A Performance Evaluation of Pointer-Based Joins

Eugene J Shekita
Michael J Carey

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT — In this paper we describe three pointer-based join algorithms that are simple variants of the nested-loops, sort-merge, and hybrid-hash join algorithms used in relational database systems Each join algorithm is described and an analysis is carried out to compare the performance of the pointer-based algorithms to their standard, non-pointer-based counterparts The results of the analysis show that the pointer-based algorithms can provide significant performance gains in many situations The results also show that the pointer-based nested-loops join algorithm, which is perhaps the most natural pointer-based join algorithm to consider using in an object-oriented database system, performs quite poorly on most medium to large joins

1. INTRODUCTION

Most semantic and object-oriented data models include the notion of reference attributes or object-valued attributes (e g , [Ship81, Zani83, Care88, Kim89, Deux90]) Semantic and object-oriented database systems typically implement such attributes with pointers or object identifiers (OIDs) OIDs can be purely logical, but often they have a physical component [Chan82, Care89, Vele89, Deux90] Relational database systems also support the notion of physical pointers in the form of record identifiers (RIDs) These are generally used to implement indexes and various internal data structures, but they can also be used to provide efficient support for referential integrity and certain kinds of joins [Care90, Haas90] In view of the fact that physical pointers are supported by a variety of database systems, it seems appropriate to examine how such pointers can be used in query processing

In this paper we describe how pointers can be used to process join queries We describe three pointer-based join algorithms that are simple variants of the well known nested-loops, sort-merge, and hybrid-hash join algorithms [Blas77, Shap86] The algorithms we describe can be utilized in any database system where physical pointers are used to link records or objects together In a relational system, our algorithms would presumably be used to process foreign-key joins whenever the relations being joined are linked internally by RIDs to provide support for referential integrity or pointer-based joins (e g , as described in [Care90, Haas90]) And in semantic and object-oriented database systems, our algorithms would presumably be used to process functional joins, that is, joins between sets that are linked together by object-valued attributes or OIDs

The remainder of this paper examines the three pointer-based join algorithms mentioned above in some detail Each algorithm is described, and then an analytical model is employed to compare the performance of the pointer-based algorithms to their standard, non-pointer-based counterparts Results are presented for large, full-relation joins and small to medium-sized joins with a selection predicate

The rest of this paper is organized as follows Section 2 illustrates the type of joins that will be analyzed and provides examples of how such joins arise in relational and object-oriented database systems In Section 3, descriptions of the join algorithms that will be analyzed are given Then in Section 4, the pointer-based algorithms are analyzed and compared to their standard, non-pointer-based counterparts Related work is mentioned in Section 5, and conclusions are finally drawn in Section 6

2. THE TYPES OF JOINS THAT ARE ANALYZED

In this section, we provide simple examples of the types of joins that will be analyzed The examples are intended to show that there are indeed practical situations in which pointer-based join algorithms can be used, both in relational database systems and in object-oriented database systems At the end of this section, we also note the types of joins which are not amenable to the pointer-based algorithms that are analyzed in this paper

Throughout this paper, we will be concerned with the join of two sets, denoted R and S, that stand in a many-to-one relationship with each other We will assume a simple pointer structure, where each object in R contains a pointer to its related object in S Although pointer-based join algorithms can be developed for other relationships with different pointer structures, this is perhaps the most natural one to consider initially

2.1. Pointer-Based Joins in a Relational Database System

In a relational context, the types of joins being considered are illustrated by the following schema

```
Emp (name char[ ],  age int,  jobid int  )
Job (jobid int,  name char[ ],  wage int  )
```

Here, Emp and Job stand in a many-to-one relationship to each other, with the attribute *jobid* serving as a foreign key for the Job relation The joins we will be analyzing have the form

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00014-85-K-0788, by the National Science Foundation under grant IRI-8657323, by DEC through its Incentives for Excellence program, and by a donation from Texas Instruments

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery To copy otherwise or to republish requires a fee and/or specific permission
© 1990 ACM 089791 365 5/90/0005/0300 $1 50

retrieve (Emp name, Emp age, Job name)
where Emp jobid = Job jobid
and Emp age < 30

These sorts of foreign-key joins are probably the most common type of join in a relational database system They are also a natural candidate for using pointer-based joins For example, it is fairly easy to imagine a situation where the database system initializes and maintains hidden pointers (i e , RIDs) in each Emp record, linking it to its related Job record This is in fact what the incremental join facility in Starburst does (as described in [Care90, Haas90]), and it could also be easily applied to the sort of reference attributes described in [Zani83]

2.2. Pointer-Based Joins in an Object-Oriented Database System

In an object-oriented database system, the opportunity for pointer-based joins arises quite naturally because pointers in the form of OIDs are commonly used to implement object-valued attributes The following schema, which has been written in the EXTRA data model [Care88], illustrates how our example join might appear in an object-oriented system (or in GEM [Zani83] for that matter)

define type EMP (name char[ ], age int, job **ref** JOB)
define type JOB (name char[ ], wage int)

create Emp {**own unique ref** EMP}
create Job {**own unique ref** JOB}

Here, the top half of the schema defines the structure of the objects in the Emp and Job sets that are created in the bottom half of the schema In the EXTRA data model, the notation **own unique ref** indicates ownership or an existence dependency, therefore, if the set Emp is deleted, all the employee objects in Emp are also deleted The key thing to recognize here is that the attribute *jobid*, which served as the foreign key in the relational schema, has been replaced by the object-valued attribute *job*, consequently

retrieve (Emp name, Emp age, Emp job name)
where Emp age < 30

has the same semantics as the foreign-key join presented above Depending on the application, these sorts of joins may be more or less common in an object-oriented database system, but we still expect them to represent an important class of queries

For our pointer-based join algorithms to work in an object-oriented setting, it is necessary for object-valued attributes to be implemented with OIDs that have a physical component (as opposed to strictly logical OIDs) It is also necessary for sets such as Emp and Job to be stored as disk files in much the same way that they would be stored in a relational database system There are currently at least two object-oriented databases being developed that fulfill these requirements [Care89, Vele89, Deux90]

2.3. The Types of Joins Not Analyzed

It is important to note that the pointer-based join algorithms which are analyzed in this paper can be used on either (i) full joins or (ii) select-project-join queries in which there is a selection predicate on Emp that is more restrictive than any selection predicate on Job The pointer-based algorithms that are analyzed cannot, however, be used effectively on select-project-join queries in which the selection predicate on Job is the more restrictive one This is because the most efficient way to process such joins is in a direction that is opposite to the pointers that link Emp and Job

To process such joins with pointers, the simple many-to-one pointer structure that we are assuming is inadequate What is needed is some sort of structure that links each object in Job to the objects in Emp that are related to it One possibility is the kind of Codasyl-like pointer structure used in [Care90] Another possibility is a one-to-many pointer structure, where each object in Job contains a set of embedded pointers linking that object to its related objects in Emp Unfortunately, space limitations prevent us from considering join algorithms based on such pointer structures in this paper They are discussed in [Care90, Shek90], and their analysis will be addressed in future work

3. DESCRIPTIONS OF THE JOIN ALGORITHMS

This section describes the pointer-based join algorithms that will be analyzed To be consistent, we shall use object-oriented database terminology throughout this section and in the analysis, but it should be clear how the discussion applies in a relational setting as well

The description and analysis of the algorithms is based on the join of two object sets, denoted R and S, which stand in a many-to-one relationship with each other Both R and S are assumed to be stored as separate disk files In the standard, non-pointer-based join algorithms, we assume that each object in R contains a foreign key for the object it references in S, while in the pointer-based join algorithms, we assume that each object in R contains a pointer to the object it references in S As indicated earlier, our pointer-based join algorithms assume that physical pointers are used In particular, we assume that each pointer includes a page identifier or PID that points directly to the disk page of the object that the pointer references

The remainder of this section describes our pointer-based join algorithms First, however, their standard, non-pointer-based counterparts are reviewed Note that only high-level descriptions of the algorithms are given here More details will be given in the analysis

3.1. Standard, Non-Pointer-Based Join Algorithms

3.1.1. Standard Nested-Loops/Index-Nested-Loops

The nested-loops join algorithm usually performs poorly on full joins, and consequently it is not considered for such joins in the analysis It will be used, however, in the analysis of small to medium-sized joins with a selection predicate on R For those types of joins, a version of the nested-loops algorithm commonly referred to as index-nested-loops is used [Blas77] Index-nested-loops is an option only when there is an index on S It executes as follows

