Parallel Query Processing Using Shared Memory Multiprocessors and Disk Arrays

by

Wei Hong

Memorandum No. UCB/ERL M93-29

27 April 1993

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California
94720

PARALLEL QUERY PROCESSING USING SHARED MEMORY MULTIPROCESSING AND DISK ARRAYS

by

Wei Hong

Memorandum No. UCB/ERL M93/28

27 April 1993

Parallel Query Processing Using Shared Memory Multiprocessors and Disk Arrays

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 pa ular, 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 in-

cluding 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: *nestloop*, *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

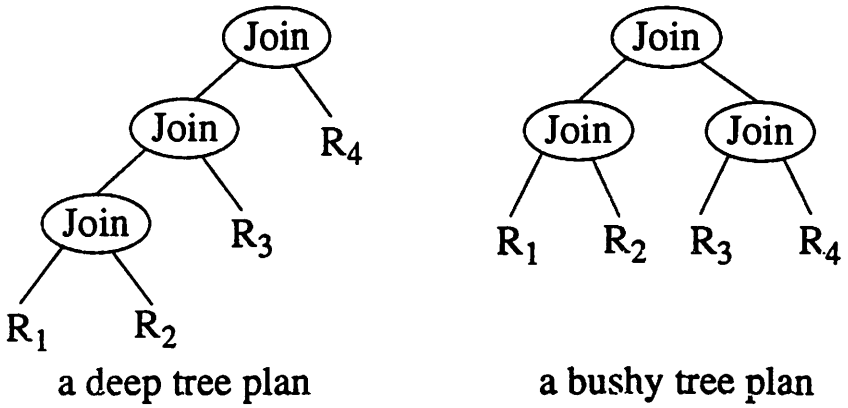

Figure 1.1: **Deep Tree Plan v.s. Bushy Tree Plan**

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 subsec n, 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

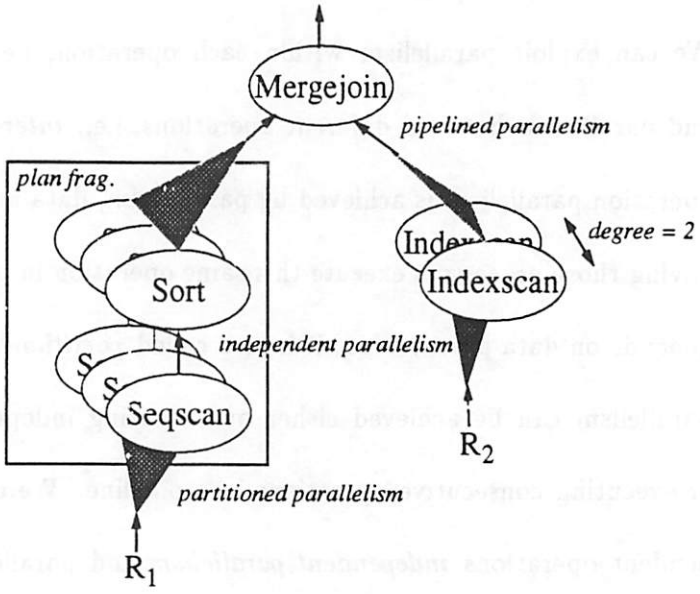

Figure 1.2: **An Example Parallel Plan**

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:

Site 1: $salary < 10K$
Site 2: $10K \leq salary < 20K$
Site 3: $20K \leq salary < 30K$
Site 4: $30K \leq salary < 40K$
Site 5: $salary \geq 40K$.

