Efficiently Following Object References for Large Object Collections and Small Main Memory

Kenneth A. Ross*

Department of Computer Science, Columbia University, New York, NY 10027
kar@cs.columbia.edu

Abstract. We consider queries over large object-oriented databases in which one class of objects contains references to another class of objects. In order to answer the query efficiently, the database system needs to be able to follow object pointers from a large collection of objects in a way that minimizes the I/O cost. Traditional techniques require significant redundant I/O when both the referencing class and the referenced class are substantially larger than main memory. We propose a new technique for processing a class of object-oriented queries that is an adaptation of the Jive-Join algorithm of Ross and Li. Our algorithm applies as long as the number of disk blocks in the referenced relation is roughly of the order of the square of the number of blocks that fit in main memory. The cost of the algorithm is at most one pass through each input class extension, one pass through an index file if there is an index, and two passes through a temporary file that contains the object-identifier of the referenced object for each output tuple. Almost all of the I/O is sequential, resulting in minimal seek and rotational latencies. We analyze the cost of our algorithm, and compare our algorithm with a naive algorithm, and with an adaptation of an algorithm due to Valduriez. We demonstrate that under a wide range of circumstances, our algorithm performs significantly better than its competitors. The performance improvement is most dramatic when there is a small amount of memory, and when the input class extensions are very large.

1 Introduction

Following object references in an object-oriented database system is the fundamental operation that allows information from different objects to be combined. We may wish to follow pointers from a large collection of objects to their corresponding referenced objects in order to answer a query. To answer such queries, it is critical that we implement efficient algorithms to perform such operations.

* This research was supported by a grant from the AT&T Foundation, by a David and Lucile Packard Foundation Fellowship in Science and Engineering, by a Sloan Foundation Fellowship, by NSF grants IRI-9209029, CDA-90-24735, and by an NSF Young Investigator award.

We focus on collections of objects that are significantly larger than main memory. (We use the phrase "small main memory" to mean small *relative to the size of the data*, rather than small in absolute terms.) A number of applications need to process huge amounts of information in a reasonable time-frame. Examples include NASA's Earth Observing System, with an estimated 1 terabyte of data *per day* of information [Doz92], and data mining and decision-support applications with massive amounts of transaction information [A$^+$94]. Multimedia applications in which objects are very large also fit our model. Further, as the technology improves, more applications will emerge to take advantage of techniques for rapidly processing large volumes of data.

We consider the most basic of operations in an object-oriented database, namely dereferencing a pointer to another object. (Assume for the moment that we are talking about *physical* object identifiers, i.e., pointers to the disk location of the referenced persistent object.) When we dereference pointers from a large collection of objects, we are performing an operation that is analogous to the join operation in relational databases. We need to be able to do such dereferencing with minimal I/O overhead.

Consider the following example of an object-oriented database schema for a medical database:

```
class Patient {
 String[20] name;
 String[40] address;
 Int age;
 Condition* diagnosis; }

class Condition {
 String[30] formal-name;
 String[30] common-name;
 String[400] description; }
```

Suppose that we ask the query "For every patient who is older than 40 years, give me the name and address of the patient and the common-name and description of his or her diagnosis." Let us assume for the moment that `Patient` objects are physically clustered together on disk, and that `Condition` objects are also clustered together.

The naive way to answer this query is to iterate through each patient in the clustering order, checking the `age`, finding the condition by following the `diagnosis` pointer, and outputting the desired fields.[2] The problem with this naive approach is that references to conditions occur randomly. If the memory buffer is small compared with the size of the class of conditions, then we will need to perform I/O on almost every such reference. The two major drawbacks are (a) that disk blocks (containing `Condition` objects) are read more than once, and (b) that there are significant seek and rotational latencies between each disk block reference.

[2] If there were an index on `age`, then one would use this index to obtain the object identifiers of those patients satisfying the age criterion, and then sort the list of object-identifiers into the physical order. One would then access in order only the objects satisfying the age criterion. Without an index on `age`, though, we must examine every `Patient` object.

A less naive way to answer this query is an adaptation of a technique due to Valduriez [Val87]. One would read in as many `Patient` objects as fit in memory, and sort them based on the (pointer) value of `diagnosis`. One would then iterate through each memory-resident patient in this order, finding the condition by following the `diagnosis` pointer, and outputting the desired fields. One would then read in the next group of `Patient` records and repeat the process.

Valduriez's approach improves on the naive approach because the references to `Condition` objects are ordered according to the physical order on the disk, thus minimizing the seek and rotational latencies. Also, each block of the `Condition` class extension is read only once on each pass of the algorithm. In particular, if the algorithm takes only one pass, then both of the problems mentioned for the naive algorithm disappear. However, when multiple passes are needed Valduriez's algorithm contains some repetitious I/O. In particular, blocks may be accessed on several passes. Multiple passes are needed when the class extensions `Patient` and `Condition` are very large.

We propose a solution that does not suffer from the problem mentioned above. Our algorithm reads blocks from the extension of the class `Condition` at most once, and reads only those blocks that contain matching diagnoses. The I/O overhead is the writing and reading of a temporary file that contains the `Condition` pointer of every `Patient` object. The algorithm is an adaptation of the "Jive-join" algorithm of [RL95b].

We compare the analytic performance estimates for our extension of Jive-join against other proposed algorithms. For a range of values of the input parameters, we find that our extension of Jive-join significantly outperforms its competitors. Our improvement is most dramatic when the input class extensions are both very large and the amount of main memory is limited. This is the critical region in which to compare the algorithms. Main memory is expensive, and we would like good performance with as little main memory as possible. Further, since many users may be using a single database server, each user may be allocated only a fraction of the physical memory available.