(1) Let R′ be the result of the selection on R The objects in R′ are first sorted by their join attribute (i e , foreign key)

(2) Each object in R′ is then examined and its join attribute is used to probe the index on S to find the object that it joins with in S

Note that this is slightly different than the conventional index-nested-loops algorithm in that R' is sorted here We have chosen this variation because when the index on S is a B+ tree, and when R' is small, as in the examples we analyze, sorting R' in this manner reduces the cost of probing the index enough to lower the overall cost of the join

3.1.2. Standard Sort-Merge

Although sort-merge has been shown to be generally inferior to hybrid-hash [Dewi85, Shap86], it is still considered in the analysis because it is used by many relational database systems In the analysis, we will assume that memory is large enough so that R and S can be sorted and merged in two passes, as described in [Shap86] Under that assumption, the algorithm executes as follows

(1) R is read into memory and sorted output runs are generated using a heap or some other priority queue structure The output runs are sorted by the value of the join attribute The same is done for S

(2) The output runs of R and S are then concurrently merged in memory As objects from R and S are produced in sorted order by these merges, they are checked to see if their join attributes match When an object from R matches one from S, the two are joined and the result is output

3.1.3. Standard Hybrid-Hash

Hybrid-hash is usually considered to be the best-performing join algorithm when a full join is performed [Shap86] Assuming that the size of S is bigger than R (otherwise exchange R and S), it executes as follows

(1) R is read into memory[1] and divided into $B + 1$ partitions $R_0, R_1, \ldots R_B$ on the basis of some hash function that is applied to the join attribute The same is done for S, thus, R_i will only join with S_i for $0 \leq i \leq B$ The value of B is chosen such that S_0 can be joined in memory with R_0 as S is being partitioned and such that a hash table for each R_i can later fit in memory for $1 \leq i \leq B$

(2) After S has been partitioned and S_0 has been joined with R_0, each R_i is joined with S_i for $1 \leq i \leq B$

(3) The joins between R_i and S_i for $0 \leq i \leq B$ are performed with hashing R_i is read into memory, and each object in R_i is hashed on its join attribute and inserted into a hash table S_i is then read into memory, and the join attribute of each object in S_i is used to probe the hash table and find the object that it joins with in R_i

In the above description, R is said to play the role of the *inner set*, while S is said to play the role of the *outer set* To minimize the number of I/Os, hybrid-hash always chooses the smaller of R and S to play the role of the inner set [Shap86]

3.1.4. Using Bit Filters

As mentioned earlier, we shall consider small to medium-sized joins with a selection predicate on R in the analysis For such joins, a technique call *bit filtering* [Dewi85, Mack86] can be used to reduce the cost of the sort-merge and hybrid-hash algorithms Let R' be the result of the selection on R Bit filtering works by hashing the join attribute of each object in R', and then using the resulting hash value to set a bit in a bit vector Later, when S is read, the join attribute of each object in S is hashed in the same way, and the resulting hash value is used to check the bit vector If the bit is not set, then that object can be safely discarded since it cannot possibly join with any object in R'

As demonstrated in [Dewi85], bit filtering can result in significant savings due to the fact that non-participating objects in S do not have to be stored on disk between the various phases of the sort-merge and hybrid-hash algorithms In the analysis of small to medium-sized joins, the analysis of both the sort-merge and hybrid-hash algorithms will include the effects of using a one-page bit filter

3.2. Pointer-Based Join Algorithms

3.2.1. Pointer-Based Nested-Loops

The pointer-based nested-loops algorithm is the algorithm that results when naive pointer traversal is used to compute the join In the algorithm, only one page of memory is allocated to read R, and the rest are allocated to read S R is read into memory page by page When a page of R is read into memory, the objects on that page are scanned one by one, and the pointer in each object r that is scanned is used to identify the object s that it joins with in S The page containing s is read into memory (if it is not already there) and r is joined with s

3.2.2. Pointer-Based Sort-Merge

As the analysis will clearly show, one of the problems with the pointer-based nested-loops algorithm is that it makes no attempt to optimize disk reads As a result, a particular disk page in S can end up being read more than once For example, suppose that two objects r_1 and r_2 reference the same object s in S Depending on how R is organized, r_1 and r_2 may not be physically clustered on the same disk page in R If that is the case, then between the time when r_1 is joined to s and the time when r_2 is joined to s, the page P containing s may be paged out of memory by the buffer replacement algorithm In that event, P would have to be read twice, once to join r_1 with s, and a second time to join r_2 with s

The pointer-based sort-merge algorithm avoids this problem by first sorting all of the objects in R by the value of the page identifiers (PIDs) stored in their pointers The effect of sorting R in this manner is to group all of the objects in R that reference the same page in S Doing so guarantees that each page in S will be read only once The algorithm executes as follows

(1) R is read into memory and sorted much like it is in the standard sort-merge algorithm, except that here the output runs are sorted by PID values rather than by the join attribute S is *not* sorted By "PID value" we mean the value of the page identifier that is stored in a pointer

(2) The output runs of R are then merged in memory Each object r produced by the merge is examined and its pointer is used to identify the object s that it joins with in S The page containing s is read into memory and r is joined with s

[1] When we say that "R is read into memory", we mean that R is incrementally read into memory page by page, that is, page P_{i+1} is read only after all the objects in page P_i have been processed

3.2 3. Pointer-Based Hybrid-Hash

The pointer-based hybrid-hash algorithm groups the objects in R by PID values much like the pointer-based sort-merge algorithm Instead of sorting R, however, hashing is used to group the objects that reference the same page in S The algorithm executes as follows

(1) R is partitioned much like it is in the standard hybrid-hash algorithm, except that here it is partitioned by PID values rather than by the join attribute S is *not* partitioned

(2) Each partition R_i of R is joined with S by taking R_i and building a hash table for it in memory The hash table is built by hashing each object r in R_i by the value of its pointer's PID The hash table is built in such a way that all objects which reference the same page in S are grouped together in the same hash entry

(3) Once the hash table for R_i has been built, each of its hash chains is scanned Each time a new hash entry H is encountered on a chain, the page in S associated with H is read, and all of the objects in R that reference that page, which have been grouped in H, are joined with their corresponding objects in S

Note that one of the key differences between this algorithm and the standard hybrid-hash algorithm is that R is the only set that is partitioned, and as such it always plays the role of the inner set This is necessary because the direction of the pointers is from R to S As we shall see, when R is significantly larger than S, this can cause the standard hybrid-hash algorithm to outperform the pointer-based hybrid-hash algorithm

4. ANALYZING THE JOIN ALGORITHMS

In this section, an analysis is performed to quantitatively compare the pointer-based join algorithms to their standard, non-pointer-based counterparts The net CPU and I/O cost of executing each join algorithm is derived and used as the basis for comparison Two types of joins are analyzed full joins, where all of R is joined to all of S, and small to medium-sized joins, where the result of a selection on R is joined to S For the small to medium-sized joins, the selectivity of the predicate on R will be varied to control the size of the joins Projections are not considered in the analysis, since they do not change the general results in any significant way

4.1. Assumptions in the Analysis

One of the key assumptions in the analysis is that R and S are *relatively unclustered* By this we mean that objects in R are *not* ordered by their references to S The reader should bear in mind that this is an important assumption and has a considerable impact on the analysis, since it makes joins between R and S more expensive We make this assumption because we feel that it represents a common case, as the objects in R would often be ordered by the value of some data field, not by their references to S It is also the most difficult case to analyze The effect of other clustering choices will be addressed in future work

For uniformity, and to make the analysis tractable, we assume that each object in S is referenced by exactly k objects in R The value k is referred to as the *sharing level* and it is varied in the analysis The result of this assumption is that the cardinality (although not necessarily the size) of S will always be equal to $1/k$ times the cardinality of R In general, this is an unrealistic assumption because in practice R may reference only part of S But in the analysis, we shall also consider joins where there is a selection predicate on R, and these are effectively like joins in which R references only part of S

