VLDB Journal, Vol 7, Number 4, 1998 (to appear)

Fast Joins Using Join Indices

Zhe Li Kenneth A. Ross*
Department of Computer Science, Columbia University, New York, NY 10027
li,kar@cs.columbia.edu

Abstract

Two new algorithms, "Jive-join" and "Slam-join," are proposed for computing the join of two relations using a join index. The algorithms are duals: Jive-join range-partitions input relation tuple-ids then processes each partition, while Slam-join forms ordered runs of input relation tuple-ids and then merges the results. Both algorithms make a single sequential pass through each input relation, in addition to one pass through the join index and two passes through a temporary file whose size is half that of the join index. Both algorithms require only that the number of blocks in main memory is of the order of the square root of the number of blocks in the smaller relation. By storing intermediate and final join results in a vertically partitioned fashion, our algorithms need to manipulate less data in memory at a given time than other algorithms. The algorithms are resistant to data skew and adaptive to memory fluctuations. Selection conditions can be incorporated into the algorithms. Using a detailed cost model, the algorithms are analyzed and compared with competing algorithms. For large input relations, our algorithms perform significantly better than Valduriez's algorithm, the TID-join algorithm, and hash join algorithms. An experimental study is also conducted to validate the analytical results and to demonstrate the performance characteristics of each algorithm in practice.

Keywords: Query processing, decision support systems.

1 Introduction

A number of applications need to process huge amounts of information in a reasonable time-frame. Examples include commercial decision-support systems, NASA's Earth Observing System with an estimated 1 terabyte of data *per day* of information [Dozier, 1992], and data mining applications with massive amounts of transaction information [Agrawal et al., 1994]. Further, as technology improves, we expect that more applications will emerge to take advantage of techniques for rapidly processing large volumes of data. Thus, we focus on relational[1] databases in which relations are significantly larger than main memory.

The *join* operation of the relational database model is the fundamental operation that allows information from different relations to be combined. Joins are typically expensive operations. Therefore, it is critical to implement joins in the most efficient way possible. The term "ad-hoc" join is used to describe the process of taking two relations and forming their join without the benefit of any pre-computed specialized data structures such as indexes. A number of techniques have been developed to perform joins in this setting [Blasgen and Eswaran, 1977, Bratbergsengen, 1984, DeWitt et al., 1984, Kim, 1980, Mishra and Eich, 1992, Shapiro, 1986].

Decision support systems are characterized by complex ad-hoc join queries over large data sets. Relational systems relying on conventional data structures and ad-hoc join methods fail to deliver the needed response time for many such interactive queries. One commonly used technique for speeding up join query processing is the precomputation of some of the join information.

In this paper, we consider joins of relations for which there exists a pre-computed access structure, namely a join index [Valduriez, 1987]. A join index between two relations maintains pairs of identifiers of tuples that

*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.

[1]Our algorithms also apply for other database models, including object-oriented databases.

would match in case of a join. The join index may be maintained by the database system, and updated when tuples are inserted and deleted in the underlying relations. In situations where joins are taken often, the cost of doing this maintenance can be more than offset by the savings achieved in performing the join. A number of commercial decision support systems are rumored to be using join indexes [O'Neill and Graefe, 1995].

In [Valduriez, 1987], Valduriez proposed and analyzed a join algorithm that uses the join index. The most important conclusion of that study was that, under many circumstances, having the join index allows one to compute the join significantly faster than the "best" ad-hoc methods such as Hybrid hash-join [DeWitt et al., 1984]. Valduriez's algorithm is relatively easy to implement, and is particularly effective for selective joins. Valduriez's work also led to other conceptual advances, such as path indices and access support relations [Kemper and Moerkotte, 1990].

In terms of performance, however, Valduriez's algorithm can perform a significant amount of repetitious I/O. Blocks are accessed often for only a small fraction of their tuples. The same block may be read multiple times on different passes within the algorithm.

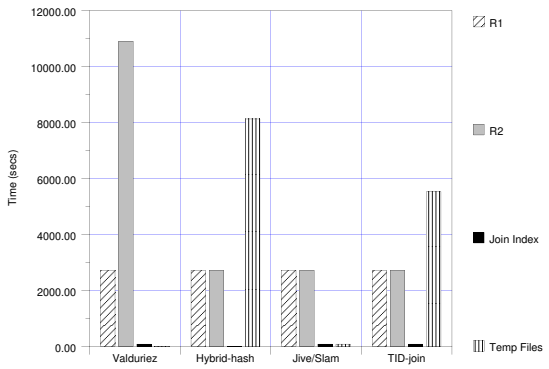
Another algorithm that uses the join index is a TID-join algorithm based on work of Kooi [Kooi, 1980]. As we shall see, that algorithm often requires large intermediate files that incur a significant I/O cost.

We propose two algorithms that significantly improve upon Valduriez's algorithm and the TID-join algorithm. The algorithms are called "Jive-join" and "Slam-join." The two algorithms are duals of one another, and have very similar performance. Jive-join range-partitions the tuple-ids of the second input relation, then processes each partition separately. Slam-join forms sorted runs of tuple-ids from the second input relation, then merges those runs. Each algorithm has features that make it preferable to the other depending on the context in which it is being used.

Important features of our algorithms include:

- Almost all of the I/O performed is sequential.
- A block of an input relation is read if and only if it contains a record that participates in the join.
- The join index is read once.
- A set of temporary files, with total size half that of the join index, is written and then read.
- A single pass of each input relation can be guaranteed as long as main memory is at least $\sqrt{|J| + 2\tau|R|}$ blocks, where $|J|$ is the number of blocks in the join index, R is the smaller input relation, and $\tau|R|$ is the number of blocks occupied by tuples from R that participate in the join.
- Skew does not affect the performance.
- Slam-join can adapt to memory fluctuations.
- Recursive application of the algorithms is possible when the memory does not satisfy the bound given above. The additional cost is one pass through a fragment of one of the inputs.
- An extension to join multiple relations is possible, retaining the single-pass property of the inputs, by using multidimensional data structures.

The crucial virtue of both algorithms is that (a) they make just *a single read pass* through each input relation under lenient memory requirements, and (b) the only temporary results stored are tuple-identifiers which are usually small. In contrast, Valduriez's algorithm makes multiple passes through one of the relations when for each input relation the total size of the participating records is larger than main memory. The TID-join makes a single pass through each input, but it writes and reads intermediate results that can be very large. This is summarized in the following chart that shows the various components (reading the inputs R_1 and R_2, reading the join index, and reading/writing temporary results) of the time taken to perform a join for a representative example. (For details, see Example 6.1.) Also included for illustration is the cost of Hybrid-hash join, a join method that does not use the join index.



The key feature of the algorithms that enables such efficient performance is the use of a vertically partitioned data structure for the join result. (We *do not* assume that the *inputs* are vertically partitioned.) Attributes from the first input relation are stored in a separate file from those of the second input relation, using transposed files [Batory, 1979]. Attributes that are common are placed arbitrarily in one of the two vertical fragments. There is a one-to-one correspondence between records in each vertical partition: The nth record in the first vertical fragment matches the nth record in the second. We will argue that such a representation has a negligible performance impact on processes that read the join result. Jive-join and Slam-join write their output in two separate passes: one pass for the vertical fragment corresponding to the attributes from each relation. The whole tuples do not have to be composed in memory at the same time, allowing us to better utilize main-memory.

Our main contributions include:

- The proposal of two novel algorithms for joining relations using a join index.
- The analytic performance analysis of the algorithms using a detailed cost model, demonstrating their efficient single-pass performance under lenient memory requirements.
- Analytic comparisons of our algorithms with Valduriez's algorithm, TID-join and with Hybrid hash-join, demonstrating significant performance improvements when the input relations are larger than main memory.
- An efficient implementation of the algorithms, validating our cost model and demonstrating the relative performance of various algorithms in practice.

For the sake of brevity, we focus on the presentation and analysis of Jive-join, and then outline the Slam-join algorithm. This should not be interpreted as implying that Jive-join is the preferred algorithm: as we shall see, each algorithm is preferable under different circumstances.

The structure of the paper is as follows. In Section 2 we discuss our assumptions and present the vertically partitioned data structure for the join result. In Section 3 we present our Jive-join algorithm, which is then analyzed in Section 4. In Section 5, Slam-join is presented and compared with Jive-join. Section 6 gives a comparison of our algorithms with Valduriez's algorithm, TID-join and Hybrid hash-join on a number of examples. In Section 7, actual implementation results are presented to validate the cost model and to demonstrate the relative performance of each algorithm. Section 8 discusses various extensions of our algorithms. In Section 9 we survey related work, and we conclude in Section 10.

2 Terminology and Assumptions

The input relations are denoted by R_1 and R_2. We assume for simplicity that a tuple-id for R_1 or R_2 is simply the "position" of the tuple within the relation (eg., 1, 2, ...). However, our algorithms apply for any sequential physical addressing scheme for which (a) input relation records in one block have smaller tuple-id values than the records in the next block, and (b) the block address can be directly computed from the tuple-id. A typical tuple-id representation would consist of a block number and an offset within the block. Tuple-ids are used in the join index and in the intermediate results of our join algorithms.

A join index is the set of pairs (t_1, t_2) of tuple-ids such that the tuple t_1 from R_1 matches the tuple t_2 from R_2 according to the join condition. We shall treat the join index as a relation, and denote it by J. Following

[Valduriez, 1987] we assume that the join index is physically ordered by one of the tuple-id fields of its tuples; without loss of generality we assume J is ordered by the R_1 tuple-id.

For simplicity, we shall assume that all of the attributes of R_1 and R_2 are required in the join result. The extension of our analysis to cases where fewer attributes are required is straightforward.

We do not assume that any indexes are available on the input relations. We also do not assume that either input relation is physically ordered by any attribute.

Following [Haas et al., 1993, Haas et al., 1997], we assume that join indexes and temporary files are stored on separate disk devices from each other and from the input relations, so that we do not encounter unnecessary disk seeks between accesses. The input relations may reside on the same disk. (This kind of configuration is recommended by most commercial vendors.)

