SEEKing the Truth About *Ad Hoc* Join Costs

Laura Haas
Michael J. Carey
Miron Livny

Technical Report #1148

May 1993

SEEKing the Truth about *Ad Hoc* Join Costs

Laura M. Haas[*] Michael J. Carey[†] Miron Livny[†]

Abstract

In this paper, we reexamine the results of prior work on methods for computing *ad hoc* joins. We develop a detailed cost model for predicting join algorithm performance, and we use the model to develop cost formulas for the major *ad hoc* join methods found in the relational database literature. We show that various pieces of "common wisdom" about join algorithm performance fail to hold up when analyzed carefully, and we use our detailed cost model to derive optimal buffer allocation schemes for each of the join methods examined here. We show that optimizing their buffer allocations can lead to large performance improvements, e.g., as much as a 400% improvement in some cases. The results of this work should be directly useful to implementors of relational query optimizers and query processing systems.

1 Introduction

The join of two sets of tuples is a fundamental operation for relational database system, and many algorithms have been proposed to compute joins [1, 2, 5, 10, 15, 17]. Some join algorithms exploit pre-computed access structures, such as B-trees or join indices [18]. These algorithms are ideal for pre-meditated joins that will be done repeatedly. However, in a decision support environment, not all joins can be anticipated. Thus, an important subclass of join algorithms are those intended to handle *ad hoc* joins; Bratbergsengen [2] aptly refers to these as algorithms of "last resort".

A query optimizer chooses the algorithm to be used for a particular join, using a cost model to compare alternatives. An important component of the cost model is the I/O cost model. In [4], we argued that when some or all of the data is stored on a tertiary device, the optimizer must have a detailed I/O cost model that takes into account the various characteristics of each of the devices that hold data referenced by the query. Using such a model, we began to profile the I/O cost of a number of standard *ad hoc* join methods, modified to take their operands from tape. When the results did not agree with our intuition as to how those algorithms should compare, we applied our detailed model to the disk-based versions of these algorithms. Still the results did not agree well with those we had seen in the literature [2, 5, 9, 17]. When we revisited these papers, we realized that they all used simpler I/O cost models than ours. For example, the most common model assumes that the I/O cost of a join is simply proportional to the number of pages read and written, ignoring latency and seek costs. (These papers and their cost models will be discussed in Section 2).

The goal of the current paper, therefore, is two-fold: to demonstrate the importance of a detailed I/O cost model, and to share with the community some of the predictions of that model. We argue that a query optimizer *must* employ a detailed model, including latency and seek costs as well as page transfer costs. This means that the optimizer must understand the details of each join method in its repertoire, as well as how

[*] IBM Almaden Research Center, K55/801, San Jose, CA 95120

[†] Computer Science Dept., University of Wisconsin, Madison, WI 53706

the I/O hardware works. While the hardware determines the cost of each step of an I/O operation (seek, latency, page transfer), the join method determines when these steps are needed, and how many of each are needed. For example, the join method determines how many seeks are required, and how many pages can be transferred during a single I/O. The results of our study also demonstrate the importance of understanding how each join method uses memory (for example, the amount of space that it allocates for input buffers), as the amount of memory available for buffering affects the number of I/Os and the number of pages that must be transferred.

Our study focuses on algorithms for handling *ad hoc* joins, including Nested Block Join, Sort-Merge Join, Simple Hash Join, Grace Hash Join and Hybrid Hash Join. (For an overview of these and other join methods, see [15]). These represent the major "last resort" join methods in the literature. We will use this collection to show how a simpler I/O cost model can easily lead the optimizer to select the wrong join method for a particular join.

The remainder of this paper is organized as follows. In Section 2, we briefly review the relevant literature on join algorithms and their performance. Then we present a detailed I/O cost model that we advocate for use in future query optimizers (Section 3), and we derive I/O cost formulas for each of the *ad hoc* join methods that we consider (Section 4). In Section 5 we put the model to work in a series of analytical "experiments". Using the model, we show that various pieces of "common wisdom" are simply not true when you take the detailed I/O costs into account; for example, Hybrid Hash Join is not always the best method for *ad hoc* joins, nor is it best to divide memory equally between the two relations in a Nested Block Join. We show how important multi-block I/Os can be if buffer pages are carefully allocated to support these I/Os. We then use the model to derive optimal buffer allocations for each of the algorithms addressed here. These formulas can be easily incorporated into a query optimizer so that it can instruct the runtime system how best to use memory for a given join operation, and so that it can compute the implied cost of this memory allocation. Finally, we present our conclusions from this work (Section 6).

2 Related Work

Many papers have been written on join algorithms. We do not attempt a survey (see [15]), but focus only on directly related papers here, with an emphasis on the I/O cost models that they used.

Perhaps the "granddaddy" of all join papers was the work of Blasgen and Eswaran [1]. This paper, done in the context of the System R project at IBM, derived costs for select/project/join queries evaluated using four join algorithms, and compared them under various "typical" scenarios. The I/O cost model used for the analyses counted page transfers, and multiplied them by the page transfer cost. The results contributed to the development of the System R approach to query optimization [16]. The algorithms introduced in this paper include two that have come to be known as Nested Loops (from which Nested Block Join [10] is descended), and Sort-Merge Join. Since this is one of the earliest papers on join methods, it did not include other important join algorithms, such as hash-based algorithms, and it only considered very small memories (by today's standards). In addition, the scenarios tested all assumed indexes on join attributes, so the paper presents no results for the *ad hoc* case covered here.

In [13], Mackert and Lohman explored the accuracy of the R* optimizer's cost predictions and algorithm choices. R* was a distributed relational DBMS, but its optimizer was a direct descendant of the System R optimizer, and again used an I/O cost model based on the number of pages transferred. The results presented in this paper for Sort-Merge versus Nested Loop Join with an index show the importance of sequential I/Os in reducing the cost of a join, as well as the importance of modeling buffer allocation among the tables and indexes involved in the join. The R* optimizer consistently overestimated I/O costs because it ignored sequential I/Os, and it had trouble capturing competition between indexes and table scans for buffer pages.

Algorithms for Grace Hash Join, Simple Hash Join and Hybrid Hash Join were introduced in [5, 17]. Cost models of these algorithms and Sort-Merge join were developed and compared in a range of memory sizes that allowed Sort-Merge to run with a single merge pass. Again, a very simple I/O cost model, counting only page transfers, was used. These papers popularized hashing as a technique for join processing, and their results led to a fairly widespread belief in the superiority of hashing techniques (especially Hybrid Hash) for *ad hoc* joins. These papers did not include any form of nested loop join in their comparisons.

Another key reference on the use of hashing for processing relational operations, especially *ad hoc* joins, is [2]. This paper, done independently of [5, 17], included algorithms, analyses, and comparisons of Nested Block join, Sort-Merge join, and a Grace-like hash join. Again, the cost model was based on transfer costs, but the paper included a discussion of using multi-page data transfers to lower I/O costs. As in [5] and [17], the main conclusion from the paper is that hashing is an important technique for relational algebra operations.

In [9], Hagmann argued that the number of I/O requests, not the number of pages transferred, should serve as the main cost metric for an optimizer. Having stated his case, he proceeded to use his new cost model to derive some interesting results. Of particular interest here, he re-examined the question of buffer allocation for Nested Block Join, concluding that the buffers should be split evenly between the two relations in order to minimize cost. Hagmann also considered buffer allocation for hash joins as in [2], and derived an optimal allocation using his cost metric (assuming a fixed number of buckets). While this paper clearly showed that the I/O cost model does affect predictions, it made no attempt to show that the metric proposed was "correct".

The main focus of [6] was to explore the dualities and differences between sort-based and hash-based join methods. The paper presented interesting discussions of these dualities, and several possible optimizations to the algorithms as a result. Included were experimental results using the Volcano system that showed that hashing was generally superior to sorting except when data was highly skewed, and was much better for operands of different sizes. One section of the paper showed that increasing the "cluster size" (the unit of I/O) could dramatically improve performance results for both hashing and sorting; the paper concluded that a cluster size of 32 KB (eight pages) worked well, and thereafter used that size for other experiments. No attempt was made to compute an optimal cluster size.

From this set of papers we get an interesting collection of methods for handling *ad hoc* joins, but, with the exception of [9], most of the work has been based on a simple model of I/O costs that counts only the number of pages transferred. Hagmann [9] argued for a different, but equally simplistic model, *i.e.*, counting only the number of I/O requests. The predominant transfer-only model has been used for most comparisons of these algorithms, resulting in a certain set of beliefs about their relative merits. While several papers [9, 6, 2] recognize the importance of multi-page I/Os, none have thoroughly studied their implications.

In this paper, we will analyze all five of the key methods for *ad hoc* joins with a more detailed (hence, more realistic) I/O cost model to produce better cost equations for a query optimizer. We will show that the detailed I/O cost model would lead an optimizer to very different conclusions that the simple models used in the literature. Given the importance of multi-page I/Os, we will also use these cost equations to compute how best to divide up memory for each of the join methods.

3 A Detailed I/O Cost Model

In this section, we describe the I/O cost model that we will use to study the various join algorithms. We assume the I/O system works as follows: when data must be read or written to disk, a target location on disk is identified. There are three steps in the I/O operation: a seek, if necessary, to move the disk head to the desired cylinder, the latency, during which the disk spins until the desired data is underneath the head, and finally, the transfer, during which one or more pages of data are moved between disk and memory.

The model assumes a fixed cost for each of these steps. The number of times each step occurs, however, is algorithm dependent. Thus, the total I/O cost, C_T, of an algorithm is equal to the sum of the three component costs: seek cost, latency cost, and page transfer cost. Each of these costs is in turn the product of the (algorithm dependent) number of actions (N_S, $N_{I/O}$, and N_X, respectively) multiplied by the (algorithm-independent) time that action consumes. Note that since latency is accrued for each disk I/O, the latency cost is equal to $N_{I/O}$ times the average latency. In other words,

$$C_T = N_S \times T_S + N_{I/O} \times T_L + N_X \times T_X$$

The values we use for T_S, T_L, and T_X approximate those observed for the Fujitsu Model M2266 disk drive, as described in [3] (see Table 1 for values). The model does not include I/Os to write the result of the join; these will be the same for all algorithms.

Though we model seek time as a constant, an approximation of the "average" seek time, seek time is actually a function of the number of cylinders traversed. However, it is not normally practical to compute this number, as it will be a function of how data is laid out on the disk(s). For example, suppose we wish to join relations R and S. These relations could be back to back on the same disk, or many cylinders apart on the same disk, or even on separate disks. Both the number of seeks and the number of cylinders traversed will depend on where the relations are located. While a query optimizer might know, and hence be able to model correctly, the locations of any base relations being joined, it is unlikely to be able to predict this for any temporary, computed relations, including any produced while doing the join. Thus any attempt to count cylinders as part of optimization will likely be inaccurate and potentially misleading. Using an "average" seek time avoids this difficulty, and provides a close approximation of expected behavior, as discussed below.

To derive the number of seeks, the model assumes that the base relations for a join are stored on a single disk, while temporaries are stored on a separate disk. Thus, reads of the base relations will interfere with each other, in the sense that reading from one of the base relations after reading from the other will typically cause a seek. On the other hand, reads of the base relations do not interfere with writes (or reads) of temporaries, that is, the model assumes that the disk arm remains positioned on the correct cylinder for the base relation while a temporary is written to the other disk. The model ignores interference by other processes that could be using the disks in a multiuser system and only counts seeks that must be done with this layout because of the join algorithm being used. While an optimizer could reasonably capture this level of detail, it could hardly be expected to model interference by other processes.

