Resource Allocation and Scheduling for Mixed Database Workloads

Kurt P. Brown
Michael J. Carey
David J. DeWitt
Manish Mehta
Jeffrey F. Naughton

Technical Report #1095

July 1992

Resource Allocation and Scheduling for Mixed Database Workloads

Kurt P. Brown
Michael J. Carey
David J. DeWitt
Manish Mehta
Jeffrey F. Naughton

Computer Sciences Department
University of Wisconsin
1210 West Dayton Street
Madison, WI 53706

ABSTRACT

This paper investigates resource management and scheduling issues that arise when a relational DBMS is faced with a workload containing both short on-line transactions and long-running queries. Relevant issues include (i) how memory should be divided among the transactions and queries, (ii) the impact of serial versus concurrent execution of long-running queries, and (iii) tradeoffs between different scheduling strategies for complex multi-join queries. Through a series of simulation experiments based on a detailed DBMS simulation model, we show that previously proposed resource allocation and scheduling algorithms are inadequate for handling mixed workloads. One particularly interesting example of this occurs when running a query together with a stream of transactions: in many cases, beginning with the memory allocation deemed optimal by previously proposed buffer management schemes, taking memory *away* from the query improves the performance of both the transactions *and* the query.

This research was partially supported by IBM Corporation through a Research Initiation Grant and by the National Science Foundation under grants IRI-8657323 and IRI-9157357.

1. INTRODUCTION

The current generation of relational database management systems are capable of providing acceptable performance for either on-line transaction processing or decision-support applications. However, the problem of achieving good performance for workloads that involve a *mix* of these activities is still open. The majority of research on the mixed workload problem deals with techniques for handling the data contention that can arise between short on-line transactions and long-running decision-support queries [DuBo82, Chan82, Care86, Agra89, Wu91, Bobe92a, Bobe92b, Merc92, Moha92, Srin92]. Very little work has been published dealing with the issues of resource allocation and/or scheduling in a mixed workload environment. Some of the issues that arise when considering a mixed workload are how best to allocate processors, memory, and disks between the different workload classes so as to optimize overall DBMS performance without penalizing any specific workload class; determining the appropriate multiprogramming level for each workload class to achieve the best balance between competing resource requirements; choosing the optimal granularity and degree of concurrency with which to execute any individual query while accounting for the effects of possible increased resource consumption on other competing queries; and so on.

In this paper, we will show that many previously proposed algorithms for DBMS resource allocation and scheduling are inadequate for mixed workloads, and that new approaches are required that take into account more global knowledge about the workload. We will demonstrate that in many cases it is possible to significantly reduce the response times of both on-line transactions *and* long-running queries relative to their performance under previous algorithms. The paper will examine three key issues using a detailed simulation model: how memory should be divided among long-running queries and short transactions, the benefits and costs of concurrent versus serial execution of multiple long-running queries, and tradeoffs between different scheduling strategies for complex, multi-join queries. In each case, our objective will be to explain the shortcomings of existing approaches, quantitatively as well as qualitatively, and to highlight the tradeoffs that any new, comprehensive algorithm must deal with. Our eventual goal is to address these (and other) issues in the context of parallel database systems. This paper focuses on centralized DBMS issues as a first step towards that goal.

Some of the tradeoffs examined in this paper are complex and non-intuitive. As an example, consider the problem of memory allocation for a workload that consists of a single long-running query containing one or more joins, running concurrently with a stream of small, "rifle shot" transactions. Assume that each small transaction accesses a single tuple from a common relation via an index lookup, and that the join query accesses separate relations, so there is no data contention between the two workload classes. Since there is no locality of reference within any given small transaction, previously proposed algorithms for relational database buffer management, such as [Effe84, Chou85, Sacc86, Corn89, Ng91, Falo91], would allocate just one page (or at most a few pages) to each small transaction, allocating the remainder of the available memory to the join query. The assumption that justifies this allocation is that there is greater marginal benefit in assigning free buffers to the long-running query because join processing algorithms can reduce the number of I/Os required in exchange for additional memory. Our results will show that in a mixed workload environment, such an allocation may be suboptimal — not only for the short transactions, but for the large join as well. In many instances, taking memory *away* from the join can actually improve the performance of both the short transactions *and* the join.

Another example of the multiclass tradeoffs of interest here is in the area of complex, multi-join query scheduling. Most previously proposed complex query scheduling strategies [Murp89, Schn90, Murp91, Wils91, Chen92] attempt to maximize the utilization of resources in order to achieve minimum response times for complex queries, typically assuming that the entire system is available for allocation to any individual query. For the example workload sketched in the preceding paragraph, one would expect that such query scheduling strategies would tend to optimize the performance of the long-running query at the expense of the short transactions. However, as discussed in the memory allocation example above, our results will show that this approach is harmful for the both the short transactions *and* the query. In most situations, less resource intensive strategies provide better performance for both workload classes.

The remainder of this paper is organized as follows: In Section 2, we survey related work and explain why it fails to adequately address the problem of mixed workload resource allocation and scheduling. Section 3 describes the simulation model that we have constructed for use in our work. Section 4 starts by using this model to study the performance of a very simple instance of a mixed DBMS workload under several different hardware configurations and transaction loads. The set of experiments is then expanded to include multiple join streams and other variations. Section 5 shows how a fairly simple analytical model can be used to capture the key performance behavior of the system; such a model could be very effective as a tool for use in run-time decision-making. Our initial conclusions and plans for future work are then summarized in Section 6.

2. RELATED WORK

Besides the large volume of work related to data contention cited in the introduction, there has been little published work that specifically addresses mixed workloads. A recent issues paper by Pirahesh et al [Pira90] discussed the mixed workload problem, but other than suggesting that a resource governor and a concept of priority are required, it does not address the resource allocation and scheduling aspects of the problem in any detail. Loosely related work exists in the area of buffer management algorithms [Effe84, Chou85, Sacc86, Corn89, Ng91, Falo91] and complex query scheduling strategies [Murp89, Schn90, Murp91, Wils91, Chen92]. The algorithms proposed in both of these areas are less than ideal, however, when applied to workloads such as the one described in the introduction, as we will explain in the remainder of this section. During our discussion of related work, we will use the term "query" in a generic sense, to mean any work request arriving at the DBMS, regardless of whether it is a short, index lookup transaction or a long-running complex query.

In the area of buffer management, two early algorithms proposed as alternatives to basic global LRU schemes were DBMIN [Chou85] and Hot Set [Sacc86]. Both of these algorithms require the query optimizer to determine a specific memory allocation for each query, allowing a new query to begin execution only when sufficient memory is available. Queries in the Hot Set algorithm use the LRU replacement policy locally, and a query's buffer allocation is determined by identifying one or more "hot points" below which the query will incur too many page faults. DBMIN is more flexible, determining an appropriate buffer allocation for each file instance that a query accesses. These units of allocation for each file instance are called "locality sets." The page replacement policy used within each DBMIN locality set depends upon the query's page reference pattern within the associated file instance. In the event that memory is scarce when a new query arrives, a load control component of Hot Set or DBMIN will queue the query (and subsequent queries) until sufficient memory becomes available.

One shortcoming of the DBMIN and Hot Set algorithms is their overly conservative allocation of buffers in the case where multiple queries are simultaneously accessing the same relations (i.e. when a high degree of concurrent data sharing is present in the workload). In this case, queries will be forced to queue for memory unnecessarily because their buffer allocations are too high; as a result, buffer utilization will decrease and performance will be suboptimal. The MG-x-y family of was algorithms proposed in [Ng91] to provide better buffer utilization than DBMIN, especially in cases with concurrent data sharing, by allocating free buffers to the query that is able to get the most "marginal benefit" from them (where marginal benefit is defined as a decrease in the number of buffer misses). This work was extended in [Falo91], where two performance-predictive algorithms were described for deciding, at runtime, how many buffers to allocate to a query's locality sets. One algorithm, Predictor-TP, is based on optimizing predicted throughput, and the other, Predictor-EDU, is based on predicted disk utilization. Both algorithms involve the solution of a simple queueing network model, and both were shown to equal the performance of the MG-x-y algorithms for static workloads and to outperform them for more dynamic workloads. A loosely related approach, based on a static, global analysis of a relational DBMS workload containing a known query mix, was proposed by Cornell and Yu and was shown to outperform the Hot Set algorithm [Corn89]. The Cornell/Yu algorithm integrates query optimization and buffer management by iterating between an optimizer whose objective function takes buffer allocation into account and a queueing network model that predicts disk response times based on a specific buffer allocation.

With the exception of Cornell/Yu and Predictor-TP/EDU, each of these algorithms determines a buffer allocation by looking at individual queries in isolation, ignoring the effect that any particular buffer allocation has on other queries and on disk response times. As we will show in Section 4, in many cases a degradation in disk response times or in the performance of other queries can more than offset any gains expected from increasing an individual query's memory allocation.

All of the preceding buffer allocation algorithms suffer from another common shortcoming as well. Specifically, they all take the view that the buffer pool should be divided up among the set of queries that are resident in the system *at any particular instant*, possibly requiring some of the queries to wait until enough memory is available. As a result, none of the algorithms consider the potential sharing of buffered data between queries that execute *at different points in time*. We will refer to this type of sharing as *inter-transaction sharing*. Note that inter-transaction sharing is quite different from concurrent data sharing, as the latter refers to situations where several active queries simultaneously reference the same data and therefore requires a multiprogramming level (MPL) greater than one. In contrast, inter-transaction sharing can occur with an MPL greater than or *equal to* one. As a result, extending traditional memory management schemes to handle concurrent data sharing will not, in general, provide the potential performance gains achievable when inter-transaction sharing is considered. As we will show in Section 4, such a narrow view of the buffer allocation problem can have serious negative performance consequences.

