Parallel Query Processing Using Shared Memory Multiprocessors and Disk Arrays

Copyright © 1992

by

Wei Hong

Parallel Query Processing Using Shared Memory Multiprocessors and Disk Arrays

Wei Hong

Computer Science Division

Department of Electrical Engineering and Computer Science

University of California

Berkeley, CA 94720

August 1992

A dissertation

submitted in partial satisfaction of

the requirements for the degree of

Doctor of Philosophy in Computer Science in

the Graduate Division of

the University of California, Berkeley.

To Nanyan.

Acknowledgments

My deep gratitude first goes to my research advisor, Professor Michael Stonebraker, for his guidance, support and encouragement throughout my research. His inexhaustible ideas and insights have been of invaluable help to me. He has taught me to catch the essence in a seemingly bewildering issue and to develop a good taste in research.

I worked with Professor Eugene Wong during my first two years of study in Berkeley. I benefited immersely from his judicious advising and his rigorous research style. I am very grateful to him.

I would like to thank the members of my thesis committee, Professors Randy Katz and Arie Segev, for taking the time to read my thesis and providing me with several suggestions for improvement.

I have enjoyed working with all the other members of the XPRS and Postgres research group, in particular, Mark Sullivan, Margo Seltzer, Mike Olson, Spyros Potamianos, Jeff Meredith, Joe Hellerstein, Jolly Chen and Chandra Ghosh. I cherish the time that I spent with them arguing over design decisions and agonizing over the "last" bug in our system. I am grateful for all the help that they have given me in my research. I will also remember that it was from them that I learned how to appreciate a good beer and enjoy a good party.

I would like to thank my fellow students Yongdong Wang and Chuen-tsai Sun for their valuable friendship and for all their help. I also would like to thank Guangrui Zhu and Yan Wei for being two special friends and making my life more interesting. Many thanks also go to my college friends Yuzheng Ding and Jiyang Liu. Our communications have always been an inspiring source in my life.

Although my parents and my sister are an ocean away, they have offered me their constant love and encouragement throughout my study. I would like to take this opportunity to thank them for everything they have done for me.

Last, but the most, I would like to thank my dear wife, Nanyan Xiong. Without her love, understanding and support throughout my Ph.D. program, this thesis would not have been possible. This thesis is dedicated to her as a small token of my deep appreciation.

Contents

List of Figures

Chapter 1

Introduction

The trend in database applications is that databases are becoming orders of magnitude larger and user queries are becoming more and more complex. This trend is driven by new application areas such as decision support, multi-media applications, scientific data visualization, and information retrieval. For example, NASA scientists have been collecting terabytes of satellite image data from space for many years, and they wish to run various queries over all the past and current data to find relevant images for their research. As another example, several department stores have started to record every product-code-scanning action of every cashier in every store in their chain. Ad-hoc complex queries are run on this historical database to discover buying patterns and make stocking decisions.

It has become increasingly difficult for conventional single processor computer systems to meet the CPU and I/O demands of relational DBMS searching terabyte databases or processing complex queries. Meanwhile, multiprocessors based on increasingly fast and inexpensive microprocessors have become widely available from a variety of vendors including Sequent, Tandem, Intel, Teradata, and nCUBE. These machines provide not only more total computing power than their mainframe counterparts, but also provide a lower price/MIPS. Moreover, the disk array technology that provides high bandwidth and high availability through redundant arrays of inexpensive disks [37] has emerged to ease the I/O bottleneck problem. Because relational queries consist of uniform operations applied to uniform streams of data, they are ideally suited to parallel execution. Therefore, the way to meet the high CPU and I/O demands of these new database applications is to build a parallel database system based on a large number of inexpensive processors and disks exploiting parallelism within as well as between queries.

In this chapter, we first introduce the issues in query processing on parallel database systems that will be addressed in this thesis. Then, related previous work on parallel database systems, especially work on parallel query processing is surveyed. The last section of this chapter presents an outline of the rest of this thesis.

1.1 Query Processing in Parallel Database Systems

One of the fundamental innovations of relational databases is their non-procedural query languages based on predicate calculus. In earlier database systems, namely those based on hierarchical and network data models, the application program must navigate through the database via links and pointers between data records. In a relational database system, a user only specifies the predicates that the retrieved data should satisfy in a relational query language such as SQL [26], and the database system determines the necessary processing steps, i.e., a *query plan* automatically. Since there may be many possible query plans which differ by orders of magnitude in processing costs (see [27] for an example), the key of database query processing is to find the cheapest and fastest query plan.

1.1.1 Conventional Query Processing

Conventional query processing assumes a uniprocessor environment and query plans are executed sequentially. A query plan for a uniprocessor environment is called a *sequential plan*. The common approach to optimization of sequential plans is to exhaustively or semi-exhaustively search through all the possible query plans, estimate a cost for each plan, and choose the one with minimum cost, as described in [47]. A sequential query plan is a binary tree of the basic relational operations, i.e., *scans* and *joins*. There are two types of scans: *sequential scan* and *index scan*. There are three types of joins: *nest-loop*, *mergejoin* and *hashjoin*. Hashjoin is only useful given a sufficient amount of main memory [48], hence has not been widely implemented until recently. All other scan and join operations, as described in any database textbook such as [30], are applicable in any environments. At run time, the query executor processes each operation in a plan sequentially. Intermediate result generation is avoided by the use of pipelining, in which the result tuples of one relational operation are immediately processed as the input tuples of the next operation.

IBM's System R requires the inner relation of any join operation to be a base table (i.e., a stored, permanent relation) [47]. The resulting query plans are called *deep tree* plans. The rationale is that this restriction allows the use of an existing index on the inner relation of a join to speed up the join processing and reduces the search space of plans significantly. In contrast to System R, both the university version and the commercial version of Ingres

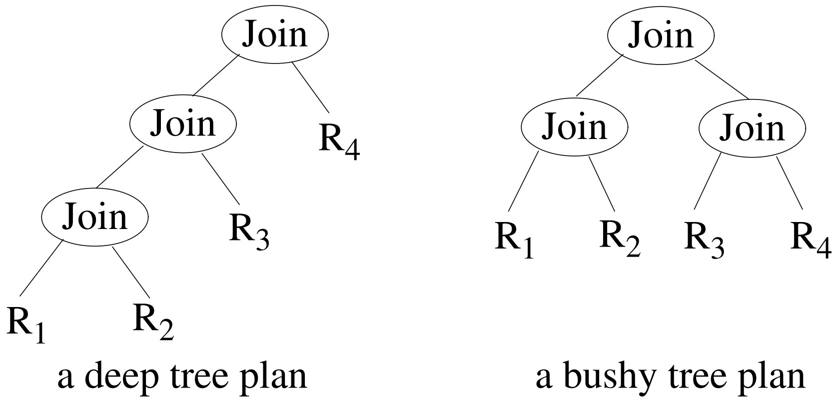

Figure 1.1: **Deep Tree Plan v.s. Bushy Tree Plan**

allow query plan trees of all shapes, i.e., *bushy tree* plans [59, 29]. Figure 1.1 shows an example of a deep tree plan and a bushy tree plan for the same four-way join query.

There are advantages and disadvantages for both deep and bushy tree plans. Apparently, query optimization becomes more difficult as the set of possible plans grows. The set of bushy tree plans is a superset of the set of deep tree plans, and is larger by several orders of magnitude. More precisely, the number of all possible deep tree plans is $n!$ for an n-way join query, whereas the number of all possible bushy tree plans is $n! \times C_{n-1}$, where C_k represents the k-th Catalan number which counts the number of ordered binary trees with k nodes [1]. Hence, restricting query plans to deep tree plans significantly reduces the search space for query optimization. On the other hand, the optimal bushy tree plan may be cheaper than the optimal deep tree plans. In [20], Graefe shows through simulations that bushy tree plans do not demonstrate much advantage over deep tree plans when the number of relations in a query is relatively small (≤ 10), but show significant advantage when the number of relations is large. This result justifies the design decision of System R

[1] C_{n-1} counts all possible arrangements of the interior nodes, i.e., join operations. There are also $n!$ ways to arrange the leaf nodes, i.e., the base relations. Therefore, the number of all possible bushy trees is $n! \times C_{n-1}$. Knuth shows in [28] that $C_k = \frac{1}{k+1}\frac{(2k)!}{k!k!}$.

of using deep tree plans for queries with a small number of relations. However, a deep tree plan eliminates the possibility of executing two joins in parallel. Therefore, it is important to consider bushy tree plans in a parallel database system so that parallelism between joins in the left subtree and those in the right subtree can be exploited. In this thesis, general bushy tree plans are considered to exploit parallelism.

Query optimization usually takes place at compile time. However, in a multi-user environment, many system parameters such as available buffer size and number of free processors in a parallel database system remain unknown until run time. These changing parameters may affect the cost of different query plans differently. Thus, we cannot simply perform compile-time optimization based on some default parameter values. This issue of query optimization with unknown parameters will be addressed in this thesis.

1.1.2 Parallel Query Processing

As we can see from the previous subsection, each sequential plan basically specifies a partial order for the relation operations. We call a query plan for a parallel environment a *parallel plan*. If a parallel plan satisfies the same partial order of operations as a sequential plan, it is called *a parallelization* of the sequential plan. Obviously, each parallel query plan is a *parallelization* of some sequential query plan and each sequential plan may have many different parallelizations. Parallelizations can be characterized in the following three aspects.

- **Form of Parallelism**

We can exploit parallelism within each operation, i.e., *intra-operation* parallelism and parallelism between different operations, i.e., *inter-operation* parallelism. Intra-operation parallelism is achieved by partitioning data among multiple processors and having those processors execute this same operation in parallel. Since intra-operation depends on data partitioning, it is also called *partitioned parallelism.* Inter-operation parallelism can be achieved either by executing independent operations in parallel or executing consecutive operations in a pipeline. We call parallelism between independent operations *independent parallelism* and parallelism of pipelined operations *pipelined parallelism.*

- **Unit of Parallelism**

  Unit of parallelism refers to the group of operations that is assigned to the same processor for execution. We also call a unit of parallelism a *plan fragments* since it is a "fragment" of a complete plan tree. In theory, a plan fragment can be any connected subgraph of a plan tree.

- **Degree of Parallelism**

  Degree of parallelism is the number of processes that are used to execute a plan fragment. In theory, the degree of parallelism can be greater than the number of available processors.

Figure 1.2 shows an example parallel plan. It illustrates the above three aspects for a parallelization of a mergejoin plan. As we can see, one input to the mergejoin is a sequential scan followed by a sort and the other input is an index scan. We choose to

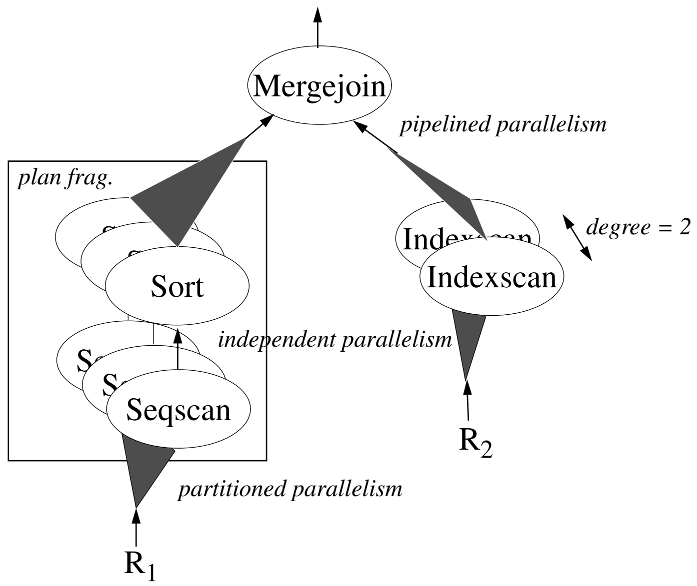

Figure 1.2: **An Example Parallel Plan**

parallelize the sequential scan and the sort together in the same plan fragment while the mergejoin and the index scan are parallelized in separate plan fragments. The sequential scan and sort are parallelized among three processes, i.e., the degree of parallelism is equal to 3. Each process scans one third of relation R_1 and sorts the qualified tuples from the sequential scan. Similarly the index scan is parallelized between two processes, each scanning half of relation R_2. (The details of how to parallelize a sequential scan or index scan will be described in Chapter 2.) These processes all pipeline their output to the mergejoin process. This parallelization includes all three forms of parallelism. There is partitioned parallelism within the sequential scan and sort on relation R_1 and the index scan on relation R_2. The sequential scan and sort on R_1 and the index scan on R_2 also run in parallel with independent parallelism, pipelining results into the mergejoin with pipelined parallelism.

Parallel query processing introduces many new issues. The important issues are how to reduce the search space of possible parallel plans which is orders of magnitude larger than the search space of sequential plans, how to schedule the processing of multiple plan fragments in an optimal way, and how to allocate main memory among multiple parallel plan fragments optimally. This thesis presents an integrated solution that addresses all these new issues.

1.2 An Overview of Previous Work

In the past decade, an enormous amount of work has been done in the field of parallel database systems. In the early days, most database machine research had focused on specialized, often trendy, hardware such as CCD memories, bubble memories, head-per-track disks, and optical disks. However, none of these technologies fulfilled their promises [8]. Parallel database systems did not become a success until the widespread adoption of the relational model and the rapid development of processor and disk technology. Now parallel database systems can be constructed economically with off-the-shelf conventional CPUs, electronic RAM, and moving-head magnetic disks. To date, many successful parallel database systems have been developed both in the commercial marketplace and in research institutions, as will be described in this section.

Parallel database systems would not have enjoyed such a big success of today had there not been a wealth of research results contributed by many researchers in this field. This section will only survey those works that are most relevant to this thesis. As pointed out in [50], there are three main architectures for parallel database systems: *shared disk*, *shared nothing*, and *shared everything*. Each of these three architectures has different characteristics for parallel query processing. In this thesis, we will concentrate on the shared everything architecture. Because shared disk has not been a popular approach and there has been little work done specifically for that architecture, we will not discuss the shared disk architecture in this thesis. Interested readers are referred to IBM's IMS/VS Data Sharing product [52] and DEC's VAX Rdb/VMS products [31] for more details about shared disk systems. Most previous research on parallel query processing is done in the context of the shared nothing architecture. Therefore, besides the shared everything architecture, we will also survey works on the shared nothing architecture. Section 1.2.1 first introduces the shared nothing architecture and describes parallel query execution in a shared nothing system. Then, Section 1.2.2 discusses the shared everything architecture and introduces XPRS, a prototype system which all the work in this thesis is based on. Last, Section 1.2.3 surveys previous work on parallel query optimization.

1.2.1 Shared Nothing Systems

The Shared Nothing Architecture

With a shared nothing system, each processor owns a portion of the database and only that portion may be directly accessed or manipulated by that processor. A two-phase commit protocol [49] is required to coordinate a transaction commit which involves multiple nodes. Examples of shared nothing systems include Tandem's NonStop SQL [56], Teradata's DBC/1012 [55], MCC's Bubba [7] and University of Wisconsin's Gamma [17].

The key to parallelism in a shared nothing system is based on the concept of *declustering*. Declustering a relation involves distributing its tuples among multiple nodes according to some distribution criteria such as applying a hash function to the key attribute of each tuple. Declustering has its origin in the concept of *horizontal partitioning* initially developed as a data distribution mechanism for distributed DBMS [43]. After declustering, each processor in a shared nothing system is in charge of a portion of the database residing on its local disk drives. This enables the DBMS software to exploit the I/O bandwidth reading and writing multiple disks in parallel.

There are three basic declustering schemes: *range partitioning*, *round-robin* and *hashing* as illustrated in Figure 1.3. Range declustering maps tuples with a key value in a certain range to a certain disk. Round-robin declustering maps the i'th tuple or page to disk i mod n (where n is the number of disks). Hashing declustering assigns tuples to disks according to a hash function. Among the shared nothing systems, Teradata only supports round-robin and hashing, while Tandem, Bubba and Gamma supports all three declustering schemes.

For example, suppose that an employee relation is declustered among 5 sites using range partitioning on the salary attribute as follows:

$$\begin{array}{ll}
\text{Site 1:} & salary < 10K \\
\text{Site 2:} & 10K \leq salary < 20K \\
\text{Site 3:} & 20K \leq salary < 30K \\
\text{Site 4:} & 30K \leq salary < 40K \\
\text{Site 5:} & salary \geq 40K.
\end{array}$$

In this case, a query to find all the employees who earn \$25K will be processed only at Site 3. In general, range partitioning can restrict the processing of a query to a minimum number of processors, which complicates the issue of load balancing. Unless salaries of employees are uniformly distributed and queries on salary ranges are likewise uniform, the five sites in the system will not have an equal amount of work to do, and the entire system

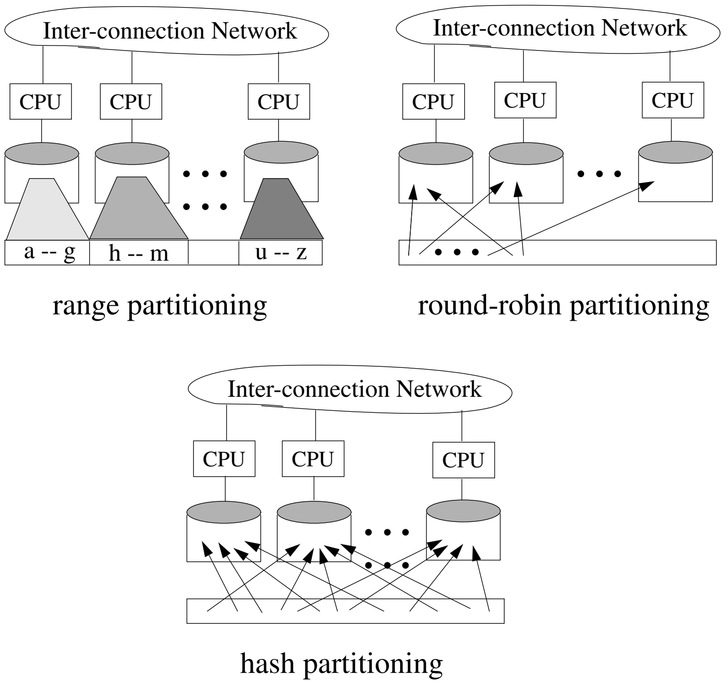

Figure 1.3: **Three Basic Declustering Schemes**

will bottleneck on the overloaded processor.

The load balancing problem has prompted people to favor round-robin and hashing declustering. However, hashing or round-robin declustering will almost guarantee that all five processors will execute every query (except for an exact-match query on the hashed attribute). For a query that only returns a single tuple, this will result in 5 times as much work as necessary, which will certainly hurt transaction processing performance.

Another problem of a shared nothing system is the communication overhead. When a query is executed in parallel on multiple systems, there is inevitable communication among the systems to synchronize multiple computation. At the same time, data must often be sent from one system to other systems to gain parallelism. For example, to perform a join in parallel among multiple nodes, one of the join relations may need to be broadcasted to all the participating nodes. This generates more traffic on the network. When the degree of declustering is increased, the response time of individual queries will decrease at first because of higher parallelism, but after a certain point, the response time will increase because the communication overhead has become a significant fraction of the overall execution cost. This problem has prompted Bubba to advocate declustering only to a subset of the disks to reduce communication cost [12]. However, this also raises a number of new issues for physical database design. In Bubba, in addition to selecting a declustering strategy for each relation, the number of disks over which a relation should be declustered must also be decided. Copeland, et al. describe in [12] an approach based on the *heat* of a tuple, i.e., the access frequency of the tuple over some period of time, which balance the frequency with which each disk is accessed rather than the actual number of tuples on each disk.

Parallel Query Execution in Gamma

We describe how query execution can be parallelized through the particular implementation in Gamma, which is representative for shared nothing systems. As is pointed out in [21], there are two models for parallelizing relational queries, i.e., the *bracket model* and the *operator model*. Gamma adopts the bracket model. The operator model will be described in the next subsection.

In the bracket model, there is a generic template process for each type of operations that can receive and send data and can execute exactly one operation at any point of time. A schematic diagram of such a template process for join operations is shown in Figure 1.4. The code that makes up the generic template calls the code for the operation which then

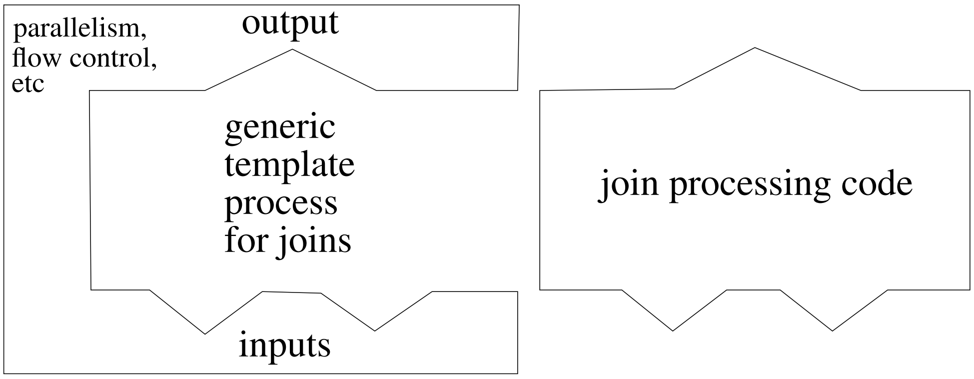

Figure 1.4: **Bracket Model of Parallelization**

controls execution; network I/O on the receiving and sending sides are performed as service to the operation on request, implemented as procedures to be called by the operation. The operation is surrounded by the generic template code which shields it from its environment. The algorithms for the relational operations are still written as if they were to be run on a uniprocessor.

In Gamma, this generic template is implemented through a structure called a *split table*. As shown in Figure 1.5, the input to an operator process is a stream of tuples and the output is a stream of tuples demultiplexed to multiple subsequent operator processes indicated in the split table. The split table defines a mapping of values to a set of destination processes. Figure 1.6 gives an example of a split table for the execution of a join operation using four processors. Each process producing tuples for the join will apply a hash function to the join attribute of each output tuple to produce a value between 0 and 3. This value is then used as an index into the split table to obtain the address of the destination process that should receive the tuple. Each operation can be executed by multiple parallel processes and processes executing different operations are connected together via the split tables. Overall, queries are executed in a data flow fashion by these parallel processes, each

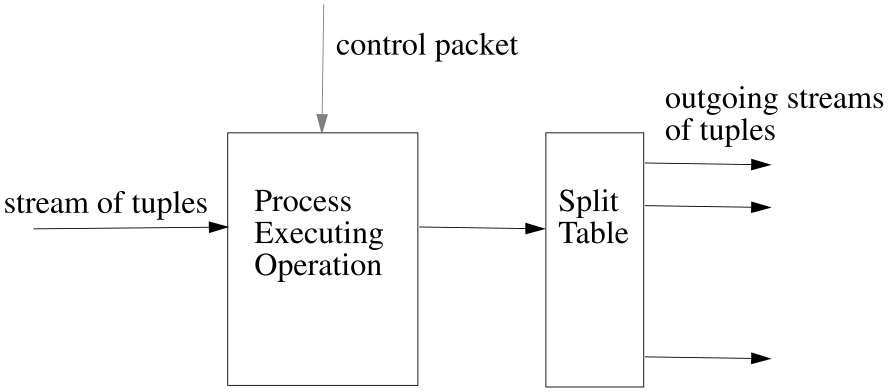

Figure 1.5: **Generic Template for Parallelization in Gamma**

Value	Destination Process
0	(Process #3, Port #5)
1	(Process #2, Port #13)
2	(Process #7, Port #6)
3	(Process #9, Port #15)

Figure 1.6: **An Example Split Table**

executing a single relational operation. In other words, tuples are pipelined between the parallel processes as soon as they are made available by an operation. In Gamma, each query also has a scheduler process that creates and terminates all the operator processes for executing the query. It is also responsible for the flow control between any processes that are in a producer-consumer relationship. See [17] for more details about the parallelization scheme of Gamma.

One problem with the bracket model is that each thread of control needs to be created. This is typically done by a separate scheduler process which requires software development beyond the actual operators, both initially and for each extension to the set of query processing algorithms. Thus, the bracket model seems unsuitable for an extensible system. In the bracket model, there are different generic template processes for different types of operations. For example, the template process for joins must listen to two input ports, while the template process for scans only need to listen to one input port. These operation-specific templates constraint the unit of parallelization as a single operation. Therefore, one operation has to pay the overhead of inter-process communication (IPC) to call another operation, whereas if more than one operations can be executed in the same process, operations can call each other much more efficiently by simple procedure calls.

Next, we discuss the shared everything systems including XPRS and Volcano. The operator model of parallelization will be described with the Volcano system since it is originally proposed for that system.

1.2.2 Shared Everything Systems

In a shared everything system, main memory, in addition to disks, is also shared across all the processors, making system management and load balancing much easier. An example of shared everything systems is University of California at Berkeley's XPRS system [41] which we will describe in details throughout this thesis.

In the context of query processing, the main advantage of a shared nothing system is its scalability. It may be possible to scale a shared nothing systems to hundreds, even thousands of processors. The main disadvantage of a shared nothing system is the communication overhead. Contrarily, a shared everything system has the advantage of no communication overhead and easy load balancing, but is limited on scalability. Ultimately bounded by the internal bus bandwidth, usually a shared everything system can at most scale up to tens of processors. However, a shared everything system can be used as a node in a shared nothing system to reduce the number of nodes and increase efficiency.

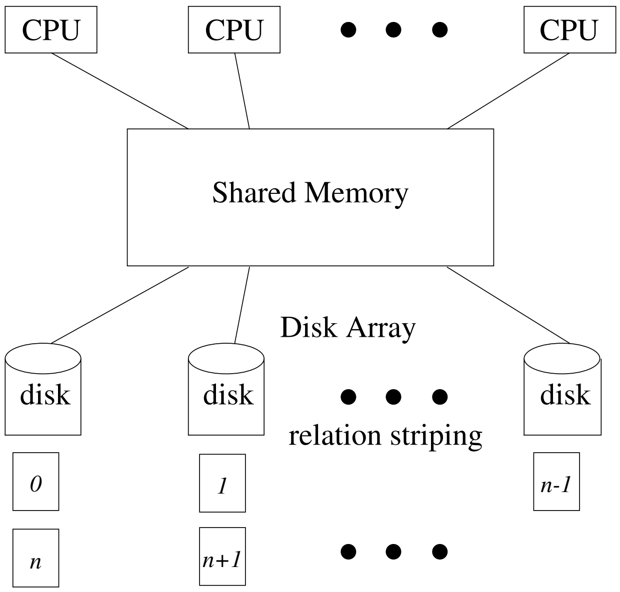

Figure 1.7: **The Overall Architecture of XPRS**

In this subsection, we discuss the shared everything architecture through the XPRS system, which this thesis is based on. We also describe another shared everything system, the Volcano system of University of Colorado at Boulder to introduce the operator model for parallelizing relational queries.

The Architecture of XPRS

XPRS (eXtended Postgres on Raid and Sprite) of University of California at Berkeley is a multi-user parallel database system based on the POSTGRES next-generation DBMS [51]. It is implemented on a shared memory multiprocessor and a disk array as shown in Figure 1.7. An initial design of XPRS can be found in [41] and the underlining reliable disk array RAID is described in [37]. Currently, XPRS is operational on a Sequent Symmetry system running Dynix operating system with 12 processors and 7 disks.