We will also assume that B+ trees exist on both R and S These will be used in the small to medium-sized joins with a selection predicate on R The index on R will be used to evaluate the selection predicate, while the index on S will be used by the index-nested-loops algorithm to avoid a file scan of S

Finally, we will ignore the minimal memory requirements of the different join algorithms in the analysis and simply assume that there is always enough memory for each of the algorithms to execute In all the examples that are analyzed, the minimal memory requirements turn out to be quite modest, so this is a reasonable assumption to make

4.2. Parameters Used in the Analysis

The parameters that are used in the analysis are listed in Table 1 Although there are a large number of parameters, only a few of them are actually varied Moreover, most of the parameters are not really parameters per se, but rather functions of a small set of "core" parameters, which consist of the parameters in the top half of Table 1 Defaults for the core parameters are listed in Table 2

The meaning of most parameters should be clear from Table 1 The parameters that are kept fixed in the analysis are noted as such in Table 2 As indicated, all times are given in terms of milliseconds The time to compare, hash, move, and swap values (in memory) has been expressed as a function of the number of instructions executed and the instruction rate of the machine The values for $move_r$ and $move_s$ are based on the time to execute a small, 4-instruction loop that moves objects in word-size chunks Finally, the so-called "fudge factor", F, which was introduced in [Shap86], is used to calculate various values that are small increments of other values For example, a hash table for R is assumed to occupy $F P_r$ pages in memory when overhead is included

It is important to note that there is only one I/O parameter, that is, we do not distinguish between sequential I/O and random I/O The impact of sequential I/O on full joins is addressed in [Shek90] It turns out that the conclusions drawn from the analysis are not altered if sequential I/O is considered

4.3. Analysis of Large Joins

In this section, full joins between R and S are analyzed As mentioned earlier, index-nested-loops is not considered for such joins Due to space limitations, we\can only briefly touch on the analysis of the standard sort-merge and hybrid-hash algorithms For more details, the reader should consult [Shap86]

4 3.1. Standard Sort-Merge

Assuming enough memory is available, the cost breakdown of the two-pass sort-merge algorithm that we described earlier is [Shap86]

- read R and S $(P_r + P_s)\ IO$
- sort R $|R| \log_2 |R| (compare + swap_r)$
- sort S $|S| \log_2 |S| (compare + swap_s)$

Parameter	Definition
Mips	instructions execution rate
P	size of a disk page
IO	time to read/write a disk page
M	number of memory buffer pages
b	B+ tree fanout
$\|R\|$	number of objects in R
k	sharing level
sel	selectivity of the predicate on R
r	size of objects in R
s	size of objects in S
$compare_i$	instructions to compare two values
$hash_i$	instructions to hash a join attribute or PID
$bhash_i$	instructions to hash a value for bit filtering
F	fudge factor for hybrid-hash [Shap86]
compare	time to compare two values $compare = compare_i / Mips$
hash	time to hash a value $hash = hash_i / Mips$
bhash	time to hash a value for bit filtering $bhash = bhash_i / Mips$
$\|S\|$	number of objects in S $\|S\| = \|R\| / k$
O_r	objects per page in R $O_r = \lfloor P / r \rfloor$
O_s	objects per page in S $O_s = \lfloor P / s \rfloor$
P_r	pages in R $P_r = \lceil \|R\| / O_r \rceil$
P_s	pages in S $P_s = \lceil \|S\| / O_s \rceil$
$move_r$	time to move an object in R $move_r = 4\lceil r / 4 \rceil / Mips$
$move_s$	time to move an object in S $move_s = 4\lceil s / 4 \rceil / Mips$
$swap_r$	time to swap two objects in R $swap_r = 3\ move_r$
$swap_s$	time to swap two objects in S $swap_s = 3\ move_s$

Table 1 Parameters Used in the Analysis

Defaults for Core Parameters	
Mips	10 (in millions of instructions per sec)
IO	20 milliseconds (fixed)
P	4096 bytes (fixed)
M	256 pages (or 1 Mbyte)
b	350 (fixed)
$\|R\|$	100,000 objects
k	1
sel	0 01
r	200 bytes
s	200 bytes
$compare_i$	2 instructions (fixed)
$hash_i$	9 instructions (fixed)
$bhash_i$	3 instructions (fixed)
F	1 2 (fixed)

Table 2 Defaults for the Core Parameters

- read and write the output runs for R and S $(P_r + P_s)\ 2\ IO$
- perform the final merge $(|R| + |S|)\ compare$
- I/O savings if extra memory is available

$$-min(P_s + P_r,\ M - \sqrt{(P_r + P_s) / 2})\ 2\ IO$$

In the final term, the value $\sqrt{(P_r + P_s) / 2}$ represents the minimal memory requirements of the algorithm [Shek90]

Note that in the above analysis we have excluded the cost to write the result of the join to disk This cost will be excluded throughout the analysis We have excluded it because the result of a join often forms the input of another database operation without ever being completely written to disk, moreover, this cost would be the same for all algorithms, anyway Finally, note that the terms above group related costs in the algorithm, and their order does not coincide with the actual step-by-step execution of the algorithm This same approach will be used throughout the analysis

4.3.2. Standard Hybrid-Hash

To analyze the cost of the hybrid-hash algorithm, recall that it begins by dividing R into $B + 1$ partitions $R_0, R_1, \ldots R_B$, and likewise for S The proper value of B to make everything work is [Shap86] $B = \lceil (F\ P_r - M) / (M - 1) \rceil$

When R is partitioned, $M - B$ pages of memory are allocated to build the hash table for R_0 The remainder of memory is allocated to serve as output buffers for $R_1, R_2, \ldots R_B$, with one page allocated per partition S is then partitioned in a similar fashion, except that S_0 is joined with R_0 as it is being partitioned Letting q equal the fraction of R represented by R_0, that is, $q = (M - B) / (F\ P_r)$, the cost of the algorithm is [Shap86]

- read R and S $(P_r + P_s)\ IO$
- partition R and S $(|R| + |S|)\ hash$
- populate partition R_i for $1 \le i \le B$ $|R|\ (1 - q)\ move_r$
- populate partition S_i for $1 \le i \le B$ $|S|\ (1 - q)\ move_s$
- read and write partitions R_i and S_i for $1 \le i \le B$ $(P_r + P_s)\ (1 - q)\ 2\ IO$
- hash the objects in each R_i, probe with S_i for $1 \le i \le B$ $(|R| + |S|)\ (1 - q)\ hash$
- build the hash table for R_i, $0 \le i \le B$ $|R|\ move_r$
- probe the hash table for each object in S $|S|\ compare\ F$

Note that here we have assumed that $P_r < P_s$, otherwise the roles of R and S should be exchanged in the above analysis

4.3.3. Pointer-Based Nested-Loops

In the pointer-based nested-loops algorithm, one page of memory is allocated to read R and the remainder are allocated to read S Because of the way memory is allocated, and because R and S are relatively unclustered, this causes S to be accessed in exactly the same manner as it would be in an unclustered index scan of S with a buffer of size $M - 1$ The I/O cost of performing an unclustered index scan was derived in [Mack89] Making use of the equations in that paper, the function $U_s(x, B)$, which counts the number of I/Os resulting from x unclustered references to S when a buffer of B pages is used, is defined as

$$let\ q = (|S| - O_s) / |S|\ \ and\ let\ p = 1 - q$$

$$let\ n = max\left\{ j \text{ in } \{0, 1, \ldots, x\} \text{ such that } P_s\ (1 - q^j) \le B \right\}$$

$$if\ x \le n\ \ then\ \ U_s(x, B) = P_s\ (1 - q^x)$$

$$else\ \ U_s(x, B) = P_s\ (1 - q^n) + (x - n)\ P_s\ p\ q^n$$

Using $U_s()$, the cost of accessing S is therefore $U_s(|R|, M - 1)\ IO$ We have defined $U_s()$ in this manner because it will be needed again later

In addition to the cost of accessing S, there is also the cost to read R, and the CPU cost to check whether the page referenced by a given object in R is in memory, which we assume is done with hashing The net cost of the algorithm is therefore

$$P_r\ IO + |R|\ hash + U_s(|R|, M-1)\ IO$$

4.3.4. Pointer-Based Sort-Merge

The analysis of the pointer-based sort-merge algorithm is similar to the analysis of the standard sort-merge algorithm The key difference here, of course, is that only R is sorted Based on the analysis of the standard sort-merge algorithm, the cost of the algorithm is