2.1 A Vertically Partitioned Data Structure for the Join Result

We use a vertically partitioned data structure known as a transposed file [Batory, 1979] to store the join result. Attributes from R_1 that are present in the join result are stored in a separate file (denoted JR_1) from those of R_2 (which are in JR_2). Join attributes that are common to both relations are placed arbitrarily in one of the two vertical fragments. The first entry in each of the files corresponds to the first join result tuple, the second entry to the second join result tuple, and so on. There is no need for any additional stored tuple-id or surrogate key. Each vertical fragment is in the same sequence. This layout is summarized in Figure 1. The

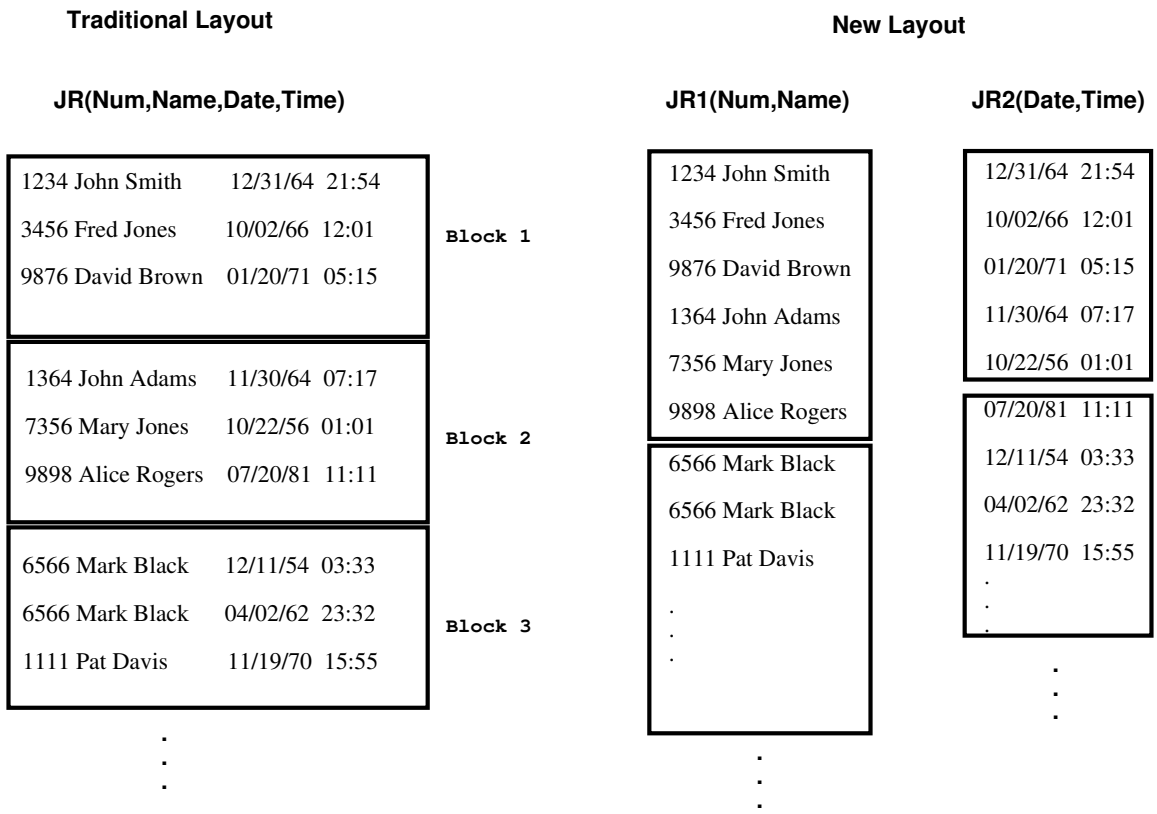


Figure 1: Physical Data Layout for the Join Result.

join result JR is shown on the left in a traditional layout, and on the right in a partitioned layout as JR_1 and JR_2. The input relation R_1 has attributes *Num* and *Name*, and R_2 has attributes *Num*, *Date* and *Time*. The total amount of space occupied by the relation is the same.[2]

We consider an output result that is vertically partitioned to be an acceptable physical storage design. In a query-intensive environment with relatively few updates, such as in a decision-support system, there

[2]Actually, there may be a small change in space utilization due to internal fragmentation.

are substantial benefits to a vertically partitioned physical structure. Commercial systems (such as Sybase IQ) allow vertical partitioning and others (such as Red Brick) are incorporating such features. Thus, a basic premise of our work is that it is applied in a context such as decision-support in which a vertically partitioned physical design is acceptable. It is unlikely that a join result will be updated directly anyway, so one could argue further that vertically partitioned join results might be acceptable even when vertically partitioned base relations are not. In this paper we assume that only the join result and intermediate results (which do not receive on-line updates) are vertically partitioned; the input relations are assumed to be stored in a standard fashion.

The advantage of the partitioned representation is that we will be able to write each vertical partition in a separate phase of the algorithm, getting better utilization of main memory. We claim that for the join result, there are few drawbacks to the partitioned approach. The most likely destiny of the join result is that it will be consumed sequentially by a subsequent process. For sequential I/O one could read a moderate number of blocks from JR_1 followed by the corresponding blocks from JR_2. This approach incurs insignificant additional seek time or rotational delays compared with reading a traditionally represented join-result. (See Appendix B.2 for an experimental verification of this claim.)

2.2 Pipelined vs. Non-pipelined Query Processing Model

We focus on a non-pipelined model (e.g., materializing the full output result) when comparing our algorithms with others. Pipelining has been used effectively to reduce temporary I/Os for both intra-operator and inter-operator query processing. However, in our context large-scale inter-operator pipelining is not always possible without introducing additional I/O because the intermediate results and base relations are often much bigger than available memory. Further, pipelining can sometimes lead to significantly *worse* performance, as we shall exemplify when analyzing Valduriez's algorithm which relies on intra-operator pipelining to avoid temporary results. The main novelty of our algorithms is that they perform much better when materializing the full output result (in a vertically partitioned way, as discussed above) compared with other algorithms.

Pipelined versions of our algorithms are briefly described in Section 8.

Materializing views involving joins in order to speed up queries is an important problem with real-world applications [Gupta and Mumick, 1997, MacNicol, 1997], and the present work is particularly relevant to materializing join results.

3 Jive-join: Range-Partition then Process

In this section we present a new algorithm called Jive-Join[3] for performing joins using a join index. The algorithm matches records from R_1 with records from the join index, partitioning each R_1 record and matching R_2 tuple-id according to the R_2 tuple-id. There are several partitions, each corresponding to a range of R_2 tuple-id values. The R_1 record and its R_2 tuple-id go to separate output files. The R_1 record goes to an output file that contains a collection of R_1 records; the R_2 tuple-id goes to a temporary file that contains a collection of R_2 tuple-ids. The temporary files are then processed, together with the R_2 records, to generate the R_2 component of the join result.

The algorithm consists of three steps:

Step J1

We choose $y-1$ R_2 tuple-ids as partitioning elements. The partitioning elements determine y tuple-id ranges called *partitions*. See Section 4.3 for a description of how the partitioning values are chosen: the aim is to evenly partition the R_2 tuples appearing in the join. Each partition has (in memory) an associated *output file buffer* of length x blocks, and an associated *temporary file buffer* of length v blocks. (When full, each of these buffers will flush its contents to a corresponding *output file* and *temporary file* on disk, and then accept new records.)

Step J2

We scan J and R_1 sequentially, in a fashion similar to a merge-join on the R_1 tuple-id. (Remember that J is in R_1 tuple-id order.) We examine a block of R_1 only if it has a tuple mentioned in the join index. On each

[3] "Jive" is an acronym for "Join Index with Voluminous relation Extensions."

match, we first identify the partition to which this tuple belongs, based on the R_2 tuple-id. We perform two operations:

(a) The attributes of R_1 that are required for the join result are written to the output file buffer for the partition.

(b) The R_2 tuple-id is written to the temporary file buffer for the partition.

When J is exhausted, all file buffers are flushed to disk, and the memory for the file buffers is deallocated.

After finishing Step J2, we have generated half of the output, namely JR_1. The partitions of JR_1 are linked together into a single file. We have also generated a temporary file that is used in Step J3 below to generate the other half of the output.

Step J3

For each partition of the temporary file (in order) we perform the following operations. We read into memory the whole partition, and sort the R_2 tuple-id column in memory into ascending order with duplicates eliminated. (We also keep the original version of the temporary file.) We then retrieve tuples from R_2 in order, retrieving only blocks that contain a matching record according to our sorted version of the temporary file.

We keep sequentially reading records from R_2 until some record from R_2 has a higher tuple-id than the largest tuple-id in the partition. At that point we write the partition's portion of JR_2 as follows. We look at the original version of the temporary file for the partition, and write the corresponding R_2 tuples in that order to a partition of JR_2, the join-result output. (We could use binary search to locate the tuples in order, or alternatively store the R_2 tuples in a hash table by hashing on the tuple-id.)

We then continue with the next partition, and so on. By the time we have finished with the final partition, we have generated all of JR_2. The partitions of JR_2 can be linked together into a single file. With JR_1 generated in Step J2, we have the required join. Note that in Steps J2 and J3 we make sure not to read a block from either R_1 or R_2 if it is known not to contain a tuple participating in the join. That way, we will get better performance if only a small proportion of each input relation participates in the join.

Example 3.1: Consider the two relations *Student* and *Course*, and their join result, given below. This particular join is a natural join in which we match the course numbers in the two input tables. The join index appears on the right. To enable the reader to keep track of various duplicate student tuples as they are processed through the algorithm, we have provided superscripts to help distinguish them.

Student	Course
Smith[1]	101
Smith[2]	109
Jones	104
Davis[1]	102
Davis[2]	105
Davis[3]	106
Brown	102
Black	103
Frick	107

Relation *Student*

Course	Instructor
101	Green
102	Yellow
103	Green
104	White
105	Evans
106	Alberts
106	Beige
108	Red
109	Grey

Relation *Course*

Student	Course	Instructor
Smith[1]	101	Green
Smith[2]	109	Grey
Jones	104	White
Davis[1]	102	Yellow
Davis[2]	105	Evans
Davis[3]	106	Alberts
Davis[3]	106	Beige
Brown	102	Yellow
Black	103	Green

Join Result

Student tuple-id	Course tuple-id
1	1
2	9
3	4
4	2
5	5
6	6
6	7
7	2
8	3

Join Index

For the purposes of exposition, we assume that each input record occupies one disk block. We also assume that we have three partitions ($y = 3$) and that each buffer can hold just one input record at a time. We choose partitioning values 3 and 6 for the tuple-ids of the matching Course tuples. The three partitions will be (id< 3), (3 ≤id< 6), and (6 ≤id). After the first two steps, we will have partitioned the join index and R_1 tuples as follows:

(a) id< 3	
Smith[1]	1
Davis[1]	2
Brown	2

JR_1(a) Temp(a)

(b) 3 ≤id< 6	
Jones	4
Davis[2]	5
Black	3

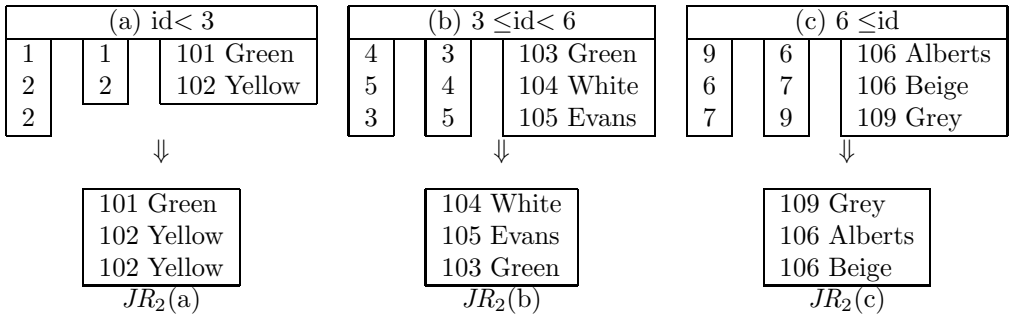
JR_1(b) Temp(b)

(c) 6 ≤id	
Smith[2]	9
Davis[3]	6
Davis[3]	7

JR_1(c) Temp(c)

The left column of each partition is an output file, corresponding to a portion of JR_1. The right column is the temporary file. The output file and the temporary file grow as a result of buffer flushes.