A different approach to memory management was used by the Bubba parallel DBMS prototype [Cope88, Bora90]. A portion of Bubba's memory is dedicated to whole file caching, and files are placed in this cache in decreasing order of *temperature*. File temperature is defined as the ratio of a file's access rate (in block references per unit time) to its size (in blocks). Because Bubba's memory management scheme uses a metric that is *time averaged*, it can exploit inter-transaction sharing, unlike the previous algorithms discussed above. The temperature metric is not a panacea, however, since it only addresses the allocation of memory that is used to cache whole files. For example, it does not consider space for hash join tables, sort work areas, or other uses of memory that lead to reductions in disk I/O or processor cycles. In addition, the Bubba scheme was primarily intended for environments with a static, well known workload (albeit a mixed one). Two other issues would arise if the Bubba scheme were to be used to decide on an appropriate memory allocation for each query entering the system, especially in cases of more dynamic, or ad hoc query dominated workloads. One issue is the frequency and granularity with which temperature statistics should be maintained and utilized, and another issue is how much memory should be devoted to file caching versus other uses. Copeland et al [Cope88] suggests that the proper amount can be chosen through a "5 minute rule" type of analysis [Gray87]. Although such an approach is appropriate for configuring a DBMS, it is not applicable for a run-time algorithm that must decide (based on the current system state) on an appropriate memory allocation for each query.

In the area of query scheduling, the vast majority of the literature concentrates on execution strategies for individual multi-join queries [Murp89, Schn90, Murp91, Wils91, Chen92]. None of this work addresses cases where another workload class competes with the multi-join queries for system resources, and as a result, the tradeoffs involved with allowing more than one of these queries to execute simultaneously with other workload components are as yet unexplored. With respect to scheduling an individual multi-join query, the relevant issues include the query tree shape (left-deep, right-deep, or bushy), operator scheduling (granularity/degrees of parallelism), and load balancing. Most complex query scheduling algorithms concentrate on minimizing response times by maximizing the utilizations of system resources (processors, memory, and disks), and they have typically assumed that the entire system is available for allocation.[1]. The problem with applying algorithms that attempt to minimize single query response times to a multi-user, multiclass environment is that different execution strategies have different resource requirements, and without the notion of *competition* for resources, the potential adverse affects of one query's additional resource requirements on other concurrently executing queries (belonging to the same or a different workload class) cannot be taken into account.

3. DATABASE SYSTEM MODEL

Since our eventual goal is to investigate resource allocation and scheduling for parallel database systems, we have implemented a simulator for a parallel database machine. The simulator is based upon an earlier, event-driven simulation model of the Gamma parallel database machine [DeWi90]. The earlier model was a useful starting point since it

[1] Actually, some algorithms allow the amount of memory and number of processors available for allocation to be specified as inputs, but any allocation smaller than the entire configuration is simply treated as a "smaller system" that is allocated in its entirety as well. Thus, there is still no notion of any *competition* for resources outside of the operators within the single query.

had been validated against the actual Gamma implementation; it had also been used extensively in previous work on parallel database machines [Ghan90, Schn90, Hsia91]. The new simulator, which is much more modular, is written in the CSIM/C++ process-oriented simulation language [Schw90]. The simulator accurately captures the algorithms and techniques used in Gamma, although it is primarily used in the special case of a single node, (i.e. centralized) system for the work reported here. The remainder of this section provides a more detailed description of the relevant portions of the current simulation model, and concludes with a table of the simulation parameter settings used for this study.

3.1. Terminals

The simulated terminals model the external workload source for the system. Each terminal submits a stream of queries of a particular class, one after another. As each query is formulated, the terminal sends it to the Scheduler process for execution (described in the next paragraph) and then waits for a response before continuing on to the next query. In between submissions, each terminal "thinks" (i.e. waits) for some random (exponentially distributed) amount of simulated time.

3.2. Scheduler

The Scheduler is a special process that accepts queries from terminals and decomposes each query into several communicating lightweight processes, one for each operator in the query plan (e.g., a two-way hash join consists of a pair of selects, a build, and a probe process). Communication channels between operator processes are set up by the Scheduler before the query is executed, and operators pass data to each other in units of 8 KByte messages (each of which incurs a simulated memory-to-memory copy operation). Another important function of the scheduler is to implement a load-control policy for queries based upon their memory requirements. As each query arrives, the scheduler calculates the query's memory requirements and initiates its execution only if sufficient memory is available. Otherwise, the scheduler forces the query to wait in a memory queue that is managed in a first-come, first-served manner.

3.3. Operators

All the queries in the work reported here use only two basic relational operators: select and join. Any result tuples from the queries are "sent back to the terminals", an operation that consumes some small amount of processor cycles for network protocol overheads. The join algorithm used in the simulator is the centralized version of the hybrid hash join algorithm [DeWi84, Schn89].

The centralized hybrid hash join algorithm [DeWi84] operates in three phases. In the first phase, the algorithm uses a hash function to divide the inner (smaller) relation, R, into N partitions. The tuples of the first partition are used to build an in-memory hash table, while tuples that fall in the remaining N-1 partitions are written to temporary files. A good hash function produces just enough remaining partitions to ensure that each partition of tuples will be small enough to fit entirely in main memory. During the second phase, the outer (larger) relation, S, is partitioned using the hash function from step 1. The last N-1 partitions are again stored in temporary files, while tuples that fall in the first partition of S are used to immediately probe the in-memory hash table of R's first partition, which was created during the first phase. Lastly, during the third phase, the algorithm joins each of the remaining N-1 partitions of relation R with the corresponding partition of relation S. If enough memory is available to contain a hash table for the entire relation R, then the third phase is not necessary and no temporary file I/O is required; in this case we say that the join is a *single partition* join, and each relation is only read once. If this is not the case, then temporary files are needed (along with additional I/Os to write and reread them), and the join is called a *multi-partition* join.

3.4. Database

The database itself is modeled as a set of relations, each of which can can have one or more associated B+ tree indices. These indices can be either clustered or unclustered. In the case of multiple disks, all relations (and their associated indices) are declustered [Ries78, Livn87] (horizontally partitioned) across all the disks in the configuration. A hashed partitioning strategy is used, where a randomizing function is applied to the key attribute of each tuple to select a particular disk drive. As is the case with Gamma [DeWi90], "point" queries of the form *"X.y = constant"*, can be routed directly to the particular disk where the selected tuple resides if *y* is the partitioning attribute of relation *X*.

3.5. Disks

The simulated disks model a Fujitsu Model M2266 (1 GB, 5.25") disk drive. This disk provides a 256 KB cache that we divide into eight 32 KB cache contexts for use in prefetching pages for sequential scans. In our model of the disk, which slightly simplifies the actual operation of the disk, the cache is managed in the following manner: Each I/O request, along with the required page number, specifies whether or not prefetching is desired. If so, one context's worth of disk blocks (5 blocks) are read into a cache context after the originally requested data page has been transferred from the disk to memory. Subsequent requests to one of the prefetched blocks can then be satisfied without incurring an I/O operation. A simple round-robin replacement policy is used to allocate cache contexts if the number of concurrent prefetch requests exceeds the number of available cache contexts. The disk queue is managed using an elevator algorithm.

3.6. CPU and Memory Management

The CPU is scheduled using a round-robin policy. The buffer pool models a set of main memory page frames. Page replacement in the buffer pool is controlled via the LRU policy extended with "love/hate" hints (like those used in the Starburst buffer manager [Haas90]). These hints are provided by the various relational operators when fixed pages are unpinned. For example, "love" hints are given by the index scan operator to keep index pages in memory; "hate" hints are used by the sequential scan operator to prevent buffer pool flooding. In addition, a memory reservation system under the control of the Scheduler process allows memory to be reserved in the buffer pool for a particular operator. This memory reservation mechanism is used by joins to ensure that enough memory is available to prevent their hash table frames from being stolen by other operators running on the same node.

3.7. Simulation Parameter Settings

The important parameters of the simulated DBMS are listed in Table 1. The CPU speed and memory configuration were chosen to reflect the characteristics of the current generation of commercially available multiprocessors (e.g., the Thinking Machines CM-5 or the Intel Paragon). The software parameters are based on instruction counts taken from the Gamma prototype when the previous simulator was validated. The disk characteristics approximate those of the Fujitsu Model M2266 disk drive, as described earlier.

Configuration/Node Parameter	**Value**	**CPU Cost Parameter**	**No. Instructions**
Transaction Terminals	100 (varied)	*Initiate Select*	20000
Query Terminals	1 (varied)	*Initiate Join*	40000
Transaction Think Time	exponential, mean 5 sec	*Initiate Store*	10000
Tuple Size	200 bytes	*Terminate Store*	5000
Number of Disks	1-4 disks(varied)	*Terminate Join*	10000
CPU Speed	20 MIPS	*Terminate Select*	5000
Memory Size	32 MB	*Read Tuple*	300
Page Size	8 KB	*Write Tuple into Output Buffer*	100
Disk Seek Factor	0.617	*Probe Hash Table*	200
Disk Rotation Time	16.667 msec	*Insert Tuple in Hash Table*	100
Disk Settle Time	2.0 msec	*Hash Tuple using Split Table*	500
Disk Transfer Rate	3.09 MB/sec	*Apply a Predicate*	100
Disk Cache Context Size	4 pages	*Test an Index Entry*	50
Disk Cache Size	8 contexts	*Copy 8K Message to Memory*	10000
Disk Cylinder Size	83 pages	*Start an I/O*	1000

Table 1: Simulator Parameters.