A shared everything system has two major advantages over a shared nothing system. First, there are no communication delays because messages are exchanged through the shared memory, and synchronization can be accomplished by cheap, low level mechanisms. Second, load balancing is much easier because the operating system can automatically allocate the next ready process to the first available processor. The simulation results in [2] show that a shared everything system will outperform a shared nothing system with an equivalent number of processors, disks and megabytes of main memory by as much as a factor of two. As will be described in the rest of this thesis, XPRS is designed and implemented to take full advantages of the shared everything architecture.

Database applications are often I/O intensive. In order to keep up with the I/O requests from multiple processors, XPRS uses a disk array to eliminate the I/O bottleneck. All relations are striped [45] sequentially, block by block, in a round-robin fashion across the disk array to allow maximum I/O bandwidth. Figure 1.7 only shows the non-redundant disk array configuration, i.e., RAID Level 0 as classified in [37], which is the default configuration for XPRS. XPRS also supports other disk array configurations. Performance implications of running XPRS on other disk array configurations will be discussed in Chapter 5.

The goals of XPRS are *high performance* and *high availability*. This thesis only deals with the high performance aspects of XPRS. Discussions on the high availability aspects of XPRS can be found in [53]. XPRS is designed to achieve high performance for both transaction processing and complex ad-hoc queries through parallelism within each individual query as well as between different queries.

Parallel Query Execution in Volcano

The Volcano query processing system is also a shared everything system. It tries to combine extensibility with parallelism and introduced the operator model of parallelization to overcome the problems of the bracket model. The basic idea is to encapsulate all the issues of parallelization into a single operator, which is called the *exchange* operator in Volcano [21].

In Volcano, each operator is implemented as a *iterator*, i.e., it supports a simple *open-next-close* interface similar to conventional file scans. For a given query plan, calling *open* for the root operator results in initializations of execution states e.g., allocation of a hash table, and open calls for all its inputs. After all iterators in a query plan are initialized recursively, the query is processed by calling *next* for the root operator repeatedly until it fails with an *end-of-stream* indicator. Finally, the *close* call recursively shut down all iterators in the query plan.

The *exchange* operator is also implemented as an iterator, but it has a special set of *open, next* and *close* procedures. An *open* call to an *exchange* operator will create one or more child processes. The parent process will serve as the consumer and the child processes will serve as the producers. Proper communication channels between the parent process and child processes will also be set up by the *open* call. For a *next* call to the *exchange* operator, the parent process will try to get the next tuple from the communication channels from the child processes. Meanwhile, the child processes will be executing some operations in parallel, packing the result tuples into packets and sending the packets to the parent process. Flow control is also exercised in the *exchange* operator to keep the child processes from overflowing the communication buffers. Finally, when a *close* call comes to the *exchange* operator, the parent process will send a message to the child processes telling them to clean up and terminate.

The advantage of implementing the *exchange* operator as an iterator is that it can be inserted at any one place or multiple places in a complex query plan. Figure 1.8 shows a query plan that includes both relational operators, i.e., scans and joins, and exchange operators. An open call to the top exchange operator by the starting process will create some processes to process the plan fragment consisting of the hashjoin, the nestloop join and the index scan. These new processes will in turn call open to the exchange operators above the two sequential scans which creates multiple processes to process the sequential scans. In this way, all the parallel processes and communication channels between them are set up. Then the query plan will be processed in parallel by these processes. As we can see, the *exchange* operator can support not only different units of parallelism, namely single operations or multiple operations (depending on where it is inserted), but also different forms of parallelism, i.e., pipelining between operations (e.g, between the sequential scans and the hashjoin), parallelism between operations (e.g., between the two sequential scans) and parallelism within operations (e.g., within each sequential scan). More details about the exchange operator are discussed in [21].

In summary, parallelization involves many new issues, such as, creation and termination of parallel processes, establishment of communication channels, flow control between asynchronous processes, etc. The bracket model tries to deal with these issues in a scheduler and the template processes for different operation types. Therefore, these issues are

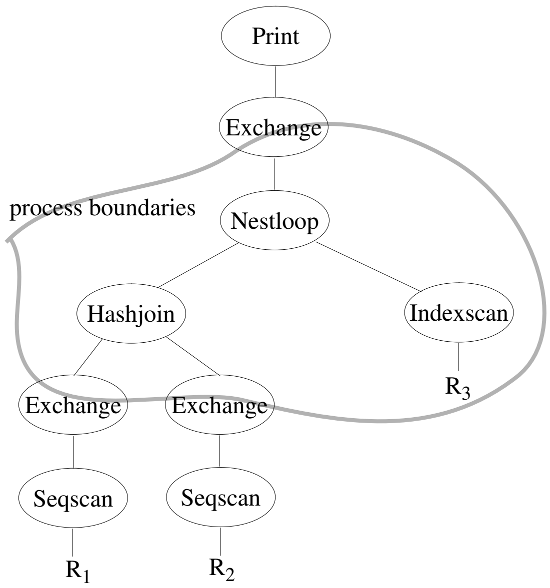

Figure 1.8: **Query Plan with Exchange Operators**

not encapsulated in one single module, which complicates extensions to the system. On the other hand, in the operator model, all the issues about parallelization are dealt with in one operator. The rest of the system can be implemented as in a conventional uniprocessor environment, without concerning with parallelization issues.

There is also a large amount of work that has been done on parallelizing specific relational operations, which we will not cover in this thesis. Interested readers are referred to [3] and [57] for some early work on parallel execution of relation operations, [4] and [35] for sorting, [14], [33] and [58] for hashjoins, also [58] for mergejoins.

1.2.3 Optimization Algorithms

After we understand the parallelization of relational queries, the next question is how to optimize all the possible parallelizations. In XPRS, we are have focused on two main issues in this optimization problem, i.e., dealing with compile-time unknown parameters and the large search space of parallel plans. In a multi-user environment as XPRS, parameters such as the amount of available buffer space and number of free processors are unknown at compile time. Therefore, compile-time optimization must generate plans for an uncertain run-time environment. Second, the number of possible parallel query execution plans is so enormous that any exhaustive search algorithm is impractical. As a result, the search space must be heuristically reduced. Previous work on these two issues will be surveyed below.

Dealing with Unknown Parameters

Most current database systems avoids the unknown parameter problem by making certain assumptions about values of system parameters (e.g., number of available buffers). However, at run time these assumptions may be violated because the run-time workload is unpredictable. When these assumptions are violated, queries need to be re-optimized to avoid performance degradation.

To reduce the need for re-optimization, some researchers have studied the problem of optimizing queries with unknown parameters. The earliest significant work in this area is presented in [22], which discusses the implementation of *dynamic query plans* in the Volcano optimizer generator. The main idea is to introduce *choose-plan* nodes to generate multiple query plans, consequently, the run-time system must go through a decision tree to choose a plan according to the current system parameters. The proposal is to introduce choose-plan nodes at all places in a plan where the choice of subplans underneath is sensitive to the values of system parameters. This work introduces many important concepts related to query optimization with unknown parameters, but does not include a complete search strategy to identify the dynamic plans and the places where the choose-plan operator should be placed.

The Starburst project has also considered incorporating a second optimization phase that chooses plans at run time [23], but no technique has been developed to find those plans. Also, [13] uses an integer programming model to optimize queries in a transaction processing environment and their buffer allocations simultaneously. However, in the end, only one plan is produced per query, and that plan is susceptible to changes in the environment.

The most recent work is presented in [24], which proposes a general framework for dealing with unknown parameters in query optimization. The buffer size parameter is considered in depth to illustrate the general approach, which is based on randomized query optimization algorithms, such as *Simulated Annealing* [25] and *Iterative Improvement* [54]. In a randomized query optimization algorithm, the query plan space is represented as a state transition graph with each state corresponding to a query plan and each transition corresponding to an event such as change of join orders or join methods. A *local optimal* plan is a plan that has lower cost than all its adjacent plans in the state transition graph. A randomized query optimization algorithm randomly walks through the state transition graph finding local optimal plans and returns the minimum-cost local optimal plan that is ever visited. In the approach proposed in [24], there is a co-routine to optimize the query plans for each value of an unknown parameter with a randomized algorithm. The key idea of this approach is to have *sideways information passing* among the co-routines, i.e., for each co-routine to let other co-routines know the minimum cost plan that it has visited. The depth of sideways information passing is defined as the number of co-routines which each co-routine pass information to. The paper shows through experiments that for the buffer size parameter, Iterative Improvement with sideways information passing of depth 1 is the most effective randomized optimization algorithm. The problem with this approach is that it is only applicable to randomized algorithms and cannot be incorporated into existing conventional query optimizers. This is because if a deterministic algorithm is used, every co-routine will make the same move at each step, which renders sideways information passing useless.

Optimizing Parallel Execution Plans

There has been little work done on this topic. To date, no query optimizers consider all parallel algorithms for each operator and all possible query plans. The main difficulty of this optimization problem is the enormous search space of all possible parallel query plans.

In [9], Bultzingsloewen outlines six key issues in query optimization in loosely or tightly coupled multiprocessors with private disks and sketches the optimization strategy that was being implemented for the KARDAMOM database machine. However, the paper does not give any details about how to cut down the size of the search space of parallel query plans.

Schneider and DeWitt present in [46] an experimental analysis of the query processing tradeoffs among *left-deep* and *right-deep* tree plans in a shared-nothing enironment. An important finding is that right-deep trees are superior given sufficient memory resources. However, there is no analytical cost expression which can be used by an optimizer to decide whether and when to switch from a left-deep tree to a right-deep tree. Moreover, no algorithms are proposed for determining the degree of parallelism for each parallel operation.

In [36], Murphy and Shan propose an algorithm to parallelize a given sequential plan to achieve minimum duration time with computational resource requirements less than the given system bounds in a shared memory environment. It does not address the problem of how to choose a sequential plan to parallelize, and it assumes that the amount of available resources is known and therefore the algorithm cannot be used at compile time for a multi-user environment. Moreover, the algorithm includes testing a series of target duration times starting from the lower bound and therefore might be too expensive for run-time optimization.

Lu, et al. propose in [32] an optimization algorithm that considers bushy tree plans and inter-operation parallelism. The algorithm only handles *synchronized* bushy tree plans, i.e., those without pipelining between joins, and uses a greedy algorithm to choose a bushy tree plan that always tries to join as many pairs of relations as possible in parallel for each step. This maximum inter-operation parallelism approach may not be a good way to parallelize a query plan. A careful balance between inter-operation parallelism and intra-operation parallelism need to be maintained.

1.3 Overview of This Thesis

In summary, extensive research and development have been done on parallelizations of relational query executions. There are two main approaches: the bracket model used in Gamma which provides a generic template process that controls the parallel execution of single operations, and the operator model used in Volcano which encapsulate all the parallelization issues within an operator. The operator model is obviously cleaner and supports automatic parallelism of any new operator added to the system. In fact, Gamma has also recently adopted the operator model and introduced a merge operator and a split operator to encapsulate parallelism issues [15]. In XPRS, parallelization of relational operations is not the main focus. Our emphases have been put on parallel query optimization, parallel task scheduling and memory allocation strategies. There have been little work done on these topics. In the following three chapters, we will present our solutions to each of these problems integrated in the XPRS environment, which constitutes a complete approach to parallel query processing.

In the following chapter, the design and implementation of parallel query processing in XPRS is described along with some performance figures from our operational prototype. We propose a two phase optimization strategy that solves the unknown parameter problem and significantly reduces the search space of parallel query plans while still preserving the optimality of resulting plans. Experiment results through XPRS benchmarks are shown to verify the effectiveness of this optimization strategy.

Most parallel database systems such as Gamma and Bubba only consider intra-operation parallelism. On the other hand, XPRS exploits both intra-operation parallelism and inter-operation parallelism. In such an environment, there may be multiple tasks (consisting of relational operations) that are ready to run at the same time and an optimal processing schedule need be determined for these tasks such that the total processing time is minimized. The task scheduling problem in parallel query processing is introduced in Chapter 3. Our solution to the problem is an adaptive scheduling algorithm which is based on the concept of IO-CPU balance point that maximizes system resource utilizations. It tries to execute IO-bound and CPU-bound tasks at their balance point and dynamically adjusts parallelism to keep running at the balance point when workload changes.

Main memory is also one of the most important resources in parallel query processing along with processors and I/O bandwidth. When multiple tasks are running in parallel, the problem of how to optimally allocate a limited amount of memory to these tasks arises. The memory allocation problem for parallel hashjoin operations is studied in Chapter 4. A theorem is developed for determining the optimal memory allocation between parallel hashjoins. A mechanism for dynamic memory allocation adjustment is designed so that the optimal memory allocation can be preserved when workload changes. Integration of our memory allocation strategy with the task scheduling algorithm is also described.

Up to Chapter 5, this thesis only considers one particular disk array configuration, the RAID Level 0 or the simplex configuration. Chapter 5 completes the discussion on parallel query processing by studying the performance of different disk array configurations in a parallel query processing environment. XPRS supports three different disk array configurations: simplex, mirror disks and parity arrays. Experiment results have been obtained from XPRS to illustrate the performance tradeoffs between these disk array configurations. Chapter 5 reports these experiment results.

Last, our conclusions and discussions on future research directions are offered in Chapter 6.

Chapter 2

Optimization of Parallel Query Execution Plans

The goal of parallel query optimization is to find the optimal parallel query execution plan for any given user query. As discussed in the previous chapter, the main difficulties in this optimization problem are the compile-time unknown parameters such as available buffer size and number of free processors, and the enormous search space of possible parallel plans. In this chapter, we will present the XPRS solution to this problem, a two phase optimization strategy, along with experimental evidence from XPRS benchmarks that supports the effectiveness of this strategy. Considering inter-operation parallelism requires a solution to the task scheduling problem, which is the topic of the next chapter. For ease of presentation, this chapter only considers intra-operation parallelism. A more complete approach that considers both intra-operation and inter-operation parallelism will be described in the next chapter after the task scheduling problem is solved.

This chapter is organized as follows. Section 2.1 defines the optimization problem and presents our two phase optimization strategy along with the hypotheses that it is based on. Section 2.2 then discusses the performance of parallelizations of a sequential plan and presents the overall architecture for parallel query processing in XPRS. Section 2.3 then describes some experiments that we have performed on XPRS which largely verify the hypotheses that our two phase optimization strategy is based on. Last, Section 2.4 summarizes the whole chapter.

2.1 Optimization of Parallel Plans

This section defines the optimization problem, presents our two phase optimization strategy and the hypotheses that it is based on, and gives our intuition behind the hypotheses. Experimental verifications of these hypotheses will be provided in Section 2.3. We also demonstrate the necessity to dynamically adjust plans at run time according to available buffer sizes and introduce a choose node to implement the plan adjustment.

2.1.1 The Optimization Problem

The overall performance goal of a parallel database system is to obtain maximum throughput as well as minimum response time in a multi-user environment. The objective function that XPRS uses for query optimization is a combination of resource consumption and response time as follows:

$$cost = resource\_consumption + w \times response\_time.$$

Here w is a system-specific weighting factor. A small w mostly optimizes resource consumption, while a large w mostly optimizes response time. Resource consumption is measured by the number of disk pages accessed and number of tuples processed, while response time is the elapsed time for executing the query.

Our optimization problem is to find the parallel plan with minimum cost among all possible parallelizations of all possible sequential plans of a query. Suppose P is a sequential plan and let $PARALLEL(P)$ be the set of possible parallelizations of P. Suppose Q is a given query and let $SPLAN(Q)$ be the set of sequential plans of Q. Then $PPLAN(Q)$, the set of parallel plans of Q, is given by

$$PPLAN(Q) = \bigcup_{P \in SPLAN(Q)} PARALLEL(P).$$

$PPLAN(Q)$ is the search space to explore for intra-query parallelism.

In a multi-user environment, many system parameters that affect query execution cost are unknown at compile time. In XPRS, we specifically consider two of such parameters: available buffer size, $NBUFS$ and number of free processors, $NPROCS$. Buffer size not only affects the buffer hit rate but also determines the number of batches in a hashjoin [48]. The number of free processors determines the possible speedup of query execution. For ease of exposition, in the following formulation, we assume that the buffer size and number of free processors are fixed during the execution of a single query. As we will describe in Section 2.2.3, our operational prototype only fixes these parameters during the execution of individual plan fragments.

For $PP \in PPLAN(Q)$, let $Cost(PP, NBUFS, NPROCS)$ be the cost of the parallel plan PP given $NBUFS$ units of buffer space and $NPROCS$ free processors. Our optimization problem is to find,

$$min\{Cost(PP, NBUFS, NPROCS)|PP \in PPLAN(Q)\}.$$

There are two major difficulties in this problem. First, the search space $PPLAN(Q)$ is orders of magnitude larger than $SPLAN(Q)$; therefore query optimization by exhaustive search [47] is impractical. Second, the dynamic parameters, $NBUFS$ and $NPROCS$ are unknown until query execution time; therefore compile-time optimization must deal with these unknown parameters. Our goals are to reduce the search space of parallel plans by heuristics and to perform as much compile-time optimization as possible.

2.1.2 Two Phase Optimization

We achieve our goals by the following two phase optimization strategy.

Let $BPP(Q, NBUFS, NPROCS)$ be the best parallel plan for query Q given $NBUFS$ units of buffer space and $NPROCS$ free processors, $BP(Q, NBUFS)$ be the best sequential plan for Q given $NBUFS$ units of buffer space, and $Cost(P, NBUFS)$ be the cost of a sequential plan P given $NBUFS$ units of buffer space.

- **Phase 1.** Find the optimal sequential plan assuming that the entire buffer pool is available, i.e., find $BP(Q, ALLBUFS)$ where $ALLBUFS$ is the size of the whole buffer pool.

- **Phase 2.** Find the optimal parallelization of the optimal sequential plan, i.e., find $min\{cost(PP, NBUFS, NPROCS) \mid PP \in PARALLEL(BP(Q, ALLBUFS))\}$

where $NBUFS$ and $NPROCS$ are the run-time available buffer size and number of free processors.

Because we have a fixed buffer size $ALLBUFS$ in Phase 1, it can be performed at compile time. Phase 2 still has to be performed at run time, because it takes the run-time parameters $NBUFS$ and $NPROCS$ into account and tries to dynamically determine the best parallelization of the sequential plan chosen in Phase 1. As we can see, this two phase optimization strategy overcomes the difficulties in our optimization problem, because it provides a nice partitioning between compile-time optimization and run-time optimization and significantly reduces the search space by restricting to the parallelizations of one particular sequential plan. The effectiveness of this strategy is based on the following two hypotheses for a shared everything environment.

1. **The Buffer Size Independent Hypothesis**

   The choice of the best sequential plan is insensitive to the amount of buffer space available as long as the buffer size is above the hashjoin threshold, i.e.,

   $$Cost(BP(Q, NBUFS), NBUFS) \approx Cost(BP(Q, NBUFS'), NBUFS),$$

   where $NBUFS \neq NBUFS', NBUFS \geq T, NBUFS' \geq T,$ *and* T *is the hashjoin threshold.*

   As we will discuss in details in the next subsection, certain exceptions to the above hypothesis do exist; however, they can be localized within specific operations and handled with a mechanism that dynamically chooses the implementation of an operation at run time according to real buffer sizes.

2. **The Two Phase Hypothesis**

   For a shared everything environment where only intra-operation parallelism is exploited, the best parallel plan is a parallelization of the best sequential plan, i.e.,

$$BPP(Q, NBUFS, NPROCS) \in PARALLEL(BP(Q, NBUFS)).$$

We will verify these two hypotheses with experiment results in the Section 2.3.

2.1.3 Introduction of Choose Nodes

We have identified two situations in the execution of single operations that may cause problems with the buffer size independent hypothesis. One is choosing between an indexscan using an unclustered index and a sequential scan. The other is choosing between a nestloop with an indexscan over the inner relation and a hashjoin. These two situations are not necessarily the only problematic cases, but they are the only cases that we have discovered among a wide variety of queries as will be described in Section 2.3.1. Moreover, the choose nodes that we are about to introduce are a general mechanism and can deal with any additional cases as they are discovered.

We will show the two problematic cases that we have found by plotting the execution costs of some sample queries against varying buffer sizes. In general, the cost of a sequential plan is measured by resource consumption [47], which is a linear combination of I/O cost and CPU cost as follows,

$$Cost = \#page\_io + c \times \#tuples\_processed.$$

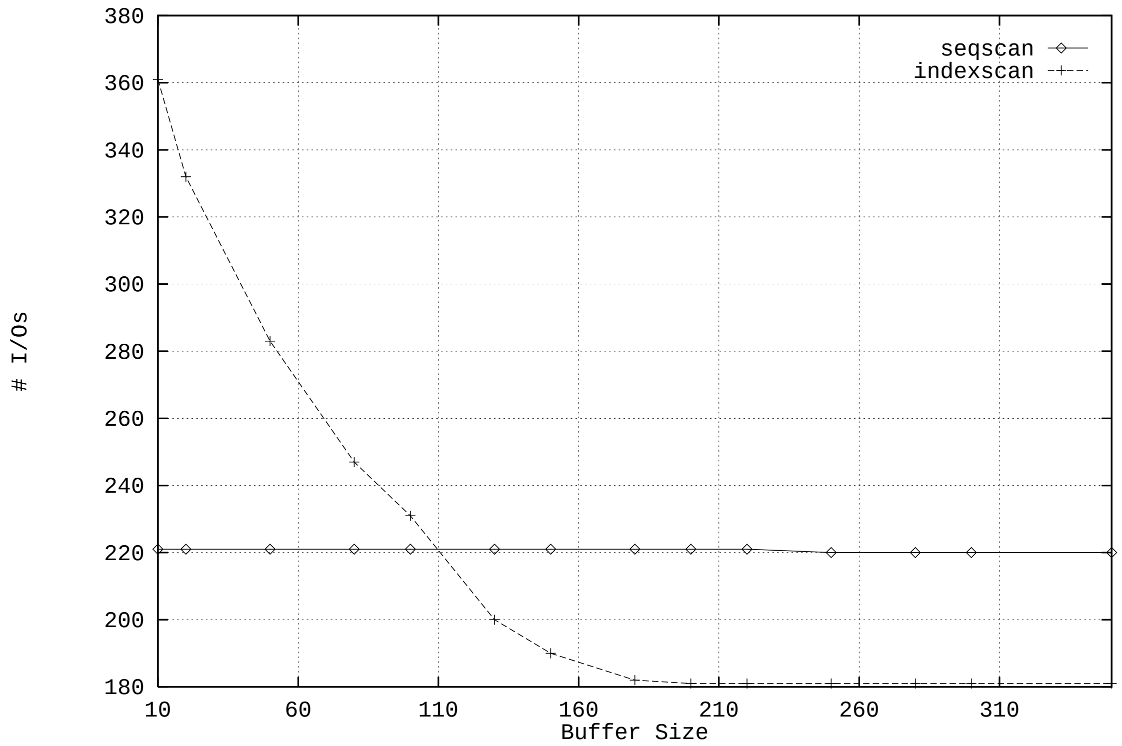

Figure 2.1: **Cost of SeqScan v.s. IndexScan**

Here c is another system-specific weighting factor. Since buffer sizes only affect the I/O cost of query execution, we will only plot the I/O costs (i.e., number of page I/Os) of query executions in the examples below.

- **Sequential Scan versus Index Scan**

  A sequential scan only needs one buffer page and additional buffer pages do not reduce query execution cost. However, the cost of an indexscan using an unclustered index is very sensitive to buffer size. If there is enough buffer space to hold all the pages that need to be fetched during the indexscan, then we only need to read each of those pages once. Therefore, with sufficient buffer space, if an indexscan touches less than all pages, it will have lower cost than a sequential scan. On the other hand, with few buffers, an indexscan may end up fetching the same page from disk many times and becomes more expensive than a sequential scan. Figure 2.1 gives an example of this situation by plotting the cost of each plan versus buffer size for the following selection query on a 10,000-tuple relation from the Wisconsin benchmark:

$$\text{retrieve (tenk1.all) where tenk1.u1} > 110 \text{ and tenk1.u1} \leq 510.$$

The I/O costs of this query are measured from real executions of this query sequentially (in a single process) on XPRS configured with different buffer sizes. Obviously, when the two curves cross, we require a mechanism to switch from a sequential scan to an index scan, or vice versa. Note that this situation does not always happen. In most cases, the two curves are far apart and do not cross each other. This situation only arises for a certain selectivity range which causes these two curves to be close together. The example query in Figure 2.1 is carefully selected to illustrate this situation.

- **Hashjoin versus Nestloop**

  A similar situation occurs in choosing between a nestloop with an indexscan over the inner relation, which we will refer to as an *indexjoin*, and a hashjoin. With sufficient buffer space, an indexjoin will fetch ultimately all the pages in the outer relation and all the pages in the inner relation that match some tuple in the outer relation plus a small number of index pages. On the other hand, a hashjoin will need to fetch all the pages in both the outer and inner relations. If the join selectivity is small, the

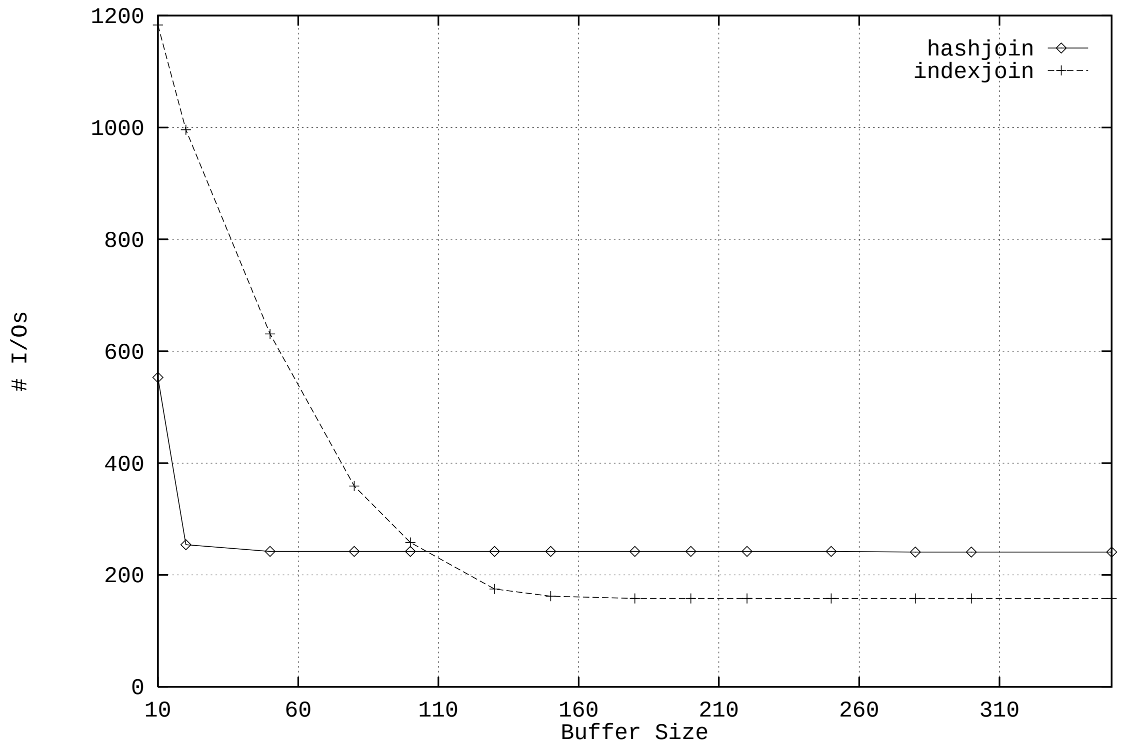

Figure 2.2: **Cost of Nestloop with Index v.s. Hashjoin**

indexjoin plan may not need to fetch all the pages in the inner relation and therefore will have a lower cost than the hashjoin plan. On the other hand, with few buffers, the indexscan in indexjoin may have to fetch the same page many times and cause the indexjoin to become more expensive than a hashjoin. Figure 2.2 shows an example of a "cross over" point between indexjoin and hashjoin of the following carefully selected join query between two Wisconsin benchmark relations:

retrieve (onek.all, tenk1.all) where onek.u1 = tenk1.a10.