In Step J3 of the algorithm, for each partition, we read in and sort the temporary file sequences (keeping the original temporary file), read in the corresponding records from the Course relation, and write the component of the join result from the Course relation (JR_2). This process is summarized in the diagram below. Each partition is handled separately so that the process fits in memory.



The middle column in this diagram contains the tuple-ids in sorted order (without duplicates), and the right column contains the matching records from the Course relation that are read in. The Course records are written to the output in the order of the original temporary file sequence above. Concatenating the various partitions of the generated join result, our output looks like this:

	Student	Course	Instructor	
	Smith[1]	101	Green	
JR_1(a)	Davis [1]	102	Yellow	JR_2(a)
	Brown	102	Yellow	
	Jones	104	White	
JR_1(b)	Davis[2]	105	Evans	JR_2(b)
	Black	103	Green	
	Smith[2]	109	Grey	
JR_1(c)	Davis[3]	106	Alberts	JR_2(c)
	Davis[3]	106	Beige	

The output is in two vertical fragments JR_1 and JR_2. The horizontal lines denote the boundaries between the three range-partitions. The result is the same as that given initially, except for the order of the tuples. □

There are several important points to note about Example 3.1:

- The algorithm reads only participating records. Records for student Frick and instructor Red are not read from disk, saving I/O.
- Participating records are read just once, even when they participate multiple times.
- By using buffering in the first step, the algorithm can operate while keeping only three full records (from either relation) in memory at any one time, together with some tuple-ids. Both input relations are larger than three records.
- Only one pass over each input relation is made. In contrast, with main memory capable of holding just three records (plus some tuple-ids), Valduriez's algorithm would make three passes through the Course relation.

4 Analytic Performance Analysis

4.1 A Detailed Join Cost-Model

A table of symbols is given in Table 1. A table of system constants, with their value (used in the analytic comparisons) is given in Table 2. The constants used follow [Haas et al., 1993, Haas et al., 1997, Brown et al., 1992], and correspond to the Fujitsu M2266 disk drive.

Symbol	Meaning
$\|R\|$	Number of blocks in R.
$\|\|R\|\|$	Number of tuples in R.
$\ll R \gg$	Width in bytes of a tuple in R.
w	Width of the join attribute in bytes.
t	Size of a tuple-id, in-memory pointer, or integer value in bytes.
p	Size of a disk pointer in bytes.
τ_i	Semijoin selectivity, i.e., the proportion of the tuples in R_i that participate in the join.
$Y(k,d,n)$	Function to estimate the number of block accesses needed to retrieve k tuples out of n tuples stored in d blocks [Yao, 1977].
m	Size of main memory, in disk blocks.
β	Number of blocks in an input relation buffer.
N_S	Number of seeks in an algorithm.
$N_{I/O}$	Number of I/O requests in an algorithm.
N_X	Number of block transfers in an algorithm.
x	Number of blocks in the partition output buffer.
v	Number of blocks in the temporary TID file buffer.
y	Number of join result partitions.

Table 1: Table of symbols

Symbol	Value	Meaning
b	8192	Number of bytes in a disk block.
c	83	Number of disk blocks in a cylinder.
D	130000	Size of a disk device, in blocks.
T_S	9.5	Time for an average disk seek (milliseconds).
T_L	8.3	Average rotational latency (milliseconds).
T_X	2.6	Block transfer time (milliseconds).

Table 2: Table of system constants

Haas, Carey and Livny have proposed a detailed I/O cost model in which seek time and rotational latency are explicit [Haas et al., 1993, Haas et al., 1997] 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 their more detailed cost model. In this paper, we shall use the detailed cost model from [Haas et al., 1993, Haas et al., 1997] to measure the cost of various join algorithms. 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 paper we choose to ignore CPU cost, and focus on the I/O cost. The main reason for this choice is that CPU cost is significantly smaller than the I/O cost when the input relations are much bigger than main memory. Almost all of the CPU-intensive work can be done while waiting for disk I/O. We do measure the CPU cost for our experiments in Section 7.2 and verify that the CPU cost is much smaller than the I/O cost.

We do not count the cost of putting together the two vertical partitions of the join result in our analytical studies. It may be more work to read two vertical partitions and to combine[4] them than to read a nonpartitioned relation from the disk. The question to ask is "*How much* more expensive is it?" We answer this question *quantitatively* in Appendix B.2. As long as one has a large enough buffer to read the partitions, the additional work (both I/O and CPU) is negligible. Our conclusion is that a vertically partitioned output relation does not adversely impact the performance of subsequent operations, and therefore it is fair to exclude this cost

[4]We emphasize that combining the partitions is a simple operation: the nth record of one partition matches the nth record of the other; no further joining is necessary.

from our comparison to simplify our analytical cost model. We did include this cost (although negligible) in our experimental performance studies as a sanity check.

We perform input buffering on the input relations in order to reduce seek and rotational latency. In most cases our results are insensitive to the buffer size for the input relations once the buffer size exceeds a small threshold (about four blocks), and so input relation buffering does not have a major impact on our results. If the input buffer size is β blocks, then that is the minimal unit of information transfer from the input relations: we must read a β-block chunk if any of the constituent blocks contains a participating tuple.[5] In our analytic graphs we use a value of β equal to the cylinder size c, i.e., 83 blocks.

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 relation. We can approximate the total seek time for one pass through relation R as $|R|/D$ times the maximum seek cost, since the maximum seek cost reflects a complete traversal across the disk.[6] The average seek cost can be estimated as one third of the maximum seek cost [Silberschatz et al., 1997] (page 298). Thus the total seek time for one pass through relation R can be estimated as $3|R|T_S/D$.

We assume that in-memory sorting is done in-place using an algorithm such as quicksort [Hoare, 1962].

In this paper we do not address the cost of maintaining the join index. Blakeley and Martin have comprehensively analyzed the tradeoff between join index maintenance cost and the join speedup [Blakeley and Martin, 1990].

4.2 Memory Requirements for Jive-Join

We need Step J1 and Step J2 to fit in main memory. Ignoring insignificant terms, the following inequality must hold:

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

Step J3 must also fit in main memory. The total size of the partitions of the temporary file is z blocks, where $z = ||J||t/b = |J|/2$. The total size of the corresponding R_2 tuples is $\tau_2|R_2|$ blocks, assuming that all of the attributes of R_2 are required in the join. The sorted version of the temporary file can be discarded incrementally (and the memory reused) as the R_2 records are read. Thus, we get

$$|J|/2 + \tau_2|R_2| \leq ym. \tag{2}$$

A subtle point in Equation 2 is the potential presence of skew. This issue will be discussed further in Section 4.3. Combining Equations 1 and 2 with the constraint that $x + v \geq 2$ yields

$$m \geq \sqrt{|J| + 2\tau_2|R_2|}. \tag{3}$$

Equation 3 specifies the minimum amount of memory necessary for Jive-join to perform with a single pass over the input relations. 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 smaller relation (R_2), and the number of blocks in the join index. Note that R_1, the larger relation, may be arbitrarily large.

To get an idea of how lenient this constraint 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 join between R_1 and R_2 with full participation by both relations. Assuming that tuples in R_2 are much wider than a tuple-id, we would be able to apply Jive-join with a single pass through each input for R_2 of size up to 1 terabyte.

In Section 4.4 we will show how to choose optimal values of x, y, and v, and will justify the claim that R_2 should be the smaller relation.

4.3 Choosing the Partitioning Values

We now show how to choose the partitioning elements in Step J1 of Jive-join. A first attempt might be to partition the tuple-ids evenly. Since the number of tuples in R_2 is known, we could simply divide the tuple-id range into y equal-sized partitions.

[5] For sparse joins, *smaller* input buffers may be better than larger ones.

[6] This model assumes no competing access to the disk during the traversal. 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.

This approach would work if the distribution of tuples in the join was uniform. However, for distributions with significant skew, we may find that some partitions contain many more participating tuples than others. For all partitions to fit in main memory, we would have to ensure that the largest partition, together with its portion of the temporary file, fits in main memory in Step J3. We thus waste some memory for all other partitions.

Fortunately, we can do better. In fact, we can *perfectly* partition the tasks of Step J3 to just fit into memory if we are prepared to perform a preprocessing step on the join index. The join index provides us with all the information we need about skew. A preprocessing step can examine the join index and calculate the partitioning elements that divide the tasks of Step J3 into equal-sized chunks. Each chunk may have a different proportion of temporary file blocks to R_2 blocks, depending on the degree to which tuples in the given chunk appear repeatedly in the join result, but the total number of blocks in each chunk is the same.

Another alternative is for the system to *maintain* a set of partitioning values at the same time that it maintains the join index. In a low-update environment such as a decision support system, such an approach would be feasible.

4.4 Measuring the Cost

We now calculate the values of N_S, $N_{I/O}$, and N_X in order to measure the cost of Jive-join. We assume that the $y-1$ 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 J1.