4. MULTICLASS WORKLOADS: EXPERIMENTS AND RESULTS

In this section we discuss three main groups of experiments related to multiclass workload resource allocation and scheduling. The first group studies inter-class memory allocation tradeoffs, and consists of four experiments that explore the two extreme ends of the inter-class memory allocation spectrum, allocating both the absolute minimum and maximum to each of two workload classes, as well as an intermediate scheme. We vary the number of disks, the transaction load, and the number of indices used. The second group of experiments looks at the issue of serial versus concurrent scheduling of multiple long-running join queries in the presence of a workload consisting of short, index lookup transactions. We repeat the experiment for both single disk and multiple disk cases. Our final set of experiments compares two strategies for scheduling complex, multi-join queries concurrently with a workload of short transactions. We again compare the two extreme ends of the resource requirement spectrum, which in this context are the left-deep and right-deep query execution strategies [Schn90]. We close the section with a very brief look at how the behaviors we observed for centralized systems match up with those exhibited by parallel systems.

4.1. Scope and Methodology

Before diving into the presentation of our results, we need to explain a few things about our objectives and the resulting experimental approach. In order to clearly isolate the key performance tradeoffs, we use a very simple, two class workload that has been designed to capture the essence of what real workloads do. For example, to capture the behavior of OLTP-style transactions, we use single tuple, nonclustered index selects. Although real OLTP transactions (e.g., debit/credit) tend to do several such lookups, as well as updates, it is mainly their data access pattern that is important for this study. Moreover, since we do not wish to address concurrency control issues in this paper, our "transactions" do not perform updates. To model the behavior of long-running queries, we focus mainly on binary joins that execute using the hybrid hash join algorithm. Hash joins allow us to easily vary the two key factors that determine the performance of long-running queries: memory allocation and disk resource consumption. Most join algorithms have a characteristic inverse relationship between memory allocated and the number of I/Os required, but for hash joins, the graph of this relationship has a smooth linear shape. Finally, because we want to limit the effects of different query optimization strategies as much as possible for this study, both the inner and outer join relations are always the same size in the experiments reported here.

For the remainder of this paper, we will use the term *transaction* to refer to the workload class that models OLTP-style transactions (as opposed to its general sense of any recoverable unit of work), and the term *query* will refer to the long-running query class (versus its general use in referring to any query language statement submitted to a DBMS). We will occasionally use the term *work request* to indicate any arbitrary work request arriving at the DBMS, be it "query-like" or "transaction-like".

The performance metrics that we use to evaluate resource allocation and scheduling alternatives are the average response times for both the queries and transactions. In nearly all cases, we study response times as a function of the join relation sizes (2-92 megabytes) relative to a fixed amount of available memory (32 megabytes). Thus, the X-axes of our graphs indicate the size of the join relations and can be interpreted as a measure of the long-running query workload's resource requirements. We vary the ratio of join relation size to available memory from .06 (where the inner relation fits in one 16th of the memory) up to about 3.0 (where the inner relation is three times as big as available memory).

Table 1, discussed in Section 3, summarizes the system configuration parameter values used in the experiments reported here. The database and workload characteristics chosen for the experiments reported in this section will be explained as the individual experiments are described.

4.2. Transaction/Query Memory Allocation Tradeoffs

We will begin with a series of four experiments, each of which sheds some light on the issue of how to allocate the memory of a DBMS between a primary workload of short transactions and a secondary workload of large two-way join queries. The workload parameters for these experiments are summarized in Table 2. Transactions each access a fairly large (100MB) relation that can be thought of as roughly corresponding to the Account file in the TPC/A and TPC/B benchmarks [Gray91]. Each query computes the join of two other randomly chosen relations, each the same size. Since

Parameter	**Experiment 1(a)**	**Experiment 1(b)**	**Experiment 1(c)**	**Experiment 1(d)**
# transaction terminals	100	100	5 - 150	100
# query terminals	1	1	1	1
mean transaction think time	5 sec	1.25 sec	5 sec	5 sec
# disks	1	4	1	1
query class relations	2 MB - 92 MB	2 MB - 92 MB	2 MB - 92 MB	2 MB - 92 MB
transaction class relation	100 MB	100 MB	100 MB	100 MB
transaction class indices	one, 4 MB	one, 4 MB	one, 4 MB	three, 4 MB each

Table 2: Workload Descriptions for Experiment Group 1.

the hybrid hash join algorithm can run with a memory allocation ranging from the size of the entire inner relation all the way down to that of its square root, a central question is how much memory the join query should be given in order to do its work. We will examine the impact of three different memory allocations here:

Greedy: When a new query arrives, it is given all but a very small fraction of the available buffer space, with just enough memory being left over to enable the concurrent transactions to run (instead of blocking behind the query in the memory wait queue). This is basically what the DBMIN, Hot Set, Cornell/Yu, MG-x-y, and Predictive Load Control relational buffer allocation strategies would do for this workload [Chou85, Sacc86, Corn89, Ng91, Falo91], since each would view the memory requirements of any *individual* transaction as minimal.

Conservative: A query is only allocated the minimum amount of memory that it needs to execute (slightly more than the square root of the inner relation size, in pages). This is what several commercial database systems do in cases where the available memory is insufficient to hold the entire inner relation. This policy attempts to maximize transaction performance by keeping the query out of the way as much as possible (i.e., by allocating relatively few resources to it).

Responsible: The transaction class as a whole is allocated enough memory to keep the Account relation index in memory at all times. When a new query arrives, it is given all of the remaining memory to use for the join. This policy attempts to incorporate the best aspects of the other two policies by having the queries behave responsibly with respect to the inter-transaction sharing behavior of the transaction class, as will be seen shortly.

To examine memory allocation policy tradeoffs, our first experiment starts by considering a case where the database is stored on a single disk, transactions originate from 100 terminals with 5 second mean think times (an average of one transaction in the system at any particular moment), and a steady stream of join queries enters the system (one after another, with no think time between arrivals, not unlike a batch job stream). The relevant workload parameters are listed under Experiment 1(a) in Table 2. Figures 1 and 2 show the average transaction and query response times, respectively, as a function of the size of the two relations being joined by the query. To help explain these results, Figures 3 and 4 show the associated numbers of disk reads per transaction and per query, respectively. Figures 5 and 6 show the disk request response times observed by the two classes, including any queueing time, since the disk is the bottleneck resource here.

Since the transaction portion of a database workload is typically the "bread and butter" for a DBMS, i.e., its most important job class, we will start by studying the transaction response time results (Figure 1). When the join relation

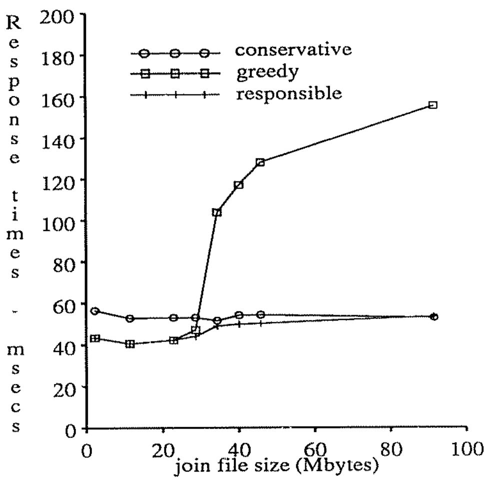

Figure 1: transaction response times.

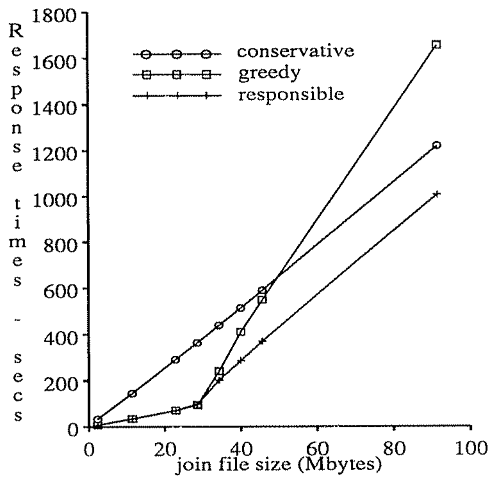

Figure 2: query response times.

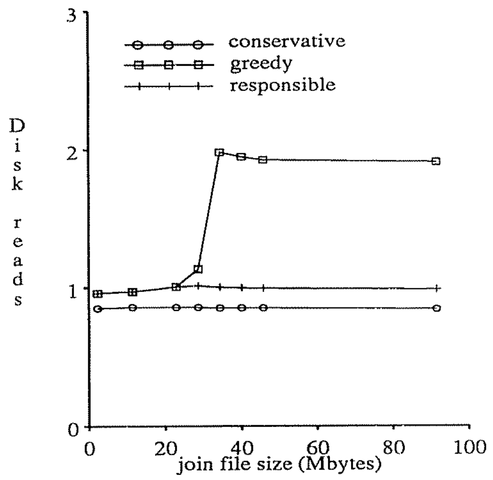

Figure 3: transaction disk reads.