Again, this situation only occurs for a certain range of join selectivities.

To solve these "cross over" problems, we introduce a new *choose* node into query plans, which can choose between the two join or two scan methods depending on which side of the cross over point the actual buffer size belongs to. Now we modify the buffer size independent hypothesis as follows.

Modified Buffer Size Independent Hypothesis

The choice of the best sequential plan with choose nodes is insensitive to the amount of buffer space available as long as the buffer size is above the hashjoin threshold, i.e.,

$$Cost(BCP(Q, NBUFS), NBUFS) \approx Cost(BCP(Q, NBUFS'), NBUFS),$$

where $BCP(Q, NBUFS)$ is the optimal sequential plan of Q under buffer size $NBUFS$ with choose nodes, $NBUFS \neq NBUFS', NBUFS \geq T, NBUFS' \geq T$, and T is the hashjoin threshold.

2.1.4 Intuition Behind Hypotheses

Now we explain the intuition behind these hypotheses. Let us first consider the buffer size independent hypothesis for a single operation, i.e., a scan or a join. If there are no indices that can be used to facilitate the operation, the hypothesis obviously holds, because only a sequential scan plan is possible for scans and a hashjoin plan is always optimal for joins as long as the buffer size is above the hashjoin threshold [48]. This is why requiring the buffer size to be above the hashjoin threshold is important. Otherwise, a hashjoin plan may not be possible and the optimal plan can be either nestloop or mergejoin depending on the situation [6], thus, the hypothesis may not hold for two-way join queries. If there are indices defined on a selection or join attribute, usually we want to take advantage of the indices to form plans that involve index scans, and the problematic cases described above may occur. However, we can deal with all the problematic cases by encapsulating all the necessary plan switching in a choose node. Therefore, with choose nodes, the buffer size independent hypothesis is guaranteed to hold for all single operations. Furthermore, we postulate that this remains to hold for complete query plans consisting of one or more operations. Section 2.3.2 will show that this is generally true with only small errors.

The two phase hypothesis only holds for a shared everything environment and only for intra-operation parallelism. In a shared nothing environment, communication cost must also be considered. Thus, the hypothesis may become false because the optimal sequential plan may incur excessive communication cost when parallelized and therefore its parallelization can only be a sub-optimal parallel plan. For example, in an optimal sequential plan, relation R_1 and relation R_2 are joined first. However, if the relevant tuples of R_1 and R_2 are declustered among different sites, it may save a lot of communication overhead if the join of R_1 and R_2 is delayed to a later stage of the plan when many tuples are already filtered out. Therefore, the two phase hypothesis is not always true in a shared nothing environment. The two phase hypothesis may also become false when inter-operation parallelism is considered. As will be shown in the next chapter, inter-operation parallelism between an IO-bound operation and a CPU-bound operation can achieve better resource utilization than only intra-operation parallelism. Therefore, a bushy plan that contains a mixture of independent IO-bound and CPU-bound operations may become cheaper than the optimal sequential plan when parallelized. On the other hand, if we only consider a shared everything environment and intra-operation parallelism, as we will show in Section 2.2.2, each sequential plan can be parallelized without incurring any extra resource consumption.

Since the optimal sequential plan minimizes resource consumption, so does its parallelization. Hence, the two phase hypothesis is trivially true if only resource consumption is considered. For response time, we will also show in Section 2.2.2, each sequential plan can achieve a near-linear speedup through intra-operation parallelism. Since the optimal sequential plan is also the fastest sequential plan, its parallelization will remain the fastest. Although this is not always true, Section 2.3.3 will show that the errors are very small.

2.2 XPRS Query Processing

Having explained our two phase optimization strategy, in this section, we present the design and implementation of XPRS query processing. We first describe the parallelization of individual operations in XPRS along with their performances, then present the overall architecture of XPRS query processing.

2.2.1 Implementation of Intra-operation Parallelism

In XPRS, intra-operation parallelism (partitioned parallelism) is implemented in two ways: *page partitioning* and *range partitioning*. In page partitioning we partition relations across disk page boundaries and assign a subset of disk pages to each participating process to work on. In range partitioning we partition relations according the value of a certain attribute. We assume that we can use the data distribution information such as histograms stored in the system catalogs to obtain a well-balanced range partition. We can also use the keys stored in the root node of a B-tree index for range partitioning. We apply either page partitioning or range partitioning to scans directly, but joins are parallelized by appropriately parallelizing its outer and/or inner paths. Details of parallelization for each type of operation are described below.

- **Sequential Scan**

  A sequential scans is parallelized through page partitioning. In XPRS, we specifically use a simple *mod* function to partition pages among processes. If the degree of parallelism is n, process i scans page x such that $x \bmod n = i$.

- **Index Scan**

  An index scan is parallelized through range partitioning. If the scan range is G, the degree of parallelism is n and the range partition is G_i, $i = 1, \ldots, n$, $(\bigcup G_i = G)$, process i scans tuple t such that $t \in G_i$.

- **Sort**

  A sort is also parallelized through range partitioning. If the entire range of the sort key is G, the degree of parallelism is n and the range partition is $G_i, i = 1, \ldots, n, (\bigcup G_i = G)$, process i sorts the tuples in G_i. In the end, the sorted segments produced by all the processes are attached together in the order of the subrange G_i's and an entire sorted relation is formed.

- **Nestloop Join**

  A nestloop join is parallelized by partitioning its outer relation and each participating process joins a portion of the outer relation with the inner relation. Suppose the outer relation is R and the inner relation is S. If the degree of parallelism is n and R is partitioned into R_i, $i = 1, \ldots, n$, process i performs join $R_i \bowtie S$.

- **Mergejoin**

  A mergejoin can be parallelized similarly to a sort through range partitioning.

- **Hashjoin**

  Parallelization of a hashjoin requires a segment of shared memory to store the shared hash table and both inputs to the hashjoin properly partitioned among the participating processes. Suppose that the hashjoin is $R \bowtie S$ with R as the smaller relation and R can fit into the available memory. Let the degree of parallelism be n, the partition of R be R_i, $i = 1, \ldots, n$ and the partition of S be S_i, $i = 1, \ldots, n$. A hash table for R is allocated in the shared memory. In the build phase, process i inserts tuples in R_i into the shared hash table, and in the probe phase, process i uses tuples in S_i to probe the hash table of R and outputs the matched result tuples. If R cannot fit into the available memory, the standard hybrid hashjoin [48] algorithm can be employed. The same parallelization as we just described can be applied to each batch of a hybrid hashjoin. Note that there may be access conflicts to the shared hash table among the parallel processes. However, the access conflicts only occur in the build phase when two processes try to insert tuples into the same hash bucket at the same time. Therefore, to minimize the probability of conflict, the number of hash buckets should be made large compared to the degree of parallelism. We also use hardware spin locks for synchronization to make the overhead negligible.

2.2.2 Performance of Intra-operation Parallelism

DeWitt, et al. has shown in [16] that intra-operation parallelism can achieve near-linear speedup in response time in a shared nothing environments. We briefly show the same near-linear speedup in XPRS in a shared memory environment. In addition, we show performance varying the number of disks and number of processes independently and also the degradation resulting from excessive parallelism.

In the experiments, we created the two 10,000-tuple relations from the Wisconsin Benchmark, *tenk1* and *tenk2*. Because the tuple sizes in the Wisconsin Benchmark relations are relatively small, we also created another 10,000-tuple relation, *ltenk1*, which has the same fields as tenk1 except that each tuple is filled to 1,000 bytes by an extra string field. Appropriate indices were created according to the benchmark specification. All the relations are striped across a set of disks using a simple *mod* function, i.e., block x, is stored on disk (x mod $\#disks$). For example, if we only use two disks, then all the odd number blocks will be on one disk and all the even number blocks will be on the other. The relations are also partitioned among the parallel scan processes with a simple *mod* function, i.e., process i scans block x such that x mod $\#processes = i$. Before each execution, the file system cache is always cleared so that no blocks of the test relations are left in memory. All the processes are pre-forked so that process startup overhead is negligible. As mentioned before, our prototype runs on a Sequent Symmetry system running Dynix operating system with 12 processors and 7 disks.

We have measured the speedup of parallel scans and joins on the above relations varying the number of processes and number of disks. We present sample results of the

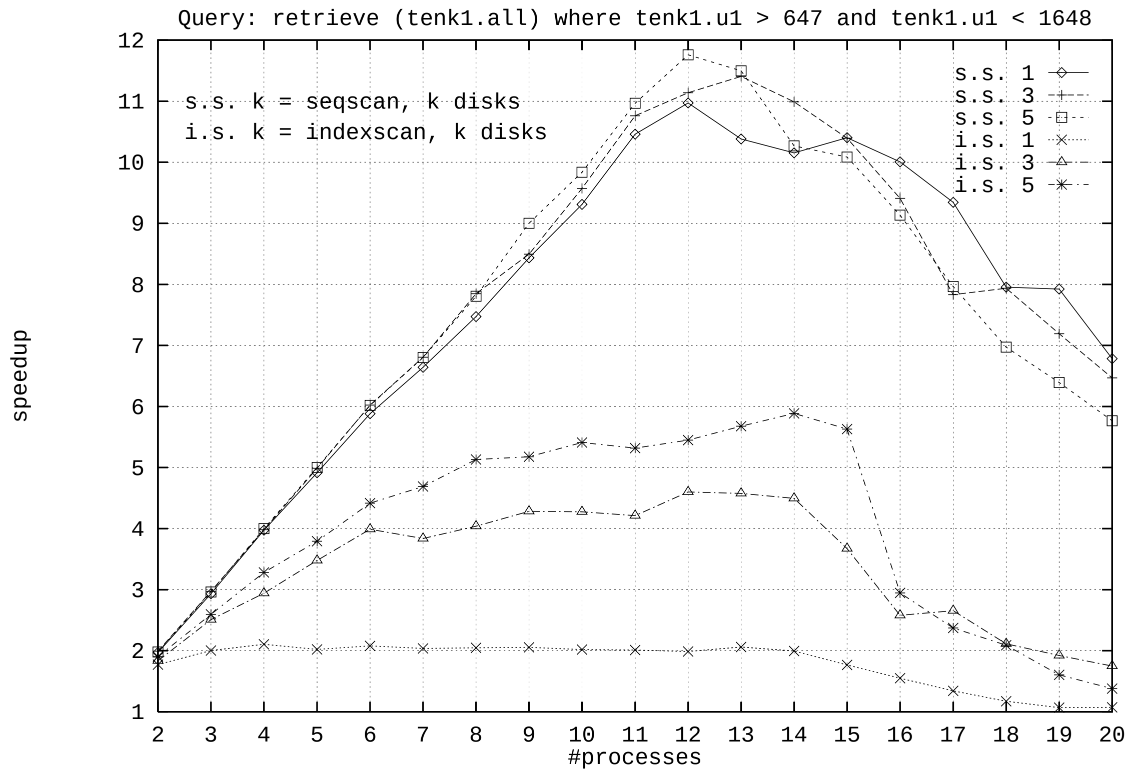

Figure 2.3: **Speedup of Parallel Scan: small tup.**

speedup measurements in Figure 2.3 - 2.5. Figure 2.3 and Figure 2.4 show speedup of scans, while Figure 2.5 shows speedup of joins. We have found that a sequential scan is CPU-bound when the tuples are small and becomes I/O-bound when the tuples get large. Moreover, index scans are I/O-bound because they do not need to examine every tuple in a page. Our experiment results show that parallel scans and joins can achieve close-to-linear speedup until they run out of processors if they are CPU-bound such as the sequential scan in Figure 2.3 and the join in Figure 2.5, or disk bandwidth if they are I/O-bound such as the index scan in Figure 2.3 and the sequential scan in Figure 2.4. Figure 2.5 also shows that the synchronization overhead caused by the shared hash table in hasjoins is negligible.

In Figure 2.4, we see a drop in the speedup when the number of processes exceeds

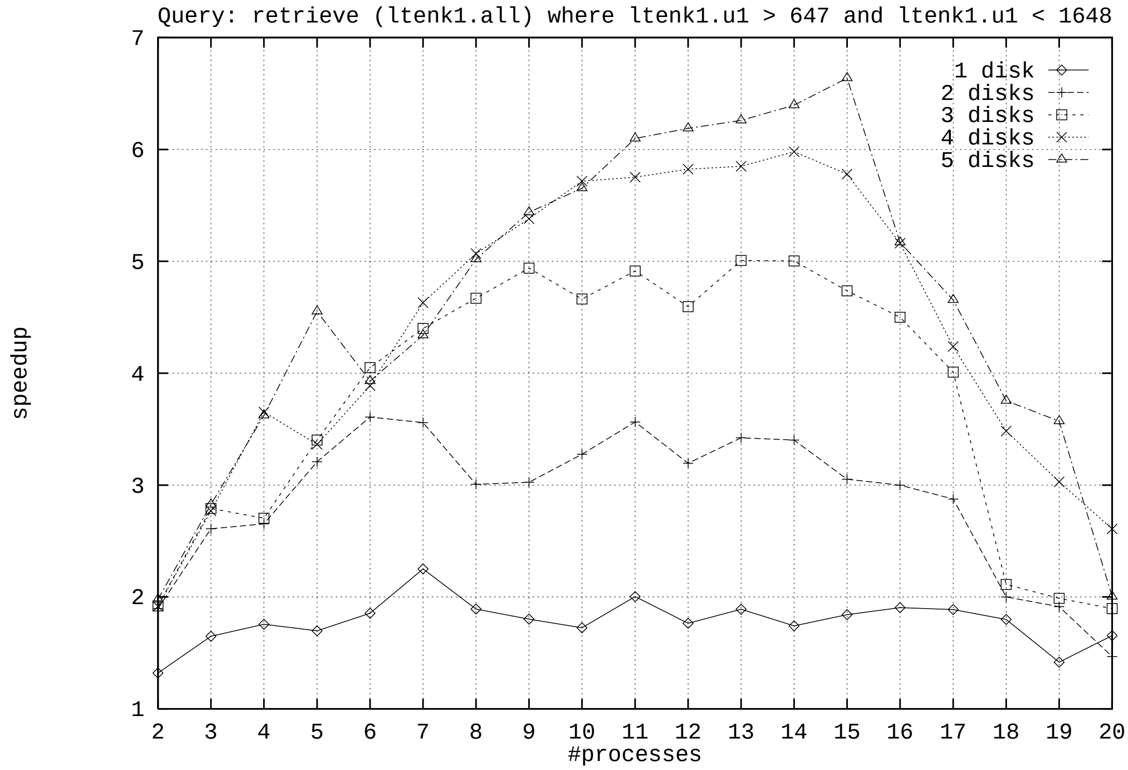

Figure 2.4: **Speedup of Parallel Seq. Scan: large tuple**

the number of disks. This is caused by the operating system read-ahead. When we have fewer processes than disks, the access pattern on each disk is sequential. Consequently normal file system read-ahead will prefetch the next disk block to be processed. When there are more processes than disks, the access pattern to each disk becomes random and the file system read-ahead is ineffective.

Observe that there is always a performance drop when the degree of parallelism exceeds the number of processors. It results from the extra context switches and virtual memory overhead generated when the number of processes exceeds the number of processors. In addition, a process holding the shared buffer pool spin lock might be descheduled and the convoy problem[5] will occur. In a multi-user environment there will be multiple commands

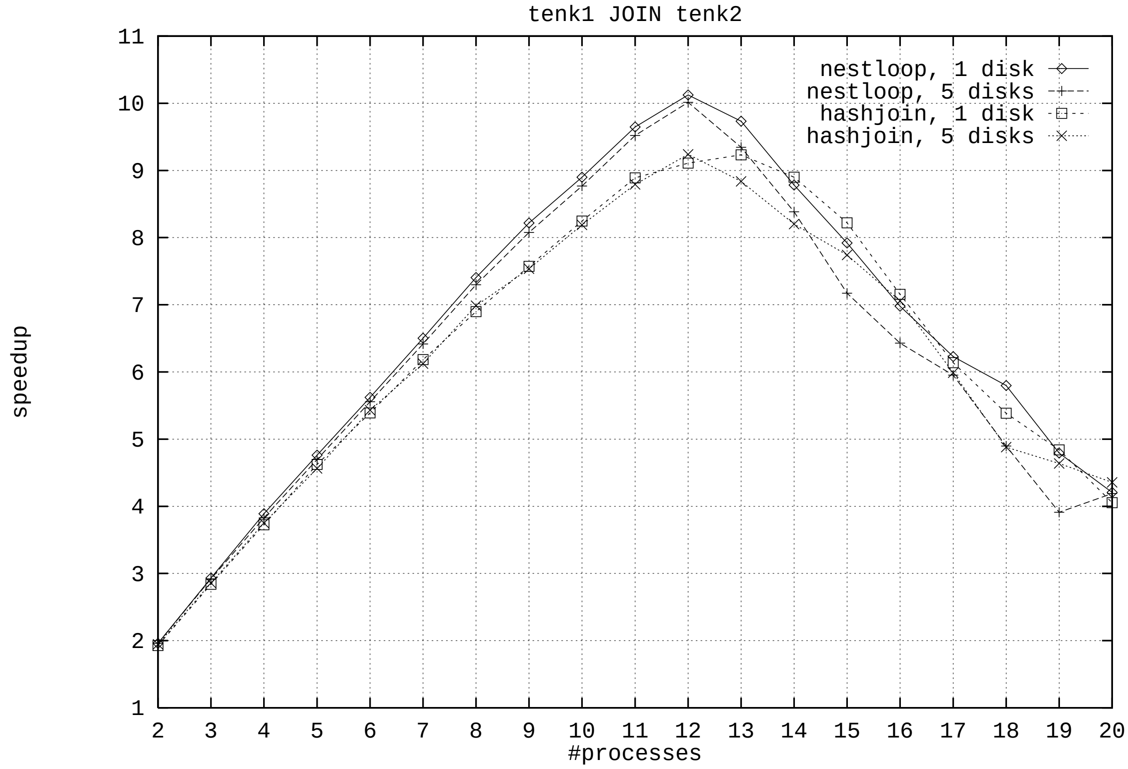

Figure 2.5: **Speedup of Parallel Join: small tuples**

concurrently being processed, it is important to ensure that the total degree of parallelism does not exceed the number of processors.

2.2.3 Architecture of Parallel Query Processing in XPRS

The previous subsection has introduced the parallelizations of single operations and their performance in XPRS. Based on those results, this subsection describes the parallelizations of complete query plans consisting of multiple operations. Figure 2.6 gives an overall architecture of XPRS query processing, which consists of a master backend process and multiple slave backend processes. The master backend is responsible for the optimization, scheduling and coordination, while the slave backends execute in parallel whatever plan

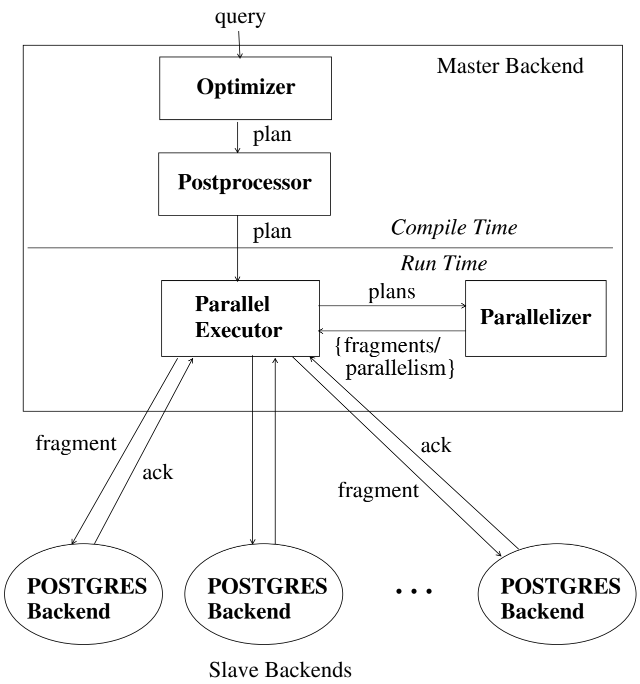

Figure 2.6: **Architecture of XPRS Query Processing**

fragments assigned to them by the master backends. The XPRS optimizer is a modified version of the POSTGRES optimizer that performs a System R style exhaustive search [47] to find the optimal sequential plan. It is also capable of enumerating bushy tree plans if a special flag is set. There is also a postprocessor that uses cost functions similar to those in [48] and [34] to determine if two join or scan methods (specifically sequential scan versus unclustered index scan and nestloop with index versus hashjoin) may have "cross over" points in their cost curves against buffer size. If such points exist, the postprocessor will modify the optimal plan generated by the optimizer by adding appropriate *choose* nodes, which completes the compile-time optimization, i.e., the first phase of our two phase optimization.

At run time, the parallel executor passes a sequential plan to the parallelizer. The parallelizer performs the second phase of our two phase optimization, i.e., to find the optimal parallelization for a selected sequential plan. It decomposes a sequential plan into a set of plan fragments, decides a processing schedule for all the plan fragments and assigns a degree of parallelism to each plan fragment that is to be executed, then passes a set of selected plan fragments along with their parallelism back to the parallel executor. The parallel executor then distributes the plan fragments to the slave backend processes according to the degree of parallelism of each fragments. The slave backends execute the plan fragments in parallel and send an acknowledgement back to the parallel executor after they finish executing. The parallel executor will then ask the parallelizer for more plan fragments to execute. The whole processes repeats until all the plan fragments are finished.

Since the XPRS parallelizer is the key component for exploiting parallelism, we will describe it in more details from the following three aspects.

- **Exploiting Different Forms of Parallelism**

  We have decided to only consider intra-operation parallelism and parallelism between independent operations. Pipelining between operations is always achieved within the same plan fragment so that operations can call each other with simple procedure calls and do not have to pay the overhead of inter-process communication. Therefore in XPRS, inter-operation parallelism does not include pipelined parallelism.

- **Choice of Plan Fragments**

  Since we want to execute each plan fragment in a pipelined fashion, blocking between any two operations in a plan fragment is not allowed. Blocking between two operations occurs when one of the operations must wait for the other operation to finish producing all the tuples before it can proceed. The common examples of blocking are introduced

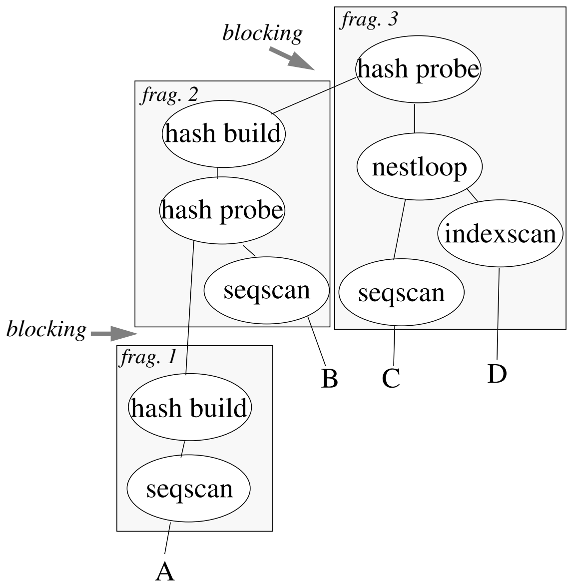

Figure 2.7: **Initial Plan Fragments**

by either hashing or sorting. For example, between the two phases of a hashjoin, the probe phase cannot start until the build phase finish constructing the entire hash table. To ensure that there are no blocking edges in any plan fragments, the parallelizer initially decomposes a sequential plan into a set of plan fragments across blocking boundaries, which is the set of largest pipelineable subgraphs. See Figure 2.7 for an example of this initial decomposition.

Obviously larger plan fragments will cause less temporary relation overhead because temporary relations can be avoided by pipelining within a plan fragment. However, the size of plan fragments is also constrained by the run-time available memory size. For example, a plan fragment can contain as many as two hashjoins, i.e., one hash probe node followed by a hash build node. If the XPRS parallelizer cannot obtain the minimum required memory for both hashjoins, it will decompose the plan fragment into two smaller fragments between the hash probe node and the hash build node, save the intermediate results of the hash probe into a temporary relation and the following hash build node will read from the temporary relation after the previous hash probe is finished. Subsequent to the decomposition, we only need enough memory for one hashjoin at a time. Even if the parallelizer can allocate enough memory to satisfy the minimum memory requirements for both hashjoins, it still has to decide whether it is better to execute the two hashjoins together or separately, and if together, how to divide the memory between the two hashjoins. This is what we call the *plan fragment decomposition problem* which will be solved in Chapter 4.

- **Determining Degree of Parallelism**

Although the architecture in Figure 2.6 also allows inter-operation parallelism, we will defer the details to the next chapter. For intra-operation parallelism, the parallelizer only needs to choose one plan fragment that is ready to execute and assign it the maximum degree of parallelism constrained by the disk bandwidth and number of free processors. The performance curves in Section 2.2.2 have shown the consequences of excessive parallelism. Suppose that N is the run-time number of available processors and B is the run-time available disk bandwidth (IOs/second). The parallelizer will estimate the I/O rate of the chosen fragment, for example, C (IOs/second), and choose the degree of parallelism for the fragment as the following,

$$parallelism = min(N, B/C).$$

2.3 Verification of Hypotheses

As described in the previous section, XPRS is designed and implemented based on the hypotheses that supports the two phase optimization strategy. In this section, we present experimental evidences that verify theses hypotheses. In the experiments, we first run a wide variety of benchmark queries using all the possible execution plans under different buffer sizes and measure the real execution costs and then calculate the the relative errors that result from our hypotheses. We will show that such errors are extremely small. We first present experiment results on the buffer size independent hypothesis, then present results on the two phase hypothesis.

2.3.1 Choice of Benchmarks

We have chosen to use two benchmarks. The first benchmark is the Wisconsin benchmark [3], a standard benchmark for measuring query processing power. We have only used 10 of the 32 queries which are selections and joins. Other queries in the benchmark that involve data manipulation are not used in our experiments. Because the Wisconsin benchmark has no more than 3-way joins and has limited join selectivities, we also developed a random benchmark generator to generate additional varieties of complex queries.

The relations in our random benchmark have the following POSTQUEL definition:

```
create r (ua1=int4, a1=int4, ua20=int4,
          a20=int4, ua50=int4, a50=int4,
          ua100=int4, a100=int4,
          filling=text)
```

We generated 10 random relations with cardinalities ranging from 100 to $10,000$. All the integer attribute values are randomly distributed between 0 and $9,999$. All the "ua" attributes are unclustered attributes and all the "a" attributes are clustered attributes. The number following "ua" or "a" indicates the number of times each value is repeated in the attribute. For example, each value in "ua20" is duplicated 20 times. We define indices on all the integer attributes so that the optimizer can always have the choice of generating an indexscan. The attribute "filling" is a variable length string, and is used to vary the tuple size of different relations. The generator can make the "filling" attribute 68 bytes or 968 bytes so that resulting tuple size is 100 bytes or $1,000$ bytes so that there can be a mixture of IO-bound and CPU-bound queries.