In order to obtain such efficient performance, we use a vertically partitioned data structure for holding the output tuples. By allowing attributes to be stored in separate files, we can access one vertical fragment without having to access the whole tuples. Our algorithm takes advantage of this saving by writing the output result in two passes: one pass for the vertical fragment corresponding to the attributes from the *referencing* class extension, and one pass for the vertical fragment containing the attributes of the *referenced* class extension. We do not have to compose the whole tuples at once, which allows us to better utilize main-memory. We argue that the overhead of subsequent processing of the output result stored in this way is minimal.

2 Terminology and Assumptions

Class Extensions. We refer to a physically clustered collection of objects of the same type as a *class extension.* If R is a class extension then $||R||$ denotes the number of *objects* in R, and $|R|$ denotes the number of *disk blocks* occupied by R. $\ll R \gg$ denotes the size of an object in R.

We assume that an object-id is the physical disk address of the referenced object. We do not assume that class extensions are physically ordered by any attribute.

Queries. We assume that a query has the form outlined below. (We consider more general kinds of queries later.) There are two class extensions R_1 and R_2 with objects in R_1 containing references to objects in R_2. R_1 objects may contain a single reference to an R_2 object, or a *set* of references to such objects. The query can ask for any attributes in either the R_1 object or the referenced R_2 object, but can restrict only attributes in the R_1 object, not those in the R_2 object. (Restrictions on R_2 object attributes may be applied to the output result of our algorithm.) For simplicity, we shall assume that all of the attributes of R_1 and R_2 are required in the join result. We leave it as an easy exercise for the reader to generalize our cost formulas when only some of the attributes are required in the join result.

The set of tuples constituting the output result, i.e., the answer set for the query, is denoted by A. $||A||$ is the number of answers (i.e., tuples) in A.

Indexes. We do not assume that any indexes are available on the input object extensions, although we will take advantage of indexes if they are present. If we have an index structure on R_1, then we let J denote the set of object-identifiers of objects in R_1 that satisfy the specified criteria on the indexed attribute. If we have several index structures on different attributes, then J represents the intersection of the object-identifiers from each index. We assume that J is available on secondary storage. We write $||J||$ for the number of object-identifiers in J, and $|J|$ for the number of disk blocks occupied by J. If there are no indexed attributes that are restricted in the query, then J is (conceptually) the complete set of object identifiers in R_1. In this case, though, we would not explicitly represent J.

Selectivities and Skew. τ_1 denotes the fraction $||J||/||R_1||$, i.e., the proportion of the objects in R_1 that satisfy the index conditions. This proportion may be larger than the proportion of R_1 objects that actually participate in the output result because some R_1 objects may not satisfy conditions on non-indexed attributes. However, τ_1 does accurately represent the proportion of R_1 objects that have to be read. τ_2 denotes the fraction of the objects in R_2 that participate in the output result. (Objects in R_2 are read if and only if they participate in the output result.)

In our algorithm, we shall split objects into partitions. Our aim will be to split the objects evenly, but in the presence of skew it may not be possible to split the objects evenly. We let σ denote a "skew factor," representing the estimated ratio of the size of the largest partition to the average size of all partitions.

Other parameters. We shall assume that an in-memory pointer or an integer value, or an object-identifier occupies t bytes. We denote the size of available main memory as m blocks. We define s to be the multiple of the size of a class extension that needs to be reserved in main memory in order to sort the class extension.

2.1 Input/Output Assumptions

We assume that J and any temporary files are stored on separate disk devices from each other and from R_1 and R_2, so that we avoid unnecessary disk seeks between accesses. The input class extensions may reside on the same disk.

In our algorithm (and in other algorithms) we try to read only those blocks in the input class extensions that contain an object that participates in the answer to the query. If all blocks contain matching tuples, then we will read the whole class extension. However, if there is a relatively small number of matching objects then we will read only a fraction of each input class extension. To estimate the number of blocks read, we shall use the function $Y(k, d, n)$, which returns the optimal number of blocks needed to find k random objects out of n objects stored in d disk blocks [Yao77]. When k is large compared with d, $Y(k, d, n) \approx d$.

We do not consider input buffering in our cost model because almost all of our disk input is sequential, and we read from at most two inputs at any one time. In principle, we would have to count seek time and rotational latency each time we filled an input buffer. However, for sequential input of only two files, it is possible to choose an input buffer size that would be sufficiently large that the incurred seek time and rotational latency is insignificant. Further, one can choose large input buffers (say 100 disk blocks) while using a very small amount of main memory (which is typically many thousands of disk blocks), leaving almost all of main memory for other purposes. If we want only a handful of blocks (in order) from a single cylinder, we assume that the disk controller is intelligent enough to process the requests as a bulk request, thus incurring seek and rotational delay once for the entire cylinder, while only transmitting the requested blocks. This is realistic for modern disk devices that internally cache an entire track or cylinder. We believe that the impact of input buffering on the cost derived in this paper would be minimal, while unnecessarily complicating the formulas. To be fair, we also omit the input buffering components of the cost for competing algorithms that perform sequential input.

Some of our disk output is not fully sequential. We shall allocate disk output buffers and optimize their size to get the best I/O cost.

In some stages of our algorithm, records are accessed in order from a contiguously stored class extension. We can approximate the total seek time for one pass through class extension R as $3|R|/D$ times the average seek cost, where D is the capacity (in blocks) of the disk unit. We count three times the "average" seek cost, since the average seek cost is equal to one third of the time taken to move from one edge of the disk to the other. This rough approximation assumes that seek time can be accumulated in a linear fashion, and that there are no competing accesses to the disk device. If there was contention on the disk device between cylinder accesses, then we would have to count one seek per cylinder, since the seeks between cylinders would not necessarily be small.

We denote the block size, i.e., the number of bytes in a block, by b. The number of blocks per cylinder is denoted by c. The number of blocks per disk device is denoted by D. We denote the size of a disk block pointer by p bytes.