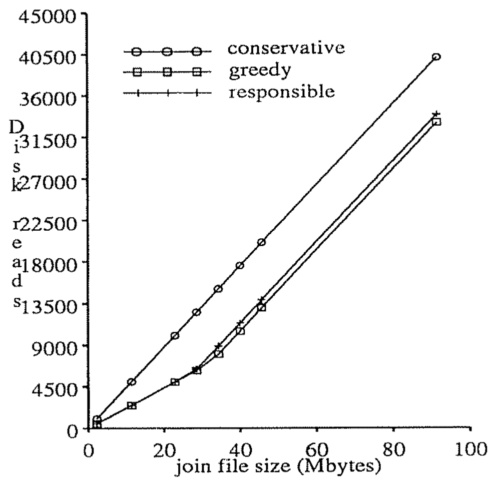

Figure 4: query disk reads.

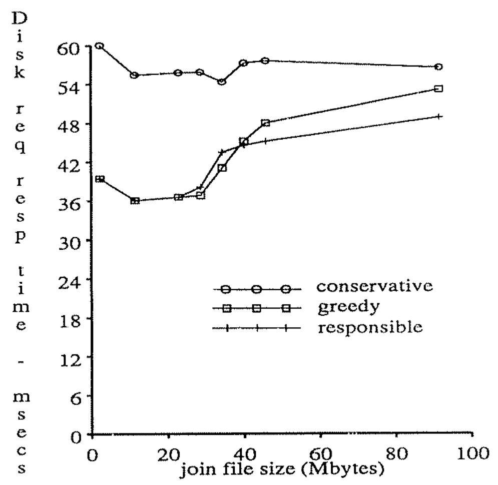

Figure 5: transaction disk resp times.

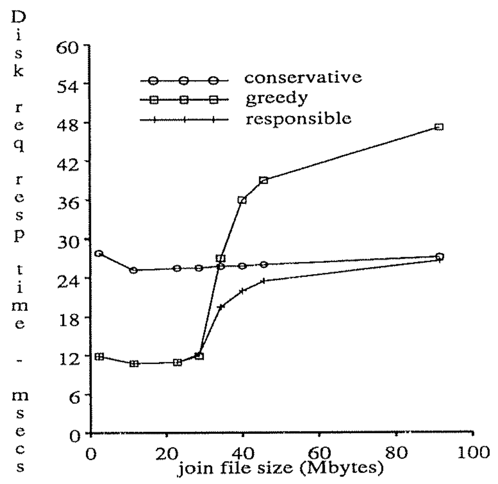

Figure 6: query disk resp times.

1 join, 1 transaction class, 1 disk, 100 transaction terminals

sizes are small, we see that the Greedy and Responsible policies for allocating memory to the query both actually give better average transaction response times than the Conservative policy. This is because the Conservative policy initially requires the query to do significantly more I/O (Figure 4), thereby increasing the degree to which the transactions end up waiting at the disk (Figure 5). However, for join relation sizes beyond about 28 MB, transaction performance degrades rapidly under the Greedy policy. This is because the Account relation index is approximately 4 MB in size, and Greedy queries cause it to be paged out at this point, as is evident from the fact that the number of disk reads per transaction jumps from about 1 (for reading one Account data page) to 2 (for also reading an Account index leaf page) at this point under Greedy (Figure 3). In contrast, the Responsible policy essentially tracks the performance of Greedy until the point where Greedy degrades; it then tracks the performance of Conservative (which is essentially constant) from that point onward.[2] Thus, the Responsible policy, where queries avoid causing the Account index to be paged out, but use all of the remaining memory, provides the best overall performance for the transactions here.

Let us now consider the corresponding query response time results (Figure 2). With small join relation sizes, we see that the Conservative policy leads to significantly worse performance for the queries in the workload. This is to be expected since it implies nearly three times as much join I/O when the other two policies are able to entirely avoid writing and re-reading the relations being joined. However, the Greedy policy's query performance quickly worsens starting at the point where its transaction performance falls apart. This is because, by paging out the Account index, and thereby worsening transaction performance, the Greedy policy also leads to much more disk competition for the query. This is evident from the average disk request response time for the queries (Figure 6), which rapidly worsens at that point. In fact, as the relation size is increased further, the Greedy policy — which is intended to favor the queries over the transactions — becomes even *worse* than the Conservative policy in terms of query response times as a result of this effect. Queries also see a disk response time increase under the Responsible policy at the point where their joins begin to require multiple partitions (Figure 6). This increase occurs because the I/O requests for writing out partitions to temporary files are now competing at the disk with the join's initial relation read I/Os; however, this effect is much less pronounced for Responsible than for Greedy due to the relatively constant transaction I/O behavior under Responsible. As

[2] Note that this is in spite of the fact that the average number of transaction disk I/Os is somewhat higher under the Responsible policy than under the Conservative policy (Figure 3). The Conservative policy has a better buffer hit rate due to Account data page hits, whereas the Responsible policy does not retain Account data pages in memory for the larger join relation sizes. However, this better hit rate for Conservative is countered by the somewhat better disk response time (Figure 5) that Responsible gives the transactions by requiring less join I/O than Conservative. (The overall response time for transactions is essentially determined by the product of the average number of transaction I/Os and the transactions' disk response time.)

a result, the Responsible policy turns out to produce the best performance for the queries here, as it did for the transactions, since it does not page out the Account index.

When we first saw these results, we were quite surprised. Our initial expectations were that Conservative would be best for the transactions, and that Greedy would be best for the queries. (In retrospect, of course, the superiority of the Responsible policy makes good sense in light of the above analysis.) To verify that the effect was indeed real, and not just an artifact of a one disk DBMS, we also ran experiments using a similar workload but with two-disk and four-disk configurations. In each case, all of the relations in the database were declustered over all of the available disks.[3] When scaling up the I/O subsystem in this way, we also scaled up the transaction intensity — by decreasing mean terminal think times from 5 seconds to 2.5 seconds and 1.25 seconds, respectively — since the decision to use a larger configuration would be most likely be driven by increased transaction processing demands in the context of a real system. These workload changes are summarized as Experiment 1(b) in Table 2.

Figures 7 and 8 show the transaction and query response times for the three alternative memory allocation policies in the four-disk case with 1.25 second transaction think times. The results are qualitatively very similar to the one-disk case, with Responsible being the memory allocation policy of choice. The only significant difference between the four-

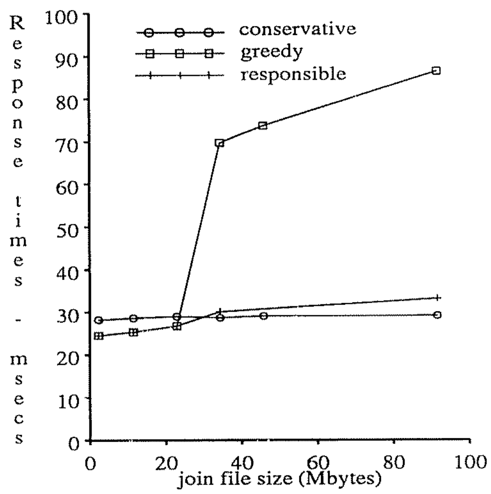

Figure 7: transaction response times.

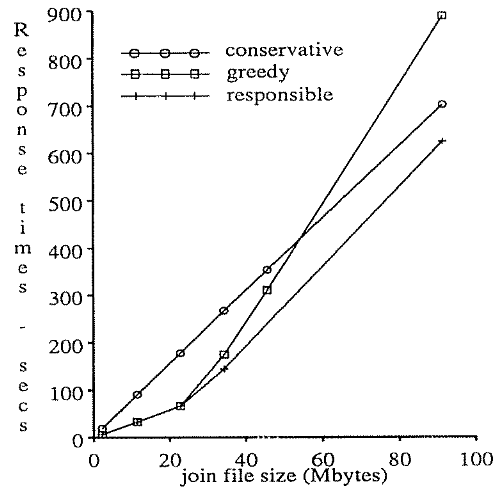

Figure 8: query response times.

1 join, 1 transaction class, 4 disks, 100 transaction terminals

[3] During join processing, the join then reads the data from each disk in sequence, reading all data from one before moving to the next one. This is how a number of current commercial database systems (e.g., the Teradata and DEC Rdb/VMS systems) handle access to partitioned relations in the single-node/multiple-disk case.

disk case (Figures 7-8) and the one-disk case (Figures 1-2) is that the Conservative allocation policy does a bit better in terms of transaction response times for large join file sizes. (As before, however, Conservative is worse in terms of the query response times.) The reason for this difference is that, with four disks, the queries generate significantly less disk contention for the transactions here than in the one-disk case. As a result, the Conservative policy benefits somewhat from allowing transactions to run with slightly fewer disk reads (as was seen in Figure 3).[4] Due to space limitations, we do not show the results from the two-disk case, as they showed exactly the same relative performance results as the four-disk case.

The next experiment in our initial series of four looks at how the previous results respond to changes in the intensity of the transaction workload. As indicated in Table 2 under Experiment 1(c), we do this by varying the number of transaction-submitting terminals in the one-disk case with the size of the join relations fixed at 45 MB. The resulting transaction and query response times are shown in Figures 9 and 10. (Note that the results for 100 terminals correspond to the 45 MB join size points in Figures 1 and 2.)

