Pointer-based Join Techniques for Object-Oriented Databases

Daniel F. Lieuwen
David J. DeWitt
Manish Mehta

Technical Report #1099

July 1992
Pointer-based Join Techniques for Object-Oriented Databases

Daniel F. Lieuwen
David J. DeWitt
Manish Mehta

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

Abstract

In this paper, we describe and analyze four parallel pointer-based joins for set-valued attributes. These joins will be common in next-generation object-oriented database systems, so efficiently supporting them is crucial to the performance of such systems. Pointer-based join algorithms based on Hybrid-hash provide good performance, but algorithms that require less replication will often produce as good or better performance, especially if each set-valued attribute references a small number of nodes.

1. Introduction

Set-valued attributes are an important feature of next-generation database systems, both for Database Programming Languages (DBPLs) and Object-Oriented Database Systems (OODBSs). Examples of systems with data-modeling facilities for set-valued attributes include Bubba [BORA90], E [RICH92], GemStone [BUTT91], Iris [FISH87], LDL [CHIM90], O++ [AGRA89], ObjectStore [LAMB91], ORION [KIM90], and O₂ [DEUX91]. Set-valued attributes often contain the object identifiers (oids) of other objects. Such a structure can be used to naturally model the relationship between a composite part and its subparts, between a program module and its functions, and between a department and its employees—to name just a few examples. Given this structure, a common form of join in these systems is to scan a set and examine the set-valued attribute of each element of the set. The following is an example of such a join expressed in O++ (where P->subparts is the set-valued attribute of object P).

(1) for (P of CompositePart; C of P->subparts) suchthat (C->cost > 100) printf("%s %s %d ", P->name, C->name, C->cost);

(In (1), if C is an element of a P->subparts instance, then we term C a child of P, and P a parent of C.)

The most straightforward way to execute (1) is to read a CompositePart and then follow its child pointers, essentially an index nested-loops join algorithm. However, that method of computing the join is often very inefficient. Since these joins are expected to be common, it is important to find efficient methods of evaluating them.

This paper considers four algorithms for evaluating joins like (1) and compares their performance. The remainder of the paper is organized as follows. Section 2 surveys related work. Section 3 describes four pointer-based join algorithms for joins like (1). Section 4 analyses the four algorithms; Section 5 compares them. Section 6

This research was partially supported by a grant from Bell Laboratories.
contains our conclusions and future work.

2. Related Work

In [VALD87], auxiliary data structures called join indices that can be used to speedup join processing are described. A join index for relations R and S essentially precomputes the join between those two relations by storing pairs of tuple identifiers (tids). Each pair contains a tid from both R and S such that the corresponding tuples join with one another. In a uni-processor system, the basic algorithm scans the index, reading the referenced tuples. [VALD87] compared the performance of join indices to the Hybrid-hash join algorithm, and showed that a more elaborate join algorithm using join indices could frequently produce better performance than Hybrid-hash in a uni-processor system. [OMIE89] compared the two algorithms in a parallel environment and showed that Hybrid-hash will almost always outperform join indices except when the join selectivity is very high.

[CARE90] describes an incremental join facility added to Starburst to enable a relational DBMS to efficiently handle many-to-one relationships. The set representation employed is similar to the representation provided by network database systems. Each parent (e.g. a CompositePart) contains a pointer to its first child (e.g. a subpart)—there is an explicit ordering on children. All children are linked together in a doubly-linked list. Each child also contains a pointer to its parent. The paper describes the results of an empirical performance study comparing the performance of a number of pointer-based join algorithms. We evaluate a different set of pointer-based join algorithms analytically. We also assume a different set representation—one where each parent contains a list of all its children’s oids. The children objects are not linked and do not necessarily have pointers to their parents.

We employed the same set representation used in [SHEK90] which presented an analytical evaluation of pointer-based join algorithms. [SHEK90] describes, analyzes, and compares uni-processor versions of the pointer-based Hybrid-hash and Hash-loops join algorithms assuming no sharing of objects. We analyze a parallel version of Hash-loops and two parallel versions of pointer-based Hybrid-hash. We compare their performance to a new algorithm, the Probe-child join algorithm. [SHEK90] implicitly assumed that all objects mentioned in the query could be accessed through an extent. We do not make this assumption, but instead propose the Find-children algorithm to compute an implicit extent when an explicit extent does not exist. The spirit of our Hash-loops analysis was influenced by [SHEK90]. However, changes were required to account for the effects of sharing, of selection and projection, and of declustering objects. [SHEK90] did not take space overhead for a hash table into account; replacing [SHEK90]’s assumption that a hash table for c pages of persistent data requires c pages of main memory also required a number of changes.
Other proposed pointer-based join algorithms include pointer-based nested loops [SHEK90], pointer-based sort-merge [SHEK90], and pointer-based PID-partitioning [SHEK91]. These three algorithms are analyzed in [SHEK91].

3. Four Pointer-based Join Algorithms

Consider the following loop where each element of Set1 contains a set-valued attribute named "set":

\[
(2) \text{for } (x_1 \text{ of } \text{Set1}; \ x_2 \text{ of } x_1->\text{set}) \text{ suchthat } \ (\text{Pred2}(x_1, x_2)) \\
S21;
\]

This section describes parallel execution strategies for query (2). In (2), we will call the (implicit) set of all children objects Set2. We assume that Set1 and Set2 are declustered [GHAN90] across \( n \) nodes, and that exactly those \( n \) nodes will be used to execute the join. The children of a Set1 object may be located on different nodes. We do not consider the case where children have back-pointers to their parents; this eliminates a number of the execution strategies that [SHEK91] presented for uni-processor systems. Such back-pointers will frequently not exist in an OODBS. This section describes and analyzes four execution strategies for loops like (2): the Hash-loops, the Probe-children, the Hybrid-hash/node-pointer, and the Hybrid-hash/page-pointer join algorithms. It also contains the Find-children algorithm which allows algorithms like [GERB86, SCHN91, DEWI92a]’s parallel Hybrid-hash and our Probe-children join algorithm to be used even when an explicit extent of children of Set1 objects does not exist.

For all of the algorithms, we assume that set-valued attributes are represented as lists of oids stored inside the Set1 objects and that the oids are "physical" [KHOS86]—that is, each oid contains information about the node and disk page of the referenced object. We term such oids page-pointers. Actually, only the Hash-loops and Hybrid-hash/page-pointer algorithms require page-pointers. Probe-children and Hybrid-hash/node-pointer only require node-pointers—oids neither completely physical nor completely logical, but which contain sufficient information to calculate the node that the corresponding object resides on. For example, a node-pointer to a Set2 object could be the partitioning attribute for Set2 provided that Set2 is an explicit extent and the partitioning attribute is a key. Thus, Probe-children and Hybrid-hash/node-pointer can be used for parallel relational database systems that support range and/or hash partitioning.