2.2 A Detailed Join Cost-Model

Haas, Carey and Livny have proposed a detailed cost model in which seek time and latency time are explicit [HCL93]. These authors reexamine a number of ad-hoc join methods using their cost model, and demonstrate that the ranking of join methods obtained by using a block-transfer-only cost model for I/O may change when the same algorithms are analyzed using the more detailed cost model. We shall use this detailed cost model. The total I/O cost of a join algorithm is measured as

$$N_S T_S + N_{I/O} T_L + N_X T_X.$$

In this formula, T_S is the time taken for an average disk seek, while N_S is the number of seeks incurred by the algorithm. T_L is the rotational latency time, equal to the time for half a rotation; since every disk I/O to a new cylinder incurs disk latency, T_L is multiplied by $N_{I/O}$, the number of disk accesses (to new cylinders). T_X is the disk transfer time for a disk block of data, and N_X is the number of block transfers.

Following [HCL93], we choose values of T_S, T_L, and T_X to model an existing disk drive, namely the Fujitsu M2266 described in [B+92]. The parameters are $T_S = 9.5$ msec., $T_L = 8.3$ msec., and $T_X = 2.6$ msec., with the block size $b = 8192$ bytes, the number of blocks per cylinder $c = 83$, and a total capacity $D = 130000$ blocks (1 gigabyte).

We choose to ignore CPU cost, and focus on the I/O cost. There are several reasons for this choice. The first is that CPU cost is significantly smaller than the I/O cost when the input class extensions are much bigger than main memory. Almost all of the CPU-intensive work can be done while waiting for disk I/O. Most of the CPU cycles in the algorithms of this paper are spent sorting; past work on sorting has shown that sorting is I/O-bound, even for systems with high-speed disk arrays [N+94]. In any case, it is difficult to predict the CPU performance of a sorting algorithm without taking into account effects such as cache behavior [N+94].

We do not include measures of the cost of writing the output result, or the cost of computing the set J of object-identifiers from the indexes on R_1, since these measures will be the same for all algorithms.

2.3 A Partitioned Data Structure for the Output Result

Our physical storage for the query output result uses a technique designed to minimize the amount of I/O. In particular, the attributes of the output result are stored separately in two different files, using a kind of vertical fragmentation. The attributes from the referencing class extension are stored in one file, while the attributes from the referenced class extension are stored in another. The first entry in each of the files corresponds to the first pair of objects, the second entry to the second pair, and so on. There is no need for any additional stored information This vertically fragmented data structure has been termed a "transposed file" [Bat79].

3 Adapting Jive-Join to Object-Oriented Databases

Jive-Join is an algorithm for performing joins in a relational database system using a join index [RL95b]. In this section we adapt the algorithm to apply to object-oriented databases. We will measure the cost of the new algorithm using the formula of Section 2.2. Our algorithm uses an internal data structure which we shall refer to as a disk buffer. The structure of a disk buffer is given in Figure 1.

<table>
<tr><td colspan="2">Object-id value from R_2.</td></tr>
<tr><td>Pointer to the start of a contiguous segment of disk memory for the output file.</td><td>Pointer to the current position within the segment of disk memory for the output file.</td></tr>
<tr><td colspan="2">A sequence (of length up to x) of blocks to be written to an output file. These blocks contain the requested attributes from R_1.</td></tr>
<tr><td>Pointer to the start of a contiguous segment of disk memory for the temporary file.</td><td>Pointer to the current position within the segment of disk memory for the temporary file.</td></tr>
<tr><td colspan="2">A sequence (of length up to v) of blocks to be written to the temporary file. These blocks contain object-ids from R_2.</td></tr>
</table>

Fig. 1. Disk buffer data structure.

When the sequence of records for the output file becomes full, i.e., there are x complete blocks, these blocks are flushed to disk at the current disk location, and the current disk location is incremented by x. Similarly, when the sequence of records for the temporary file becomes full, i.e., there are v complete blocks, then these blocks are flushed to disk at the current disk location, and the current disk location is incremented by v.

In the algorithm below we shall have an array of disk buffers. The object-ids from R_2 will be in increasing order in the array. The idea will be to partition the R_2 objects evenly among the buffers. An object will be placed in a buffer B if its R_2 object-id is at least the object-id of B, but not more than the object-id of the next buffer after B. We assume that the first object-id value in the array is smaller than the initial object-id in R_2. The set of objects that pass through a single disk buffer is called a *partition*. Given a particular object-id from R_2, we can apply binary search to the array to determine the appropriate buffer for that tuple-id. We assume that the disk pointers in consecutive buffers are to adjacent regions in disk memory, so that their physical order corresponds to the order of the partitions. The algorithm consists of three steps:

Step 1 The first step of the algorithm is to allocate an array (of size y) of disk buffers in memory. We specify values of the R_2 object-id value for each buffer, and allocate the appropriate segments of disk memory. See Section 3.2 for a description of how the partitioning values are chosen.

Step 2 We scan J and R_1 sequentially, in a fashion similar to a merge-join on the R_1 object-id. (Remember that J is in R_1 object-id order.) We examine a block of R_1 only if it has an object referred to in J. We check conditions on attributes of R_1 that are not indexed, and discard the object if it fails the check. On each remaining object, we identify the disk buffer B to which this object belongs, based on the R_2 object-id. (If the R_1 object refers to a *set* of R_2 objects then we perform the following steps for each such reference.[3]) We perform two operations on the buffer:

(a) The attributes of R_1 that are required for the output result are written as a tuple to the output sequence of B.
(b) The R_2 object-id is written to the temporary file sequence of B.

When a sequence of blocks in a disk buffer is full, it is flushed to disk as previously described. When R_1 is exhausted, a record is appended to each disk buffer sequence indicating where the subsequent partition begins on disk, and all disk buffers are flushed to disk.