The queries in the random benchmark are generated in the following way. To generate a random join of k relations, we first randomly choose k relations. Then we start with the first relation in the chosen list and the rest in the unchosen list. We randomly pick a relation in the unchosen list, join it with a randomly picked relation in the chosen list on two randomly chosen attributes and move it from the unchosen list to the chosen list. We repeat this operation until the unchosen list becomes empty; and we have generated a random join on k relations. Next we generate random selections on the relations. Each relation has a 50% probability of having a restriction of either an equality condition or inequality conditions with a lower bound and a upper bound. The target list is also randomly selected from all the attributes of all the relations with each attribute having 50% probability of being chosen.

2.3.2 Experiments on the Buffer Size Independent Hypothesis

In the experiments, we first have the XPRS query optimizer generate all possible sequential plans for each query. The XPRS optimizer is capable of generating all possible plans in the style of [47]. If a plan is known to be dominated by another plan, it is discarded. For example, since we know that hashjoin is the best join plan without the use indices (assuming that our buffer size is above the hashjoin threshold) [48], we can always remove nestloop and mergejoin plans which do not use indices from the test plans, which cuts down our experiment running time substantially. Another example is that mergejoin plans with inner relation and outer relation exchanged always have the same cost and therefore only one of them need to be executed. After selecting interesting execution plans, we run each selected plan varying the buffer size from the minimum amount to make all hashjoins possible to $MAXBUFS$, the buffer size that can hold all the relations in main memory. For each execution we measure the actual execution cost. To avoid the problem of inaccurate estimate of intermediate result sizes to hashjoin, for a given join order, we always execute the plans that do not contain hashjoins first. Size information of intermediate results is collected during the execution of these plans. In this way, we can always use the the actual size of intermediate results for hashjoins. The effect of inaccurate size estimates on our results is left as a future research topic.

From the statistics collected in the experiments, we know for each query, Q, the real execution cost of each query plan under different buffer sizes, i.e., given any plan P and buffer size $NBUFS$, we know $Cost(P, NBUFS)$. Therefore, we can identify $BP(Q, NBUFS)$. Let $BCP(Q, NBUFS)$ be the plan obtained by adding appropriate *choose* nodes to plan $BP(Q, NBUFS)$. Since we know the cost of all the plans that are only different from $BP(Q, NBUFS)$ in some join or scan methods, we can also calculate the cost of $BCP(Q, NBUFS)$ under any buffer sizes. The relative error of the buffer size independent hypothesis is computed with the following formula:

$$FixBufCost = Cost(BCP(Q, MAXBUFS), NBUFS),$$

$$MinCost = Cost(BP(Q, NBUFS), NBUFS),$$

$$Error(Q, NBUFS) = \frac{FixBufCost - MinCost}{MinCost}$$

Figure 2.8 shows the graph of average relative errors of the Wisconsin benchmark queries. For each query Q, we first compute $Error(Q, NBUFS)$ for each buffer size, then we compute the average relative error over the buffer sizes. As we can see from the graph, the first few queries have 0 error. This is because those are the selection queries and two-way join queries for which the *choose* nodes can always choose the optimal plan. Errors occur in the three-way join queries, but they are never above 4%.

Figure 2.9 shows the graph of average relative error on the random benchmark. 120 queries of up to 6-way joins are executed in the experiment. The relative error is averaged over queries that have the same number of relations (20 of them each). As we can see from the graph, the average relative errors remain below 5%.

Thus, two different benchmarks support the buffer size independent hypothesis. With choose nodes we have encapsulated enough of the plan switches required when buffer size changes so that average relative error is never above 5%.

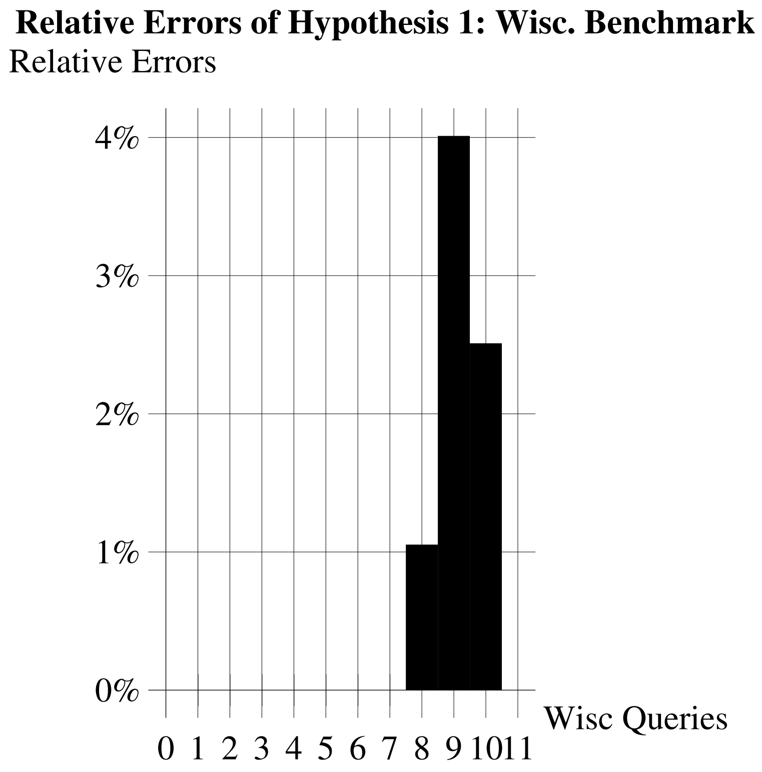

Figure 2.8: **Relative Errors of the Buffer Size Independent Hypothesis on the Wisconsin Benchmark**

We have also studied the detailed statistics from our experiments to find out the cause of these errors. We found that these errors are caused by plans with similar costs but different join orders and thus different I/O access patterns. For example, the plan $P_1 : (A \bowtie B) \bowtie C$ may have a similar cost as the plan $P_2 : (B \bowtie C) \bowtie A$ when the buffer size is fixed to $MAXBUFS$, but they have different I/O access patterns. Let $H_P(B)$ stand for the buffer hit rate of plan P with buffer size B. Suppose that P_1 is the optimal plan with buffer size $MAXBUF$. Because H_{P_1} and H_{P_2} are different functions, it is possible for H_{P_1} to decrease faster than H_{P_2} as B decreases. Therefore, P_1 may become more expensive than P_2 at some buffer size, thus violate the hypothesis. However, since this only happens between plans with similar costs to start with, the errors are very small, as shown by our

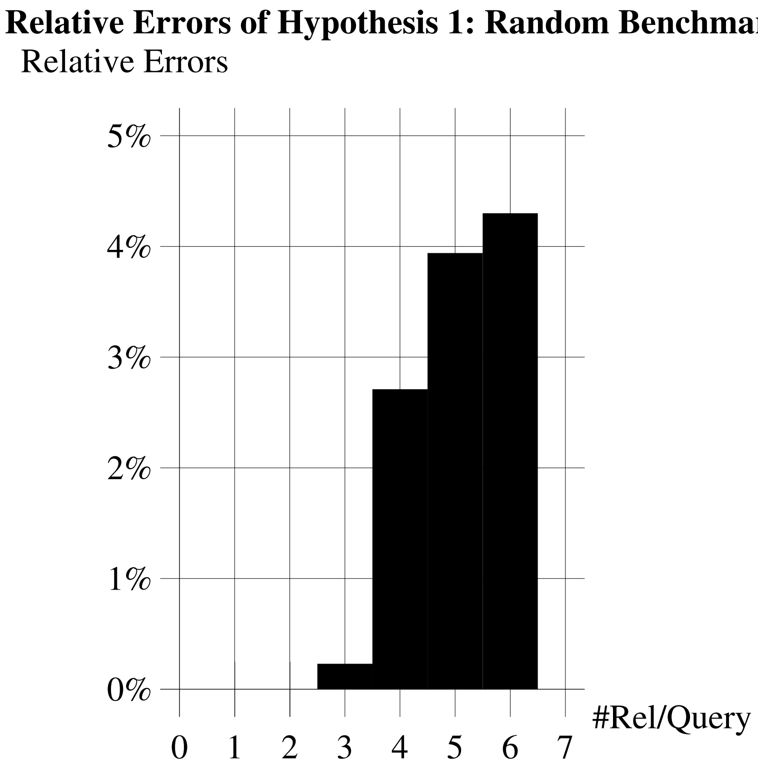

Figure 2.9: **Relative Errors of the Buffer Size Independent Hypothesis on the Random Benchmark**

experiment results.

2.3.3 Experiments on the Two-Phase Hypothesis

The experiments that we run to justify the two phase hypothesis are similar to those described in the previous subsection. We have also used the queries from the Wisconsin benchmark and the random benchmark. For each test query, we first generate all the possible sequential plans as described in the previous subsection. We ran each plan with varying degrees of intra-operation parallelism and buffer sizes. The degree of parallelism is varied from 1 to 12 (the number of processors in our system) and the buffer size is varied in the same way as in the previous subsection. For each execution of a plan, P, with $NBUFS$ buffers and $NPROCS$ processes, we measure the resource consumption and response time, and compute the cost of the execution, $Cost(P, NBUFS, NPROCS)$. Let $BP(Q, NBUFS)$ be the best sequential plan with buffer size $NBUFS$. The relative error of the two phase hypothesis is calculated with the following formula:

$$TwoPhaseCost = Cost(BP(Q, NBUFS), NBUFS, NPROCS),$$

$$MinCost = min_{\{P\}}\ Cost(P, NBUFS, NPROCS),$$

$$Error(Q, NBUFS, NPROCS) = \frac{TwoPhaseCost - MinCost}{MinCost}$$

Because the cost of parallel plans depends on the system-specific weighting factor, w, we need to calculate errors for different values w.

Figure 2.10 shows the average relative error of the Wisconsin benchmark queries. For each query, Q, the relative error, $Error(Q, NBUFS, NPROCS)$, is averaged over all combinations of $NBUFS$ and $NPROCS$. As we can see, for small values of w, (which means that we mostly optimize resource consumption,) the relative errors are near 0. This is because the parallelization of the optimal sequential plan also has the minimum resource consumption. For large values of w, (which means that we mostly optimize response time,) the relative error never exceeds 8%. This indicates that the parallelization of the optimal sequential plan may not always have the minimum response time, but it is close to the minimum.

Figure 2.11 shows the average relative error of the random benchmark. Due to the enormous computing resources demands of this experiment, we have only run queries

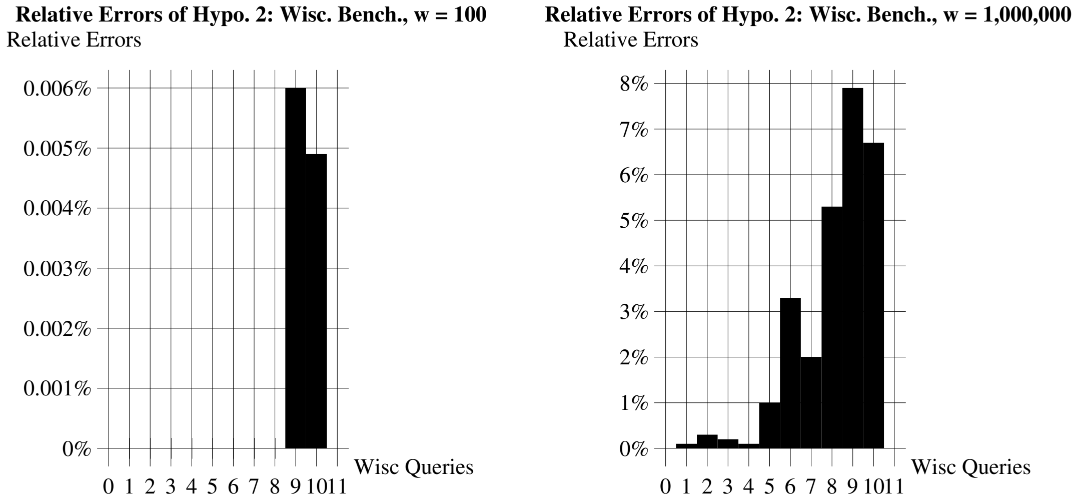

Figure 2.10: **Relative Errors of the Two-phase Hypothesis on the Wisconsin Benchmark**

of up to 5-way joins from the random benchmark. As we can see, the average relative error never exceeds 6%.

By looking into the detailed statistics in our experiments, we have discovered that errors only occur when there are two plans that have very close sequential costs but are parallelized in different ways. For example, for a one-variable query with certain selectivities, the cost of the index scan plan can be approximately the same as the cost of the sequential scan plan (i.e., the sum of the number of index pages and the number of selected data pages is about the same as the total number of data pages). However, index scans are parallelized through range partitioning while sequential scans are parallelized through page partitioning, which is easier to guarantee load balance. Therefore, even though the sequential execution of the sequential scan plan may be slightly slower than the sequential execution of the index scan plan, the sequential scan plan may get better speedup when parallelized. Thus, the parallel version of the sequential scan plan may run faster than the parallel version of the

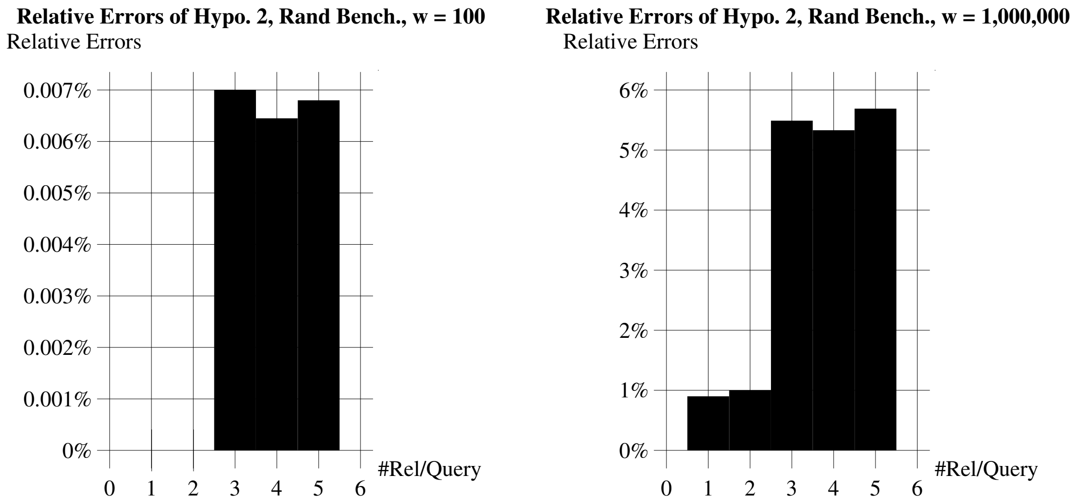

Figure 2.11: **Relative Errors of the Two-phase Hypothesis on the Random Benchmark**

index scan plan because of their different ways of parallelization, which violates the two phase hypothesis if we optimize response time. Our experiment results show that such errors are very small because each plan can achieve a near-linear speedup when parallelized through intra-operation parallelism and these plans have close sequential costs to start with. Although we have only validated this for queries with a small number of relations due to limited computational resources, we believe that this result is also true for queries with a large number of relations.

2.4 Summary

In this chapter, we have described our approach to the optimization of parallel query execution plans in a shared everything environment. We have demonstrated that we can achieve near-linear speedup with intra-operation parallelism in XPRS and there is severe performance penalty for excessive parallelism. We have presented experiment results that justify our two phase optimization approach, which reduces the complexity of parallel query optimization without significantly compromising optimality of the resulting parallel plan. The first phase of of our two phase optimization is a conventional query optimization with a fixed buffer size that finds the optimal sequential plan augmented with appropriate *choose* nodes that are encapsulated in individual operations. The second phase finds the best parallelization of the sequential plan obtained from the first phase according to run-time enironment. Our approach achieves run-time flexibility while still doing most of the optimization at compile time.

In the experiments presented this chapter, we have assumed perfect estimates of intermediate result sizes and uniform data distribution. The effects of inaccurate size estimates and skewed data distributions also need to be studied more carefully. This chapter has only considered intra-operation parallelism. Issues involving inter-operation parallelism will be discussed in the next chapter.

Chapter 3

Parallel Task Scheduling

The previous chapter has only considered intra-operation parallelism. However, intra-operation alone cannot guarantee maximum performance. As will be shown, a combination of intra-operation parallelism and inter-operation parallelism must be exploited to achieve maximum performance. In this chapter, we will address the issues involving inter-operation parallelism, particularly the scheduling problem of parallel tasks. We will first present a clean and simple algorithm for optimally scheduling a set or a sequence of tasks (i.e., plan fragments) in parallel processing of a single query or multiple queries. The main idea is to use inter-operation parallelism to combine IO-bound and CPU-bound tasks to increase system resource utilization. The algorithm matches up IO-bound and CPU-bound tasks with appropriate degrees of intra-operation parallelism to make both the processors and the disks operate as close to their full utilizations as possible and thus to minimize the elapsed time. Moreover, a complex packing problem in the optimization of task schedules is avoided by a mechanism to dynamically adjust the degree of intra-operation parallelism of a running task to keep the system operating at the maximum utilization point as the workload changes. The previous chapter has only addressed the parallel query optimization problem with intra-operation parallelism. Having solved the scheduling problem, we can estimate the cost of a parallel plan with inter-operation parallelism based on our scheduling algorithm. We will extend the two phase optimization strategy presented in the previous chapter to consider inter-operation parallelism by introducing a new cost estimation method to the query optimizer.

This chapter is organized as follows. Section 3.1 first defines the scheduling problem that we are solving. Section 3.2 then describes our adaptive task scheduling algorithm including calculation of IO-CPU balance point, dynamic adjustment to intra-operation parallelism and task re-ordering heuristics. Section 3.3 then examines variations of our scheduling algorithm and compares their performances through experiment results. After the scheduling algorithm and its performance are presented, Section 3.4 extends the two-phase optimization strategy to consider inter-operation parallelism based on the scheduling algorithm, and last, this chapter is summarized in Section 3.5.

3.1 Problem Definition

Under the architecture of XPRS query processing given in Figure 2.6, plan fragments are used as the units of parallel execution. In this chapter, they are also called *tasks* for scheduling purposes. By inter-operation parallelism, we, in fact, mean *inter-fragment* or *inter-task* parallelism. At any give time, there may well be multiple plan fragments (i.e., tasks) that are ready to run, such as the left subtree and the right subtree of a bushy tree plan, or in a multi-user environment, plan fragments from queries simultaneously submitted by different users. As we can see in Figure 2.6, XPRS can also accept multiple user queries at the same time and try to execute plan fragments from different queries in parallel. This greatly increases the opportunity for inter-operation parallelism.

When there are multiple plan fragments that are all ready to run, it becomes the XPRS parallelizer's responsibility to schedule these tasks optimally. The following is the scheduling problem that needs to be solved for the XPRS parallelizer.

Given n ($n = 1, 2, \ldots, \infty$) *runable plan fragments (tasks),* $f_1, f_2, \ldots, f_n$, *where the plan fragments may be from a bushy tree plan of the same query or from different queries that are simultaneously submitted,*

1. decide a processing schedule for $f_1, f_2, \ldots, f_n$*;*

2. choose a degree of parallelism for each f_i*,*

such that the total elapsed time of processing $f_1, f_2, \ldots, f_n$ *is minimized.*

As we will see, our solution to the above described problem works for both a fixed set of tasks or a sequence of tasks. Namely, we also allow n to be infinity in the above description for online scheduling.

Some researchers have tried to model this problem as a modified version of the bin packing problem or the multi-processor scheduling problem in combinatorial optimization as in [38]. For example, each task can be represented as a rectangle with the width of each rectangle as the elapsed time of the task and the height of each rectangle as the CPU usage of the task. The problem becomes to pack the set of rectangles as tightly as possible into a horizontal stripe as shown in Figure 3.1. There are two problems with this approach.

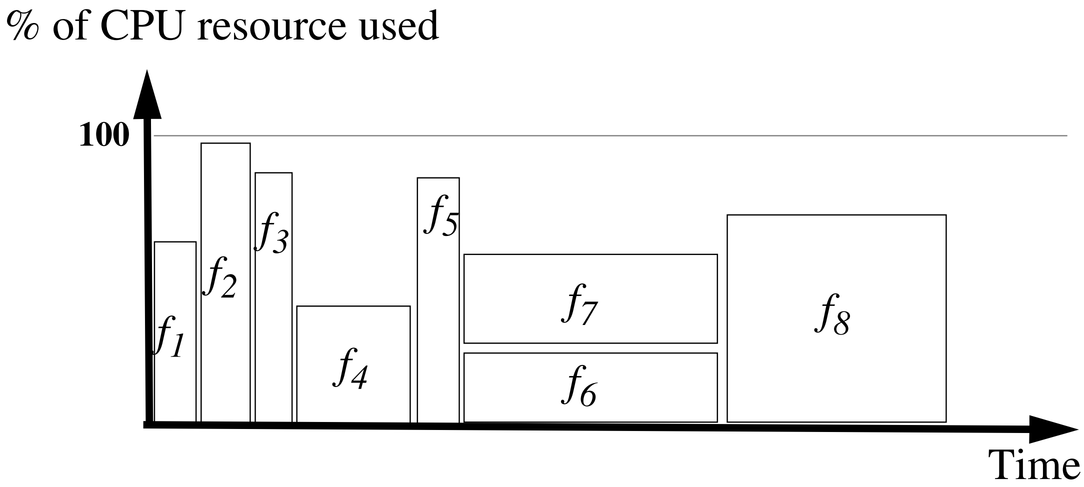

Figure 3.1: **A Bin Packing Formulation of the Scheduling Problem**

First, given the NP complete complexity of the bin packing problem, this formulation of the task scheduling problem does not seem to be promising. Second, this approach does not exactly model the problem we are solving. In our problem, we considers not only parallelism between different tasks but also parallelism within each tasks. When we adjust the internal parallelism of each task, the shape of the corresponding rectangle changes. Therefore, this approach does not work for our problem. We decide to take a completely different approach for XPRS which will be described in the next section.

3.2 Adaptive Scheduling Algorithm for XPRS

We present our solution to the task scheduling problem, an adaptive scheduling algorithm in this section. First, we describe our classification of IO-bound and CPU-bound tasks and the calculation of the IO-CPU balance point. Then, a mechanism for dynamic adjustment of parallelism is presented. Last, we integrate our ideas into an adaptive scheduling algorithm.

3.2.1 IO-bound and CPU-bound tasks

The key idea of our solution to the task scheduling problem is to combine IO-bound and CPU-bound tasks through inter-operation parallelism so that the utilization of both the processors and the disks is maximized, and thus the elapsed time is minimized. Before we describe our solution, we need to define our classification of IO-bound and CPU-bound tasks.

Let C_i (IOs/second) be the rate of I/O requests of task f_i if processed sequentially. When f_i is executed with parallelism x, its I/O rate becomes

$$IO_i(x) = C_i \times x.$$

Suppose that the total disk I/O bandwidth is B (IOs/second) and the total number of processors is N.

We define task f_i as *IO-bound* if $C_i > B/N$ and *CPU-bound* if otherwise. Obviously, the function $y = IO_i(x)$ is a straight line with slope C_i. If we draw the line with the rectangle bounded by B and N as in Figure 3.2, we can see that IO-bound tasks are those corresponding to the lines above the diagonal and CPU-bound tasks are those corresponding to the lines below the diagonal.

Because the disk page size and tuple sizes are known, we can estimate how many tuples of a relation can fit into one disk block. We can also how much CPU time will be spent on each tuple for different database operations. Therefore, we can estimate the I/O rate, C_i and do the classification beforehand.

As we can see in Figure 3.2, the parallelism of a task is limited by the rectangle

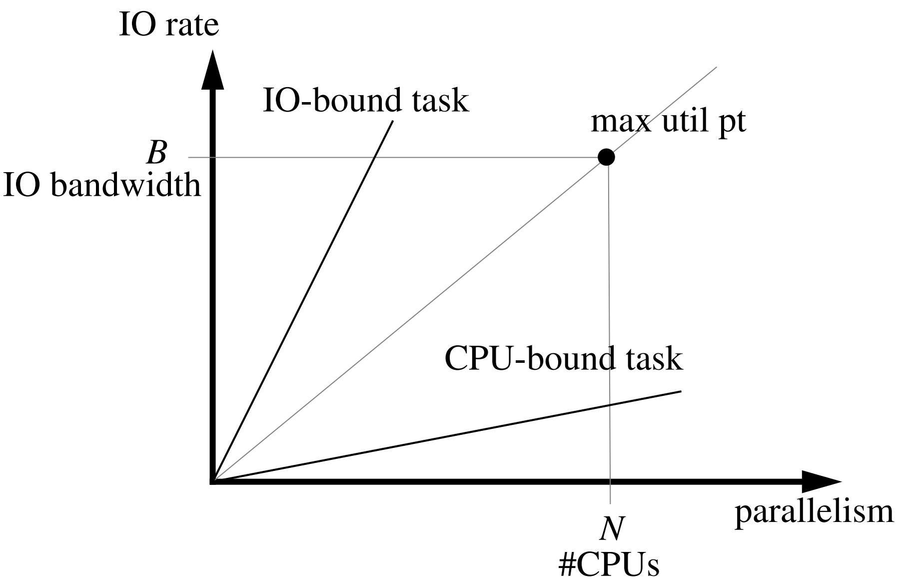

Figure 3.2: **IO-bound and CPU-bound tasks**

boundaries and the maximum parallelism of a task is achieved at the intersection between the line corresponding to the task and one of the rectangle boundaries. An IO-bound task will run out of disk bandwidth before it runs out of processors. Its maximum parallelism $maxp(f_i) = B/C_i$. On the other hand, the parallelism of a CPU-bound task is only bounded by the number of processors N, i.e., $maxp(f_i) = N$. In general,

$$maxp(f_i) = min(B/C_i, N).$$

3.2.2 Calculation of IO-CPU Balance Point

Intuitively, in order to maximize the utilization of both the disks and processors, we want the system to be running at the upper right corner of the rectangle in Figure 3.2, i.e., the point with coordinate (N, B). Obviously, if we run one task at a time using only intra-operation parallelism, unless the line corresponding to a task is exactly the diagonal line, the system will not be running at the upper right corner of the rectangle. Either some I/O bandwidth or some processors may be wasted.

When we run two tasks f_i with parallelism x_i and f_j with parallelism x_j together, the system is running at the point with coordinate $(x_i + x_j, C_i x_i + C_j x_j)$. We can maximize the system resource utilization by choosing x_i and x_j according to the following equations:

$$\begin{cases} x_i + x_j & = & N \\ C_i x_i + C_j x_j & = & B \end{cases}$$

By solving the above equations, we get,

$$\begin{cases} x_i & = & (B - C_j N)/(C_i - C_j) \\ x_j & = & (C_i N - B)/(C_i - C_j). \end{cases}$$

We call (x_i, x_j) the *IO-CPU balance point* for tasks f_i and f_j.