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 $n_i = ||J||* \ll R_i \gg /b$. The number of seeks in Step J2 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 J3 is $3|R_2|/D$ for R_2 (since R_2 is read sequentially), plus one seek each time one starts a new partition in the temporary file. The number of partition starts is y, but since the partitions are ordered, the total time spent seeking corresponds to a unidirectional traversal across the relation (of size z) on disk. Thus, we obtain the formula

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

The number of I/O requests in Step J2 is $|J|/\beta$ for J and $Y(\tau_1||R_1||, |R_1|/\beta, ||R_1||)$ for R_1, plus z/v I/O requests to write the temporary file. We also have n_1/x requests to write JR_1. The number of I/O requests in Step J3 is $Y(\tau_2||R_2||, |R_2|/\beta, ||R_2||)$ for R_2, plus one new request each time one starts a partition in the temporary file, plus one request each time one starts a partition in JR_2. The number of partition switches is y. Thus, we obtain the formula

$$N_{I/O} = 2y + |J|/\beta + Y(\tau_1||R_1||, |R_1|/\beta, ||R_1||) + Y(\tau_2||R_2||, |R_2|/\beta, ||R_2||) + z/v + n_1/x.$$

Unless the size of the join index is very small, $N_{I/O}$ is $2y + |J|/\beta + |R_1|/\beta + |R_2|/\beta + z/v + n_1/x$.

The number of block transfers in Step J2 is $|J|$ for J and $\beta Y(\tau_1||R_1||, |R_1|/\beta, ||R_1||)$ for R_1. Additionally, we need z block transfers to write the temporary file. The number of block transfers in Step J3 is $\beta Y(\tau_2||R_2||, |R_2|/\beta, ||R_2||)$ 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

$$N_X = |J| + 2z + \beta Y(\tau_1||R_1||, |R_1|/\beta, ||R_1||) + \beta Y(\tau_2||R_2||, |R_2|/\beta, ||R_2||).$$

In the event that all input relation blocks participate in the join, we can simplify the equation above to $N_X = 2|J| + |R_1| + |R_2|$.

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

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

Some elementary calculus, together with the constraints of Section 4.2, show that this expression is minimized with respect to x, y and v when

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

where $L = |J|/2 + \tau_2|R_2|$. Finally, since our algorithm will work best when x and v are large, it is clear from the formulas above that R_2 should be the relation with fewer participating blocks (since $L = |J|/2 + \tau_2|R_2|$). Usually this will happen when R_2 is the smaller relation.

5 Slam-join: Build Runs then Merge

We now present the "Slam-join"[7] algorithm. Initially, both R_1 and the join index are read until all of memory is used. The join index is then sorted into ascending R_2 tuple-id order. The R_1 tuples are written to an output file (in the R_2 tuple-id order) and the sorted R_2 tuple-ids are written to a temporary file. Once all of R_1 and the join index has been processed in this way, we compose the R_2 fragment of the output result. All of the temporary files are incrementally merged: for each R_2 tuple-id from a temporary file, the matching R_2 record is read and then written to an output file corresponding to that temporary file.

Step S1

As much of J and R_1 as will fit in memory is read (sequentially) from secondary storage. We read a block from R_1 only if it contains a tuple whose tuple-id is in J. When a tuple from R_1 is read into memory, we replace the corresponding R_1 tuple-id in our in-memory copy of the join index with the memory address of the tuple. Tuples from R_1 are read at most once. We continue until the parts of R_1 and J read use all of main memory. Let us call the in-memory fragment of the join index $\overline{J}$. (Up to this point, Slam-join and Valduriez's algorithm coincide.)

$\overline{J}$ is sorted by the tuple-id value from R_2. Then records in $\overline{J}$ are processed one by one; for each record, the R_2 tuple-id is written (sequentially and in sorted-order) to a temporary file. We make a second pass through $\overline{J}$, this time writing out the corresponding R_1 records (again in R_2 tuple-id order) to an output file containing a partition of JR_1. Note that replacement selection can be easily applied to this stage to reduce the number of runs generated.

If more of J and R_1 remain, then we repeat the procedures above until J and R_1 have been exhausted. Let the number of passes required be y. At this point we have generated y horizontal partitions of JR_1 (one per pass), and y sorted runs of R_2 tuple-ids in temporary files.

Step S2

For each of the y temporary files we allocate in memory a "temporary file buffer" of length v blocks for reading the R_2 tuple-id values, and an "output file buffer" of length x blocks for writing the corresponding partition of JR_2. All of the temporary file buffers are initially filled from the sorted runs in the corresponding temporary files, and refilled whenever they are emptied.

Additionally, we maintain a priority queue consisting of the initial R_2 tuple-ids from each temporary file together with an indicator of the partition that they came from. The priority ordering is that smaller R_2 tuple-ids have higher priority than larger tuple-ids. When a tuple-id is removed from the priority queue, it is replaced by the same partition's next-smallest tuple-id, which is itself then removed from the corresponding temporary file buffer. The size of the priority queue is decremented by one once we reach the end of a temporary file.

Step S3

We proceed as follows until the priority queue of R_2 tuple-ids is empty. We get the next R_2 tuple-id and partition indicator from the priority queue, and read the corresponding record from R_2 (unless we had read that same record on the previous iteration). That record is placed in the output file buffer for the specified partition. The output file buffer is flushed to disk when full.

At this point we are done processing the R_2 relation and the y temporary files. We flush all the output buffers corresponding to horizontal partitions of JR_2 and close all the relations.

[7] "Slam" is an acronym for "SkewLess and Adaptive to Memory".

5.1 Analysis of Slam Join

The reader has probably noticed that we have re-used variables x, v and y in our description of Slam-join. This re-use is not accidental. There is a strong dual relationship between the two algorithms. Step J1 corresponds to Step S2, Step J2 corresponds to Step S3, and Step J3 corresponds to Step S1.

The duality extends to the performance analysis of Slam-join. The derived formulas for Slam-join are *identical* to those for Jive-join, using the common variables x, y and v, provided that the join inputs are reversed. In other words, Jive-joining R_2 with R_1 has the same performance characteristics as Slam-joining R_1 with R_2. As a result, Slam-join performs best when R_1 is the smaller relation.

5.2 Selection Conditions

Selection Conditions on Indexed or Join Attributes

Both Jive-join and Slam-join apply when one uses a *subset* of the join index rather than the full join index. Thus one could apply selection conditions to the input relations' indexes (which are much smaller than the input relations themselves), combine the lists of resulting tuple-ids (using union for an "or" condition and intersection for an "and" condition [Mohan et al., 1990]), and use the result to semijoin the join index [Valduriez, 1987]. Jive-join or Slam-join can then be applied to the input relations and the reduced join index. Similarly, if the join index also included the join attribute value (for easy maintenance) then selections on the join attribute could be made on the join index itself.

Selection Conditions on Nonindexed Attributes

Consider a query that consists of a selection on a nonindexed attribute and a join. In this case every participating record in the relation being selected must be read in order to apply the selection condition. Slam-join and Jive-join perform in a complementary fashion. Both algorithms are easily adapted to handle selections on the first input relation: If an R_1 record is read that does not satisfy the selection conditions then it is simply discarded (along with the corresponding join index records).

If there are selection conditions on nonindexed attributes in *both* relations, then one of the relations R can be scanned before performing the join to find the tuple-ids of records satisfying the selection condition. These tuple-ids can be combined with the join index as in the indexed attribute case. (One can potentially optimize this process by only checking records whose tuple-ids actually appear in the join index.) The overhead in this case (beyond the cost formula for Jive-join or Slam-join) would be $|R| + 2s||R||t/b$ block transfers, where s is the fraction of the records in R satisfying the selection condition.

An improvement to the process mentioned above is to write a new relation consisting of the records found to satisfy the selection condition, together with a temporary file matching tuple-ids from the original relation with tuple-ids in this new relation. Attributes not needed for the query can be projected out. The temporary file and new relation are then read (sequentially) during the later Jive-join or Slam-join step. If the selection condition is very selective, or if we only need a small number of attributes from the original relation, then this can be a substantial improvement. Assuming that all attributes are required in the join result, the overhead now is

$$2s|R| + 2s||R||(2t)/b \text{ block transfers.} \tag{5}$$

When the selectivity is sufficiently small, this represents a small fraction of the overall join cost.

5.3 Comparing Jive-Join with Slam-Join

Despite having identical analytical cost formulas, there are structural properties of each algorithm that make each preferable in certain circumstances.

Skew Handling As outlined in Section 4.3, Jive-join needs to perform a preprocessing step in order to avoid skew. Slam-join, on the other hand, does not require any such preprocessing. Slam-join maximally utilizes memory in the initial phase (Step S1) independent of the skew.

Memory Adaptiveness Jive-join partitions the input in such a way that the resulting partitions just fit in memory. If the amount of available memory changes between the partitioning step (Step J2) and the processing of the partitions (Step J3) then we face one of two problems. Either (a) more memory is available and so memory is underutilized in Step J3, or (b) there is not enough memory available to

fit the partition into memory, and so additional I/O is required. Slam-join, on the other hand, does not need to perform a *static* analysis of the amount of memory required. In Step S1, Slam-join reads in as much information as fits in the *currently* available memory. If the memory decreases, then the corresponding runs generated will be shorter. If the memory increases, then the corresponding runs will be longer. The performance of the final merging step of Slam-join is independent of the length of the runs, and depends only on the number of runs.

Pipelining Neither Jive-join nor Slam-join can provide full pipelining of the join result to a subsequent process because the two halves of the join result tuples are never actually in memory at the same time. Jive-join can achieve partial pipelining, though, by pipelining the JR_2 output of Step J3 directly into the subsequent process without writing it to disk. That subsequent process can read the corresponding JR_1 output from disk and construct the full tuples on the fly. Slam-join cannot achieve such pipelining because Step S3 writes to all partitions of JR_2 in parallel.

Selection Conditions on Nonindexed Attributes In Section 5.2 we described how selection conditions on nonindexed attributes can be applied. As observed above, Slam-join prefers to process the smaller relation first, while Jive-join prefers to process the larger relation first. Thus, if the selection condition is on an attribute of the smaller relation one would prefer Slam-join, and one would choose Jive-join for a selection condition on the larger relation. If there are selection conditions on nonindexed attributes in *both* relations, then one of the relations will need to be scanned before performing the join, as described in Section 5.2. In general, one would prefer to scan the smaller relation. Thus one would apply the selection condition on the larger relation within Jive-join, as described above.

Order of the join index Because of the duality, Slam-join prefers the join index to be ordered by the smaller relation's tuple-ids, while Jive-join prefers an ordering based on the larger relation. If the join index order is fixed, perhaps for maintenance reasons, then the choice of algorithms is clear if one wants to avoid the overhead of first sorting a copy of the join index. Conversely, if one of the algorithms has been identified as preferable for other reasons, then the join index order could be chosen accordingly.