After finishing Step 2, we have generated half of the output, namely that vertical fragment of the output corresponding to the attributes from R_1. We have also generated a temporary intermediate file that is used in Step 3 below to generate the other half of the output.

Step 3 For each partition (in order) of the temporary file we perform the following operations. We read into memory the whole partition, and sort the R_2 object-id column in memory in ascending order. (We also keep the original version of the temporary file.) We then retrieve objects from R_2 in order, retrieving only blocks that contain a matching object according to our sorted version of the temporary file. For each matching R_2 object read, the attributes of the object needed for the output result are appended to an in-memory sequence of tuples.

We keep reading objects from R_2 until we have finished the partition. At that point we write the "R_2-segment" of the output as follows. We look at the original version of the temporary file, and write the corresponding R_2 tuples in that order to a segment of the output result. The R_2 tuples can be found

[3] We discard any R_1 object that has a NULL R_2 reference. If R_1 objects with NULL R_2 references are wanted in the output result, then we can put these R_1 objects in a special buffer at the end of the buffer sequence; the result would be analogous to a left outer-join.

each time using binary search on the ordered version of the temporary file to determine the offset into the array.

We then continue with the next partition, and so on. By the time we have finished with the final partition, we have generated the R_2 portion of the output result. With the R_1 portion generated in Step 2, we have the required result. Note that in Steps 2 and 3 we make sure not to read a block from either R_1 or R_2 if it is known not to contain an object participating in the join. That way, we will get better performance if only a small proportion of each input class extension participates in the output result.

3.1 Memory Requirements

We need Step 1 and Step 2 to fit in main memory. $y(x+v+\frac{t+4p}{b})$ blocks are used for the disk buffers. Hence, removing insignificant terms, the following inequality must hold:

$$y(x+v) \leq m. \tag{1}$$

Step 3 must also fit in main memory. The total size of the partitions of the temporary file is z blocks, where $z = ||A||t/b$. The total size of the corresponding R_2 objects is $\tau_2|R_2|$ blocks, assuming that all of the attributes of R_2 are required in the output result. For each partition of size h, we need sh blocks as working space in order to do the sort in memory, plus and extra h to keep the original sorted temporary file. Thus, we get

$$(s+1)z + \tau_2|R_2| \leq ym/\sigma. \tag{2}$$

We let the variable L denote $(s+1)z + \tau_2|R_2|$. A subtle point in Equation 2 is the potential presence of skew. It may conceivably happen that a few objects participate many times in the output result, while other objects participate only once. Thus, we may need to divide Step 3 of Jive-join in such a way that the individual partitions have different sizes. This issue will be discussed further in Section 3.2. Combining Equations 1 and 2 with the constraint that $x + v \geq 2$, and assuming that $s = 1$ yields

$$m \geq \sqrt{2\sigma(2z + \tau_2|R_2|)}. \tag{3}$$

Equation 3 specifies the minimum amount of memory necessary for the extension of Jive-join to be applicable. This is a very reasonable condition, stating that the number of blocks in main memory should be at least of the order of the square root of both the number of participating blocks in the referenced relation, and the number of blocks in the temporary file. Note that R_1 may be much larger than R_2, since the contribution from R_1 (via z) has a much smaller effective coefficient than the contribution from R_2.

To get an idea of how lenient the constraint of Equation 3 is in practice, imagine we had 128 megabytes of main memory, that disk blocks were 8K bytes, and that we had a one-to-one relationship between objects in R_1 and R_2, with full participation by both object extensions. Assuming that objects in R_2 are much wider than an object-id, and that our skew factor σ is 1.2, we would be able to apply the extension of Jive-join for R_2 of size up to 850 gigabytes.

3.2 Choosing the Partitioning Values

We now show how to choose the partitioning elements in Step 1 of Jive-join. A first attempt might be to partition the object-ids evenly. Since the number of objects in R_2 is known, and since the object-ids are uniformly distributed within the R_2 class extension, we can simply divide the tuple-id range into y equal-sized partitions.

This approach would work if the distribution of tuples *in the output result* was uniform. However, for distributions with significant skew, we may find that some partitions contain many more participating objects than others. For all partitions to fit in main memory, we would have to ensure that the largest partition, together with its fragment of the temporary file, fits in main memory. We thus waste some memory for all other partitions.

An alternative approach would be to sample the R_1 objects occurring in J to obtain their R_2 object identifiers. One would partition the sample, and expect that, for a sufficiently large sample, the partitioning elements of the sample are close to the partitioning elements of the full extension of R_1. An analysis of this kind of sampling approach is provided in [DWNS91, SN91]. We need to balance the I/O cost of the initial sampling step with the probability that skew will make one of the partitions in our algorithm too big to fit in main memory in Step 3.

Whether or not we perform sampling, we assume a skew factor σ determines the ratio of the size of the largest partition to the average partition size. We will have to take σ into account when allocating space for the partitions in memory. We do not include the cost of determining the partitioning values. This cost is likely to not be significant, compared with the other costs involved; also, the cost of sampling to find the partitions can be amortized over several executions of the query at different times, since a small number of updates is unlikely to significantly affect the partitioning values.

3.3 Measuring the Cost

We now calculate the values of N_S, $N_{I/O}$, and N_X in order to measure the cost of our algorithm. We assume that the y partitioning values have already been chosen and do not need any significant I/O to read in. Thus, there is no measured I/O in Step 1.

Let n_1 denote the number of blocks in the output file containing attributes from R_1, and similarly for n_2 and R_2. Then $n_i = ||A||* \ll R_i \gg /b$.