We believe that the seek-related assumptions that we have made in this model are reasonable, and do not unduly affect the results we report. To test this hypothesis, we implemented a more detailed, cylinder based model of seeks, and compared the costs predicted by that model with those predicted by the average seek time model, for two different layouts of data. In one layout we assumed that relations R and S are stored next to each other on one disk, and any temporary results from the join (e.g., hash buckets for a hash join) are stored contiguously on a second. In the other layout, R, S, and any temporaries are all stored on separate disks. In this configuration there is less seeking, as separate positions can be held on each disk. For each layout, we generated two sets of numbers: one, by counting the number of seeks and multiplying by a constant seek cost (9.5 ms), and the other, by computing the distance (in cylinders) traveled in each seek and applying Gray's formula [8] for converting distance to seconds. The different predictions are very close. For Hybrid Hash (see Section 4), the algorithm among those studied here that is most sensitive to seek cost, the differences between layouts are imperceptible (less than .3%), while predictions based on the seek cost

Parameter	**Meaning**	**Value(s)**
P	Page Size	8 KB
T_S	Average Seek Time	9.5 msec
T_L	Latency	8.3 msec
T_X	Page Transfer Time	2.6 msec
F	Universal Fudge Factor	1.2
$\|R\|$	Size of R in pages	1250 pages (10 MB)
$\|S\|$	Size of S in pages	$1 \times \|R\|$, $10 \times \|R\|$
M	Memory size in pages	62 - 1625 pages (.35 - 13 MB)

Table 1: Parameters of the Model

constant were about 2.5% higher than those based on the seek cost formula, over a wide range of memory sizes. Results for the other algorithms that we studied are even better. It might be possible to get even closer by playing with the value of the average seek cost. We did not feel that this was important, as even the distance-based model is an estimate of the real seek costs, relying on assumptions about data placement that might not hold in a real system. For example, if the two relations are further apart on the disk in the first layout, seek costs will be higher.

For the latency component, the model approximates the cost of the latency as half the disk rotation time. Finally, the third cost component, the transfer cost, is computed using the transfer rate for the device. We assume that our system is capable of *blocked I/O*; that is, it can transfer multiple pages (reading or writing) per disk I/O. In reality, a disk can transfer only a certain number of pages before being forced to seek (move to the next cylinder). In our model, we ignore these tiny seeks, as the cost of a seek is very small compared to the cost of transferring an entire cylinder of data (9.5 ms versus 2.2 seconds with the parameter values of Table 1). These seeks would not add significantly to the cost predictions.

4 Counting the Steps

In this section, we present equations for the number of disk I/Os, number of seeks, and number of pages transferred by each of the five algorithms studied in this paper. Each algorithm joins two relations, R and S, and without loss of generality, we assume that $|R| \leq |S|$. We assume a uniform distribution of key values for R; this allows us to ignore hash bucket overflow for the hashing algorithms. Each of the algorithms requires some space in memory for additional structures (e.g., a hash table) roughly proportional to the size of the data being processed. We approximate this extra space using the universal fudge factor, F [5].

The next five subsections derive the counts for each of these classic algorithms: Nested Block Join, Sort-Merge Join, Simple Hash Join, Grace Hash Join, and Hybrid Hash Join.

4.1 Nested Block Join

The Nested Block Join algorithm, NBJ, divides memory into two parts. M_R pages are used for relation R, $M_S = M - M_R$ pages for S. The smaller relation, R, is read from disk in chunks of size M_R/F. This guarantees sufficient memory to build a hash table in memory for the chunk. For each chunk of R that is read, *all* of S is read in pieces of size M_S, and the join is performed by probing the hash table for the R chunk with S tuples.

Let NB be the number of chunks needed to read all of R. The number of pages transferred by the NBJ algorithm is:

$$N_X = |R| + (NB \times |S|)$$

where

$$NB = \left\lceil \frac{|R|F}{M_R} \right\rceil$$

The number of disk I/Os for NBJ is given by:

$$N_{I/O} = NB \times \left(1 + \left\lceil \frac{|S|}{M_S} \right\rceil\right)$$

In other words, for each chunk of R, we need to start reading this chunk, and then read S in ($|S|/M_S$) I/Os.

Since R and S are assumed to be on the same disk(Section 3), a seek to the beginning of S must be performed after each read of R and another is needed after the read of S, back to the next chunk of R. Since there are NB chunks of R, there will be

$$N_S = 2 \times NB$$

seeks for NBJ.

4.2 Sort-Merge Join

The Sort-Merge join method has two phases. In the first phase, each relation is sorted into runs, and in the second phase the runs from both relations are merged together. We assume $M > \sqrt{F|S|}$. This guarantees one merge pass. With less memory, Sort-Merge join will require multiple merge passes in the second phase. Each relation is read twice, and written, in the form of sorted runs, once. Thus,

$$N_X = 3|R| + 3|S|$$

The number of disk I/Os is computed by dividing the amount being read or written by the number of pages being read or written at a time.

$$N_{I/O} = \left\lceil \frac{|R|}{I} \right\rceil + \left\lceil \frac{|R|}{O} \right\rceil + \left\lceil \frac{|S|}{I} \right\rceil + \left\lceil \frac{|S|}{O} \right\rceil + \left\lceil \frac{|R|}{MPR} \right\rceil + \left\lceil \frac{|S|}{MPR} \right\rceil$$

During the first phase, memory is divided into three sections: an input buffer of size I, an output buffer of size O, and working space to build a tournament tree for the sort, of size $WS = M - I - O$. In the second phase memory is divided evenly between the runs of R and S, with MPR pages per run. The first four terms of the equation for $N_{I/O}$ account for the reads and writes during the first phase. During this phase, the relation being sorted is read in in chunks of size I, and copied from the input buffer into the tournament tree. Output runs are built up in the output buffer, and written to disk whenever the output buffer is full. Each run will be, on average,

$$RL = \left\lceil \frac{2 \times WS}{F} \right\rceil$$

pages long [11] and will be written to disk contiguously. The last two terms represent the phase two reads of these runs in chunks of size MPR, where MPR is the number of pages of memory divided by the sum of the number of runs of R (NR_R) and the number of runs of S (NR_S). Thus,

$$MPR = \frac{M}{NR_R + NR_S}$$

while,

$$NR_R = \left\lceil \frac{|R|}{RL} \right\rceil$$

and

$$NR_S = \left\lceil \frac{|S|}{RL} \right\rceil$$

Finally, we can count the number of seeks for Sort-Merge Join. The phase one reads of the relations and the writes are sequential, i.e., they incur only initial seeks (four in all). The phase two reads are not sequential, however, as only MPR pages of each run can be read at a time, causing a seek to be incurred from one run to the next for each of these reads. This gives us

$$N_S = 4 + \left\lceil \frac{|R|}{MPR} \right\rceil + \left\lceil \frac{|S|}{MPR} \right\rceil$$

4.3 Simple Hash Join

In Simple Hash Join, relation R is read and reduced repeatedly, as follows. Each time R is read, a hash function is applied. Records matching the function are inserted into an in-memory hash table. Those not matching are written back to disk (producing a reduced version of R for input to the next iteration). Relation S is then read, the hash table is probed, and matches are returned. Records that do not match are written back to disk, creating a reduced version of S for the next iteration.

For this join method, memory is again divided into three parts, the input buffer of size I, the output buffer of size O, and the working space for the hash bucket of size $WS = M - I - O$. The number of iterations, NI, is given by

$$NI = \left\lceil \frac{|R|F}{WS} \right\rceil$$

The number of pages of R kept in memory on each iteration is K_R, where

$$K_R = \left\lfloor \frac{WS}{F} \right\rfloor$$

and the number of pages of S that match the in-memory portion of R on each iteration (and hence are not written back out) is on average

$$K_S = \left\lfloor \frac{|S|K_R}{|R|} \right\rfloor$$

Then the number of transfers for the simple hash join is:

$$\begin{aligned} N_X \quad = \quad & NI \times |R| - \sum_{i=1}^{NI}(i-1)K_R + \\ & (NI-1) \times |R| - \sum_{i=1}^{NI-1} i \times K_R + \\ & NI \times |S| - \sum_{i=1}^{NI}(i-1)K_S + \\ & (NI-1) \times |S| - \sum_{i=1}^{NI-1} i \times K_S \end{aligned}$$

The first two terms give the number of pages of R read. On each iteration we read K_R pages less than the iteration before. The next two terms show that we write R $(NI - 1)$ times, again omitting K_R more pages each time. The last four terms correspond to the same counts of reads and writes of S. After simplification,

$$N_X = (2 \times NI - 1) \times (|R| + |S|) - NI \times (NI - 1) \times (K_R + K_S)$$

Similar reasoning leads to

$$\begin{aligned} N_{I/O} &= \frac{1}{I}\left(NI \times (|R|+|S|) - \frac{1}{2}NI \times (NI-1) \times (K_R + K_S)\right) + \\ & \frac{1}{O}\left((NI-1) \times (|R|+|S|) - \frac{1}{2}NI \times (NI-1) \times (K_R + K_S)\right) \end{aligned}$$

Finally, we assume that the reduced versions of R and S are written to a disk other than the one that they are read from on each iteration, so that reads do not conflict with writes. The seeks for Simple Hash Join are therefore only those necessary to get to the beginning of each relation (source R, source S, and reduced R, reduced S) on each iteration. Since we read on each iteration, and write on all but the last iteration,

$$N_S = 2 \times NI + 2 \times (NI - 1) = 4 \times NI - 2$$

4.4 Grace Hash Join

In the Grace Hash join method there are two phases. In the first phase, each relation is read, and hashed into buckets which are written to disk. The number of buckets, B, is determined by the size of the smaller relation,

$$B = \left\lceil \frac{|R|F}{M - I_1} \right\rceil$$

where I_1 is the number of pages reserved for the input buffer in the first phase. Each bucket gets

$$O = \left\lfloor \frac{M - I_1}{B} \right\rfloor$$

pages of memory during the first phase. The number of buckets is chosen so that each individual bucket will fit in memory with its hash table in the second phase. We assume $M > \sqrt{F|R|}$ so that this is feasible. In the second phase, buckets of R are read into memory one at a time, a hash table is built, and then the corresponding bucket of S is read I_2 pages at a time and used to probe the hash table.

As with Sort-Merge, each relation is read twice, and written once, so the number of page transfers is

$$N_X = 3|R| + 3|S|$$

The number of disk I/Os is given by:

$$N_{I/O} = \left\lceil \frac{|R|}{I_1} \right\rceil + \left\lceil \frac{|R|}{O} \right\rceil + \left\lceil \frac{|S|}{I_1} \right\rceil + \left\lceil \frac{|S|}{O} \right\rceil + B + \left\lceil \frac{|S|}{I_2} \right\rceil$$

The terms are, in order, the number of I/Os needed to read R in the first phase, to write R in the first phase, to read S in the first phase, to write S in the first phase and finally, to read R in the second phase, and to read S in the second phase.