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

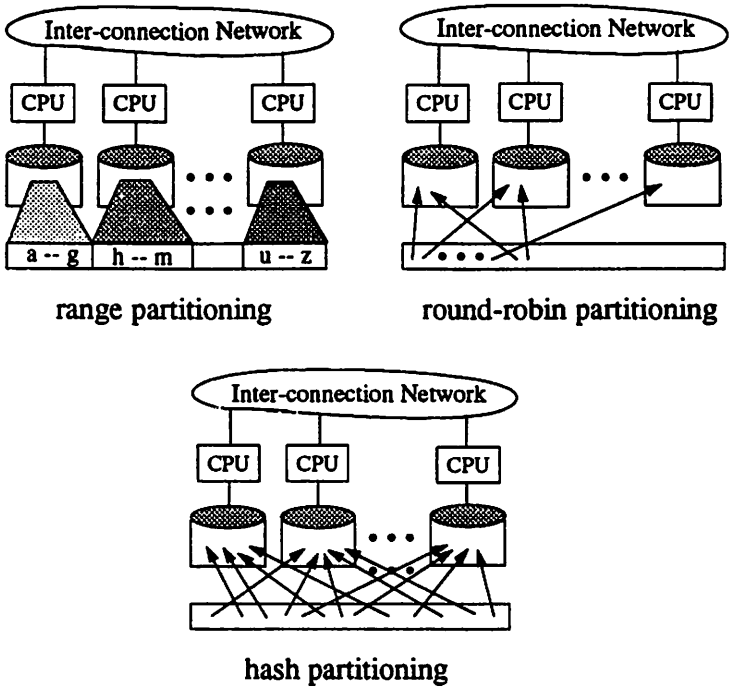

Figure 1.3: **Three Basic Declustering Schemes**

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

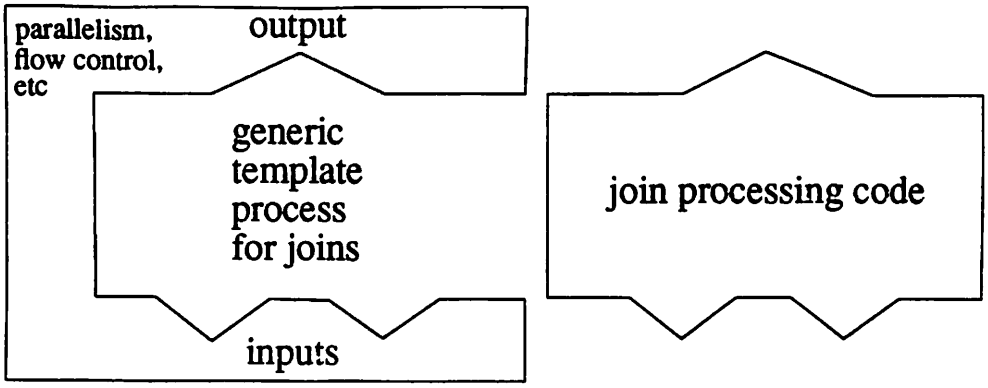

Figure 1.4: **Bracket Model of Parallelization**

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

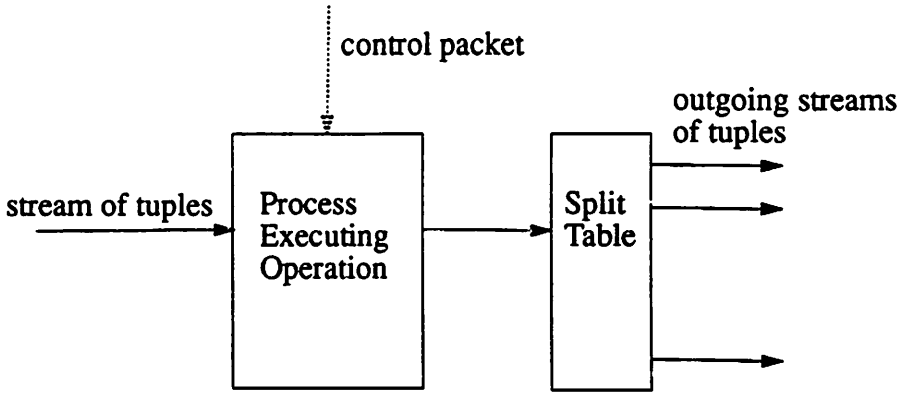

Figure 1.5: **Generic Template for Parallelization in Gamma**

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.