6 An Analytic Comparison of Algorithms

We now compare the analytic performance estimates of Jive-Join and Slam-Join against those of the join-index algorithm of Valduriez, presented in Appendix A.1, the TID-join algorithm presented in Appendix A.2, and the ad-hoc Hybrid hash-join, presented in Appendix A.3. We use the detailed cost model of [Haas et al., 1993, Haas et al., 1997] for the comparison. A comparison with Hybrid hash-join may be considered unfair because Hybrid hash join does not have access to a join index. Nevertheless, we include the comparison to illustrate how much better Jive-join and Slam-join perform *given a join index*, even for joins that are not particularly selective. (The performance of Hybrid-hash join could be improved for examples with partial participation of one or both relations if we allow it to use the join index as a semijoin filter on each relation. One could alternatively extend Hybrid hash join to maintain an in-memory hash filter on the join attribute to reduce the intermediate data for the second relation processed. Such optimizations would help Hybrid hash join for some of the more selective examples presented here, but would not help much for examples with high degrees of participation by both relations, and would not affect the basic conclusions drawn.)

In general we expect our algorithms to perform much better than others. The fundamental reason is our algorithms make a *single pass over the full base data*, with multiple passes over just the tuple-ids, while other algorithms make *multiple passes over full tuples* from at least one of the base relations. The number of tuple-identifiers our algorithms write as temporary files is proportional to the cardinality of the join result; Nevertheless, since a tuple-identifier is substantially smaller than a full record, the savings in base relation processing cost usually far outweighs the overhead of processing temporary files. Big sequential I/Os prevent seek and rotational costs dominating, for both our algorithms and the others mentioned in this paper.

Our comparisons will compare the I/O time taken to do the join (excluding the output block transfer cost) against the memory size for a variety of input relations. The horizontal axis is the number of megabytes of main-memory available, and the vertical axis is the number of seconds taken in I/O time by each algorithm. For reference we shall also include the lower-bound cost, namely the block transfer cost of sequentially reading the join index and all the participating blocks in R_1 and R_2.

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 4.1. 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. We chose very large relations for the analytic comparison in order to demonstrate the utility of the algorithms on extremely large data sets. Bear in mind that the performance figures refer to a disk with a 3.1 MB/sec peak throughput. Faster disks or disk arrays would substantially reduce the time results. Our experimental evaluation in Section 7 shows the performance for relations of more modest size.

For Jive-join and Slam-join, the number of disk blocks transferred is constant, independent of the memory size. Thus we expect the performance curves to be relatively flat, increasing only when memory is scarce, when there are many seeks and small I/O requests generated in processing the buffers. Both algorithms have essentially identical performance on the examples described here; we show just a single curve on each performance graph for clarity.

When comparing algorithms on examples that include selection conditions on nonindexed attributes we use straightforward extensions of Valduriez's algorithm, Hybrid-hash join, and TID-join that check the selection condition on each record as it is read, and discard records that don't satisfy the condition. Jive-join and Slam-join handle selection conditions as described in Section 5.2; the cost overhead calculation uses Equation 5.

There are more interesting scenarios than we can present in this paper. We have selected a handful that are representative. In each of these examples we suppose that the width of a tuple in both R_1 and R_2 is 256 bytes, so that there are 32 records per block in each relation.

Example 6.1: In our first example, let us take $|R_1| = |R_2| = 2^{20} \approx 10^6$ blocks, so that the size of each input relation is 8 gigabytes. $||R_1|| = ||R_2|| = 2^{25} \approx 33 \times 10^6$ tuples. We assume $t = w = 4$ bytes. We consider several scenarios.

(a) The join is a one-to-one join with full participation of both R_1 and R_2. In this case $||J|| = ||R_1|| = ||R_2||$, and $|J| = 2^{15}$ blocks. The performance graph for this scenario appears in Figure 2 (a). Jive/Slam-join perform close to optimal, while Hybrid Hash join and Valduriez's algorithm perform significantly worse. The explanation for these differences in performance is apparent in Figure 2 (a') where the individual components of the cost at memory size 2520 MB (the right edge of the graph) are given. At this memory size, all algorithms benefit from large I/O units, and so seek times and rotational latencies will not play a role in the comparison. The costs of each algorithm are split into four components: Reading R_1, reading R_2, reading the join index, and reading/writing temporary files. Valduriez's algorithm performs badly because it needs to perform multiple passes over R_2, hence the high R_2 component of the cost. (The step-like performance graph for Valduriez's algorithm is due to the change in the integral number of passes through R_2.) Hybrid-hash join has a high component for reading and writing the temporary files, corresponding to reading and writing full tuples in hash partitions. The TID-join algorithm also has a large cost component corresponding to reading and writing temporary files, since it reads and writes full R_1 tuples within its intermediate result. Jive-join and Slam-join have very small temporary files, as indicated in these graphs, due to the fact that the temporary files contain just tuple-identifiers rather than full tuples.

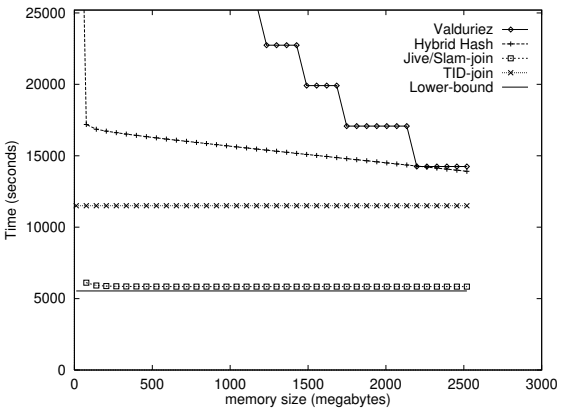
(b) The join is a many-to-many join with full participation of both R_1 and R_2. We choose $||J|| = 16 * ||R_1|| = 2^{29}$, and $|J| = 2^{19}$ blocks. The performance graph for this scenario appears in Figure 2 (b). Valduriez's algorithm performs poorly because it makes multiple passes through R_2; Hybrid-hash join's intermediate hash tables account for most of the gap between Hybrid-hash join and Jive/Slam-join; the temporary file of TID-join is particularly large in this case, and as a result TID-join doesn't make it onto the graph.

(c) The join is a one-to-one join with partial participation of both R_1 and R_2. We choose $||J|| = ||R_1 \ltimes R_2|| = ||R_2 \ltimes R_1|| = 2^{19}$, and $|J| = 2^9$ blocks. The performance graph for this scenario appears in Figure 2 (c). The join index allows for a significant improvement over a traditional ad hoc join. As can be seen, both Jive/Slam-join and TID-join perform comparably, and Valduriez's algorithm performs much better than Hybrid-hash join for reasonable memory sizes. The additional cost of reading and writing JR_1 by TID-join is negligible compared to the I/O cost of reading the input relations because the join cardinality is so small.

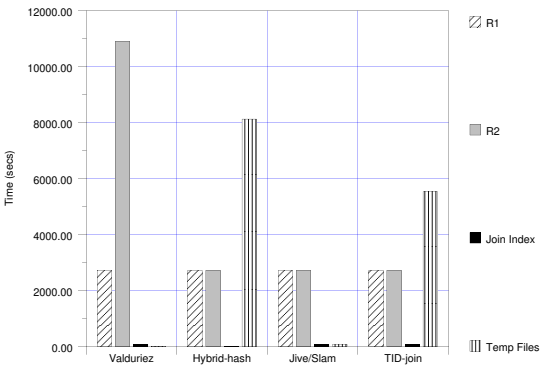
(d) The join is a one-to-one join as in part (a), except that a selection condition is applied to a nonindexed attribute of R_1. The selection condition selects just one record in every ten. Valduriez's algorithm, TID-join and Hybrid-hash join have improved compared with part (a) because of the reduction in intermediate-data volume. The transition just beyond 800 megabytes for TID-join corresponds to the

point where the intermediate result just fits into memory, and thus doesn't need to be written to disk. Jive/Slam-join still wins for memories smaller than 800 megabytes.

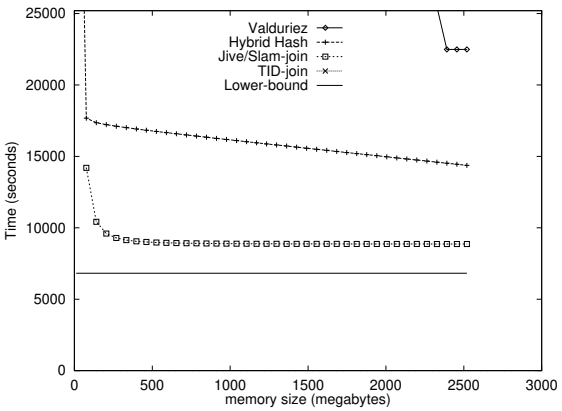
(e) The join is a one-to-one join as in part (a), except that selection conditions are applied to nonindexed attributes of both R_1 and R_2. The selection conditions each selects just one record in every ten. Hybrid hash-join has improved compared with part (d) and is now competitive. The TID-join algorithm performs comparably with Jive/Slam-join for memories below 800 megabytes because the overhead that Jive/Slam-join have of processing a selection condition on the second relation is comparable to the intermediate-result overhead of TID-join. Jive/Slam-join is slightly more expensive than in part (d) due to this overhead. □



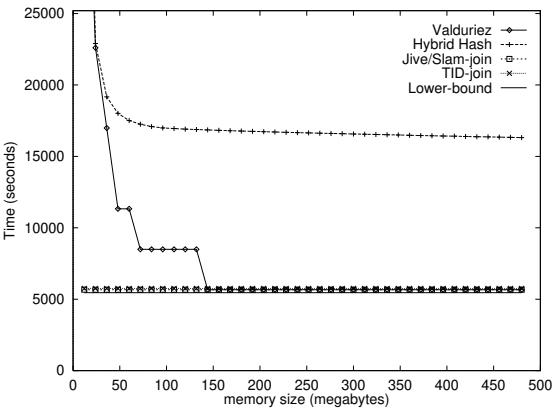
(a)



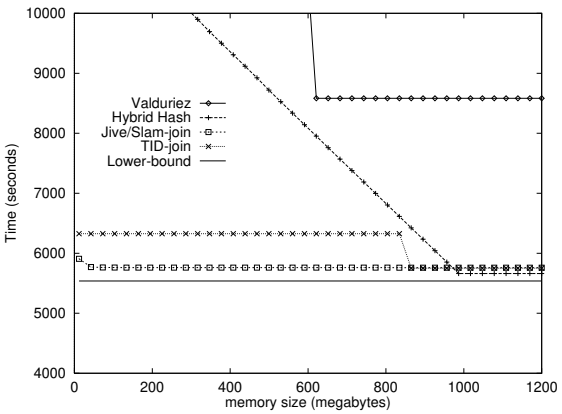
(a')



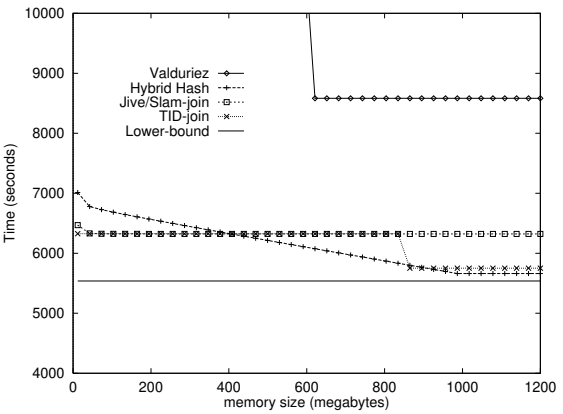
(b)



(c)



(d)



(e)

Figure 2: Performance comparison for Example 6.1.