The number of seeks is

$$N_S = 2 + \left\lceil \frac{|R|}{O} \right\rceil + \left\lceil \frac{|S|}{O} \right\rceil + 2B$$

The first term reflects the initial seeks to the beginnings of R and S. The next two terms account for the random I/O while writing buckets (each write incurs a seek). Then, since all the buckets are on the same disk, the method has to seek back and forth between buckets of R and S in the second phase.

4.5 Hybrid Hash Join

Our last algorithm is Hybrid Hash Join. This algorithm is designed to combine the best behavior of Simple Hash and Grace Hash. As in Grace, the algorithm has two phases, assuming again that $M > \sqrt{F|R|}$. In the first, the relations are read, hashed into buckets, and written out, as in Grace. However, during this phase, a portion of the memory is reserved for an in-memory hash bucket for R. This bucket of R will never be written to disk. Further, as S is read and hashed, tuples of S matching with this in-memory bucket can be output immediately; they need not be written to disk either. The second phase proceeds analogously to Grace's: the buckets of R on disk are read in one at a time, the matching buckets of S are read in I_2 bytes at a time, and the join is performed.

Memory in the first phase is divided into three pieces, one of size I_1 for input, one of size $K * O$ for output buffers, where K is the number of buckets excluding the in-memory bucket and O is the number of pages allocated to buffering each of these buckets, and one of size $WS = M - K \times O - I_1$ for the in-memory bucket. The optimal K, given I_1, I_2 and O is the smallest K for which

$$K \times (M - I_2) + WS \geq |R|F$$

The first term corresponds to the amount of data stored in the K buckets (each bucket can be as big as $(M - I_2)$ pages), the second to the amount kept in memory. Substituting in the definition of WS and solving, we get

$$K = \left\lceil \frac{|R|F - (M - I_1)}{M - I_2 - O} \right\rceil$$

The size of the in-memory portion of R will be

$$|R_0| = \left\lfloor \frac{WS}{F} \right\rfloor$$

and the size of R', the relation obtained from R after extracting that bucket, will be $|R| - |R_0|$. We assume that S is reduced proportionately, so that

$$|S'| = \left\lceil |S| \times (1 - \frac{|R_0|}{|R|}) \right\rceil$$

The number of page transfers for Hybrid Hash is then

$$N_X = |R| + |S| + 2|R'| + 2|S'|$$

The number of disk I/Os will be

$$N_{I/O} = \left\lceil \frac{|R|}{I_1} \right\rceil + \left\lceil \frac{|R'|}{O} \right\rceil + \left\lceil \frac{|S|}{I_1} \right\rceil + \left\lceil \frac{|S'|}{O} \right\rceil + K + \left\lceil \frac{|S'|}{I_2} \right\rceil$$

Again, the terms reflect the cost to read the full relation R in phase 1, to write the reduced relation R, to read the full S, to write the reduced S, and to read both reduced relations in phase 2.

The number of seeks is analogous to that in Grace, namely,

$$N_S = 2 + \left\lceil \frac{|R'|}{O} \right\rceil + \left\lceil \frac{|S'|}{O} \right\rceil + 2K$$

5 Results

In this section we will look at the results of a variety of analytical "experiments" using the I/O cost model described above. We first explore the ramifications of our detailed I/O cost model, and compare its predictions to those made by earlier models. We then consider the affect of buffer allocations on performance, and use the cost model to derive formulas for "optimal" buffer allocations for each of the algorithms. Finally, we consider whether the detailed model is really necessary and discuss our conclusions from these experiments.

5.1 Debunking the "Common Wisdom"

Our first experiment compares the predictions of our model with the results predicted by the earlier, transfer only, model described in section 2. For this transfer only model, $C_T = C_X = N_X \times T_X$, as $T_L = T_S = 0$ under the assumptions of this model. For purposes of this experiment, we set the buffer sizes I, I_1, I_2 and O to one disk page, consistent with typical formulations in the literature. For NBJ, we use two different values for M_S, based on two different proposals in the literature. The first uses one disk page for S ($M_S = 1$); the second uses half the available memory for S ($M_S = M/2$), per [9]. We call these two versions of the algorithm NBJ_1 and NBJ_{50}, respectively.

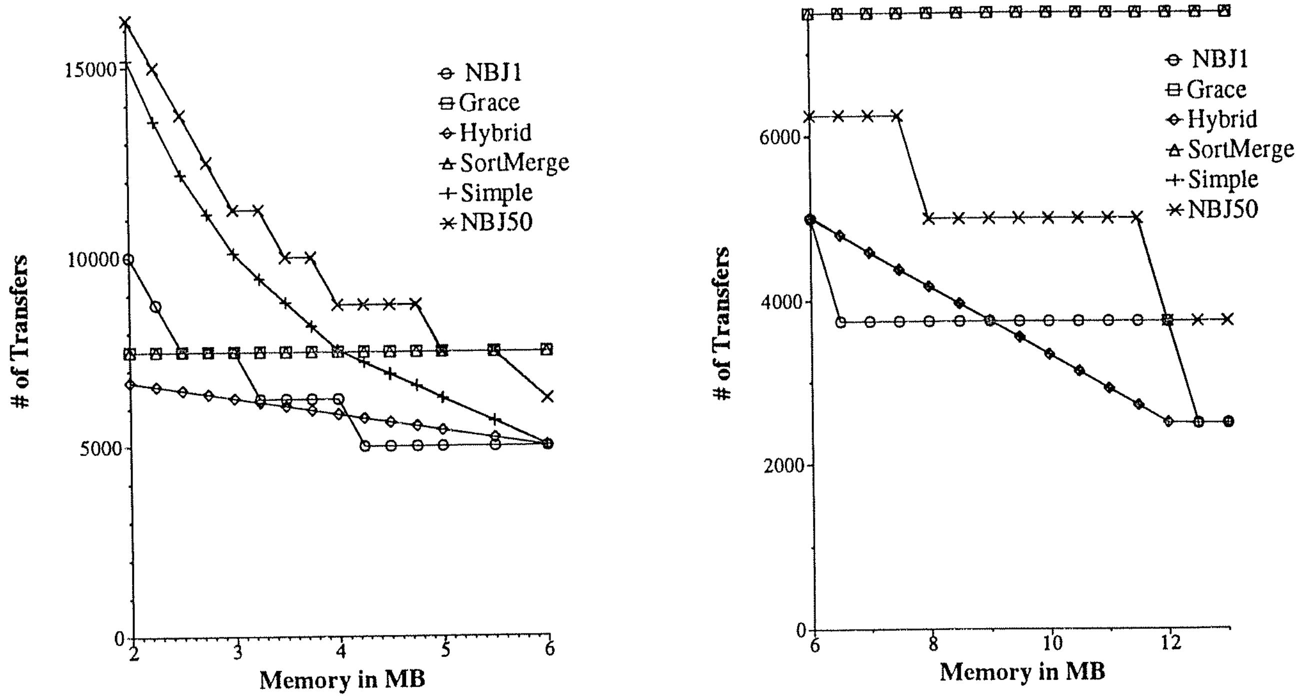
We computed N_X for each of these six algorithms for memory sizes ranging from .35MB to 13MB, using the parameter values given in Table 1. We use N_X instead of C_X because N_X is independent of the value of T_X, hence more general (and since T_X is a constant, the shapes of the curves are identical). The lower memory bound (.35 MB) is slightly more than the minimum memory required to make Hybrid, Grace and Sort-Merge run as two phase algorithms. At the upper end of the range, $M > |R|F$, or, in other words, R fits in memory. In Figure 1 we show the number of transfers, N_X, for $M = 2$ to 13 MB, where $|S| = |R| = 10$ MB.

The graphs in Figure 1 predict that Hybrid will outperform the other algorithms for much of the memory range. Its only rival is NBJ_1, which, when $M > 4$ MB (when R can be read in three chunks or less), will at times outperform Hybrid. As expected, Simple Hash is terrible when memory is small, but improves to rival Hybrid once memory reaches about 6 MB. The worst in small memory is NBJ_{50}; in larger memory sizes, though, it too drops below Grace and SortMerge, each of which transfer a constant amount of data. For smaller memories the trends at the left edge of Figure 1(a) continue, with NBJ_1, NBJ_{50} and Simple Hash getting rapidly worse, while Grace, Hybrid and Sort-Merge are stable, with Hybrid being slightly better than the other two. Results for $|S| = 10 \times |R|$ (not shown) are qualitatively similar.

We computed C_T using our detailed I/O cost model over the same parameters, with the same values for the buffer sizes. The results are shown in Figure 2. The y-axis in these figures is in seconds, computed using the weights T_X, T_S and T_L from Table 1.

The more detailed model yields different results. In fact, the results are strikingly different. With the detailed model, NBJ_{50} is the winner over the whole range of memory sizes shown in Figure 2. Also surprising relative to the common wisdom, Grace and SortMerge both out-perform Hybrid until memory becomes very large (5 MB for SortMerge, and 7 MB for Grace). Hybrid is never better than fourth choice among the algorithms over this range of memory sizes. Moreover, the differences are significant; Hybrid is two to three times worse than NBJ_{50}, and as much as 50% worse than Grace. The reason for these results is that Hybrid, with these buffer allocations, uses most of its phase one memory for bucket R_0, reading and writing the rest of R in many tiny pieces, thus requiring many disk I/Os. Grace, on the other hand, divides phase one memory over all the buckets, so while its transfer cost is constant, it does those transfers very efficiently in terms of the number of disk I/Os required. Grace's cost is still dominated by transfer time, which is why it appears constant at this resolution (it is actually declining slightly with increasing memory). NBJ_{50} provides the best balance between pages transferred and I/Os, hence its strong showing.

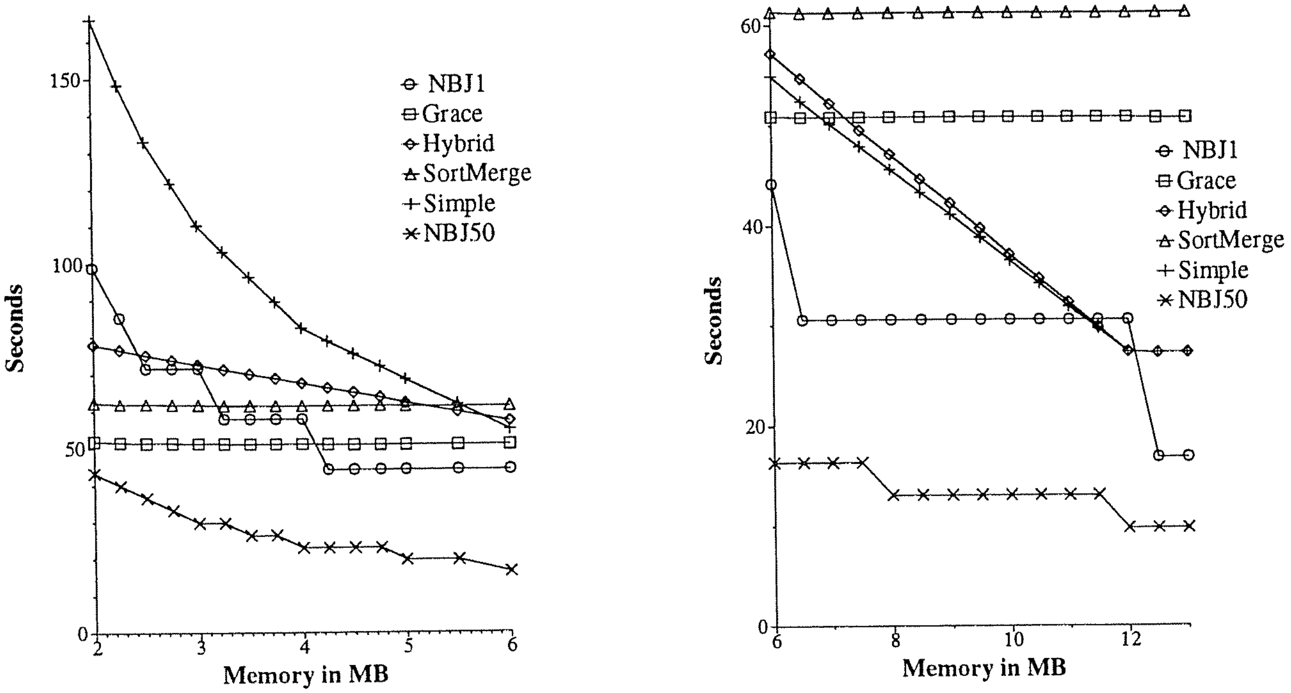
For memories under 2 MB (not shown), NBJ_{50}'s performance degrades, falling to fourth place for $M <$ 1.5 MB. Although it continues to do an excellent job of reducing the number of disk I/Os, the transfer costs sky-rocket in this range (because the small size of M_R forces NB up). Hybrid never climbs above third



