Accurate Modeling of The Hybrid Hash Join Algorithm*

Jignesh M. Patel **Michael J. Carey** **Mary K. Vernon**

Computer Sciences Department,
University of Wisconsin, Madison
{jignesh, carey, vernon}@cs.wisc.edu

Abstract: The join of two relations is an important operation in database systems. It occurs frequently in relational queries, and join performance is a significant factor in overall system performance. Cost models for join algorithms are used by query optimizers to choose efficient query execution strategies. This paper presents an efficient analytical model of an important join method, the hybrid hash join algorithm, that captures several key features of the algorithm's performance – including its intra–operator parallelism, interference between disk reads and writes, caching of disk pages, and placement of data on disk(s). Validation of the model against a detailed simulation of a database system shows that the response time estimates produced by the model are quite accurate.

1 Introduction

Relational database systems organize information into a collection of tables. The relational *join* operator is used to relate information from two or more tables. Thus, joins are a frequently occurring operation in relational queries. Additionally, joins are one of the most expensive operations that a relational database system performs. Joining two large tables can consume a significant amount of the system's CPU cycles, disk bandwidth, and buffer memory. For this reason, efficient join algorithms are a critical factor in determining relational database system performance.

Accurate and efficient join algorithm cost models are also important, as they are needed by relational query optimizers (which employ cost-based optimization algorithms) in order to derive efficient processing plans for relational queries. Furthermore, simulation is currently used extensively for studying parallel execution strategies for complex queries [CLYY92] and for evaluating strategies for handling complex multiuser workloads in centralized database systems [BCL93, MD93]. Most simulation studies of parallel database systems have not explored truly large systems (e.g. 100's of nodes) due to the prohibitive costs of simulating such systems in detail. If accurate analytical models could be developed, such scheduling strategies could be evaluated more quickly and over wider regions of the system design space, including more realistic system sizes. As a starting point, accurate analytical models that capture centralized query performance are required. Our focus here is the development of such a model for the hybrid hash join algorithm that was introduced in [DKO+84].

Simple analytic cost models for hash based join algorithms were first presented in [DKO+84, Sha86]. These models only count the total number of page I/O operations in the algorithm. A more complete model, recently presented in [HCL93], breaks up the cost of performing disk I/O operations into various parts and carefully counts each part. However, intra–operator disk contention and CPU costs are not addressed in this model. In [LY90], the authors investigated the effectiveness of parallel hybrid hash join algorithms for a system running a single join. However, their model of the individual processing nodes is quite simplistic, e.g., it does not consider the impact of intra–operator disk contention. This paper develops an accurate model of the hybrid hash join algorithm that includes these costs and is nevertheless efficient to evaluate.

We develop an analytical model of the hybrid hash join algorithm for the case of a single join operation, implemented using two processes, running on a single node of a database system. This situation presents two particular challenges for creating an accurate analytical model. First, the intra-operator disk interference patterns are more complex to analyze than the random interference that typically occurs between unrelated processes. Second, disk seek times are influenced by file placement as well as by interference. The model also captures other significant system behavior, such as intra–operator synchronization and caching of disk pages, which also affects the performance of the algorithm. Our model is based on the approximate mean–value analysis, an approach that has proven accurate in modeling parallel architectures [VLZ88, WE90, CS91]. Specific system effects are captured by modifying the response time equations. Validation of the model against the simulator used in [BCL93, MD93] shows that the model yields accurate results.

The remainder of the paper is organized as follows: Section 2 describes the hybrid hash join algorithm. Section 3 describes the analytical model for the hybrid hash join. The results of the validation of the model and several other experiments, including a comparison with previous models, are described in Section 4. Finally, Section 5 contains our conclusions.

*This work was partially supported by the IBM Corporation through a Research Initiation Grant, and by the NSF Grants CCR 9024144 and CDA 9024618.

2 Background

The tables in a relational database system are called relations. Each relation is structured as a set of tuples, with each tuple consisting of an ordered list of attributes. A (binary) join operator, one of the fundamental operations in a relational database system, *relates* tuples from two relations by matching one or more attributes of the tuples according to some specified condition. For example, the condition for the equijoin operator, by far the most common form of join, is that the attribute values be equal.

Various join algorithms have been proposed for performing the equijoin operation [ME92, Gra93]. In some cases, the process of matching tuples is made faster by the existence of an access structure, such as a B-tree [Com79], on the join attribute of one of the relations. However, in the case of an unanticipated join, or when the relations to be joined are both very large, an *ad hoc* join algorithm is normally used. Moreover, as database sizes increase, and interest in running complex decision support queries grows, the importance of efficient *ad hoc* join algorithms continues to increase. Sort–merge and hash based join algorithms [BE77, Bra84, DKO$^+$84, Sha86] are favored for *ad hoc* joins. A hash based algorithm known as hybrid hash has been shown to be particularly effective for performing ad hoc joins [DKO$^+$84, Sha86]. The details of this algorithm and key aspects of its implementation are discussed in the next two sections.

2.1 The Hybrid Hash Join Algorithm

Hybrid hash, like other hash–based join algorithms, uses hashing to improve the speed of matching tuples. That is, hashing is used to partition the two input relations such that a hash table for each partition of the smaller input relation can fit in main memory. Corresponding partitions of the two input relations are then joined by building an in–memory hash table for the tuples from the smaller input relation, and then probing the hash table with the tuples from the corresponding partition of the larger input relation.

The tasks involved in a hybrid hash join are frequently implemented as a collection of processes, particularly in parallel database systems [DG92]. One benefit of doing so is that the construction (and later probing) of the hash table for a given join operation can proceed in parallel with the reading of the input relations from disk. A more substantial benefit, particularly for complex queries, is that scalable parallel data flow implementations are possible. For example, the tasks of reading the input relations and of building/probing the hash table can each be implemented as a set of parallel processes, with each process running on a separate node. Each process in the set performs the same task on different data (e.g., the data that is stored on its node), passing its output tuples to the process responsible for performing the next task on those tuples. In a parallel database system, such a multiple process implementation allows for pipelining between multiple join operators. For these reasons, we will focus our attention here on the multiple process implementation of the hybrid hash algorithm.

The details of the hybrid hash join algorithm for a centralized database system, or for a node of a parallel database system, are as follows. Let R and S be the two input relations to be joined and let $|R|$ and $|S|$ denote the number of pages in each relation. Without loss of generality, we assume that $|R| \leq |S|$. The smaller relation, R, is called the building relation, and the larger relation, S, is called the probing relation. The join is implemented by two processes, a scan process and a join process.[1] The scan process reads pages from the input relations and passes them to the join process via a buffer.[2] Frequently, join queries also involve selection predicates that restrict the tuples of the base relations (R and S) that participate in the join. When such predicates are present, the scan process applies the predicates to the tuples of the input relations and only passes on those tuples that satisfy the predicates.