Example 6.2: In this example, we consider relations 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 $||R_1|| = 2^{32} \approx 4.3 \times 10^9$ tuples and $||R_2|| = 2^{25} \approx 3.3 \times 10^7$ tuples. We assume $t = 4$ bytes and $w = 8$ bytes. In the following analysis we repeated the cost calculations when the roles of R_1 and R_2 were switched, and the graphs plot the minimum cost alternative at each data point. We consider several scenarios.

(a) The join is a one-to-many join with full participation of both R_1 and R_2. In this case $||J|| = ||R_1||$, and $|J| = 2^{22}$ blocks. The performance graph for this scenario appears in Figure 3 (a). Valduriez's algorithm takes too long to appear in the graph. The components of the cost of each algorithm are qualitatively similar to those given in Figure 2(a'); Jive-join and Slam-join win because their intermediate files are small, containing just tuple-ids.

(b) The join is a one-to-one join with full participation of R_2, but partial participation of R_1. $||J|| = ||R_2||$, and $|J| = 2^{15}$ blocks. 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/Slam-join. The TID-join algorithm also benefits from the low selectivity, and performs comparably to Jive/Slam join since the cost of reading R_1 dominates.

(c) The join is a one-to-one join with partial participation of both R_1 and R_2. We choose $||J|| = ||R_1 \ltimes R_2|| = ||R_2 \ltimes R_1|| = 2^{19}$, and $|J| = 2^9$ blocks. The performance graph for this scenario appears in Figure 3 (c). Here Hybrid-hash join does not appear on the graph because it took too long; without the optimizations outlined above it has to scan both relations, and is thus not competitive. TID-join, Valduriez's algorithm and Jive/Slam-join perform comparably, and significantly above the lower bound due to the relatively large contribution of rotational latency: there is on average one matching tuple in R_1 every 3.1 cylinders.

(d) The join is a one-to-many join as in part (a), except that a selection condition is applied to a nonindexed attribute of R_1. The selection condition selects just one record in every ten. We suppose that every record in R_2 still participates in the result. The performance of Hybrid hash-join has improved dramatically compared with part (a): its cost is now dominated by the initial pass through R_1 and hence is only slightly worse than that of Jive/Slam-join. The performance of Valduriez's algorithm has also improved, but it is still not competitive on the range of the graph. The gap between Jive/Slam-join and TID-join is due to the size of TID-join's intermediate file.

(e) The join is a one-to-many join as in part (a), except that selection conditions are applied to nonindexed attributes of both R_1 and R_2. The selection conditions each select just one record in every ten. The performance is very similar to part (d), since the cost of reading R_1 is dominant. □

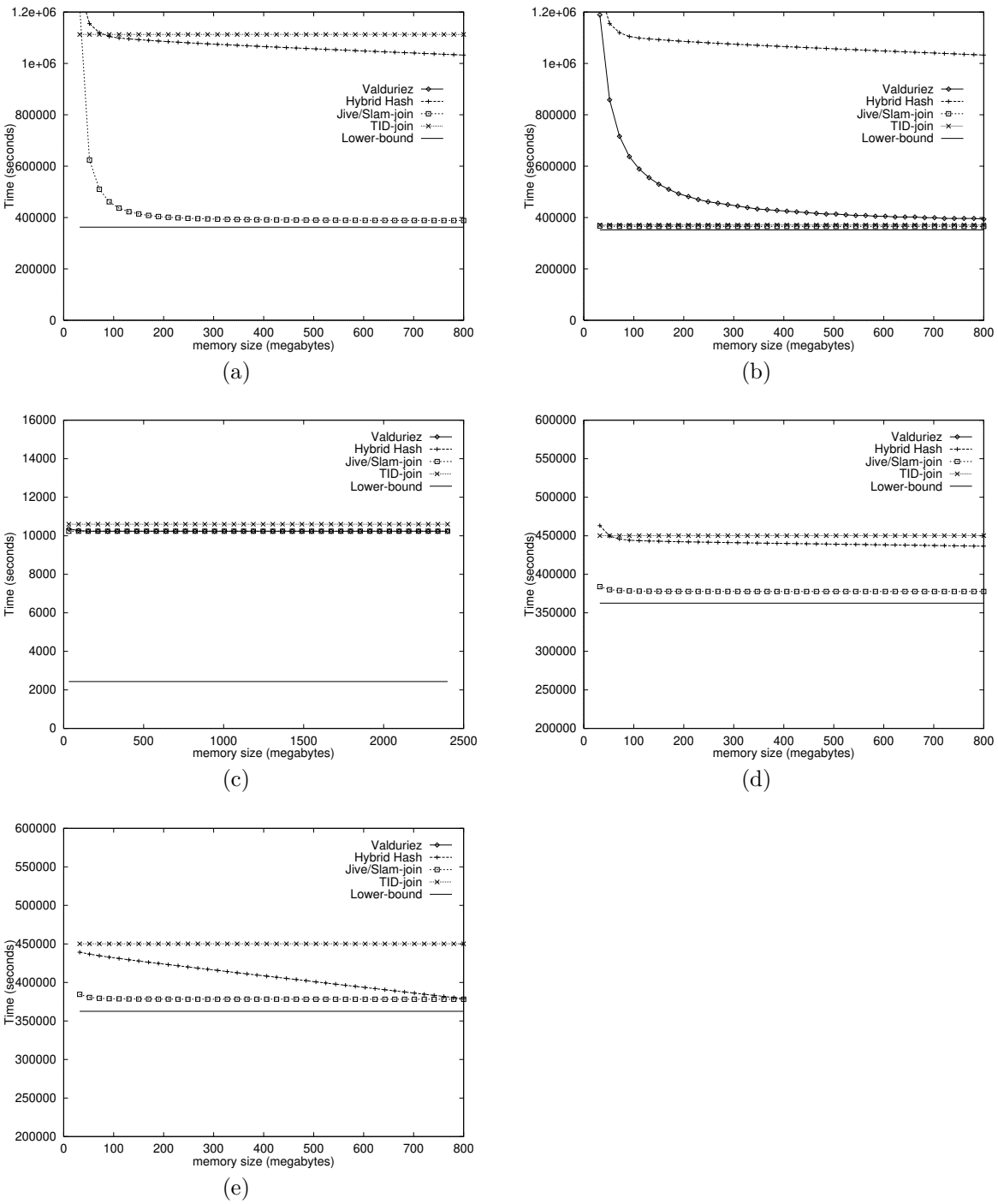


Figure 3: Performance comparison for Example 6.2.

Example 6.3: In this example, we consider relations that could fit into main memory. Let us take $|R_1| = |R_2| = 2^{14} = 16384$ blocks (128 megabytes). Then $||R_1|| = ||R_2|| = 2^{19} \approx 5.2 \times 10^5$ tuples. We assume $t = 4$ bytes and $w = 4$ bytes. We consider two scenarios.

(a) The join is a one-to-one join with full participation of both R_1 and R_2. In this case $||J|| = ||R_1||$, and $|J| = 2^9$ blocks. The performance graph for this scenario appears in Figure 4 (a). Observe that all of the algorithms converge on the optimal time as memory approaches the size of the relations. Jive/Slam-join still outperform the other algorithms for small memories, where the cost components are qualitatively similar to previous examples.

(b) The join is a many-to-many join with full participation of R_1 and R_2, and $||J|| = 2^{26} \approx 6.7 \times 10^7$, and $|J| = 2^{16} = 65536$ blocks. The performance graph for this scenario appears in Figure 4 (b). In this scenario, Hybrid hash join is the clear winner. Note that Hybrid-hash join is not limited by the lower bound, since the lower bound also includes the cost to read the join index. The reason for the observed behavior is apparent from Figure 4(b') where the components of the cost are given for each algorithm when main memory is 100 megabytes. Jive/Slam-join, Valduriez's algorithm and TID-join all read the join index, which for this example is particularly large; since Hybrid-hash join doesn't have to read the join index, this cost is not incurred. The temporary files for Hybrid-hash join are relatively small here because they correspond to hash partitions that are proportional to the size of the input, not to the size of the join result. Even though Jive-join and Slam-join process temporary files containing tuple-ids only, the size of the join result in this case is so big that the temporary files are larger than those of Hybrid-hash-join. The temporary file of TID-join is enormous (over 11,000 on the given scale) because it is not only proportional in size to the join result, but it also contains full tuples from R_1. As before, Valduriez's algorithm performs badly because it makes multiple passes through R_2.

Note that if the join index is large, then the join result must also be large: For this example, the output cost is over 3 hours, and would dominate the total cost. Thus the differences between algorithms may not be significant here if the full join result is materialized.

Seek and rotational latencies play a significant role in the cost of Jive-join for small memories. If the cost was measured simply in terms of blocks transferred, the curve for Jive/Slam-join would be flat. Thus, it is important that we do not simply measure the number of block I/Os; we need to measure seek and rotational latencies as well in order to get an accurate measure of each algorithm's performance. □

Example 6.4: In this example we compare the performance of Valduriez's algorithm, TID-join, and Jive/Slam-join as the join selectivity is varied. In particular, we examine the performance when the number of tuples in the join index is small. We do not consider Hybrid-hash join in this example because its performance is insensitive to the join selectivity.

We take Example 6.1(a), and fix the memory size at 256 megabytes. We then vary the size of the join index by multiplying by a selectivity factor varying from 10^{-7} to 1. The graph of time versus selectivity factor is given in Figure 5(a), with a close-up on the smallest range given in Figure 5(b). Note the logarithmic scale on the horizontal axis.