(a) Small memory

(b) Large memory

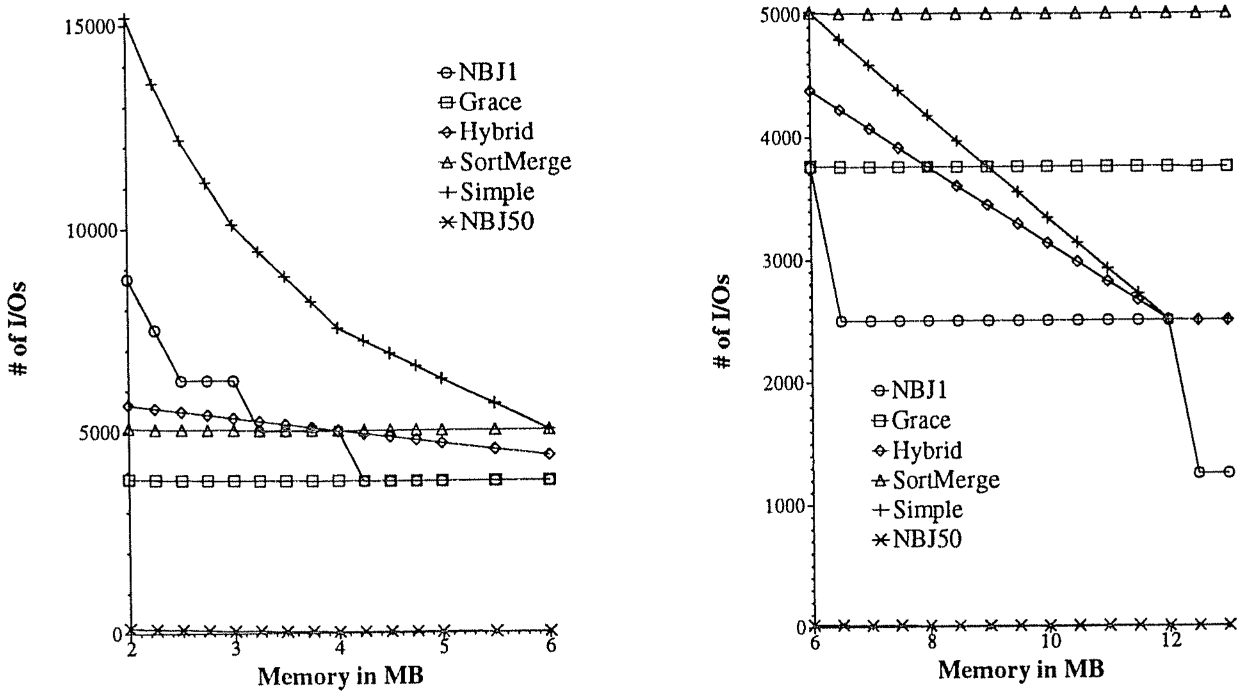
Figure 1: Predictions of the transfer-only model



(a) Small memory

(b) Large memory

Figure 2: Predictions of the detailed model



(a) Small memory

(b) Large memory

Figure 3: Predictions obtained by counting I/Os only

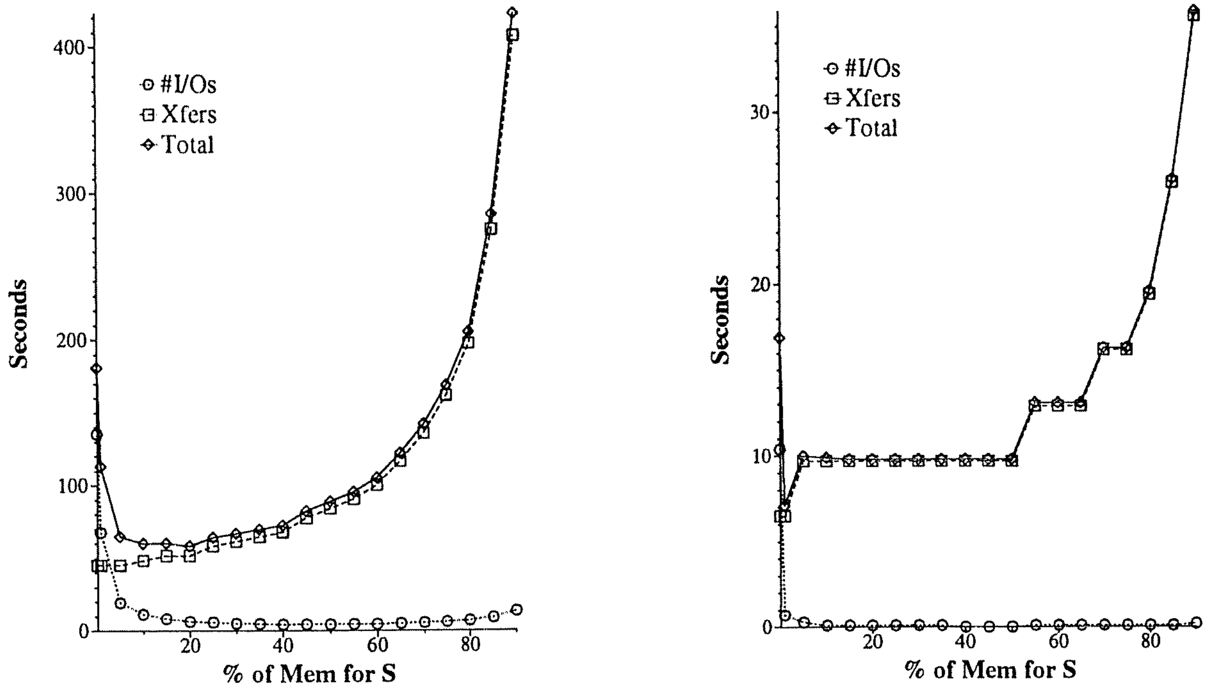
place, even when $M < 1$ MB, though it does grow closer to Grace. As memory gets smaller, Grace needs more buckets, and each bucket gets less space; hence it cannot reduce disk I/Os as much. On the other hand, Hybrid also needs more buckets, and has little memory left for reducing transfers.

What do we learn from this exercise? First, that the "common wisdom" on join algorithms, based on analyses that only count page reads and writes, is potentially misleading. This is because, as Hagmann [9] asserts, latency is an important component of I/O costs. In fact, if we had modeled latency costs only (or equivalently, only counted the I/Os), as Hagmann suggests, we would have been less surprised by the results of the detailed model. Figure 3 shows the predictions obtained by counting disk I/Os only.

It is clear from comparing the very different predictions (under our detailed model) for NBJ_1 and NBJ_{50} that buffer allocations play an important role in reducing the number of I/Os that are needed to perform a join, and hence, reducing the cost of the join. If latency is an important cost factor, then it stands to reason that by selecting the "right" buffer allocations, we may be able to improve the performance of each of these algorithms. In the next two subsections we will use the detailed I/O cost model to first explore the effect of buffer allocation on the performance of the various algorithms, and then derive the "optimal" allocation for each algorithm.

5.2 Exploring Buffer Allocation

It has been observed [6] that increasing the size of the input and output buffers can improve the performance of Hybrid Hash and Sort-Merge joins. In fact, for all five join methods, increasing the amount of memory dedicated to I/O will improve performance – to a point. After a while there comes a point of diminishing returns, followed in general by increasing costs. The difference between performance at the "optimal" I/O buffer size and a more naive buffer allocation can be significant – over 300% in some cases. Beyond these



(a) M = 1 MB

(b) M = 12.5 MB

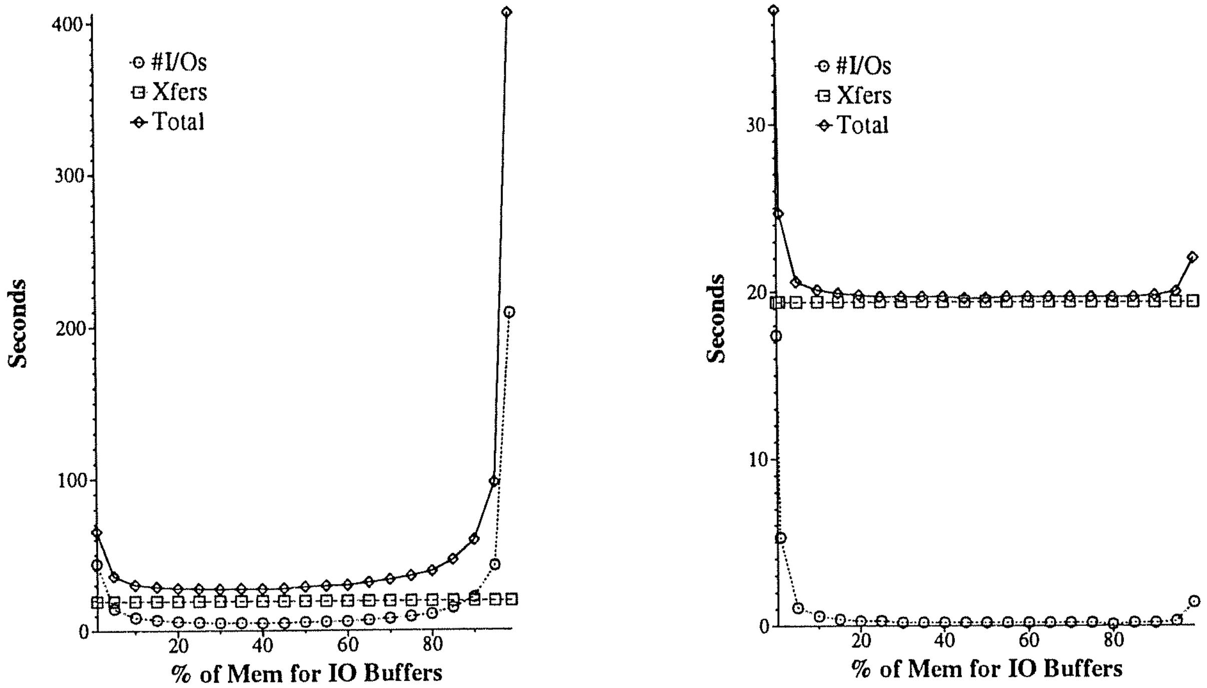
Figure 4: Varying % of memory given to S for NBJ

general observations, the algorithms behave quite differently, so we discuss each individually below.