Let $B + 1$ be the number of partitions of each input relation. The join process divides the two relations into these partitions and then processes each partition. The execution of the join proceeds in $B + 1$ consecutive phases, with each phase having two consecutive operations called the build and probe operations. Phase 0 is illustrated in Figure 1. In the first part of this phase (phase-0-build), the scan process scans the pages of the building relation R and applies any selection predicate. Tuples that satisfy the predicate, and are thus eligible for the join, are buffered and passed to the join process in page-size chunks. The join process reads the pages from the buffer and hashes each tuple to a value between 0 and B. Tuples that hash to the value 0 are inserted into an in–memory hash table. Tuples that hash to the values 1..B are written to a page-sized output buffer that is allocated for that partition. Each of the partitions, $1..B$, has a file on disk (called a "bucket file") to which the corresponding output buffer page is flushed each time it becomes full.

In the second part of phase 0 (phase-0-probe), the scan process scans the probing relation S, applies any selection predicate, and writes the selected tuples to the buffer. The join process reads pages from the buffer and applies the same hash function used in phase-0-build. If a tuple hashes to the value 0, the join process probes the in–memory hash table for matching tuples and outputs any matching tuple pairs; these constitute the result of the join. The join process writes tuples that hash to the values $1..B$ to the appropriate output buffer pages, which are flushed to corresponding S bucket files on disk when full.

At the end of phase-0, the R and S tuples in partition

[1]Actually, the join is usually implemented by three processes – a scan process for each of the input relations and a join process. However, the two scan processes are used serially. Hence, conceptually and for the purpose of modeling, we can treat the two scan processes as a single process with a synchronization step in the middle.

[2]For an implementation of the hybrid hash join on a parallel database machine, the join and scan processes may be on different nodes. In that case, the buffer is used for sending and receiving pages over the network.

0 have been joined and the remaining tuples are waiting in appropriate pairs of bucket files on disk. In each of the remaining B phases, the corresponding partitions of the building relation R and the probing relation S are joined by the join process. Each phase-i ($1 \leq i \leq B$) has two parts, phase–i–build and phase–i–probe. In phase–i–build, the join process reads the i^{th} partition of the building relation R from disk, one page at a time, and builds an in–memory hash table containing its tuples. The join process then moves to phase–i–probe, where it reads the i^{th} partition of the probing relation S one page at a time. For each tuple in the partition, the join process probes the in–memory hash table for matches, and it appends any matching tuple pairs to the result of the join.

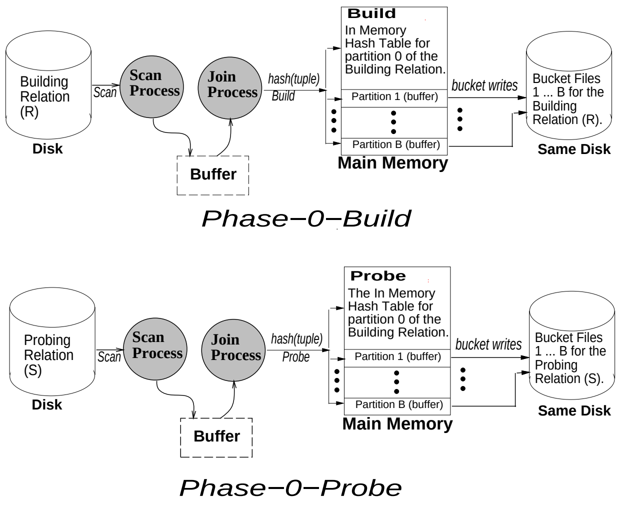

Figure 1: Phase-0 of the hybrid hash join.

It should be noted that if the *hash function* used by the join results in a non-uniform distribution of tuples across the $B + 1$ partitions, then some partitions may not fit entirely in main memory. This may happen if the hash function is imperfect or if the values of the join attribute in the building relation R are skewed. To handle most such cases, the database system typically allocates a few extra pages to the hash table (i.e., it plans for a small number of overflow pages when deciding how many partitions are needed). Techniques for handling more significant data skew while joining two relations are discussed in [Sha86, KNT89]. The model developed in Section 3 assumes that no partitions overflow, but it could be extended to model overflows if desired.

2.2 Salient Aspects of the Algorithm

In this section we discuss several special characteristics of the implementation of the algorithm that must be considered when constructing the model.

First, the *buffer* that is used for passing pages between the scan and join processes (illustrated in Figure 1) is of finite size. As a result, in phase–0, the scan process must block whenever this buffer is full. Furthermore, the scan process can never get more than N pages ahead of the join process (where N is the size of the buffer in pages).

Second, to reduce the cost of sequential reads, disks often have a *disk cache* for prefetching data. With a disk cache of size K, a disk read can prefetch up to K sequential blocks of additional data when processing a read request. Subsequent sequential reads will find the requested data in the disk cache and will not have to actually perform a disk I/O. Furthermore, in current systems, disk scheduling is done by the operating system (which does not know which pages are available in the disk cache). Thus, all read requests will actually queue for the disk, but requests that are serviced from the disk cache complete quickly once the read request is issued. (Future disk controllers may become responsible for disk scheduling and may eliminate the unnecessary queueing.)

Third, the *placement* of files on the disk(s) can have a significant effect on the execution time of the join. If the input relations (R and S) and the temporary bucket files are on the same disk, then typically the input relations are read from one cylinder while the bucket file pages are written on another cylinder. Compared to the case where a separate disk is used for the bucket files, when a single disk is used the disk seek times can be highly non-uniform and the disk arm may experience a lot of movement in phase–0. Specifically, read requests that do not have intervening write requests will have much smaller seek times than read requests that occur immediately after an intervening write request. The same holds for write requests with and without intervening read requests. Thus, interleaved read and write requests increase each other's service times.[3]

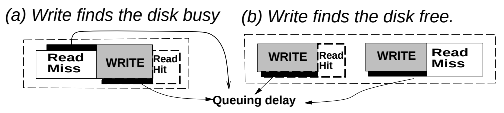

Figure 2: The Read/Write Interference Patterns.

Finally, when the R and S data files and the temporary bucket files are on the same disk, particular *I/O interference patterns* occur between the scan and join processes in phase 0. These interference patterns are illustrated in Figure 2. Recall that in phase–0, the scan process repeatedly reads a disk page, applies a selection predicate to the tuples on the page, and then writes the selected tuples to a buffer. Thus, the scan performs very little computation between two disk read requests. This implies that when a write operation queues behind a read request, it will cause the next read request (which will arrive very soon after the write is initiated) to queue for nearly the entire time of the write operation. Furthermore, the read request behind which the write queues is likely to be a read–miss, because read hits have negligible service times. In addition, since the scan process is I/O bound, the write operation that

[3]From this description, it would seem advantageous to use a separate disk for the temporary bucket files, thereby avoiding interference between the disk reads and writes. However, with current technological trends, a data placement strategy that fully declusters all (except for very small) relations across all available disks actually achieves the best performance [MD].

queues behind the read–miss is likely to queue for most of the read access time. This scenario is depicted in Figure 2(a). Figure 2(b) shows the case where a write arrives at an idle disk; the write request will also cause the next read request (a read–hit or a read–miss) to queue for nearly the entire write access time.

