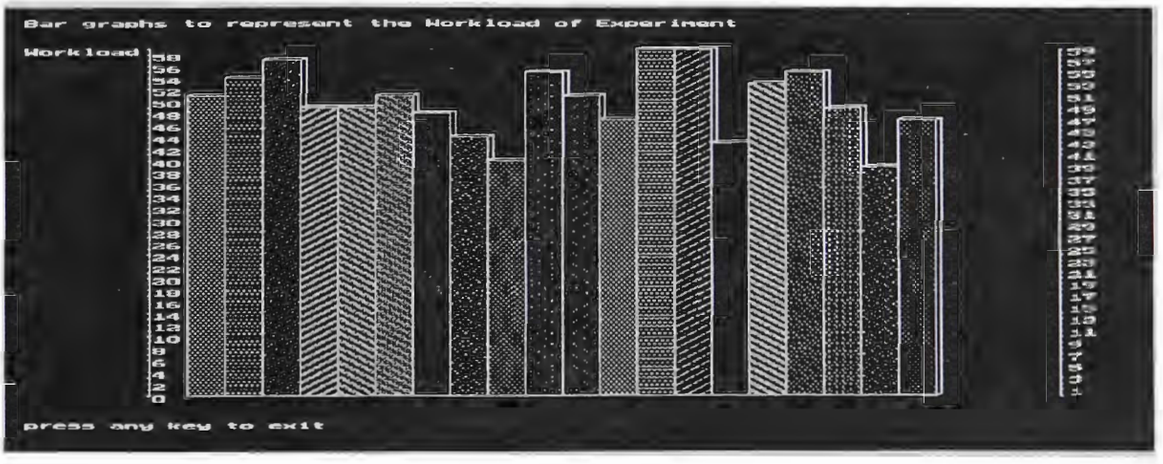
borlandc\lib	// C++ library
borlandc\tvision\lib	// Library for application framework Turbo Vision

B.3 An Example of Data Representation - Data Skew

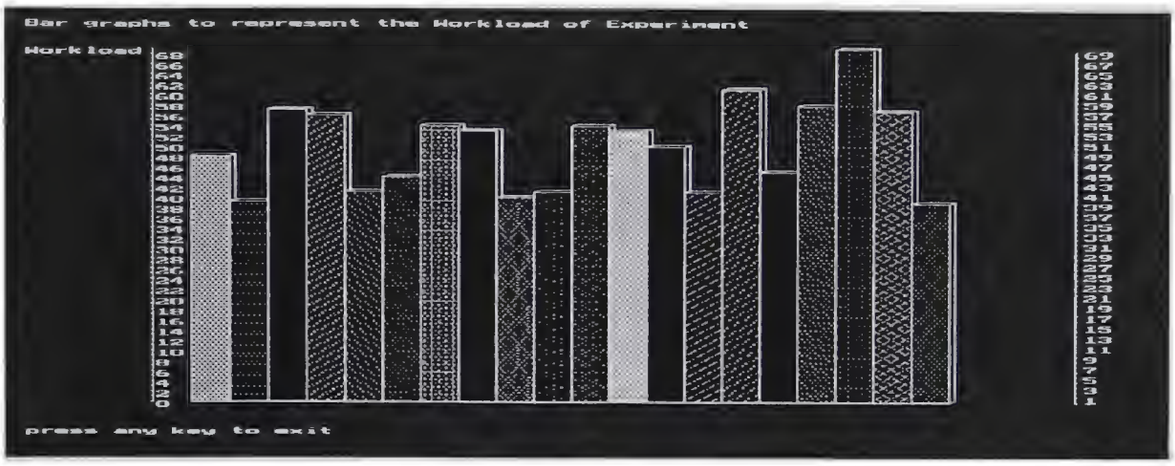
TDBSM has employed visualisation on the result representation. An example of data representation on data skew is provided in the following figures. In the figures,

- each 3D-bar represents one processor,
- the length of the 3D-bar indicates the number of tuples in that processor,
- two vertical lines with coordinates provide scale of workload, e.g. 60 means 60 tuples.

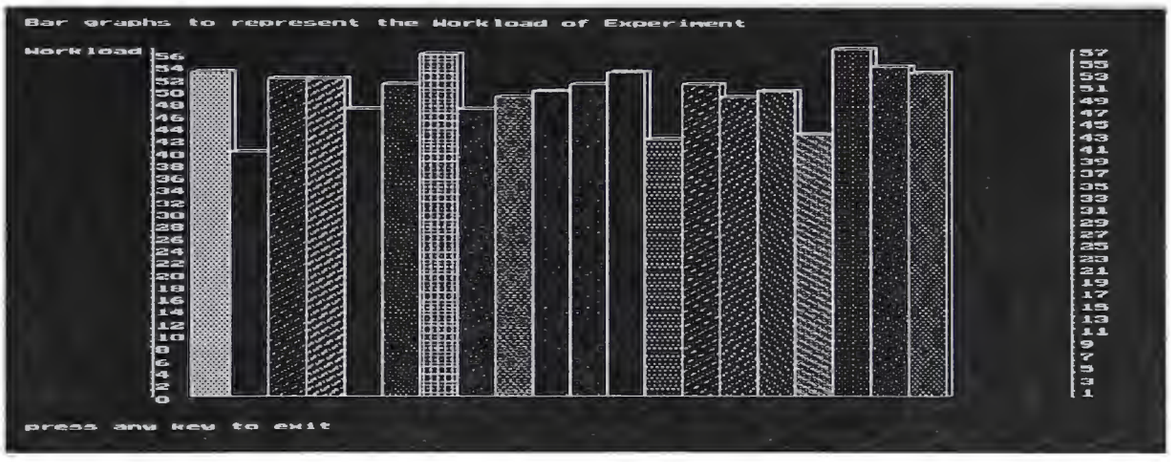
⇒ The first experiment with 20 processors and 1000 tuples



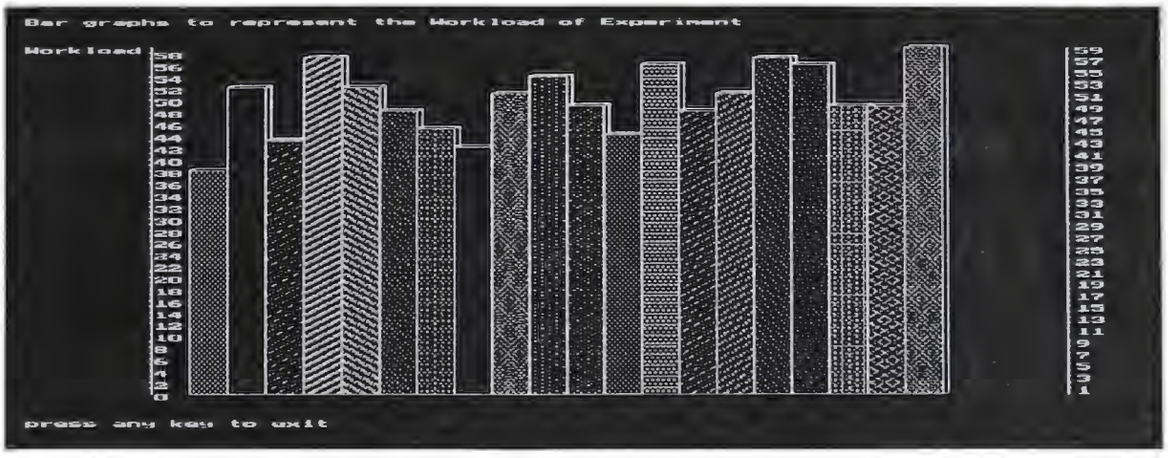
⇒ The second experiment with 20 processors and 1000 tuples



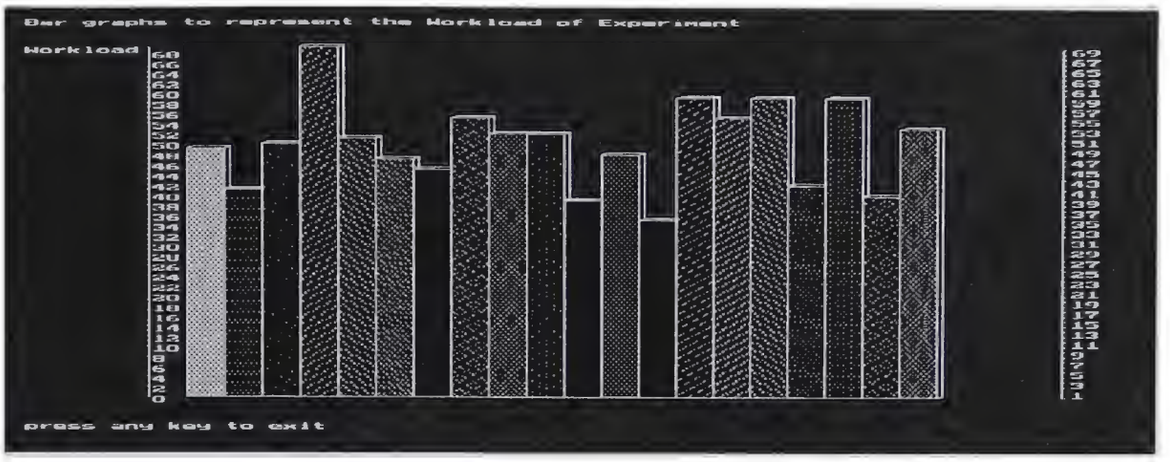
⇒ The third experiment with 20 processors and 1000 tuples



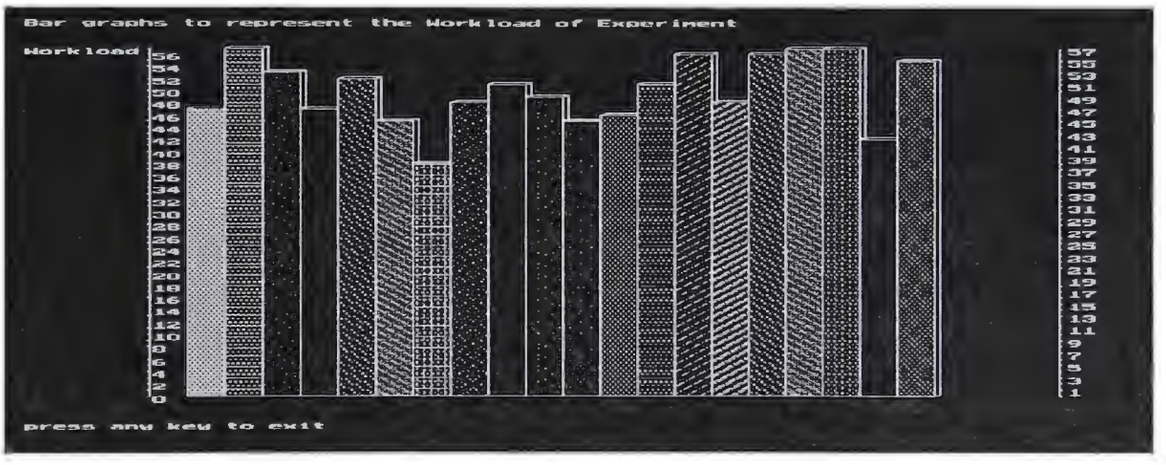
⇒ The fourth experiment with 20 processors and 1000 tuples



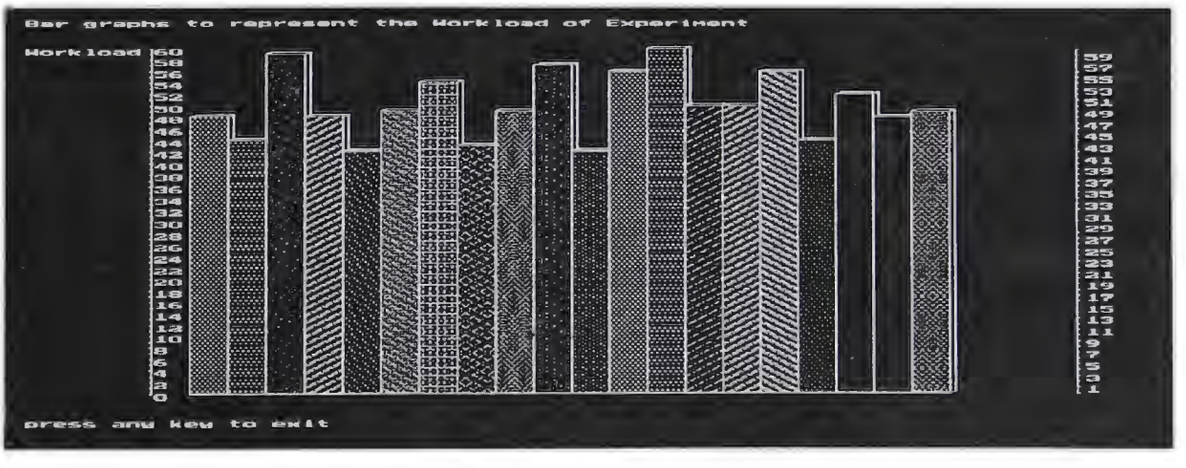
⇒ The fifth experiment with 20 processors and 1000 tuples



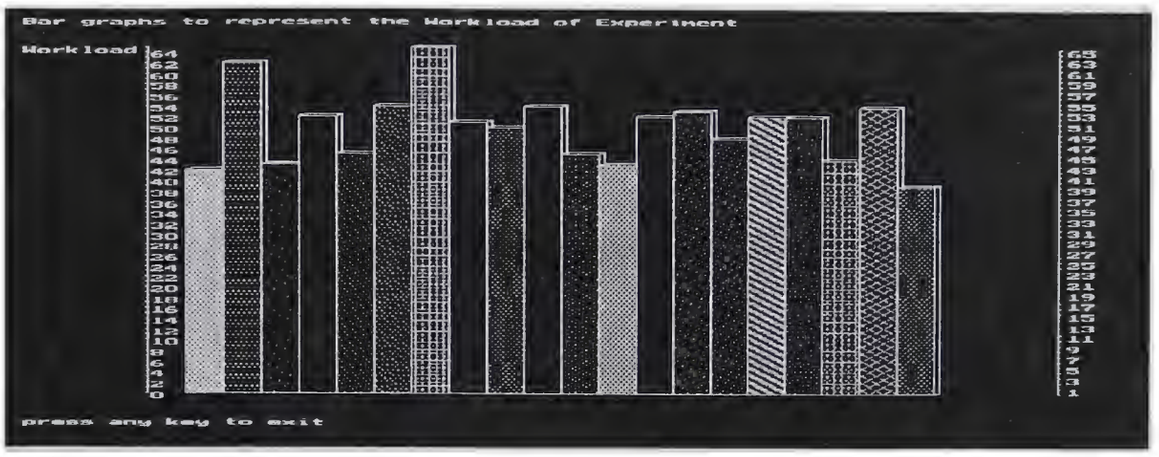
⇒ The sixth experiment with 20 processors and 1000 tuples