Suppose that $C_i > C_j$, in order to make x_i and x_j both positive, we must have $C_i > B/N$ and $C_j < B/N$. In other words, valid solutions exist for the above equations if and only if one task is IO-bound and the other CPU-bound. This formula also tells us that one IO-bound task plus one CPU-bound task can always achieve maximum system resource utilization with appropriate parallelism assignment. Although a combination of more than two tasks may also achieve the same effect, it complicates the scheduling algorithm and consumes more memory. Therefore, in exploiting inter-operation parallelism, it is sufficient to only run two tasks at a time. In other words, we never need to run more than two tasks

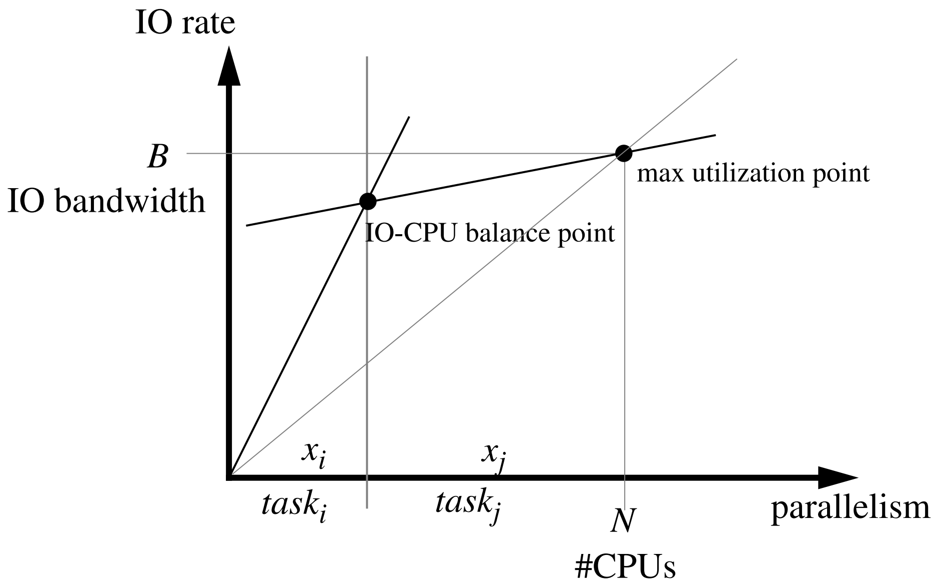

Figure 3.3: **IO-CPU Balance Point**

in parallel. This result significantly simplifies the scheduling problem.

Figure 3.3 gives a graphical interpretation to the above analysis. If we draw the line corresponding to an IO-bound task $task_i$ through the origin and the line corresponding to a CPU-bound task $task_j$ through the upper right corner of the rectangle, there will always be an intersection within the rectangle between the two lines. This intersection is the IO-CPU balance point. We can see that if we run $task_i$ with parallelism x_i and $task_j$ with parallelism x_j, the system will be running at the maximum utilization point.

The above calculation assumes that the disk bandwidth B is a predefined constant. However, in reality, disks have two bandwidths, i.e., a sequential I/O bandwidth and a random I/O bandwidth, where the random I/O bandwidth is only about 1/3 of the sequential I/O bandwidth. We have to take this into account to do a more careful calculation for parallelism involving sequential I/O tasks.

We first consider two sequential I/O tasks, f_i and f_j. Let B_s be the sequential I/O bandwidth and B_r be the random I/O bandwidth. We can compute the effective bandwidth B from B_s and B_r. There are the following two cases depending on the disk block size.

- **Large Block Size**

  If we use track-size blocks, even though there may be seeks between the two tasks, each task still sees a sequential bandwidth. Therefore, $B = B_s$.

- **Small Block Size**

  Normally, file systems use much smaller block sizes, for example, 8K bytes in XPRS. In this situation, the effective bandwidth will be less than the sequential bandwidth because of contention on the disk arms. We use a simple linear interpolation to estimate B. There are two boundary cases. The best case is that the disks spend most of their time handling I/O requests from one task and only occasionally seek to the other task, thus $B \approx B_s$. The worst case is that the disks spend half the time on one task and half on the other seeking between the two tasks, thus $B \approx B_r$. Note that the ratio of I/O time is given by $C_i x_i / C_j x_j$, or $C_j x_j / C_i x_i$. Therefore we can calculate B as below,

$$B = \begin{cases} B_r + (1 - C_i x_i / C_j x_j)(B_s - B_r) & \text{if } C_i x_i \le C_j x_j \\ B_r + (1 - C_j x_j / C_i x_i)(B_s - B_r) & \text{otherwise.} \end{cases}$$

Similarly, we can also compute the effective bandwidth for a sequential I/O task and a random I/O task running in parallel.

Because of the possible loss of the sequential bandwidth in inter-operation parallelism, it may be more efficient to run two sequential I/O tasks separately using only intra-operation parallelism even though one task is IO-bound and the other is CPU-bound. For example, we have run a small experiment on XPRS using two sequential scans making sure that one scan is CPU-bound and the other is IO-bound. When they are run separately using intra-operation parallelism only, the total elapsed time is 45 seconds. However, when they are run together using inter-operation parallelism at their IO-CPU balance point, the total elapsed time becomes 65 seconds.

Therefore, it is not always better to run an IO-bound task and a CPU-bound task in parallel if the tasks generate sequential I/O's. In our scheduling algorithm to be described in Section 3.2.4, we compare the estimated time of executing two sequential I/O tasks in parallel and the estimated time of executing them separately to decide whether inter-operation parallelism is worthwhile.

3.2.3 Dynamic Adjustment of Parallelism

Even though we have known how to calculate the IO-CPU balance point given an IO-bound task and a CPU-bound task, we have not yet solved the scheduling problem. Running two tasks at their IO-CPU balance point only guarantees full resource utilization while both tasks are running. When one task finishes first and there is no other task to fill in the newly available resources, resources are still wasted. An execution order of all the tasks that minimizes resource waste need to be found. This is still the complex combinatorial optimization problem discussed before. In XPRS, this combinatorial optimization problem is avoided by a mechanism of dynamic adjustment of intra-operation parallelism taking advantage of the low communication overhead feature in a shared memory environment.

As described in Chapter 2, in XPRS, intra-operation parallelism is implemented in two ways, *page partitioning* and *range partitioning*. In page partitioning, we partition relations across disk page boundaries and assign a subset of disk pages to each participating processors to work on. Specifically, given n processors, processor i processes disk pages $\{p \mid p\ mod\ n = i\}$, where $i = 0, 1, \ldots, n-1$. For example, if we have 3 processors, then processor 0 scans pages $\{0, 3, 6, \ldots\}$, processor 1 scans pages $\{1, 4, 7, \ldots\}$, and processor 2 scans pages $\{2, 5, 8, \ldots\}$. Obviously, using our page partitioning scheme, unless the number of processors that we use is exactly the same as the number of disks in the disk array, the access to each disk is not exactly sequential because of the asynchronousness of the parallel backends. However, assuming all the processors proceed at about the same speed, the page in one disk request is always very close to the page in the previous request to the same disk. Therefore, the seek distances are likely to be very small, and we can still get a close-to-sequential bandwidth which is still much higher than the random bandwidth.

In range partitioning, relations are partitioned according to the value of a certain attribute. For example, suppose that the employee relation is to be partitioned between 2 processors. We may have one processor work on tuples with $emp.salary > 30000$ and the other processor work on tuples with $emp.salary \leq 30000$. We can try to find a balanced range partition with data distribution information in the system catalog or in the root node of an index.

Page partitioning is used for sequential scans while range partitioning is used for index scans. Joins are parallelized using either page partitioning or range partitioning depending on the type of scans in their inner and outer plans. Different parallelism adjustment

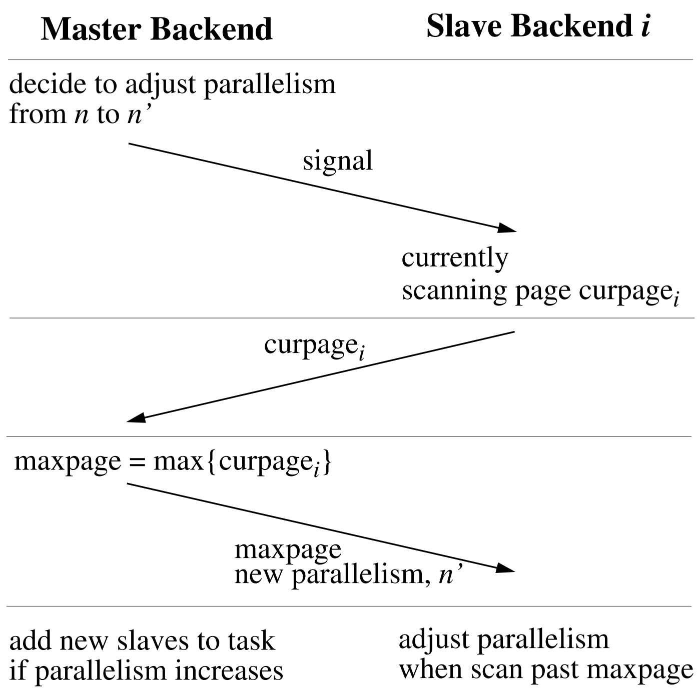

Figure 3.4: **Page Partitioning Parallelism Adjustment**

mechanisms have been designed for page partitioning and range partitioning operations.

Suppose that the current degree of parallelism for a task is n and we want to adjust it to a new degree of n', where n' can be greater than n, which means that we are putting in more available processors to work on this task, or smaller than n, which means that we are taking some processors away from this task to work on another task. We have implemented dynamic parallelism adjustment in XPRS as follows:

- For Page Partitioning

  The master backend sends a signal to all participating slave backends on the task. Upon receiving the signal, each slave backend, $i = 0, 1, \ldots, n-1$, sends back the current page number $curpage_i$ that it is scanning and then waits for a message from the master backend. After receiving all the page numbers from the slave backends, the master backend computes the maximum page number,

$$maxpage = max\{curpage_i\},\ i = 0, 1, \ldots, n-1.$$

Then the master backend sends $maxpage$ and new parallelism n' to all the slave backends, which completes the communications between the master and the slaves for the parallelism adjustment. If $n' > n$, the master backend will start $n' - n$ free slave backends to work on the task and make each of them start scanning after page $maxpage$. After receiving $maxpage$ and n', all the slave backends will resume their work until they finish scanning all the pages before $maxpage$, at which point, they will change from scanning every nth page to scanning every n'th page and complete the parallelism adjustment. If $n > n'$, upon scanning past $maxpage$, the slave backends i, $i >= n' - 1$ will finish processing the current task and report back to the master backend as available. The communication between the master backend and the slave backends for page partitioning parallelism adjustment is shown in Figure 3.4.

- For Range Partitioning

The master backend sends a signal to all the participating slave backends on the task. Upon receiving the signal, each slave backend sends back the intervals of values that remain for them to scan. For example, if a slave backend is assigned to scan for values in interval $[l, h]$ and the current value that is being examined is c, the interval that will be sent back to the master backend is $[c, h]$. After parallelism adjustment, each slave backend may get more than one intervals to scan instead of only one contiguous interval. Upon receiving all the intervals from the slave backends, the master backend

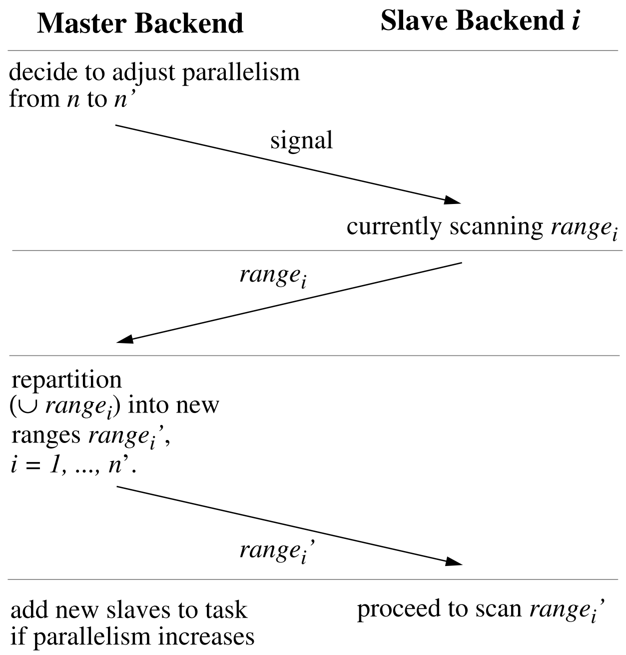

Figure 3.5: **Range Partitioning Parallelism Adjustment**

redistributes the intervals among n' slave backends and sends each slave backend a set of repartitioned search intervals. If $n' > n$, the master backend will start $n' - n$ free slave backends to work on the task. Meanwhile, the old slave backends will resume their processing with the new search intervals. If $n' < n$, the extra slave backends will finish the processing of the current task and report back to the master backend as available. Figure 3.5 shows the communications between the master backend and the slave backends for range partitioning parallelism adjustment.

As we can see, the efficiency of our parallelism adjustment mechanism depends on the low communication delay advantage of a shared memory system.

Our dynamic parallelism adjustment mechanism is not only useful for solving the scheduling algorithm, it is also useful for correcting optimizer cost estimation errors.

Because of errors in cost estimation models, the optimizer may label an IO-bound task CPU-bound or vice versa, which may cause performance problems. The run time system can detect this error by monitoring the I/O rate while the mislabeled task is running and adjust it to its correct parallelism.

3.2.4 Adaptive Scheduling Algorithm

The main idea of our scheduling algorithm is to use our dynamic parallelism adjustment mechanism to keep the system running at the IO-CPU balance point, i.e., at the maximum system resource utilization. At the same time, the algorithm also considers the possibility that inter-operation parallelism may be disadvantageous when sequential I/O is involved because of the disk seeks between tasks, in which case only intra-operation parallelism will be used to take advantage of the sequential disk bandwidth.

Given a set of n runable tasks, $S = \{f_1, f_2, \ldots, f_n\}$, suppose that the sequential execution time of f_i is T_i. And suppose that $T_{intra}(f_i)$ is the execution time of f_i running alone with only intra-operation parallelism and $T_{inter}(f_i, f_j)$ is the execution time of f_i and f_j running at their IO-CPU balance point (x_i, x_j) with inter-operation parallelism. We have,

$$T_{intra}(f_i) = T_i/maxp(f_i),$$

$$T_{inter}(f_i, f_j) = min(T_i/x_i, T_j/x_j) + T_{ij}/maxp_{ij}.$$

where T_{ij} is the execution time of the remaining task when either f_i or f_j finishes first and $maxp_{ij}$ is the maximum intra-operation parallelism of the remaining task. We have,

$$T_{ij} = \begin{cases} T_i - T_j x_i / x_j & \text{if } T_i/x_i > T_j/x_j \\ T_j - T_i x_j / x_i & \text{otherwise,} \end{cases}$$

$$maxp_{ij} = \begin{cases} maxp(f_i) & \text{if } T_i/x_i > T_j/x_j \\ maxp(f_j) & \text{otherwise.} \end{cases}$$

The following is the description of our algorithm.

1. Divide S into S_{io} and S_{cpu} such that $S = S_{io} \bigcup S_{cpu}$, S_{io} contains all the IO-bound tasks and S_{cpu} contains all the CPU-bound tasks.

2. Choose $task_1 \in S_{io}$ and set $S_1 = S_{io}$, or if $S_{io} = \emptyset$, choose $task_1 \in S_{cpu}$ and set $S_1 = S_{cpu}$.

3. Choose $task_2 \in S - S_1$, such that

$$T_{inter}(task_1, task_2) < T_{intra}(task_1) + T_{intra}(task_2).$$

4. If $task_2$ does not exist, run $task_1$ alone with parallelism $maxp(task_1)$ if $task_1$ is a new task, or adjust the current parallelism of $task_1$ to $maxp(task_1)$ if $task_1$ is a running task, set $S = S - \{task_1\}$[1], go to 2 when $task_1$ completes.

5. If $task_2$ exists, calculate the IO-CPU balance point between $task_1$ and $task_2$, (x_1, x_2).

[1]When we remove a task from S, we also implicitly remove the task from S_{io} and S_{cpu}.

6. Execute $task_1$ with parallelism x_1 if $task_1$ is a new task, or adjust the current parallelism of $task_1$ to x_1 if $task_1$ is a running task; Execute $task_2$ with parallelism x_2.

7. If $task_1$ finishes first while $task_2$ is still running, set $S = S - \{task_1\}$, $task_1 = task_2$, $S_1 = S - S_1$, go to 3.

8. If $task_2$ finishes first while $task_1$ is still running, set $S = S - \{task_2\}$, go to 3.

9. If $S = \emptyset$, algorithm terminates.

In the above algorithm, we try to pair up an IO-bound task and a CPU-bound task for inter-operation parallelism only if it is better than running them separately with intra-operation parallelism. If either IO-bound tasks or CPU-bound tasks run out, we will simply execute the remaining tasks with intra-operation parallelism only.

At any point, there may be more than one possible pairs of IO-bound tasks and CPU-bound tasks, f_i and f_j, that can be chosen. Our strategy is to pair up the most IO-bound task (the task with the greatest I/O rate) and the most CPU-bound task (the task with the smallest I/O rate). In this way, we can keep the system running closer to the maximum utilization point (the upper-right corner of the rectangle in Figure 3.3) when either IO-bound or CPU-bound tasks run out first, because the remaining tasks will be those corresponding to lines closer to the diagonal in Figure 3.3.

In a multi-user environment, if we want to minimize the response time of individual queries instead of the the total elapsed time, a shortest-job-first heuristic can be used, i.e., to execute the tasks from the shortest query first.

The above algorithm can be easily extended to handle a sequence of tasks $\{f_1, f_2, f_3, \ldots\}$

instead of a fixed set of tasks. In other words, the above algorithm can also be used for online scheduling. All we need to do is to represent S_{io} and S_{cpu} as queues. When a task arrives, it is put into either the S_{io} queue or the S_{cpu} queue according to its I/O rate. For our most IO-bound and most CPU-bound task first strategy, we can easily keep the tasks in the S_{io} queue in descending order of I/O rate and the tasks in the S_{cpu} queue in ascending order of I/O rate. Each time we simply take a task off the head of the task queues. The rest of the algorithm still work as described.

3.3 Evaluation of Scheduling Algorithms

In this section, we evaluate the performance of our scheduling algorithm described in the previous section through XPRS benchmark experiments. Experiments are performed on XPRS configured with 8 processors and 4 disks[2]. In the experiments, we compare the performance of the following three scheduling algorithms.

- **INTRA-ONLY** – No Inter-Operation Parallelism, i.e., executing tasks one by one using intra-operation parallelism only.

- **INTER-WITHOUT-ADJ** – Inter-Operation Parallelism without Dynamic Adjustment

  This is almost the same as the algorithm in the previous section, except that when one task finishes first, no dynamic parallelism adjustment is performed. The master backend will simply start the task that can get closest to maximum utilization point

[2] We decided not to use the full configuration of XPRS, i.e., 12 processors and 7 disks, in order to make it easier to generate IO-bound and CPU-bound tasks in our experiments.

if executed using the currently available processors in parallel with the running task.

- **INTER-WITH-ADJ** – Inter-Operation Parallelism with Dynamic Adjustment, i.e., the algorithm described in the previous section.

We run the following four workloads against each of the three algorithms:

- all IO-bound tasks,
- all CPU-bound tasks,
- extremely IO-bound tasks with extremely CPU-bound tasks,
- random-mix tasks.

Each workload consists of ten tasks. Since our algorithms only depend on the I/O rate of each task and other details of the operations in the tasks do not affect the performance. we choose all the queries to be one-variable selection queries for simplicity and ease of constructing tasks with specific I/O rates. Hence, all the tasks will be either a sequential scan or an index scan. The length of each task is randomly chosen between scanning 100 tuples and scanning $10,000$ tuples. We adjust the I/O rate of each task by varying the size of tuples that are scanned. All relations in the workloads have the same schema:

$$r_i(a = int4, b = text),$$

where attribute b is a variable-size string and is used to adjust the tuple sizes. All queries will be a selection on $r_i.a$. An unclustered index may be created on a to make index scans possible. For sequential scans, the I/O rate is determined by the tuple size. There is a fixed per-tuple overhead (evaluation of query qualifications) after each tuple is read into memory from disks. Therefore, the time between two I/O requests is equal to the time to read in a disk page plus the time to process all the tuples that reside in the read-in disk page. When the tuple size is small, many tuples can be packed into one disk page, thus it takes longer to process all the tuples in a page and the I/O rate is lower, so the task is likely to be CPU-bound. On the other hand, if the tuple size is large, the I/O rate is higher and the task is likely to be IO-bound. For index scans on an unclustered index, however, the I/O rate is always high because index scans can follow the pointer in an index to a qualified tuple on a disk page and hence the time between two I/O requests is small. Therefore, index scans on an unclustered index are most likely IO-bound. On the other hand, index scans on a clustered index have more or less the same situation as sequential scans.

In our experiments, the most CPU-bound task is a sequential scan on relation r_{min} in which the b attribute in all the tuples is set to NULL so that the tuple size is the smallest. The most IO-bound task is a sequential scan on relation r_{max} in which the b attribute in all the tuples is set large enough so that each disk page can only hold one tuple. In XPRS, the disk page size is 8K bytes. We have measured the I/O rate of sequential scans on both r_{min} and r_{max}. The r_{min} I/O rate is 5 (IOs/second) and the r_{max} I/O rate is 70 (IOs/second). All other tasks will have I/O rate in between these extremes. We have measured the bandwidth of our disks (bandwidth after file system overhead) to be 97 io's/second for sequential reads, 60 io's/second for almost sequential reads and 35 io's/second for random reads. Even for parallel sequential scans, the effective bandwidth is at best the almost sequential bandwidth because of the asynchronousness of the parallel backends. Since we use 4 disks, we have a

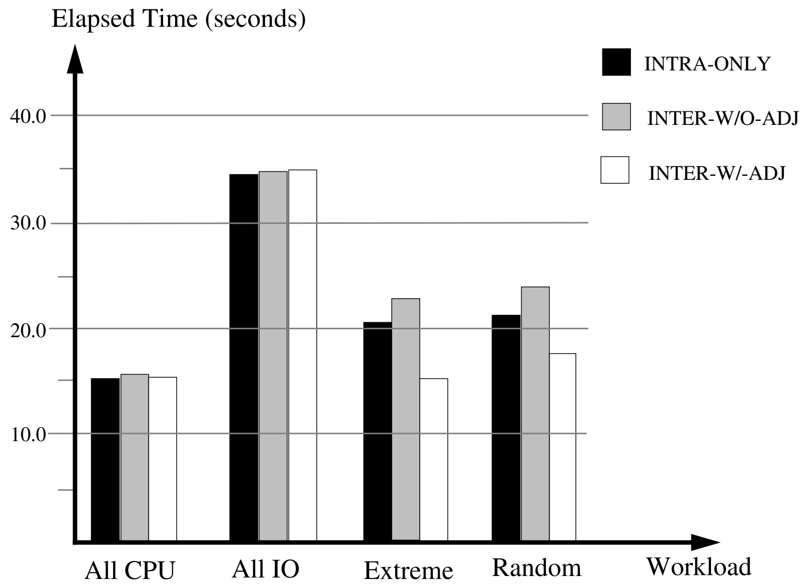

Figure 3.6: **Experiment Results of Scheduling Algorithms**

total I/O bandwidth of $4 * 60 = 240$ io's/second. Given 8 processors and a bandwidth of 240 io's/second, according to our definition, those tasks with I/O rate above $240/8 = 30$ io's/second are IO-bound and those below 30 io's/second are CPU-bound. We choose the I/O rate of the tasks in our experiments as in the following table.

Type of Tasks	IO Rate (IOs/second)
CPU-bound	randomly chosen in [5, 30)
IO-bound	randomly chosen in (30, 60]
Extremely CPU-bound	randomly chosen in [5, 15]
Extremely IO-bound	randomly chosen in [60, 70]

We run each workload in XPRS using each of the above three algorithms and measure the turnaround time of each run. Our experiment result is presented in Figure 3.6. When the workload is all IO-bound or CPU-bound, our result shows that all three algorithms have roughly the same performance and inter-operation parallelism does not help. It is sufficient to use intra-operation parallelism only in such cases. However, when there is a mixed workload of IO-bound and CPU-bound tasks, our proposed scheduling algorithm

INTER-WITH-ADJ can improve performance by as much as 25% over INTRA-ONLY. We can also see that INTER-WITHOUT-ADJ loses to INTRA-ONLY because without parallelism adjustment a task may have to run with a low parallelism even when other tasks have finished and more processors have become available.

3.4 Optimization of Bushy Tree Plans for Parallelism

In the previous sections, we have studied the scheduling problem of a set or a sequence of runable tasks. Our algorithm can be used regardless of whether the parallel tasks are from a bushy tree plan of the same query or from different queries. In this section, we will concentrate on the optimization problem of parallel execution plans for a single query and propose an optimization strategy based on the scheduling algorithm in the previous sections.

Since we have shown that a proper combination of intra-operation parallelism and inter-operation parallelism wins over only intra-operation parallelism given a workload of mixed IO-bound and CPU-bound tasks. The intra-operation-parallelism-only optimization strategy presented in the previous chapter obviously cannot always take full advantage of all available resources and thus cannot guarantee the optimality of the execution plan chosen. However, in a multi-user environment, this problem can be easily solved by combining the two phase optimization strategy in the previous chapter with our scheduling algorithm. We still find the best parallel plan for each query using only intra-operation parallelism with the algorithm in the previous chapter, but we rely on the tasks from different queries submitted by multiple users to achieve maximum resource utilizations using our scheduling algorithm.

In this section, we will focus on the optimization problem of a single query in a single-user environment where we have to depend on the tasks within a same plan to achieve IO-CPU balance and where bushy tree plans have to be considered. Our idea is to preserve the same optimization scheme as in the previous chapter, but use a new cost estimation method to estimate and compare the costs of bushy tree plans.

Since use of parallelism only helps reduce response time of a query execution, we will consider response time alone as our cost measurements in the following discussions. For each sequential plan p, let $seqcost(p)$ be the cost of sequential execution of p and $parcost(p, n)$ be the cost of parallel execution of p on n processors. As described before, plan p can be decomposed into a set of plan fragments which are what we call tasks in parallel executions. Unlike the situation in the previous sections, the tasks here have order-dependencies among themselves because some task may take the output of another task as input. However, obviously our scheduling algorithm can be easily modified to deal with the order-dependencies. It only needs to check if a task is ready before choosing it to execute and only execute the ready tasks. Suppose that $F(p) = \{f_1, f_2, \ldots, f_k\}$ is the set of plan fragments (tasks) of plan p. Using the cost estimation methods in conventional query optimization, we can estimate the sequential execution time of each task i, T_i. We can also estimate the number of I/O's of each task i, D_i. Thus, we can estimate the I/O rate of each task i as

$$C_i = D_i/T_i.$$

Let $T_n(S)$ be the elapsed time of executing a set of tasks, S with n processors. We can compute $T_n(S)$ with the following recursive formula:

$$T_n(S) = \begin{cases} T_i/maxp(f_i) + T_n(S - \{f_i\}) \\ \text{if } f_i \text{ is run alone,} \\ \\ min(T_i/x_1, T_j/x_2) + T_n((S - \{f_i, f_j\}) \bigcup \{f_{ij}\}) \\ \text{if } f_i \text{ and } f_j \text{ is run in parallel.} \end{cases}$$

where f_i and f_j are two ready tasks chosen in S according to our scheduling algorithm to execute in parallel at IO-CPU balance point (x_i, x_j), f_{ij} is the remaining task of the longer of f_i and f_j when one of them finishes first. This formula basically simulates each iteration of our scheduling algorithm and computes the total elapsed time of processing all the tasks. We compute parallel execution cost of a plan as,

$$parcost(p, n) = T_n(F(p)).$$

Now for each plan p, we can estimate $parcost(p, n)$ given n and since we are assuming a single-user environment, n is known beforehand. Therefore, we can find the plan that minimizes $parcost(p, n)$. The optimal plan can be found by a conventional query optimization algorithm with $parcost(p, n)$ replacing $seqcost(p)$. Note that the calculation of $parcost(p, n)$ depends on the structure of the entire plan tree of p which makes local pruning, a common complexity-reducing technique in conventional query optimization infeasible. Aside from this factor, we can solve the parallel optimization problem with the same algorithmic complexity as in conventional query optimization.