3 The Analytical Model

This section develops an approximate mean value analysis (MVA) model of a single hybrid hash join query executing on a stand–alone system consisting of a single CPU and a single disk. The specific behaviors of the processes discussed in Section 2.2, such as their I/O interference pattern and the non-uniform seek times, are captured by modifying the MVA response time equations. Two models, a process model and a system model, are used to define the behavior of the join. The process model is used to capture the overall synchronization behavior of the processes as they move from one phase of the algorithm to the other, while the system model represents the physical resources in the database system as service centers in a queueing network.

3.1 The Process Model

The process model represents the process structure (i.e., the phase behavior of the scan and join processes) in the algorithm's implementation. Both processes are simultaneously active throughout phase–0 of the join. As explained in Section 2, the scan process performs the disk reads and does some computation on the tuples in the pages read, while the join process performs other computation on the pages received from the scan and occasionally writes data pages to the disk. The two processes are modeled as two customers in the queueing network model, each in a separate class, switching classes as processing progresses from the build phase to the probe phase. The class switching, illustrated in Figure 3, is necessary so that different resource requirements can be specified in the different phases. In phases $1..B$, only the join process is active, and there is no interference from other customers; the time spent executing these phases, denoted by D_5 and D_6, can be computed by simply summing their resource requirements.[4]

	Phase–0–Build	Phase–0–Probe		Phase–i–Build	Phase–i–Probe
Scan/ Reader	Class 1	Class 3		——	——
Join/ Writer	Class 2	Class 4	Join/ Reader	Class 5	Class 6

Figure 3: The process model.

To determine the overall response time (i.e. execution time) for the join, we use the system queueing network model described in the following section to compute the mean response times for classes 1 through 4, denoted by $R_1...R_4$. The overall mean response time of the join (R) is then estimated by

$$R \approx max(R_1, R_2) + max(R_3, R_4) + D_5 + D_6 \qquad (1)$$

We note that the overall mean response time (R) is an approximation, as each of the first two terms is the maximum of two mean response times (and not the mean of the maximum response time).

3.2 The System Model

The system that we are modeling consists of two resources, a CPU and a disk. The simple queueing network model that represents the system is shown in Figure 4. The delay center in the network represents time during which either the scan process or the join process blocks due to synchronization on the buffer that they use to communicate. A similar approach was used to model synchronization delays in [HT83]. The scan process blocks if the buffer is full, whereas the join process blocks if the buffer is empty, as described in Section 2. Since the processes are fairly tightly synchronized (i.e., the size of the buffer is assumed to be fairly small), we assume that in fact both processes complete almost simultaneously. We thus determine the mean time at the delay center for the faster process by iteratively solving the model and setting this value to the difference between the estimated mean execution times of the two processes. For example, in a phase where the join is the faster process, a synchronization delay is added for the join; this delay is set to a value that makes the overall mean response time of the join the same as that of the scan. The queueing network model is solved using approximate MVA techniques [RL80, LZGS84]. To capture the specific interference patterns and service times of the customers at the disk, we create customized response time equations for the disk requests. These customized equations are presented in Section 3.3.2.

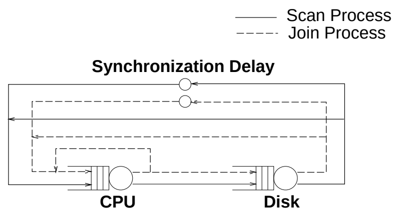

Figure 4: The queueing network model.

3.3 The Queueing Model Equations

The inputs to the model, shown in Tables 1 and 2, consist of the system hardware and the software parameters. The seek factor mentioned in Table 1 is a constant that, when multiplied by the square root of the seek distance (in cylinders), determines the seek time in milliseconds [BG88]. The settle time is the average time that it takes for the disk arm to settle over the appropriate cylinder after a seek. When the hash table of a partition of the building relation is formed, some additional memory is required due to data structure overhead (i.e., this is additional memory beyond the space required to simply hold the R tuples of the partition). The hash factor in Table 2, F_{hash}, gives the factor by which the memory allocation is expanded. In other words, when

[4]For systems with multiple queries and/or transactions, of course, the join process would experience interference in these phases. Thus, the customer representing the join process would cycle in the queueing network during these phases, whereas the customer representing the scan process would visit a delay center to represent its period of inactivity. The mean time at the delay center would be computed from an iterative solution of the overall model.

multiplied by the number and size of the tuples in the R partition being built, this factor gives the total amount of memory required for holding the partition's hash table. (Note that F_{hash} is sometimes referred to as the "fudge factor" in the database literature). $Size_{tuple}$ in Table 2 gives the size of the R and S tuples in bytes. In general, of course, the tuple sizes of relations R and S will be different; for simplicity, we assume here that the two relations have tuples of the same size, though generalizing the model in this regard is straightforward. The meanings of the remaining parameters should be clear from their descriptions in the tables.

Symbol	Parameter
$MIPS$	MIPS rating of the CPU
$Size_{mem}$	Main memory size (in pages)
$Size_{page}$	Size of a page (in bytes)
F_{seek}	Seek factor
F_{pref}	Num. of pgs. fetched by a disk read (incl. the requested and prefetched pgs)
T_{rot}	Maximum rotational latency
T_{xfer}	Transfer rate
T_{settle}	Settle time

Table 1: Hardware Input Parameters.

Symbol	Meaning
F_{hash}	Extra space required by a hashed tuple
I_{send}	#instr to initiate a page send
I_{recv}	#instr to initiate a page receive
I_{sel}	#instr for applying the select predicate to a tuple
I_{hash}	#instr for inserting a tuple into a hash table
I_{probe}	#instr for probing the hash table once
I_{copy}	#instr for a byte copy
$I_{startIO}$	#instr for starting a disk I/O
$Size_{tuple}$	Size of a tuple (in bytes)
$\|R\|$	Size of the building relation (in pages)
$\|S\|$	Size of the probing relation (in pages)

Table 2: Software Input Parameters.

3.3.1 Estimating The Mean Visit Counts for the Queueing Network

In this section, we derive the mean visit counts for the resources in the queueing network using the parameters presented in the previous section and the algorithm description of Section 2.1. Let:

$\|R_0\|$	= the size (in pages) of the in-memory portion of R during phase 0
$\|R'\|$	= the number of pages of R that are written to the disk
$\|S'\|$	= the number of pages of S that are written to the disk
B	= the number of buckets for each input relation, excluding the in-memory bucket

In terms of these quantities, the numbers of visits to the CPU and disk for each customer class are given in Table 3. Note that the number of visits by class c to center k is denoted by $V_{c,k}$, where k is either the CPU or the disk.

Visit Counts	Class (c)					
	1	**2**	**3**	**4**	**5**	**6**
$V_{c,CPU}$	$\|R\|$	$\|R\|$	$\|S\|$	$\|S\|$	$\|R'\|$	$\|S'\|$
$V_{c,Disk}$	$\|R\|$	$\|R'\|$	$\|S\|$	$\|S'\|$	$\|R'\|$	$\|S'\|$

Table 3: Visit Counts for the Queueing Model.