All algorithms perform equally well for small selectivity values. The cost of the algorithms initially rises steeply as each new join index record results in an extra block of the base relation(s) being read. Eventually, we reach a point at which all base relation blocks are read because some tuple in each block participates in the join. This is the flat area between selectivity 0.001 and 0.01. The cost of Valduriez's algorithm jumps significantly at the point where the part of R_1 that participates in the join cannot fit in main memory, requiring the extra pass through R_2. This is the same point at which the TID-join algorithm begins to write full records to temporary files, and so the TID-join performance also increases. In contrast, the Slam-join and Jive-join performance increases much more slowly; the only extra I/O being done by Slam-join and Jive-join is on tuple-ids, and not full records.

We perform a similar analysis for Example 6.2(a), with the memory size at 256 megabytes. The results appear in Figure 5(c), with a close-up on the smallest range given in Figure 5(d). The results are similar to the previous case. □

Jive-join and Slam-join perform better than Valduriez's algorithm because they limit relation accesses to single sequential scans of the input relations. Valduriez's algorithm processes the input relations sequentially,

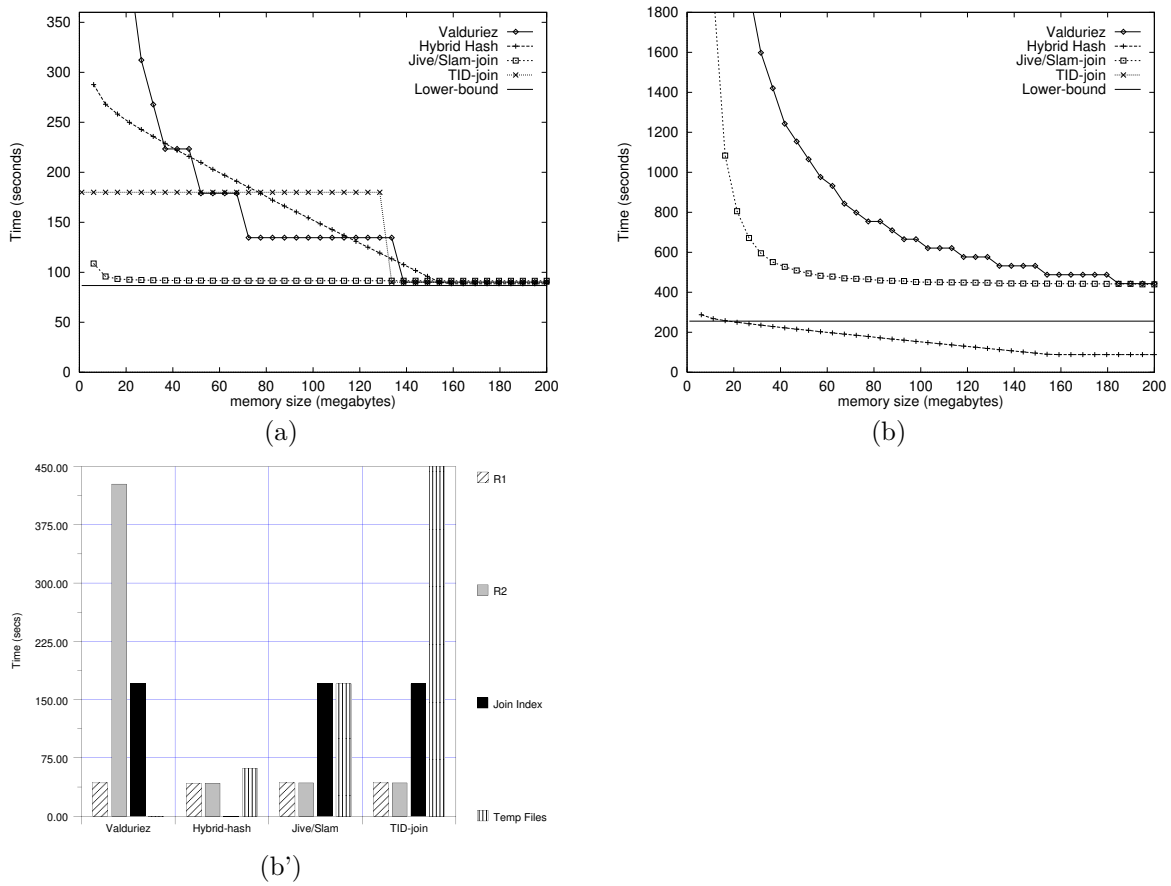


Figure 4: Performance comparison for Example 6.3.

but processes one of the relations multiple times. In Example 6.1 (a), Valduriez's algorithm makes 8 passes through R_2 for main memory of 1000 megabytes, and 4 passes through R_2 at 2000 megabytes.

When the join index and the participating tuples from one of the input relations fit into memory, Valduriez's algorithm performs fastest since it does not write temporary files. The difference in performance is small, as illustrated by the performance graphs. Nevertheless, there is a sense in which Jive-join and Slam-join are preferable even in this case. Consider the graph of Figure 4(a) with the axes switched, given in Figure 6. The graph shows how much main memory is needed in order for each algorithm to perform in the time given on the horizontal axis. When the performance is required to be very close to the lower bound of about 87 seconds, all algorithms need a substantial amount of memory to achieve that performance level. However, if one could tolerate a performance that was slightly above the lower bound, Jive-join and Slam-join use much less memory. For example, with a 15% increase in time to 100 seconds, Jive-join and Slam-join could achieve that performance with less than 9 megabytes of main memory. In contrast, both Valduriez's algorithm and hybrid hash require over 130 megabytes.

The memory advantage of Jive-join and Slam-join illustrated above is important in two ways. Firstly, it can bring the cost of the machine performing the joins down by requiring a smaller amount of expensive main memory. Secondly, the machine performing the join may be performing a number of other tasks needing main memory. In particular, the join computation may be a part of a larger query processing plan whose other parts also consume some main memory. By using less main memory for the join we free more main memory for other query processing steps whose performance may depend heavily on the amount of available main memory.

Our algorithms have assumed the pre-existence of a join index. Nevertheless, there are circumstances under which the cost of *building* a join index is relatively small, for example when there exist indexes for each relation on the join attribute(s). In these cases, Jive-join or Slam-join can outperform ad-hoc join algorithms even if the cost of building the join index is included.

The fundamental conclusions to be drawn from these results are:

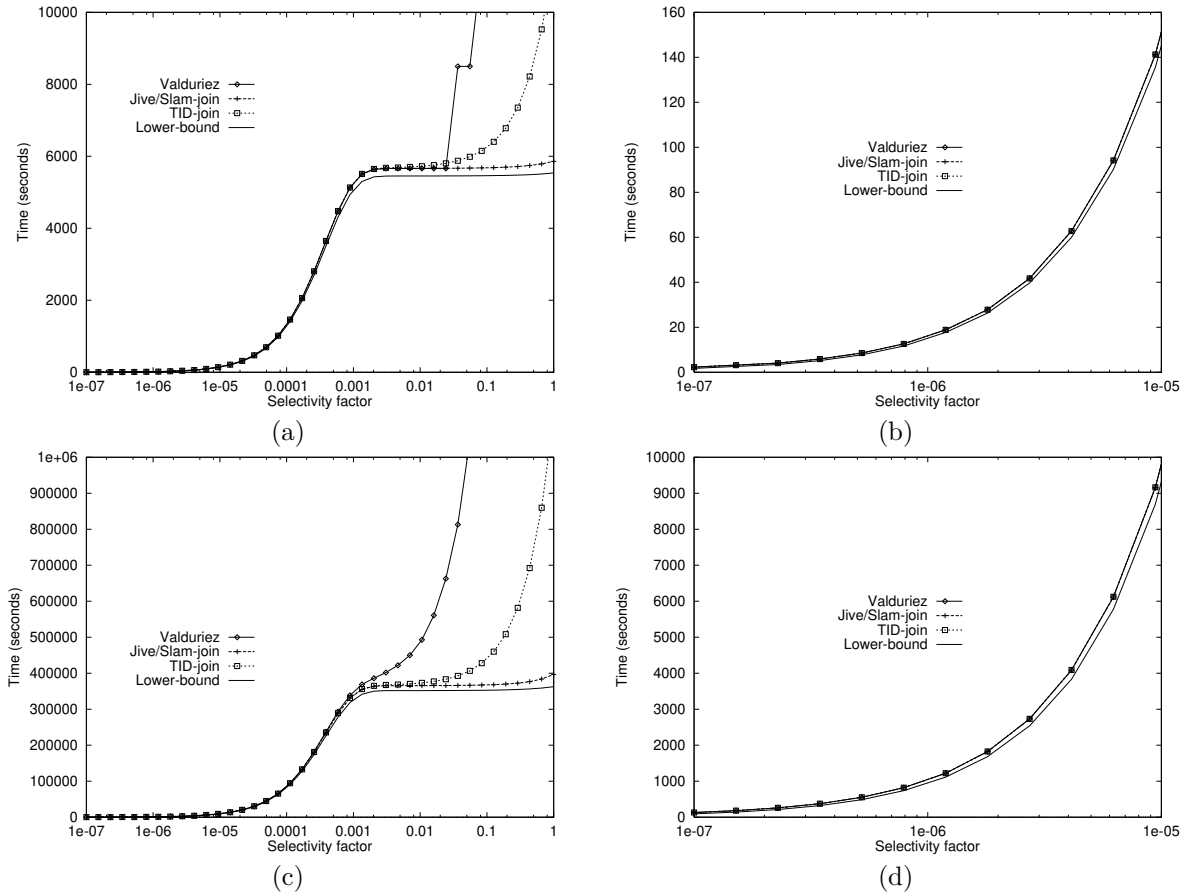


Figure 5: Performance comparison for Example 6.4.

(a) Jive-join and Slam-join perform better than their competitors in a wide range of settings. When the input relations are much larger than main memory, the improvement can be orders of magnitude. This is the crucial range in which to evaluate the algorithms for use with large input relations such as in a decision support system.

(b) Jive-join and Slam-join consume substantially less memory than their competitors for the same level of performance. Main memory is expensive, and we would like to get good performance with as little main memory as possible.

(c) Jive-join and Slam-join may perform worse than Hybrid-hash join in a situation in which the size of the join index is particularly large. However, when the join index is large relative to the input relations, then the join result will be huge and writing the join result will dominate the join cost anyway.

7 Experimental Validation and Performance Results

In this section we present the results of a performance study in which we implemented Jive-join, Slam-join, Valduriez's algorithm and Hybrid-hash join.[8] Our purpose is to both validate the cost model used, and to demonstrate the relative performance of the various algorithms in practice.

[8]If one ignores the seek and rotational latency differences, the performance of TID-join is identical to that of Slam-join for large joins, except for the additional overhead of reading and writing JR_1. Thus we expected that Jive/Slam-join would consistently outperform TID-join, and that it was not worth the effort to separately implement TID-join.