- read R and S $(P_r + P_s)\ IO$
- sort R $|R| \log_2 |R|\ (compare + swap_r)$
- read and write the output runs for R $P_r\ 2\ IO$
- I/O savings if extra memory is available

$$-min(P_r, M - \sqrt{P_r / 2})\ 2\ IO$$

Note that here the I/O savings is a function of $\sqrt{P_r / 2}$ Less memory is required than in the standard sort-merge algorithm because only R is sorted

4.3.5. Pointer-Based Hybrid-Hash

As one would expect, the analysis of the pointer-based hybrid-hash algorithm is similar to the analysis of the standard hybrid-hash algorithm The main difference here is that only R is partitioned Based on the analysis of the standard hybrid-hash algorithm, the cost of the algorithm is

- read R and S $(P_r + P_s)\ IO$
- partition R $|R|\ hash$
- populate partition R_i for $1 \le i \le B$ $|R|\ (1-q)\ move_r$
- read and write partition R_i for $1 \le i \le B$ $P_r\ (1-q)\ 2\ IO$
- hash the objects in each R_i for $1 \le i \le B$ $|R|\ (1-q)\ hash$
- build the hash table for each R_i, $0 \le i \le B$ $|R|\ move_r$

4.4. Analysis of Small to Medium-Sized Joins

In this section, we analyze small to medium-sized joins, where the result of a selection on R is joined to S As mentioned earlier, the analysis will assume that B+ tree indexes exist on both R and S The index on R will be used to evaluate the selection predicate, while the index on S, which is assumed to be a unique primary-key index, will be used by the index-nested-loops algorithm to avoid a file scan of S

The only index clustering combination that will be analyzed in this section is the combination where the index on R is a clustered index and the index on S is an unclustered index We will denote this combination as clustered/unclustered, and similarly for the other combinations The clustered/unclustered index combination has been chosen here because it is the most difficult to analyze The analysis of other index combinations can be found in [Shek90] Note that we will still present results for the unclustered/clustered index combination, as well comment on the results for the two remaining index combinations

Another thing to note is that we will ignore the small amount of memory space that is required to read the indexes on R and S A small, two-page MRU buffer would be sufficient for reading either of these indexes in the examples that are analyzed, and accounting for that little space would not change the results in any significant way In addition, we will assume that the selectivity of the predicate on R is such that R', which is defined as the result of the selection on R, fits completely in memory In contrast to the indexes on R and S, we will account for the memory space used by R'

Finally, to simplify the equations of this section, we let $|R'|$ denote the cardinality of R' and P_r' denote the number of pages in R', that is, $|R'| = sel\ |R|$ and $P_r' = sel\ P_r$

4.4.1. Standard Index-Nested-Loops

The index-nested-loops algorithm begins by using the B+ tree index on R to obtain R' This is done by descending the index to a leaf, and then scanning across the leaves to obtain the objects in R that satisfy the selection predicate In all the examples that are analyzed, the height of the index on R is equal to 2, and similarly for S The cost to read the index on R is therefore

- descend the index on R $2\ IO$
- CPU cost to descend the index $2 \log_2 b\ compare$
- scan across the index leaves $\lceil |R'| / b - 1 \rceil\ IO$

R' is read into memory using the index and then sorted by foreign key (i e , by join attribute) This costs

- read R' $P_r'\ IO$
- extract R' from R in memory before sorting $|R'|\ move_r$
- sort R' $|R'| \log_2 |R'|\ (compare + swap_r)$

After R' has been sorted, the foreign key in each object of R' is used to probe the index on S Assuming that the height of the index is 2, the I/O cost to probe the index consists of the cost to read the root page of the index plus the cost of reading whatever leaf pages of the index are accessed by the index probes Because R' is sorted by foreign key, the leaf pages will be accessed in ascending key order, and consequently a simple two-page MRU buffer group will ensure that no leaf page is read more than once All that remains, therefore, is to determine the number of leaf pages that are accessed by the index probes

To determine the number of leaf pages that are accessed by the index probes, we first consider the probability that a particular leaf page L_i is *not* accessed Since R and S are relatively unclustered, we can assume that any particular leaf page of the index is just as likely to be accessed as any other leaf page Therefore, the probability that a leaf page L_i is not accessed is equal to the probability of choosing a subset of $|R'|$ objects from R such that the chosen subset contains no object with a foreign key in L_i Since each leaf page contains b keys, and since there are k objects in R that share each key value, the probability that L_i is not accessed is

$$\binom{|R| - b\ k}{|R'|} / \binom{|R|}{|R'|}$$

Since there are a total of $\lceil |S| / b \rceil$ leaf pages, the expected number of leaf pages that are accessed by the index probes is therefore

$$\lceil |S| / b \rceil \left[1 - \binom{|R| - b\ k}{|R'|} / \binom{|R|}{|R'|}\right]$$

This same quantity was derived in another context [Yao77] If we let $Y(\ )$ denote the so-called Yao function

$$Y(u, v, w) = 1 - \binom{u-v}{w} / \binom{u}{w}$$

then the total cost to probe the index on S is

- read the root page IO
- probe the index for each object in R' $2\,|R'|\log_2 b$ *compare*
- access the leaf pages $\lceil |S| / b \rceil\, Y(|R|, b\,k, |R'|)\, IO$

The final cost that needs to be included is the cost to read S Here, we assume that S is read using all of the available memory that is not allocated to hold R' In effect, this means that S ends up being read in the same manner as in the pointer-based nested-loops algorithm, except that here index pointers take the place of object pointers from R, in addition, S is referenced $|R'|$ times now and there are only $M - P_r'$ memory pages available to read S Based on these observations and using the I/O function $U_s(\,)$ that was defined earlier, the cost to read S is simply

- check memory for each index reference to S $|R'|$ *hash*
- access S $U_s(|R'|, M - P_r')\, IO$

Before going on to the analysis of the sort-merge algorithm, it is important to point out that we also analyzed the normal index-nested-loop algorithm where R' is not sorted In all of the examples that we will consider, it performed considerably worse than the algorithm that we have described due to the cost of also accessing the leaf pages of the index on S in a random fashion

4.4 2. Standard Sort-Merge

The sort-merge algorithm begins by reading R' into memory and sorting it, just as in the index-nested-loops algorithm In addition, each object in R' is also hashed to turn on a bit in the bit filter In the examples that are analyzed, a simple page-sized bit filter turns out to be sufficient to filter out virtually all of the non-participating objects in S (Details on how to design and calculate the effectiveness of a bit filter can be found in [Seve76])

After R' has been sorted, S is read into memory, filtered, and the resulting objects are sorted As mentioned, we are assuming that virtually all of the non-participating objects in S are screened out by the bit filter Consequently, only the objects in S that join with R' are sorted Let S' denote these objects, and let $|S'|$ and P_s' denote the cardinality and the number of pages in S', respectively To determine $|S'|$, the same sort of analysis that was used to determine the number of index leaf pages accessed by the index-nested-loops algorithm can be applied The analysis proceeds by considering the probability that no object in R' joins with a given object in S Since each object in S joins with k objects in R, a similar application of the previous analysis yields

$$|S'| = |S|\, Y(|R|, k, |R'|)$$

The corresponding value for P_s' is simply $P_s' = |S'| / O_s$ Based on the preceding analysis and the analysis of the standard sort-merge algorithm for full joins, the cost of the algorithm is

- descend the index on R $2\,IO + 2\log_2 b$ *compare*
- scan across the index leaves $\lceil |R'| / b - 1 \rceil\, IO$
- read R' and S $(P_r' + P_s)\, IO$
- bit filter R' and S $(|R'| + |S|)$ *bhash*
- extract R' from R before sorting $|R'|$ *move*$_r$
- extract S' from S before sorting $|S'|$ *move*$_s$
- sort R' $|R'|\log_2|R'|$ (*compare* + *swap*$_r$)
- sort S' $|S'|\log_2|S'|$ (*compare* + *swap*$_s$)
- read and write the output runs for R' and S' $(P_r' + P_s')\, 2\, IO$
- perform the final merge $(|R'| + |S'|)$ *compare*
- I/O savings if extra memory is available