In Figure 4 we show the result of varying the percentage of memory dedicated to buffering the inner relation, S, for the NBJ algorithm. We show the results for two different memory sizes, a small memory of 1 MB, and a large memory of 12.5 MB. The model used is our detailed model, counting I/Os, seeks, and page transfers. The leftmost point on each graph corresponds to the allocation of a single page for S, as in the NBJ_1 algorithm; at $X = 50$ half of memory is allocated to S, as is done for NBJ_{50}. The minimum predicted by the detailed model lies between these two points in each case. Each graph shows, in addition to the total cost, C_T, the component costs for latency ($N_{I/O} \times T_L$) and for transfers ($N_X \times T_X$). When the percentage of memory given to S is small, the number of I/Os is high; as more memory is given to S, the number of I/Os falls off quickly, but the number of chunks of R, NB, increases; the total cost then begins to be dominated by the cost of transferring increasing amounts of data, as the amount of data transferred is proportional to NB. Finally, note that for the two different memory sizes, the optimal value of M_S is different. For the smaller memory, the minimum occurs at around 20%, or 25 pages. For the larger memory, the minimum occurs at around 4%, or 63 pages (this is the largest amount of memory we can give to S and still have R fit in memory in one chunk, minimizing transfer costs).

In [9], Hagmann predicts that the minimum should occur at 50% of memory, regardless of memory size. We will show in section 5.3 that this is in fact true in systems where latency cost is so high that transfer costs are negligible, as Hagmann assumes. However, for more realistic (for the current day) values for T_X and T_L, page reads must be considered, and hence the discrepancy.

Simple Hash join, which we do not show, behaves very similarly to NBJ as we increase the amount of space given to the I/O buffers, I and O. The main difference is that the curves are smoother (the step function is less pronounced). The minima occur at about the same points or slightly earlier. However, the optimal performance of Simple Hash in the small memory case is almost a factor of two worse than that of



(a) M = 1 MB

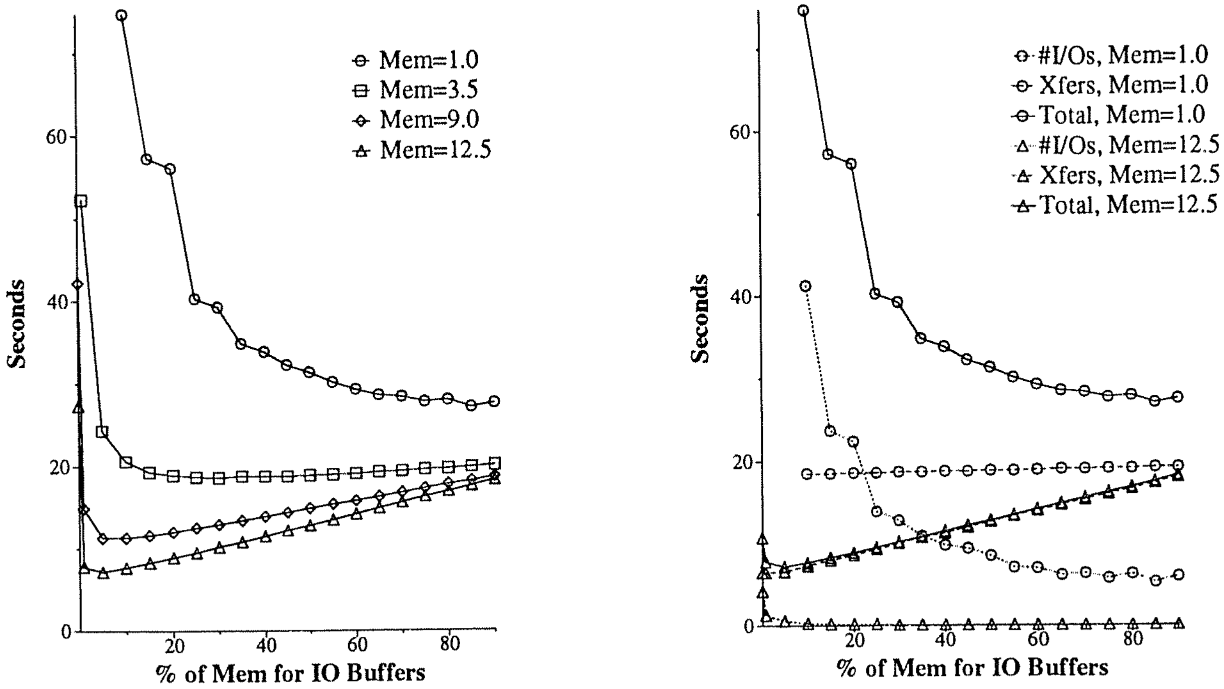
(b) M = 12.5 MB

Figure 5: Varying % of memory for I/O buffers for Sort-Merge

NBJ; for the large memory case, where R fits in one gulp, they are about the same.

The results of the same experiment for Sort-Merge join are given in Figure 5 for the same two memory sizes. For the smaller memory, the leftmost point again corresponds to the naive allocation of a single page per buffer (i.e., one page for I, one for O) in the first phase. For the larger memory $I = 3$ and $O = 2$ at the leftmost point. (For smaller allocations the results are much worse, and the graph becomes hard to read). Both curves are U-shaped, and very flat at the bottom. This is because changes in buffer allocations do not affect the amount of data transferred (the dominant component of cost) and quickly reduce the cost due to latency to close to zero. Again, the actual optimum is different for the two memory sizes (around 50% for the large memory, and between 25 and 40% for the small), but the performance of Sort-Merge is relatively insensitive to small changes in the allocation in the neighborhood of the optimum. For example, if the allocation is 10% away from the optimal buffer size, this might mean a performance loss of 2%, whereas a comparable error for NBJ could mean a factor of 50% or more in performance for the high memory case, and 15% for the small memory. However, it is still important to get in the right ballpark, as the naive allocation results in performance about two times worse than that of the optimal allocation.

For Grace Hash (not shown), we varied the amount of space given to the input buffer in phase 1, and split the remainder evenly among the output buffers for the hash buckets. The shape of the curves for Grace are similar to those for Sort-Merge. However, the minima come somewhat earlier for Grace, and the bottom of the "U" is narrower and less flat. The reason for this is that while increasing the size of I_1 reduces the number of I/Os for reads, it also reduces the amount of space available for output buffers, thus increasing the number of I/Os for writes. For Sort-Merge, on the other hand, we increased the amount of both I and O, reducing I/Os for both reads and writes. Again, performance is relatively insensitive to small deviations from the optimal allocation because latency is still only a small component of overall performance, but being on the wrong section of the curve will produce very bad results.



(a) Four Memory Sizes

(b) Component Costs

Figure 6: Varying % of memory for I/O buffers for Hybrid

Finally, Figure 6(a) shows the effect of varying the amount of space given to the input and output buffers for the Hybrid Hash algorithm for four different memory sizes. The more space given to the input and output buffers, the less space there is for R_0. Interestingly, when memory is small relative to $|R|$, the best performance is achieved when all of memory is used as buffers, that is, when there is *no* R_0. In other words, when memory is tight, Hybrid performs best if it is simply run as Grace Hash. As memory gets larger, the minimum occurs earlier and earlier, until the best performance is achieved when almost all of memory is given to R_0. The reason for these trends can be seen by examining Figure 6(b). In this figure we have shown the curves for the smallest and highest memory, with their component costs for transfers and latency. For small memory, the transfer cost is virtually constant, regardless of how much space is used for R_0, as R_0 is never big enough to reduce significantly the amount of data transferred. The more space used for input and output buffers, therefore, the fewer disk I/Os needed. Since transfers are essentially constant, the fewer the disk I/Os the better performance will be. In the large memory case, however, the number of disk I/Os is almost constant, after an initial drop, whereas space used for R_0 has a big effect on the amount of data transferred (1% of the large memory is around 15 pages).

Once again, choosing the buffer allocation correctly is important. The results of using a naive allocation of one page per buffer ($I = O = 1$) are shown as the leftmost points of Figure 6. Performance there is as much as 400% worse than the optimal. We also see that larger memories are more sensitive to small deviations from the optimum than small to medium memories, because, again, each percentage point increase in buffer size for the large memory causes a large decrease in the size of R_0, and thus, a significant increase in transfer costs. For the larger memories, being 10% from the optimum can mean a loss of about 15% in performance, as opposed to 3 to 5% for the smaller memory.

Algorithm	Estimator
NBJ	$M_S = \left\lceil \frac{\sqrt{y\vert S\vert(y\vert S\vert+M(y+\vert S\vert))}-y\vert S\vert}{y+\vert S\vert} \right\rceil$, $y = T_L/T_X$
Sort-Merge	$I = O = \lceil(\sqrt{2z} - 4)M/(z-8)\rceil$ $z = xF(\vert R\vert + \vert S\vert)/M$, $x = (T_L + T_S)/T_L$
Simple	$I = O = \lceil\sqrt{y^2 + .5yM} - y\rceil$, $y = T_L/T_X$
Grace	$B = \left\lceil \frac{\vert R\vert F+\sqrt{\vert R\vert^2F^2+4M\vert R\vert F}}{2M} \right\rceil$ $O = \left\lfloor \frac{M}{B+1} \right\rfloor$ $I_1 = M - B \times O$ $I_2 = M - \lceil\vert R\vert F/B\rceil$
Hybrid	$I_1 = I_2 = O = \lceil 1.1\sqrt{M}\rceil$

Table 2: Formulas to Estimate "Optimal" Buffer Allocations

5.3 Finding Optimal Buffer Allocations

To enable a query optimizer to correctly choose between join methods, and to get the best performance from the method selected, the preceding results show that it is important to choose a good I/O buffer allocation for each join method considered. We note that each algorithm has one critical parameter (the *determining* parameter) whose value always depends on memory allocations. For NBJ, the determining parameter is NB, the number of chunks in which R can be read; for Simple, it is NI, the number of iterations; for Grace, it is the number of buckets, B; for Hybrid, K; and for Sort-Merge it is the run length, RL. We examine each algorithm's use of memory, with the goal of developing a formula for the optimal allocation.

Our general approach is to minimize the cost function via differentiation. To do this, we approximate the cost of each algorithm by a continuous function, parameterized by the buffer size. For some algorithms with multiple buffers (e.g., Sort-Merge), we have to make further assumptions, for example, that the size of the input buffer and the output buffer are equal, in order to reduce it to an equation in one variable. Once we have a continuous function in a single variable, we differentiate it, and set the result equal to zero. For most of the algorithms, we ignore seek costs. Initially, we included them, but they complicated the equations without significantly affecting the results. This is because for most of the algorithms, it is the ratio of the number of I/Os to the number of pages transferred that is the key tradeoff for determining buffer allocations. Often, changing that ratio will incidentally change the ratio of random to sequential I/Os, but this is a smaller, secondary effect. The one notable exception to this is Sort-Merge join; for Sort-Merge there is a significant tradeoff between sequential and random I/Os, based on buffer allocations. We do, therefore, include seek costs in the Sort-Merge calculations. The results of this exercise are summarized in Table 2, and their derivations are sketched below.