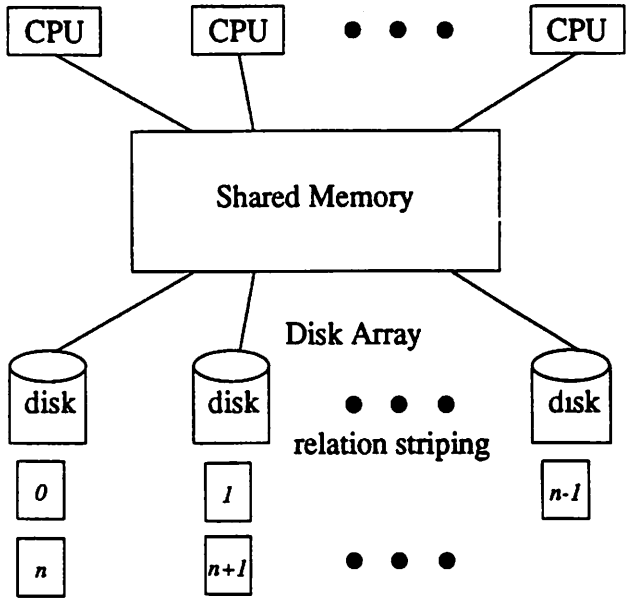

Figure 1.7: **The Overall Architecture of XPRS**

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 sys-

tem. 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

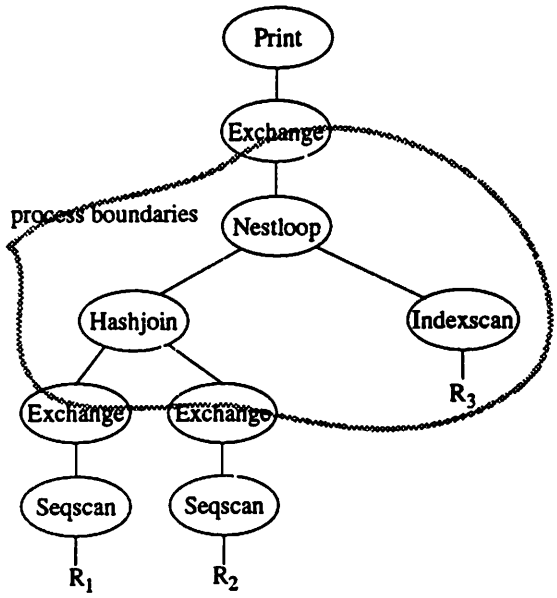

Figure 1.8: **Query Plan with Exchange Operators**

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 algo-

rithms 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,