For the workloads considered in the simulation experiments that we have used to validate the model, all of the R and S tuples participate in the join, i.e. the selection predicate applied by the scan process does not eliminate any tuples. In this case,

$$|R'| = |R| - |R_0|$$

The values for the quantities $|R_0|$ and $|S'|$ are computed as follows [Sha86, HCL93]. During the processing of partition 0, the building relation R has to be scanned and B buckets of relation R have to be written to the disk. If we assume one input buffer for reading R and one output buffer for each of the B buckets, we have $Size_{mem} - B - 1$ pages for holding the hash table for partition 0 of R. In phases 1..B of the algorithm, we will have $Size_{mem} - 1$ memory pages for holding the hash table (with one page being reserved for the input buffer). Thus, taking hash table overheads into account, the optimal value of B will be the smallest value satisfying the equation:

$$Size_{mem} - B - 1 + B \times (Size_{mem} - 1) \geq |R| \times F_{hash}$$

which yields

$$\begin{aligned} B &= \lceil\left(\frac{|R| \times F_{hash} - Size_{mem} + 1}{Size_{mem} - 2}\right)\rceil \\ &\textit{and} \\ |R_0| &= \lfloor\left(\frac{Size_{mem} - B - 1}{F_{hash}}\right)\rfloor \end{aligned}$$

Assuming that the fraction of tuples in S that hash to buckets 1 through B is the same as the corresponding fraction of tuples in R,

$$|S'| = \lceil |S| \times \tfrac{|R'|}{|R|} \rceil \ .$$

3.3.2 Disk Response Time Equations

To reflect the read/write disk interference patterns and the non-uniform average seek times described in Section 2.2, we modify the standard MVA response time equations for the disk as described below. The notation used in this section is given in Table 4.

T_{Write} includes the average rotational latency, the settle time, and the time required to transfer a page during a disk write. Since a disk read–miss reads a total of F_{pref} pages, the transfer time in T_{Read} represents the time for transferring all of these pages. Since $\frac{|R|}{F_{pref}}$ represents the number of read–misses in the Phase–0–build, ω represents the ratio of writes to read misses in this phase. $Ratio_{\frac{RM}{W}}$ represents the fraction of write requests that incur extra seek time due to an interfering read–miss. Again, since the scan process is I/O bound,

Symbol	**Meaning**	**Value**
T_{Write}	Average disk access time for a write (excluding seek time)	$\frac{T_{rot}}{2} + T_{settle} + T_{xfer}$
T_{Read}	Average disk access time for a read–miss (excluding seek time)	$\frac{T_{rot}}{2} + T_{settle} + T_{xfer} \times F_{pref}$
ω	Ratio of writes to read misses	$\frac{\lvert R'\rvert \times F_{pref}}{\lvert R\rvert}$
$Ratio_{\frac{RM}{W}}$	fraction of write requests that find the arm over the read cylinders	$min\left(1, \frac{1}{\omega}\right)$
$Ratio_{\frac{W}{RM}}$	fraction of read requests that find the arm over the write cylinders	$min\,(1, \omega)$
$Ratio_{\frac{W}{R}}$	# writes between two reads	$\frac{\lvert R'\rvert}{\lvert R\rvert}$
f_{CW}	Fraction of writes that occur consecutively	$\frac{max[0,\omega-1]}{max[0,\omega-1]+1}$
$Seek_{\phi-0-B}$	Average seek time of an interfered disk request in a Phase-0-Build	$\sqrt{AvgSeek_{Ph\_0\_B}} \times F_{seek}$
$Seek_{\phi-0-P}$	Average seek time of an interfered disk request in a Phase-0-Probe	$\sqrt{AvgSeek_{Ph\_0\_P}} \times F_{seek}$
$Seek_{CW}$	Average seek time seen by consecutive writes in a Phase-0	$\sqrt{AvgSeek_{CW}} \times F_{seek}$

Table 4: Notation for the Disk Response Time Equations.

we assume that writes are interleaved with reads in an approximately uniform way. If $\frac{1}{\omega} > 1$, there is (on average) more than one read miss between two writes. Since only the last read-miss interferes with the write, we take the minimum of $\frac{1}{\omega}$ and 1 in computing $ratio_{\frac{RM}{W}}$. Similarly, $Ratio_{\frac{W}{RM}}$ represents the fraction of read–miss requests that incur extra seek time due to an interfering write. Furthermore, since most writes cause a read to queue (recall Figure 2), $Ratio_{\frac{W}{R}}$ represents the fraction of read requests that have to queue behind a write request. Note that $Ratio_{\frac{W}{R}}$ is always less than one since the number of reads always exceeds the number of writes. If the number of writes exceeds the number of read–misses, there will be sequences of writes that are not interfered with by read–misses. However, these writes are to random bucket files, as opposed to consecutive read operations, which are sequential. Thus, f_{CW} of the writes incur a relatively low seek cost equal to $Seek_{CW}$.

To explain the last three terms of Table 4, we must consider the particular placement of data on disk. For the experiments in Section 4, we will assume that the building and probing relation files are laid out on the disk in file groups as shown in Figure 5. A file group consists of a collection of files of the same size, and input relation files are randomly picked from the files in their file group. Thus, the building relation R is randomly chosen from the building file group while the probing relation S is randomly chosen from the probing file group. The bucket files produced during the join are written at the end of the disk. Knowing the layout of the files, the average seek distances for interfered disk requests can be estimated as indicated in Figure 5. For phase–0–build, the estimated seek distance includes half of the cylinders in the building file group, all of the cylinders in the probing file group, and half of the cylinders used for the bucket files. For phase-i, the average seek distance is estimated as $\frac{1}{3}$ of the total number of cylinders used for writing the bucket files [BG88]. (Although here we assume detailed knowledge of the data layout, we will show later that the model is quite accurate even when knowledge of exact file locations is not assumed.)

Given the notation and calculations described above, the disk response time equation for phase-0-build is de-

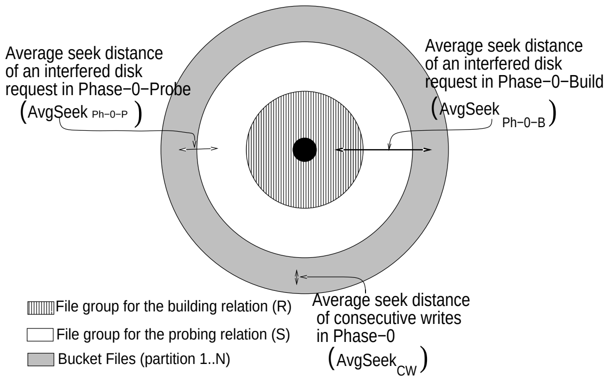

Figure 5: File layout and average seek distances.