3.5 Summary

In this chapter, we have presented our approach to exploit inter-operation parallelism in XPRS. We have first studied the scheduling problem of a set or a sequence of independent tasks that are either from a bushy tree plan of a single query or from the plans of multiple queries and proposed a clean and simple scheduling algorithm. The scheduling algorithm achieves maximum resource utilizations by running two carefully selected tasks in parallel at their IO-CPU balance point, and avoids the combinatorial optimization problem by dynamically adjusting the degree of parallelism of the tasks to keep the system running with maximum resource utilizations. It takes full advantage of the low communication overhead feature of a shared memory system, which a shared nothing system does not have. We have also studied the optimization problem of parallel execution plans of a single query and extended our optimization strategy in the previous chapter to consider inter-operation parallelism by introducing a cost estimation method for parallel execution costs of a sequential plan based on the scheduling algorithm.

We have neglected the memory constraints on parallelism in the chapter. For example, we cannot run two hashjoins in parallel unless there is enough memory for both hash tables. Memory allocation strategies for parallel operations will be the topic of the next chapter.

Chapter 4

Memory Allocation Strategies

Processors, disk bandwidth and main memory are the three most important resources in parallel query processing. The previous chapter has dealt with the allocations of processors and disk bandwidth. This chapter will add in the third dimension, main memory to the resource allocation problem. The processing of a query ultimately consists of a sequence of CPU instructions and disk I/O requests. Therefore, the first two resources, processors and disk bandwidth, directly contribute to query processing. On the other hand, main memory only contributes in an indirect way by reducing the number of CPU instructions and I/O requests in the query processing. Because of the special role of main memory, we will treat the memory allocation problem separately in this chapter. However, in the end of this chapter, we will integrate the memory allocation strategy to be presented in this chapter with the task scheduling algorithm presented in the previous chapter to complete our approach to resource allocation in XPRS.

The question to be answered in this chapter is how to allocate memory among parallel operations optimally. In general, the contribution of main memory to query processing is very hard to model, because memory usage depends on access patterns of different operations, competition for the shared buffer pool among simultaneous operations, and buffer replacement policies. Therefore, we will not try to solve the memory allocation problem for general operations, rather we will focus on the hashjoin operation, since it is the most memory intensive operation. Because of its special memory usage, hashjoins usually allocate separate memory space that bypasses the buffer manager. For other operations, we will rely on the buffer manager to make the right decisions based on database buffer management strategies such as those in [11].

Our memory allocation strategy for parallel hashjoins follows the same framework as the task scheduling algorithm presented in the previous chapter. The main idea is to find the optimal memory allocation for two parallel hashjoins and to dynamically adjust the memory allocation when tasks change to preserve optimality.

In this chapter, the memory allocation problem to be solved is defined in Section 4.1. Then, in Section 4.2, we describe our memory allocation strategy for XPRS in details including finding the optimal memory allocation and dynamic adjustment of memory allocations. Last, our memory allocation strategy is integrated into our task scheduling algorithm in Section 4.3 and the whole chapter is summarized in Section 4.4.

4.1 Notations and Problem Definition

This section introduces notations that will be used throughout this chapter and defines the memory allocation problem.

The important notations used in this chapter are summarized in Figure 4.1. Only one I/O access time parameter is given in our notations because we do not distinguish sequential I/O and random I/O in this chapter to simplify the formulas. This is reasonable since all reads and writes of any temporary files in a hashjoin are always sequential and random I/Os may only happen during the initial read of the join relations. Therefore, our formulas can be easily extended to also consider random I/Os. In this chapter, we also do not consider the possible overlap between CPU and I/O processing because it is too difficult to model. Since hash tables take extra space for meta data, the incremental factor F is introduced to indicate hash table overhead. For example, the size of the hash table of relation R is $F \times pages(R)$ instead of just $pages(R)$. The available memory size parameter M is not necessarily a fixed constant. Since XPRS is designed as a multi-user system, M is allowed to change during query execution, but we assume that M is always above the minimum memory requirement for any hashjoins as mentioned in Chapter 2. Our algorithm is designed to be capable of adapting itself to the changing memory sizes. In this chapter, we only consider the hybrid hashjoin since it is shown to be the best hashjoin algorithm [48] and only consider elapsed time as the optimization objective. Also throughout this chapter, we assume that the relation on the left side of a join operation is the smaller relation, i.e., for join $R \bowtie S$, $pages(R) \leq pages(S)$, unless explicitly mentioned otherwise.

For simplicity, we will omit the degree of parallelism in the formulas in this chapter. Therefore, the estimated execution time of a hashjoin is its sequential execution time. Since we have shown in Section 2.2.2 that a hashjoin can be parallelized with linear speedup, the actual execution time for a hashjoin is its sequential execution time divided by its degree

Notation	Meaning
$comp$	time to compare keys in main memory
$hash$	time to hash a key that is in main memory
$move$	time to move a tuple in memory
IO	time to read or write a page between disk and main memory
F	hash table incremental factor
$pages(R)$	number of pages in relation R
$tuples(R)$	number of tuples in relation R
M	total available memory size

Figure 4.1: **Important Notations in This Chapter**

of parallelism.

As shown in [48], each hashjoin has a minimum memory requirement, $minM$ and a maximum memory requirement, $maxM$. For hashjoin $R \bowtie S$,

$$minM(R \bowtie S) = \sqrt{F \times pages(R)}, \quad maxM(R \bowtie S) = F \times pages(R).$$

Chapter 3 shows that it is sufficient to only run two tasks at a time. Therefore, in this chapter, we will only consider the memory allocation problem between two hashjoin operations. Suppose that we have two hashjoins, $R_1 \bowtie S_1$ and $R_2 \bowtie S_2$ that can be executed in parallel. If $maxM(R_1 \bowtie S_1) + maxM(R_2 \bowtie S_2) \leq M$, the memory allocation problem does not exist because we can satisfy the maximum memory requirement for both hashjoins. If $minM(R_1 \bowtie S_1) + minM(R_2 \bowtie S_2) > M$, the memory allocation problem also does not exist because there is not enough memory for executing both hashjoins together and each hashjoin must be executed separately. The memory allocation problem arises only if $minM(R_1 \bowtie S_1) + minM(R_2 \bowtie S_2) \leq M < maxM(R_1 \bowtie S_1) + maxM(R_2 \bowtie S_2)$, when we have enough memory to execute both hashjoins in parallel but not enough to satisfy the maximum requirements of both hashjoins. In this case, we must answer the following three questions.

- **Question 1.**

  Is it worthwhile to execute the two hashjoins in parallel? If we execute each hashjoin separately, each will have the entire M units of memory and will run faster. Hence, we must decide whether to execute the two hashjoins in parallel or one after another.

- **Question 2.**

  If we decide to execute the two hashjoins together, how do we divide M between the two hashjoins optimally?

- **Question 3.**

  If the optimal memory allocation changes when one hashjoin finishes and a new hashjoin enters, how can we adjust the current allocation to the optimal allocation?

Section 2.2.3 also raised the issue of plan fragment decomposition. As discussed in Section 2.2.3, there are at most two hashjoins in a plan fragment (actually a hash probe followed by a hash build as illustrated in Figure 2.7) because of the blocking edge between the two phases of a hashjoin. Questions 1 and 2 must also be answered for two hashjoins in the same plan fragments. The only difference is that in this case, the two hashjoins are not executed independently; they are in a pipeline. If the answer is to execute them separately, the plan fragment will be decomposed into two, each containing one hashjoin; and instead of pipelining, the results of the first hashjoin will be stored in a temporary relation. Hence, the temporary relation overhead must also be considered. We will give a simple solution to the plan fragment decomposition problem based on our answers for the parallel hashjoins.

4.2 Memory Allocation Strategies for Hashjoins

This section presents our memory allocation strategy for parallel hashjoins and gives our answers to Questions 2 and 3 posed in the previous section. Because the answer to Question 1 is based on the answers to Questions 2 and 3, it is delayed to the next section. This section is organized into three subsections. The first subsection introduces a cost formula for hashjoins as the basis for our analysis. The second subsection then introduces a theorem for optimal memory allocation for hashjoins and presents the answer to Question 2 and a solution to the plan fragment decomposition problem. The third subsection then describes our mechanism of dynamic memory allocation adjustment for hashjoins and answers Question 3.

4.2.1 Cost Analysis of Hashjoins

We will first briefly recapitulate the hybrid hashjoin algorithm. A hybrid hashjoin consists of a number of *batches* of simple hashjoins. In a hashjoin $R \bowtie S$, if $F \times pages(R) \leq M$, there is only one batch, i.e., the entire R is read and hashed into the main memory hash table, then S is read to probe the hash table of R for matching tuples. If $F \times pages(R) > M$, there will be $B+1$ batches in the hybrid hashjoin as described below, where

$$B = \lceil \frac{pages(R) \times F - M}{M - 1} \rceil.$$

1. First, we choose a hash function h and a partition of its hash values which will divide R into $R_0, \ldots, R_B$, such that the hash table for R_0 has $M - B$ pages, and $R_1, \ldots, R_B$ are of equal size. Then allocate B pages in memory to B output buffers, one for each $R_1, \ldots, R_B$, and assign the remaining $M - B$ pages of memory to the hash table of R_0.

2. Scan R and hash each tuple with h. If it belongs to R_0 it is inserted into the hash table of R_0 in memory. Otherwise it belongs to R_i with $i > 0$, so move it to the output buffer of R_i. When an output buffer is filled up, it is written to a disk file. When this step finishes, the hash table of R_0 is in memory and $R_1, \ldots, R_B$ are on disk.

3. Partition S with h in the same way as partitioning R, into $S_0, \ldots, S_B$. Assign the output buffer for R_i in the previous step to S_i, for $i = 1, \ldots, B$. Scan S and hash each tuple with h. If the tuple is in S_0, probe the hash table of R_0 for a match. If there is a match, output the result tuple, otherwise drop the tuple. If the tuple is not in S_0, it belongs to S_i for $i > 0$, so move it to the output buffer of S_i. When an output buffer is filled up, it is written to a disk file. In the end, $R_0 \bowtie S_0$ is produced and $R_1, \ldots, R_B$ and $S_1, \ldots, S_B$ are left on disk.

4. For $i = 1, \ldots, B$, perform simple hashjoin of $R_i \bowtie S_i$, since we can easily prove that the hash table R_i fits in memory, i.e., $R_i \times F \leq M$.

For cost computation, we denote by $T(R \bowtie S, M)$ the elapsed time for executing the hashjoin $R \bowtie S$ sequentially with M units of memory. The following cost formula is directly quoted from [48].

$T(R \bowtie S, M) =$

$(pages(R) + pages(S)) \times IO$	Initial Read of R and S.
$+(tuples(R) + tuples(S)) \times hash$	Partition R and S.
$+(tuples(R) + tuples(S)) \times (1-q) \times move$	Move tuples to output buffers.
$+(pages(R) + pages(S)) \times (1-q) \times IO$	Write from output buffers.
$+(pages(R) + pages(S)) \times (1-q) \times IO$	Read batches into memory.
$+(tuples(R) + tuples(S)) \times (1-q) \times hash$	Hash in joining $R_i \bowtie S_i, i = 1, \ldots, B$.
$+tuples(R) \times move$	Move tuples to hash tables for R.
$+tuples(S) \times comp \times F$	Probe for each tuple of S.

where

$$q = pages(R_0)/pages(R)$$

and

$$pages(R_0) = (M - B)/F = (M - \frac{pages(R) \times F - M}{M-1})/F = \frac{M^2 - pages(R) \times F}{(M-1)F}.$$

Thus,

$$q = \frac{M^2 - pages(R) \times F}{pages(R) \times (M-1)F}.$$

4.2.2 Optimal Memory Allocation for Hashjoins

This subsection proves a theorem for optimal memory allocation between two parallel hashjoins and answers our Question 2.

Definition 4.1 *For hashjoin $R \bowtie S$, let*

$$MC = \frac{2 \times pages(S) \times IO + (tuples(R) + tuples(S))(move + hash)}{pages(R)}.$$

We call MC the **memory coefficient** *of the hashjoin.*

Theorem 4.1 *If two hashjoins, $J_1 : R_1 \bowtie S_1$ and $J_2 : R_2 \bowtie S_2$ are to be executed in parallel, and if $minM(J_1) + minM(J_2) \leq M < maxM(J_1) + maxM(J_2)$, the optimal memory allocation for these two hashjoins is to allocate as much memory as possible to the hashjoin with larger memory coefficient leaving only the remaining memory or the minimum required memory to the other hashjoin, i.e., if J_1 has the larger memory coefficient, J_1 should be given $M - max(M - maxM(J_1), minM(J_2))$ units of memory while J_2 should only get $max(M - maxM(J_1), minM(J_2))$ units of memory.*

Proof.

We can reorganize the formula for $T(R \bowtie S, M)$ as follows.

$$\begin{aligned} T(R \bowtie S, M) &= (pages(R) + pages(S)) \times IO + (tuples(R) + tuples(S)) \times hash \\ &\quad + tuples(R) \times move + tuples(S) \times comp \times F \\ &\quad + 2 \times (pages(R) + pages(S)) \times (1 - q) \times IO \\ &\quad + (tuples(R) + tuples(S)) \times (1 - q) \times (move + hash). \end{aligned}$$

Since

$$q = \frac{M^2 - pages(R) \times F}{pages(R) \times (M - 1)F},$$

we have

$$1 - q = \frac{pages(R) \times F - M}{pages(R) \times F} \frac{M}{M-1}.$$

Because we always assume a large amount of memory M, $\frac{M}{M-1} \approx 1$. Hence,

$$1 - q \approx \frac{pages(R) \times F - M}{pages(R) \times F}.$$

By substituting this into the formula for $T(R \bowtie S, M)$, we can derive the following formula.

$$\begin{aligned}
T(R \bowtie S, M) \quad = \quad & 3 \times (pages(R) + pages(S)) \times IO \\
& + 2 \times tuples(R) \times (move + hash) \\
& + tuples(S) \times (2 \times hash + comp \times F) \\
& - \frac{1}{F}(2IO + MC) \times M.
\end{aligned}$$

In brief,

$$T(R \bowtie S, M) = A - \frac{1}{F}(2IO + MC)M,$$

where

$$\begin{aligned}
A \quad = \quad & 3 \times (pages(R) + pages(S)) \times IO \\
& + 2 \times tuples(R) \times (move + hash) \\
& + tuples(S) \times (2 \times hash + comp \times F).
\end{aligned}$$

Obviously, A is independent of memory size M.

Suppose that M_1 units of memory are allocated to hashjoin J_1, and M_2 units of memory are allocated to hashjoin J_2. We have, $M_1 + M_2 = M, minM(J_i) \leq M_i \leq maxM(J_i), i = 1, 2$. The total cost of executing both hashjoins is,

$$\begin{aligned} T(J_1, M_1) + T(J_2, M_2) &= A_1 - \frac{1}{F}(2IO + MC_1)M_1 + A_2 - \frac{1}{F}(2IO + MC_2)M_2 \\ &= (A_1 + A_2) - \frac{1}{F}2IO(M_1 + M_2) - \frac{1}{F}(MC_1 \times M_1 + MC_2 \times M_2) \\ &= (A_1 + A_2) - \frac{1}{F}2IO \times M - \frac{1}{F}(MC_1 \times M_1 + MC_2 \times M_2). \end{aligned}$$

As we can see, in order to minimize the total cost, the term $MC_1 \times M_1 + MC_2 \times M_2$ must be maximized, for which we should give as much memory as possible to the hashjoin with larger memory coefficient.

Q.E.D.

Theorem 4.1 answers our Question 2. It shows that memory coefficient is a measure of the contribution of each unit of memory to a hashjoin. Therefore, the answer to Question 2 is to allocate as much memory as possible to the hashjoin with larger memory coefficient.

Theorem 4.1 can also be applied to solve the plan fragment decomposition problem. Suppose that we have two consecutive hashjoins, $(R \bowtie S) \bowtie T$. Let $U = R \bowtie S$. We assume that $pages(U) \leq pages(T)$. If $minM(R \bowtie S) + minM(U \bowtie T) \leq M < maxM(R \bowtie S) + maxM(U \bowtie T)$, we need to decide whether to execute the two joins together in a pipeline, or to execute $R \bowtie S$ first and store the result in a temporary relation U, then execute $U \bowtie T$. Suppose that M_1 and M_2 $(M_1 + M_2 = M)$ are the optimal memory allocation for these two hashjoins according to Theorem 4.1. We can calculate the cost of executing the two hashjoins together as:

$$T(R \bowtie S, M_1) + T(U \bowtie T, M_2) - pages(R \bowtie S) \times IO,$$

where $pages(R \bowtie S) \times IO$ is the I/O cost saved by pipelining. We can also calculate the cost of executing the two hashjoins separately as:

$$T(R \bowtie S, M) + T(U \bowtie T, M) + pages(R \bowtie S) \times IO,$$

where $pages(R \bowtie S) \times IO$ is the cost for writing the temporary relation to disk (buffering is not considered here). These formulas can be easily modified by exchanging U and T if $pages(U) > pages(T)$. Therefore, we can decide whether to execute the two hashjoins together or separately by comparing their costs and choose the way with lower cost.

4.2.3 Dynamic Memory Adjustment in Hashjoin

If two hashjoins are executed in parallel, when one of them finishes and a new hashjoin starts, the optimal memory allocation may very well change, thus we may need to adjust the amount of memory allocated to the currently running hashjoin. If the memory coefficient of the running hashjoin is larger than that of the new hashjoin, more memory may need to be added to the running hashjoin. On the other hand, if the memory coefficient of the running hashjoin is smaller than that of the new hashjoin, a certain amount of memory may need to be taken away from the running task to be given to the new hashjoin. This subsection describes our mechanism to dynamically increase or decrease the amount of memory of a hashjoin while it is running.

We will first describe the implementation of hashjoin in XPRS before the dynamic memory adjustment mechanism is presented. The XPRS implementation of hashjoin is similar to that described in [60], which also deals with dynamic changes in the amount of available memory. However, [60] only considers dynamic memory adjustment in the build phase of a hashjoin, while our implementation of hashjoin allows dynamic memory adjustment in both phases of a hashjoin. Also [60] focuses on the situations when the amount of available memory is less than expected or the join relation size is larger than estimated, thus is mainly concerned with decreasing memory sizes. On the other hand, we consider both decreasing and increasing memory sizes. If more memory becomes available during the execution of a hashjoin, we also adjust the hashjoin to take advantage of the newly available memory.

The important concepts in our hashjoin implementation are *hash chains*, *hash buckets* and *hash batches*. A hash chain is a linked list of tuples with the same hash value. A hash bucket is a set of hash chains that share the same set of pages. A hash batch is a set of hash buckets that are used in the same batch in a hybrid hashjoin. As we can see, each hash chain corresponds to one hash value, each hash bucket corresponds to a range of hash values, while each hash batch corresponds to a larger range of hash values. Our hash table consists of an in-memory *hash directory* which associates hash value ranges to corresponding hash buckets, and a set of pages for all the hash buckets, as shown in Figure 4.2. Each page also contains a local hash directory that identifies all the hash chains within the page. We use relative addressing for pointers which allows the format of the pages in a hash bucket

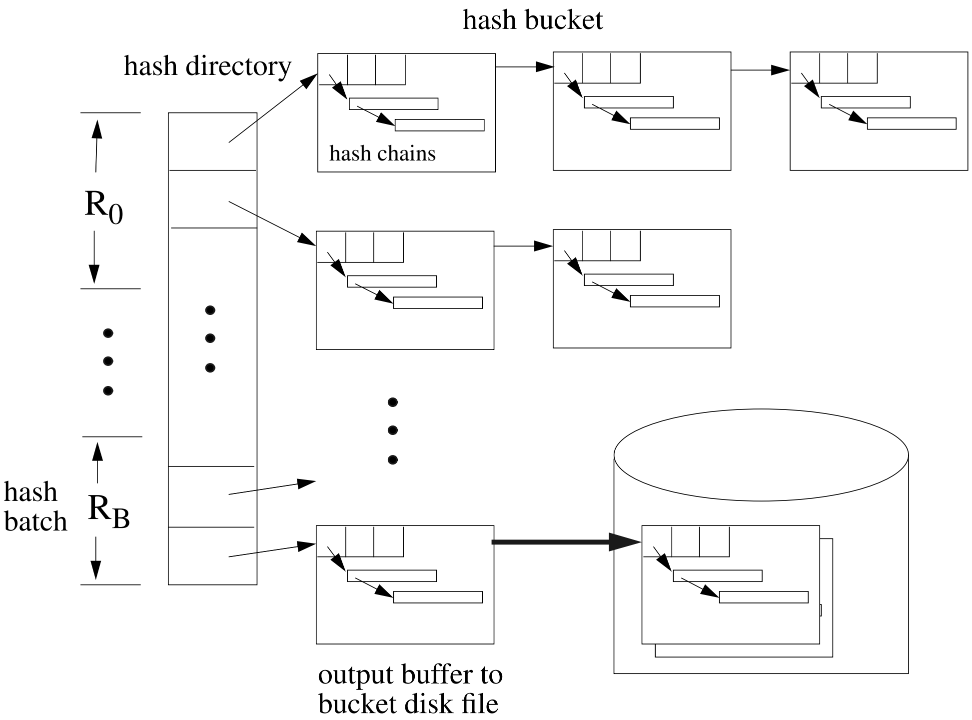

Figure 4.2: **The Hash Table Structure for Hashjoins in XPRS**

to be the same regardless of whether they reside in memory or on disk. Therefore, when there is not enough memory to hold a hash bucket, its pages can be swapped from memory to disk and later on read back to memory without changing any internal format.

The advantage of our hashjoin implementation is that it is not tied to a particular continuous segment of memory, thus can be very flexible. If too many tuples are hashed into the same hash bucket and cause a *bucket overflow*, we can partition the hash bucket into two hash buckets by dividing the range of hash values corresponding to the old hash bucket and move tuples into new hash buckets according to the new hash value ranges. This approach avoids excessively long hash chains which degrades hashing performance and is similar to the idea of *extendible hashing* [18]. Under our hashjoin scheme, a hash bucket can either reside in memory or on disk depending on memory availability. If there is not enough memory initially, a hash bucket may be written to a *bucket file* on disk, but later on if more memory becomes available, it can be read back into memory and become memory-resident. We can also move a hash bucket from one hash batch to another by adjusting the hash value ranges of the hash batches. In this way, we can dynamically adjust the hash-partitioning in a hashjoin by rearranging the hash batches.

Let us consider hashjoin $R \bowtie S$. When there is a fixed amount of memory, our hashjoin implementation performs exactly the same as the hybrid hashjoin algorithm described in Section 4.2.1. First, R is hash-partitioned to batches $R_0, R_1, \ldots, R_B$, where all the hash buckets of R_0 are memory-resident, while all the hash buckets of $R_i, i = 1, \ldots, B$ are on disk with only one page for each hash bucket in memory as an output buffer. Then S is hash-partitioned in the same way as R while performing $R_0 \bowtie S_0$ during the same pass through S. After hash partitioning is finished, $R_i \bowtie S_i, i = 1, \ldots, B$ are performed in turn. As we can see, in fact a hybrid hashjoin can be viewed as three stages: hash-partitioning R, hash-partitioning S and subjoins $R_i \bowtie S_i, i = 1, \ldots, B$. Now suppose that the amount of memory changes during the execution of a hashjoin. We consider dynamic memory adjustment for each stage in a hashjoin separately. Suppose that the old memory size is M and the new memory size is M'. Let $k = \lfloor |M' - M|/bsize \rfloor$, where $bsize$ is the bucket size for the hash table of R.

- If the memory size changes while R is being hash-partitioned.

  If $M' > M$, k more hash buckets of R will be read from disk to memory. Namely, R_0 will be increased by k hash buckets. Since these hash buckets will have to be read into memory for the subjoins, we do not introduce any extra overhead in this adjustment.

By increasing R_0, we not only save the disk write cost of corresponding hash buckets of S when S is hash-partitioned, but also may reduce the number of batches in the hashjoin.

If $M' < M$, k hash buckets of R_0 will be swapped from memory to disk. The number of batches in this hashjoin will therefore be increased. However, $M - M'$ units of memory can be freed and allocated to another hashjoin with larger memory coefficient. The overall cost is still reduced.

- If the memory size changes while S is being hash-partitioned.

  If $M' > M$, k hash buckets of R will be read from disk to memory, The k corresponding hash buckets of S will also be read and each tuple will be used to probe into the k hash buckets of R. If a match is discovered, a result tuple is output. Otherwise, the tuple is dropped. After all k hash buckets of S are scanned, the hashjoin just proceeds with a larger R_0 in memory. Since these k hash buckets of R and S will have to be read into memory for the subjoins, this adjustment does not incur any extra overhead. By increasing R_0, further writes to the k hash buckets of S are saved and the number of batches in the hashjoin may also be reduced.

  If $M' < M$, k hash buckets of R_0 will be swapped from memory to disk. Tuples of S that belong to corresponding hash buckets and have not yet been scanned will from now on be written to disk instead of be used to probe the k hash buckets of R_0. The number of batches in this hashjoin will therefore be increased. However, $M - M'$ unites of memory can be freed and allocated to another hashjoin with larger memory coefficient and the overall cost is still reduced.

- If the memory size changes while subjoins are being performed.

  In executing a subjoin, we first always read as many hash buckets of R into memory as possible then scan the corresponding hash buckets of S to probe the in-memory hash table of R. Therefore, the actual sizes of $R_i, S_i, i = 1, \ldots, B$ change dynamically according to the available memory size when each subjoin starts. If the memory size changes during a subjoin, the same memory adjustment techniques as we have described so far can be applied to each subjoin.

As we can see, the granularity of our dynamic memory adjustment is a hash bucket. Therefore, the hash bucket size is an important parameter for performance tuning.

4.3 Integration with Task Scheduling Algorithm

The previous section has answered Question 2 and 3 raised in Section 4.1. This section gives our answer to Question 1 and integrates our memory allocation strategy with the task scheduling algorithm presented in Chapter 3. Our task scheduling algorithm use the I/O rate of each task to decide whether it is IO-bound or CPU-bound. The memory allocation problem complicates task scheduling because the I/O rate of a hashjoin varies with different amount of memory allocated. Therefore this variable I/O rate issue must be dealt with in the integration of our memory allocation strategy and our task scheduling algorithm. In this section, we will first give the answer to Question 1, then deal with variations of I/O rates, and last present the modified task scheduling algorithm integrated with our memory allocation strategy.