  $-min(P_s' + P_r', M - \sqrt{(P_r' + P_s') / 2})\, 2\, IO$

4.4.3. Standard Hybrid-Hash

The analysis of the hybrid-hash algorithm follows from the analysis that was used earlier on full joins With S' defined the same as in the preceding analysis, and assuming that R' fits in memory, the cost of the algorithm is

- descend the index on R $2\,IO + 2\log_2 b$ *compare*
- scan across the index leaves $\lceil |R'| / b - 1 \rceil\, IO$
- read R' and S $(P_r' + P_s)\, IO$
- bit filter R' and S $(|R'| + |S|)$ *bhash*
- partition R' and S' $(|R'| + |S'|)$ *hash*
- build the hash table for R' $|R'|$ *move*$_r$
- probe the hash table for each object in S' $|S'|$ *compare* F

4.4.4. Pointer-Based Nested-Loops

The analysis of the pointer-based nested-loops algorithm follows directly from the analysis that was used earlier on full joins Based on that analysis, the cost of the algorithm is

- descend the index on R $2\,IO + 2\log_2 b$ *compare*
- scan across the leaves of the index on R $\lceil |R'| / b - 1 \rceil\, IO$
- read R' $P_r'\, IO$
- check memory for each pointer reference in R' $|R'|$ *hash*
- access S $U_s(|R'|, M - 1)\, IO$

4.4.5. Pointer-Based Sort-Merge

In order to analyze the pointer-based sort-merge algorithm, we need to estimate how many pages in S participate in the join Let P_s' denote number of pages in S that participate in the join (Note that P_s' as defined here is not the same as the P_s' that was defined in the standard algorithms) To determine P_s', the analysis that was used to determine the number of index leaf pages accessed by the index-nested-loops algorithm can be applied once again The analysis proceeds by considering the probability that no object in R' joins with a given page in S Since each page in S joins with k O_s objects in R, a similar application of the previous analysis yields

$$P_s' = P_s\, Y(|R|, k\,O_s, |R'|)$$

With the exception of P_s', the rest of the analysis is straightforward and follows from the analysis that was used earlier on full joins Assuming that R' fits in memory, the cost of the algorithm is

- descend the index on R $2\,IO + 2\log_2 b$ *compare*
- scan across the index leaves $\lceil |R'| / b - 1 \rceil\, IO$
- read R' and S' $(P_r' + P_s')\, IO$
- extract R' from R before sorting $|R'|$ *move*$_r$
- sort R' $|R'|\log_2|R'|$ (*compare* + *swap*$_r$)

4.4.6. Pointer-Based Hybrid-Hash

The analysis of the hybrid-hash algorithm follows directly from the analysis that was used earlier on full joins With P_s' defined the same as in the preceding analysis, and assuming that R' fits in memory, the cost of the algorithm is

- descend the index on R $2\,IO + 2\log_2 b$ *compare*
- scan across the index leaves $\lceil |R'| / b - 1 \rceil\, IO$
- read R′ and S′ $(P_r' + P_s')\, IO$
- hash R′ $|R'|$ *hash*
- build the hash table for R′ $|R'|$ $move_r$

4.5. Performance Results for Large Joins

The results for full joins between R and S are presented in Graphs A-D The graphs were obtained by using the equations from the analysis to compute the total time in seconds to run each join algorithm for a particular memory size The size of memory was increased in 1/2 Mbyte increments and ranged from the minimal size required to execute all the join algorithms, which was approximately 1/4 Mbytes, all the way up to 10 Mbytes In all of the graphs, the lines for the standard, non-pointer-based algorithms are labeled wo/ptr for "without pointer", while the lines for the pointer-based algorithms are labeled w/ptr for "with pointer" The number of objects in R was fixed at 100,000 in all the graphs, and the number of objects in S was varied by changing k, the sharing level (Recall that $|R| = k\,|S|$)

Graph A shows the performance of the join algorithms when R and S are the same size (in pages) In that graph, both R and S consist of 100,000 objects with 200 bytes per object As shown, the pointer-based algorithms can provide significant performance gains in this situation Compared to the standard hybrid-hash algorithm, the pointer-based hybrid-hash algorithm reduces the join time by approximately 30% over the whole range of memory sizes considered This, of course, is because S is not partitioned in the pointer-based hybrid-hash algorithm A similar relationship is seen between the standard sort-merge algorithm and the pointer-based sort-merge algorithm

One of the interesting things to notice about Graph A is just how poorly the pointer-based nested-loops algorithm performs As mentioned earlier, this is because it does not try to optimize its disk reads of S like the other pointer-based algorithms These results demonstrate that, even with pointers, something more intelligent than a nested-loops approach (or naive pointer traversal) is often needed for high performance

Graph B shows the performance of the join algorithms when R is ten times the size of S In this case, the pointer-based hybrid-hash algorithm takes three times longer to execute than the standard hybrid-hash algorithm when the size of memory is 2 5 Mbytes, which is the amount of memory that would presumably be allocated by a query optimizer to execute the standard hybrid-hash algorithm This large difference in performance is due to the fact that the pointer-based hybrid-hash algorithm always chooses R as its inner set, which in this case is a poor choice because R is so much bigger than S Also note that the pointer-based nested-loops algorithm performs as well as or better than the standard hybrid-hash algorithm when the size of memory is 2 Mbytes or more This is because all of S actually fits in memory in that case

Finally, Graphs C and D further demonstrate how the relative performance of the pointer-based algorithms depends on the size ratio of R and S If R is roughly the same size as S or smaller, then the pointer-based algorithms always perform better Otherwise, the standard algorithms perform better, unless there is enough memory available to hold S, in which case the pointer-based nested-loops algorithm performs as well as the standard hybrid-hash algorithm

Graph C shows the performance of the algorithms when R is one fifth the size of S The graph has roughly the same shape as Graph A, but in this case, the pointer-based hybrid-hash and sort-merge algorithms outperform their standard counterparts by up to 55% This is because the relative cost to partition or sort S in the standard algorithms is larger in this case due to its increased size

Graph D shows the performance of the algorithms when R is twice the size of S This is perhaps the most interesting of all the graphs because of the way the lines for the pointer-based and standard hybrid-hash algorithms cross each other The reason for the crossover is because the pointer-based hybrid-hash algorithm always chooses R as its inner set, which is a poor choice in this case because R is larger than S However, even with R as the inner set, the pointer-based hybrid-hash algorithm performs better initially because it does not have to partition S But with 6 Mbytes or more of memory, enough of S fits in memory so that the savings from not partitioning S are insufficient to offset the added costs of using R as the inner set, at that point, the standard hybrid-hash algorithm starts to perform better

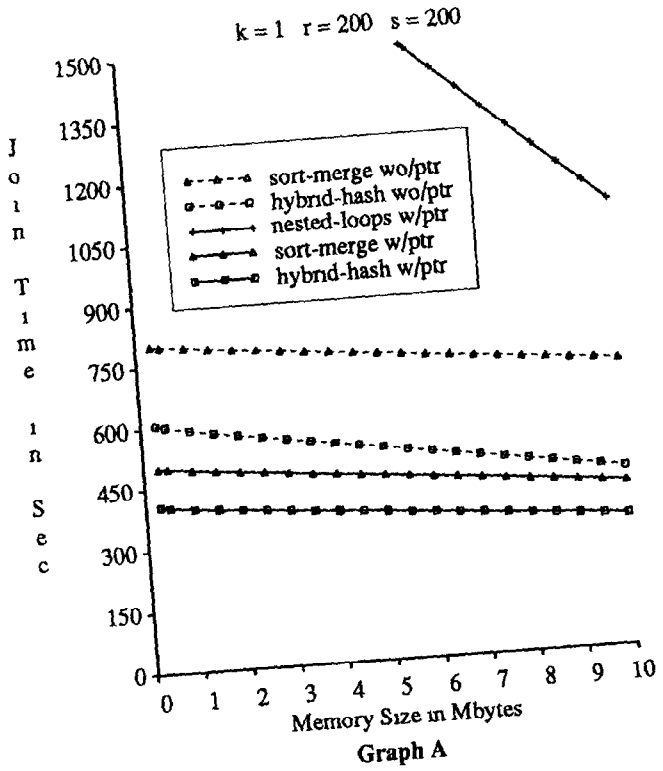
Finally, it is important to note that, on the joins in which they perform the best, the CPU times of the pointer-based join algorithms are significantly less than that of their standard counterparts (This is true for the small to medium-sized joins as well) Therefore, in a CPU bound system, the pointer-based algorithms could improve performance by even larger margins than those suggested by Graphs A and C