As before, let us begin by examining the transaction response time results (Figure 9). As shown, the relative ordering of the three memory allocation policies is the same as before, but the degree to which the Greedy policy is worse

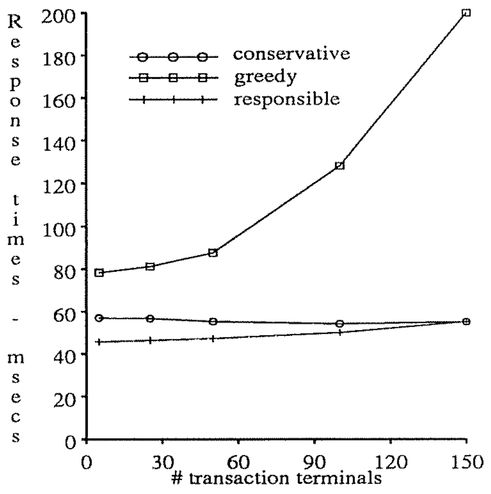

Figure 9: transaction response times.

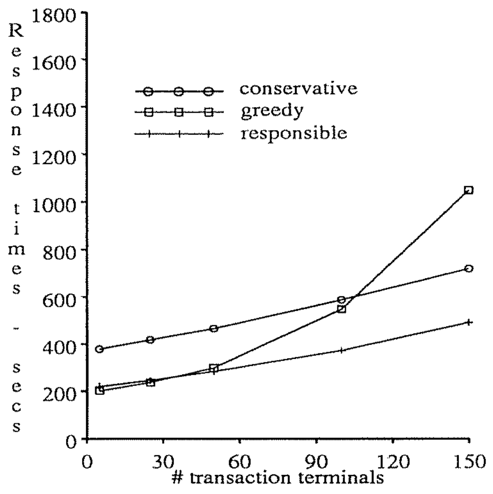

Figure 10: query response times.

1 join, 1 transaction class, 1 disk, varying transaction terminals

[4] It should be noted that this particular benefit is directly related to the memory and Account relation sizes chosen here, as transactions benefit from a 20-25% hit rate for Account relation data pages under the Conservative policy. Had the Account relation been substantially larger relative to the buffer pool size, which seems likely in real applications, this benefit would disappear and Responsible would perform at least as well as Conservative over the entire range of join sizes.

than the others is less under lighter transaction loads; it is always significantly worse, though, as transactions do two I/Os under Greedy versus one under Responsible and slightly less than one under Conservative. The lessening of the difference as the transaction load goes down is due to reduced disk contention as the system moves toward a region where multiple transactions are rarely co-resident in the system; this is also why the curves become flat at the lightest loads.

Turning to the query response times (Figure 10), we see that Greedy is by far the worst memory allocation scheme at the higher transaction loads; this is due to its impact on transaction performance and therefore on the disk response times observed by the queries, as discussed earlier. As the transaction load is reduced, however, it is able to outperform the Conservative strategy; at the lightest loads it becomes a bit better than the Responsible allocation strategy. The explanation for this trend is straightforward: As the transaction load becomes very light, the queries experience little or no disk contention due to transactions — no matter how badly the transactions perform due to the query memory allocation policy. Thus, under very light transaction loads, the query response times are determined only by the number of join I/Os required under each policy. Greedy thus becomes slightly better than Responsible, and Conservative is by far the worst policy, as would be expected.

In the experiments described thus far, the transactions have all shared a single common index. In the last of our first set of experiments, we repeat the initial one-disk experiment but we have transactions access the Account relation via any of three unclustered indices (summarized as Experiment 1(d) in Table 2). The transaction and query response times for this experiment are shown in Figures 11 and 12. Briefly, the effects seen here agree well with the intuition provided by the preceding experiments. For transactions, Responsible is the memory allocation policy of choice; here, this policy permits all three of the Account indices to remain memory-resident. The Greedy policy provides very poor transaction performance once the join relations become large enough to force the transactions to do index I/Os. In this case, the transactions end up doing an average of 2.5 I/Os (1 data page I/O plus 1.5 index page I/Os) as a result of the Greedy policy. For the queries, Greedy is still eventually the worst policy, as before, but there is now a region in which it provides somewhat better query response times than Responsible. With three indices instead of one, the Greedy policy is able to use 12 MB more memory for each query than the Responsible policy, producing a larger difference in the number of additional I/Os required for Responsible queries. This gives Greedy a performance advantage for the queries until the point where its use of memory causes their disk response times to degrade due to multiple transaction I/Os.

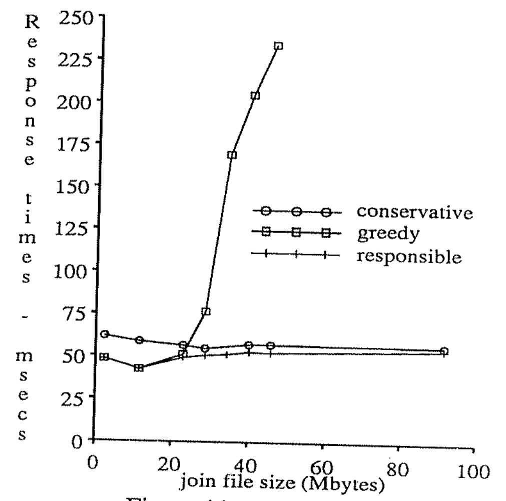

Figure 11: transaction response times.

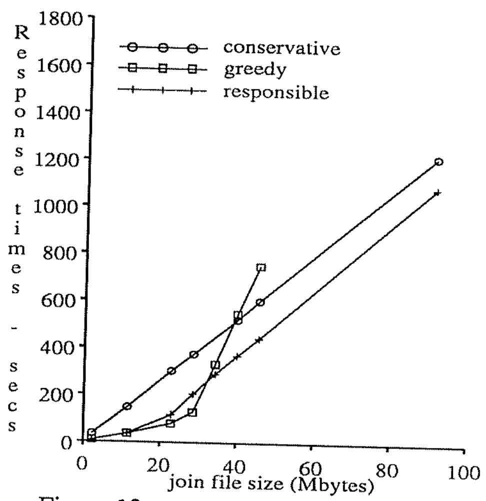

Figure 12: query response times.

1 join, 3 transaction classes (multiple indices), 1 disk, 100 transaction terminals

The preceding experiments clearly show that, for our representative mixed workload, significant performance gains are available — both for the transactions *and* the query — by maximizing the amount of memory given to the query subject to the constraint that the index(es) be kept memory-resident. As mentioned in Section 2, these gains are due to the inter-transaction sharing that occurs for the transaction class, a type of data sharing that is not addressed by most previously proposed relational database memory allocation schemes. In those algorithms, if the set of active work requests can consume all of memory with the pages that they are accessing, then all of memory will indeed be allocated to them. Since each transaction in our workload accesses only a tiny fraction of (i.e., one path through) the Account index, these schemes would cause it to be largely disk-resident.

Our experiments also demonstrate that any comprehensive memory allocation scheme cannot base its decisions solely on the characteristics of individual work requests in isolation, but must also account for potential adverse effects on other work requests (either currently active in the system, or expected to arrive in the future) and the impact of memory allocation on disk response times. We have also seen that the actual quantitative impacts of these two factors are a function of the ratio of join relation sizes to available memory, the available disk bandwidth, and the intensity of the transaction workload. In summary, good memory allocation in a mixed workload environment requires a fairly detailed global knowledge of the workload as well as the ability to gauge the impact of any memory allocation on disk utilizations and response times.

4.3. Scheduling Multiple Queries

Our second series of experiments looks at the issue of how to manage *multiple* streams of binary join queries in the presence of transactions. The workload parameters for this set of experiments are summarized in Table 3, which shows an increase in the number of terminals submitting join queries from one to four. Two options will be considered: Serial, where only one query is permitted to run at a time (together with the transactions), and Concurrent, where all of the query streams are permitted to execute simultaneously. We consider only the Responsible memory allocation strategy, given our previous results, dividing the approximately 28 MB of non-index memory up evenly between the four queries in the case of Concurrent query scheduling. As shown in Table 3, we ran both one-disk and four-disk versions of this experiment; the results are shown in Figures 13-16.

Let us first examine the one-disk results (Figures 13 and 14). As should be expected, Serial query scheduling is significantly better in terms of transaction response times in the one-disk case, as Concurrent scheduling leads to much more disk activity for the transactions to contend with. In terms of query response times, Serial scheduling is again the better approach over the entire range of join relation sizes. This is strictly due to disk I/O considerations. A graph of the number of disk I/Os per query (not shown) would have the same shape as Figure 14; since the amount of memory available for the queries is divided four ways in the Concurrent case, their joins become multi-partition joins much sooner and do more I/O as a result.

Let us now turn to the four-disk case (Figures 15 and 16). Serial scheduling is again the better approach as far as the transactions are concerned, and the reason is the same as before. However, the tradeoff for the queries is not as clear in this case. The Concurrent strategy is better for small or large queries, but medium-size queries are better done via Serial scheduling. In the regions where concurrent queries dominate, there is much more disk bandwidth available here compared to the one-disk case. Since a given join utilizes only one disk at a time, the Concurrent strategy benefits by

Parameter	**Experiment 2(a)**	**Experiment 2(b)**
# transaction terminals	100	100
# query terminals	4	4
mean transaction think time	5 sec	1.25 sec
# disks	1	4
query class relations	2 MB - 92 MB	2 MB - 92 MB
transaction class relation	100 MB	100 MB
transaction class indices	one, 4 MB	one, 4 MB

Table 3: Workload Descriptions for Experiment Group 2.