The number of seeks in Step 2 is $3|J|/D$ for J and $3|R_1|/D$ for R_1 (since they are read sequentially and they reside on different disks), plus one seek for each buffer flush. The number of buffer flushes is $z/v + n_1/x$. The number of seeks in Step 3 is $3|R_2|/D$ for R_2 (since R_2 is read sequentially), plus one seek each time one switches partitions in the temporary file. The number of partition/segment switches is $y-1$, but since the partitions are ordered, the total time spent seeking corresponds to a unidirectional traversal across the class extension on disk. Thus, we obtain the formula

$$N_S = \frac{3}{D}\left(|J| + |R_1| + |R_2| + z\right) + z/v + n_1/x.$$

The number of I/O requests in Step 2 is $|J|/c$ for J and $Y(||J||, |R_1|/c, ||R_1||)$ for R_1, plus z/v I/O requests to write the temporary file. We also have n_1/x requests to write the "R_1-fragment" of the output file. The number of I/O requests in Step 3 is $Y(\tau_2||R_2||, |R_2|/c, ||R_2||)$ for R_2, plus one new request each time one switches partitions in the temporary file, plus one request each time one switches segments in the "R_2-fragment" of the output file. The number of partition/segment switches is $y - 1$. Thus, we obtain the formula

$$\begin{aligned} N_{I/O} = {} & 2y - 2 + |J|/c + Y(||J||, |R_1|/c, ||R_1||) \\ & + Y(\tau_2||R_2||, |R_2|/c, ||R_2||) + z/v + n_1/x. \end{aligned}$$

The number of block transfers in Step 2 is $|J|$ for J and $Y(||J||, |R_1|, ||R_1||)$ for R_1. Additionally, we need z block transfers to write the temporary file. The number of block transfers in Step 3 is $Y(\tau_2||R_2||, |R_2|, ||R_2||)$ for R_2, plus z I/Os to read the temporary file. Since the output result block transfers are the same for every join algorithm, we do not count them here. Thus, we obtain the formula

$$N_X = |J| + Y(||J||, |R_1|, ||R_1||) + Y(\tau_2||R_2||, |R_2|, ||R_2||) + 2z.$$

In the event that all blocks of the input class extensions participate in the output result, we can simplify the equation above to $N_X = |J| + |R_1| + |R_2| + 2z$.

Now we are in a position to choose optimal values for x, y, and v. Since we want to use as much memory as is available, we can interpret Equation 1 as stating that $y(x + v) = m$. If one looks at the cost function, one can isolate the part that depends on x, y and v as

$$c(x, y, v) = (\frac{z}{v} + \frac{n_1}{x})T_S + (2y + \frac{z}{v} + \frac{n_1}{x})T_{I/O} \qquad (4)$$

Some elementary calculus, together with the constraints of Section 3.1, show that this expression is minimized with respect to x, y and v when

$$x = \frac{m^2}{L\sigma(1+\sqrt{z/n_1})},\ v = \frac{m^2}{L\sigma(1+\sqrt{n_1/z})},\ y = \frac{L\sigma}{m}.$$

4 A Comparison of the Algorithms

We now compare the analytic performance of our extension of Jive-Join against two other algorithms. We compare firstly with the naive approach outlined in Section 1 and discussed in Appendix B. We also compare our algorithm with the modified algorithm of Valduriez, also outlined in Section 1 and presented in detail in Appendix A. We use the detailed cost model of [HCL93] for the comparison. Our comparisons will compare the I/O time taken to compute the output result against the memory size for a variety of input class extensions. The horizontal axis is the number of megabytes of main-memory available, and the vertical axis is the number of hours taken in I/O time by each algorithm. Bear in mind that the performance figures refer to a disk with a 3.1 MB/sec throughput. Faster disks or parallel disk drives would substantially reduce the time results. For reference we shall also include the lower-bound cost, namely the cost of sequentially reading J and all the participating blocks in R_1 and R_2. (If we use no indexes on R_1, then the lower-bound cost does not include the cost of reading J.)

For some of the examples, the sizes of the relations may be larger than the size (1 gigabyte) of the reference disk drive mentioned in Section 2.2. For these comparisons we shall assume that the data is spread over multiple disk units. While in principle it may be possible to parallelize the I/O to multiple disk units, we shall not do so here.

For our extension of Jive-join, the number of disk blocks transferred is independent of the memory size. Thus we expect the Jive-join curve to be relatively flat, increasing only when memory is scarce when there are many seeks and small I/O requests.

There are more interesting scenarios than we can present in this abstract. We have selected a handful that are representative. In Examples 1, 2, and 3 we suppose that the size of an object in both R_1 and R_2 is 256 bytes, so that there are 32 objects per block in each relation.

Example 1. In our first example, let us take $|R_1| = |R_2| = 2^{20} \approx 10^6$ blocks, so that the size of each input class extension is 8 gigabytes. $||R_1|| = ||R_2|| = 2^{25} \approx 33 \times 10^6$ objects. We assume $t = 4$ bytes. We consider several scenarios.

(a) The relationship between R_1 and R_2 is one-to-one with full participation of both R_1 and R_2, and we access all objects in R_1. In this case $||J|| = ||A|| = ||R_1|| = ||R_2||$, $\tau_1 = \tau_2 = 1$, and $|J| = 2^{14}$ blocks. The performance graph for this scenario appears in Figure 2 (a). Jive-join performs close to optimal, while Valduriez's algorithm performs significantly worse. The naive algorithm is off the scale, taking 196 hours.

(b) The relationship is many-to-many with full participation of both R_1 and R_2. We choose $||A|| = 16 * ||R_1|| = 2^{29}$ so that, on average, each R_1 object references 16 R_2 objects. The performance graph for this scenario is similar to part (a) and is omitted. The naive algorithm takes more than 3,000 hours on the range of this graph.

(c) The relationship is one-to-one with partial participation of both R_1 and R_2. We choose $||J|| = ||A|| = \tau_1||R_1|| = \tau_2||R_2|| = 2^{19}$. The performance graph for this scenario appears in Figure 2 (c). In this example and in part (d) below, Jive-join performs better than Valduriez's algorithm for small memories, while they perform equivalently for memories larger than some threshold value.