Nested Block Join

Let us begin by examining NBJ. The cost of NBJ (ignoring seeks) is

$$C_{NBJ} = T_X \times (|R| + (NB \times |S|)) + T_L \times NB \times \left(1 + \left\lceil \frac{|S|}{M_S} \right\rceil\right)$$

Approximating by a continuous function, this is equivalent to

$$C_{NBJ} = T_X \times \left(|R| + \frac{|R|F|S|}{M - M_S}\right) + \frac{T_L|R|F}{M - M_S}\left(1 + \frac{|S|}{M_S}\right)$$

Taking the derivative with respect to M_S yields

$$\frac{dC_{NBJ}}{dM_S} = \frac{T_X|R|F|S|}{(M-M_S)^2} - \frac{T_L|R|F|S|}{(M-M_S)M_S^2} + \frac{T_L|R|F}{(M-M_S)^2}\left(1+\frac{|S|}{M_S}\right) = 0$$

which can be simplified to

$$(|S|+y)M_S^2 + 2y|S|M_S - y|S|M = 0$$

where $y = T_L/T_X$. Application of the quadratic formula yields the result in Table 2.

Note that if we take the limit of this formula as y approaches zero (this is the case in which latency is small, and only page transfers matter), then M_S approaches zero. In other words, if only transfers count, very little space should be devoted to S. Likewise, if we take the limit as $|S|$ approaches zero, again M_S approaches zero. In other words, when S is small, we need less buffer space for S.

Finally, we note that if we take the limit as y goes to infinity, we get to the case in which latency is so great that only disk I/Os count (page transfer costs can be ignored). This is the case that Hagmann considers in [9].

$$\lim_{y\to\infty} M_S = \sqrt{|S|\times(M+|S|)} - |S| = M_S'$$

This may not look much like $.5M$ (Hagmann's result), but in fact, as S becomes large, it converges to this. As proof, write $|S|$ as a multiple of M, say $|S| = c \times M$. Then the formula above becomes

$$M_S' = (\sqrt{c\times(c+1)} - c)\times M$$

To take the limit of this as c goes to infinity, we rewrite the square root as a Taylor series expansion, as follows,

$$M_S' = (c\times\sqrt{1+1/c} - c)\times M$$

$$M_S' = \left(c\times\left(1+\frac{1}{2c}-\frac{1}{8c^2}+\cdots\right)-c\right)\times M$$

$$M_S' = \left(c+\frac{1}{2}-\frac{1}{8c}+\cdots-c\right)\times M$$

$$M_S' = \left(\frac{1}{2}-\frac{1}{8c}+\cdots\right)\times M$$

Taking the limit as c goes to infinity yields:

$$\lim_{c\to\infty} M_S' = \frac{1}{2}M$$

Q.E.D.

Sort Merge Join

For Sort Merge, we assume that I = O. Ignoring page transfers, which are constant, the cost of Sort Merge is

$$C_{SMJ} = 2T_L \times \left(\left\lceil\frac{|R|}{I}\right\rceil + \left\lceil\frac{|S|}{I}\right\rceil\right) + (T_L+T_S)\times\left(\left\lceil\frac{|R|}{MPR}\right\rceil + \left\lceil\frac{|S|}{MPR}\right\rceil\right)$$

If we approximate C_T with a continuous function, we arrive at

$$C_{SMJ} = \frac{2T_L(|R|+|S|)}{I} + \frac{(T_L+T_S)F(|R|+|S|)^2}{2M(M-2I)}$$

Differentiating,

$$\frac{dC_{SMJ}}{dI} = \frac{-2T_L(|R|+|S|)}{I^2} + \frac{(T_L+T_S)F(|R|+|S|)^2}{M(M-2I)^2} = 0$$

After simplification,

$$(xF(|R|+|S|)-8M)I^2+8M^2I-2M^3=0$$

where $x=(T_L+T_S)/T_L$, the ratio of random to sequential I/O costs. Application of the quadratic formula yields the result given in Table 2.

If we look at the limits we again get sensible results. For example, as $|R|$ and $|S|$ become large, I and O approach zero, that is, it becomes important to make the runs as long as possible. As M increases, so should I and O; as M decreases, I and O will decrease. If seek costs are very high, x approaches infinity; in this case buffers will shrink towards zero, as this increases run length and MPR, reducing the number of seeks.

Simple Hash Join

Simple Hash lives up to its name; its solution is remarkably tractable. Again, we assume that I = O. Once we approximate C_T with a continuous function, we get

$$C_{SH}=\frac{T_X|R|F\times(|R|+|S|)}{M-2I}+\frac{T_L|R|F\times(|R|+|S|)}{I(M-2I)}$$

Taking the derivative yields

$$\frac{dC_{SH}}{dI}=|R|F\times(|R|+|S|)\times\left(\frac{2T_X}{(M-2I)^2}-\frac{T_L(M-4I)}{I^2(M-2I)^2}\right)=0$$

which simplifies out to

$$2I^2+4yI-yM=0$$

where once again, y is the ratio of latency to transfer cost (T_L/T_X). Again, the quadratic formula can be used to reach the solution shown in Table 2. As the result is proportional to y, the buffer size will be small when latency is small, that is, when it is not important to minimize I/Os. When latency is large, and it does, therefore, pay to reduce I/Os, the buffer size will also be large.

Grace Hash Join

If we try the same procedure using the cost equation for Grace Hash, the result after differentiation and simplification is a cubic. Using a system such as Mathematica [19], we can solve the cubic, but the result is difficult to understand (and annoying to program!). We note that for Grace, all of memory in phase one is used for I/O. The problem is therefore only to decide how much should be used for the single input buffer, and how much for each of the B output buffers. We note that there is no great advantage to be gained from making any one of these buffers much larger than the others, as the same amount of data is input and output. We therefore hypothesized that the best performance would be achieved when all buffers were the same size, and ran a series of experiments to test this, with excellent results.

When all buffers are equal, $I_1=M/(B+1)=O$ (if we pretend memory can be divided infinitely finely). Since $B=|R|F/(M-I_1)$ (again, approximating with a continuous function), we can substitute the desired formula for I_1, and obtain the following quadratic in B:

$$B=\frac{|R|F}{M-\frac{M}{B+1}}$$

This can be reduced to

$$MB^2-|R|FB-|R|F=0$$

which can be solved using the quadratic formula again. The result is given in Table 2. Given B, and remembering that memory comes in discrete units, we then choose values for I_1, I_2, and O. Clearly, I_2 should be chosen as large as possible, that is, it should use up all the space in the second phase not taken up by the resident R bucket. For O and I_1, we subdivide the space as evenly as possible, then give the extra pages, if any, to I_1.

Hybrid Hash Join

Finally, we come to Hybrid Hash Join. The result of differentiating and trying to solve the resulting equation for Hybrid is a quartic, with a uselessly long and complex solution[1]. However, by making a series of approximations we were able to arrive at a cost formula that we could minimize. We used the following approximations: $I = O = I_2$, $|R| \gg K$, $|R|F \gg I$, and $|S| = |R|$. We argue that these are normally reasonable assumptions (with the exception of the last; however, the solution is not highly sensitive to this ratio, and the simplification is significant). With these assumptions, we get an approximate cost C'_{HH} (for clarity, we have divided through by T_X; minimizing this function will produce the same result as minimizing C'_{HH}, as T_X is a constant).

$$\frac{C'_{HH}}{T_X} = \frac{y}{I}(5|R| - 3M/F) + |R|\frac{4I + 3y}{M - 2I}$$

Remember that y is the ratio of latency to transfer cost, T_L/T_X. Taking the derivative with respect to I yields:

$$\frac{dC'_{HH}/T_X}{dI} = \frac{-y}{I^2}(5|R| - 3M/F) + |R|\frac{4M + 6y}{(M - 2I)^2} = 0$$

$$(12yM/F + 4M|R| - 14y|R|)I^2 + (20yM|R| - 12yM^2/F)I - (5y|R| - 3yM/F)M^2 = 0$$

Using our old friend, the quadratic formula, we arrive at

$$I = \frac{-(5 - \frac{3M}{|R|F}) + \sqrt{7.5 - \frac{4.5M}{|R|F} + \frac{5M}{y} - \frac{3M^2}{y|R|F}}}{6/|R|F + 2/y - 7/M}$$

Since this is already a fairly rough approximation, due to the simplifications used to reduce the cost to a quadratic, we feel justified in simplifying further, by taking the limit as $|R|$ goes to infinity (basically, looking at what happens when R and S are much bigger than M), and noting that $M \gg y$. This yields the simple result given in Table 2. In fact, this last approximation tends to compensate for the earlier approximations; this simple formula yields excellent results, often 10% better than those achieved using the value for I given by the quadratic above. However, we do not know if these results are strictly optimal for all memory sizes and values of $|S|$ and $|R|$.

Using the Formulas

For NBJ, Sort Merge and Simple Hash, the formulas in Table 2 should be used as an initial estimate of the buffer sizes. To account for the discrete nature of the actual cost formulas, this estimate should be used to compute an optimal value of the determining parameter for the algorithm. Then for Sort-Merge and NBJ the buffer space should be re-computed as the maximum number of pages that still produces this value for the determining parameter. For example, suppose we have a memory size of 500 pages ($M = 500$) and a ratio of latency to transfer cost of about three ($y = 3$), with $|R| = |S| = 1250$. For NBJ, we get the value $M_S = 37$. Then $M_R = 500 - 37 = 463$. For this value of M_R, with $|R| = 1250$, we get $NB = \lceil |R|F/M_R \rceil = \lceil 1500/463 \rceil = 4$. Note that if we are reading R in 4 chunks, we need only 375 pages of

[1] The solution, using Mathematica, covered tens of pages, unformatted!