We will ignore statement S21 in query (2) in the following discussion. S21 will be executed once for each result tuple produced. [LIEU92b] discusses the conditions under which the join algorithms can be modified to execute S21 at the join nodes. Otherwise, S21 must be executed centrally by the application program. Leaving out S21 will make it clearer that the algorithms are equally applicable to OODBS and DBPLs. Analysis of the
algorithms can be found in Section 4.

3.1. Hash-loops

This algorithm is a parallel version of the Hash-loops join algorithm. We will first present the uni-processor version [SHEK90], and then examine a parallel version. Both require physical oids. Let $S_i$ be the subset of set $S$ at node $i$. The uni-processor Hash-loops algorithm repeats the following two steps until all of the objects in $Set_1$ (the outer set in query (2)) have been processed:

1. A memory-sized chunk of $Set_1$ (or the remaining portion of $Set_1$ if that is smaller) is read into main memory. For each $Set_1$ object, the page identifier (PID) components of all the oids contained in the object’s set-valued attribute are extracted and inserted into a hash table. Each hash entry contains both a PID and a list of pointers to the $Set_1$ objects with children on that page (e.g. in Figure 1, Page #20 contains child objects of both Ralph and Pat).

2. Once the hash table is built, the hash entries are processed sequentially by reading the corresponding page into the buffer pool. Then, each $Set_1$ object that references the page is joined with the relevant child objects on the page.
Figure 1: Uni-processor Hash-loops example

For example, assume that the current iteration loads two \texttt{Set1} objects into the hash table shown in Figure 1. Step (2) executes as follows. First, Page #10 is read from disk. The only pointer in the hash entry for Page #10 is Ralph. Since Ralph references Sam, the result tuple Ralph/Sam is produced. Next, Page #20 is read from disk. The first pointer in its hash entry is Pat which references Kyle. The result tuple Pat/Kyle is produced. The second pointer in the hash entry is Ralph, and the result tuple Ralph/Kim is produced.

The parallel version executes as follows at each node, \( \forall i \leq n \):

1. Scan \texttt{Set1}. Project each selected \texttt{Set1} object to produce a tuple. We will term these tuples \texttt{Set1}-tuples (to distinguish them from the original objects). A \texttt{Set1}-tuple is sent to each node that contains a child object of the original \texttt{Set1} object. (The tuple sent to node \( j \) will only contain oids that reference node \( j \).) As \texttt{Set1}-tuples arrive at node \( i \), they are inserted into a hash table. On memory overflow, newly arriving tuples are written to a single overflow partition on disk. <Synchronization>

2. Execute step (2) of the uni-processor algorithm.
(3) After the initial hash table is processed, repeat steps (1) and (2) of the uni-processor algorithm (with the overflow tuples taking the place of Set1) until there are no more overflow tuples to process.

The <Synchronization> at the end of step (1) means that no node can start step (2) until all nodes have completed step (1).

Figure 2: "Replicating" a Set1 object during step (1) of Parallel Hash-loops

Figure 2 illustrates the projection and "replication" of a single selected Set1 object to produce two Set1-tuples during step (1).

3.2. Find-children

If the implicit set Set2 is not maintained as an explicit extent, one must use a parallel version of the Hash-loops, pointer-based nested loops, or pointer-based Hybrid-hash join algorithm [SHEK90]. Alternatively, an extent can be explicitly computed—which is what the Find-children algorithm does. Find-children computes which of each node’s pages contain elements of the implicit set Set2. Given the computed extent, a standard parallel Hybrid-hash algorithm can be used by producing Set1-tuples which contain exactly one child oid each and then using the oid as the join attribute. Finding the children also allows our new Probe-children join algorithm, described in the following section, to be applied. Find-children is not a join algorithm; it is an algorithm that can compute information required by certain join algorithms.

The Find-children algorithm proceeds as follows at each node, \( \forall i \leq n \):
(1) Set1i is scanned. Each child pointer (an oid) contained in a selected Set1i object is stripped of all information other than the PID component of the oid (i.e. information about the node and about the object's position on the page are removed). Each stripped pointer is put in the bucket corresponding to the node that the original pointer referenced (one of n buckets). If only some of the stripped pointers can fit in main memory, the excess stripped pointers are sent to the node that they reference. Hash tables are built locally for the memory-resident stripped pointers to make it easy to eliminate duplicates.

(2) Once Set1i has been processed, the memory-resident stripped pointers are sent to the appropriate nodes, where they are written to disk as they arrive. (If there is sufficient room, sorted runs of stripped pointer should be produced before writing them to disk.) <Synchronization>

(3) Pointers are sorted to remove duplicates and written back to disk at each node. (Even if step (1) eliminates all duplicate references to nodei locally at each of the n nodes, multiple nodes may reference the same page on nodei. Thus, duplicate elimination is always needed during step (3).)

Essentially, Find-children performs a semi-join between the pages of Set2 and the relevant pointers of Set1. As a simple example, assume that two Set1 objects are stored at nodei, each of which contains a set-valued attribute with two children, and there is sufficient room in main memory to hold all the child pointers. Figure 3 illustrates the steps the algorithm goes through up to the synchronization point. Steps (a)-(d) are from step (1) of the algorithm; steps (e) and (f) are from step (2). Stripped pointers may be received from other nodes while steps (a)-(f) are occurring, and may continue to arrive until the synchronization point is reached.

We analyzed this algorithm in some detail, and used the analysis in our algorithm comparisons in Section 5. We do not include the analysis because the cost of Find-children at nodei is roughly the cost of an extra scan of Set1i unless the number of distinct stripped pointers contained in Set1i objects is huge. They will usually all fit in main memory, so there will be no need to write stripped pointers to disk during step (1). The cost of sorting and writing stripped pointers to disk during step (3) is minimal compared to the cost of scanning Set2i for the join. The comparisons include these smaller costs associated with applying Find-children, but the results would not be qualitatively different if only the extra scan of Set1i was accounted for.
3.3. Probe-children Join

This algorithm requires knowing the extent $\text{Set}_2$. (The **Find-children** algorithm can be used to compute the extent if necessary.) **Probe-children** proceeds as follows at each node_{i} \ \forall i \leq n:

1. $\text{Set}_2$, is scanned and the selection predicate is applied. Each selected, projected $\text{Set}_2$-tuple (tagged with its oid) is put into a local memory-resident hash table based on its oid. This continues until there is no more room in main memory. <Synchronization>

2. Scan $\text{Set}_1$, producing and distributing $\text{Set}_1$-tuples as in step 1) of parallel **Hash-loops**. The oids contained in the set-valued attribute of each arriving $\text{Set}_1$-tuple replica are hashed to find the relevant children. Produce one result tuple for each match. If all of the $\text{Set}_2$ objects referenced by the $\text{Set}_1$-tuple are currently in the hash table, the tuple is discarded. Otherwise, the parent and unseen children oids are written to disk. <Synchronization>