4.6. Performance Results for Medium-Sized Joins

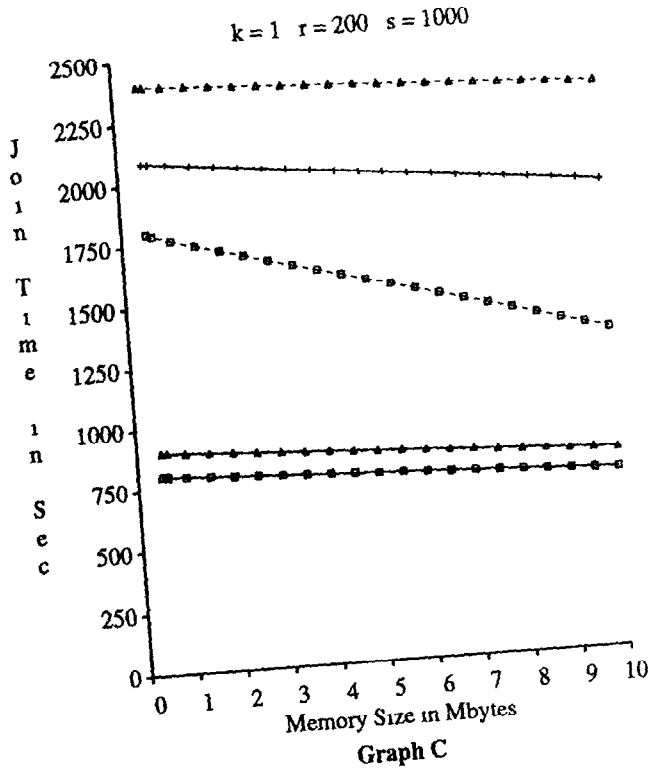
The results for medium-sized joins are presented in Graphs E-H In all of these graphs, the size of R was kept fixed at 100,000 objects, and the selectivity of the predicate on R was kept fixed at 0 01 Consequently, each graph represents the join of 1,000 objects in R with S As shown, the size of memory was increased in 1/8 Mbyte increments and ranged from the minimal size required to hold the selection on R in memory, which was approximately 1/4 Mbytes, all the way up to 2 Mbytes In all of the graphs, the label "C/U" indicates that the clustered/unclustered index combination was used, while the label "U/C" indicates that the unclustered/clustered index combination was used

Graphs E and F are for the clustered/unclustered index combination, where the index on R is a clustered index and the index on S is an unclustered index These graphs represent the situation where the pointer-based algorithms perform their best in relation to the index-nested-loops algorithm, since the relative cost of accessing S via its index is high in this case

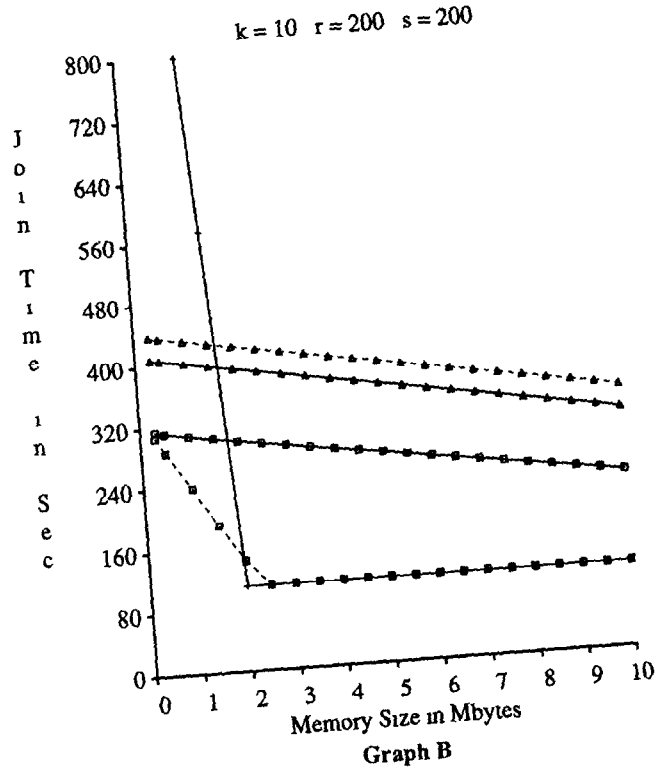
Graph E shows the performance of the join algorithms when R is the same size as S As shown, compared to the index-nested-loops algorithm, the pointer-based algorithms reduce the join time by approximately 30% over the whole range of memory sizes considered This, of course, is because the index on S is not read by the pointer-based algorithms Note that the standard sort-merge and hybrid-hash algorithms perform poorly here because, unlike their pointer-based counterparts, they read all of S, even though only a small fraction of S actually participates in



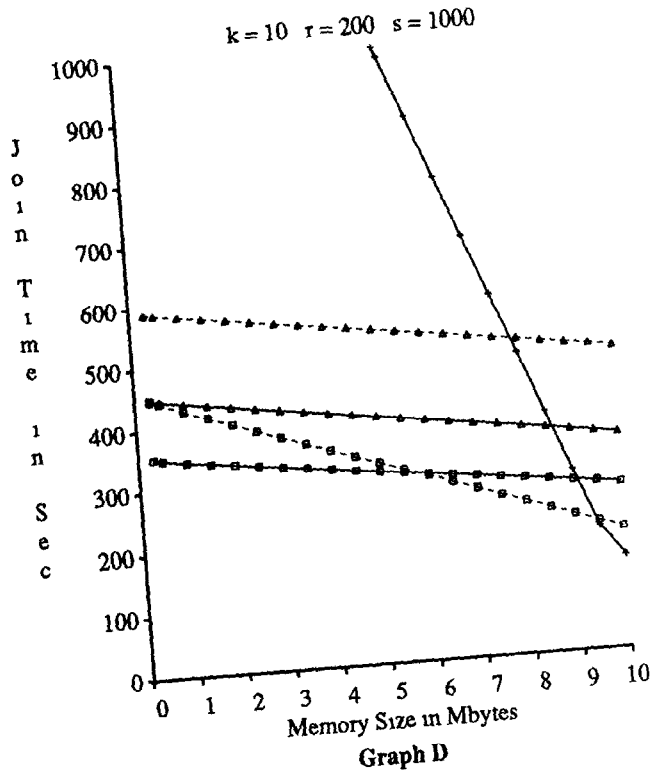
Graph A



Graph C



Graph B



Graph D

the join

Graph F shows the performance of the join algorithms when R is ten times the size of S In this case, the fraction of S that participates in the join is large enough so that both the index-nested-loops algorithm and the pointer-based nested-loops algorithm perform poorly in relation to the other join algorithms This is because neither of these algorithms try to optimize their disk reads of S Compared to the standard hybrid-hash algorithm, the pointer-based hybrid-hash algorithm reduces the join time by approximately 15% over the whole range of memory sizes considered A similar relationship is seen between the standard sort-merge algorithm and the pointer-based sort-merge algorithm

Graphs G and H are for the same cases as Graphs E and F but with the unclustered/clustered index combination It should be clear that these graphs represent the situation where the pointer-based algorithms perform their worst in relation to the index-nested-loops algorithm This is because the relative cost of accessing S via its index is low in this case As shown, the pointer-based algorithms perform approximately 15% better than the index-nested-loops algorithm in Graph G and approximately 5% better in Graph H

Note that the graphs for the clustered/clustered and unclustered/unclustered index combinations have not been presented here Although those graphs do not follow quite the same patterns as those in Graphs E-H, similar benefits and tradeoffs were observed, the pointer-based algorithms always outperformed their standard counterparts

4.7. Performance Results for Small Joins

The results for small joins are presented in Graphs I-L Only the index-nested-loops algorithm and the pointer-based nested-loops algorithm are compared here because they perform as well as or better than the other algorithms in their respective classes on small joins In all of the graphs, the size of memory was kept fixed at 1/4 Mbytes