rived as follows. First, the average time that it takes to service a read–miss is:

$$S_{readMiss} = T_{Read} + Ratio_{\frac{W}{RM}} \times Seek_{\phi-0-B} \quad (2)$$

where T_{Read} includes the rotational latency, settle time, and transfer time, and the second term represents the average seek time. Similarly, the average time that it takes to service a write request, including the seek time for consecutive writes, is:

$$\begin{aligned} S_{write} &= T_{Write} + Ratio_{\frac{RM}{W}} \times Seek_{\phi-0-B} \\ &\quad + f_{CW} \times Seek_{CW} \quad (3) \end{aligned}$$

We can now express the disk response time equations for classes 1 and 2, in terms of $S_{readMiss}$ and S_{write}, as follows:

$$R_{1,Disk} = \frac{S_{readMiss}}{F_{pref}} + Ratio_{\frac{W}{R}} \times S_{write} \quad (4)$$

$$R_{2,Disk} = S_{write} + Ratio_{\frac{RM}{W}} \times S_{readMiss} \quad (5)$$

Both response time equations have two parts: the time required to perform the disk I/O, and the time spent waiting in the disk queue. The queueing time of a read at the disk in equation (4) is estimated as $Ratio_{\frac{W}{R}} \times S_{write}$ because most writes cause a read to queue (refer to Section 2.2). $Ratio_{\frac{W}{R}}$ gives the fraction of reads that queue behind a write, while S_{write} gives the queueing time of such read requests. Similarly, in equation 3, $Ratio_{\frac{RM}{W}} \times S_{readMiss}$ gives the average queueing time of a write at the disk.

Disk response time equations for classes 3 and 4 in phase-0-probe ($R_{3,Disk}$ *and* $R_{4,Disk}$) can be derived similarly. The only changes required are using $|S|$ instead of $|R|$ and $|S'|$ instead of $|R'|$ in computing the ratios, and using $Seek_{\phi-0-P}$ instead of $Seek_{\phi-0-B}$.

Since phase–i consists only of disk reads that are largely sequential, the mean phase–i disk response times can be estimated as:

$$R_{5,Disk} = R_{6,Disk} = \frac{T_{read}}{F_{pref}} \tag{6}$$

3.3.3 Per Visit CPU Service Requirements

To compute the CPU service requirements, we first calculate the number of tuples per page as:

$$N_{tuples} = \lfloor \frac{Size_{page}}{Size_{tuple}} \rfloor \tag{7}$$

Using this and the other input parameters listed in Tables 1 and 2, the per visit CPU service time requirements can be expressed by the following equations:

$$\begin{aligned} R^{pv}_{1,CPU} &= R^{pv}_{3,CPU} \\ &= (N_{tuples} \times Size_{tuple} \times I_{copy})\,/MIPS \\ &\quad + (N_{tuples} \times I_{sel} + I_{send})\,/MIPS \\ R^{pv}_{2,CPU} &= (I_{recv} + N_{tuples} \times (I_{sel} + I_{hash}))\,/MIPS \\ R^{pv}_{5,CPU} &= (I_{startIO} + N_{tuples} \times (I_{sel} + I_{hash}))\,/MIPS \end{aligned}$$

The service requirements $R^{pv}_{4,CPU}$ and $R^{pv}_{6,CPU}$ are similar to $R^{pv}_{2,CPU}$ and $R^{pv}_{5,CPU}$, respectively, with the only difference being that the term I_{hash} is replaced by I_{probe} .

4 Validation of the Model

In this section, we first evaluate the accuracy of our base analytical model by comparing it against a detailed simulation of a database system. We then describe the results of several experiments where we change some of the model assumptions. Finally, we compare the model with two previous models and highlight the differences in their accuracy.

4.1 Simulation Model

The simulator that we used for validating our analytical model, ZetaSim [Bro92], is a detailed simulation model of the Gamma parallel database machine [DGS+90]. For the purpose of validating the analytical model, the simulated system is configured with a single node consisting of one CPU and a 1.2GB disk, which is similar to the configuration used to study memory management issues in [BCL93]. The parameter settings of the simulator are listed in Table 5. The parameters for the disk closely match the characteristics of a Fujitsu disk model M2266, while the instruction counts are largely based on measurements from the Gamma database machine implementation. The simulation model includes details of data placement, buffer management, the elevator disk scheduling algorithm, disk cache behavior, and so on [Bro92, BCL93].

For validation purposes, the relations were laid out on the disk in two file groups, the building and probing groups, as assumed in the analytical model (see Figure 5). We varied the size of the relations in the file groups while keeping the total size of each file group at

Hardware Parameters	**Value**
Mips rating of the CPU	20 MIPS
Page size (in bytes)	8192
Main memory size (in pages)	4096
Size of buffer between the join and the scan processes	8 pages
Disk size	1.2GB
Additional number of pages prefetched by a disk read	4
Seek factor	0.617
Maximum rotational latency time	16.667 ms
Transfer rate	3.09 MB/sec
Settle time	2.0 ms
Algorithm Cost Parameters	**Value**
Hash factor	1.2
Tuple size (in bytes)	200
# instr to initiate a page send	1000
# instr to initiate a page receive	1000
# instr for applying the select pred.	300
# instr for inserting a tuple into the hash table	100
# instr for probing the hash table	200
# instr for a byte copy	1
# instr for starting a disk I/O	1000

Table 5: Simulation Parameters.

0.5 GB. Since the largest relation that we evaluate has 500K tuples ($\approx$ 100MB), the configured disk of size of 1.2GB was sufficient for joining the relations. The system workload consists of a single join query which randomly chooses its building and probing input relations from the corresponding file groups. Workloads similar to this were used in [BCL93, MD93].

4.2 Results of the Validation

In our initial model validations, we let the build and probe relations be of equal size. The graphs in Figures 6 to 9 compare various measures estimated by the analytical model and by the simulator as the size of the relations is varied. A key point to note in each of the graphs is that there is a discontinuity in the curve at the point where the build relation becomes too large to fit in memory (i.e., just after 100K tuples). Figure 6 gives the overall join execution time predicted by the model and by the simulation, showing that the overall predictions of the analytical model are highly accurate.

Looking at a more detailed measure, Figure 7 shows that the qualitative behavior of the mean response time for disk write operations in phase 0 is also accurately predicted by the analytical model; however the model overestimates the mean write response time in the probe part of phase 0 for relation sizes that are slightly larger than the allocated memory. The reason for the model overestimating the write times for these relation sizes is as follows. In the response time equations for the disk (equation 3), we assumed that the queueing time of a write behind a read–miss is nearly the same as the read access time. This is not entirely true for small relations; small relations have few buckets and hence do

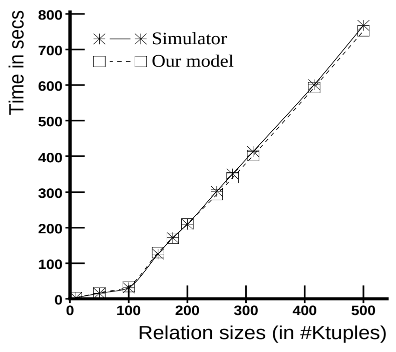

Figure 6: Join Execution Time ($|R| = |S|$)