3. Repeat the following two steps until all of $\text{Set}_2$, has been processed.

   a. Scan the unread portion of $\text{Set}_2$, and put the selected $\text{Set}_2$-tuples (tagged with their oids) into the hash table until the hash table fills or the set is exhausted.
(b) Read the parents and probe the hash table for children.\footnote{A variant of this algorithm would only keep parents that have unseen children. This requires a write of such parents during step (b). The variant has some similarities to the Simple-Hash join algorithm [DEW84]. In both, the tuples that need to be processed later are written to a new file and that new file is used for the next iteration. The actual Probe-child join algorithm, however, proceeds in much the same way as the Hashed Loops join algorithm for centralized databases [GERB86]. Both build a memory-sized hash table for some fraction of the inner set Set2. They read the whole (local partition of the) set of Set1-tuples to probe the hash table. This continues until all of Set2 has been processed. There are two major differences between the uni-processor Probe-children and Hashed Loops. First, Probe-children probes the hash table once for each pointer in its set-valued attribute; Hashed Loops probes the hash table exactly once. Second, Probe-children eliminates some Set1-tuples during step (2)—much like Simple-Hash; Hashed Loops reads all of Set1 once for each hash table.}

Here is an example (using data from the example in the Find-children section) of processing a single Set1-tuple at node5.

![Figure 4: Processing a Set1-tuple by Probe-children at node5](image)

For simplicity, we use the page identifier as the hash value of a pointer in this example—in general a more complicated hash function will be used. The object’s pointer is hashed and the hash entry for Page #7 is found. The hash entry’s pointers to Set2\_5-tuples are followed and the oids in the tuples are compared to the oid in Sue’s set-valued attribute. Jill matches, so the result tuple Sue/Jill is produced. If the match was found during step (2), Sue need not be written to disk for processing during step (3) as there will be no more Set2\_5-tuples that Sue will join with.

3.4. Hybrid-hash/node-pointer

**Hybrid-hash/node-pointer**, like Probe-children, requires knowing the extent Set2. The algorithm proceeds as follows at each nodei \(i \leq n\):

1. Set2\_i is scanned and the selection predicate is applied. Each selected, projected Set2\_i-tuple (tagged with its oid) is hashed on its oid and inserted into one of \(B+1\) buckets. Tuples in the first bucket are inserted into a main
memory hash table. All others are written to disk. The value of $B$ is chosen such that (1) the 1st bucket of Set1-tuples can be joined in memory with the first bucket of Set2-1-tuples during step (2), and (2) the hash table for the j-th Set2 bucket can fit in main memory $\forall j 2 \leq j \leq B+1$. <Synchronization>

(2) Scan Set1, and distribute Set1-tuples, each of which contains exactly one Set2 (child) oid, to the relevant Set2 node. As a tuple arrives, its child oid is hashed and the tuple is put into one of $B+1$ buckets. The j-th Set1 bucket will only join with the j-th Set2 bucket. The Set1-tuples of the first bucket probe the hash table; the other Set1 elements are written to disk.

(3) The next Set2 bucket is loaded into the hash table, and the tuples of the corresponding Set1 bucket probe it. This continues until all the buckets have been processed.

This algorithm is very similar to Probe-child. There are three main differences. First, Hybrid-hash/node-pointer may write and reread part of Set2; Probe-children will only read Set2 once. Second, Probe-children produces one replica of a Set1 object per referenced node; Hybrid-hash/node-pointer produces one replica per pointer. Thus, the Probe-children algorithm will potentially produce fewer Set1-tuples. Third, Probe-children may reread the same Set1-tuples multiple times; Hybrid-hash/node-pointer will reread Set1-tuples at most once because it partitions Set1- and Set2- tuples into buckets.

3.5. Hybrid-hash/page-pointer

This algorithm is almost identical to the pointer-based Hybrid-hash join algorithm of [SHEK90]. Only step (1) which redistributes Set1 is different. The algorithm proceeds as follows at each node, $\forall i 1 \leq i \leq n$:

(1) Scan Set1, and distribute Set1-tuples, each of which contains exactly one Set2 (child) oid, to the relevant Set2 node. As a tuple arrives, the PID component of its child oid is hashed and the tuple is put into one of $B+1$ buckets. Tuples in the first bucket are inserted into a main memory hash table. All others are written to disk. <Synchronization>

(2) Execute step (2) of the uni-processor Hash-loops join algorithm.

(3) After the initial hash table is processed, load the next Set1 bucket into the hash table. Process the table using step (2) of the uni-processor Hash-loops join algorithm. This continues until all the buckets have been processed.

The only differences between this algorithm and Hash-loops are that (1) each tuple has only one pointer, and (2) each Set2 page is read only once (because of partitioning tuples into buckets).
4. Analysis of Pointer-based Join Algorithms

This section analyses the performance of the four join algorithms described in Section 3 for queries like:

(3) \textbf{for} (X1 \textbf{of} Set1; X2 \textbf{of} X1->set) \textbf{suchthat} (Pred2(X1,X2)) \textbf{S21};

To simplify our analysis, we assume that I/O is only performed on behalf of the join—that S21 does not perform any I/O.

4.1. Assumptions for Analysis

We assume that the selection predicates on Set1 and Set2 are equally selective at each node (i.e. no Selectivity Skew [WALT91]). We define two functions for use in the analysis. The first is $\delta(s,p)$, the number of objects of size $s$ that fit on a page of size $p$:

$$\delta(s,p) = \left\lfloor \frac{p}{s} \right\rfloor$$

The second is $\theta(m,s,p)$, the number of pages of size $p$ required to hold $m$ objects of size $s$:

$$\theta(m,s,p) = \left\lfloor \frac{m}{\delta(s,p)} \right\rfloor$$

Table 1 shows the system parameters and Table 2 shows the catalog information used by the analysis in later sections.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>number of nodes</td>
<td>32</td>
</tr>
<tr>
<td>$P$</td>
<td>size of a disk page</td>
<td>8192 bytes</td>
</tr>
<tr>
<td>$M_i$</td>
<td>number of memory buffer pages at node $i$</td>
<td>varied</td>
</tr>
<tr>
<td>IO</td>
<td>time to read or write a page</td>
<td>20 msec</td>
</tr>
<tr>
<td>$\text{size}_{\text{ptr}}$</td>
<td>size in bytes of a persistent pointer</td>
<td>12 bytes</td>
</tr>
<tr>
<td>$F$</td>
<td>hash table for $m$ pages of data requires $F \cdot m$ pages</td>
<td>1.2 bytes</td>
</tr>
<tr>
<td>$k$</td>
<td>each Set1 object has $k$ children</td>
<td>10</td>
</tr>
<tr>
<td>$f$</td>
<td>each Set2 object has $f$ parents</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 1: System Parameters

The assumption that each object of Set1 has $k$ children, and each child has $f$ parents implies that each Set2 object is equally likely to be referenced from a randomly chosen Set1 object. Also, dividing the number of pointers that reference Set2, by $f$ gives the cardinality of Set2. Thus, $|\text{Set2}| = \sum_{j=1}^{n} (|\text{Set1}_j| \cdot \alpha_{ji} \cdot k) / f$.