As indicated, Graphs I-L have been plotted in percentage terms The total time to execute the join using the index-nested-loops algorithm was computed and the percentage difference between that time and the time to execute the pointer-based nested-loops algorithm was then plotted The reason for displaying the results in this manner is to make the performance differences between the two algorithms clearer This was not done for the previous results because they were not uniform enough to make this a viable approach

Graphs I and J are for the clustered/unclustered index combination Graph I shows the performance of the join algorithms when R is the same size as S As shown, the pointer-based nested-loops algorithm reduces the join time by almost 50% when more than 10 objects in R are joined with S In this case, the cost to execute the index-nested-loops algorithm is dominated by the I/O cost to read the index on S and S itself, with roughly the same number of pages being read in each instance Therefore, the cost to read the index on S accounts for about one half of the cost to execute the index-nested-loops algorithm Since the pointer-based nested-loops algorithm eliminates the cost to read the index on S, it effectively reduces the cost of the join by 50%

Graph J shows the performance of the join algorithms when R is ten times the size of S As shown, the performance benefit of the pointer-based nested-loops algorithm decreases as the size of the join increases In this case, the index on S is small enough so that some of its leaf pages end up being referenced more than once by the index probes of the index-nested-loops algorithm (Recall that in the index-nested-loops algorithm, a given leaf page L_i may be referenced several times, but because the result of the selection on R is sorted by the join attribute, L_i is guaranteed to be read from disk only once even if it is referenced several times) In contrast to its index, however, S is still large enough in this case so that each probe of S causes a different page in S to be accessed The net effect is that as the size of the join increases, the proportional cost of the index probes in the index-nested-loops algorithm decreases, this in turn causes the performance benefit of the pointer-based nested-loops algorithm to decrease somewhat

Finally, Graphs K and L are for the unclustered/clustered index combination In this case, the cost to execute the index-nested-loops algorithm is dominated by the I/O cost to read R, the index on S, and S itself, with roughly the same number of pages being read in each instance Therefore, the cost to read the index on S accounts for about one third of the cost to execute the index-nested-loops algorithm Since the pointer-based nested-loops algorithm eliminates the cost to read the index on S, it effectively reduces the cost of the join by 33% The line for the pointer-based nested-loops algorithm has an upward slope in Graph L for the same reason it has an upward slope in Graph J

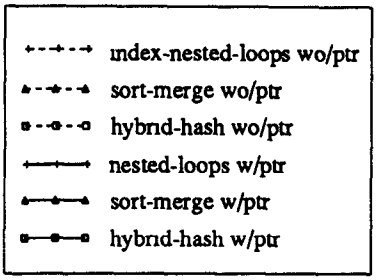
Once again, the graphs for the clustered/clustered and unclustered/unclustered index combinations have not been presented here Although the graphs for those index combinations do not follow quite the same patterns as those in Graphs I-L, similar benefits and tradeoffs were observed, the pointer-based algorithms always outperformed their standard counterparts by 30% or more

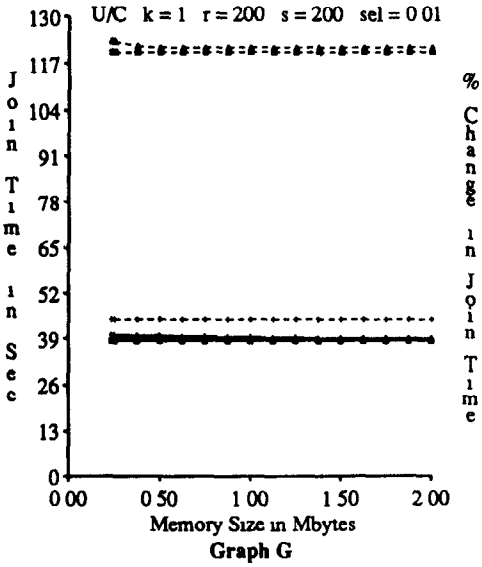
5. RELATED WORK

Clearly, our work is related to the previous studies that have looked at the performance of different join algorithms, particularly [Blas77] and [Shap86] In [Blas77], the nested-loops and sort-merge algorithms were described and analyzed, while in [Shap86], the hybrid-hash algorithm was analyzed and shown to be generally superior to the sort-merge as well as the Grace and simple-hash join algorithms Both of these papers were concerned with joins in a purely relational context, however, and consequently neither considered pointer-based joins

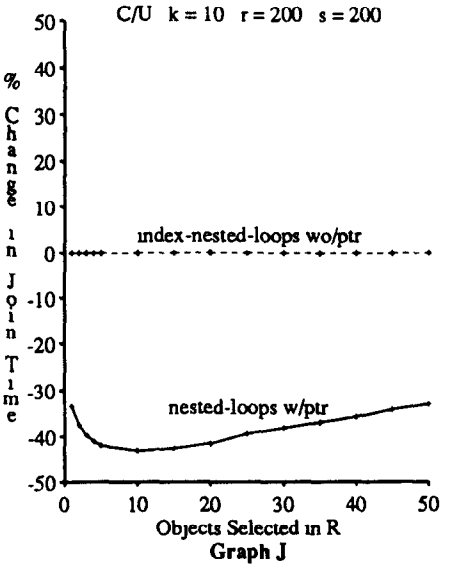
Perhaps the work that is most closely related to ours is that of [Vald87] In that paper, auxiliary data structures called *join indices* were described as a way to speedup join processing If, for example, the join of relations R and S was frequently needed, then a join index for R and S would be maintained The join index would consist of pairs of pointers, matching each record in R with the record(s) it joins with in S, and it would effectively implement the pre-computed join of R and S

In [Vald87], joins using a join index were shown to be superior to hybrid-hash joins in many situations For the many-to-one types of joins we have analyzed here, however, it should be clear that our pointer-based algorithms will always outperform a join