(d) The relationship is many-to-many with partial participation of both R_1 and R_2. We choose $||A|| = 2^{25}$, $\tau_1||R_1|| = \tau_2||R_2|| = 2^{19}$. The performance graph for this scenario is qualitatively similar to that of part (c) and is omitted. □

Example 2. In this example, we consider class extensions of vastly differing size. Let us take $|R_1| = 2^{27} \approx 1.3 \times 10^8$ blocks (1 terabyte), and $|R_2| = 2^{20} \approx 10^6$ blocks (8 gigabytes). Then $||R_1|| = 2^{32} \approx 4.3 \times 10^9$ objects and $||R_2|| = 2^{25} \approx 33 \times 10^6$ objects. We assume $t = 4$ bytes.

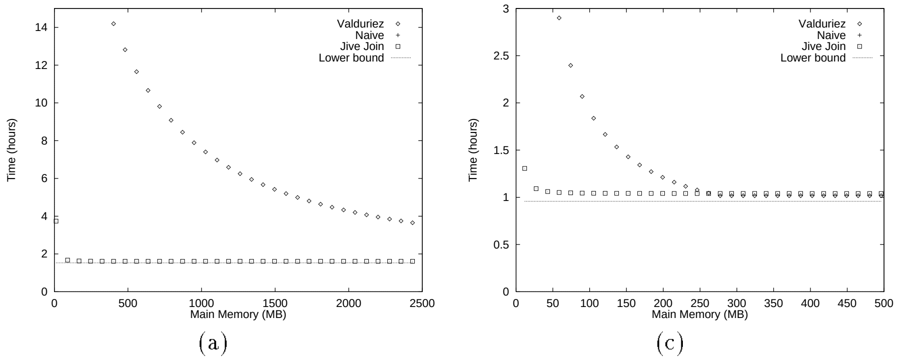


Fig. 2. Performance comparison for Example 1.

(a) The relationship between R_1 and R_2 is many-to-one with full participation of both R_1 and R_2. In this case $||J|| = ||A|| = ||R_1||$. The performance graph for this scenario appears in Figure 3 (a). Valduriez's algorithm takes more than 1,000 hours over the whole range, and so doesn't appear in the graph. Jive-join performs close to the optimal. The naive algorithm takes over 25,000 hours (2.8 years)!
(b) The relationship is one-to-one with full participation of R_2, but partial participation of R_1. $||A|| = ||J|| = ||R_2||$, $\tau_2 = 1$. The performance graph for this scenario appears in Figure 3 (b). Because of the low selectivity in R_1, Valduriez's algorithm does much better for this example than for part (a), but still significantly worse than Jive-join.
(c) The relationship is one-to-one with partial participation of both R_1 and R_2. We choose $||A|| = ||J|| = \tau_1||R_1|| = \tau_2||R_2|| = 2^{19}$. The performance graph for this scenario appears in Figure 3 (c). The naive algorithm just appears on the graph, taking almost 6 hours. Valduriez's algorithm and Jive-join perform comparably, and significantly above the lower bound due to the relatively large contribution of rotational latency: there is on average one matching object in R_1 every 3.1 cylinders.
(d) The relationship is many-to-many with partial participation of both R_1 and R_2. We choose $||A|| = 2^{25}$, $||J|| = \tau_2||R_2|| = 2^{19}$. The performance graph for this scenario appears in Figure 3 (d). Again, the naive algorithm does not appear on the graph; it took 193 hours over the range of this graph. The explanation for this graph is similar to that of part (c). □

Example 3. In this example, we consider class extensions that could fit into main memory. Let us take $|R_1| = |R_2| = 2^{14} = 16384$ blocks (128 megabytes). Then $||R_1|| = ||R_2|| = 2^{19} \approx 5.2 \times 10^5$ objects. We assume $t = 4$ bytes bytes. We consider two scenarios.

(a) The relationship between R_1 and R_2 is one-to-one with full participation of both R_1 and R_2. In this case $||A|| = ||J|| = ||R_1||$, and $\tau_2 = 1$. The

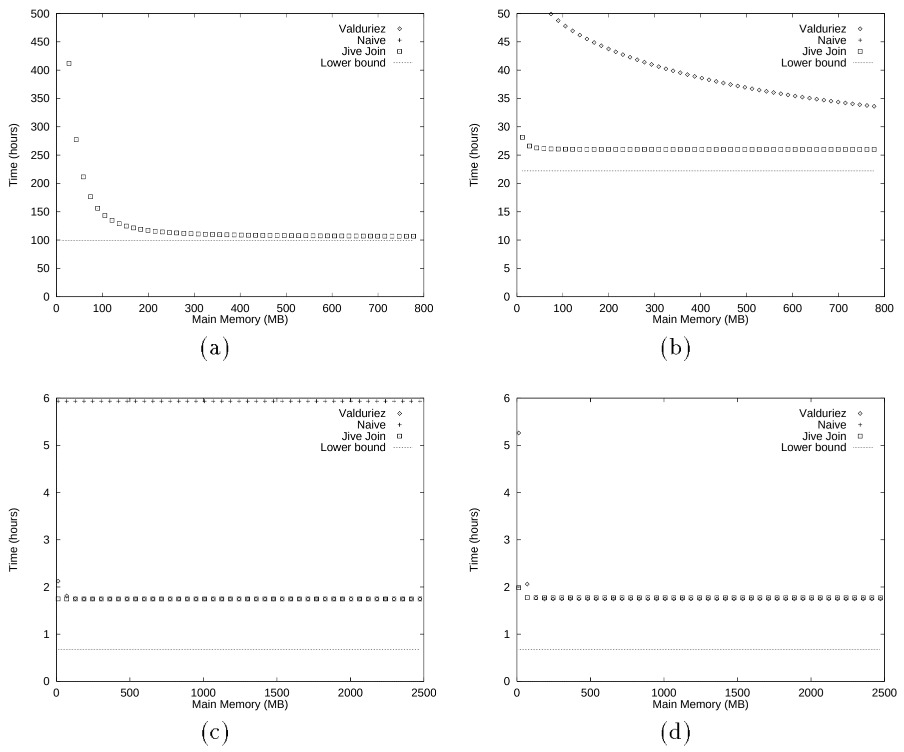
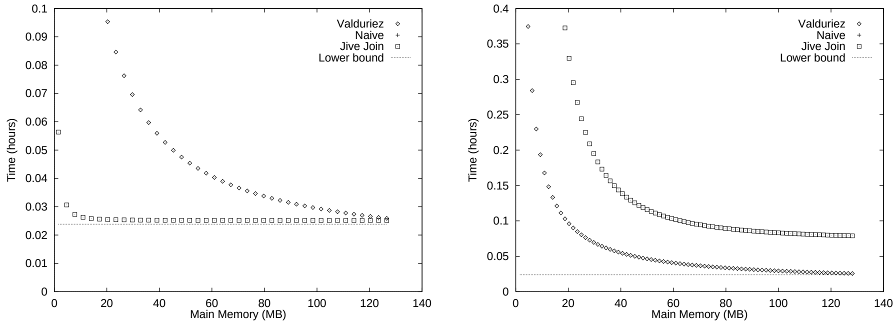