Table 2 implicitly assumes that each data page contains only Set1 or Set2 objects. We also assume that the $\alpha_{ji}$ are the same for the selected subset of Set1, as for all of Set1 (this assumption shows up in the formula for $\rho_{ij}$.
in Table 2). Since our examples used to compare algorithms do not select Set1, this assumption does not affect our performance results.

We do not include CPU time in our analysis because the CPU work required is proportional to the larger of the I/O and CPU times in the algorithms considered. [WALT91] included CPU time, but none of the queries were CPU bound—the CPU time was always lost in the overlap with I/O and communication. Thus, we feel it is safe to neglect CPU time in the analysis. Originally, we used [WALT91]'s style of capturing overlapped communication and I/O. We removed communication from this analysis for two reasons. First, it is easy to derive the number of messages sent using the analysis of I/Os performed. Second, communication was completely overlapped with I/O in all our algorithm comparisons (which assumed that sending a 8K message takes 5 msec and performing an I/O with an 8K page takes 20 msec). Communication will only become the dominant cost for any algorithm considered in this section if each object must be replicated many times and each node has enough main memory to hold all its hash table data in

3Selection predicates only refer to a single iterator variable—like $P$ or $C$ in (1)—and to expressions constant for the duration of the loop. ($P$ $cost$ $> 100$) is an example.
a single hash table.

4.2. Analysis of Hash-loops

Before we can estimate the number of I/Os performed, we must estimate several other quantities. During step (1), node, is expected to receive

\[ \phi_i = \sum_{j=1}^{n} \phi_j \]

Set 1-tuples and \( \sum_{j=1}^{n} \rho_j \) pointers. Thus, the average number of pointers per Set 1-tuple received at node; is

\[ \alpha_i = \frac{\rho}{\phi_i} \]

where \( \rho = \sum_{j=1}^{n} \rho_j \)

The tuples received at node; will be \( \pi \text{width}_{\text{Set 1}} \) bytes long from the projected fields other than the set-valued attribute. The set-valued attribute can be represented with an integer count and a list of oids, so the average length of a Set 1-tuple will be

\[ \pi \text{avg} \cdot \pi \text{width}_{\text{Set 1}} + \text{sizeof(int)} + \alpha_i \cdot \text{sizeof(pr)} \]

Thus, node; will receive approximately

\[ \text{PagesReceived}_{i}^{\pi \sigma} = \theta(\phi_i, \pi \text{avg}, P) \]

pages of selected, projected Set 1-tuples, each of which contains approximately

\[ \text{TuplesPerPage}_{i}^{HL} = \delta(\pi \text{avg}, P) \]

tuples. During step (1), node; will need one input buffer for reading the Set 1, objects it must distribute to the n nodes, n output buffers for sending (replicated) Set 1-tuples, one input buffer to receive incoming Set 1-tuples from other nodes, and one output buffer for writing overflow Set 1-tuples to disk. Thus, node; will have \( M_i - (n+3) \) pages available for its hash table. Since a hash table for c pages of data is assumed to take \( c \cdot F \) pages of main memory,

\[ \text{PagesInMem}_{i}^{HL} = \min \left( \text{PagesReceived}_{i}^{\pi \sigma}, \left\lfloor \frac{M_i - (n+3)}{F} \right\rfloor \right) \]

pages of Set 1-tuples can be put into the hash table at node; and

\[ \text{ToDisk}_{i}^{HL} = \text{PagesReceived}_{i}^{\pi \sigma} - \text{PagesInMem}_{i}^{HL} \]

pages must be written to disk for processing during step (3). Thus, node; is expected to perform the following number of I/Os during step (1):
\[ T^{HL}_{i} = P_{set_{i}} \cdot read\ Set_{i}, \]
\[ + ToDis^{HL}_{i} \cdot write\ overflow\ pages\ of\ Set_{i}-tuples\ to\ disk \]

To estimate the amount of work done by node \( i \) during step (2), note that a given \( Set_{2i} \) page will be read at most once per hash table. To calculate the number of page reads for \( Set_{2i} \) objects, we use a formula from [YAO77] for calculating the expected fraction of pages of a set with cardinality \( |S| \) that must be read to examine a subset of size \( |S_{\sigma}| \) provided \( |S_{\sigma}| \) objects fit on a page. Yao's formula is:

\[
Y(|S|, |S_{\sigma}|) = 1 - \frac{|S| - |S_{\sigma}|}{|S|} \cdot \frac{|S| - |S_{\sigma}| - 1 + 1}{|S| - 1 + 1}
\]

To use Yao's formula to calculate the fraction of \( Set_{2i} \) pages that must be read for a particular hash table, an estimate of the number of \( Set_{2i} \) objects that are referenced by \( r \) \( Set_{1} \) objects is needed. To estimate this quantity, we used combinatorics to derive \( Obj(r, z, c) \), the expected number of \( Set_{1n} \) objects referenced by \( r \) \( Set_{out} \) objects where each \( Set_{out} \) object contains a set-valued attribute with an average of \( z \) pointers to \( Set_{1n} \) objects (\(|Set_{1n}| = c\)). We derived \( Obj(r, z, c) = c \cdot \left( 1 - \left( 1 - \frac{z}{c} \right)^r \right) \), but found that replacing it with the much simpler approximation \( Obj(r, z, c) = \min(\lceil r \cdot z \rceil, c) \) produced nearly identical timing results.

We use this formula to estimate the number of I/Os performed during step (2). The \((PagesInMem_{i}^{HL}, \ TuplesPerPage_{i}^{HL})\) hash table entries of step (2) will reference approximately \( Obj((PagesInMem_{i}^{HL}, \ TuplesPerPage_{i}^{HL}), a_{r}, |Set_{2i}|) \) \( Set_{2i} \) objects. Using Yao's formula with this estimated \( Set_{2i} \) subset size, the fraction of \( Set_{2i} \) pages that must be accessed is approximately \( Y(|Set_{2i}|, O_{set_{2}}, Obj((PagesInMem_{i}^{HL}, \ TuplesPerPage_{i}^{HL}), a_{r}, |Set_{2i}|)) \). Thus, step (2) is expected to perform

\[
Step_{2i} = P_{set_{2i}} \cdot Y(|Set_{2i}|, O_{set_{2}}, Obj((PagesInMem_{i}^{HL}, \ TuplesPerPage_{i}^{HL}), a_{r}, |Set_{2i}|))
\]

\( Set_{2i} \) page reads.