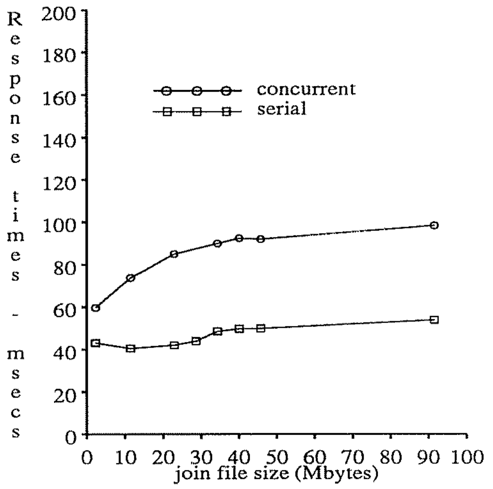

Figure 13: transaction response times.

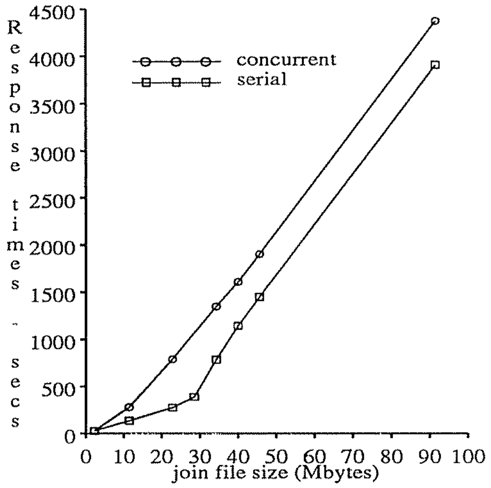

Figure 14: query response times.

Multiple joins (4), 1 transaction class, 1 disk, 100 transaction terminals

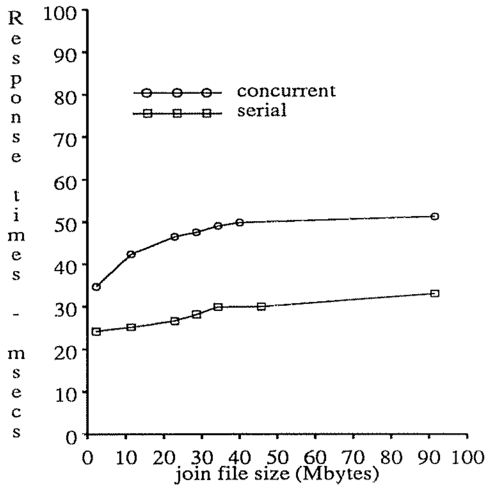

Figure 15: transaction response times.

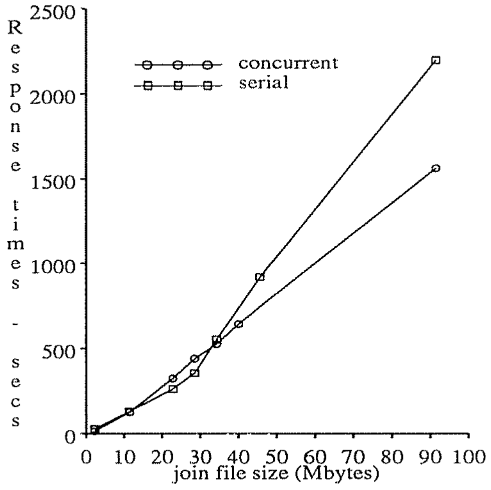

Figure 16: query response times.

Multiple joins (4), 1 transaction class, 4 disks, 100 transaction terminals

running the four queries together, leading to a much better utilization of the I/O subsystem by the queries. In the middle region in the graph, the temporary inferiority of the Concurrent strategy stems from its having to do more I/O, sooner, than the Serial strategy. In this region, the Serial strategy avoids writing and re-reading join partitions to temporary files whereas the Concurrent strategy cannot avoid doing so (due to having 1/4 the memory for each query). To the left of this region neither strategy does any temporary file I/O and for large queries, both do enough temporary file I/O that the relative difference in total I/O is more than counteracted by the improved disk utilization that Concurrent scheduling provides for the queries.

4.4. Scheduling Complex Queries

Our final set of experiments, whose workload descriptions are shown in Table 4, considers a different issue. These experiments explore the interaction between a more realistic query workload, containing complex queries, and transactions. For these experiments, the query workload consists of a single stream of five-way (i.e., five-relation) joins. As in the previous experiments, we use the Responsible memory allocation strategy for dividing memory between the queries and transactions.

Several execution strategies (left-deep, right deep, and bushy) have been proposed in the literature for executing multi-join queries [Schn90]. Each exploits intra-query concurrency in a different manner, so they have very different resource requirements. Since there are a large number of possible execution strategies, we decided to study the impact of the two extreme ends of the spectrum — left-deep and right-deep query plans. A left-deep plan is a pipelined execution strategy where the results of the previous join are used to build the hash table for the next join; this strategy offers the least intra-query concurrency (scanning one relation at a time) and requires fewer resources (as only two hash tables reside in memory simultaneously). In contrast, an N-way right-deep plan reads N-1 relations simultaneously into hash tables and then probes them all at once in a pipelined fashion. This is a highly concurrent strategy, but it is also much more resource-intensive.

The first variant of this experiment uses one disk; all of the join relations reside on this single disk. Figures 17 and 18 show the transaction and query response times for this experiment. The results clearly show the left-deep strategy to be better than the right-deep strategy for both the queries and the transactions. This is because the disk is a bottleneck in this case, making the left-deep approach better since it causes less disk contention. This is consistent with the previous experiment's one-disk results, where Serial query scheduling was found to dominate to Concurrent scheduling.

Parameter	**Experiment 3(a)**	**Experiment 3(b)**
# transaction terminals	100	100
# query terminals	1	1
mean transaction think time	5 sec	1 sec
# disks	1	5
query class relations	2 MB - 92 MB	2 MB - 92 MB
transaction class relation	100 MB	100 MB
transaction class indices	one, 4 MB	one, 4 MB

Table 4: Workload Descriptions for Experiment 3.

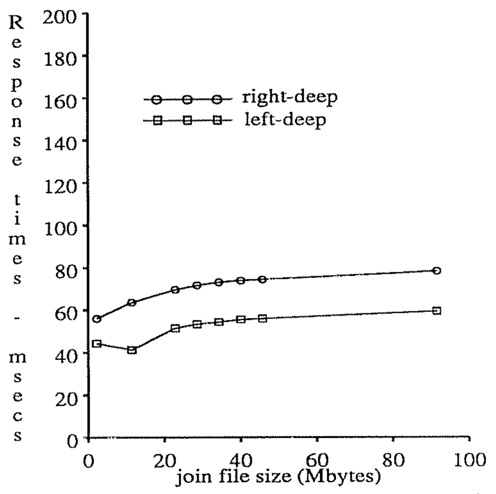

Figure 17: transaction response times.

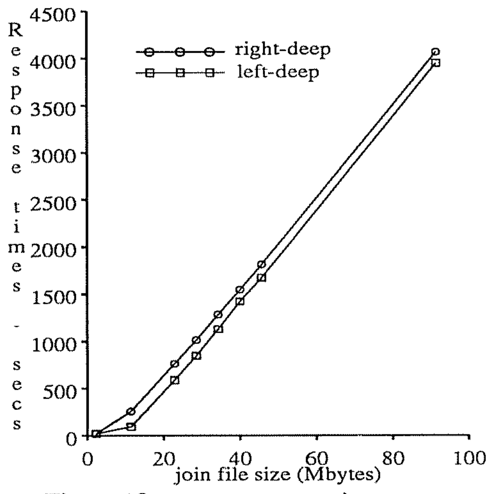

Figure 18: query response times.

Complex query (1), 1 transaction class, 1 disk, 100 transaction terminals

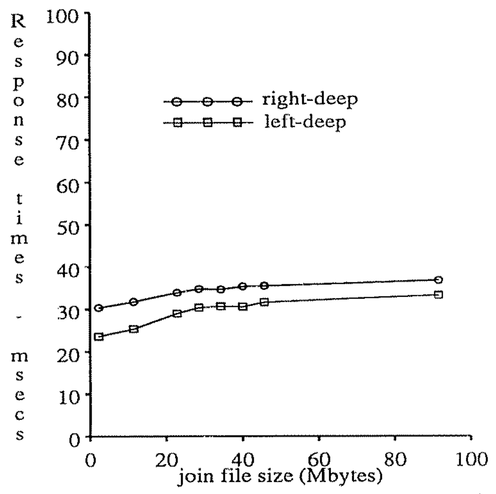

Figure 19: transaction response times.

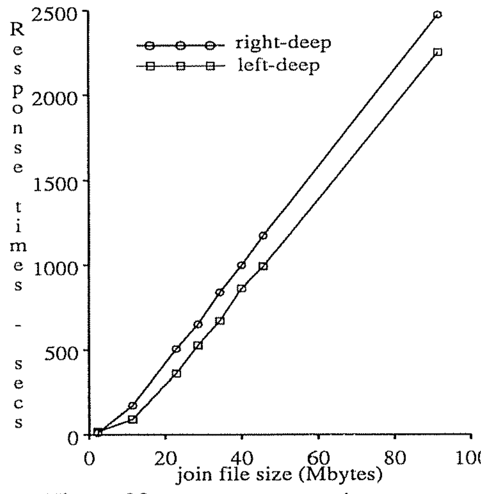

Figure 20: query response times.