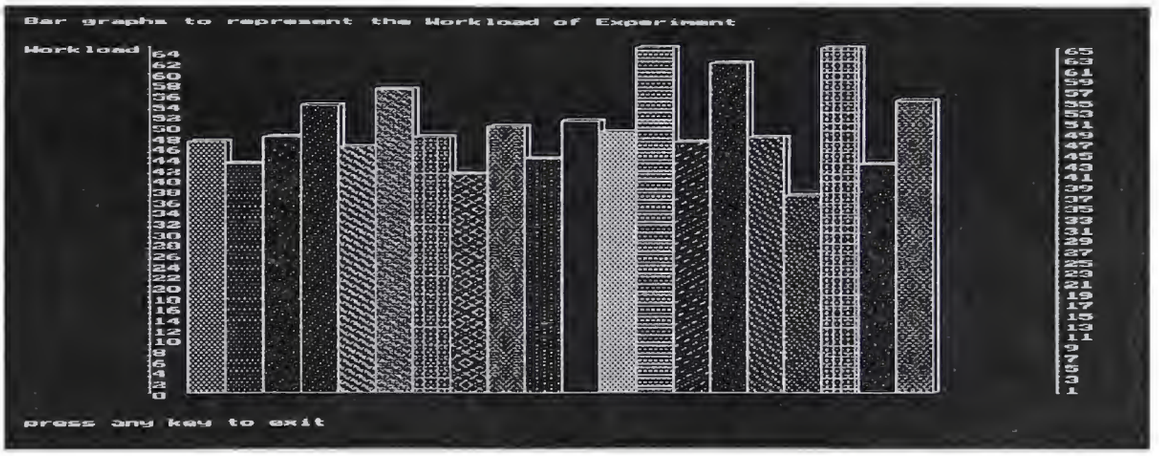
⇒ The seventh experiment with 20 processors and 1000 tuples



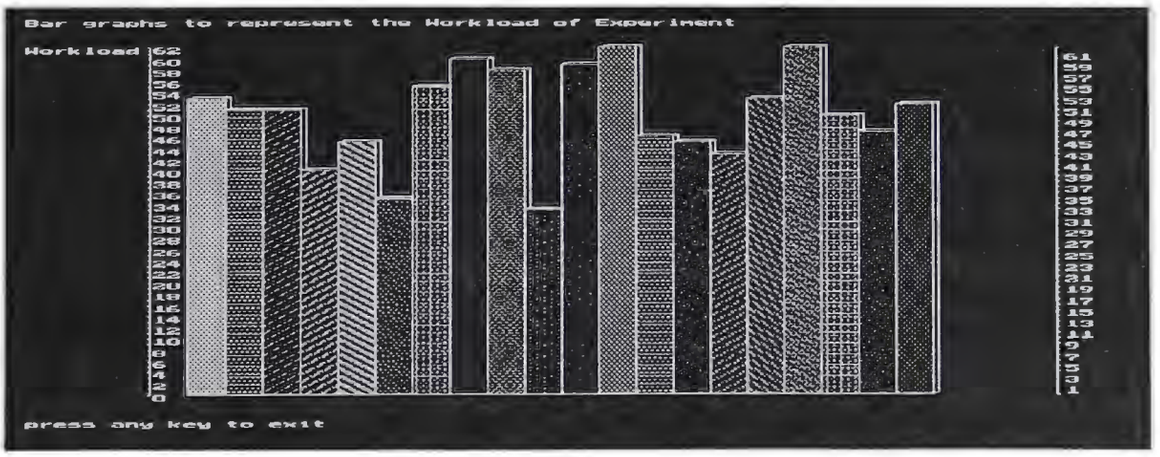
⇒ The eighth experiment with 20 processors and 1000 tuples



⇒ The ninth experiment with 20 processors and 1000 tuples

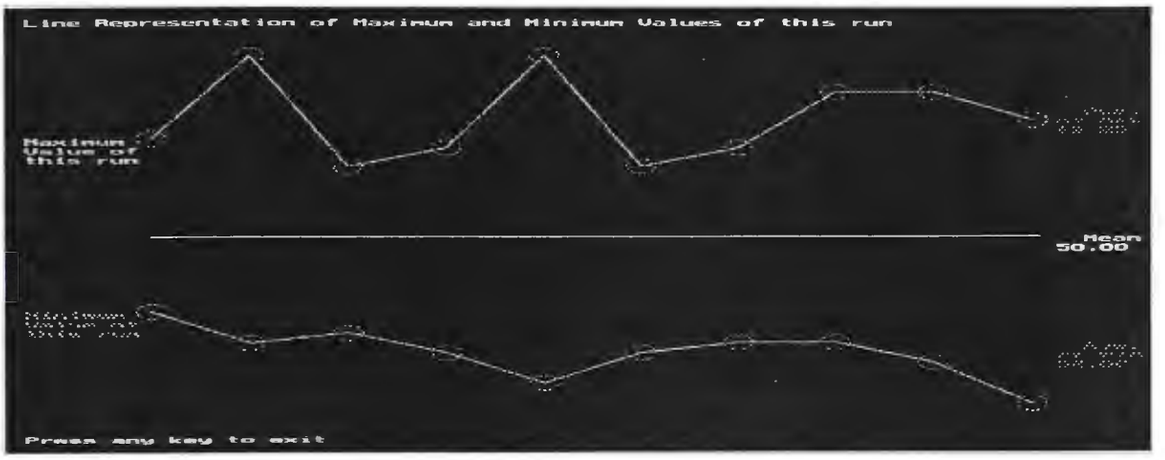


⇒ The tenth experiment with 20 processors and 1000 tuples

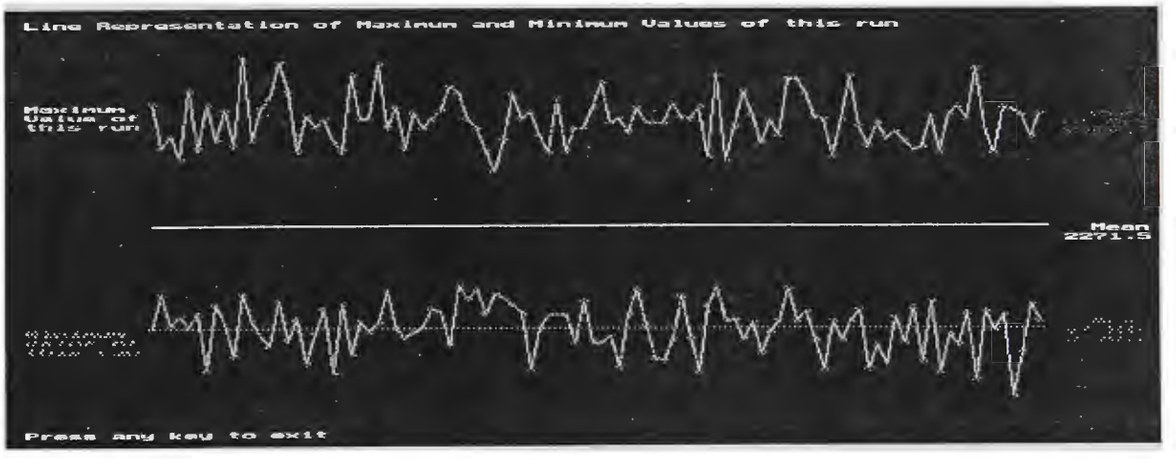


Data representation of the average of the Minimum and Maximum values of the above ten experiments in this run. In the figure,

- there are two points (minimum value and maximum value) for each experiment,
- the relative positions of the points are determined by their maximum and minimum values and standard deviation from the mean load.



Data representation of the average of the Minimum and Maximum values of 100 experiments, each of which consists of 25 processors and 56,789 tuples.



B.4 Source Codes for Main Programs

Source Code for TDBSIMU1.CPP

```
// ************************************************************
// ---- Terabyte Database Simulation Model ----
// simulating the basic relaitonal operations such as Joining, Selection, and Projection;
// collecting the existing simualtion models for Skew, Load Balancing, Updating,
```

```
// and Image Processing; providing a user friendly environment for the group
// developing research interests.
//*************************************************************

/*-------------------------------------------------------------*/
// Main program One to define Constructor and Destructor, and a few member functions
/*-------------------------------------------------------------*/

#define Uses_TView
#define Uses_TRect
#define Uses_TStatusLine
#define Uses_TStatusDef
#define Uses_TStatusItem
#define Uses_TKeys
#define Uses_MsgBox
#define Uses_fpstream
#define Uses_TEvent
#define Uses_TDeskTop
#define Uses_TApplication
#define Uses_TWindow
#define Uses_TDeskTop

#include <tv.h>
#include "bgi.h"

#if !defined( __BGII_H )
#include "bgii.h"
#endif  // __BGII_H

#include "TDBSimu.h"
#include "gadgets.h"
#include "fileview.h"
#include "puzzle.h"
#include "tdbhelp.h"
#include <help.h>
#include <dir.h>
#include <stdio.h>
#include <string.h>

//  main: create an application object.  Constructor takes care of all initialization.
//  Calling run() from TProgram makes it tick and the destructor will destroy the world.
//  File names can be specified on the command line for automatic opening.

int main(int argc, char **argv)
{
        TDBSimu tdbProgram(argc, argv);       // Application Instance.
        tdbProgram.run();
        return 0;
}

// Constructor for the application.  Command line parameters are interpreted
//  as file names and opened.  Wildcards are accepted and put up a dialog
//  box with the appropriate search path.

TDBSimu::TDBSimu( int argc, char **argv ) :
        TProgInit( &TDBSimu::initStatusLine,
                   &TDBSimu::initMenuBar,
                   &TDBSimu::initDeskTop )
{
```

```
        TView *w;
        char fileSpec[128];
        int len;
        TRect r = getExtent();      // Create the clock view.
        r.a.x = r.b.x - 9;     r.b.y = r.a.y + 1;
        clock = new TClockView( r );
        insert(clock);
        r = getExtent();            // Create the heap view.
        r.a.x = r.b.x - 13;    r.a.y = r.b.y - 1;
        heap = new THeapView( r );
        insert(heap);
        while (--argc > 0)          // Display files specified
                {                   //  on command line.
                strcpy( fileSpec, *++argv );
                len = strlen( fileSpec );
                if( fileSpec[len-1] == '\\' )
                                strcat( fileSpec, "*.*" );
                if( strchr( fileSpec, '*' ) || strchr( fileSpec, '?'))
                                openFile( fileSpec );
                else
                                {
                                w = validView( new TFileWindow(fileSpec));
                                if( w != 0 )
                                        deskTop->insert(w);
                                }
                }
        fexpand(pathToDrivers);
        bgiPath = new char[sizeof(pathToDrivers)];
        strcpy(bgiPath, pathToDrivers);
        appDriver = DETECT;
        appMode = 0;
        if (graphAppInit(appDriver, appMode, bgiPath, True) == False)
          messageBox("Cannot load graphics driver.",mfError|mfOKButton);
}

// destructor to delete the graphics path and switch off the
// graphics mode if it is on
TDBSimu::~TDBSimu()
{
        graphAppDone();
        delete bgiPath;
}

// tdbApp::getEvent()
//  Event loop to check for context help request
void TDBSimu::getEvent(TEvent &event)
{
        TWindow *w;
        THelpFile *hFile;
        fpstream *helpStrm;
        static Boolean helpInUse = False;
        TApplication::getEvent(event);
        switch (event.what)
        {
        case evCommand:
                if ((event.message.command == cmHelp)&&(helpInUse==False))
                {
                 helpInUse = True;
                 helpStrm = new fpstream("TDBHELP.HLP",ios::in|ios::binary);
```

```
                    hFile = new THelpFile(*helpStrm);
                    if (!helpStrm)
                            {
                            messageBox("Could not open help file",mfError|mfOKButton);
                            delete hFile;
                            }
                    else {
                            w = new THelpWindow(hFile, getHelpCtx());
                            if (validView(w) != 0)      {
                            execView(w);
                            destroy( w );
                            }
                            clearEvent(event);
                    }
//                  helpInUse = False;
                  }
                  break;
        case evMouseDown:
                  if (event.mouse.buttons != 1)
                          event.what = evNothing;
                  break;
        }
        helpInUse=False;
}

// Create statusline.
TStatusLine *TDBSimu::initStatusLine( TRect r )
{
        r.a.y = r.b.y - 1;
        return (new TStatusLine( r,
                  *new TStatusDef( 0, 0xFFFF ) +
                   *new TStatusItem( "~F1~ Help", kbF1, cmHelp ) +
                   *new TStatusItem( "~Alt-X~ Exit", kbAltX, cmQuit ) +
                   *new TStatusItem( "~F10~ Menu", kbF10, cmMenu )
                   )
        );
}

// Puzzle function
void TDBSimu::puzzle()
{
  TPuzzleWindow *puzz = (TPuzzleWindow *) validView(new TPuzzleWindow);
       if(puzz != 0)
       {
       puzz->helpCtx = hcPuzzle;
       deskTop->insert(puzz);
       }
}

// retrieveDesktop() function ( restores the previously stored Desktop )
void TDBSimu::retrieveDesktop()
{
  struct ffblk ffblk;
  if (findfirst("TDBSimu.DST", &ffblk, 0))
       messageBox("Could not find desktop file", mfOKButton | mfError);
  else
       {
       fpstream *f = new fpstream("TDBSimu.DST", ios::inlios::binary);
       if( !f )
```

```
            messageBox("Could not open desktop file", mfOKButton | mfError);
          else
            {
            TDBSimu::loadDesktop(*f);
            if( !f )
            messageBox("Error reading desktop file", mfOKButton | mfError);
            }
          delete f;
          }
}

// saveDesktop() function -- saves the DeskTop by
// calling storeDesktop function
void TDBSimu::saveDesktop()
{
         fpstream *f = new fpstream("TDBSimu.DST", ios::outlios::binary);

   if( f )
         {  TDBSimu::storeDesktop(*f);
         if( !f )
            {
            messageBox("Could not create TDBSimu.DST.",mfOKButton|mfError);
            delete f;
            ::remove("TDBSimu.DST");
            return;
            }
         }
   delete f;
}

// writeView() function ( writes a view object to a resource file )
static void writeView(TView *p, void *strm)
{
  fpstream *s = (fpstream *) strm;
  if (p != TProgram::deskTop->last)
    *s << p;
}

// storeDesktop() function ( stores the Desktop in a resource file )
void TDBSimu::storeDesktop(fpstream& s)
{
 deskTop->forEach(::writeView, &s);  s << 0;
}

// Tile function
void TDBSimu::tile()
{   deskTop->tile( deskTop->getExtent() );          }
```