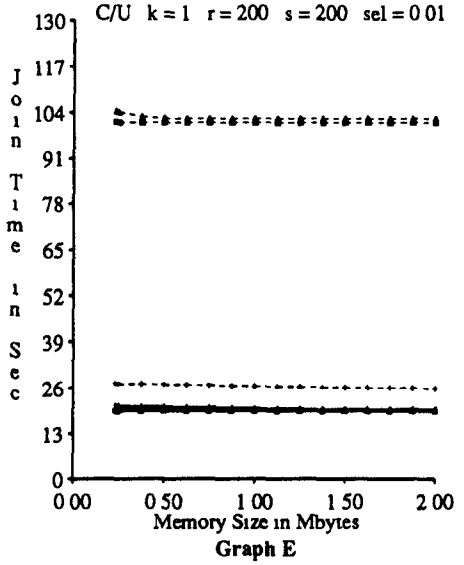




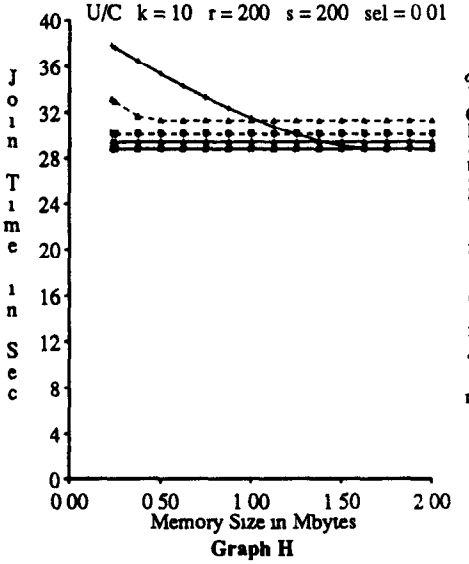
Graph G



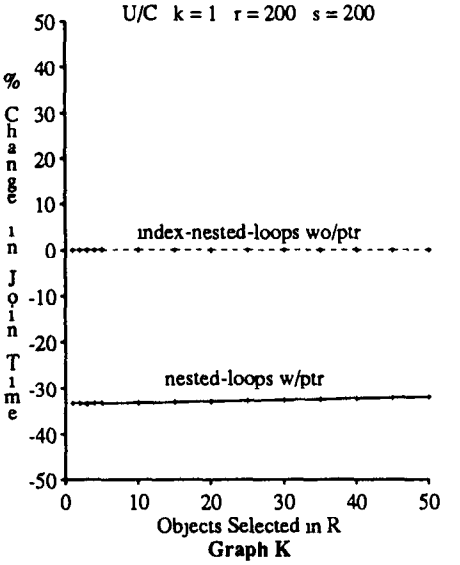
Graph J



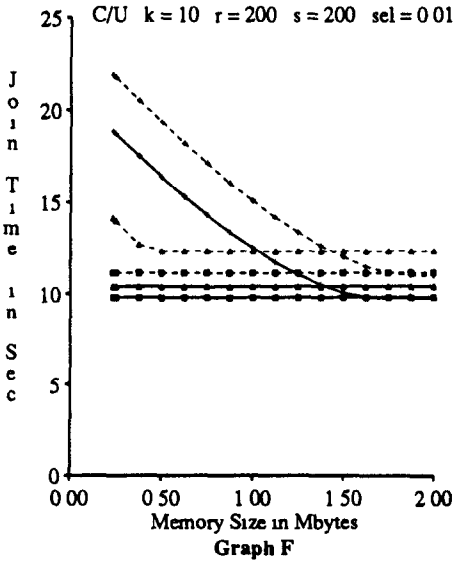
Graph E



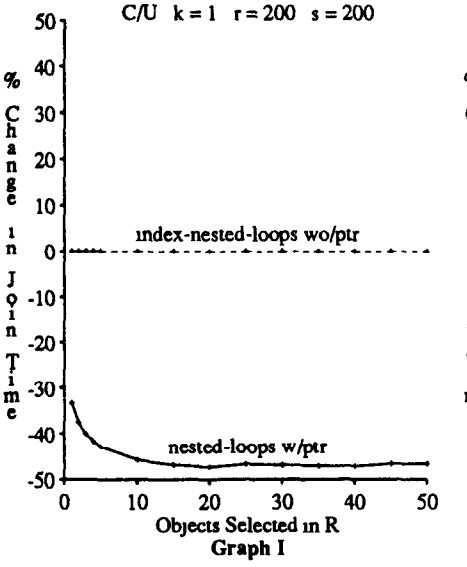
Graph H



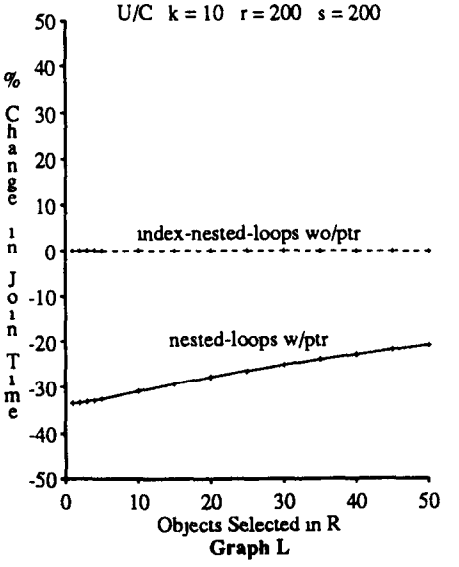
Graph K



Graph F



Graph I



Graph L

index This is because the embedded pointers we have assumed effectively implement a join index without the overhead of accessing (and maintaining) an auxiliary data structure

6 CONCLUSION

This paper described three pointer-based join algorithms that are simple variants of the standard nested-loops, sort-merge, and hybrid-hash join algorithms used in relational database systems We showed that, given the appropriate pointer structures, some common types of joins in relational and object-oriented database systems can be executed using our pointer-based algorithms

For much of the paper, an analysis was carried out to compare the pointer-based algorithms to their standard, non-pointer-based counterparts In addition to providing a basis for comparison, the cost equations that were derived in the analysis can also be used in query optimization and physical database design In the analysis, the join of two sets R and S in a many-to-one relationship was studied A simple pointer structure was assumed, where each object in R contained a pointer to its related object in S Two types of joins were analyzed full joins of R and S, and small to medium-sized joins with a selection predicate on R

For full joins, the results of the analysis showed that the pointer-based sort-merge and hybrid-hash join algorithms can provide savings of 30% or more when the size of R (in pages) is roughly the same size as S or smaller These results are due to the fact that S does not need to be sorted or partitioned in the pointer-based algorithms The results also showed, however, that the pointer-based algorithms do not perform as well as the standard hybrid-hash algorithm when the size of R is significantly larger than the size of S Moreover, the pointer-based nested-loops join algorithm was shown to perform very poorly in almost all cases These negative results are important because they show that it is unwise for object-oriented database systems to support only pointer-based join algorithms They also show that to make effective use of pointers, something more intelligent than a nested-loops approach (i e , naive pointer traversal) is needed for high performance on large joins

For medium-sized joins, where 1% of the objects in R were joined with S, the results showed that the pointer-based sort-merge and hybrid-hash algorithms always outperformed their standard counterparts, providing gains of up to 30% in many cases The pointer-based nested-loops algorithm was again shown to perform poorly, although not in all cases Finally, for small joins, where 0 01% of the objects in R were joined with S, all of the pointer-based join algorithms outperformed their standard counterparts by 30% or more Both of these results are largely due to the fact that index lookups on S are eliminated in the pointer-based algorithms The conclusion to be drawn here is that, for small to medium-sized joins, which are probably a very common type of join, pointer-based join algorithms can provide significant performance gains

As far as implementation results go, we are currently in the process of obtaining experimental results for some of the join algorithms described here using the incremental join facility of Starburst [Care90, Haas90] The experiments being conducted, which are described in [Care90], compare a variety of pointer-based join algorithms to their standard counterparts under three different clustering strategies

REFERENCES

[Blas77] M Blasgen and K Eswaran, "Storage and Access in Relational Databases," *IBM Syst Journal*, 16(4), 1977

[Care88] M Carey et al , "A Data Model and Query Language for EXODUS," *Proc of the 1988 ACM-SIGMOD Conf*, Chicago, IL, 1988

[Care89] M Carey et al, "The EXODUS Extensible DBMS Project An Overview," in *Readings in Object-Oriented Databases*, S Zdonik and D Maier, eds , Morgan-Kaufman Publ Co , 1989

[Care90] M Carey et al, "An Incremental Join Attachment for Starburst," submitted for publication

[Chan82] A Chan, "Storage and Access Structures to Support a Semantic Data Model," *Proc of the 1982 VLDB Conf*, Mexico City, Mexico, 1982

[Dewi85] D DeWitt and R Gerber, "Multiprocessor Hash-Based Join Algorithms," *Proc of the 1985 VLDB Conf*, Stockholm, Sweden, Aug 1985

[Deaux90] O Deux et al, "The Story of O2," *IEEE Trans on Knowledge and Data Eng* , March 1990

[Haas90] L Haas et al, "Starburst Mid-Flight As the Dust Clears," *IEEE Trans on Knowledge and Data Engineering*, March 1990

[Kim89] W Kim, "A Model of Queries for Object-Oriented Databases," *Proc of the 1989 VLDB Conf*, Amsterdam, The Netherlands, Aug 1989

[Mack86] L Mackert and G Lohman, "R* Optimizer Validation and Performance Evaluation for Local Queries," *Proc of the ACM-SIGMOD Conf*, Washington, D C, 1986

[Mack89] L Mackert and G Lohman, "Index Scans Using a Finite LRU Buffer A Validated I/O Model," *ACM Trans on Database Systems* 14(3), Sept 1989

[Seve76] D Severance and G Lohman, "Differential Files Their Application to the Maintenance of Large Databases," *ACM Trans on Database Systems* 1(3), Sept 1976

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

[Shek90] E Shekita and M Carey, "A Performance Evaluation of Pointer-Based Joins," Univ of Wisconsin Tech Report #916, March 1990

[Ship81] D Shipman, "The Functional Data Model and the Data Language DAPLEX," *ACM Trans on Database Sys* 6(1), Sept 1987

[Vald87] P Valduriez, "Join Indices," *ACM Trans on Database Systems* 12(2), June 1987

[Vele89] F Velez et al, "The O2 Object Manager An Overview," *Proc 1989 VLDB Conf*, Amsterdam, The Netherlands, Aug 1989

[Yao77] S Yao, "Approximating Block Accesses in Database Organizations," *Comm of the ACM* 20(4), April 1977

[Zani83] C Zaniolo, "The Database Language GEM," *Proc of the ACM-SIGMOD Conf*, San Jose, CA, 1983