Complex query (1), 1 transaction class, 5 disks, 100 transaction terminals

Since left- versus right-deep query plan performance is known to be sensitive to to the availability of disk bandwidth [Schn90], we also conducted a five-disk version of the same experiment. In this case, each of the join relations resides on a separate disk. We would expect the right-deep query plan to significantly benefit from this, as its four relation reads can now proceed concurrently and thus exploit the additional I/O bandwidth. In contrast, the left-deep plan can execute only two selects concurrently and will thus not benefit as much. Somewhat surprisingly, the results, which are shown in Figures 19 and 20, show that the left-deep execution strategy is again the strategy of choice in the presence of a transaction workload except when all of the relations fit in memory together (the 2MB case, barely visible); there, the right-deep query strategy is almost twice as fast as the left-deep strategy.

The reason that the right-deep strategy initially performs better for the queries is that right-deep plans can exploit nearly twice as much I/O bandwidth as left-deep plans (though the improvement factor is less than two due to the presence of transactions). However, as soon as right-deep plans must move to multiple partitions (which occurs for relation sizes of 10 MB and higher), they become worse than left-deep plans. This is because the memory for the right-deep plans is divided among four queries, as compared to two for left-deep, which implies less memory for each query and thus a higher number of hash join partitions in the right-deep case. As a result, right-deep query plans have to perform a substantially higher number of I/Os (e.g., about 18,000 more disk reads for a join relation size of 40 MB). This increase in I/O more than nullifies the gains due to higher I/O bandwidth.

4.5. Applicability to Parallel Database Systems

After completing our initial experiments on resource allocation and scheduling for a centralized (single processing node) DBMS, we were curious to verify our intuition that the same issues would have to be addressed for parallel database systems. Since our simulator has this capability already, we decided to re-run experiment 1(a) on an eight node system. In order to use the same workload that was used in experiment 1(a), and to retain the same scales on our graphs, we maintained the same amount of aggregate memory in the eight node system. Since the single node system had 32 megabytes, we configured each node of the eight node system with four megabytes. Each node has only one disk, and as was the case in our single node, 4-disk experiment (experiment 1(b) in Table 2), all relations are fully declustered across the eight disks and the transaction workload intensity is scaled up by a factor of eight as well. Because the transactions in our workload were simple "point" queries, each one is routed randomly to a single node (using a uniform distribution to model hash partitioning).

Figures 21 and 22 show the transaction and query response times for the eight node case. The results are qualitatively very similar to the single node results of Figures 1 and 2, with the quantitative differences caused primarily by the increased disk bandwidth available in the eight node system. Based on this experiment, we feel confident that the tradeoffs shown in our study for centralized systems are relevant to parallel systems as well. Parallel systems introduce additional tradeoffs, of course, such as processor allocation and assignment, additional load balancing considerations, and how partitioning can best be used to isolate different workload classes from each other, to name a few. These and other issues will be addressed in our future work.

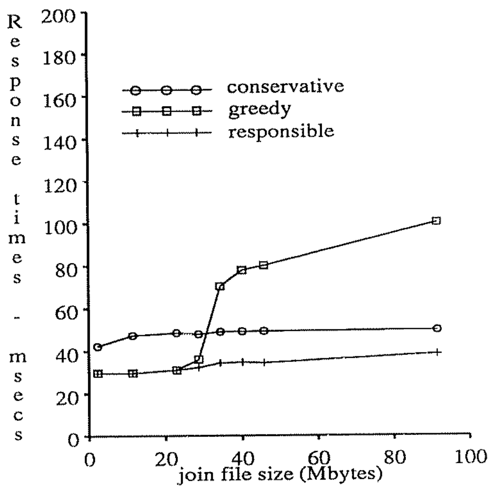

Figure 21: transaction response times.

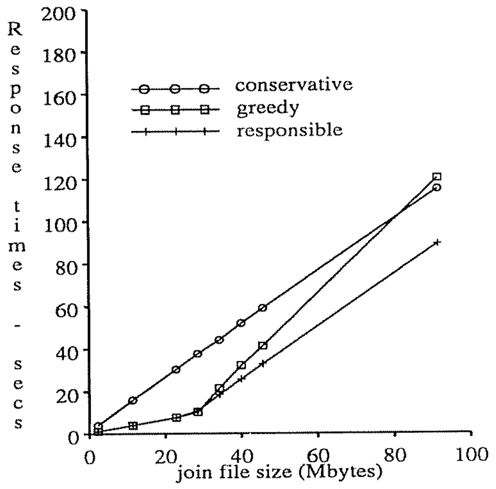

Figure 22: query response times.

1 join, 1 transaction class, 8 nodes, 1 disk each

5. TOWARDS MIXED WORKLOAD RESOURCE ALLOCATION

In considering the implications of our results, we realized that their usefulness would be limited if they could not be applied to identify good resource allocation and scheduling schemes for any given workload. However, because of the complexity of the factors identified in the previous section for just one simple workload, we were skeptical that a fixed set of heuristics would work well for any arbitrary, perhaps dynamically changing workload. On the other hand, if we could develop an analytical model that reproduced the results of the previous experiments, it could perhaps be useful as part of a run-time resource allocator. Our first cut at such a model is based on a simple queueing network that we solve using the Mean Value Analysis (MVA) technique [Reis80]. Our model is composed of three servers: an infinite capacity server that models the terminal think time, a fixed-rate (load-independent) CPU server scheduled using the processor sharing discipline, and a fixed-rate disk server scheduled in a first-come, first-served manner. The model has two closed job classes, transactions and queries.

Queueing network models are parameterized by specifying the mean service time at each server for each job class plus the mean "visit count" at each server for a typical job of each class. The necessary visit counts at the disk and CPU are calculated in our model using the same techniques that cost-based query optimizers would use. For example, the average number of visits to the disk and CPU by a hybrid hash join can be calculated using the equations presented in [Shap86]. The service time per CPU visit can be calculated from the number of instructions needed to perform each operation and the MIPS rating of the processor. The disk service time for the transactions is modeled as the cost of a single page I/O. The 80% disk cache hit rate enjoyed by the query class complicates matters somewhat for the queries; we took the simplistic approach of reducing their disk visit count by 80% and charging them for a five-page I/O during each visit (i.e. the cost of a complete cache context prefetch). To estimate the number of disk I/Os required per transaction based on the amount of memory available for the index, we used a very simple approach. We assumed that the index fills the available memory in a breadth-first fashion, from the root down to the leaves, until either the available memory or the index pages are exhausted. Each index level that is completely memory-resident reduces the expected I/O count by one, and each partially resident level reduces the I/O count by a fraction equal to the percentage of that level that is memory-resident.

Figures 23 and 24 show the model's predictions for the transaction and query response times of Figures 1 and 2. The results are not entirely accurate, but they are encouraging given the simplistic nature of the model. The qualitative query performance results are captured successfully, and they are reasonably well predicted from a quantitative standpoint as well. For the transactions, the model is less successful, but it represents a good start. The behavior of Greedy is handled well, but Conservative is predicted to perform somewhat better than Responsible for the transactions due to the slightly smaller average number of transaction I/Os under the Conservative policy. The problem with the model is that it does not capture the parallelism inherent in a multi-partition join, where the joins's temporary file I/O competes at the disk with its initial relation reads. This increased disk utilization by the join is what led Responsible to outperform Conservative in the first experiment (Figure 1). We are currently investigating ways to improve this aspect of the model.

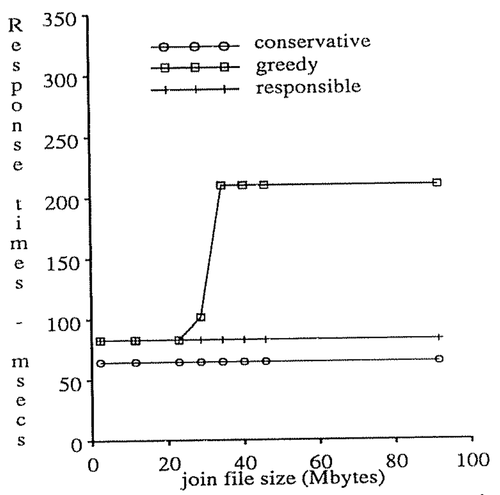

Figure 23: transaction response times.

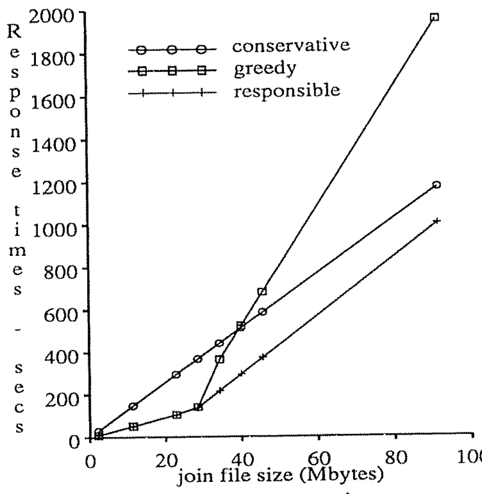

Figure 24: query response times.