Fig. 3. Performance comparison for Example 2.

performance graph for this scenario appears in Figure 4 (a). Both algorithms converge on the optimal time as memory approaches the size of the class extensions. Jive-join still outperforms the other algorithms for small memories. The naive algorithm takes over 3 hours.

(b) The relationship is many-to-many with full participation of R_1 and R_2, and $||A|| = ||J|| = 2^{26} \approx 6.7 \times 10^7$, and $|J| = 2^{16} = 65536$ blocks. The performance graph for this scenario appears in Figure 4 (b). The naive algorithm takes 380 hours. In this scenario, Valduriez's algorithm performs better than Jive-join. The reason for the observed behavior is that Jive-join has a cost component that is proportional to the size of the output result, namely the cost of writing and reading the temporary file. In this case, the temporary file is significantly larger than the input class extensions. This is the typical case in which Jive-join performs worse than Valduriez's algorithm. However, if the temporary file is large, then the output result must also be large: For this example, the output cost is over 3 hours, and would dominate the total cost. □

Jive-join is an improvement over Valduriez's algorithm for large class extensions because it limits class extension accesses to single sequential scans of each



(a) (b)

Fig. 4. Performance comparison for Example 3.

input. Valduriez's algorithm processes the class extensions sequentially, but processes one of the class extensions multiple times. In Example 1 (a), Valduriez's algorithm makes 8 passes through R_2 for main memory of 1100 megabytes, and 4 passes through R_2 at 2200 megabytes.

The fundamental conclusions to be drawn from these results are:

(a) Jive-join performs better than its competitors in a wide range of settings. When the main memory is small compared to the input class extensions, the improvement can be orders of magnitude. This is the crucial range in which to evaluate the algorithms: Main memory is expensive, and we would like to get good performance with as little main memory as possible.

(b) Jive-join may perform worse than Valduriez's algorithm in a situation in which the size of the temporary file is particularly large. However, when the temporary is large relative to the input class extensions the output result will be huge, and writing the output result will dominate the join cost anyway.

5 Further Issues

Referencing More Than Two Objects from R_1. One can extend the algorithm described in this paper to the case where there are several objects from several class extensions referenced by objects in R_1. The details are beyond the scope of this paper. See [RL95a] for a description of how multiple participating relations are handled.

Nested References. It is much more difficult to extend the algorithm presented here to queries in which the R_2 objects contain references to a third class R_3 of objects that are referenced by the query. The difference between this case and the one above is that when an R_1 object references several objects, we can process the R_1 object based on the *object-identifiers* for each referenced object. With the nested reference, we don't get the R_3 identifier until we've already looked-up the R_2 object, in which case there is less freedom to order the accesses to the R_2 and R_3 objects without performing redundant I/O.

One could apply Jive-join multiple times, one time for each level of nesting. However, we would have to read and write temporary intermediate results, increasing the I/O costs. A better solution might be to maintain (or build) a *path index* containing all of the object-ids along each path of nested references. This path index could be treated like a join index, and processed as in [RL95a].

For the reason mentioned above, it would be significantly more difficult to use a version of this algorithm in a system with *logical* object identifiers. The logical-to-physical mapping must, in general, be achieved using I/O, and the pattern of access is similar to the case with the nested R_3 object. Even if the logical-to-physical mapping table could fit into memory, it would use space reducing the effective space for the Jive-join.

A similar problem occurs when the query restricts an attribute mentioned in R_2. Since we process R_1 and R_2 separately, we cannot know at the time we're processing the R_1 objects whether their R_2 objects will satisfy the query's restrictions. We could apply our algorithm if the R_1 objects "cached" some of the R_2 attributes, but the tradeoffs inherent in such an approach can only be measured in the context of a complete system.

Differences from the Relational Version of Jive-Join. The basic ideas for Jive-join and its analysis come from [RL95b]. We now discuss the basic differences between this paper and [RL95b]. The most obvious difference is that here we do not have access to the join index. Instead, we use the object-identifiers embedded in the referencing objects. There are several consequences of this difference. Firstly, there is a fundamental *asymmetry* between R_1 and R_2 since the referencing object has to be accessed in order to find the referenced object. In a value-based relational join, there is a symmetry between the two input relations: either relation may be denoted by R_1 or R_2.

A second consequence of the absence of the join index is that we have to deal with skew. With the join index present (or, in the process of building the join index) we have enough information to perfectly partition the input relations. Without the join index, we have to resort to sampling techniques, and set aside some memory "leeway" to allow for skew.