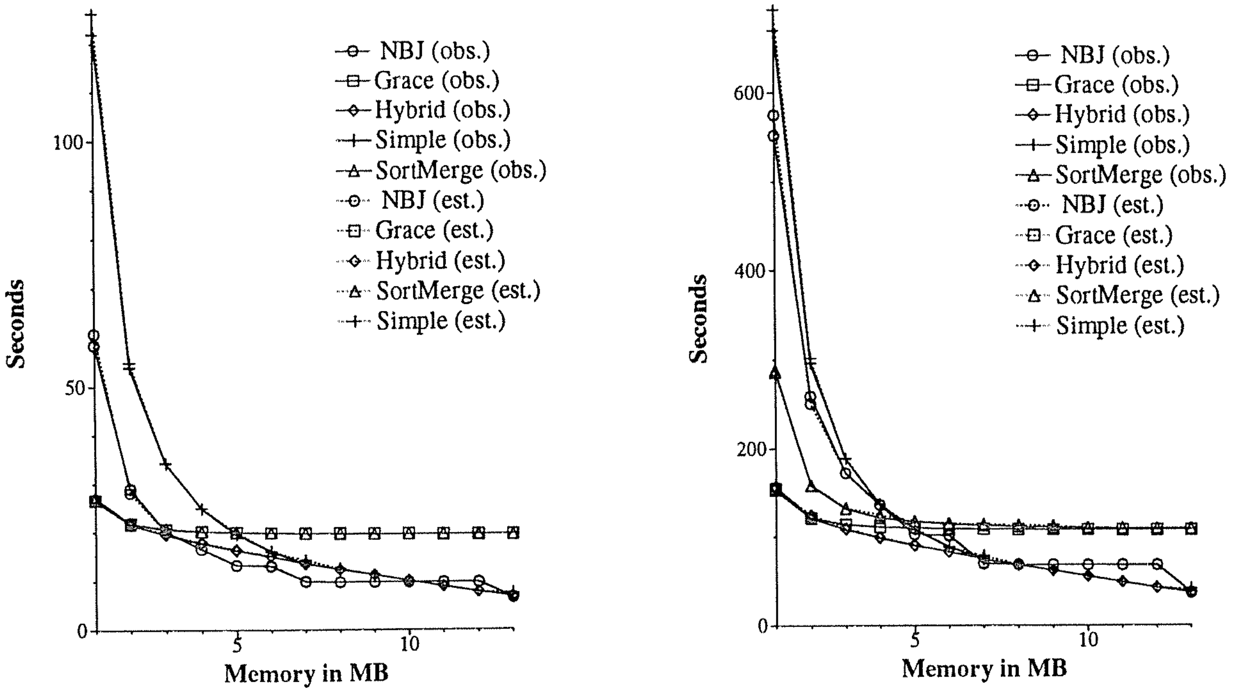
memory for each chunk ($\lceil |R|F/NB \rceil$). Thus, $M_S = M - M_R = 125$. This will be strictly better than the original estimate, as it will decrease the number of I/Os needed to read S, while leaving the number of I/Os for R and the amount of data transferred the same. Similar optimizations can be made for Sort Merge using an initial estimate of buffer size to set run length.

For Simple Hash, some adjustment of the estimate is necessary, as the continuous approximation to the cost formula was a very rough one, introducing significant error in the results in some cases. However, each change in the buffer space changes the amount of data transferred, so using the maximum number of pages is not a solution. We have found through experimentation that a good approximation results from the following procedure. Let the initial estimate obtained from the formula in Table 2 be I_0. The largest buffer size for the number of iterations, NI, determined by I_0 is $I_{max} = \frac{1}{2}\left(M - \left\lceil \frac{|R|F}{NI} \right\rceil\right)$. The optimum lies between I_0 and I_{max}, as I_0 is an underestimate. Since I_0 is more accurate for larger memories, we would like the estimate to be close to I_0 when M is large, and closer to I_{max} when M is small. We therefore choose $I = O = I_0 + \frac{w}{M}(I_{max} - I_0)$, where w represents a small memory size. For our parameter values, we found $w = \min(M, 125)$ worked well. (I.e., for memories up to 125 pages, $I = O = I_{max}$. For bigger memories, $I = O$ slides back towards I_0, as desired).

Our formula for Grace sets the determining parameter directly, and this "sliding" process for the buffer allocations is done by the equations in Table 2. For Hybrid, the formula we arrived at works well as it is; some improvement using similar tricks may be possible, but it must be done carefully, as each change in the buffer space always affects the overall amount of data that will be transferred (by changing the size of R_0). We did not find it worthwhile to pursue this.

The formulas that we arrived at are as diverse as the algorithms themselves. Each algorithm uses memory differently, and none of the derivations was particularly easy. To test our results, we experimentally varied the buffer size for each algorithm (for M between 1 and 13 MB) and compared the observed minima to those predicted by our formulas when used as described above. In Figure 7 we show the predicted response times in seconds for our formulas for all five algorithms ("estimated" curves) as well as the observed minima for those algorithms ("observed" curves), for $|S| = |R|$ and $|S| = 10 \times |R|$. In most cases, the formulas are amazingly accurate, typically within 2% of the optimal. Looking at Table 3, we get a better sense of the errors involved in Figure 7. As indicated above, the roughest estimates are for Simple and Hybrid Hash Join, and even those estimates are normally within 5%, except when memory is sufficiently large to hold R and its hash table, with some space left over, i.e., $M > |R|F$. In this case, too large a buffer can keep R from fitting, greatly increasing the cost of algorithms such as NBJ, Hybrid and Simple Hash. When memory is this large, an additional sanity check, comparing the cost of two chunks versus a single one, should be made. From the table, we can also see that the estimates tend to get worse as $|S|$ increases. This is to be expected, given the simplifications we made when deriving the formulas for the estimates. However, the percent error increases only slightly with the ten-fold increase in $|S|$.

To see what we gain from correctly tuning buffer allocations, compare the graphs in Figure 8 to those in Figure 9 (repeated for convenience from Figure 2 on page 11). The differences are fairly dramatic. First, note that the expected performance of all of the algorithms is significantly better in Figure 8. For example, Grace improves by a factor of about 2.5, and NBJ by as much as 3.5 times (relative to NBJ_1; it is 1.3 to 1.5 times better than NBJ_{50}). Hybrid also improves substantially, making it much more competitive, though it still only beats NBJ in low memory (under 3 MB), when it essentially behaves like Grace, and in very high memory ($M > |R|F$), where it strongly resembles Simple. Differences in performance that were significant under the naive buffer allocations become insignificant when the allocations are well-tuned. For example, when buffer sizes are set appropriately, Grace and Sort-Merge are virtually indistinguishable. This is because, with the buffer sizes set correctly, transfer costs dominate the total cost, and these costs are



(a) $|S| = |R|$

(b) $|S| = 10 \times |R|$

Figure 7: Performance using observed optimal buffer allocations vs. estimates

	% Error when $\lvert S\rvert = \lvert R\rvert$					% Error when $\lvert S\rvert = 10 \times \lvert R\rvert$				
M (MB)	NBJ	S-MJ	Grace	Simple	Hybrid	NBJ	S-MJ	Grace	Simple	Hybrid
1.0	3.9	0.0	0.0	3.4	0.0	4.2	0.0	1.4	3.5	0.0
2.0	3.1	0.0	0.5	1.9	1.9	0.0	0.1	0.3	1.9	3.2
3.0	0.0	0.0	0.5	0.3	1.0	0.0	0.7	0.2	0.3	1.3
4.0	0.0	0.0	0.0	0.0	1.1	0.0	3.0	0.0	0.1	1.1
5.0	0.0	0.5	0.0	2.0	1.2	0.0	0.4	0.1	1.9	0.9
6.0	0.0	0.0	0.0	0.0	0.7	0.0	0.9	0.0	0.0	0.5
7.0	0.0	0.0	0.0	4.4	0.7	0.0	1.1	0.0	5.2	0.5
8.0	0.0	0.0	0.5	1.6	0.0	0.0	0.1	0.1	1.6	0.3
9.0	0.0	0.0	0.0	0.0	0.0	0.0	1.6	0.0	0.0	0.0
10.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
11.0	0.0	0.0	0.0	0.0	1.1	0.0	0.0	0.0	0.2	0.2
12.0	0.0	0.5	0.0	1.3	0.0	0.0	0.1	0.0	0.5	0.5
13.0	0.0	0.0	0.0	7.0	0.0	0.0	0.1	0.0	7.6	0.0

Table 3: Percent Error in Performance Using Estimated Buffer Sizes

	Transfer-only optimizer		I/O count-only optimizer		Detailed optimizer	
M (MB)	Winner	Actual Cost	Winner	Actual Cost	Winner	Actual Cost
0.35	Hybrid	88.2	S-MJ	69.4	S-MJ	69.4
0.5	Hybrid	86.7	Grace/Hyb.	43.4/43.4	Grace/Hyb.	43.4/43.4
1.0	Hybrid	83.4	NBJ	85.7	Grace/Hyb.	26.6/26.6
2.0	Hybrid	78.0	NBJ	43.3	Grace/S-MJ	21.8/21.8
3.0	Hybrid	72.5	NBJ	29.8	Hybrid	19.7
4.0	Hybrid	67.5	NBJ	23.0	NBJ	16.6
5.0	NBJ	44.2	NBJ	19.7	NBJ	13.3
6.0	NBJ	44.2	NBJ	16.4	NBJ	13.1
7.0	NBJ	30.6	NBJ	16.4	NBJ	9.9
8.0	NBJ	30.6	NBJ	13.1	NBJ	9.8
9.0	NBJ	30.6	NBJ	13.1	NBJ	9.8
10.0	Simple	36.5	NBJ	13.1	NBJ	9.8
11.0	Simp./Hyb.	31.9/32.3	NBJ	13.1	Simple	8.9
12.0	Simp./Hyb.	27.4/27.4	NBJ	9.8	Simp./Hyb.	7.8/7.8
13.0	NBJ/Simp./Hyb.	16.9/27.3/27.3	NBJ	9.8	NBJ	6.6

Table 4: Comparison of Optimizer Predictions under Three I/O Cost Models

identical for the two algorithms (see Section 4).

5.4 Yes, The Model Matters

The differences between Figure 8 and Figure 9 illustrate, once again, that it is necessary to model buffer allocations in a query optimizer in order to correctly choose among join methods. Since buffer allocation and latency have such important effects on performance, it is natural to ask whether a model based only on counting disk I/Os would be sufficient, as suggested by [9]. That is, do we really need to include the cost of page transfers?

Table 4 dramatizes the answer. This table compares the join algorithm recommendations of the two simpler models (transfer-only and I/O count-only) with those of the detailed I/O cost model. Our intent was to approximate the behavior of an optimizer based on one of these models. Thus, we assumed buffer allocations would be tuned to be optimal according to the model doing the cost predictions. For example, the transfer-only model uses NBJ_1 for NBJ, while the I/O count-only model uses NBJ_{50}. The detailed model uses the formulas for optimizing buffer allocations derived in Section 5.3.

The table shows, for a range of memory sizes from .35 to 13 MB, which algorithm each model would pick as the winner (the cheapest way to execute the join), and the "actual" cost (according to the detailed model) in seconds of executing the join using that algorithm with the buffer allocations chosen given that model. In cases where more than one algorithm is listed, the model was unable to distinguish between them (that is, an optimizer based on that model would predict identical performance for those algorithms). However, the actual cost of the algorithms according to the detailed model may differ; thus, the cost of each algorithm is given separately. Clearly, an optimizer based on either of the two simpler models would make some serious mistakes, choosing both the wrong algorithms and the wrong buffer allocations. These decisions could lead to performance as much as four times worse than the "optimal" picked by the detailed model.

Why is the number of I/Os such a bad predictor of the optimized algorithms' performance, when we showed earlier (in Section 5.1) that latency is so important? The answer is simply that by optimizing the buffer allocations, we reduced the negative effect of the excessive I/Os that were occurring with the naive allocations. Thus, the effect of page transfers was increased – but only because we significantly reduced the number of I/Os. If the model that we used for buffer allocation had not included both page transfers and latency, of course, we could not have achieved these results.

5.5 Discussion

We have made several simplifying assumptions in our work. Several of the algorithms presented require a certain minimum amount of memory in order to compute the join in two phases. For example, the Sort-Merge algorithm requires $M > \sqrt{F|S|}$. With less memory, the algorithm will require multiple merge passes. We are confident that our thesis, that a detailed model is necessary, is still equally valid when this occurs, as indicated by the empirical results in [6]. However, we have not directly verified this, nor do we predict which algorithms will perform better than others when memory becomes extremely scarce. We expect extensions for this case to be straightforward.