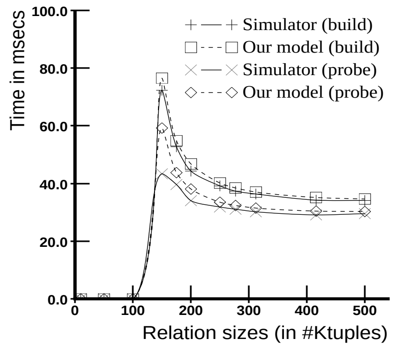

Figure 7: Disk Write Time for Phase–0 ($|R| = |S|$)

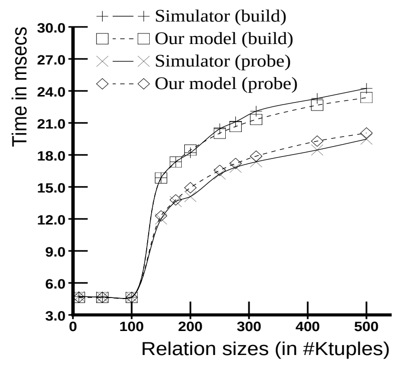

Figure 8: Disk Read Time for Phase–0 ($|R| = |S|$)

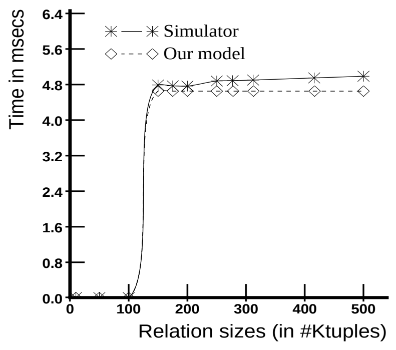

Figure 9: Disk Read Time for Phase–i, $1 \leq i \leq B$ ($|R| = |S|$)

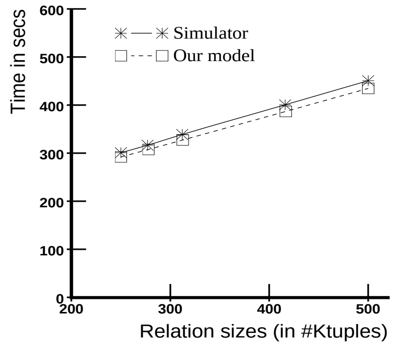

Figure 10: Join Execution Time With Inner Relation Size of 250K ($|R| = 250K, |S|$ varied)

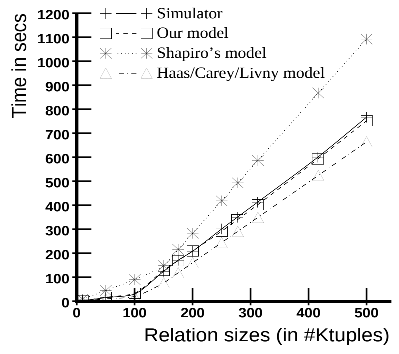

Figure 11: Join Execution Time of Various Models Using One Disk ($|R| = |S|$)

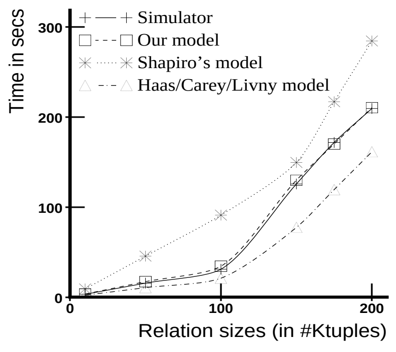

Figure 12: Join Execution Time of Various Models Using One Disk ($|R| = |S|$); blown up version of the lower half of Figure 11

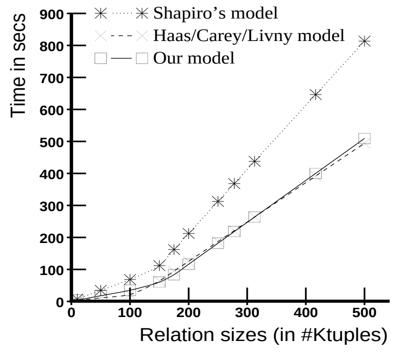

Figure 13: Join Execution Time of Various Models Using a Separate Bucket File Disk ($|R| = |S|$)

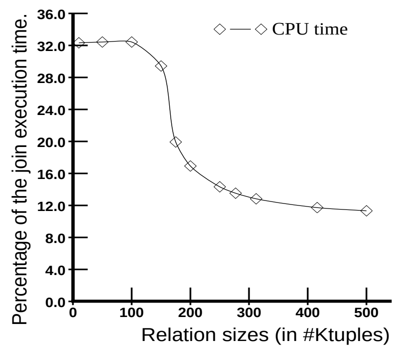

Figure 14: Percentage Of Time Spent At The CPU When Using a Separate Bucket File Disk ($|R| = |S|$)

relatively few writes per read. In this case, the join may process many pages before filling up one of its output buffers, and the resulting write may therefore arrive significantly after the start of the read access. Thus, the difference in time between the arrival of a read–miss and a write may be much greater for small relations than for larger relations. However, it is not necessary to derive a more complex expression to correct this discrepancy in the model – for these relation sizes the scan process is slower than the join process, so the scan determines the overall execution time. Mean disk read response time is modeled accurately for the scan process at all relation sizes (as discussed next). For larger relation sizes, the join process is the slower process and determines the overall join execution time; in this case, the mean disk write response times are accurate, as can be seen in Figure 7.

As shown in Figure 8, the mean response times predicted by the model for disk read operations in phase–0 match closely with those of the simulator over the entire range of relation sizes examined. The match is particularly close for smaller relation sizes, where (as mentioned above) these read response times partially determine the total execution time. Finally, Figure 9 shows the mean response time for read operations in phases other than phase 0. The model assumes that reads are sequential and hence estimates a constant read time for phase–i (see equation 6). However, the reads of the bucket files may involve small seeks, as the pages for the bucket files are allocated by the simulated system in extents. These extents may not be contiguous, leading to occasional inter–extent seeks that our analytical model does not account for.

To test the robustness of the model, we carried out a few additional experiments where we changed some of the model's assumptions.

In the first of these experiments, we modified the assumption of how files are laid out on disk (refer to Section 3.3). While the data for the base relations in a complex multi-join query may be laid out in a particular known fashion, the location of intermediate relations created during multi-join query processing may not be known a priori. For this experiment, the building and probing relations are again assumed to be of the same size, but are selected randomly from a *single* file group of size 1 GB, which is again populated with relations of the specified size. The modifications required in the analytical model are in the seek times used in the disk response time equation in phase–0 (equations 2 through 3). In this case, the arm seek in each part of phase 0 is equal to the average of the values that we used previously for phase–0–build and phase–0–probe (i.e., $Seek_{\phi-0-B}$ and $Seek_{\phi-0-P}$). The join execution times predicted by the model for this experiment matched the simulation estimates even more closely than in Figure 6. Since the curves are very similar to those in Figure 6, and space is limited, we do not include the graph here.