6 Conclusions

We have presented an adaptation of Jive-join for following object references in an object-oriented database. Our algorithm requires one pass through each input class extension, one pass through a set of index records (if an index exists), and two passes through a temporary file of object-identifiers. For small memories and large object collections this performance is a significant improvement over a straightforward extension of Valduriez's algorithm, which needs to make multiple passes over one of the input class extensions. Our extension of Jive-join has a number of good properties:

- It applies under a relatively lenient condition in which the referenced class extension is assumed to take a number of blocks less than half the *square* of the number of blocks in memory.

– It writes the output result in two phases, into two separate files. By doing so it avoids having to reread part of the output result.
– It performs better than its competitors in a wide range of settings. For small memories and large input class extensions the improvement can be dramatic.
– Its performance is worse than Valduriez's algorithm only when the size of the temporary file is particularly large. In this case, the output result will also be particularly large, and the cost of writing the output will dominate the total cost.

Our extension of Jive-join would be an important component of the query processing subsystem of an object oriented database system. We are presently examining potential platforms for the implementation of our algorithm.

References

[A+94] R. Agrawal et al. Quest: A project on database mining. In *Proceedings of the ACM SIGMOD Conference*, page 514, May 1994.

[B+92] K. Brown et al. Resource allocation and scheduling for mixed database workloads. Technical Report 1095, University of Wisconsin, Madison, 1992.

[Bat79] D. S. Batory. On searching transposed files. *ACM Transactions on Database Systems*, 4(4):531–544, 1979.

[Doz92] J. Dozier. Access to data in NASA's Earth Observing System. In *Proceedings of the ACM SIGMOD Conference*, page 1, June 1992.

[DWNS91] D. De Witt, J. F. Naughton, and D. A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proceedings of the Conference on Parallel and Distributed Information Systems*, pages 280–291, 1991.

[HCL93] L. M. Haas, M. J. Carey, and M. Livny. Seeking the truth about ad hoc join costs. Technical Report RJ9368, IBM Almaden Research Center, 1993.

[N+94] C. Nyberg et al. Alphasort: A RISC machine sort. In *Proceedings of the ACM SIGMOD Conference*, pages 233–242, May 1994.

[RL95a] K. Ross and Z. Li. Efficiently joining multiple large relations. Submitted for publication, 1995.

[RL95b] K. A. Ross and Z. Li. Jive-join and Smash-join: Efficient join techniques for large relations and small main memory. Submitted for publication, 1995.

[SN91] S. Seshadri and J. F. Naughton. Sampling issues in parallel database systems. (manuscript), 1991.

[Val87] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2):218–246, 1987.

[Yao77] S. B. Yao. Approximating block accesses in database organizations. *Communications of the ACM*, 20(4):260–261, 1977.

A The Algorithm of Valduriez

In this section we describe an adaptation of the algorithm of Valduriez to compute the output result. We then derive a detailed cost measurement for this algorithm using the formula of Section 2.2.

We first read in (sequentially) as much of J and R_1 as will fit in memory. We read a block from R_1 only if it contains an object whose object-id is in J. Those objects of R_1 that match an object-id in J are kept in memory, but the actual match with R_2 is not computed at this stage. When an object from R_1 is read into memory, a pair consisting of its memory address and its R_2 pointer is appended to an array of such entries. (We repeat this step for each R_2 object referenced by a given R_1 object.) We continue until the parts of R_1 and J read use all of main memory (except for some auxiliary space set aside for in-memory sorting). Let us call the array of pairs $\overline{J}$.

$\overline{J}$ is sorted by the object-id value from R_2. Records in $\overline{J}$ are processed one by one; for each record, the corresponding R_2 object-id is located, the matching object is retrieved, the corresponding R_1 object is located in memory, and the resulting output tuple is written to the output file.

If J and R_1 have been exhausted, then we are finished. If not, we repeat the previous steps until all of J and R_1 are consumed.

Measuring the Cost Let u denote the number of passes made in the algorithm. We can calculate u as $(2|J| * s + |J| + \tau_1|R_1|)/m$. We can compute K, the number of blocks of R_2 accessed in one pass, using the formula $K = Y((1-(1-1/u)^g)\tau_2||R_2||, |R_2|, ||R_2||)$ where $g = ||A||/(\tau_2||R_2||)$.[4] One can then derive the formulas

$$\begin{aligned}
N_S &= 2u + \tfrac{3}{D}(|J| + |R_1| + u|R_2|) \\
N_{I/O} &= 2u + |J|/c + Y(||J||, |R_1|/c, ||R_1||) + u * Y(||A||/u, |R_2|/c, ||R_2||) \\
N_X &= |J| + Y(||J||, |R_1|, ||R_1||) + u * K
\end{aligned}$$

B The Naive Method

For simplicity we shall take a pessimistic approach and assume that every access to a record in R_2 requires a block transfer, with an associated seek time and rotational latency. This is realistic when R_2 is large compared with main memory, and so the system cannot effectively buffer R_2 blocks for later use. J can be read sequentially without seeks or rotational latency, since we assume that it is on a separate disk device. However, since R_1 and R_2 are on the same disk, their accesses interfere, and each block access to R_1 requires a seek and rotational latency.

$$\begin{aligned}
N_X &= |J| + Y(\tau_1||R_1||, |R_1|, ||R_1||) + ||A|| \\
N_{I/O} &= 1 + Y(\tau_1||R_1||, |R_1|, ||R_1||) + ||A|| \\
N_S &= 1 + Y(\tau_1||R_1||, |R_1|, ||R_1||) + ||A||
\end{aligned}$$

[4] This is not the estimate derived by Valduriez in [Val87]. See [RL95b] for an explanation of why this is a better estimate of K than that given by Valduriez.