Source Code for TDBSIMU2.CPP

```
/*-------------------------------------------------------------------*/
// Main program Two: Member function definition and Event function declaration
/*-------------------------------------------------------------------*/

#define Uses_TDialog
#define Uses_TCheckBoxes
#define Uses_TRect
```

```
#define Uses_TStaticText
#define Uses_TButton
#define Uses_TEvent
#define Uses_TWindow
#define Uses_TColorGroup
#define Uses_TColorItem
#define Uses_TColorDialog
#define Uses_TPalette
#define Uses_TDeskTop
#define Uses_TApplication
#define Uses_TChDirDialog
#define Uses_THistory
#define Uses_TInputLine
#define Uses_MsgBox
#define Uses_TLabel
#define Uses_TSItem
#define Uses_TProgram

#include <tv.h>
#include "bgi.h"              // graphic displaying header file
//#include "dialog.h"         // dialog header file
#include "sk_simu.h"          // skew simulation header file
#include "TDBSimu.h"          // tdbsimu header file
#include "tvcmds.h"
#include "tdbhelp.h"
#include "ascii.h"
#include "calendar.h"
#include "calc.h"
#include "cost51.h"
#if !defined( __STRING_H )
#include <string.h>
#endif  // __STRING_H
#if !defined( __FSTREAM_H )
#include <fstream.h>
#endif  // __FSTREAM_H
#if !defined( __STRSTREA_H )
#include <strstrea.h>
#endif  // __STRSTREA_H
#if !defined( __IO_H )
#include <io.h>
#endif  // __IO_H
#include <stdlib.h>
const MAXSIZE = 80;
//  DOS Shell Command.
void TDBSimu::shell()
{
        suspend();
        system("cls");
        cout << "Type EXIT to return...";
        system( getenv( "COMSPEC"));
        resume();
        redraw();
}

// TDBApp::handleEvent()
//  Event loop to distribute the work.
void TDBSimu::handleEvent(TEvent &event)
{
        TApplication::handleEvent(event);
```

```
if (event.what == evCommand)
{
        switch (event.message.command)
                        {
        case cmAboutCmd:        // About Dialog Box
            aboutDlgBox();
            break;
        case cmCalendarCmd:     // Calendar Window
            calendar();
            break;
        case cmAsciiCmd:        // Ascii Table
            asciiTable();
            break;
        case cmCalcCmd:         // Calculator
            calculator();
            break;
        case cmPuzzleCmd:       // Puzzle
            puzzle();
            break;
        case cmOpenCmd:         // View a file
            openFile("*.*");
            break;
        case cmChDirCmd:        // Change directory
            changeDir();
            break;
        case cmDOS_Cmd:         // DOS shell
            shell();
            break;
        case cmSelectionCmd:    // selection
            randomize();
            Selection();
            break;
        case cmWDProjectionCmd:  // projection with duplicate
            WDProjection();
            break;
        case cmWODProjectionCmd: // projection without duplicate
            randomize();
            WODProjection();
            break;
        case cmAggregationCmd:   // Change directory
            Aggregation();
            break;
        case cmNLJoinCmd:       // nest loops join
            randomize();
            NLJoin();
            break;
        case cmHBJoinCmd:       // hash basd join
            HBJoin();
            break;
        case cmDataMiningCmd:    // data mining
            break;
        case cmArchitectureCmd:  // architecture design
            break;
        case cmUpdatingCmd:     // Updating
            updating();
            break;
        case cmRASkewCmd:       // skew handling
            skew();
            break;
```

```
                    case cmISSkewCmd:       // skew handling
                        ISSkew();
                        break;
                    case cmDHSkewCmd:       // skew handling
                        DHSkew();
                        break;
                    case cmLoadBalancingCmd: // load balancing
                                    //SiteAllocation();
                        break;
                    case cmPEJoinCmd:       // path expression join
                        pe_join();
                        break;
                    case cmSystemCmd:       // setting systems parameters
                        SystemDlg();
                        break;
                    case cmDB_SiteCmd:      // setting DB sites
                        DB_SiteDlg();
                        break;
                    case cmWorkloadCmd:     // setting workload
                        WorkloadDlg();
                        break;
                    case cmTile:            // Tile current file windows
                        tile();
                        break;
                    case cmCascade:         // Cascade current file windows
                        cascade();
                        break;
                    case cmSetBGIPath:      // Mouse control dialog box
                        setBGIPath();
                        break;
                    case cmMouseCmd:        // Mouse control dialog box
                        mouse();
                        break;
                    case cmColorCmd:        // Color control dialog box
                        colors();
                        break;
                    case cmSaveCmd:         // Save current desktop
                        saveDesktop();
                        break;
                    case cmRestoreCmd:      // Restore saved desktop
                        retrieveDesktop();
                        break;
                    default:            // Unknown command
                        return;
                    }
                    clearEvent (event);
                    }
}

// About Box function()
void TDBSimu::aboutDlgBox()
{
        TDialog *aboutBox = new TDialog(TRect(0, 0, 55,13), "About");
        aboutBox->insert(
                new TStaticText(TRect(9, 2, 46,9),
"\003Terabyte Database Simulation Model\n\n\003\n"// These strings will be
"\003Designed by  Kevin Liu\n\003\n"     // concatenated by the compiler.
"\003Copyright (C) 1994\n\003\n"         // The Ctrl-C centers the line.
"\003Victoria University of Technology"
```

```
                    )
                    );
        aboutBox->insert(
                    new TButton(TRect(20, 10, 35, 12), " OK", cmOK, bfDefault)
                    );
        aboutBox->options |= ofCentered;
        deskTop->execView(aboutBox);
        destroy( aboutBox );
}

// Ascii Chart function
void TDBSimu::asciiTable()
{
        TAsciiChart *chart = (TAsciiChart *) validView(new TAsciiChart);
        if(chart != 0)
        {
                chart->helpCtx = hcAsciiTable;
                deskTop->insert(chart);
        }
}

// Calendar function()
void TDBSimu::calendar()
{
TCalendarWindow *cal=(TCalendarWindow *) validView(new TCalendarWindow);
        if(cal != 0)
        {
                cal->helpCtx = hcCalendar;
                deskTop->insert( cal );
        }
}

// Calculator function
void TDBSimu::calculator()
{
        TCalculator *calc = (TCalculator *) validView(new TCalculator);
        if(calc != 0)
        {
                calc->helpCtx = hcCalculator;
                deskTop->insert(calc);
        }
}

// Cascade function
void TDBSimu::cascade()
{
        deskTop->cascade( deskTop->getExtent() );
}

// Change Directory function
void TDBSimu::changeDir()
{
        TView *d = validView( new TChDirDialog( 0, cmChangeDir ) );
        if( d != 0 )
                {
                d->helpCtx = hcFCChDirDBox;
                deskTop->execView( d );
                destroy( d );
        }
```

```
}

void TDBSimu::pe_join()
{
        double td1, td2, td3, tp1, tp2, tp3, tp4;
        char again;
        suspend();
        do {
                data_entry();
                processing_td12(&td1, &td2);
                processing_td3(&td3);
                processing_tp123(&tp1, &tp2, &tp3, &tp4);
//      printf("\nTe1 = %.2lf + %.2lf = %.2lf", td1, tp1, td1+tp1);
//      printf("\nTe2 = %.2lf + %.2lf = %.2lf", td2, tp2, td2+tp2);
//      printf("\nTe3 = %.2lf + %.2lf = %.2lf", td3, tp3, td3+tp3);
                printf("\nElapsed time (without re-distribution)    %.2lf", td1+tp1);
                printf("\nElapsed time (with re-distribution)      %.2lf", td2+tp2);
                printf("\nElapsed time (with full data replication) %.2lf", td3+tp3);
                printf("\nUniprocessor Elapsed Time = %.2lf", tp4);
                printf("\nLinear speed up = %.2lf", (td3+tp4)/(td2+tp2));
                printf("\nTry again (y/n) > ");
                scanf("%c%*c", &again);
        } while(again=='y');
        resume();
        redraw();
}

void TDBSimu::updating()  // a demonstation of the Borland's random number
{               // generator
        const int maxPts = 5;
        const int CLIP_ON = 1;
        typedef TPoint PolygonType[maxPts];
        char errorMsg[MAXSIZE];
        TEvent event;
        PolygonType poly;
        ushort i, color;
        ushort maxX, maxY;
        suspend();
        if (graphicsStart() == False)
        {
        strcpy(errorMsg,grapherrormsg(graphresult()));
        strcat( errorMsg,"." );
        messageBox(errorMsg, mfError | mfOKButton);
        }
        else
        {
        maxX = getmaxx();
        maxY = getmaxy();
        outtextxy(0, (maxY - textheight("M")),
                "Press any key to return...");
        setviewport(0, 0, maxX - 1, (maxY - (textheight("M") + 5)), CLIP_ON);
        do {
                color = random(getmaxcolor()) + 1;
                setfillstyle(random(11) + 1, color);
                setcolor(color);
                for(i = 0; i < maxPts; ++i)
                {
                poly[i].x = random(maxX);
                poly[i].y = random(maxY);
```

```
                    }
                    fillpoly(maxPts, (int *) poly);
                    event.getKeyEvent();
                    } while (event.what == evNothing);
            graphicsStop();
            }
            resume();
}

// Skew Handling function  --- Random Allocation
void TDBSimu::skew()
{
            suspend();
            char ans, c_ans;    // user's response
            unsigned exp=1;     // the number of experiments at ont time
            do {
            display0();     // display the fancy screen
            exp=display1(exp);// prompt the user input
            display4(exp);   // display the user selection
            // ask the user to confirm his selection
            cputs("\n\n  Do you confirm the above variables setting ? (Y or N) ");
            fflush(stdin);
            scanf("%c",&c_ans);
            fflush(stdin);
            while (toupper(c_ans)!='Y')
                        {
                        textcolor(LIGHTGRAY);
                        clrscr();
                        exp=display1(exp);
                        display4(exp);
                        cputs("\n\n  Do you confirm the above variables setting ? (Y or N) ");
                        fflush(stdin);
                        scanf("%c",&c_ans);
                        fflush(stdin);
                        }
            display5();     // prompt the user for listing numbers
            display3();     // prompt the user for Graph Display
            textcolor(LIGHTGRAY);
            // build up the final result structures -- min, max, and sd
            randomize();
            result * pt=new result [exp];
            for (unsigned ctr=1; ctr<=exp; ctr++)
                        distribute(ctr, pt);     // random allocation per exp
            final(pt, exp);                 // display the final result
            delete pt;                      // free memory
            gotoxy(1,10);
            cputs("    Do you want to try another group of experiments ? (Y or N) ");
            fflush(stdin);
            scanf("%c",&ans);
            fflush(stdin);
            }
             while (toupper(ans)=='Y');
             resume();
             redraw();
}

// Skew Handling function  --- Interval Simulation
void TDBSimu::ISSkew()
{
```

```
        suspend();
        char ans, c_ans;      // user's response
        unsigned exp=1;       // the number of experiments at ont time
        do {
       displayIS0();        // display the fancy screen
       exp=displayIS1(exp); // prompt the user input
       display4(exp);      // display the user selection
       // ask the user to confirm his selection
       cputs("\n\n  Do you confirm the above variables setting ? (Y or N) ");
       fflush(stdin);
       scanf("%c",&c_ans);
       fflush(stdin);
       while (toupper(c_ans)!='Y')
                {
                textcolor(LIGHTGRAY);
                clrscr();
                exp=displayIS1(exp);
                display4(exp);
       cputs("\n\n  Do you confirm the above variables setting ? (Y or N) ");
                fflush(stdin);
                scanf("%c",&c_ans);
                fflush(stdin);
                }
       displayIS5();       // prompt the user for listing numbers
       display3();       // prompt the user for Graph Display
       textcolor(LIGHTGRAY);
       // build up the final result structures -- min, max, and sd
       randomize();
       result * pt=new result [exp];
       for (unsigned ctr=1; ctr<=exp; ctr++)
                distribute(ctr, pt);      // random allocation per exp
       final(pt, exp);                  // display the final result
       delete pt;                      // free memory
//     jctr=0;
       gotoxy(1,10);
       cputs("    Do you want to try another group of experiments ? (Y or N) ");
       fflush(stdin);
       scanf("%c",&ans);
       fflush(stdin);
       }
        while (toupper(ans)=='Y');
        window(1,1,80,25);
        textattr(LIGHTGRAY+(BLACK<<4));
        clrscr();
        resume();
        redraw();
}

// Skew Handling function --- Double Hashing
void TDBSimu::DHSkew()
{
        suspend();
        char ans, c_ans;     // user's response
        unsigned exp=1;      // the number of experiments at ont time
        do {
       displayDH0();       // display the fancy screen
       exp=displayDH1(exp);// prompt the user input
       displayDH4(exp);    // display the user selection
       // ask the user to confirm his selection
```

```
        cputs("\n\n  Do you confirm the above variables setting ? (Y or N) ");
        fflush(stdin);
        scanf("%c",&c_ans);
        fflush(stdin);
        while (toupper(c_ans)!='Y')
                {
                textcolor(LIGHTGRAY);
                clrscr();
                exp=displayDH1(exp);
                displayDH4(exp);
        cputs("\n\n  Do you confirm the above variables setting ? (Y or N) ");
                fflush(stdin);
                scanf("%c",&c_ans);
                fflush(stdin);
                }
        display5();     // prompt the user for listing numbers
        display3();     // prompt the user for Graph Display
        textcolor(LIGHTGRAY);
        // build up the final result structures -- min, max, and sd
        randomize();
        result * pt=new result [exp];
        for (unsigned ctr=1; ctr<=exp; ctr++)
                distributeDH(pt);       // random allocation per exp
        final(pt, exp);                 // display the final result
        delete pt;                      // free memory
        gotoxy(1,10);
        cputs("    Do you want to try another group of experiments ? (Y or N) ");
        fflush(stdin);
        scanf("%c",&ans);
        fflush(stdin);
        }
         while (toupper(ans)=='Y');
         window(1,1,80,25);
         textattr(LIGHTGRAY+(BLACK<<4));
         clrscr();
         resume();
         redraw();
}

// Color Control Dialog Box function
void TDBSimu::colors()
{
        TColorGroup &group1 =
                *new TColorGroup("Desktop") +
                        *new TColorItem("Color",            1)+
                *new TColorGroup("Menus") +
                        *new TColorItem("Normal",           2)+
                        *new TColorItem("Disabled",         3)+
                        *new TColorItem("Shortcut",         4)+
                        *new TColorItem("Selected",         5)+
                        *new TColorItem("Selected disabled", 6)+
                        *new TColorItem("Shortcut selected", 7
                );
        TColorGroup &group2 =
                *new TColorGroup("Dialogs/Calc") +
                        *new TColorItem("Frame/background",  33)+
                        *new TColorItem("Frame icons",       34)+
                        *new TColorItem("Scroll bar page",   35)+
                        *new TColorItem("Scroll bar icons",  36)+
```

```
                                        *new TColorItem("Static text",      37)+
                                        *new TColorItem("Label normal",     38)+
                                        *new TColorItem("Label selected",   39)+
                                        *new TColorItem("Label shortcut",   40
                        );
                TColorItem &item_coll1 =
                        *new TColorItem("Button normal",    41)+
                        *new TColorItem("Button default",   42)+
                        *new TColorItem("Button selected",  43)+
                        *new TColorItem("Button disabled",  44)+
                        *new TColorItem("Button shortcut",  45)+
                        *new TColorItem("Button shadow",    46)+
                        *new TColorItem("Cluster normal",   47)+
                        *new TColorItem("Cluster selected", 48)+
                        *new TColorItem("Cluster shortcut", 49
                        );
                TColorItem &item_coll2 =
                        *new TColorItem("Input normal",     50)+
                        *new TColorItem("Input selected",   51)+
                        *new TColorItem("Input arrow",      52)+
                        *new TColorItem("History button",   53)+
                        *new TColorItem("History sides",    54)+
                        *new TColorItem("History bar page", 55)+
                        *new TColorItem("History bar icons",56)+
                        *new TColorItem("List normal",      57)+
                        *new TColorItem("List focused",     58)+
                        *new TColorItem("List selected",    59)+
                        *new TColorItem("List divider",     60)+
                        *new TColorItem("Information pane", 61
                        );
                 group2 = group2 + item_coll1 + item_coll2;
                 TColorGroup &group3 =
                                *new TColorGroup("Viewer") +
                                        *new TColorItem("Frame passive",     8)+
                                        *new TColorItem("Frame active",      9)+
                                        *new TColorItem("Frame icons",      10)+
                                        *new TColorItem("Scroll bar page",  11)+
                                        *new TColorItem("Scroll bar icons", 12)+
                                        *new TColorItem("Text",             13)+
                                *new TColorGroup("Puzzle")+
                                        *new TColorItem("Frame passive",     8)+
                                        *new TColorItem("Frame active",      9)+
                                        *new TColorItem("Frame icons",      10)+
                                        *new TColorItem("Scroll bar page",  11)+
                                        *new TColorItem("Scroll bar icons", 12)+
                                        *new TColorItem("Normal text",      13)+
                                        *new TColorItem("Highlighted text", 14
                                );
                 TColorGroup &group4 =
                *new TColorGroup("Calendar") + *new TColorItem("Frame passive", 16)+
                *new TColorItem("Frame active", 17)+ *new TColorItem("Frame icons", 18)+
                *new TColorItem("Scroll bar page", 19)+ *new TColorItem("Scroll bar icons",
20)+
                *new TColorItem("Normal text", 21)+ *new TColorItem("Current day", 22)+
                *new TColorGroup("Ascii table") + *new TColorItem("Frame passive", 24)+
                *new TColorItem("Frame active", 25)+ *new TColorItem("Frame icons", 26)+
                *new TColorItem("Scroll bar page",  27)+ *new TColorItem("Scroll bar icons",
28)+*new TColorItem("Text", 29);
                 TColorGroup &group5 = group1 + group2 + group3 + group4;
```

```
        TColorDialog *c = new TColorDialog((TPalette*)0, &group5 );
        if( validView( c ) != 0 )
                {
                c->helpCtx = hcOCColorsDBox;  // set context help constant
                c->setData(&getPalette());
        if( deskTop->execView( c ) != cmCancel )
          {
          getPalette() = *(c->pal);
          deskTop->setState( sfVisible, False );
          deskTop->setState( sfVisible, True );
          }
        destroy( c );
        }
}
```