To estimate the number of \( Set_{2i} \) reads that must be performed during step (3) to process the \( ToDis^{HL}_{i} \) pages written to disk during step (1), we use reasoning similar to the above. Step (3) at node \( i \) should perform
\[ \text{Step}^3_i = \begin{cases} (N_i-1) \cdot P_{\text{Set}^2_i} \cdot \gamma(|\text{Set}^2_i|, O_{\text{Set}^2_i}, Obj(x_i, a_r, |\text{Set}^2_i|)) & \text{if } T_{\text{Disk}}^{HL} > 0 \\ + P_{\text{Set}^2_i} \cdot \gamma(|\text{Set}^2_i|, O_{\text{Set}^2_i}, Obj(y_i, a_r, |\text{Set}^2_i|)) & \text{otherwise} \end{cases} \]

Set2 reads where:

\[ o_i = T_{\text{Disk}}^{HL}_i \cdot \text{TuplesPerPage}^{HL}_i \]

number of Set1-tuples to process

\[ x_i = \left[ \frac{M_i-1}{F} \right] \cdot \text{TuplesPerPage}^{HL}_i \]

number of Set1-tuples that fit in a main memory hash table—one page needed for reading Set2_i objects

\[ N_i = \left[ \frac{o_i}{x_i} \right] \]

number of Hash-loops iterations at node_i for step (3)

\[ y_i = o_i - (N_i-1) \cdot x_i \]

number of Set1-tuples for the N_i-th iteration

Thus, node_i should perform the following number of I/Os during steps (2) and (3):

\[ T^{\text{IO}}_{i,3} = \begin{cases} \text{Step}^3_i & \text{read Set2_i pages to process hash table of step (2)} \\ + T_{\text{Disk}}^{HL}_i & \text{read } T_{\text{Disk}}^{HL}_i \text{ pages of Set1-tuples during step (3)} \\ + \text{Step}^3_i & \text{read Set2_i pages to process hash tables of step (3)} \end{cases} \]

To compute the expected run time, we must add the the length of time spent getting to each synchronization point (i.e. to the end of step (1) and then to the end of step (3)). Thus the expected run-time is:

\[ \left( \max \left\{ T_{\text{IO}}^{HL,i} \right\}_{i \in \{1, \ldots, n\}} \right) + \max \left\{ T_{\text{IO}}^{HL,3,i} \right\}_{i \in \{1, \ldots, n\}} \right) \cdot IO \]

4.3. Analysis of Probe-children Join

There is only one difference between applying Probe-children with an explicit and a computed extent. In the second case, the stripped pointers must be read from disk and one buffer page must be reserved for this purpose. This will not make any qualitative difference, so we will ignore it in the analysis that follows in this paper (we included it in the experiments we ran, but including or not including these costs makes no qualitative difference in the results).

At most \((M_i-(n+3))\) pages can be devoted to the hash table loaded in step (1), since during step (2), one buffer will be needed for reading Set1_i objects, n buffers for sending Set1_i-tuples, one buffer for receiving Set1-tuples from other nodes, and one buffer for writing Set1-tuples to disk. There are \(\Xi_{\text{Set}^2_i}\), Set2_i-tuples that will need to be loaded into a hash table during steps (1) and (3). Each of these tuples will need to be tagged by its oid, so each hash table entry will be \((\pi \text{width}_{\text{Set}^2_i} + \text{size}_{\text{pr}})\) bytes long and

\[ O_{\text{Set}^2_{pr}} = \delta ( (\pi \text{width}_{\text{Set}^2_i} + \text{size}_{\text{pr}}), P) \]

entries will fit on a page. Thus step (1) will load
\[ T_{io}^{PC1} = \min \left( \Xi_{Set2_i} \cdot v \right) \text{ where } v = \left[ \frac{M_i - (n+3)}{F} \right] \cdot O_{set2_{psr}} \]

tuples into the hash table at node \( i \). Assuming the selected \( Set2 \) objects are uniformly distributed across pages of \( Set2_i \),

\[ T_{io}^{PC1} = \frac{Tuples_{Mem}^{PC1}}{\Xi_{Set2_i}} \cdot P_{set2_i} \]

\( Set2_i \) pages will be read during step (1). (This assumption is made to attempt to fairly divide up the cost of reading \( Set2_i \) among steps (1) and (3); the same expected run-time would be obtained for the algorithm if the complete cost of the read was charged to either step (1) or (3) alone.)

During step (2), if all the \( Set2_i \)-tuples fit in main memory (i.e. \( Tuples_{Mem}^{PC1} = \Xi_{Set2_i} \)), then no \( Set1 \)-tuples need to be written to disk. Otherwise, some fraction of them must. In this case, we will make the worst case assumption that all \( PagesReceived_i^{\sigma} \) pages of \( Set1 \)-tuples will be written to disk.\(^3\) Thus, node \( i \) will perform the following number of I/Os during step (2):

\[ T_{io}^{PC2} = P_{set1_i} \]

\[ + \begin{cases} 
0 & \text{if } Tuples_{Mem}^{PC1} = \Xi_{Set2_i} \\
PagesReceived_i^{\sigma} & \text{otherwise}
\end{cases} \]

write \( Set1 \) pages to disk during step (2)

During step (3), there will be \( (\Xi_{Set2_i} - Tuples_{Mem}^{PC1}) \) \( Set2_i \)-tuples to load into a hash table. During each iteration of step (3), one page must be reserved for reading \( Set1 \)-tuples, so \( (M_i-1) \) pages are available for a hash table. Thus,

\[ Tuples_{Processed}^{PC3} = \left[ \frac{M_i - 1}{F} \right] \cdot O_{set2_{psr}} \]

\( Set2 \)-tuples can be processed in each step (3) iteration. It follows that

\[ \text{iterations}^{PC3} = \left[ \frac{\Xi_{Set2_i} - Tuples_{Mem}^{PC1}}{Tuples_{Processed}^{PC3}} \right] \]

iterations of step (3) will be required—each of which will require reading all \( PagesReceived_i^{\sigma} \) pages of \( Set1 \)-tuples.

\(^3\)PagesReceived_i^{\sigma} is the number of pages of selected, projected \( Set1 \)-tuples that reference \( Set2_i \)—see Analysis of Hash-loops for the formula.
tuples. Thus, node\(_i\) should perform the following number of I/Os during step (3):

\[
I_{IO}^{PC_3} = \text{Iterations}_{PC_3}^{i} \cdot \text{PagesReceived}_{i}^{\pi \sigma} \quad \text{read pages of Set} \_1 \text{ written during step (2) once for each iteration of step (3)}
\]

\[
\frac{\Xi_{Set2_i}^{\text{-TupsInMem}_{PC_1}^{i}}}{\Xi_{Set2_i}^{\text{-TupsInMem}_{PC_1}^{i}}} \cdot P_{Set2_i}^{i} \quad \text{read Set}_2 \_i \text{ pages during step (3) — } P_{Set2_i} \text{ Set}_2 \_i \text{ pages read during steps (1) and (3)}
\]

Since synchronization is required after each step, the expected run-time for the query is:

\[
\left[ \max \left\{ T_{IO}^{PC_1} \mid i \in \{1, \ldots, n\} \right\} + \max \left\{ T_{IO}^{PC_2} \mid i \in \{1, \ldots, n\} \right\} + \max \left\{ T_{IO}^{PC_3} \mid i \in \{1, \ldots, n\} \right\} \right] \cdot IO
\]