In the experiments thus far, the building and probing relation have both been of the same size. We now present an experiment where we vary the relative size of the building and probing relations. For this experiment, we used building and probing file groups of the same size (about 0.5 GB each). We fixed the building relation size to 250K tuples (50 MB), which is considerably larger than the size of the main memory hash table, and varied only the probing relation size. Figure 10 shows the join execution times from the model and the simulator; the x–axis in this graph represents the size of the probing relation (S). As can be seen from the graphs, the predictions of the model are again in close agreement with the results of the simulator.

4.3 Comparison With Previous Join Cost Models

To further illustrate the value of our relatively complete analytical model, we next compare this model with two previous hash join cost models. We use the same parameter values for all of the models, namely the values listed in Table 5, and we let the two input relations have equal size for these experiments.

The first model for comparison is the one proposed by Shapiro [Sha86]. Using the notation defined in this paper and letting IO denote the average time for an I/O request, and q denote the ratio $\frac{|R_0|}{|R|}$, the hybrid hash cost derived in [Sha86] is:

$$\begin{aligned} Cost_S \;=\; & (|R| + |S|) \times N_{tuples} \times I_{hash} + \\ & + (|R| + |S|) \times N_{tuples} \times (1 - q) \times \\ & \quad I_{copy} \times Size_{tuple} \\ & + 2 \times (|R| + |S|) \times (1 - q) \times IO \\ & + (|R| + |S|) \times N_{tuples} \times (1 - q) \times I_{hash} \\ & + |R| \times N_{tuples} \times I_{copy} \times Size_{tuple} \\ & + |S| \times N_{tuples} \times F_{hash} \times I_{probe} \qquad (8) \end{aligned}$$

Because the costs derived in [Sha86] were used for comparing alternative join algorithms, they did not include the I/O costs for reading the base relations (which were the same for all join algorithms). To reflect the total cost of the join, we add this additional cost, $(|R| + |S|) \times IO$, to the above formula. Also, to be precise, we compute IO to be the arithmetic mean of the following two quantities:

$$IO_{Read} = \frac{T_{read} + \sqrt{\frac{1}{3}\; total\ \#\ cylinders} \times F_{seek}}{F_{pref}} \qquad (9)$$

and

$$IO_{Write} = T_{Write} + \sqrt{\frac{1}{3} total\ \#\ cylinders} \times F_{seek} \quad (10)$$

The other model that we compare our results to is the one recently developed by Haas et. al. [HCL93]. This model, which we shall refer to as the HCL model, estimates the cost of the join by computing the number of seeks (N_s), the number of (possibly multi-page) I/O's (N_{io}), and the number of page transfers (N_x), and then multiplying each of these counts by the cost of the corresponding action (T_s, T_{io} and T_x). The HCL model has input parameters for the size of the input and the output buffers. Since four pages are prefetched by the disk on every read (refer to Table 5), we set the input buffer size to five[5]. The output buffer size is one. Using the notation in Table 5, the various terms in the HCL cost model are computed as follows:

$$\begin{aligned}
T_x &= 2.52ms(T_{xfer} = 3.09Mb/s, Size_{page} = 8KB) \\
T_{io} &= \frac{T_{rot}}{2} \qquad (11) \\
T_s &= \sqrt{\frac{1}{3}\,total\ \#\ cylinders} \times F_{seek} \qquad (12) \\
N_x &= |R| + |S| + 2 \times |R'| + 2 \times |S'| \\
N_{io} &= \lceil\frac{|R|}{5}\rceil + \lceil\frac{|R'|}{1}\rceil + \lceil\frac{|S|}{5}\rceil + \lceil\frac{|S'|}{1}\rceil + B + \lceil\frac{|S'|}{5}\rceil \\
N_s &= 2 + \lceil\frac{|R'|}{1}\rceil + \lceil\frac{|S'|}{1}\rceil + 2 \times B
\end{aligned}$$

The cost of the join, $Cost_{HCL}$, is then given by

$$Cost_{HCL} = N_x \times T_x + N_{io} \times T_{io} + N_s \times T_s$$

The results of the three models for various input relation sizes are shown in Figures 11 and 12. As can be seen from equation 8, the disk cost in the Shapiro model is based solely on the number of pages transferred from the disk. The effect of interference between reads and writes is not considered. Also, a simple average I/O cost is used in the model; the model does not account for sequential I/Os. As a result, the Shapiro model over-estimates the join cost. The HCL model treats the disk costs more carefully than the Shapiro model. However, since it does not consider the effect of intra–operator contention, it ends up under–estimating the cost of the join, as shown in the figures. Note that the error in the Shapiro model is 193% at relation size 100, and that the absolute error increases for relation sizes beyond 150. Note also that the error in the HCL model is as high as 38% (at relation size 150), and that the absolute error increases (gradually) as the relation sizes increase.

The reason that the HCL model does not consider interference between reads and writes is that they assumed (for simplicity) a disk configuration in which the bucket files are on a separate disk from the base relation files (R and S). In the next experiment, we adapt our model to this configuration and again report the comparison. Here we will assume that the base files are held on a 1GB disk and that the bucket files are written to a separate 200 MB disk.

The cost formula for the Shapiro model here is similar to the one used in the previous experiment, except that we now use the total number of cylinders in the 200MB disk for computing the seek time in IO_{Write} (equation 10) and the total number of cylinders in the 1GB disk in computing IO_{Read} (equation 9); the two are then averaged, as before, to compute the average I/O cost term. Similarly, for the HCL model, the number of cylinders in the 200MB disk was used in equation 12 for computing the seek cost T_s.

[5]The HCL model does not consider disk caching precisely, but setting the input buffer size to five models the effect of reading five–page blocks.

The changes required for our own model in this case include setting $Seek_{CW}$ to represent the average seek for the writes on the 200MB disk and removing the queueing terms from the disk response time equations, yielding

$$\begin{aligned}
S_{readMiss} &= T_{Read} \\
S_{write} &= T_{Write} + 1.0 \times Seek_{CW} \\
R_{1,Disk} &= \frac{S_{readMiss}}{F_{pref}} \\
R_{2,Disk} &= S_{write}
\end{aligned}$$

The results of this final comparison are shown in Figure 13. The Shapiro model behaves in the same way as before, while the HCL model and our model match surprisingly closely. One would have expected the join times estimated by the HCL model to be lower than the join times of our model, as the HCL model does not include CPU costs. Moreover, as shown in Figure 14, which plots the CPU cost for performing a join as a percentage of the total join execution time, the CPU cost can be significant. [6] (The CPU cost that contributes to the total join execution time in our model is estimated in Figure 14 by adding the CPU costs of the slower process in each phase.) Another modeling difference, however, is that the HCL model does not consider the intra–operator parallelism that arises here with two processes (joins and scans) being simultaneously active in the system. As an example, in counting the number of I/Os (N_{io}), the HCL model adds together the I/Os for reading the building relation ($\lceil\frac{|R|}{5}\rceil$) and for writing the bucket files ($\lceil\frac{|R'|}{1}\rceil$). However, these I/Os could be occurring in parallel, as they are issued to different disks. Our model captures the intra–operator parallelism by separately accounting for the two processes (the joins and the scans) and using the process model to predict the final execution time. For the system parameters used here, the HCL model's overly high I/O cost estimate, due to not considering the intra–operator parallelism, seems to offset its lack of a CPU cost component.