Analytical model, 1 join, 1 transaction class, 1 disk, 100 transaction terminals

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented results from an experimental study of resource allocation and scheduling issues for mixed database workloads. The dominant conclusion from our experiments is that, for systems with workloads that contain transactions and queries, the traditional approach of basing memory allocation decisions solely upon the access characteristics of individual transactions and queries is inadequate. Rather, it is critical that memory allocation decisions take into account not only the potential for data sharing among concurrently executing transactions and queries, but also the potential for inter-transaction sharing by entire classes of transactions and/or queries. In retrospect, this is perhaps not surprising; however, it is interesting to note that the former is still viewed as an open problem [Falo91], and that the latter has received virtually no attention in any of the published work on relational database buffer management.

Our experimental results also clearly demonstrate that good resource allocation decisions depend on some non-trivial interactions between the characteristics of the workload and the system configuration. For example, the choice of whether to run multiple queries concurrently or serially, as well as the degree of concurrency that should be used to schedule each individual query (e.g. left-deep versus right-deep scheduling), must take into account the ratio of the join operand sizes to the available memory as well as the degree to which I/O resources are currently available in the system. As another example, when running join queries concurrently with a stream of transactions, the amount of memory to allocate to the query class depends upon several factors including the ratio of the join operand sizes to the amount of available memory, the ratio of the size of the index(es) used by the transaction stream to the size of memory, and the ratio of the join operand sizes to the size of this index.

Since the performance provided by a specific resource allocation decision can be somewhat unintuitive (e.g., we showed that "stealing" memory from a query can actually speed up the query), it is encouraging that simple analytical modeling can do quite well in predicting such performance trends. In our future work, we intend to improve upon our simple analytic model with the goal of eventually using it as a performance prediction tool within a runtime resource allocator. The detailed design of such an allocator is, of course, a major topic for future work. Finally, we ultimately plan to extend the results and intuition gained from this study to attack the even more challenging problem of resource allocation and scheduling for complex workloads in parallel database machines.

ACKNOWLEDGMENTS

We would like to thank Mike Franklin for his insight into Bubba's memory management and other helpful comments on an earlier version of this paper.

REFERENCES

[Agra89] Agrawal, D. and Sengupta, S., "Modular Syncronization in Multiversion Databases: Version Control and Concurrency Control," *Proc. ACM SIGMOD Conf.*, Portland, OR, June 1989.

[Bobe92a] Bober, P., and Carey, M., "On Mixing Queries and Transactions via Multiversion Locking," *Proc. 8th IEEE Data Engineering Conf.*, Phoenix, AZ, Feb. 1992.

[Bobe92b] Bober, P., and Carey, M., "Multiversion Query Locking," *Proc. 18th Int'l. VLDB Conf.*, Vancouver, BC, Canada, Aug. 1992, to appear.

[Bora90] Boral, H. et al, "Prototyping Bubba: A Highly Parallel Database System," *IEEE Trans. on Knowledge and Data Engineering* 2(1), March 1990.

[Care86] Carey, M. and Muhananna, W., "The Performance of Multiversion Concurrency Control Algorithms," *ACM TODS*, 8(4), December 1983.

[Chan82] Chan, A., et al, "The Implementation of an Integrated Concurrency Control and Recovery Scheme," *Proc. ACM SIGMOD Conf.*, Orlando, FL, June 1982.

[Chen92] Chen, M., Yu, P., Wu, K., "Scheduling and Processor Allocation for Parallel Execution of Multi-Join Queries," *Proc. 8th IEEE Data Engineering Conf.*, Phoenix, AZ, Feb. 1992.

[Chou85] Chou, H., and DeWitt, D., "An Evaluation of Buffer Management Strategies for Relational Database Systems, *Proc. 11th Int'l. VLDB Conf.*, Stockholm, Sweden, Aug. 1985.

[Cope88] Copeland, G., et al, "Data Placement in Bubba" *Proc. ACM SIGMOD Conf.*, Chicago, IL, June 1988.

[Corn89] Cornell, D., and Yu, P., "Integration of Buffer Management and Query Optimization in Relational Database Environment, *Proc. 15th VLDB Conf.*, Amsterdam, The Netherlands, Aug. 1989.

[DeWi84] DeWitt, D., et al, "Implementation Techniques for Main Memory Database Systems," *Proc. ACM SIGMOD Conf.*, Boston, MA, June 1984.

[DeWi90] DeWitt, D., et al, "The Gamma Database Machine Project," *IEEE Trans. on Knowledge and Data Engineering*, 2(1), March 1990.

[DuBo82] DuBourdieu, D., "Implementation of Distributed Transactions," *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, 1982.

[Effe84] Effelsberg, W., and Haerder, T., "Principles of Database Buffer Management," *ACM Trans. on Database Systems* 9(4), Dec. 1984.

[Falo91] Faloutsos, C., Ng, R., and Sellis, T., "Predictive Load Control for Flexible Buffer Allocation," *Proc. 17th Int'l. VLDB Conf.*, Barcelona, Spain, Sept. 1991.

[Ghan90] Ghandeharizadeh, S. and D. J. DeWitt, "Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines," *Proc. 16th VLDB Conf.*, Melbourne, Australia, Aug. 1990.

[Gray87] Gray, J., and Putzolu, F., "The 5 Minute Rule for Trading Memory for Disk Accesses and the 10 Byte Rule for Trading Memory for CPU Time," *Proc. ACM SIGMOD Conf.*, San Fransisco, CA, May 1987.

[Gray91] Gray, J., ed., *The Benchmark Handbook for Database and Transaction Processing Systems*, Morgan Kaufman Publishers, Inc., 1991.

[Haas90] Haas, L., et al, "Starburst Mid-Flight: As the Dust Clears," *IEEE Trans. on Knowledge and Data Eng.* 2(1), March 1990.

[Hong91] Hong, W., and Stonebraker, M., "Optimization of Parallel Query Execution Plans in XPRS," *Proc. 1st Int'l. Conf. on Parallel and Distributed Information Systems*, Miami Beach, FL, Dec. 1991.

[Hsia91] Hsiao, H. I. and D. J. DeWitt, "A Performance Study of Three High-Availability Data Replication Strategies," *Proc. 1st Int'l. Conf. on Parallel and Distributed Information Systems*, Miami Beach, FL, Dec. 1991.

[Livn87] Livny, M., S. Khoshafian, and H. Boral, "Multi-Disk Management Algorithms," *Proc. ACM SIGMETRICS Conf.*, Banff, Alberta, Canada, May 1987.

[Merc92] Merchant, A., Wu, K., Yu, P., Chen, M., "Performance Analysis of Dynamic Finite Versioning for Concurrent Transactions and Query Processing," *Proc. ACM SIGMETRICS Conf.*, Newport, RI, June 1992.

[Moha92] Mohan, C., Pirahesh, H., Lorie, R., "Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions," *Proc. ACM SIGMOD Conf.*, San Diego, CA, June 1992.

[Murp89] Murhpy, M. and Rotem, D., "Effective Resource Utilization for Multiprocessor Join Execution," *Proc. 15th VLDB Conf.*, Amsterdam, The Netherlands, Aug. 1989.

[Murp91] Murhpy, M. and Shan, M., "Execution Plan Balancing," *Proc. 1st Int'l. Conf. on Parallel and Distributed Information Systems*, Miami Beach, FL, Dec. 1991.

[Ng91] Ng, R., et al "Flexible Buffer Allocation Based on Marginal Gains," *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991.

[Pira90] Pirahesh, H., et al, "Parallelism in Relational Database Systems: Architectual Issues and Design Approaches," *IEEE 2nd Int'l Symposium on Databases in Parallel and Distributed Systems*, Dublin, Ireland, July 1990.

[Reis80] Reiser, M. and S. Lavenberg, "Mean Value Analysis of Closed Multichain Queueing Networks," *JACM* 27(2), April 1980.

[Ries78] Ries, D. and R. Epstein, *Evaluation of Distribution Criteria for Distributed Database Systems*, UCB/ERL Technical Report M78/22, UC Berkeley, May 1978.

[Sacc86] Sacca, D., and Schkolnik, M., "Buffer Management in Relational Database Systems," *ACM Trans. on Database Systems* 11(4), Dec. 1986.

[Schn89] Schneider, D. and D. DeWitt, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment," *Proc. ACM SIGMOD Conf.*, Portland, OR, June 1989.

[Schn90] Schneider, D. and D. DeWitt, "Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines," *Proc. 16th VLDB Conf.*, Melbourne, Australia, Aug. 1990.

[Schw90] Schwetman, H., *CSIM Users' Guide*, MCC Technical Report No. ACT-126-90, Microelectronics and Computer Technology Corp., Austin, TX, March 1990.

[Shap86] Shapiro, L., "Join Processing in Database Systems with Large Main Memories," *ACM TODS*, 11(3), Sept. 1986.

[Srin92] Srinivasan, V. and Carey, M. J., "Compensation-Based On-Line Query Processing," *Proc. ACM SIGMOD Conf.*, San Diego, CA, June 1992.

[Wils91] Wilschut, A., and Apers, P., "Dataflow Query Execution in a Parallel Main-Memory Environment," *Proc. 1st Int'l. Conf. on Parallel and Distributed Information Systems*, Miami Beach, FL, Dec. 1991.

[Wu91] Wu, K., Yu, P., Chen, S., *Dynamic Finite Versioning for Concurrent Transaction and Query Processing*, TR # RC 16633, IBM T.J. Watson Research Center, March 1991.