4.4. Analysis of Hybrid-hash/node-pointer

As in our analysis of Probe-children, we will ignore the minor differences between using an explicit and using a computed extent. During step (1), selected, projected Set\(_2\_i\)-tuples will be produced. Since each tuple is tagged with the original object’s oid, the tuples will require

\[
p_{HH-H}^{H} = 0 \left( \Xi_{Set2_i}^{\text{-width}_{Set2} + \text{size}_{pr}}, P \right)
\]

pages of storage. During step (2), one page is required for reading Set\(_1\)_i objects, \(n\) pages are required for sending Set\(_1\)_tuples, and one page is required for receiving tuples from other nodes. Thus,

\[
M_{HH}^{i} = M_{i} - (n+2)
\]

pages are available for the hash table and the output buffers at node\(_i\). Using reasoning from [DEW184],

\[
B_{HH-H}^{i} = \frac{p_{HH-H}^{i} \cdot F - M_{HH}^{i}}{M_{HH}^{i} - 1}
\]

output buffers are needed and the fraction

\[
q_{HH-H}^{i} = \min \left\{ 1.0, \frac{M_{HH}^{i} - B_{HH-H}^{i}}{p_{HH-H}^{i} \cdot F} \right\}
\]

of pages of Set\(_2\_i\)-tuples can be put in the hash table (the min is required because at most 100% can be put into hash table). Thus,

\[
\text{Overflow}_{HH}^{i} = \left[ p_{HH-H}^{i} \cdot (1 - q_{HH-H}^{i}) \right]
\]

pages of Set\(_2\_i\)-tuples must be written to disk, and step (1) will require the following number of I/Os:
\[ T_{i}^{HH-H} = P_{Set2} \quad \text{read Set2 to select and project} \]
\[ + \text{Overflow}_{i}^{H} \quad \text{write overflow Set2-tuples} \]

During step (2), Set1 must be read. Node, will receive \( \sum_{j=1}^{n} \rho_{j} \) Set1-tuples. Since each tuple contains exactly one pointer, these tuples will require

\[ p_{HH}^{i} = \theta \left( \sum_{j=1}^{n} \rho_{j}, (\pi width_{Set1} + size_{pr}), P \right) \]

pages of memory, and

\[ \text{Overflow}_{i}^{H2} = \left[ p_{HH}^{i} \cdot (1-q^{HH-H}_{i}) \right] \]

pages of these tuples must be written to disk. The other tuples probe the first hash table. Thus, step (2) must perform:

\[ T_{i}^{HH-H2} = P_{Set1} \quad \text{read Set1,} \]
\[ + \text{Overflow}_{i}^{H2} \quad \text{write overflow Set1-tuples} \]

I/Os. Step (3) must read the partitions written to disk, so

\[ T_{i}^{HH-H3} = \text{Overflow}_{i}^{H1} \quad \text{read overflow Set2-tuples} \]
\[ + \text{Overflow}_{i}^{H2} \quad \text{read overflow Set1-tuples} \]

I/Os must be performed. The only synchronization required is at the end of step (1), so the expected run-time is:

\[ \left[ \max \left( T_{i}^{HH-H1}_{i} \mid i \in \{1,...,n\} \right) \right] + \max \left[ T_{i}^{HH-H2}_{i} + T_{i}^{HH-H3}_{i} \mid i \in \{1,...,n\} \right] \cdot IO \]

4.5. Analysis of Hybrid-hash/page-pointer

Using the analysis of Hybrid-hash/node-pointer, we estimate that node, will receive \( p_{HH}^{i} \) pages of Set1-tuples. Since the space requirements of step (1) of Hybrid-hash/page-pointer and step (2) of Hybrid-hash/node-pointer for re-clustering Set1 are identical, \( M_{HH}^{i} \) pages are available for the hash table and the output buffers at node. Using analysis in [DEW184],

\[ B_{i}^{HH-P} = \left\lfloor \frac{p_{HH}^{i} \cdot F - M_{HH}^{i}}{M_{HH}^{i} - 1} \right\rfloor \]

output buffers will be needed, and the fraction
\[ q_{HH-P}^i = \min \left\{ 1.0, \frac{M_{HH}^i - B_{HH-P}^i}{P_{HH}^i \cdot F} \right\} \]

of the arriving pages of Set1-tuples can be put in the hash table. Thus,

\[ \text{Overflow}_{i}^{P1} = \left[ p_{HH}^i \cdot (1-q_{HH-P}^i) \right] \]

pages must be written to disk, and the cost of step (1) is:

\[ T_{IO}^{HH1}_{i} = \begin{cases} P_{\text{Set}1_i} & \text{read Set1}_i \text{ to select, project, and replicate} \\ + \text{Overflow}_{i}^{P1} & \text{write overflow buffers to disk.} \end{cases} \]

Since partitioning is on the PID component of an oid, no Set2 pages will be reread. Thus, steps (2) and (3) will require the following number of I/Os.

\[ T_{IO}^{HH2,3}_{i} = \begin{cases} \text{Overflow}_{i}^{P1} & \text{read overflow buffers} \\ + P_{\text{Set}2_i} & \text{read Set2 pages} \end{cases} \]

This will overestimate the number of Set2, pages read if the selection predicate on Set1 is very restrictive, since in this case some Set2, pages will not need to be read at all. However, since our algorithm comparisons involve selecting all of Set1, such a correction would not affect the results. Thus, the expected run-time is:

\[ \left\{ \max \left\{ T_{IO}^{HH1}_{i} \mid i \in \{1,\ldots,n\} \right\} + \max \left\{ T_{IO}^{HH2,3}_{i} \mid i \in \{1,\ldots,n\} \right\} \right\} \cdot IO \]

5. Comparison of the Algorithms for Set-Valued Attributes

In this section, we will compare the four algorithms presented in Section 3 and analyzed in Section 4. We will call Hybrid-hash/node-pointer and Probe-children the load-child algorithms; Hybrid-hash/page-pointer and Hash-loops the load-parent algorithms; and Probe-children and Hash-loops the low-replication algorithms (because they produce one replica per node rather than per pointer). We assume in all our algorithm comparisons that selection predicates are equally selective at each node (i.e. no Selectivity Skew [WALT91]).

5.1. Poorly Clustered Database

In the first comparisons, the system defaults from the Section 4.1 were used. Data was uniformly distributed across \( n=32 \) nodes, with \( |\text{Set1}_i| = 6080 \), and \( |\text{Set2}_i| = 30,400 \) \( \forall i \leq n \). Each Set1 object had \( k=10 \) children, and each Set2 object had \( f=2 \) parents. Also, each Set1 object was \( \text{size}_{\text{Set1}} = (256 + \text{sizeof(int)} + k \cdot \text{size}_{\text{pr}}) = 380 \) bytes long, and each Set2 object was \( \text{size}_{\text{Set2}} = 256 \) bytes
long. Projected Set1-tuples were 128 bytes for fields other than the set-valued attribute (which contained one or more pointers). Projected Set2-tuples were exactly 128 bytes long. The average number of pointers per object received at node, for the low-replication algorithms (i.e. Probe-children and Hash-loops), $a_r$, was set to 1.15 (the number that would be expected if no clustering of references was attempted, and pointers from objects at node, are randomly distributed across the 32 nodes). $M_i$ (the number of memory buffer pages at each node,) and $sel_{Set2}$ (the selectivity of the predicate on Set2) were both individually varied; the unvaried parameter’s value appears at the top of the graph describing the results. All the objects of Set1 are selected. Set2 does not exist as an explicit extent, so the load-child (i.e. Hybrid-hash/node-pointer and Probe-children) algorithms must compute it using Find-children.