Source Code for TDBSIMU3.CPP

```
/*-------------------------------------------------------*/
// Main program Three:  Event function declaration and menu initialization
/*-------------------------------------------------------*/

#define Uses_TRect
#define Uses_TButton
#define Uses_TMenuBar
#define Uses_TSubMenu
#define Uses_TMenu
#define Uses_TMenuItem
#define Uses_TKeys
#define Uses_fpstream
#define Uses_TView
#define Uses_TPalette
#define Uses_MsgBox
#define Uses_TFileDialog
#define Uses_TApplication
#define Uses_TDeskTop
#define Uses_TStaticText
#define Uses_TDialog
#define Uses_TEventQueue
#define Uses_TInputLine
#define Uses_THistory

#include <tv.h>
#include "bgi.h"
#include "TDBSimu.h"
#include "tvcmds.h"
#include "gadgets.h"
#include "mousedlg.h"
#include "tdbhelp.h"
#include "fileview.h"

#if !defined( __STRING_H )
#include <string.h>
#endif  // __STRING_H
#include <help.h>

// Mouse Control Dialog Box function
```

```
void TDBSimu::setBGIPath()
{
        char s[MAXPATH];
        TDialog *d = new TDialog(TRect(0,0,35,8), "Path to BGI Files");
        d->options |= ofCentered;
        // Buttons
        d->insert(new TButton( TRect(23,5,33,7), "Cancel", cmCancel, bfNormal) );
        d->insert(new TButton( TRect(12,5,22,7),"O~K~", cmOK, bfDefault) );
        // Input line, history list and label
        TInputLine *pathInput = new TInputLine( TRect(3,3,30,4), 68 );
        d->insert( pathInput );
        d->insert( new THistory(TRect(30,3,33,4), pathInput, cmSetBGIPath) );
        fexpand(bgiPath);
        strcpy(s,bgiPath);
        d->setData(s);
        d = (TDialog *) validView(d);
        if (d != NULL)
        {
        if (deskTop->execView(d) == cmOK)
                {
                d->getData(s);
                delete bgiPath;
                if ( (strlen(s) > 0) && (s[strlen(s)-1] != '\\') )
                strcat(s,"\\");
                bgiPath = new char [sizeof(s)];
                strcpy(bgiPath, s);
                if ( graphAppInit(appDriver, appMode, bgiPath, True) == False )
                messageBox("Cannot load graphics driver.", mfError | mfOKButton);
                }
        destroy( d );
        }
}

void TDBSimu::mouse()
{
        TMouseDialog *mouseCage = (TMouseDialog *) validView( new
TMouseDialog() );

        if (mouseCage != 0)
                {
    mouseCage->helpCtx = hcOMMouseDBox;
    mouseCage->setData(&(TEventQueue::mouseReverse));
                if (deskTop->execView(mouseCage) != cmCancel)
      mouseCage->getData(&(TEventQueue::mouseReverse));
    }
  destroy( mouseCage );

}

// File Viewer function
void TDBSimu::openFile( char *fileSpec )
{
  TFileDialog *d= (TFileDialog *)validView(
  new TFileDialog(fileSpec, "Open a File", "~N~ame", fdOpenButton, 100 ));
  if( d != 0 && deskTop->execView( d ) != cmCancel )
    {
    char fileName[150];
    d->getFileName( fileName );
      d->helpCtx = hcFOFileOpenDBox;
```

```
      TView *w= validView( new TFileWindow( fileName ) );
      if( w != 0 )
         deskTop->insert(w);
   }
        destroy( d );
}

// "Out of Memory" function ( called by validView() )
void TDBSimu::outOfMemory()
{
   messageBox( "Not enough memory available to complete operation.",
    mfError | mfOKButton );
}

// getPalette() function ( returns application's palette )
TPalette& TDBSimu::getPalette() const
{
   static TPalette newcolor ( cpColor cHelpColor, sizeof( cpColor cHelpColor )-1 );
        static TPalette newblackwhite( cpBlackWhite cHelpBlackWhite, sizeof(
cpBlackWhite cHelpBlackWhite)-1 );
   static TPalette newmonochrome( cpMonochrome cHelpMonochrome, sizeof(
cpMonochrome cHelpMonochrome)-1 );
   static TPalette *palettes[] =
                  {
      &newcolor,
                  &newblackwhite,
      &newmonochrome
      };
   return *(palettes[appPalette]);
}

// isTileable() function ( checks a view on desktop is tileable or not )
static Boolean isTileable(TView *p, void * )
{
        if( (p->options & ofTileable) != 0)
     return True;
        else
     return False;
}

// idle() function ( updates heap and clock views for this program. )
void TDBSimu::idle()
{
   TProgram::idle();
        clock->update();
   heap->update();
   if (deskTop->firstThat(isTileable, 0) != 0 )
      {
      enableCommand(cmTile);
      enableCommand(cmCascade);
      }
   else   {
      disableCommand(cmTile);
      disableCommand(cmCascade);
      }
}

// closeView() function
static void closeView(TView *p, void *p1)
```

```
{
   message(p, evCommand, cmClose, p1);
}

// loadDesktop() function
void TDBSimu::loadDesktop(fpstream &s)
{
   TView  *p;
        if (deskTop->valid(cmClose))
                {
                deskTop->forEach(::closeView, 0);  // Clear the desktop
                do {
                        s >> p;
                        deskTop->insertBefore(validView(p), deskTop->last);
                        }
                        while (p != 0);
                }
}

// Menubar initialization.
TMenuBar *TDBSimu::initMenuBar(TRect r)
{
        // manually build sub-menu for skew
        TMenuItem *skew =new TMenuItem("~R~andom Allocation",
        cmRASkewCmd,kbNoKey, hcNoContext);
        TMenuItem *skew1=new TMenuItem("~I~nterval Simulation",
        cmISSkewCmd,kbNoKey, hcNoContext);
        skew->append(skew1);
        skew1->append(new TMenuItem("~D~ouble Hashing",
        cmDHSkewCmd,kbNoKey, hcNoContext));
        // manually build sub-menu for projection
        TMenuItem *projection=new TMenuItem("With ~D~uplicate...",
        cmWDProjectionCmd,kbNoKey,hcNoContext);
        projection->append(new TMenuItem("With~o~ut Duplicate...",
        cmWODProjectionCmd,kbNoKey,hcNoContext));
        // manually build sub-menu for join
        TMenuItem *join=new TMenuItem("~N~est Loops...",
        cmNLJoinCmd,kbNoKey,hcNoContext);
        join->append(new TMenuItem("~H~ash Based...",
        cmHBJoinCmd,kbNoKey, hcNoContext));
        // manually build sub-menu for skew
        TMenuItem *qo =new TMenuItem("~R~andomness",
        cmUpdatingCmd, kbNoKey, hcNoContext);
       TMenuItem *qo1=new TMenuItem( "S~k~ew", kbNoKey, new TMenu( *skew ));
        TMenuItem *qo2=new TMenuItem( "~L~oadBalancing",
        cmLoadBalancingCmd,kbNoKey,hcNoContext);
        TMenuItem *qo3=new TMenuItem("~P~ath Expression",
        cmPEJoinCmd,kbNoKey,hcNoContext);
        qo->append(qo1);
        qo1->append(qo2);
        qo2->append(qo3);
        r.b.y =  r.a.y + 1;
        return new TMenuBar( r, *new TSubMenu( "~\360~", 0, hcSystem ) +
        *new TMenuItem( "~A~bout...", cmAboutCmd, kbNoKey, hcSAbout ) +
         newLine() +*new TMenuItem( "~P~uzzle", cmPuzzleCmd, kbNoKey, hcSPuzzle
) +*new TMenuItem( "Ca~l~endar", cmCalendarCmd, kbNoKey, hcSCalendar)+
        *new TMenuItem( "Ascii ~T~able", cmAsciiCmd, kbNoKey,
hcSAsciiTable)+*new TMenuItem( "~C~alculator", cmCalcCmd, kbNoKey,
hcCalculator)+ *new TSubMenu( "~F~ile", 0, hcFile ) +*new TMenuItem( "~O~pen...",
```

```
cmOpenCmd, kbF3, hcFOpen, "F3" )+*new TMenuItem( "~C~hange Dir...",
cmChDirCmd, kbNoKey, hcFChangeDir)+ newLine() +*new TMenuItem( "~D~OS
Shell", cmDOS_Cmd, kbNoKey, hcFDosShell)+*new TMenuItem( "E~x~it", cmQuit,
kbAltX, hcFExit, "Alt-X")+ *new TSubMenu( "O~p~erations", 0, hcNoContext)+*new
TMenuItem( "~S~election...", cmSelectionCmd,kbNoKey, hcNoContext)+*new
TMenuItem( "~P~rojection",kbNoKey, new TMenu( *projection ))+   *new TMenuItem(
"~A~ggregation...",cmAggregationCmd,kbNoKey,hcNoContext)+newLine()+  *new
TMenuItem( "~J~oin",kbNoKey, new TMenu( *join ))+  *new TSubMenu(
"~A~pplications", 0, hcNoContext) +*new TMenuItem( "~D~ata
Mining",cmDataMiningCmd,kbNoKey,hcNoContext)+newLine()+*new
TMenuItem("~A~rchitecture",cmArchitectureCmd,kbNoKey,hcNoContext)+newLine()+
*new TMenuItem("~Q~uery Optimization",kbNoKey,new TMenu( *qo) ) +  *new
TSubMenu( "~S~etting", 0, hcNoContext) +*new TMenuItem( "~S~ystem...",
cmSystemCmd, kbNoKey,hcNoContext ) +*new TMenuItem( "D~B~_Site...",
cmDB_SiteCmd,  kbNoKey,hcNoContext)+ newLine()+*new TMenuItem(
"~W~orkload...",  cmWorkloadCmd, kbNoKey,hcNoContext )+ *new TSubMenu(
"~W~indows", 0, hcWindows ) +  *new TMenuItem( "~R~esize/move", cmResize,
kbCtrlF5, hcWSizeMove, "Ctrl-F5" ) +*new TMenuItem( "~Z~oom", cmZoom, kbF5,
hcWZoom, "F5" ) +*new TMenuItem( "~N~ext", cmNext, kbF6, hcWNext, "F6" )
+*new TMenuItem( "~C~lose", cmClose, kbAltF3, hcWClose, "Alt-F3" ) +*new
TMenuItem( "~T~ile", cmTile, kbNoKey, hcWTile ) +*new TMenuItem( "C~a~scade",
cmCascade, kbNoKey, hcWCascade )+  *new TSubMenu( "~O~ptions", 0, hcOptions )
+*new TMenuItem( "Set ~B~GI Path", cmSetBGIPath, kbNoKey, hcNoContext ) +
newLine()+*new TMenuItem( "~M~ouse...", cmMouseCmd, kbNoKey, hcOMouse )
+*new TMenuItem( "~C~olors...", cmColorCmd, kbNoKey, hcOColors ) +
*new TMenuItem( "~S~ave desktop", cmSaveCmd, kbNoKey, hcOSaveDesktop ) +
*new TMenuItem( "~R~etrieve desktop",cmRestoreCmd,kbNoKey,hcORestoreDesktop )
 );
}
```