4.3.1 Answer to Question 1

Question 1 can be answered by introducing the memory factor into the formula of $T_{intra}()$ and $T_{inter}()$ of Section 3.2.4 for hashjoins. Consider two hashjoins: $J_1 : R_1 \bowtie S_1$ and $J_2 : R_2 \bowtie S_2$. Suppose that J_1's memory coefficient is larger than J_2's and M is the total memory size. According to Theorem 4.1, the optimal memory allocation for the two hashjoins is $M_1 = M - max(M - maxM(J_1), minM(J_2))$ to J_1 and $M_2 = max(M - maxM(J_1), minM(J_2)$ to J_2. Therefore, if we run the two hashjoins separately, we have

$$T_{intra}(J_i) = T(J_i, M)/maxp(J_i),\ i = 1, 2.$$

On the other hand, if we run the two hashjoin together and suppose that (x_1, x_2) is the IO-CPU balance point defined in Chapter 3 for the two joins, we have,

$$T_{inter}(J_1, J_2) = min(T(J_1, M_1)/x_1, T(J_2, M_2)/x_2) + T(J_{12}, M)/maxp(J_{12}) + T_{adjust},$$

where J_{12} represents the processing that remains to be done for the running join when the other join has finished first, and T_{adjust} represents the overhead for memory adjustment that happens for the longer-running hashjoin to take advantage of the newly available memory. We have,

$$T(J_{12}, M) = \begin{cases} T(J_1, M) - T(J_2, M_2)x_1/x_2 & \text{if } T(J_1, M_1)/x_1 > T(J_2, M_2)/x_2 \\ T(J_2, M) - T(J_1, M_1)x_2/x_1 & \text{otherwise,} \end{cases}$$

$$T_{adjust} = \begin{cases} 2 \times M_2 \times IO & \text{if } T(J_1, M_1)/x_1 > T(J_2, M_2)/x_2 \\ 2 \times M_1 \times IO & \text{otherwise,} \end{cases}$$

The answer to Question 1 is to compute $T_{inter}(J_1, J_2)$ and $T_{intra}(J_1) + T_{intra}(J_2)$. If $T_{inter}(J_1, J_2) < T_{intra}(J_1) + T_{intra}(J_2)$, we should run the two joins together, otherwise we should run them separately.

Note that the above analysis is somewhat simplified. It assumes each hashjoin as one single task, while in fact each hashjoin has two phases and each phase belongs to a different plan fragment because of the blocking between the two phases as discussed in Section 2.2.3. However, the above analysis can be easily changed to treat each hashjoin as two tasks. The details are omitted here. Moreover, the above analysis only considers the case in which both hashjoins are to be executed from scratch. In general, a new hashjoin may be chosen to start while another hashjoin still running. Therefore, we need to decide whether to start a new hashjoin to run in parallel with the currently running hashjoin, or to leave the currently running hashjoin alone until its completion. The above analysis can also be easily extended to deal with this general situation, in which the memory size of a hashjoin may be adjusted larger or smaller, but in either case, we can calculate T_{adjust} as,

$$T_{adjust} = 2 \times |new\_memory\_size - old\_memory\_size| \times IO,$$

i.e., the overhead for swapping in and out and hash buckets held in the memory added or removed.

4.3.2 Variation of I/O Rate

In Chapter 3, we have assumed that the I/O rate of a task is a constant that can be pre-determined. However, for a hashjoin, the I/O rate varies depending on the amount of memory allocated to the hashjoin. The range of I/O rate variation of a hashjoin is analyzed below.

We will only consider the I/O rate in the build phase of a hashjoin $R \bowtie S$. The I/O rate in the probe phase can be similarly analyzed. Suppose that R will be hash-partitioned into $R_0, R_1, \ldots, R_B$. The I/O rate C can be written as $C = C_r + C_w$, where C_r is the rate to read pages of R from disk and C_w is the rate to write pages of $R_i, i = 1, \ldots, B$ to disk. Here C_r is a constant independent of memory size, but C_w varies with memory size. Let P be the probability of an arbitrary tuple of R belongs to batch $R_i, i > 0$. We have $C_w = P \times C_r$. We can also calculate P as $P = pages(R_i)/pages(R)$, where

$$pages(R_i) = \frac{pages(R) - pages(R_0)}{B},$$

$$pages(R_0) = \frac{M - B}{F},$$

$$B = \lceil \frac{F \times pages(R) - M}{M - 1} \rceil.$$

Therefore, we can express C_w and thus C as a function of memory size M. We represent the function for C of M as $C(M)$.

We are mainly interested in the lower bound and upper bound of C_w. Obviously, when $M \geq F \times pages(R)$, $C_w = 0$. Hence the lower bound of C_w is 0. Because $pages(R_0) \geq$

$pages(R_i)$, and $B >= 1$, we know that

$$pages(R_i) = (pages(R) - pages(R_0))/B \leq pages(R) - pages(R_i).$$

Therefore, $pages(R_i) \leq pages(R)/2$. Hence, $P \leq 1/2$, i.e., the upper bound of C_w is $C_r/2$. Thus, we have,

$$C_r \leq C(M) \leq 1.5C_r.$$

In other words, the range of variation of I/O rate C is $[C_r, 1.5C_r]$. We may use this range to determine if a task is IO-bound or CPU-bound in the modified scheduling algorithm to be presented next.

4.3.3 Modified Scheduling Algorithm

Having answered Questions 1, 2 and 3, now we are ready to integrate our memory allocation strategy with the task scheduling algorithm presented in Chapter 3. The main idea of the integration is the following.

- In addition to running two tasks at their IO-CPU balance point, if the tasks are hashjoins, we also allocate the optimal amount of memory to each task according to Theorem 4.1.
- In addition to dynamically adjusting parallelism to the new IO-CPU balance point when a new task enters, we also dynamically adjust memory allocation to the new optimal memory allocation using the techniques described in Section 4.2.3.

Specific modifications to the task scheduling algorithm in Section 3.2.4 are described below.

In Step 1, the set of tasks can no longer be divided to only S_{io} and S_{cpu} because the I/O rate of a task may vary with the amount of memory allocated to the task, thus we may not be able to determine whether a task is IO-bound or CPU-bound. However, even for a hashjoin, it is still possible to determine if it is IO-bound or CPU-bound using the lower bound $minC$ and upper bound $maxC$ of its I/O rate. According to the definition of IO-bound and CPU-bound tasks in Section 3.2.1, if $minC > B/N$, we know that the hashjoin is guaranteed to be IO-bound, and if $maxC \leq B/N$, we know that the hashjoin is guaranteed to be CPU-bound (where B is the total disk bandwidth and N is the number of processors). However, if $minC \leq B/N < maxC$, we cannot determine if the hashjoin is IO-bound or CPU-bound until memory is allocated. Therefore, we introduce a third set $S_{unknown}$ which contains all the tasks whose class (IO-bound or CPU-bound) cannot be pre-determined. Now we have $S = S_{io} \bigcup S_{cpu} \bigcup S_{unknown}$. When choosing a task in Step 2 or 3, if S_{io} or S_{cpu} becomes empty, tasks in $S_{unknown}$ will be chosen.

In Step 3, if the two tasks are hashjoins, the check to decide whether to run the two tasks together or separately should be performed with the new formulas for T_{inter} and T_{intra} in Section 4.3.1. If the two tasks are chosen from $S_{unknown}$, we will first determine the optimal memory allocation (M_1, M_2) between the two tasks, then calculate the actual I/O rate $C_1(M_1)$ and $C_2(M_2)$ of the two tasks to make sure that one task is IO-bound and the other is CPU-bound. If it is not the case, the check fails.

In Step 4, if $task_1$ is a running task, in addition to adjusting its parallelism to $maxp(task_1)$, we also adjust it to use all the available memory. In Step 5, in addition to calculating the IO-CPU balance point, (x_1, x_2), we also calculate the optimal memory allocation, (M_1, M_2). In Step 6, whenever parallelism is adjusted, we also adjust memory allocation to the new optimal allocation, (M_1, M_2).

Section 3.2.4 also proposed some heuristics for choosing the pair of IO-bound and CPU-bound tasks, such as "most IO-bound and most CPU-bound first" and "shortest query first". Here we introduce yet another heuristic for a set of hashjoin tasks. Because memory adjustment introduces extra overhead, we want to have a heuristic that tries to minimize memory adjustment overhead. According to Theorem 4.1, we always allocate as much memory as possible to the hashjoin with larger memory coefficient. We call a memory adjustment a *major memory adjustment* if a hashjoin changes from the one with smaller memory coefficient to the one with larger memory coefficient or vice versa. Obviously, a major memory adjustment is most likely to be expensive because it has to adjust from the minimum required memory to most of the available memory or vice versa. Our heuristic minimizes the number of major memory adjustments, in which we choose the IO-bound hashjoins in memory coefficient descending order and the CPU-bound hashjoins in memory coefficient ascending order. It can be easily shown that we only need to make at most one major memory adjustment under this heuristic. This is because that there may be exactly one point when the memory coefficient of the IO-bound tasks becomes smaller than that of the CPU-bound tasks. If such a point exists, then one major memory adjustment is made at this point, but there is no major memory adjustment before or after this point. If such a point does not exist, then no major memory adjustment will be made.

4.4 Summary

In this chapter, we have presented our memory allocation strategy for hashjoins in XPRS and integrated it with the task scheduling algorithm proposed in Chapter 3. Our memory allocation strategy follows the same framework as the task scheduling algorithm. We run two parallel hashjoins with their optimal memory allocation and dynamically adjust memory allocation to the new optimal memory allocation when a new task starts. A theorem on optimal memory allocation between two hashjoins is developed. According to the theorem, if there is not enough memory to satisfy the maximum memory requirement of both hashjoins, in the optimal memory allocation, as much memory as possible should go to the hashjoin with larger memory coefficient leaving only the minimum required memory or the remaining memory after the maximum memory requirement of this hashjoin is satisfied to the hashjoin with smaller memory coefficient. The plan fragment decomposition problem raised in Section 2.2.3 is also solved using this theorem. Our mechanism for dynamic memory adjustment in hashjoins is made possible by the careful design of our hashjoin implementation such that hash buckets can be moved in and out of memory easily depending on memory availability. This chapter along with the previous two chapters completes the approach to parallel query processing in XPRS.

Chapter 5

Performance of Disk Array Configurations

The previous chapters have only considered one particular disk array configuration, the RAID Level 0 or the simplex configuration. This chapter completes the discussion on parallel query processing by studying the performance implications of different disk array configurations to parallel query processing. Disk arrays and shared memory multiprocessors are complementary to each other because disk arrays automatically balance the I/O workload by striping data across all disks while shared memory multiprocessors automatically balance the processing workload across multiple processors. XPRS supports several different disk array configurations and each of them has its pros and cons. The question is how these different disk array configurations perform in a parallel query processing environment compared with each other. This chapter intends to answer this question through experiment results on XPRS. As we will show, although different disk array configurations have different performance characteristics, the difference does not affect the results presented in the previous chapters.

This chapter is organized as follows. In the next section, we will introduce the concept of disk arrays and different configurations of disk arrays, particularly the mirrored configuration and the parity array configuration in more detail. Then in Section 5.2, we will describe the experiments that we have run on XPRS with different disk array configurations and present the experiment results that illustrate the performance of these disk array configurations. Last, this chapter is concluded and summarized in Section 5.3.

5.1 Disk Array Configurations

A disk array consists of many small form factor disks which provide high aggregate I/O rate through data striping [45]. A significant disadvantage of striping is that if a single disk within the stripe set fails, none of the data can be accessed. In other words, a single failure renders the entire array unavailable. Even though disk technology has made each individual disk highly reliable, a disk array may still fail very often because its failure rate scales up with the number of disks in the array. Therefore, it is critical that a disk array include some fault-tolerant mechanism to increase the availability. It is well known that the best approach to supporting high data availability is through redundancy. Data is stored in such a fashion that each bit of data contributes to another redundant bit store elsewhere in the array. A disk array that stores redundant data for the purpose of achieving high data availability is called a *RAID*, or a *Redundant Array of Inexpensive Disks*.

There are two intrinsic penalties to data redundancy: first, storage capacity is sacrificed to store the extra redundant data, and second, write bandwidth is wasted in keeping the redundant data up to date. There is a hierarchy of alternative RAID configurations from level 0 to level 5 that represent different tradeoffs between availability, I/O performance, and capacity [37]. Data striping without redundancy is called RAID Level 0, or the *simplex* configuration. The experiment results presented in the previous chapters are all based on a RAID 0 configuration. The two configurations best suited for both high I/O rate and high availability are RAID Level 1 also called *mirrored disks* and RAID Level 5 also called *parity arrays*. These two configurations are described in more details below.

5.1.1 RAID Level 1: Mirrored Disks

In a mirrored disk configuration, each disk is associated with another disk as its mirrored partner. Each data block on one disk is also stored redundantly on its mirrored partner. Hence the capacity overhead for redundancy is 100%. Obviously, since each disk has a mirrored partner, data can be read from either disk, but writes must be made to both disks. Therefore, the effective write rate is only half of what the disks can collectively provide. Because data is placed under two independent disks, we can further improve read rate by a technique called *seek scheduling*, which schedules the disk with its head closer to the requested data to service the request. Fifteen to twenty percent improvement to read rate has been observed [10]. Assuming a reasonably short repair time, such as 72 hours, the probability that both members of a mirrored pair fail at the same time is very small, and thus a very large expected mean time to data loss (MTTDL) can be achieved.

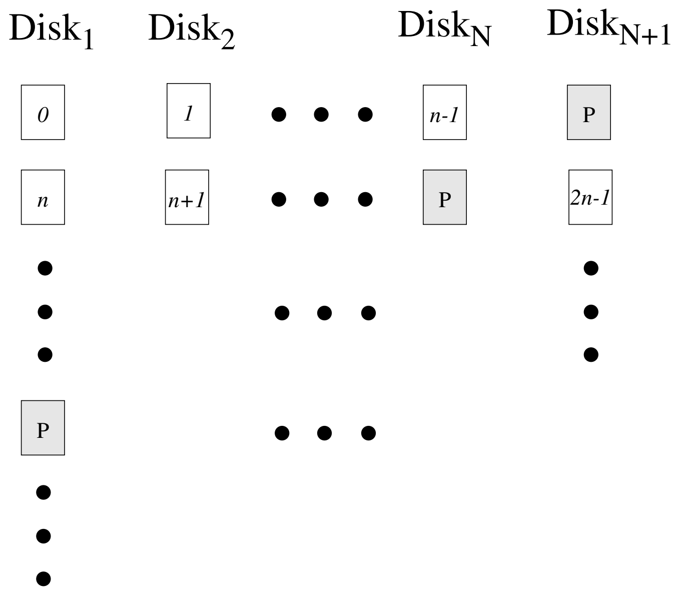

Figure 5.1: **RAID Level 5: Parity Array**

5.1.2 RAID Level 5: Parity Array

In a $N+1$ disk parity array configuration, N disks are used to store real data and one extra disk is used to store the redundant data in the form of parities. Every N data blocks that horizontally spread across the N data disks are exclusive-OR'ed into a parity block that is stored in the extra disk. In fact, the parity blocks do not have to be stored in the same disk, they are actually spread across all $N+1$ disks in the array, interleaved with data blocks, as shown in Figure 5.1, to avoid bottlenecking on the parity disk.

As we can see, if any single disk in the array fails, the data blocks in the failed disk can always be reconstructed by exclusive-OR'ing the available $N-1$ data blocks and the parity block. If hot spares are included in the array, accesses can continue to the failed disk while its contents are being reconstructed into a spare disk. With this scheme, the "window of vulnerability" to a second failure among the working disks can be reduced from the usual 72 hours for repair to $1-2$ hours, thus the MTTDL is significantly increased.

Gibson shows in [19] that parity arrays with hot spares can offer about the same availability as mirrored disks. However, a parity array only requires one extra disk for every N data disks to achieve tolerance to single failures while the mirrored configuration needs one extra disk for every data disk.

The problem of parity arrays is write performance. In a parity array, for each data block written, the associated parity block must also be updated. The new parity block can only be computed by exclusive-OR'ing the old data block, the new data block and the old parity block. Therefore, one logical write actually becomes four physical I/Os: read old data, read old parity, write new data, write new parity. Thus the write rate is reduced to 25% of what a non-redundant array can deliver. However, this "read before write" overhead can be avoided if data is written in units of N blocks at a time. Therefore, parity arrays only suffer performance penalties for small writes.

5.2 Performance of Parallel Query Processing on RAID

This section examines the performance of parallel query processing on different RAID configurations, specifically simplex, mirrored disks and parity arrays. A set of benchmark queries are run on XPRS with each of the three disk array configurations and their performance are reported to compare the three disk array configurations. We will first describe the experiments on XPRS that are run and then present the experiment results.

5.2.1 Experiments on XPRS

In our experiments, we run a standard set of benchmark queries on each of the three disk array configurations, measuring the performance in terms of elapsed time. The benchmark queries can be characterized as follows:

- sequential read (sequential scan),
- random read (unclustered index scan),
- sequential write (update consecutive tuples through sequential scan),
- random write (update through an unclustered index).

Each query is run on three disk array configurations: simplex, mirror and parity array. For each configuration, we consider two situations: fixed total capacity and fixed user capacity. Specifically, each query will be run six times on the configurations in the following table.

# of disks	fixed total capacity	fixed user capacity
simplex	6	3
mirror	2×3	2×3
parity array	$5 + 1$	$3 + 1$

We choose to use queries from a modified version of the Wisconsin Benchmark [3]. As shown in Section 2.2.2, a sequential scan is CPU-bound when the tuple size is small and IO-bound when the tuple size becomes large. CPU-bound queries are uninteresting in our experiments since our primary purpose is to compare the performance of different I/O configurations. In the original Wisconsin Benchmark, the tuple size is very small, only 200 bytes, which makes all the sequential scan queries in the benchmark CPU-bound. In our version of the benchmark, we have introduced an extra text field to pad each tuple to $1,000$ bytes so that the sequential scan queries become IO-bound.

On the other hand, index scans using an unclustered index are all IO-bound because they do not need to examine every tuple in a fetched disk block. Rather, they can directly access the qualified tuple in a block through an index pointer. The largest relations, tenk1 and tenk2 in the Wisconsin Benchmark only have $10,000$ tuples. We have increased the number of tuples in these relations to $20,000$ so that each query will run longer and we can get more stable results. Specifically, we have used the following query for both sequential scans and index scans:

retrieve (twentyk.all) where twentyk.unique1 < 2648 and twentyk.unique1 > 647.

For index scan experiments, an unclustered index is defined on attribute unique1. We also use the same qualifications for updates.

Because XPRS runs on top of the Dynix file system, it is difficult for us to implement seek scheduling for the mirror configuration. Our experiments are run without seek scheduling. This understates the performance results for the mirror configuration to be presented below. However, we make a special effort to achieve better load balancing for this configuration by directing I/O requests from the even number slave backends to one set of disks and those from the odd number slave backends to the disks' mirrored partners.

5.2.2 Experiment Results

The experiment results are presented in Figures 5.2 - 5.7. For the retrieve queries, we plot the query elapsed time against the degree of parallelism. Figures 5.2 - 5.4 show

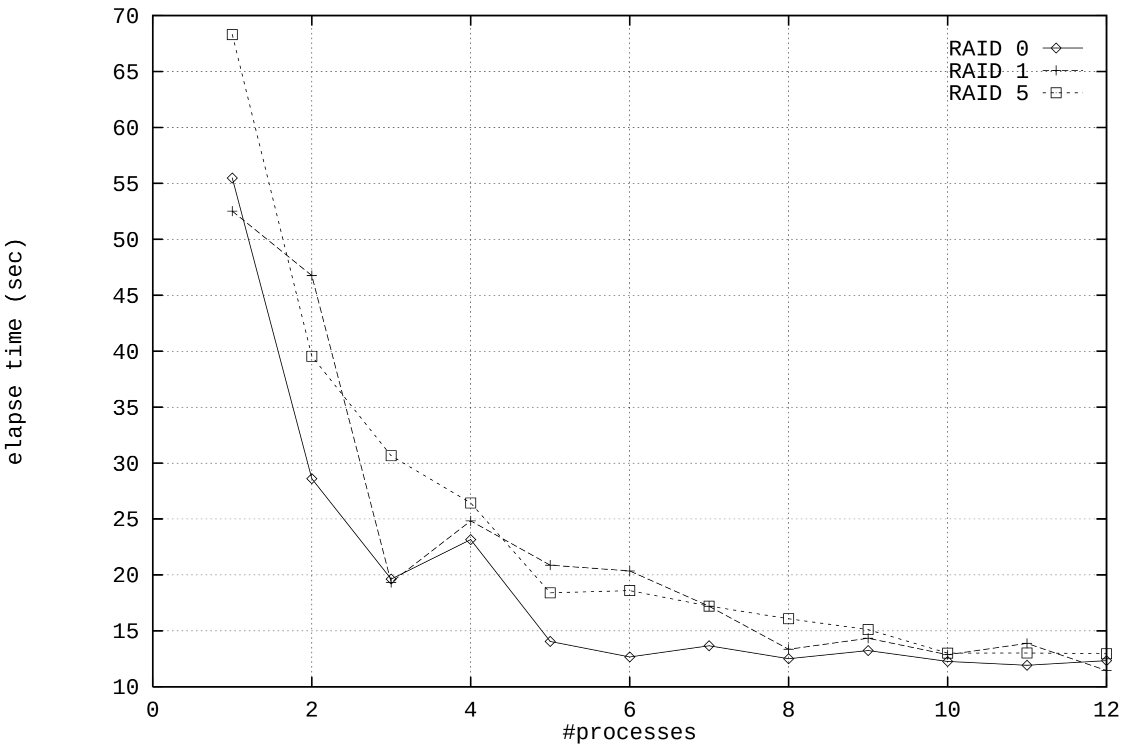

Figure 5.2: **Performance of Seq. Scan $8K$ blocks, Fixed Total Capacity**

the performance of read-oriented queries on fixed total capacity configurations. Figures 5.5 and 5.6 report the performance for the fixed capacity configurations. Figure 5.7 shows the performance of the fixed capacity configurations normalized by the number of disks used.

The performance of sequential scan is shown in Figure 5.2. This figure shows that when the degree of parallelism is low, both the mirror and the parity configuration perform worse than the simplex configuration. However, when the degree of parallelism increases, all three configurations have approximately the same performance. This is due to the fact that the default block size (i.e., the stripe unit size) of XPRS is only $8K$ bytes. When the degree of parallelism is low relative to the size of a stripe, the access pattern to each disk in the simplex configuration is more or less sequential. Therefore, it takes advantage of the

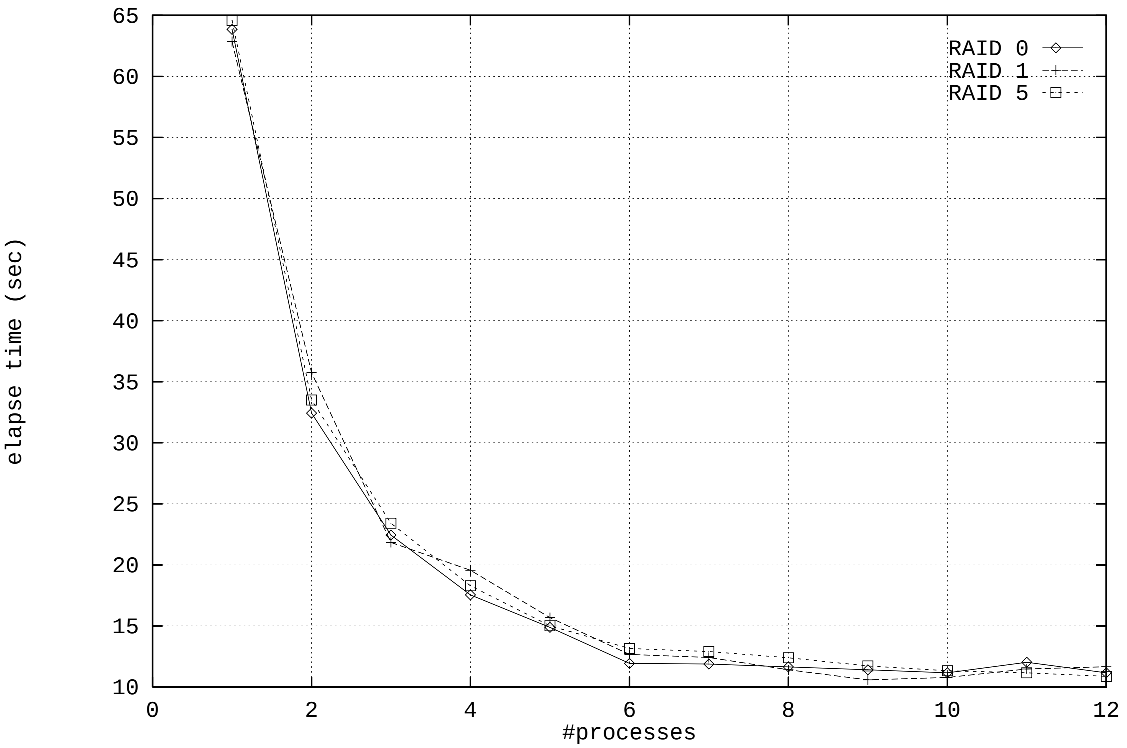

Figure 5.3: **Performance of Seq. Scan $32K$ blocks, Fixed Total Capacity**

read-ahead in the Dynix file system.

On the other hand, in the parity array configuration, the parity blocks are scattered across all the disks. Thus each disk has to seek past the parity blocks frequently, which turns off the file system read-ahead. The performance loss of the mirror configuration can be explained by its smaller stripe size (half of simplex's), hence the access pattern on each disk becomes random at lower degrees of parallelism. Another factor for the mirror configuration is the way we assign even number slaves to one set of disks and odd number slaves to the mirror set does not balance the I/O requests perfectly. When the degree of parallelism becomes large, access patterns to each disk in all configurations become random and thus there is essentially no performance difference between different configurations.