In Figure 5, we compare the four algorithms across a range of memory allocations. Probe-children is a step function because node must reread the replicated Set1-tuples sent to it once for each Set2, hash table. The analysis actually over-estimates both the cost of the algorithm and the size of the step because it assumes that no replicated Set1-tuples are eliminated during step (2), although some almost certainly are. Also, in reality the load-child algorithms should perform better relative to the load-parent (i.e. Hybrid-hash/page-pointer and Hash-loops) algorithms than the graph indicates because the load-child algorithms read Set2 pages sequentially (since Find-children sorts the page identifiers) while the load-parent algorithms read them randomly. However, following [WALT91], our analysis did not take the different types of I/O into account.
**Hybrid-hash/node-pointer** outperforms **Probe-children** at low memory sizes because it must perform I/Os for only a fraction of the Set1-tuples sent to each node, while **Probe-children** must write and read them all (several times). **Probe-children** and **Hybrid-hash/node-pointer** have the best performance at moderate memory sizes (where all the Set2-tuples fit in a main memory hash table). Since the number of selected Set2 objects is considerably less than the number of replicated Set1-tuples sent to node1, the hash tables for the load-child algorithms (i.e. **Hybrid-hash/node-pointer** and **Probe-children**) require much less space than those for the load-parent algorithms (i.e. **Hybrid-hash/page-pointer** and **Hash-loops**). However, the load-parent algorithms outperform the load-child algorithms for very large memory sizes, since the load-parent algorithms then read both Set1 and Set2 once. The load-child algorithms read Set1 at least twice: once to compute the Set2 extent and once to do the join. **Hash-loops** reaches optimal performance with slightly less memory than **Hybrid-hash/page-pointer** because **Hybrid-hash**'s Set1-tuples contain one pointer each, while those for **Hash-loops** contain an average of 1.15. Thus, the replicated tuples for **Hybrid-hash** require more space.

Figure 6 demonstrates that the load-child algorithms work well if there is a fairly restrictive predicate on the inner set. The more restrictive the predicate, the better they perform because they must reread Set1-tuples less frequently. The load-parent algorithms gain no benefit from a predicate on Set2 because they cannot apply the predicate until the Set2 page has already been read. (**Hybrid-hash/page-pointer** included only for reference.)

### 5.2. Well Clustered Database

The data in the first comparisons had poor clustering of references, so the full potential benefits of the low-replication algorithms (i.e. **Probe-children** and **Hash-loops**) were not seen. To illustrate the effects of good reference clustering, consider a database identical to the last one except that \( a_r \) was 2.65 (the number that would be expected if each object at node1, referenced between one and four other nodes). We compare the four algorithms across a range of memory allocations in Figure 7. The improved reference clustering does not affect the performance of either **Hybrid-hash** algorithm relative to Figure 5—they do the same amount of work because each node receives the same number of Set1 replicas. However, the performance of the low-replication algorithms improves dramatically because each node now receives 22,944 Set1-tuples, (with an average of 2.65 pointers each) instead of 52,870 tuples (with an average of 1.15 pointers each). With good clustering, **Hash-loops** reaches optimal performance long before **Hybrid-hash/page-pointer** because it has far fewer replicated Set1-tuples to process. Also, good clustering makes **Probe-children** very competitive with **Hybrid-hash/node-pointer**—as opposed to the situation in Figure 5 where **Probe-children** was the clear loser until both load-child algorithms reached optimal performance.
Figure 8 compares the load-child algorithms (i.e., Hybrid-hash/node-pointer and Probe-children) in the well-clustered database where the memory size is fixed, but the selectivity of the predicate on Set2 is varied. The performance of the Hybrid-hash algorithms is the same in Figures 6 and 8, because the same number of replicas are received at each node whether the reference clustering is good or bad. Probe-children receives fewer so its performance improves. By avoiding replication, its performance can exceed that of the Hybrid-hash algorithms.

5.3. Database with Tuple Placement Skew

In our next algorithm comparison, we considered a well-clustered database with tuple-placement skew [WALT91]. Since the performance of the whole query is determined by the slowest node, we use 31 evenly balanced nodes, each of which has 29,488 Set2 objects, and one node with 58,672 Set2 objects. Set2 has the same number of elements as in the past comparisons—it is just differently distributed. The $a_r$ for the most heavily loaded node was 2.65 as in the last example. The algorithms are compared across a range of memory allocations in Figure 9. First, we note that the larger Set2 is, the bigger the payoff of minimizing the replication of Set1-tuples, a point that is orthogonal to skew. If tuple-placement-skew on the inner set is significant, using the pointers is too expensive unless the join algorithm's hash table data can fit in a single memory-resident table. Otherwise, it will be better to replicate each Set1 object once per child pointer, redecluster both Set1 and Set2 (tagging each Set2-tuple with its oid), and use a standard parallel Hybrid-hash algorithm [GERB86, SCHN89, DEWI92a] with the oids as join attributes—in which case we would expect performance similar to Hybrid-hash/node-pointer in Figure 7 after shift-
ing it up about 19 seconds everywhere to account for having approximately twice as much to read to partition \textit{Set2} at the node with tuple-placement-skew. A hash function that ignores the node but uses the page and slot identifier from the pointer should produce fairly uniformly sized partitions. Alternatively, a skew resistant join technique [KITS90, WOLF90, HUA91, WALT91, DEW92b] might be used after producing \textit{Set1}-tuples. Note that the \textbf{Find-children} algorithm must be used to allow either of these techniques if \textit{Set2} is not an explicit extent.

![Figure 9: Database with Tuple Placement Skew](image1.png)

![Figure 10: Speedup for a Well Clustered Database](image2.png)

**5.4. Speedup and Scaleup**

Next, we compared speedups for the algorithms; we varied the number of nodes, but kept the same number of \textit{Set1} and \textit{Set2} objects as in previous examples. The objects are uniformly distributed across $n$ nodes, where $n$ is varied from 16 to 88. The references are well clustered ($\alpha^2=2.65$). Figure 10 compares the algorithms’ performance.

The load-child algorithms (i.e. \textit{Hybrid-hash/node-pointer} and \textit{Probe-children}) make relatively modest performance improvements once there are more than 40 nodes. With 40 nodes, their \textit{Set2} hash tables will fit in main memory. \textit{Hash-loops}’ performance is poor until most of the \textit{Set1}-tuples at each node will fit into the hash table.