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

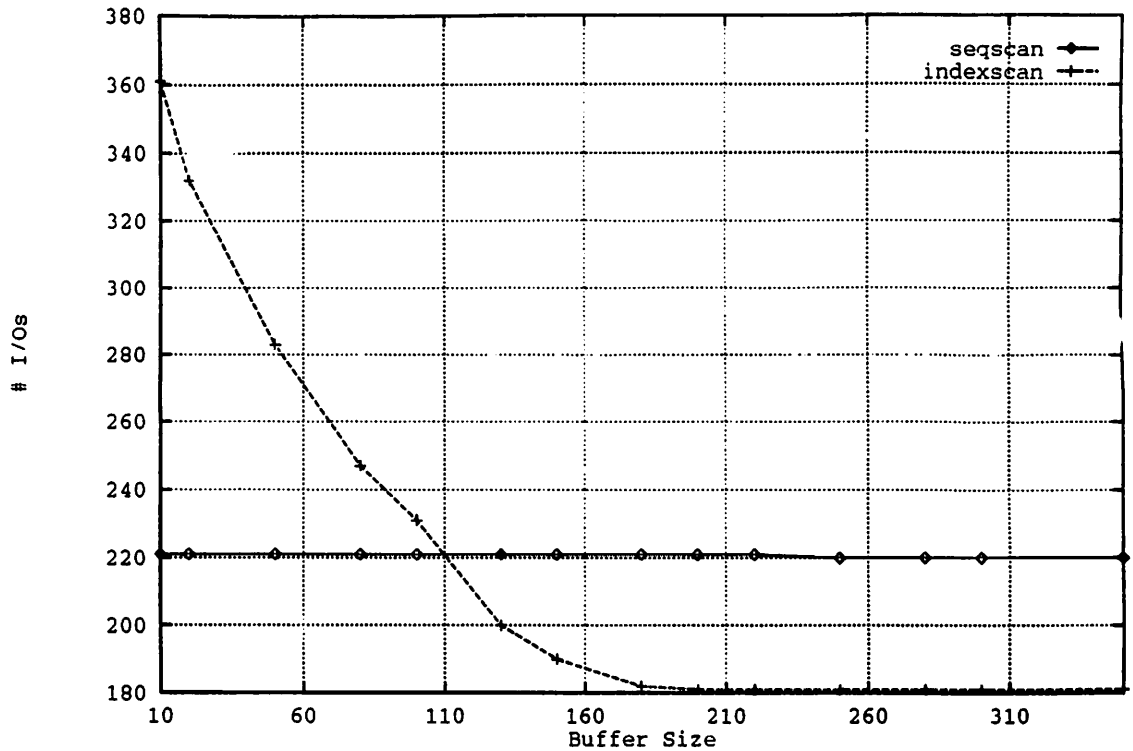

Figure 2.1: **Cost of SeqScan v.s. IndexScan**

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:

retrieve (tenk1.all) where tenk1.u1 > 110 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

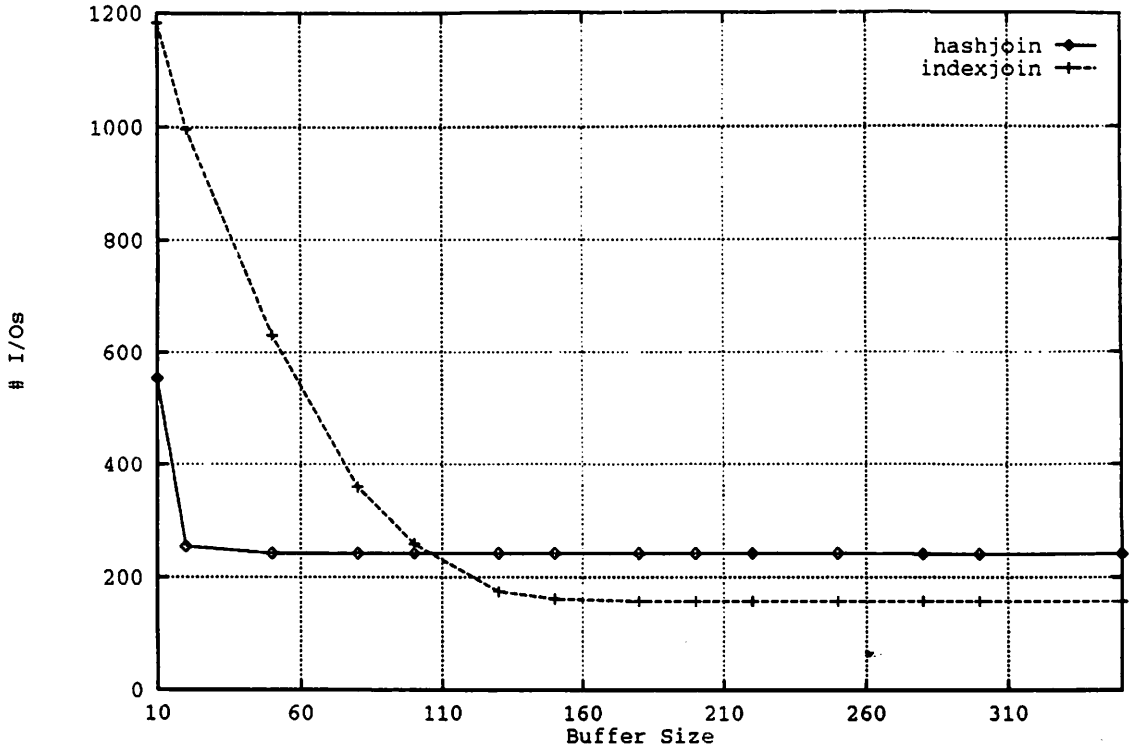

Figure 2.2: **Cost of Nestloop with Index v.s. Hashjoin**

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.