Source Code for TDBSIMU4.CPP

```
#define Uses_TDialog
#define Uses_TCheckBoxes
#define Uses_TStaticText
#define Uses_TEvent
#define Uses_TWindow
#define Uses_TPalette
#define Uses_TApplication
#define Uses_TChDirDialog
#define Uses_THistory
#define Uses_TInputLine
#define Uses_MsgBox
#define Uses_TLabel
#define Uses_TSItem
#define Uses_TProgram
#define Uses_TRect
#define Uses_TButton
#define Uses_TLabel
#define Uses_TView
#define Uses_TDeskTop
class opstream;
#include <tv.h>
#include "bgi.h"              // graphic displaying header file
#include "TDBSimu.h"
#include "tvcmds.h"
#include "tdbhelp.h"
```

```
#include "dialog.h"          // dialog header file
#include "tdbsimu4.h"
//#if !defined( __SITEALL_H )
//#include "siteall.h"
//#endif // __SITEALL_H

void TDBSimu::SystemDlg()
{
        TDialog *sdialog = new TDialog(TRect(0,0,38,14),"System Parameters");
        sdialog->options |= ofCentered;
        abStdInputLine(sdialog,2,2,"~NoNode~","~NoRela~");
        abStdInputLine(sdialog,2,4,"~IntCst~","~TrmCst~");
        abStdInputLine(sdialog,2,6,"~NetBan~","~BlkSiz~");
        TCheckBoxes *check= new TCheckBoxes(TRect(3,9,35,10),
                        new TSItem("~G~raphic Show",0));
        sdialog->insert(check);
        cancelOKStdButtonsH(sdialog,3,11);
        sdialog->selectNext(False);
        sdialog->setData(&data);
        ushort result=TProgram::deskTop->execView(sdialog);
        if (result==cmOK) {
                sdialog->getData(&data);
                }
        TObject::destroy(sdialog);
}

void TDBSimu::DB_SiteDlg()
{
        TDialog *sdialog = new TDialog(TRect(0,0,38,19),"DB Sites Parameters");
        sdialog->options |= ofCentered;
        TView *c;
        sdialog->insert(c=new TInputLine(TRect(10,2,35,3),numLen));
        sdialog->insert(new TLabel(TRect(2,2,10,3), "~NoPro~", c));
        abStdInputLine(sdialog,2,4,"~SekTim~","~LatTim~");
        abStdInputLine(sdialog,2,6,"~XerTim~","~RevTim~");
        abStdInputLine(sdialog,2,8,"~PCTim~","~WriTim~");
        abStdInputLine(sdialog,2.10,"~PETim~","~MemSiz~");
        abStdInputLine(sdialog,2,12,"~PHTim~","~PPTim~");
        cancelOKStdButtonsH(sdialog.3,15);
        sdialog->selectNext(False);
        sdialog->setData(&data1);
        ushort result=TProgram::deskTop->execView(sdialog);
        if (result==cmOK) {
                sdialog->getData(&data1);
                }
        TObject::destroy(sdialog);
}

void TDBSimu::WorkloadDlg()
{
        TDialog *sdialog = new TDialog(TRect(0,0,38,15),"Workload Parameters");
        sdialog->options |= ofCentered;
        abStdInputLine(sdialog,2,2,"~LenPre~","~NoColn~");
        abStdInputLine(sdialog,2,4,"~QuySiz~","~TupSiz~");
        TView *cb;
        sdialog->insert(cb=new TInputLine(TRect(10,6,35,7),numLen));
        sdialog->insert(new TLabel(TRect(2,6,10,7), "~R_Size~", cb));
        TView *cc;
        sdialog->insert(cc=new TInputLine(TRect(10,8,35,9),numLen));
```

```
        sdialog->insert(new TLabel(TRect(2,8,10,9), "~S_Size~", cc));
        cancelOKStdButtonsH(sdialog,3,12);
        sdialog->selectNext(False);
        sdialog->setData(&data2);
        ushort result=TProgram::deskTop->execView(sdialog);
        if (result==cmOK) {
                sdialog->getData(&data2);
                }
        TObject::destroy(sdialog);
}

void TDBSimu::Selection()
{
        static SelData sdata;
        static double a, b;
        static int sig=5;
        static char str[12], str1[12];
        operation();
        TDialog *seledia = new TDialog(TRect(0,0,38,9),"Selection");
        seledia->options |= ofCentered;
        TView *s1;
        seledia->insert(s1=new TInputLine(TRect(10,2,35,3),12));
        seledia->insert(s1);
        seledia->insert(new TLabel(TRect(2,2,10,3), "~selfac~", s1));
        TView *s2;
        seledia->insert(s2=new TInputLine(TRect(10,4,35,5),12));
        seledia->insert(s2);
        seledia->insert(new TLabel(TRect(2,4,10,5), "~Ts(sel)~", s2));
        a=rand()%100;
        a=a/100;
        gcvt(a,sig, str);
//      bc=selfac/32767;            // 0 <= selafc <= 1
        b=ReaTim*rk+rpk*atof(data.sNetBan)+rpk*atof(data1.sRevTim)+
                atof(data2.sR_Size)*atof(data2.sLenPre)*atof(data1.sPETim)/Totn+
                ceil(a*atof(data2.sR_Size)*atof(data2.sTupSiz)/
                atof(data.sBlkSiz)/Totn)*atof(data1.sWriTim);
        gcvt(b,sig, str1);
        strcpy (sdata.selfac,str);
        strcpy (sdata.ts, str1);
        seledia->selectNext(False);
        seledia->setData(&sdata);
        seledia->insert(
                new TButton(TRect(14, 6, 24, 8), " OK", cmOK, bfDefault)
        );
        deskTop->execView(seledia);
        destroy(seledia);
}

void TDBSimu::WDProjection()
{
        static WDPData wdpdata;
        static double a;
        static int sig=5;
        static char str[12];
        operation();
        TDialog *seledia = new TDialog(TRect(0,0,38,7),"Projection (Dupl)");
        seledia->options |= ofCentered;
        TView *s1;
        seledia->insert(s1=new TInputLine(TRect(10,2,35,3),12));
```

```
        seledia->insert(s1);
        seledia->insert(new TLabel(TRect(2,2,10,3), "~Tp1(wd)~", s1));
        a=ReaTim*rk+rpk*atof(data.sNetBan)+rpk*atof(data1.sRevTim)+
                atof(data2.sR_Size)*atof(data2.sNoColn)*atof(data1.sPETim)/Totn+
                ceil(atof(data2.sR_Size)*atof(data2.sTupSiz)/atof(data.sBlkSiz)/Totn)*
                atof(data1.sWriTim);
        gcvt(a,sig, str);
        strcpy (wdpdata.tp1, str);
        seledia->selectNext(False);
        seledia->setData(&wdpdata);
        seledia->insert(
                new TButton(TRect(14, 4, 24, 6), " OK", cmOK, bfDefault)
        );
        deskTop->execView(seledia);
        destroy(seledia);
}

void TDBSimu::WODProjection()
{
        static double tp1;
        static WODPData wodpdata;
        static double a, b;
        static int sig=5;
        static char str[12], str1[12];
        operation();
        TDialog *seledia = new TDialog(TRect(0,0,38,9),"Projection (No Dupl)");
        seledia->options |= ofCentered;
        TView *s1;
        seledia->insert(s1=new TInputLine(TRect(10,2,35,3),12));
        seledia->insert(s1);
        seledia->insert(new TLabel(TRect(2,2,10,3), "~UniRat~", s1));
        TView *s2;
        seledia->insert(s2=new TInputLine(TRect(10,4,35,5),12));
        seledia->insert(s2);
        seledia->insert(new TLabel(TRect(2,4,10,5), "~Tp2(nd)~", s2));
        a=rand()%100;
        a=a/100;
        gcvt(a,sig, str);
        tp1=ReaTim*rk+rpk*atof(data.sNetBan)+rpk*atof(data1.sRevTim)+
                atof(data2.sR_Size)*atof(data2.sNoColn)*atof(data1.sPETim)/Totn+
                ceil(atof(data2.sR_Size)*atof(data2.sTupSiz)/atof(data.sBlkSiz)/Totn)*
                atof(data1.sWriTim);
        b=tp1+ceil(a*atof(data2.sR_Size)*atof(data2.sTupSiz)/
        atof(data.sBlkSiz)/Totn)*atof(data1.sWriTim);
        gcvt(b,sig, str1);
        strcpy (wodpdata.UniRat,str);
        strcpy (wodpdata.tp2, str1);
        seledia->selectNext(False);
        seledia->setData(&wodpdata);
        seledia->insert(
                new TButton(TRect(14, 6, 24, 8), " OK", cmOK, bfDefault)
        );
        deskTop->execView(seledia);
        destroy(seledia);
}

void TDBSimu::Aggregation()
{
        static AggData aggdata;
```

```
        static double a;
        static int sig=5;
        static char str[12];
        operation();
        TDialog *seledia = new TDialog(TRect(0,0,38,7),"Aggregation (min)");
        seledia->options |= ofCentered;
        TView *s1;
        seledia->insert(s1=new TInputLine(TRect(10,2,35,3),12));
        seledia->insert(s1);
        seledia->insert(new TLabel(TRect(2,2,10,3), "~Ta1(mi)~", s1));
        a=ReaTim*rk+rpk*atof(data.sNetBan)+rpk*atof(data1.sRevTim)+
                (atof(data2.sR_Size)-1)*atof(data1.sPCTim)/Totn+
                ceil(atof(data2.sTupSiz)/atof(data.sBlkSiz))*atof(data.sNetBan)+
        ceil(atof(data2.sTupSiz)*Totn/atof(data.sBlkSiz))*atof(data1.sRevTim)+
                (Totn-1)*atof(data1.sPCTim)+
                ceil(atof(data2.sTupSiz)/atof(data.sBlkSiz))*atof(data1.sWriTim);
        gcvt(a,sig, str);
        strcpy (aggdata.ta1, str);
        seledia->selectNext(False);
        seledia->setData(&aggdata);
        seledia->insert(
                new TButton(TRect(14, 4, 24, 6), " OK", cmOK, bfDefault)
        );
        deskTop->execView(seledia);
        destroy(seledia);
}

void TDBSimu::NLJoin()
{
        static NLJoinData nljdata;
        static double a, b;
        static int sig=5;
        static char str[12], str1[12];
        operation();
        TDialog *seledia = new TDialog(TRect(0,0,38,9),"Join (Nest Loops)");
        seledia->options |= ofCentered;
        TView *s1;
        seledia->insert(s1=new TInputLine(TRect(10,2,35,3),12));
        seledia->insert(s1);
        seledia->insert(new TLabel(TRect(2,2,10,3), "~selfac~", s1));
        TView *s2;
        seledia->insert(s2=new TInputLine(TRect(10,4,35,5),12));
        seledia->insert(s2);
        seledia->insert(new TLabel(TRect(2,4,10,5), "~Tjn(nl)~", s2));
        a=rand()%100;
        a=a/100;
        gcvt(a,sig, str);
        b=ReaTim*rk+rpk*atof(data.sNetBan)+ReaTim*sk+
                sk*atof(data.sNetBan)+(rk+sk)*atof(data1.sRevTim)/Totn+
                atof(data2.sR_Size)*atof(data2.sS_Size)*atof(data1.sPCTim)+
                ceil(a*atof(data2.sR_Size)*atof(data2.sS_Size)*
                atof(data2.sTupSiz)/atof(data.sBlkSiz)/Totn)*atof(data1.sWriTim);
        gcvt(b,sig, str1);
        strcpy (nljdata.jselfac,str);
        strcpy (nljdata.tjn, str1);
        seledia->selectNext(False);
        seledia->setData(&nljdata);
        seledia->insert(
```

```
                new TButton(TRect(14, 6, 24, 8), " OK", cmOK, bfDefault)
        );
        deskTop->execView(seledia);
        destroy(seledia);
}

void TDBSimu::HBJoin()
{
        static HBJoinData hbjdata;
        static double a, b;
        static int sig=5;
        static char str[12], str1[12];
        operation();
        TDialog *seledia = new TDialog(TRect(0,0,38,9),"Join (Hash Based)");
        seledia->options |= ofCentered;
        TView *s1;
        seledia->insert(s1=new TInputLine(TRect(10,2,35,3),12));
        seledia->insert(s1);
        seledia->insert(new TLabel(TRect(2,2,10,3), "~selfac~", s1));
        TView *s2;
        seledia->insert(s2=new TInputLine(TRect(10,4,35,5),12));
        seledia->insert(s2);
        seledia->insert(new TLabel(TRect(2,4,10,5), "~Tjh(hb)~", s2));
        a=rand()%100;
        a=a/100;
        gcvt(a,sig, str);
//      bc=selfac/32767;            // 0 <= selafc <= 1
        b=ReaTim*rk+rpk*atof(data.sNetBan)+ReaTim*sk+
                spk*atof(data.sNetBan)+(rk+sk)*atof(data1.sRevTim)/Totn+
                atof(data2.sS_Size)*atof(data1.sPHTim)/Totn+atof(data2.sR_Size)*
                atof(data1.sPPTim)/Totn+ceil(a*atof(data2.sR_Size)*
                atof(data2.sS_Size)*atof(data2.sTupSiz)/atof(data.sBlkSiz)/Totn)*
                atof(data1.sWriTim);
        gcvt(b,sig, str1);
        strcpy (hbjdata.jselfac,str);
        strcpy (hbjdata.tjhh, str1);
        seledia->selectNext(False);
        seledia->setData(&hbjdata);
        seledia->insert(
                new TButton(TRect(14, 6, 24, 8), " OK", cmOK, bfDefault)
        );
        deskTop->execView(seledia);
        destroy(seledia);
}

/* void TDBSimu::SiteAllocation()
{
static int h,i,j,k, avgsize;
get_para();
num_site=0;
for(h=0;h<8;h++)
{
num_site =num_site+2;
printf("sites=%d: ",num_site);
fp=fopen("c:\\tempdir\\set_qry","r");
fscanf(fp,"%d %d",&num_query,&num_qtype);
num_run =0;  sum_phase =0;
sum_ttime =0.0;  sum_cratio =0.0;  sum_ut =0.0;
mtd_hash=0; mtd_part=0;
```

```
for(i=0;i<num_qtype;i++)          // j5 to j11
        {
        qry_num_run =0; qry_sum_phase =0;
        qry_sum_ttime =0.0;  qry_sum_cratio =0.0;  qry_sum_ut =0.0;
        qry_mtd_hash=0; qry_mtd_part=0;
        for(j=0;j<num_query;j++)// each query in Jx
        {
        root = get_query();
        for(k=0;k<nnn;k++)        // varying rel sizes
                {
                init(root);
                avgsize =compute_avgsize(root)/num_rel;   // for varying cardinalilty
         //     printf("RUN %d -- ",num_run);
         //     print_inorder(root);
                phase_partition();
                stat();
                update_size(root,avgsize,szfactor);
                }
        free_node(root);
        }
         // resultqry();
         }
result();
fclose(fp);
}
} */
```