Since \textit{Hybrid-hash/page-pointer} has 2.65 times as many tuples to put into its hash table as \textit{Hash-loops}, \textit{Hash-loops} is eventually able to provide better performance. It provides the best performance of any of the algorithms from $n=84$ on, since then all its \textit{Set1}-tuples will fit in a hash table. \textit{Hybrid-hash/page-pointer} continues to have the worst performance all the way to the point where adding more processors actually degrades performance (past $n=244$—not
on the graph⁴.

If the speedup curves are displayed in typical \( \frac{\text{small system elapsed time}}{\text{big system elapsed time}} \), all the algorithms display super-linear speedup over part of the range if the small system is one with fewer than \( n=40 \) nodes, because having one fewer \( \text{Set}_2 \) object at node \( i \) means that \( k=10 \) fewer \( \text{Set}_1 \)-tuples are sent to node, \( \left( \frac{10}{2.65} \right) \) fewer for the low-replication algorithms). Since, the load-children algorithms’ hash tables fit in main memory once \( n=40 \), if \( n=40 \) is used as the small system, they have linear speedup beyond that point to at least 164 nodes. The same is true for Hash-loops if \( n=84 \) is used as the small system.

A scaleup algorithm comparison was also run where \( \text{sel}_{\text{Set}_2}=0.50 \), \( |\text{Set}_1|=6080 \), \( |\text{Set}_2|=30400 \), and \( M_i=300 \ \forall i \ 8 \leq i \leq 248 \), as was the case in several previous comparisons. As seen in Figure 11, all of the algorithms except Hybrid-hash/node-pointer displayed near-linear scaleup over the range \( n=8 \) to 44. After that, the execution time of Hash-loops increased rapidly. Adding a new node requires taking one page from the hash table during step (1) of Hash-loops, and, eventually, this leads to degrading scaleup performance. Reducing the size of the initial hash table produced smaller performance degradation for the Hybrid-hash/page-pointer algorithm, as Hash-loops is much more sensitive to the amount of available memory than it is. The Hybrid-hash/node-pointer curve had a slope of about 0.16 (the curve for perfect scaleup has a slope of zero) across the range. It initially had the best performance

![Figure 11: Scaleup for a Well Clustered Database](image)

---

⁴Speedup performance eventually degrades as more nodes are added because adding a new node requires taking one page away from step (1)'s hash table. Eventually, losing this page hurts performance more than having less data to process helps performance.
but its performance became worse than \textit{Probe-children} by \( n=100 \) and worse than \textit{Hybrid-hash/page-pointer} by \( n=184 \). Initially, because most of the \( \text{Set}_2 \)-tuples fit in main memory at each node, its work at node was roughly a single read of \( \text{Set}_2 \) and \( \text{Set}_1 \). However, as adding nodes took more and more pages from the initial hash table, \textit{Hybrid-hash/node-pointer} had to write and read most of the \( \text{Set}_2 \)- and \( \text{Set}_1 \)-tuples. Since \textit{Probe-children} and \textit{Hybrid-hash/page-pointer} read \( \text{Set}_2 \), only once, eventually they achieve better performance than \textit{Hybrid-hash/node-pointer}.

5.5. Summary of Algorithm Comparisons

This section demonstrated that using pointer-based join algorithms can be undesirable if there is tuple-placement-skew on the inner set. If data is relatively uniformly distributed across nodes, however, such algorithms can be desirable; this is because standard \textit{Hybrid-hash}'s performance on replicated \( \text{Set}_1 \) objects will be roughly comparable to \textit{Hybrid-hash/node-pointer} in this case—good, but not necessarily the best. This section also demonstrated that algorithms that avoid replication can produce significant performance advantages. \textit{Hash-loops} looks much more attractive in an environment where using \textit{Hybrid-hash} requires replication than it did in [SHEK90] provided that most of the \( \text{Set}_1 \) objects reference \( \text{Set}_2 \) objects on a small number of nodes and that data is relatively uniformly distributed across nodes. [SHEK90] only examined the performance of algorithms that have sets of pointers from parents-to-children when there was a corresponding child-to-parent pointer; this gave more options and made \textit{Hash-loops} look less attractive. However, in an OODBS, child-to-parent pointers frequently do not exist, and each child may potentially have many parents. Thus, even in a centralized system, replication may be required in order to use \textit{Hybrid-hash} algorithms—making \textit{Hash-loops} a better choice more often than it was in [SHEK90]. We also showed that using \textit{Find-children} and a \textit{load-child} algorithm can be a clear winner at moderate memory sizes. In the presence of tuple-placement-skew, \textit{Find-children} can be indispensable to improving performance because it allows \( \text{Set}_2 \) to be redistributed across the nodes.

The four algorithms we considered must sometimes partition their join stream.\footnote{Hash-loops more attractive if each set-valued attribute contains a "large" number of oids relative to \( n \). For instance, for \( n=32 \), if each set-valued attribute contains 32 oids, \( \sigma \) is expected to be 1.37 even if no clustering was attempted and references are randomly distributed across the \( n \) nodes.} We considered a parallel version of [SHEK90]'s pointer-based nested-loops algorithm that had each \( \text{Set}_1 \) node concurrently request \( \text{Set}_2 \) pages from other nodes, since we wanted a parallel algorithm that did not require partitioning. We found that this

\begin{verbatim}
for (P of CompositePart)
  for (C of P->subparts) suchthat (C->cost > 100)
    printf("%s %s %d ",P->name,C->name,C->cost);
\end{verbatim}
offered only modest performance improvements, and only in the case where the selection predicate on Set1 was very restrictive. Thus, we concluded that the algorithm was not worth using. However, requiring partitioning does produce a non-uniform join stream since the whole join must be computed before any result tuples can be used. If a more uniform stream (where tuples are produced throughout the join process and not just at the end) is desired, a centralized pointer-based nested-loops algorithm may be a good choice.

6. Conclusions and Future Work

We described and analyzed several parallel join algorithms for set-valued attributes. We also presented the Find-children algorithm which can be used to compute an implicit extent of the objects referenced in the event that an explicit set does not exist. Using this algorithm gives the system much more flexibility in how it evaluates the join.

The comparisons demonstrate that the Hash-loops and Probe-children join algorithms can be very competitive with parallel pointer-based Hybrid-hash join algorithms. These pointer-based join algorithms show great promise for parallelizing a DBPL. Since some of the pointer-based joins were originally proposed for centralized relational database systems with referential integrity support [SHEK90], these algorithms should also be useful for such relational systems.

We are currently developing techniques to parallelize a wider class of programs written in a DBPL. We are interested in developing new parallel algorithms for processing the bulk-data structures of OODBSs and DBPLs. We would like to simulate the algorithms described in this paper to validate our cost formulas. Finally, we would like to build a parallel version of a DBPL (probably O++) that employs our algorithms (and our program transformations [LIEU92a, LIEU92b]).

7. Bibliography