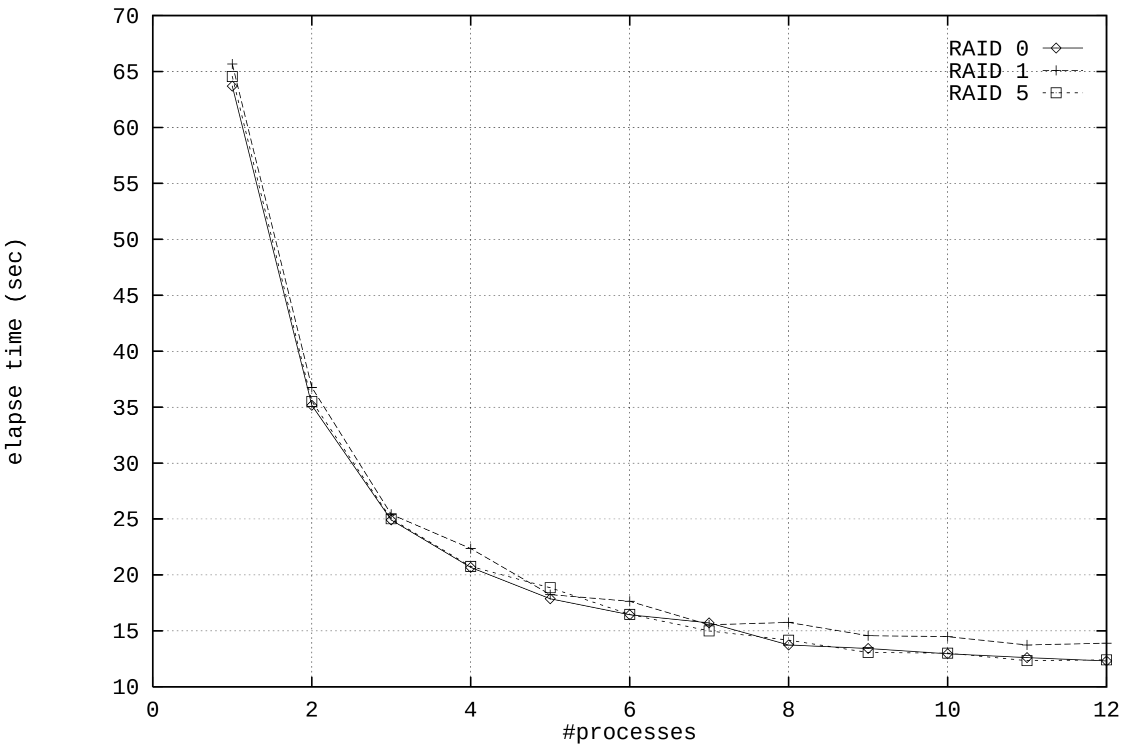

Figure 5.4: **Performance of Index Scan, Fixed Total Capacity**

Figure 5.3 shows the performance of the same query when the block size of XPRS is increased to $32K$ bytes. In this case, all three configurations have almost the same performance because they all enjoy a sequential I/O bandwidth through the large track-size blocks.

Figure 5.4 shows the performance of an unclustered index scan query on the three disk array configurations. It shows that the three configurations have virtually the same performance for random reads when the degree of parallelism is low. For higher degrees of parallelism, the mirror configuration starts to show performance that is a little worse than the other two configurations. This can be explained by the small stripe size of the mirror configuration. Because the stripe size is small, it is more likely that two processes try to

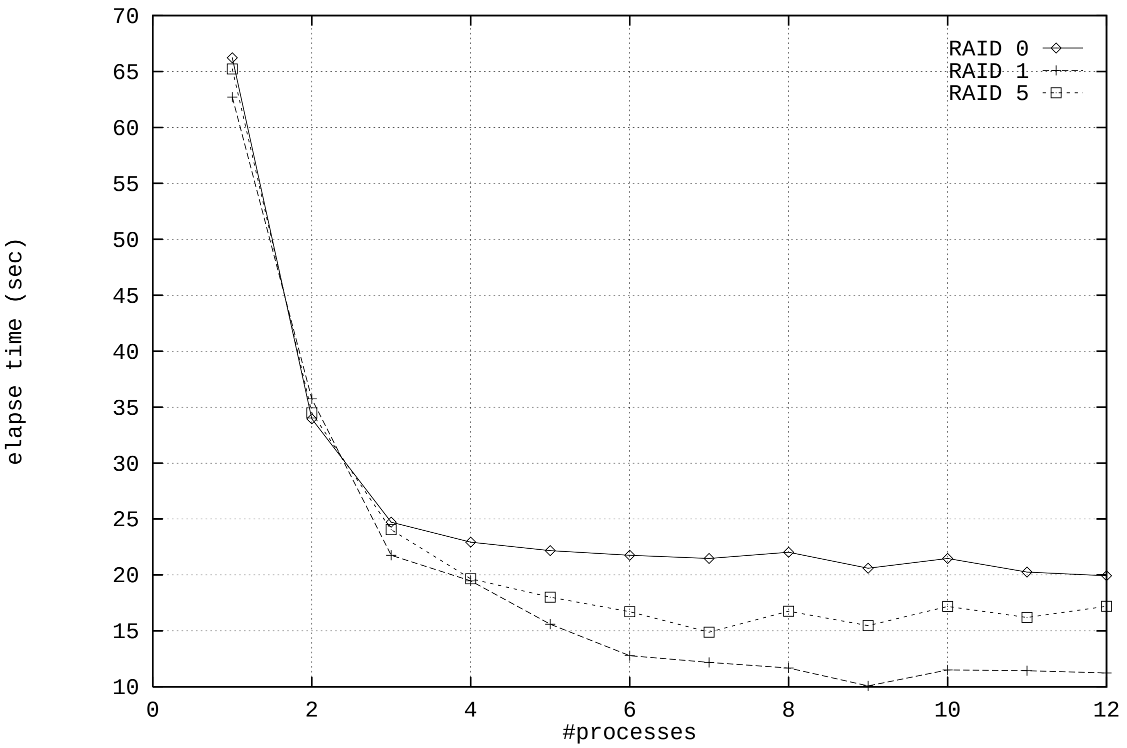

Figure 5.5: **Performance of Seq. Scan $32K$ blocks, Fixed User Capacity**

access blocks from the same disk at the same time, thereby causing more queuing delays.

In summary, if we fix the total capacity, all three configurations can achieve the same query processing performance. Although the parity array configuration may have the problem of losing sequential bandwidth because of the scattered parity blocks, it can be avoided by increasing block size. If we compare these three configurations with fixed total capacity for read-oriented queries, they can achieve the same performance, but the simplex configuration has poor availability and the mirror configuration can only offer half the user capacity; therefore the parity array configuration is obviously the best choice.

On the other hand, if we fix the user capacity, as in Figures 5.5 and 5.6, the mirrored configuration has the best performance. This is not too surprising, since the mirror

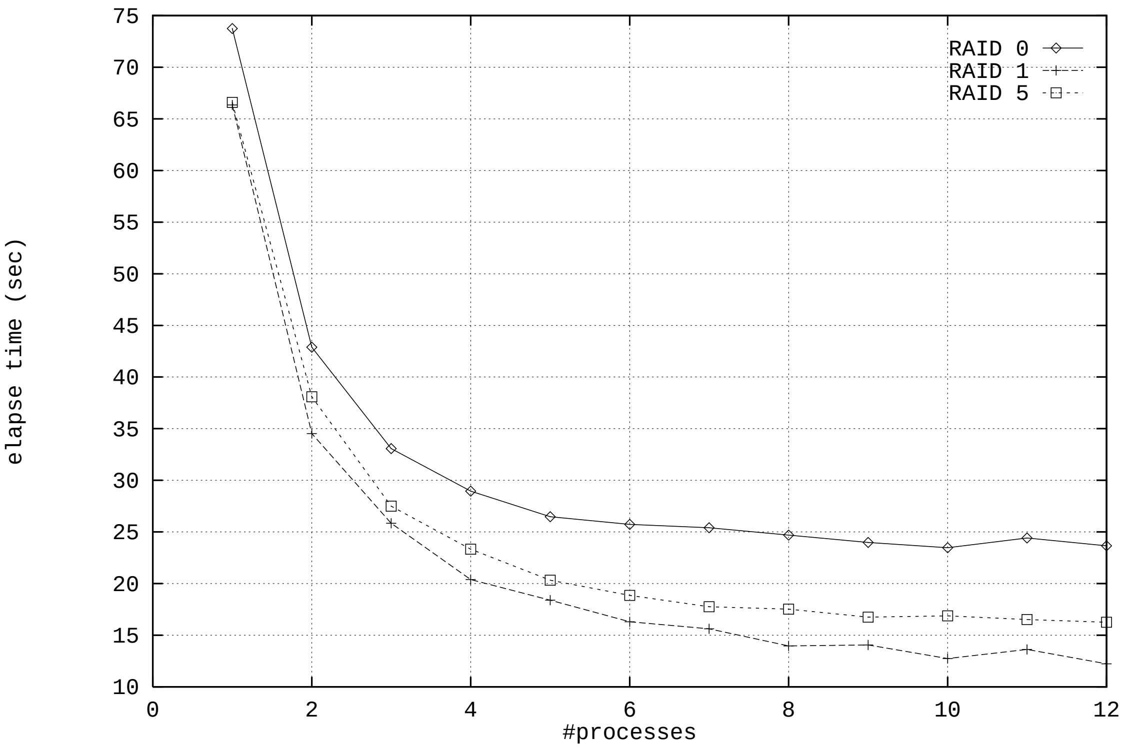

Figure 5.6: **Performance of Index Scan, Fixed User Capacity**

configuration has almost twice as many disks as the other two. However, Figure 5.7 shows if we normalized the performance in terms number of I/Os per second to performance per disk, the parity array configuration has the best performance per disk except for low degrees of parallelism. For high parallelism, the parity array configuration achieves better performance than simplex because it spreads data across more disks. However, this advantage is not obvious at low parallelism because there are not enough outstanding I/O requests. Although the mirror configuration has the best raw performance in this case, it also wastes the most space for redundancy. As a result, it has lower normalized performance.

Performance of write-oriented queries on the three disk array configurations is shown in Figure 5.8, in which we present the measured elapsed time of updates to 1,000

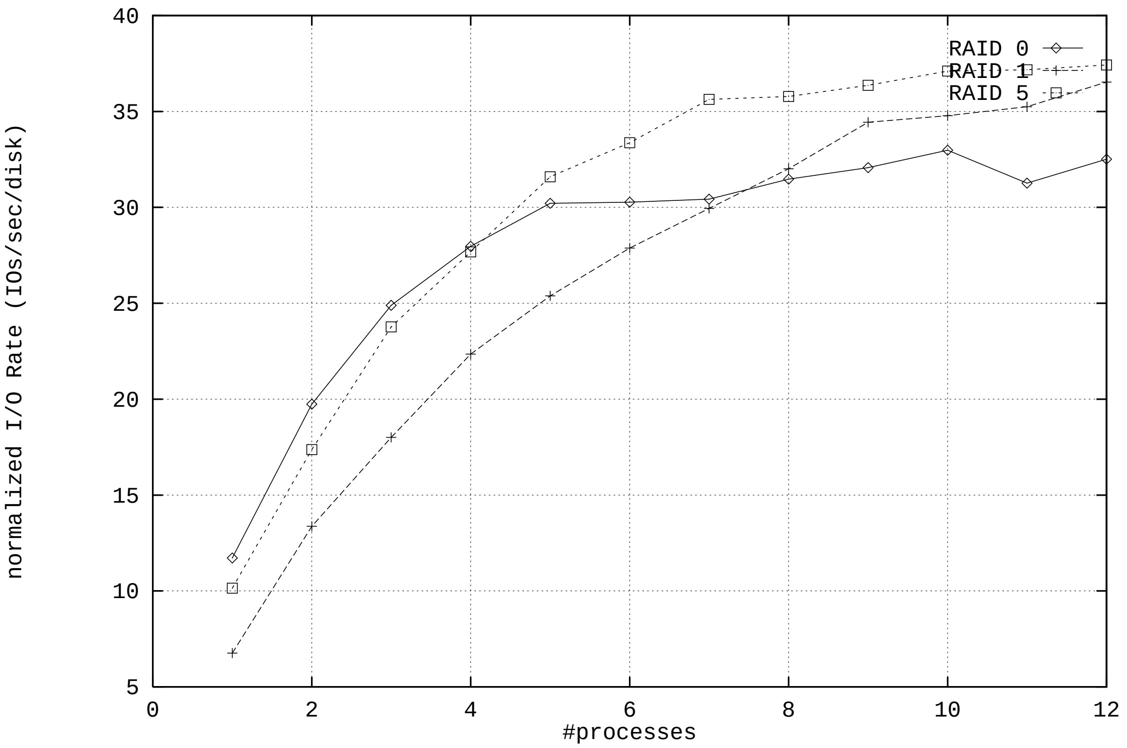

Figure 5.7: **Normalized Performance of Index Scan, Fixed User Capacity**

tuples through sequential scan or unclustered index scan in a table. To effectively make use of the parity array configuration, a special *stripe buffer* is implemented which buffers blocks in the same stripe and writes out an entire stripe at the same time if possible so that we can avoid reading the old data block and the old parity block to compute the new parity block for each write. Obviously, this stripe buffer is only effective for sequential writes. As we can see in the table, the parity array configuration has the worst performance for random updates because of its read-modify-write scheme. However, parity arrays perform much better for sequential updates with the help of the stripe buffer. In fact, it can even perform better than the mirror configuration because it has more disk drives that can write in parallel.

Elapse Time (sec)	Fixed Total Capacity			Fixed User Capacity		
	simplex	mirror	array	simplex	mirror	array
	(6)	(2×3)	$(5+1)$	(3)	(2×3)	$(3+1)$
sequential update	52.55	53.84	53.07	50.99	54.26	51.68
random update	53.44	56.34	73.42	54.09	55.28	71.86

Figure 5.8: **Performance of Update Queries**

5.3 Summary

In this chapter, we have compared the performance of three RAID configurations, simplex, mirrored disks and parity arrays in a parallel query processing environment through experiments on XPRS. From the experiment results, we have shown that for a fixed total disk space, the parity array configuration can achieve the same read performance as the mirror configuration while providing almost twice as much user disk space. For a fixed user disk space, the parity array does not win on raw performance, but it wins on normalized performance for read-oriented applications. For write-oriented applications, the parity array configuration can achieve comparable performance with other configurations for sequential writes, but loses in random writes. However, a write-optimized file system such as Log Structured File System (LFS) [44] can help solve the random write problem of parity arrays by grouping random writes and turning them into sequential writes.

This chapter completes our discussion on parallel query processing in a shared everything environment. Although different disk array configurations have different performance characteristics, the difference does not affect the results presented in the previous chapters. Our parallel query processing strategies can be applied in a shared everything system with any disk array configuration.

Chapter 6

Conclusions and Future Work

This chapter concludes the whole thesis. Our conclusions are also followed by some discussions on future research directions.

It is the general consensus that parallel database systems are the key to high performance data management demanded by applications such as decision support systems and multi-media data management. However, parallel database systems also introduce new problems to query processing especially for query optimization and resource management. If these problems are not well solved, parallel database systems still cannot achieve high performance. This thesis solves these problems in parallel query processing in a particular environment, i.e., the shared everything enironment and presents a complete design and implementation of parallel query processing in our prototype multi-user parallel database system, XPRS, with focus on three main issues: parallel query optimization, task scheduling and memory allocation.

Query optimization has been traditionally handled with an exhaustive or semiexhaustive search algorithm to find the optimal query execution plan. Although this approach is reasonable for sequential query plans, it is bound to fail for parallel query plans because the search space of possible parallel query plans is orders of magnitude larger than the space of sequential query plans. Another problem that has not been well solved in query optimization is the unknown parameter problem. In a multi-user environment, many parameters that affect query execution costs are unknown until run time. Therefore, compile-time optimization must deal with these unknown parameters. XPRS specifically considers two parameters, available buffer size and number of free processors. Both the enormous search space problem and the unknown parameter problem are solved in XPRS by a two phase optimization strategy. The first phase of our two phase optimization strategy only optimizes sequential query execution plans based on fixed parameters at compile time, and the second phase optimizes parallelizations of the chosen sequential plan from the first phase based on resource availabilities at run time. Experiment results from XPRS benchmarks have shown that this two phase optimization strategy almost always produce an optimal or close-to-optimal parallel query plan.

In parallel query processing, there may be multiple tasks that are ready to run at the same time. Therefore, an optimal processing schedule need be decided for these tasks such that the total processing time is minimized. This scheduling problem is different from the conventional multiprocessor scheduling problem. In this scheduling problem, both parallelism between multiple tasks and parallelism within each task must be considered to achieve maximum performance. A simple and efficient adaptive scheduling algorithm has been proposed as a solution to this task scheduling problem. The scheduling algorithm is based on the concept of an IO-CPU balance point that maximizes system resource utilizations. It executes an IO-bound task and a CPU-bound task at their IO-CPU balance and dynamically adjusts the degree of parallelism of the running tasks to keep the system running at the IO-CPU balance point. Experiment results have also confirmed the efficiency and effectiveness of this scheduling algorithm.

Main memory is also one of the most important resources in parallel query processing along with processors and I/O bandwidth. When multiple tasks are running in parallel, the problem of how to optimally allocate a limited amount of memory to these tasks arises. This thesis specifically deals with memory allocation between parallel or pipelined hashjoin operations. The optimal memory allocation between multiple hashjoins is discovered through careful cost analysis on hashjoin executions. The memory allocation strategy of XPRS always allocates the optimal amount of memory to parallel hashjoins and also dynamically adjust the memory allocation to the new optimal allocation when one hashjoin finishes and a new one starts. Because the memory allocation strategy follows the same framework as the task scheduling algorithm, it can be easily integrated into the task scheduling algorithm.

A theme through the approach to parallel query processing presented in this thesis is adaptiveness, which is crucial for a multi-user environment where the amount of available resources is unpredicatable. In this approach, query plans can be dynamically adjusted through Choose nodes according to buffer space availability; parallelism can be dynamically adjusted to fully utilize both the processors and disk bandwidth; and memory allocation can be dynamically adjusted to optimally utilize available main memory. We believe that this thesis presents the first complete approach to parallel query processing that integrates optimization and resource allocation and has the ability to adjust to the changing environment.

A shared everything parallel database system relies on disk arrays to provide enough I/O bandwidth for parallel query processing. This thesis also presents a performance study of different disk array configurations based XPRS. The experiment results show that parity arrays (RAID Level 5) can provide comparable read performance as other configurations while providing high availability and high capacity efficiency. Parity arrays can also provide comparable write performance if used under a write-optimized file system such as LFS.

Although this thesis presents results that constitutes a step forward towards making parallel database systems highly effective and widely available, there are still many research issues which need to be addressed in the future and among which only a few are discussed below.

- **Inter-Query Optimization**

  This thesis has only considered optimization of a single query. However, in a multi-user environment, multiple queries may be issued simultaneously by different users, or in a transaction processing environment, each transaction may consist of multiple queries. Thus, it is desirable to optimize multiple queries at the same time. Although multiple query optimization has been studied before, the focus was on sharing the evaluation of common subexpressions. There are a lot more issues in this subject especially in a parallel database. For example, even though two queries do not contain any common subexpressions, it is still possible to run them together to achieve better system resource utilization. The results on task scheduling and memory allocation in this thesis are a good starting point for further research on this topic.

- **Parallel Updates**

  Currently, XPRS only parallelizes data retrievals. Loading and updates are still handled sequentially. However, loading and update performance is crucial for large databases. Therefore, it is important to design parallel algorithms for loading, index construction and maintenance, and large updates. For a multi-user environment, this also leads to more issues in query optimization and resource management, such as how to optimize and schedule queries while loading is proceeding.

- **Memory Allocation for General Operations**

  The memory allocation problem has only been solved for hashjoin operations in this thesis. Although hashjoins are shown to be a very efficient hashjoin method, the memory allocation problem also needs to be solved for general operations. This will require more cooperation between the database buffer manager and the query optimizer and executor, thus lead to many new research issues.

- **New Architectures**

  Because of the limitation on scalability of a shared everything system, sooner or later, a shared everything system will become a node in a shared nothing system. More research needs to be done before a unified approach for parallel query processing in such a hybrid environment can be designed. Another promising architecture is the massively parallel machines manufactured by vendors such as Thinking Machines, nCUBE, and KSR. New algorithms need to be developed to fully utilize the massive parallelism provided by such architectures.

Bibliography

[1] *Proc. 12th International Conference on Very Large Data Bases*, Kyoto, August 1986.

[2] A. Bhide and M. Stonebraker. A Performance Comparison of Two Architectures for Fast Transaction Processing. In *Proc. 4th IEEE International Conference on Data Engineering*, Los Angeles, February 1988.

[3] Dina Bitton, Haran Boral, David DeWitt, and Kevin Wilkinson. Parallel algorithms for the execution of relational database operations. *ACM Transactions on Database Systems*, 8(3):324–353, Sep 1983.

[4] Dina Bitton, David DeWitt, David Hsiao, and Jai Menon. A taxonomy of parallel sorting. *Computing Surveys*, 16(3), Sep 1984.

[5] M. Blasgen and et. al. The convoy phenomenon. *Operating System Review*, 13(2), April 1979.

[6] M.W. Blasgen and K.P. Eswaran. Storage and access in relational data bases. *IBM Systems Journal*, 16(4):363–377, 1977.

[7] H. Boral, 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, 2(1), March 1990.

[8] H. Boral and D. DeWitt. Database machines: An idea whose time has passed? a critique of the future of database machines. In H.-O. Leilich and M. Missikoff, editor, *the 1983 Workshop on Database Machines*. Springer-Verlag, 1983.

[9] G. Bultzingsloewen. Optimizing sql queries for parallel execution. *SIGMOD Record*, 18(4), December 1989.

[10] P. Chen, G. Gibson, R. Katz, and D. Patterson. An evaluation of redundant arrays of disks using amdahl 5890. In *ACM SIGMETRICS Conference*, Boulder, CO, May 1990.

[11] Hong-Tai Chou and David DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proc. 11th International Conference on Very Large Data Bases*, October 1984.

[12] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data Placement in Bubba. In Proc. SIGMOD 88 [39].

[13] D. Cornell and P. Yu. Integration of buffer management and query optimization in relational database environment. In *Proc. 15th International Conference on Very Large Data Bases*, pages 247–255, Amsterdam, The Netherlands, August 1989.

[14] D. DeWitt D.A. Schneider. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In Proc. SIGMOD 89 [40].

[15] David DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[16] David J. DeWitt, R.H. Gerber, G. Graefe, M.L. Heytens, K.B. Kumar, and M. Muralikrishna. Gamma-a high performance dataflow database machine. In *Proc. 12th International Conference on Very Large Data Bases* [1].

[17] David J. DeWitt, Shahram Ghandeharizadeh, Donovan Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, Mar 1990.

[18] R. Fagin and et al. Extendible hashing – a fast access method for dynamic files. *ACM Transactions on Database Systems*, September 1979.

[19] G. A. Gibson. *Performance and Reliability of Disk Arrays*. PhD thesis, University of California at Berkeley, January 1991.

[20] Goetz Graefe. Rule-based query optimization in extensible database systems. Computer Sciences Technical Report #724, University of Wisconsin at Madison, November 1987.

[21] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 102–111, Atlantic City, May 1990.

[22] Goetz Graefe and Karen Ward. Dynamic query evaluation plans. In Proc. SIGMOD 89 [40], pages 358–366.

[23] Waqar Hasan and Hamid Pirahesh. Query Rewrite Optimization in Starburst. Research Report RJ 6367 , IBM Almaden Research Center, August 1988.

[24] Yannis Ioannidis, Raymond Ng, Kyuseok Shim, and Timos Sellis. Parametric query optimization. In *1992 VLDB Conference*, 1992.

[25] Yannis Ioannidis and Eugene Wong. Query optimization by simulated annealing. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 9–22, San Francisco, CA, May 1987.

[26] ISO_ANSI. Database Language SQL ISO/IEC 9075:1992, 1991.

[27] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[28] Donald Ervin Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley Publishing Co., 2nd edition, 1973.

[29] R. Kooi. Query optimization in ingres. *IEEE Database Engineering*, 5(3):2–5, 1982.

[30] Henry F. Korth and Abraham Silberschatz. *Database System Concepts*. McGraw-Hill Book Company, 1986.

[31] N. Kronenberg, H. Levy, and W. Strecker. Vaxclusters: A closely-coupled distributed system. *ACM Transactions on Computer Systems*, 4(2), May 1986.

[32] Hongjun Lu, Ming-Chien Shan, and Kian-Lee Tan. Optimization of multi-way join queries for parallel execution. In *Proc. 17th International Conference on Very Large Data Bases*, Barcelona, September 1991.

[33] Hongjun Lu, Kian-Lee Tan, and Ming-Chien Shan. Hash-based join algorithms for multiprocessor computers with shared memory. In Proc. VLDB 90 [42].

[34] Lothar Mackert and Guy Lohman. Index scans using a finite lru buffer: A validated i/o model. *ACM Transactions on Database Systems*, 14(3):401–424, September 1989.

[35] Jai Menon. A study of sort algorithms for multiprocessor database machines. In *Proc. 12th International Conference on Very Large Data Bases* [1].

[36] M. Murphy and M. Shan. Execution plan balancing. In *1991 IEEE Data Engineering Conference*, 1991.

[37] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In Proc. SIGMOD 88 [39].

[38] Hamid Pirahesh, C. Mohan, J. Cheng, T.S. Liu, and Patricia Selinger. Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches. In *Proc. 2nd International Symposium on Databases in Parallel and Distributed Systems*, Dublin, July 1990.

[39] *Proc. ACM-SIGMOD International Conference on Management of Data*, Chicago, June 1988.

[40] *Proc. ACM-SIGMOD International Conference on Management of Data*, Portland, May-June 1989.

[41] *The Design of XPRS*, Los Angeles, August-September 1988.

[42] *Proc. 16th International Conference on Very Large Data Bases*, Brisbane, August 1990.

[43] D. Ries and R. Epstein. Evaluation of distribution criteria for distributed database systems. UCB/ERL Technical Report M78/22, University of California at Berkeley, May 1978.

[44] Mendel Rosenblum and John Ousterhout. The lfs storage manager. In *the Summer 1990 USENIX Technical Conference*, Anaheim, CA, June 1990.

[45] K. Salem and H. Garcia-Molina. Disk striping. In *1986 IEEE Data Engineering Conference*, Los Angeles, CA, February 1986.

[46] D. Schneider and D. Dewitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In Proc. VLDB 90 [42].

[47] Patricia G. Selinger, M. Astrahan, D. Chamberlin, Raymond Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Boston, June 1979.

[48] Leonard D. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems*, 11(3), September 1986.

[49] D. Skeen. Nonblocking commit protocols. In *Proc. ACM-SIGMOD International Conference on Management of Data*, June 1981.

[50] Michael Stonebraker. The case for shared nothing. *Database Engineering*, 9(1), 1986.

[51] Michael Stonebraker and Greg Kemnitz. The Postgres Next Generation DBMS. *Communications of the ACM*, 34(10), October 1991.

[52] J. Strickland, P. Uhrowczik, and V. Watts. IMS/VS: An Evolving System. *IBM Systems Journal*, 21(4), 1982.

[53] Mark Sullivan. *System Support for Software Fault Tolerance in Highly Available Database Management Systems*. PhD thesis, University of California at Berkeley, August 1992.

[54] Arun Swami and Anoop Gupta. Optimization of large join queries. In Proc. SIGMOD 88 [39], pages 8–17.

[55] Teradata. DBC/1012 Data Base Computer Concepts and Facilities - Release 3.1. Document Number C02-0001-05, Teradata Corp., 1988.

[56] The Tandem Database Group. NonStop SQL: A Distributed, High-Performance, High-Availability Implementation of SQL. In *2nd International Workshop on High Performance Transaction Systems*, Asilomar, September 1987.

[57] Patrick Valduriez and Georges Gardarin. Join and semijoin algorithms for a multiprocessor database machine. *ACM Transactions on Database Systems*, 9(1):133–161, mar 1984.

[58] Joel Wolf, Daniel Dias, and Philip Yu. An effective algorithm for parallelizing sort merge joins in the presence of data skew. In *2nd International Symposium on Databases in Parallel and Distributed Systems*, pages 103–115, Dublin, Ireland, 1990.

[59] E. Wong and K. Youssefi. Decomposition - a strategy for query processing. *ACM Transactions on Database Systems*, 1(3):223–241, September 1976.

[60] Hansjorg Zeller and Jim Gray. An Adaptive Hash Join Algorithm for Multiuser Environments. In Proc. VLDB 90 [42].