APPENDIX C
SIMULATION MODEL FOR INTRA-QUERY PROCESSOR ALLOCATION

Source Code for Intra-query Processor Allocation

```
/*****************************************************************/
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <math.h>
#define MAX_NODE 48
#define MAX_PHASE 24
#define MAX_OP 24
#define w1 0.0001
#define w2 0.0001
#define w3 0.0001
#define thash 0.0001
#define tdata 0.00003
#define tinit 0.001

/*****************************************************************/
typedef struct nnode {
        int pid;
        int type;
        int num_tuple;
        float skew;
        int num_pro;                    // real no. of processors
        float ttime;
        int allocated;
        float op_time;
        long op_num_pro;        // optimal no. of processors
        struct nnode *lnode, *rnode, *pnode;
        } NODE;

typedef struct subset_ready {
        int num_op_phase;
        float ph_time;
```

```
        int num_pro_phase;
        NODE *nodeptr[MAX_OP];
        } SUBSET;

long num_site=100;                  // total no. of processors
int num_op=4, num_phase;
//float sel=1.0;            // assume join_sel=1/(r+s)
float sum_ph_time;              // query execution time
float avg_qtime=0.0;
int sum_no_pro;  // real num of processors for the query
ifstream ifquery;
ofstream ofquery;
NODE *root=NULL;
SUBSET phase[MAX_PHASE];

void print_tree(NODE *p);
NODE *get_query(void);
void phase_partition(void);
void find_ready_op(NODE *p, int phs);
void print_phase(void);
NODE *optimize(NODE *q, int rz, int sz, float skew);
float hash_join(long n, int rz, int sz, float skew);
float Harmo (int k, float skew);
void merge_point(NODE *p, int phs);
void time_equal(int temp_phs);
void collect_data(NODE *p);

/*************************************************************/
int main()
{
int num_query=1, num_qtype=1;
ifquery.open("ifquery.dat");                    // checking I/O
if (ifquery.fail())
        {
        cout<<"\n\nCan't open file\n";
        return 1;
        }
ofquery.open("ofquery.dat");
if (ofquery.fail())
        {
        cout<<"\n\nCan't open file\n";
        return 1;
        }
ifquery>>num_query>>num_qtype>>num_site;
ofquery<<"\n                        Optimised Processor Algorithm";
ofquery<<"\n\n%%%  This is the beginning of the simulation program";
ofquery<<"\nnum_query="<<num_query<<", num_qtype="<<num_qtype<<",
num_pro="<<num_site;
cout<<"\nnum_site "<<num_site;
for (int i=1; i<=num_query; i++)
        {
        sum_ph_time=0.0;                    // initialisation
        sum_no_pro=0;
        root=get_query();           // establish the tree
        cout<<"\n\nNew Query"<<i;
        ofquery<<"\n\nNew Query "<<i;
        collect_data(root);    // collect optimal time and no. of pro
        cout<<endl<<sum_no_pro<<endl;
        if (sum_no_pro>num_site)      // not enough processors
```

```
                {
                sum_ph_time=0.0;
                sum_no_pro=0;
                phase_partition();    // merge-point approach
                print_tree(root);
                print_phase();
                }
        else
                ofquery<<"\n========= Global Optimisation is achieved =="
            <<"\nOptimal processors required are "<<sum_no_pro;
        ofquery<<"\nThe execution time for this query is "<<sum_ph_time;
        avg_qtime+=sum_ph_time;
        }
avg_qtime = avg_qtime/num_query;
ofquery<<"\nthe average of this group of queries exe time is "<<avg_qtime;
ofquery<<"\n%%%  This is the end of the Simulation program ";
ifquery.close();
ofquery.close();
return 0;
}

/*************************************************************/
void print_phase(void)   // print the operations within each phase
{
for (int i=0; i<num_phase; I++)
        {
        ofquery<<"\nphase["<<i<<"].num_op="<<phase[i].num_op_phase<<",
ph_time=";
        ofquery<<phase[i].ph_time<<", num_pro="<<phase[i].num_pro_phase;
        sum_ph_time = sum_ph_time+phase[i].ph_time;
        for (int j=0; j<phase[i].num_op_phase; j++)
        ofquery<<", pid="<<phase[i].nodeptr[j]->pid;
        }
}

/*************************************************************/
void collect_data(NODE *p)     // travel the binary tree
{
if (p!=NULL) {
        collect_data(p->lnode);
        collect_data(p->rnode);
        if (p->type>0)
                {
                p=optimize(p,p->lnode->num_tuple,p->rnode->num_tuple,p->skew);
                cout<<"\npid "<<p->pid+100<<"\t"<<p->lnode->num_tuple<<"\t"
                <<"\t"<<p->rnode->num_tuple<<","<<p->skew;
                if ((p->lnode->type>0) || (p->rnode->type>0))
                  {
                if (p->lnode->op_num_pro+p->rnode->op_num_pro > p->op_num_pro)
                  p->op_num_pro=p->lnode->op_num_pro+p->rnode->op_num_pro;
                if (p->lnode->op_time > p->rnode->op_time)
                        {p->op_time += p->lnode->op_time;}
                else
                        {p->op_time += p->rnode->op_time;}
                  }
/*              ofquery<<"\npid="<<p->pid+100<<", num_tuple="<<p->num_tuple
                <<", skew="<<p->skew<<", type"<<p->type<<", op-time="
                <<p->op_time<<", op-num-pro"<<p->op_num_pro;
```

```
                    if (p->lnode->type>0)
                            {
                            ofquery<<", lnode="<<p->lnode->pid+100;
                            }
                    else
                            ofquery<<", lnode="<<p->lnode->pid;
                    if (p->type==1)
                            {
                            if (p->rnode->type>0)
                                { ofquery<<", rnode="<<p->rnode->pid+100; }
                            else
                                    ofquery<<", rnode="<<p->rnode->pid;
                            }
                    else
                            ofquery<<"rnode = NULL\n";
*/
                    }
/*      else
                    ofquery<<"\npid="<<p->pid<<", num_tuple="<<p->num_tuple
                    <<", skew="<<p->skew<<", type"<<p->type;
*/
                    }
sum_ph_time=p->op_time;
sum_no_pro=p->op_num_pro;
}

/*************************************************************/
void phase_partition(void)     // phase partitioning strategy
{
int temp_phs,i;

temp_phs=0;
phase[temp_phs].num_op_phase=0;
phase[temp_phs].num_pro_phase=0;
phase[temp_phs].ph_time = -1;
find_ready_op(root,temp_phs);
cout<<"\nph_time0 = "<<phase[temp_phs].ph_time;
if (phase[temp_phs].num_pro_phase > num_site){
        phase[temp_phs].num_op_phase=0;
        phase[temp_phs].num_pro_phase=0;
        phase[temp_phs].ph_time = -1;
        merge_point(root,temp_phs);
        cout<<"\nph_time1 = "<<phase[temp_phs].ph_time;
        if (phase[temp_phs].num_pro_phase > num_site)
                        time_equal(temp_phs);
        cout<<"\nph_time2 = "<<phase[temp_phs].ph_time;}
//cout<<"\nThe number of operations in this phase ="<<phase[temp_phs].num_op_phase;

phase[temp_phs].nodeptr[phase[temp_phs].num_op_phase] = NULL;
while(phase[temp_phs].num_op_phase >0)
   {
   i = 0;                                          // mark op as allocated
   while(phase[temp_phs].nodeptr[i] !=NULL)
        { phase[temp_phs].nodeptr[i]->allocated = 1; i++; }
   temp_phs++;
   phase[temp_phs].num_pro_phase=0;
   phase[temp_phs].ph_time = -1;
   phase[temp_phs].num_op_phase=0;
   find_ready_op(root,temp_phs);
```

```
   cout<<"\nph_time0 = "<<phase[temp_phs].ph_time;
   if (phase[temp_phs].num_pro_phase > num_site) {
          phase[temp_phs].num_op_phase=0;
          merge_point(root, temp_phs);
          cout<<"\nph_time1 = "<<phase[temp_phs].ph_time;
          if (phase[temp_phs].num_pro_phase > num_site)
                {cout<<"\nthis is in the phase No. "<<temp_phs;
                 cout<<"\nthere are "<<phase[temp_phs].num_op_phase<<" operaions in
this phase";
                 time_equal(temp_phs);
                 cout<<"\nph_time2 = "<<phase[temp_phs].ph_time;  }}
//   cout<<"\nthe number of operations in this phase
="<<phase[temp_phs].num_op_phase;
   phase[temp_phs].nodeptr[phase[temp_phs].num_op_phase] = NULL;
//   cout<<"\nph_time = "<<phase[temp_phs].ph_time;
   }
num_phase = temp_phs;
ofquery<<"\nthe number of phases for this query is "<<num_phase;
}

/*************************************************************/
void time_equal(int temp_phs)          // time equalisation
{
int te_flag=1;      // flag to indicate are there enough processors
int i;
float y, max_y=-1.0;
int temp=num_site-phase[temp_phs].num_op_phase;
                          // the available number of processors
int ttemp=0;           // record the critcal operation

if (phase[temp_phs].num_op_phase<=1)
        { if (num_site<=phase[temp_phs].nodeptr[0]->op_num_pro)
        phase[temp_phs].ph_time=hash_join(num_site,
                  phase[temp_phs].nodeptr[0]->lnode->num_tuple,
                  phase[temp_phs].nodeptr[0]->rnode->num_tuple,
                  phase[temp_phs].nodeptr[0]->skew);
          else
phase[temp_phs].ph_time=hash_join(phase[temp_phs].nodeptr[0]->op_num_pro,
                  phase[temp_phs].nodeptr[0]->lnode->num_tuple,
                  phase[temp_phs].nodeptr[0]->rnode->num_tuple,
                  phase[temp_phs].nodeptr[0]->skew);
                  }
else {if (num_site>phase[temp_phs].num_op_phase)
        {
        for (i=0; i<phase[temp_phs].num_op_phase; i++)
        phase[temp_phs].nodeptr[i]->num_pro=1;
        }
else
        {
        te_flag=0;
        max_y=-1;
        for (i=0; i<num_site; i++)
                {
                y=hash_join(1,
                phase[temp_phs].nodeptr[i]->lnode->num_tuple,
                phase[temp_phs].nodeptr[i]->rnode->num_tuple,
                phase[temp_phs].nodeptr[i]->skew);
                if (y>max_y) max_y=y;
                }
```

```
        phase[temp_phs].ph_time=max_y;

        for (i=num_site; i<phase[temp_phs].num_op_phase; i++)
                {
                y=hash_join(num_site,
                phase[temp_phs].nodeptr[i]->lnode->num_tuple,
                phase[temp_phs].nodeptr[i]->rnode->num_tuple,
                phase[temp_phs].nodeptr[i]->skew);
                phase[temp_phs].ph_time += y;
                }   // find out the critical operation
        } /* end if else */

if (te_flag==1) {
        while (temp!=0) {
                ttemp=0;
                max_y=hash_join(phase[temp_phs].nodeptr[0]->num_pro,
                phase[temp_phs].nodeptr[0]->lnode->num_tuple,
                phase[temp_phs].nodeptr[0]->rnode->num_tuple,
                phase[temp_phs].nodeptr[0]->skew);
                for (i=1; i<phase[temp_phs].num_op_phase; i++)
                 {
                 y=hash_join(phase[temp_phs].nodeptr[i]->num_pro,
                  phase[temp_phs].nodeptr[i]->lnode->num_tuple,
                  phase[temp_phs].nodeptr[i]->rnode->num_tuple,
                  phase[temp_phs].nodeptr[i]->skew);
                 if (max_y<y)
                        { max_y=y; ttemp=i; }
                 }   // find out the critical operation
                phase[temp_phs].nodeptr[ttemp]->num_pro++;
                if (phase[temp_phs].nodeptr[ttemp]->num_pro >=
                   phase[temp_phs].nodeptr[ttemp]->op_num_pro)
                   break;
                temp--;
                }     /* end while */
phase[temp_phs].ph_time=max_y;
} /* end if */
    }
cout<<"\nphase time is "<<phase[temp_phs].ph_time;
if (te_flag) {
        phase[temp_phs].num_pro_phase=0;
        for (i=0; i<phase[temp_phs].num_op_phase; i++)  // display results
                {//cout<<"\nThe "<<phase[temp_phs].nodeptr[i]->pid<<"th operation
has "<< //phase[temp_phs].nodeptr[i]->num_pro<<" processors";
                phase[temp_phs].num_pro_phase += phase[temp_phs].nodeptr[i]-
>num_pro;
                }
        }
else phase[temp_phs].num_pro_phase=num_site;
}

/*************************************************************/
void find_ready_op(NODE *p, int phs)      // find all ready operations
{
//if(phase[phs].num_op >= num_site) return 0;       /* --- no further search */
if(p !=NULL && p->allocated==0)
        {
        if(((p->lnode->allocated==1 && p->rnode==NULL)
         || (p->lnode->allocated==1 && p->rnode->allocated==1))
                && (p->type>0))
```

```
                {
                  p = optimize(p, p->lnode->num_tuple,
                  p->rnode->num_tuple, p->skew);
                  phase[phs].num_pro_phase += p->op_num_pro; // each phase
                  if (p->op_time > phase[phs].ph_time)
                          phase[phs].ph_time = p->op_time;    // each phase
                  phase[phs].nodeptr[phase[phs].num_op_phase] = p;
                  p->allocated=-1;          // temp marking
                  phase[phs].num_op_phase++;
                }
        else    {
                find_ready_op(p->lnode,phs);
                find_ready_op(p->rnode,phs);
                }
        }
}

/*************************************************************/
void merge_point(NODE *p, int phs)
{
if(p !=NULL && (p->allocated==0 || p->allocated==-1) )
{
if (((p->lnode->allocated==1 && p->rnode==NULL)
|| (p->lnode->allocated==1 && p->rnode->allocated==1)) && (p->type>0))
        {
          if ((p->pnode->rnode->allocated==-1 &&
             p->pnode->lnode->allocated==-1) ||
             (p->pnode->rnode->allocated==-1 &&
             p->pnode->lnode->allocated==1) ||
             (p->pnode->rnode->allocated==1 &&
             p->pnode->lnode->allocated==-1))
//      || (p->pnode->lnode==NULL) || (p->pnode->rnode==NULL))
                {
                p=optimize(p, p->lnode->num_tuple,
                p->rnode->num_tuple, p->skew);
                phase[phs].num_pro_phase += p->op_num_pro; // each phase
                if (p->op_time > phase[phs].ph_time)
                  phase[phs].ph_time = p->op_time;       // each phase
                phase[phs].nodeptr[phase[phs].num_op_phase] = p;
                phase[phs].num_op_phase++;
                }
          else
                p->allocated=0;
        }
else    {
        merge_point(p->lnode,phs);
        merge_point(p->rnode,phs);
        }
}
}

/*************************************************************/
NODE *get_query(void) // read in queries from input files and
{                             // store them as a binary tree -- return root
NODE *rel_in[MAX_NODE], *op_in[MAX_NODE], *qroot;
int oplink[MAX_NODE][4];
int i, num_rel=6;

   ifquery>>num_rel>>num_op;
```

```
    for(i=0;i<num_rel;i++)                    // read in relations
        {
        rel_in[i]=(NODE *)malloc(sizeof(NODE));
        ifquery>>rel_in[i]->pid>>rel_in[i]->type>>rel_in[i]->type
        >>rel_in[i]->skew>>rel_in[i]->num_tuple;
        rel_in[i]->lnode=NULL;
        rel_in[i]->rnode=NULL;
        rel_in[i]->num_pro=0;
        rel_in[i]->ttime=0.0;
        rel_in[i]->allocated=1;
        }

    for(i=0;i<num_op;i++)          // read in operations of the query
        {                          // query is stored as post tree traversal
        op_in[i]=(NODE *)malloc(sizeof(NODE));
        ifquery>>op_in[i]->pid>>op_in[i]->type>>op_in[i]->skew>>
        op_in[i]->num_tuple>>oplink[i][0]>>oplink[i][1]>>
        oplink[i][2]>>oplink[i][3];
        op_in[i]->ttime =0.0;
        op_in[i]->allocated=0;
        }

    qroot=op_in[num_op-1];     // set up the binary tree
    for(i=0;i<num_op;i++)
        {
        if(oplink[i][0]==0)          {
                                     op_in[i]->lnode=rel_in[oplink[i][1]];
                                     rel_in[oplink[i][1]]->pnode=op_in[i];
                                     }
        else      {
                  op_in[i]->lnode=op_in[oplink[i][1]-1];
                  op_in[oplink[i][1]-1]->pnode=op_in[i];
                  }
        if (op_in[i]->type==1)
                  {
                  if(oplink[i][2]==0)          {
                                               op_in[i]->rnode=rel_in[oplink[i][3]];
                                               rel_in[oplink[i][3]]->pnode=op_in[i];
                                               }
                  else      {
                            op_in[i]->rnode=op_in[oplink[i][3]-1];
                            op_in[oplink[i][3]-1]->pnode=op_in[i];
                            }
                  }
        else
                  op_in[i]->rnode=NULL;
        }
return(qroot);
}

/*************************************************************/
void print_tree(NODE *p)   // travel the binary tree and print out data
{
if (p!=NULL) {
        print_tree(p->lnode);
        if (p->type>0)
                  {
                  ofquery<<"\npid="<<p->pid+100<<", num_tuple="<<p->num_tuple
                  <<", skew="<<p->skew<<", type"<<p->type<<", op-time="
```

```
                <<p->op_time<<", op-num-pro"<<p->op_num_pro;
                if (p->lnode->type>0)
                        ofquery<<", lnode="<<p->lnode->pid+100;
                else
                        ofquery<<", lnode="<<p->lnode->pid;
                if (p->type==1)
                        {
                        if (p->rnode->type>0)
                                ofquery<<", rnode="<<p->rnode->pid+100;
                        else
                                ofquery<<", rnode="<<p->rnode->pid;
                        print_tree(p->rnode);
                        }
                else
                        ofquery<<"rnode = NULL\n";
                }
        else
                ofquery<<"\npid="<<p->pid<<", num_tuple="<<p->num_tuple
                <<", skew="<<p->skew<<", type"<<p->type;
        }
}

/*************************************************************/
NODE *optimize(NODE *q, int rz, int sz, float skew)
{               // find out the optimal number of processors for each operation
long n=1;
int flag=1;
float miny=hash_join(n, rz, sz, skew);
float y;

//cout<<"\n"<<miny<<", \t\t"<<n;
n++;
do   {
        y=hash_join(n, rz, sz, skew);
//      cout<<"\n"<<y<<", \t"<<n;
        if (y<miny)
                {
                miny=y;
                n++;
                }
        else flag=0;
        }
while (flag && (n<num_site));
q->op_num_pro=n;
q->op_time=miny;
return q;
}

/*************************************************************/
float hash_join(long n, int rz, int sz, float skew)
{
float tt;
float skew_f;           // work out the execution time
if (n<1)
        {
        cout<<"\problem with the number of pro";
        return 0;
        }
if (skew==0)
```

```
        skew_f=n;
else
//      skew_f=0.57721+log(n);
        skew_f=Harmo(n, skew);                  // based on the cost model
if (skew_f==0)
        cout<<"\nproblem is here with skew";
tt=(thash+(n-1)*tdata+w1+w2)*(rz+sz)/skew_f;
tt=tt+tinit*(n+1)+w3*rz/skew_f/(rz+sz)*sz;
return tt;
}

/*********************************************************/
float Harmo (int k, float skew)     // find out the Harmonic number of k
{
float sumH=0.0, c=0.0;
int i;
for (i=0; i<k; i++)
        {
        c=c+1.0;
        sumH=sumH+1.0/pow(c,skew);
        }
return(sumH);
}
```