For ease of exposition, we used a simple version of each algorithm. Variations of several of the join methods have been proposed that generally improve the performance of the methods. For example, Won Kim [10] has proposed a variation on NBJ, in which S is read first forwards, and then backwards. This variation will generally perform better than the version described here, as the number of disk I/Os and of page transfers will be (slightly) less. Many variations on SMJ have been proposed [6], and Graefe [7] has detailed several optimizations of Hybrid Hash Join. We have purposely chosen simple versions: the point of this work is not to say which is the best join method, but to show how the model used affects our view of "best".

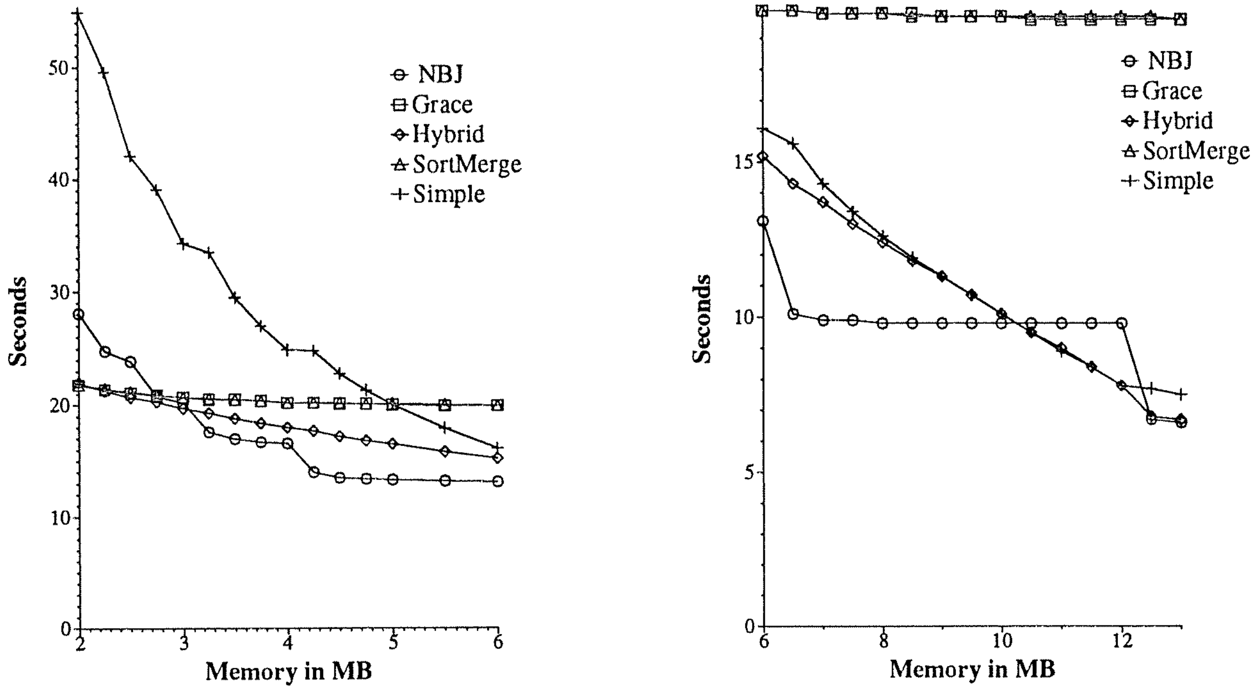
It should also be stressed that we used only one set of values for the key parameters, T_X, T_L and T_S. Different weights would again change our view of which algorithm is "best", but would only emphasize the need for a detailed I/O cost model. Finally, different I/O systems may include other features that should be included in the cost model, for example, overlapped or parallel I/O.

6 Conclusions

In this paper, we have looked at three I/O cost models: the transfer-only model common in the literature, an I/O count-only model advocated by [9], and a detailed model that we proposed that includes latency, seek and page transfer costs. We showed that the common wisdom from previously published work is not wholly reliable. Hybrid is not *always* the method of choice for *ad hoc* joins; Nested Block Join does not perform best when 50% of the available memory is given to each relation (or with a single page for the larger relation).

The results in the preceding pages indicate that good predictions of join performance require a detailed I/O cost model. We have shown that a query optimizer needs to consider all three components of I/O cost, and needs to have a thorough understanding of the algorithms it is modeling, in order to correctly choose a good join method. Furthermore, the optimizer should be aware of how the implementation of a particular join method allocates buffers for I/O, and the join method implementation should pay careful attention to buffer allocation. Systems that allow hints to be passed between join execution and the buffer manager will be at an advantage here [12].

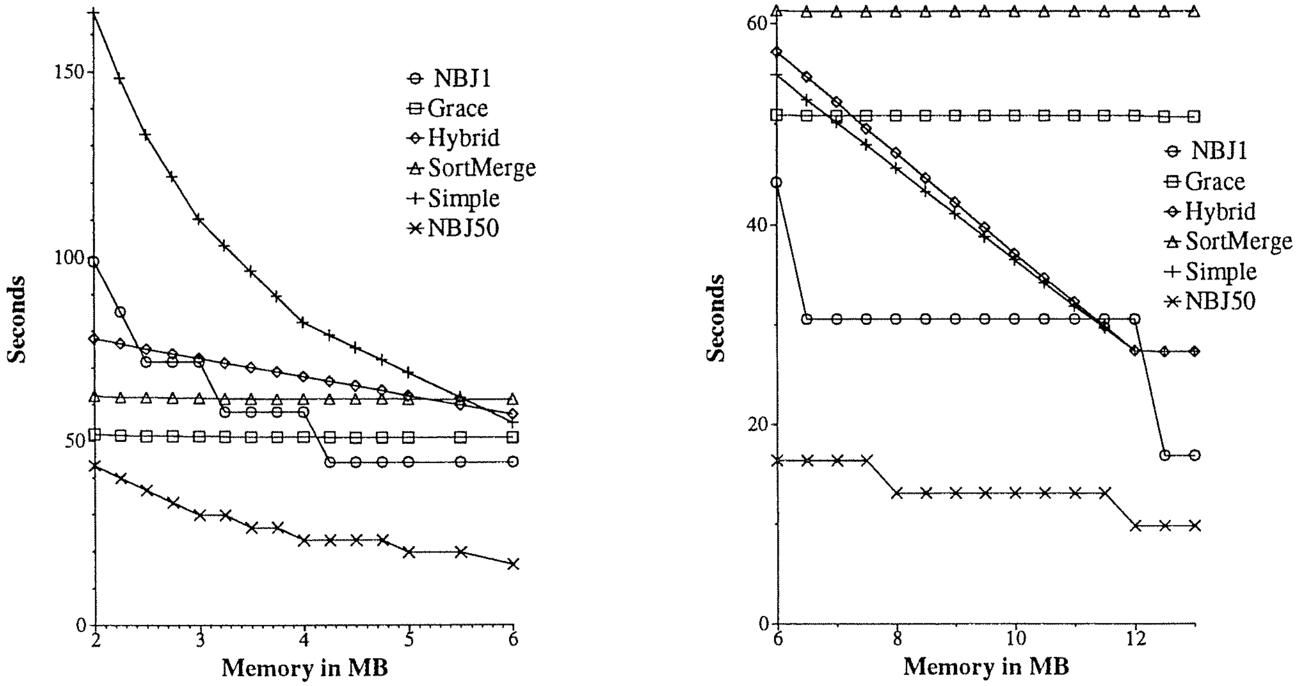
Once again, we stress that the reader should not interpret our work as simply, for example, proving that NBJ is the best choice for a large range of memory sizes. Instead, we hope this work will inspire database system builders and optimizer "gurus" to evaluate which algorithms are appropriate for their own hardware



(a) Small memory

(b) Large memory

Figure 8: Predictions of the detailed model, optimized buffer allocations



(a) Small memory

(b) Large memory

Figure 9: Predictions of the detailed model, naive buffer allocations

and software systems, using a detailed model such as that which we proposed here.

With our newly gained understanding of disk-based join costs, we now intend to go back to our study of DBMS controlled tertiary store. We will start by examining the performance of various *ad hoc* join methods when they need to access data on a tertiary store, in order to gain an understanding of the key parameters and to get a better feel for the technology. We hope that we will learn which techniques to use under which circumstances to successfully cope with large volumes of data. Once we have an understanding of the basics, we plan to tackle some of the system level issues, for example, the open problems related to caching, migration, multiple queries, and so on. A detailed discussion of these issues can be found in [4].

Acknowledgements

We gratefully acknowledge the assistance of Peter Haas, who reminded us how to differentiate, ran Mathematica for us, and contributed the proof in section 5.3 that Hagmann's 50% rule holds when latency is large.

References

[1] M. Blasgen and K. Eswaran, "Storage and Access in Relational Data Bases". *IBM Sys. Journal*, 16(4), 1977.

[2] B. Bratbergsengen, "Hashing Methods and Relational Algebra Operations". *Proc. 10th VLDB*, Aug, 1984, Singapore, pp. 323-333.

[3] K. Brown et al., "Resource Allocation and Scheduling for Mixed Database Workloads". Tech. Rep. 1095, Univ. of Wisconsin, Madison, July 1992.

[4] M. Carey, L. Haas and M. Livny, "Tapes Hold Data, Too: Challenges of Tuples on Tertiary Store". *Proc. SIGMOD*, May 1993, Washington, D.C.

[5] D. DeWitt, et al., "Implementation Techniques for Main Memory Database Systems". *Proc. SIGMOD*, Jun 1984, Boston, pp. 1-8.

[6] G. Graefe, A. Linville, L. Shapiro, "Sort versus Hash Revisited". Tech. Rep. No. CU-CS-534-91, Comp. Sci. Dept., Univ. of Colorado at Boulder, July 1991.

[7] G. Graefe, "Performance Enhancements for Hybrid Hash Join". Submitted for publication, 1993. An earlier version is available as a Colorado University Technical Report, CS TR 606, July 1992.

[8] J. Gray, private communication.

[9] R. Hagmann, "An Observation on Database Buffering Performance Metrics". *Proc. 12th VLDB*, Aug, 1986, Kyoto, pp. 289-293.

[10] W. Kim, "A New Way to Compute the Product and Join of Relations". *Proc. SIGMOD*, May 1980, Santa Monica, pp. 179-187.

[11] D. Knuth, The Art Of Computer Programming, Vol. 3: Sorting and Searching. Addison-Wesley Publishing, 1973.

[12] M. Lee, "Interaction between the query processor and buffer manager of a relational database system". Masters Thesis, MIT, May 1989. Also available as IBM Research Report RJ6884, San Jose, CA, Jun 1989.

[13] L. Mackert and G. Lohman, "R* Optimizer Validation and Performance Evaluation for Local Queries". *Proc. SIGMOD*, May 1986, Washington, DC.

[14] Mathematica

[15] P. Mishra and M. Eich, "Join Processing in Relational Databases". *ACM Comp. Surveys*, 24(1), March 1992.

[16] P. Selinger et al., "Access Path Selection in a Relational Database Management System". *Proc. SIGMOD*, May 1979, Boston.

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

[18] P. Valduriez, "Join Indices". *ACM Trans. on Database Sys.*, 12(2), Jun. 1987.

[19] S. Wolfram, "Mathematica: A System for Doing Mathematics by Computer", 2nd edition, Addison-Wesley, Redwood City, 1991.