5 Conclusions and Future Work

In this paper we have developed an analytical model of the execution time for the hybrid hash join algorithm in the case of a single join running stand-alone on a single node. Even this simple case required that complex behavior be accounted for, including intra-operator disk interference patterns, disk seek times that are influenced by file placement as well as interference, intra–operator synchronization, and caching of disk pages. Through comparisons with results from a detailed database system simulator, the model was shown to be highly accurate – not only in predicting overall join processing

[6]The CPU contributions, shown in Figure 14, are high for joins of small relations because the building relation can be held entirely in memory. Since the reads are sequential and not interfered with in this case, the cost of reading a page is small; the CPU time thus becomes comparable to the disk time. Beyond a size of 100K, however, the building relation is too big to fit in memory and bucket writes are incurred. Since writes take longer than reads, the contribution of the CPU cost as a percentage of the total join cost then decreases.

times, but also for more detailed performance measures such as I/O response times. The model was also shown to be more accurate than previously published join cost models.

It is interesting to note that, in the course of developing and validating the analytical model, various assumptions that were made implicitly when building the simulator were exposed and reexamined. In some cases, the simulator was modified as a result of these reflections. As one example, the simulator initially assumed that the buffer used for communication between the scan and join processes was unbounded. As another example, it assumed that tuples could span page boundaries (i.e., that a 200 byte tuple could have its first 50 bytes on one page and the remaining 150 bytes on the next page). While simulations are often used to validate analytical models, these sorts of assumptions are easily overlooked when implementing a simulator. Thus, it is important to note that analytical models can actually be quite useful for improving confidence in the validity of the simulator (rather than only the vice versa).

The model that we have developed thus far is intended primarily as a proof of concept and as a starting point for developing models for studying query scheduling and memory management strategies for multiprogrammed systems and/or parallel systems. As pointed out in the Introduction, accurate and efficient analytical models of such systems would facilitate a more complete exploration of the system design space as well as enabling the study of very large systems. As an example of how much more efficient analytical models might be, a single data point for the case of joining two 500K tuple relations (in Figure 6) required approximately 29 minutes to simulate; evaluating the analytical model took only 0.75 milliseconds for the same case. (Note that we ran the join 20 times in the simulation and then averaged the resulting join execution times.) Moreover, the system that we were simulating was relatively simple, consisting of a single node and a small amount of main memory (4096 8K pages), and the cost of simulation grows rapidly as systems become more complex.

6 Acknowledgement

We would like to thank Kurt Brown for inspiring this work, for many useful comments, and for taking time to explain various details of the simulator (ZetaSim) that was used to validate the model presented here. We would also like to thank all of the ZetaSim developers for having provided us with an excellent simulator; without it, much of this work would not have been possible.

References

[BCL93] K. P. Brown, M. J. Carey, and M. Livny. "Managing Memory to Meet Multiclass Workload Response Time Goals". In *Proc. of the 19th VLDB Conf.*, Dublin, Ireland, August 1993.

[BE77] M. W. Blasgen and K. P. Eswaran. "Storage and access in Relational Databases". *IBM Sys. Journal*, 16(4), 1977.

[BG88] D. Bitton and J. Gray. "Disk Shadowing". In *Proc. of the 14th VLDB Conf.*, Los Angeles, August 1988.

[Bra84] B. Bratbergsengen. "Hashing Methods and Relational Algebra Operations". In *Proc. of the 10th VLDB Conf.*, August 1984.

[Bro92] K. P. Brown. "PRPL: A Database Workload Specification Language, Version 1.3". Master's thesis, Computer Sciences Department, University of Wisconsin—Madison, November 1992.

[CLYY92] M. S. Chen, M. L. Lo, P. S. Yu, and H. C. Young. "Using Segmented Right–Deep Trees for the Execution of Pipelined Hash Joins". In *Proc. of the 18th VLDB Conf.*, August 1992.

[Com79] D. Comer. "The Ubiquitous B-Tree". *ACM Computing Surveys*, June 1979.

[CS91] M-C Chiang and G. S. Sohi. "Experiences With Mean Value Analysis Models for Evaluating Shared Bus Throughput-Oriented Multiprocessors". In *Proc. SIGMETRICS*, May 1991.

[DG92] D. J. DeWitt and Jim Gray. "Parallel Database Systems: The Future of Database Processing or a Passing Fad?". *Communication of the ACM*, June, 1992.

[DGS$^+$90] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. "The Gamma Database Machine Project". *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[DKO$^+$84] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. "Implementation Techniques for Main Memory Database Systems". In *Proc. SIGMOD*, June 1984.

[Gra93] G. Graefe. "Query Evaluation Techniques for Large Databases". *ACM Computing Surveys*, 25(2), June 1993.

[HCL93] L. Haas, M. J. Carey, and M. Livny. "SEEKing the Truth About Ad Hoc Join Costs". Technical Report 1148, Computer Sciences Department, University of Wisconsin—Madison, May 1993.

[HT83] P. Heidelberger and K. S. Trivedi. "Analytic Queueing Models for Programs with Internal Concurrency". In *IEEE Transactions on Computers*, January 1983.

[KNT89] M. Kitsuregawa, M. Nakayama, and M. Takagi. "The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method". In *Proc. of the 15th VLDB Conf.*, Amsterdam, The Netherlands, August 1989.

[LY90] M. S. Lakshmi and P. S. Yu. "Effectiveness of Parallel Joins". In *IEEE Transactions on Knowledge and Data Engineering*, December 1990.

[LZGS84] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *"Quantitative System Performance"*. Prentice Hall Inc., Englewood Cliffs, New Jersey, 1984.

[MD] M. Mehta and D. J. DeWitt. "Data Placement in Shared–Nothing Parallel Database Systems". Submitted for publication.

[MD93] M. Mehta and D. J. DeWitt. "Dynamic Memory Allocation for Multiple–Query Workloads". In *Proc. of the 19th VLDB Conf.*, August 1993.

[ME92] P. Mishra and M. Eich. "Join Processing in Relation Databases". In *Communication of the ACM*, March 1992.

[RL80] M. Reiser and S. Lavenberg. "Mean Value Analysis of Closed Multichain Queuing Networks". In *Journal of ACM*, volume 27, April 1980.

[Sha86] L. Shapiro. "Join Processing In Database Systems with Large Main Memories". *ACM TODS*, 11(3), September 1986.

[VLZ88] M. K. Vernon, E. D. Lazowska, and J. Zahorjan. "An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache-Consistency Protocols". In *15th Annual Int'l. Symp. on Computer Architecture*, Honolulu, Hawaii, May30-June 2 1988.

[WE90] D. Willick and D. Eager. "An Analytical Model for Multistage Interconnection Networks". In *Proc. SIGMETRICS*, May 1990.