Input File: IFQUERY.DAT

```
9 1 32
6 5
0 0 0.0 1000
1 0 0.0 2000
2 0 0.0 3000
3 0 0.0 4000
4 0 0.0 5000
5 0 0.0 6000
1 1 0.0 2000 0 0 0 1
2 1 0.0 4000 0 2 0 3
3 1 1.0 4000 1 1 1 2
4 1 0.0 6000 0 4 0 5
5 1 0.0 6000 1 3 1 4
10 9
0 0 0.0 1000
1 0 0.0 2000
2 0 0.0 3000
3 0 0.0 4000
4 0 0.0 5000
5 0 0.0 6000
6 0 0.0 7000
7 0 0.0 8000
8 0 0.0 9000
9 0 0.0 10000
1 1 0.0 10000 0 0 0 1
2 1 0.0 10000 0 2 0 3
3 1 0.0 10000 0 4 0 5
4 1 0.05 10000 1 2 1 3
5 1 0.0 10000 0 6 0 7
6 1 0.05 10000 1 4 1 5
7 1 0.05 10000 1 1 1 6
```

```
8 1 0.0 10000 0 8 0 9
9 1 0.0 10000 1 7 1 8
10 9
0 0 0.0 1000
1 0 0.0 2000
2 0 0.0 3000
3 0 0.0 4000
4 0 0.0 5000
5 0 0.0 6000
6 0 0.0 7000
7 0 0.0 8000
8 0 0.0 9000
9 0 0.0 10000
1 1 0.0 2000 0 0 0 1
2 1 0.0 4000 0 2 0 3
3 1 0.0 6000 0 4 0 5
4 1 0.0 8000 0 6 0 7
5 1 0.0 10000 0 8 0 9
6 1 0.05 10000 1 4 1 5
7 1 0.05 10000 1 3 1 6
8 1 0.05 10000 1 2 1 7
9 1 0.0 10000 1 1 1 8
10 9
0 0 0.0 1000
1 0 0.0 2000
2 0 0.0 3000
3 0 0.0 4000
4 0 0.0 5000
5 0 0.0 6000
6 0 0.0 7000
7 0 0.0 8000
8 0 0.0 9000
9 0 0.0 10000
1 1 0.0 10000 0 0 0 1
2 1 0.0 10000 0 2 0 3
3 1 0.05 10000 1 1 1 2
4 1 0.0 10000 0 4 0 5
5 1 0.05 10000 1 3 1 4
6 1 0.0 10000 0 6 0 7
7 1 0.05 10000 1 5 1 6
8 1 0.0 10000 0 8 0 9
9 1 0.0 10000 1 7 1 8
10 9
0 0 0.0 1000
1 0 0.0 2000
2 0 0.0 3000
3 0 0.0 4000
4 0 0.0 5000
5 0 0.0 6000
6 0 0.0 7000
7 0 0.0 8000
8 0 0.0 9000
9 0 0.0 10000
1 1 0.0 10000 0 0 0 1
2 1 0.0 10000 0 2 0 3
3 1 0.05 10000 1 1 1 2
4 1 0.0 10000 0 4 0 5
5 1 0.05 10000 1 3 1 4
6 1 0.0 10000 0 6 0 7
```

```
7  1  0.0  10000  0  8  0  9
8  1  0.0  10000  1  6  1  7
9  1  0.0  10000  1  5  1  8
10  9
0  0  0.0  1000
1  0  0.0  2000
2  0  0.0  3000
3  0  0.0  4000
4  0  0.0  5000
5  0  0.0  6000
6  0  0.0  7000
7  0  0.0  8000
8  0  0.0  9000
9  0  0.0  10000
1  1  0.0  10000  0  0  0  1
2  1  0.0  10000  0  2  0  3
3  1  0.05  10000  1  1  1  2
4  1  0.0  10000  0  4  0  5
5  1  0.05  10000  1  3  1  4
6  1  0.0  10000  0  6  0  7
7  1  0.0  10000 0  8  0  9
8  1  0.0  10000 1  6  1  7
9  1  0.0  10000 1  5  1  8
10  9
0  0  0.0  1000
1  0  0.0  2000
2  0  0.0  3000
3  0  0.0  4000
4  0  0.0  5000
5  0  0.0  6000
6  0  0.0  7000
7  0  0.0  8000
8  0  0.0  9000
9  0  0.0  10000
1  1  0.0  10000  0  0  0  1
2  1  0.0  10000  1  1  0  2
3  1  0.0  10000  0  3  0  4
4  1  0.05  10000  1  2  1  3
5  1  0.0  10000  0  5  0  6
6  1  0.05  10000  1  5  0  7
7  1  0.05  10000 1  4  1  6
8  1  0.0  10000 0  8  0  9
9  1  0.0  10000 1  7  1  8
12  11
0  0  0.0  1000
1  0  0.0  2000
2  0  0.0  3000
3  0  0.0  4000
4  0  0.0  5000
5  0  0.0  6000
6  0  0.0  7000
7  0  0.0  8000
8  0  0.0  9000
9  0  0.0  10000
10 0  0.0  10000
11 0  0.0  10000
1  1  0.0  10000  0  0  0  1
2  1  0.0  10000  0  2  0  3
3  1  0.05  10000  1  1  1  2
```

```
4  1  0.0  10000  0  4  0  5
5  1  0.05  10000  1  3  1  4
6  1  0.0  10000  0  6  0  7
7  1  0.0  10000  0  8  0  9
8  1  0.05  10000  1  6  1  7
9  1  0.0  10000  0  10 0  11
10 1  0.0  10000  1  8  1  9
11 1  0.0  10000  1  5  1  10
12  11
0  0  0.0  1000
1  0  0.0  2000
2  0  0.0  3000
3  0  0.0  4000
4  0  0.0  5000
5  0  0.0  6000
6  0  0.0  7000
7  0  0.0  8000
8  0  0.0  9000
9  0  0.0  10000
10 0  0.0  10000
11 0  0.0  10000
1  1  0.0  10000  0  0  0  1
2  1  0.05  10000  1  1  0  2
3  1  0.0  10000  0  3  0  4
4  1  0.0  10000  1  2  1  3
5  1  0.0  10000  0  5  0  6
6  1  0.0  10000  1  5  0  7
7  1  0.0  10000 0  8  0  9
8  1  0.05  10000 1  6  1  7
9  1  0.05  10000 1  4  1  8
10 1  0.0  10000 0  10  0 11
11 1  0.0  10000 1  9  1  10
```

Output File: OFQUERY.DAT

New Optimised Algorithm

```
%%%  This is the beginning of the simulation program
num_query=9, num_qtype=1

New Query
the number of phases for this query is 3
pid=0, num_tuple=1000, skew=0, type0
pid=101, num_tuple=2000, skew=0, type1, op-time=28.405186, op-num-pro18, lnode=0,
rnode=1
pid=1, num_tuple=2000, skew=0, type0
pid=103, num_tuple=4000, skew=1, type1, op-time=95.91362, op-num-pro32, lnode=101,
rnode=102
pid=2, num_tuple=3000, skew=0, type0
pid=102, num_tuple=4000, skew=0, type1, op-time=35.620136, op-num-pro32, lnode=2,
rnode=3
pid=3, num_tuple=4000, skew=0, type0
pid=105, num_tuple=6000, skew=0, type1, op-time=40.249237, op-num-pro32, lnode=103,
rnode=104
pid=4, num_tuple=5000, skew=0, type0
pid=104, num_tuple=6000, skew=0, type1, op-time=41.83366, op-num-pro32, lnode=4,
rnode=5
```

pid=5, num_tuple=6000, skew=0, type0
phase[0].num_op=2, ph_time=36.232773, num_pro=32, pid=1, pid=2
phase[1].num_op=2, ph_time=96.837158, num_pro=32, pid=3, pid=4
phase[2].num_op=1, ph_time=40.249237, num_pro=32, pid=5
The execution time for this query is 173.319168

New Query
the number of phases for this query is 5
pid=0, num_tuple=1000, skew=0, type0
pid=101, num_tuple=10000, skew=0, type1, op-time=28.405186, op-num-pro18, lnode=0, rnode=1
pid=1, num_tuple=2000, skew=0, type0
pid=107, num_tuple=10000, skew=0.05, type1, op-time=59.946423, op-num-pro32, lnode=101, rnode=106
pid=2, num_tuple=3000, skew=0, type0
pid=102, num_tuple=10000, skew=0, type1, op-time=35.620136, op-num-pro32, lnode=2, rnode=3
pid=3, num_tuple=4000, skew=0, type0
pid=104, num_tuple=10000, skew=0.05, type1, op-time=59.946423, op-num-pro32, lnode=102, rnode=103
pid=4, num_tuple=5000, skew=0, type0
pid=103, num_tuple=10000, skew=0, type1, op-time=41.83366, op-num-pro32, lnode=4, rnode=5
pid=5, num_tuple=6000, skew=0, type0
pid=106, num_tuple=10000, skew=0.05, type1, op-time=59.946423, op-num-pro32, lnode=104, rnode=105
pid=6, num_tuple=7000, skew=0, type0
pid=105, num_tuple=10000, skew=0, type1, op-time=48.044277, op-num-pro32, lnode=6, rnode=7
pid=7, num_tuple=8000, skew=0, type0
pid=109, num_tuple=10000, skew=0, type1, op-time=55.811359, op-num-pro32, lnode=107, rnode=108
pid=8, num_tuple=9000, skew=0, type0
pid=108, num_tuple=10000, skew=0, type1, op-time=54.253826, op-num-pro32, lnode=8, rnode=9
pid=9, num_tuple=10000, skew=0, type0
phase[0].num_op=2, ph_time=46.814262, num_pro=32, pid=2, pid=3
phase[1].num_op=2, ph_time=74.892899, num_pro=32, pid=4, pid=5
phase[2].num_op=2, ph_time=61.550564, num_pro=32, pid=1, pid=6
phase[3].num_op=2, ph_time=79.650932, num_pro=32, pid=7, pid=8
phase[4].num_op=1, ph_time=55.811359, num_pro=32, pid=9
The execution time for this query is 318.720032

New Query
the number of phases for this query is 4
pid=0, num_tuple=1000, skew=0, type0
pid=101, num_tuple=2000, skew=0, type1, op-time=28.405186, op-num-pro18, lnode=0, rnode=1
pid=1, num_tuple=2000, skew=0, type0
pid=109, num_tuple=10000, skew=0, type1, op-time=42.836071, op-num-pro32, lnode=101, rnode=108
pid=2, num_tuple=3000, skew=0, type0
pid=102, num_tuple=4000, skew=0, type1, op-time=35.620136, op-num-pro32, lnode=2, rnode=3
pid=3, num_tuple=4000, skew=0, type0
pid=108, num_tuple=10000, skew=0.05, type1, op-time=49.088291, op-num-pro32, lnode=102, rnode=107
pid=4, num_tuple=5000, skew=0, type0

pid=103, num_tuple=6000, skew=0, type1, op-time=41.83366, op-num-pro32, lnode=4, rnode=5
pid=5, num_tuple=6000, skew=0, type0
pid=107, num_tuple=10000, skew=0.05, type1, op-time=52.79253, op-num-pro32, lnode=103, rnode=106
pid=6, num_tuple=7000, skew=0, type0
pid=104, num_tuple=8000, skew=0, type1, op-time=48.044277, op-num-pro32, lnode=6, rnode=7
pid=7, num_tuple=8000, skew=0, type0
pid=106, num_tuple=10000, skew=0.05, type1, op-time=56.402477, op-num-pro32, lnode=104, rnode=105
pid=8, num_tuple=9000, skew=0, type0
pid=105, num_tuple=10000, skew=0, type1, op-time=54.253826, op-num-pro32, lnode=8, rnode=9
pid=9, num_tuple=10000, skew=0, type0
phase[0].num_op=2, ph_time=69.660133, num_pro=32, pid=4, pid=5
phase[1].num_op=2, ph_time=66.222633, num_pro=32, pid=3, pid=6
phase[2].num_op=2, ph_time=57.868973, num_pro=32, pid=2, pid=7
phase[3].num_op=2, ph_time=50.584797, num_pro=32, pid=1, pid=8
The execution time for this query is 244.336533

New Query
the number of phases for this query is 5
pid=0, num_tuple=1000, skew=0, type0
pid=101, num_tuple=10000, skew=0, type1, op-time=28.405186, op-num-pro18, lnode=0, rnode=1
pid=1, num_tuple=2000, skew=0, type0
pid=103, num_tuple=10000, skew=0.05, type1, op-time=59.946423, op-num-pro32, lnode=101, rnode=102
pid=2, num_tuple=3000, skew=0, type0
pid=102, num_tuple=10000, skew=0, type1, op-time=35.620136, op-num-pro32, lnode=2, rnode=3
pid=3, num_tuple=4000, skew=0, type0
pid=105, num_tuple=10000, skew=0.05, type1, op-time=59.946423, op-num-pro32, lnode=103, rnode=104
pid=4, num_tuple=5000, skew=0, type0
pid=104, num_tuple=10000, skew=0, type1, op-time=41.83366, op-num-pro32, lnode=4, rnode=5
pid=5, num_tuple=6000, skew=0, type0
pid=107, num_tuple=10000, skew=0.05, type1, op-time=59.946423, op-num-pro32, lnode=105, rnode=106
pid=6, num_tuple=7000, skew=0, type0
pid=106, num_tuple=10000, skew=0, type1, op-time=48.044277, op-num-pro32, lnode=6, rnode=7
pid=7, num_tuple=8000, skew=0, type0
pid=109, num_tuple=10000, skew=0, type1, op-time=55.811359, op-num-pro32, lnode=107, rnode=108
pid=8, num_tuple=9000, skew=0, type0
pid=108, num_tuple=10000, skew=0, type1, op-time=54.253826, op-num-pro32, lnode=8, rnode=9
pid=9, num_tuple=10000, skew=0, type0
phase[0].num_op=2, ph_time=36.232773, num_pro=32, pid=1, pid=2
phase[1].num_op=2, ph_time=69.584389, num_pro=32, pid=3, pid=4
phase[2].num_op=2, ph_time=74.892899, num_pro=32, pid=5, pid=6
phase[3].num_op=2, ph_time=79.650932, num_pro=32, pid=7, pid=8
phase[4].num_op=1, ph_time=55.811359, num_pro=32, pid=9
The execution time for this query is 316.172363

New Query

the number of phases for this query is 3
pid=0, num_tuple=1000, skew=0, type0
pid=101, num_tuple=10000, skew=0, type1, op-time=28.405186, op-num-pro18, lnode=0, rnode=1
pid=1, num_tuple=2000, skew=0, type0
pid=103, num_tuple=10000, skew=0.05, type1, op-time=59.946423, op-num-pro32, lnode=101, rnode=102
pid=2, num_tuple=3000, skew=0, type0
pid=102, num_tuple=10000, skew=0, type1, op-time=35.620136, op-num-pro32, lnode=2, rnode=3
pid=3, num_tuple=4000, skew=0, type0
pid=105, num_tuple=10000, skew=0.05, type1, op-time=59.946423, op-num-pro32, lnode=103, rnode=104
pid=4, num_tuple=5000, skew=0, type0
pid=104, num_tuple=10000, skew=0, type1, op-time=41.83366, op-num-pro32, lnode=4, rnode=5
pid=5, num_tuple=6000, skew=0, type0
pid=109, num_tuple=10000, skew=0, type1, op-time=55.811359, op-num-pro32, lnode=105, rnode=108
pid=6, num_tuple=7000, skew=0, type0
pid=106, num_tuple=10000, skew=0, type1, op-time=48.044277, op-num-pro32, lnode=6, rnode=7
pid=7, num_tuple=8000, skew=0, type0
pid=108, num_tuple=10000, skew=0, type1, op-time=55.811359, op-num-pro32, lnode=106, rnode=107
pid=8, num_tuple=9000, skew=0, type0
pid=107, num_tuple=10000, skew=0, type1, op-time=54.253826, op-num-pro32, lnode=8, rnode=9
pid=9, num_tuple=10000, skew=0, type0
phase[0].num_op=4, ph_time=79.650932, num_pro=32, pid=1, pid=2, pid=6, pid=7
phase[1].num_op=2, ph_time=69.584389, num_pro=32, pid=3, pid=4
phase[2].num_op=2, ph_time=82.206299, num_pro=32, pid=5, pid=8
The execution time for this query is 231.44162

New Query
the number of phases for this query is 4
pid=0, num_tuple=1000, skew=0, type0
pid=101, num_tuple=10000, skew=0, type1, op-time=28.405186, op-num-pro18, lnode=0, rnode=1
pid=1, num_tuple=2000, skew=0, type0
pid=103, num_tuple=10000, skew=0.05, type1, op-time=59.946423, op-num-pro32, lnode=101, rnode=102
pid=2, num_tuple=3000, skew=0, type0
pid=102, num_tuple=10000, skew=0, type1, op-time=35.620136, op-num-pro32, lnode=2, rnode=3
pid=3, num_tuple=4000, skew=0, type0
pid=105, num_tuple=10000, skew=0.05, type1, op-time=59.946423, op-num-pro32, lnode=103, rnode=104
pid=4, num_tuple=5000, skew=0, type0
pid=104, num_tuple=10000, skew=0, type1, op-time=41.83366, op-num-pro32, lnode=4, rnode=5
pid=5, num_tuple=6000, skew=0, type0
pid=109, num_tuple=10000, skew=0, type1, op-time=55.811359, op-num-pro32, lnode=105, rnode=108
pid=6, num_tuple=7000, skew=0, type0
pid=106, num_tuple=10000, skew=0, type1, op-time=48.044277, op-num-pro32, lnode=6, rnode=7
pid=7, num_tuple=8000, skew=0, type0

pid=108, num_tuple=10000, skew=0, type1, op-time=55.811359, op-num-pro32, lnode=106, rnode=107
pid=8, num_tuple=9000, skew=0, type0
pid=107, num_tuple=10000, skew=0, type1, op-time=54.253826, op-num-pro32, lnode=8, rnode=9
pid=9, num_tuple=10000, skew=0, type0
phase[0].num_op=4, ph_time=79.650932, num_pro=32, pid=1, pid=2, pid=6, pid=7
phase[1].num_op=2, ph_time=69.584389, num_pro=32, pid=3, pid=4
phase[2].num_op=2, ph_time=82.206299, num_pro=32, pid=5, pid=8
phase[3].num_op=1, ph_time=55.811359, num_pro=32, pid=9
The execution time for this query is 287.252991

New Query
the number of phases for this query is 4
pid=0, num_tuple=1000, skew=0, type0
pid=101, num_tuple=10000, skew=0, type1, op-time=28.405186, op-num-pro18, lnode=0, rnode=1
pid=1, num_tuple=2000, skew=0, type0
pid=102, num_tuple=10000, skew=0, type1, op-time=44.552067, op-num-pro32, lnode=101, rnode=2
pid=2, num_tuple=3000, skew=0, type0
pid=104, num_tuple=10000, skew=0.05, type1, op-time=59.946423, op-num-pro32, lnode=102, rnode=103
pid=3, num_tuple=4000, skew=0, type0
pid=103, num_tuple=10000, skew=0, type1, op-time=38.727505, op-num-pro32, lnode=3, rnode=4
pid=4, num_tuple=5000, skew=0, type0
pid=107, num_tuple=10000, skew=0.05, type1, op-time=59.946423, op-num-pro32, lnode=104, rnode=106
pid=5, num_tuple=6000, skew=0, type0
pid=105, num_tuple=10000, skew=0, type1, op-time=44.939163, op-num-pro32, lnode=5, rnode=6
pid=6, num_tuple=7000, skew=0, type0
pid=106, num_tuple=10000, skew=0.05, type1, op-time=56.402477, op-num-pro32, lnode=105, rnode=7
pid=7, num_tuple=8000, skew=0, type0
pid=109, num_tuple=10000, skew=0, type1, op-time=55.811359, op-num-pro32, lnode=107, rnode=108
pid=8, num_tuple=9000, skew=0, type0
pid=108, num_tuple=10000, skew=0, type1, op-time=54.253826, op-num-pro32, lnode=8, rnode=9
pid=9, num_tuple=10000, skew=0, type0
phase[0].num_op=2, ph_time=46.161304, num_pro=32, pid=1, pid=5
phase[1].num_op=2, ph_time=52.064331, num_pro=32, pid=2, pid=3
phase[2].num_op=2, ph_time=81.778625, num_pro=32, pid=4, pid=6
phase[3].num_op=2, ph_time=79.650932, num_pro=32, pid=7, pid=8
The execution time for this query is 259.655182

New Query
the number of phases for this query is 3
pid=0, num_tuple=1000, skew=0, type0
pid=101, num_tuple=10000, skew=0, type1, op-time=28.405186, op-num-pro18, lnode=0, rnode=1
pid=1, num_tuple=2000, skew=0, type0
pid=103, num_tuple=10000, skew=0.05, type1, op-time=59.946423, op-num-pro32, lnode=101, rnode=102
pid=2, num_tuple=3000, skew=0, type0
pid=102, num_tuple=10000, skew=0, type1, op-time=35.620136, op-num-pro32, lnode=2, rnode=3

pid=3, num_tuple=4000, skew=0, type0
pid=105, num_tuple=10000, skew=0.05, type1, op-time=59.946423, op-num-pro32, lnode=103, rnode=104
pid=4, num_tuple=5000, skew=0, type0
pid=104, num_tuple=10000, skew=0, type1, op-time=41.83366, op-num-pro32, lnode=4, rnode=5
pid=5, num_tuple=6000, skew=0, type0
pid=111, num_tuple=10000, skew=0, type1, op-time=55.811359, op-num-pro32, lnode=105, rnode=110
pid=6, num_tuple=7000, skew=0, type0
pid=106, num_tuple=10000, skew=0, type1, op-time=48.044277, op-num-pro32, lnode=6, rnode=7
pid=7, num_tuple=8000, skew=0, type0
pid=108, num_tuple=10000, skew=0.05, type1, op-time=59.946423, op-num-pro32, lnode=106, rnode=107
pid=8, num_tuple=9000, skew=0, type0
pid=107, num_tuple=10000, skew=0, type1, op-time=54.253826, op-num-pro32, lnode=8, rnode=9
pid=9, num_tuple=10000, skew=0, type0
pid=110, num_tuple=10000, skew=0, type1, op-time=55.811359, op-num-pro32, lnode=108, rnode=109
pid=10, num_tuple=10000, skew=0, type0
pid=109, num_tuple=10000, skew=0, type1, op-time=55.811359, op-num-pro32, lnode=10, rnode=11
pid=11, num_tuple=10000, skew=0, type0
phase[0].num_op=4, ph_time=79.650932, num_pro=32, pid=1, pid=2, pid=6, pid=7
phase[1].num_op=4, ph_time=117.86364, num_pro=32, pid=3, pid=4, pid=8, pid=9
phase[2].num_op=2, ph_time=82.206299, num_pro=32, pid=5, pid=10
The execution time for this query is 279.720886

New Query
the number of phases for this query is 5
pid=0, num_tuple=1000, skew=0, type0
pid=101, num_tuple=10000, skew=0, type1, op-time=28.405186, op-num-pro18, lnode=0, rnode=1
pid=1, num_tuple=2000, skew=0, type0
pid=102, num_tuple=10000, skew=0.05, type1, op-time=47.187218, op-num-pro32, lnode=101, rnode=2
pid=2, num_tuple=3000, skew=0, type0
pid=104, num_tuple=10000, skew=0, type1, op-time=55.811359, op-num-pro32, lnode=102, rnode=103
pid=3, num_tuple=4000, skew=0, type0
pid=103, num_tuple=10000, skew=0, type1, op-time=38.727505, op-num-pro32, lnode=3, rnode=4
pid=4, num_tuple=5000, skew=0, type0
pid=109, num_tuple=10000, skew=0.05, type1, op-time=59.946423, op-num-pro32, lnode=104, rnode=108
pid=5, num_tuple=6000, skew=0, type0
pid=105, num_tuple=10000, skew=0, type1, op-time=44.939163, op-num-pro32, lnode=5, rnode=6
pid=6, num_tuple=7000, skew=0, type0
pid=106, num_tuple=10000, skew=0, type1, op-time=52.684025, op-num-pro32, lnode=105, rnode=7
pid=7, num_tuple=8000, skew=0, type0
pid=108, num_tuple=10000, skew=0.05, type1, op-time=59.946423, op-num-pro32, lnode=106, rnode=107
pid=8, num_tuple=9000, skew=0, type0
pid=107, num_tuple=10000, skew=0, type1, op-time=54.253826, op-num-pro32, lnode=8, rnode=9

pid=9, num_tuple=10000, skew=0, type0
pid=111, num_tuple=10000, skew=0, type1, op-time=55.811359, op-num-pro32, lnode=109, rnode=110
pid=10, num_tuple=10000, skew=0, type0
pid=110, num_tuple=10000, skew=0, type1, op-time=55.811359, op-num-pro32, lnode=10, rnode=11
pid=11, num_tuple=10000, skew=0, type0
phase[0].num_op=2, ph_time=46.161304, num_pro=32, pid=1, pid=5
phase[1].num_op=4, ph_time=98.702522, num_pro=32, pid=2, pid=3, pid=6, pid=7
phase[2].num_op=2, ph_time=82.206299, num_pro=32, pid=4, pid=8
phase[3].num_op=2, ph_time=82.206299, num_pro=32, pid=9, pid=10
phase[4].num_op=1, ph_time=55.811359, num_pro=32, pid=11
The execution time for this query is 365.087799
%%% This is the end of the Simulation program