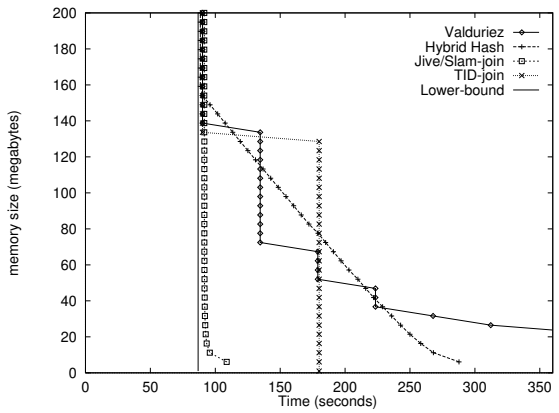


Figure 6: Main memory usage comparison.

7.1 Test Environment

Jive-join, Slam-join, Valduriez's algorithm and Hybrid hash join are hand-coded on top of PDQPS (Parallel and Distributed Query Processing System), a research database system developed at Columbia University. The underlying storage manager is a raw file system similar to WiSS [Chou et al., 1985], which supports extent-based contiguous space allocation and bulk I/O requests. Memory management is done using a homegrown buffer manager, which avoids copying data blocks whenever possible. Therefore our system eliminates most of the performance disadvantages discussed in [Stonebraker, 1981]. Similar to [Haas et al., 1993, Haas et al., 1997], a large chunk of memory is statically allocated for each algorithm execution at startup time. Some of the blocks are then used to input relations, output temporary and join results, store auxiliary and data blocks of a hash table or other structures, based on the memory management scheme specific to each algorithm.

Our Hybrid hash-join implementation is based on the description in [Haas et al., 1993, Haas et al., 1997], which has been demonstrated to yield much higher performance than a vanilla implementation due to the enhanced memory and I/O management. We implemented Valduriez's algorithm according to [Valduriez, 1987].

Two joining relations are produced using the Wisconsin synthetic database generator [Bitton et al., 1983]. Each tuple consists of a four-byte integer as join attribute, plus any remaining padding bytes as a combination of integer and string attributes. The join result tuple is a concatenation of two joining tuples projecting out one of the join attributes. The tuple-id is a logical record number which will be mapped into a physical block number and a record offset pair. The join index is ordered on one of the joining relation tuple-ids. All three input relations are stored contiguously on disks as a sequence of 8KB blocks.

All the experiments are performed on a dedicated 50MHz Sun Sparc 10 workstation running Solaris 2.4 operating system, configured with 32MByte of physical memory. We use four identical Seagate ST11200N disk drives each having 1.05GByte unformatted capacity. Each drive has an embedded fast SCSI-2 controller with a raw data rate of 3.16MByte/s, average seek time 11 ms, average rotational latency 5.56 ms and block transfer time (8KB) 2.5 ms. We allocate one disk to store two joining input relations, one disk for the join index, and one disk to store the intermediate results and final join output. The system has sufficient bus bandwidth to access the three disks concurrently. For simplicity, our system does not overlap CPU and I/O processing; an industrial strength system would do CPU work while waiting for I/O. The I/O time of each algorithm is measured by subtracting the CPU time (sum of user time and system time) from the total elapsed time (wall clock time). Each experiment was run multiple times, with only small variances observed among the results.

7.2 Comparing Jive/Slam Join with Other Algorithms

For all the experiments, we observed that the I/O cost is consistently more than 70% of the total elapsed time. This validates our claim in Section 4.1 that the algorithms will be I/O bound, and justifies our omission of CPU time from the cost model. (The CPU cost ratio of Slam-join, Jive-join and Valduriez's algorithm are slightly higher than that for Hybrid-hash join, but in all cases the CPU effort could easily be overlapped with

I/O.)

Thus, to validate our cost model we compare the measured I/O time versus the predicted analytic I/O time for Slam-join and Jive-join. (Validating cost models for the other algorithms is beyond the scope of this paper.) We also show the measured I/O time for all four algorithms to demonstrate the relative performance in practice.

There are more interesting scenarios than we can present in this abstract. We have selected a handful that represent full and partial participation of one-to-one, one-to-many, and many-to-many joins. In each of these examples we suppose that the R_1 tuple width is 64 bytes, the R_2 tuple width is 104 bytes, the join index tuple width is 8 bytes, and the join result tuple width is 164 bytes. We choose $t = w = 4$ bytes.

Example 7.1: R_1 and R_2 join with full participation by both relations.

(a) The join is one-to-one. We choose $||R_1|| = ||R_2|| = ||J|| = 100,000$, $|R_1| = 782$ blocks (or 6.1MB), $|R_2| = 1283$ blocks (or 10MB), $|J| = 98$ blocks and $|JR| = 2041$ blocks. The performance graph for this scenario appears in Figures 7(a).

(b) The join is a many-to-many join. We choose $||R_1|| = ||R_2|| = 100,000$, $||J|| = 5 * ||R_1|| = 500,000$, and $|J| = 489$ blocks, and $|JR| = 10205$ blocks. The performance graph for this scenario appears in Figure 7(b).

(c) The join is a one-to-many join. We choose $||R_1|| = 20,000$, $|R_1| = 157$ blocks, $||R_2|| = ||J|| = 100,000$, $|R_2| = 1283$ blocks, $|J| = 98$ blocks, $|JR| = 2041$ blocks. The performance graph for this scenario appears in Figure 7(c).

Comparisons of the experimental and analytic graphs for each case are given in Figure 7(d)–(f). □

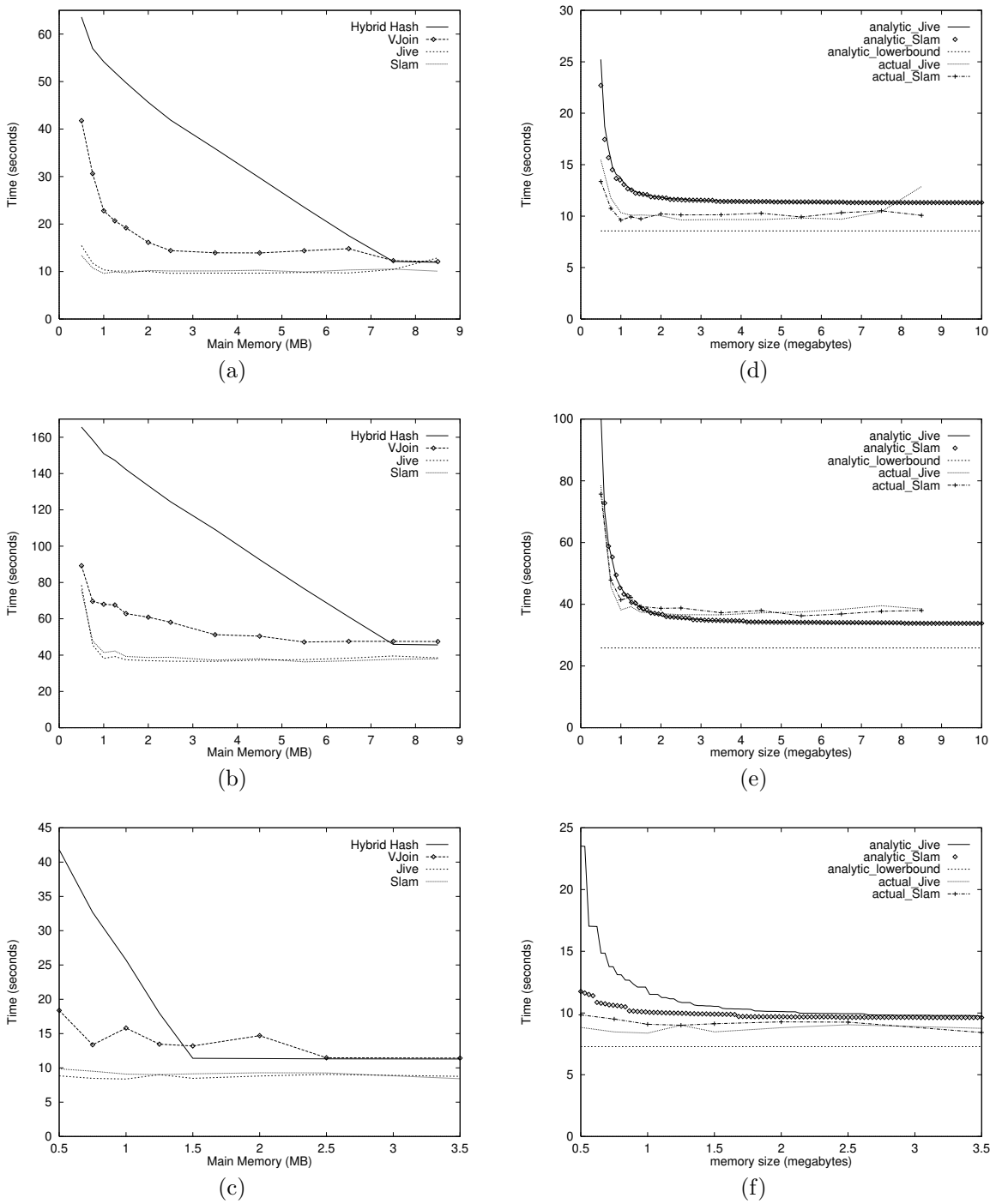


Figure 7: Performance comparison for Example 7.1.

Example 7.2: R_1 and R_2 join with partial participation.

(a) The join is a one-to-one join with partial participation of both R_1 and R_2. We choose $||R_1|| = ||R_2|| = 100,000$, $||J|| = ||R_1 \ltimes R_2|| = ||R_2 \ltimes R_1|| = 20,000$, and $|J| = 20$ blocks, $|JR| = 409$ blocks. The performance graph for this scenario appears in Figure 8(a).

(b) The join is a many-to-many join with partial participation of both R_1 and R_2. We choose $||R_1|| = ||R_2|| = ||J|| = 100,000$, $||R_1 \ltimes R_2|| = ||R_2 \ltimes R_1|| = 20,000$, and $|J| = 98$ blocks, $|JR| = 2041$ blocks. The performance graph for this scenario appears in Figure 8(b).

(c) The join is a one-to-many join with partial participation of both R_1 and R_2. We choose $||R_1|| = 20,000$, $|R_1| = 157$ blocks, $||R_2|| = 100,000$, $|R_2| = 1283$ blocks, $||J|| = 50,000$, $|J| = 49$ blocks, $|JR| = 1021$ blocks. The performance graph for this scenario appears in Figure 8(c).

Comparisons of the experimental and analytic graphs for each case are given in Figures 8(d)–(f). □

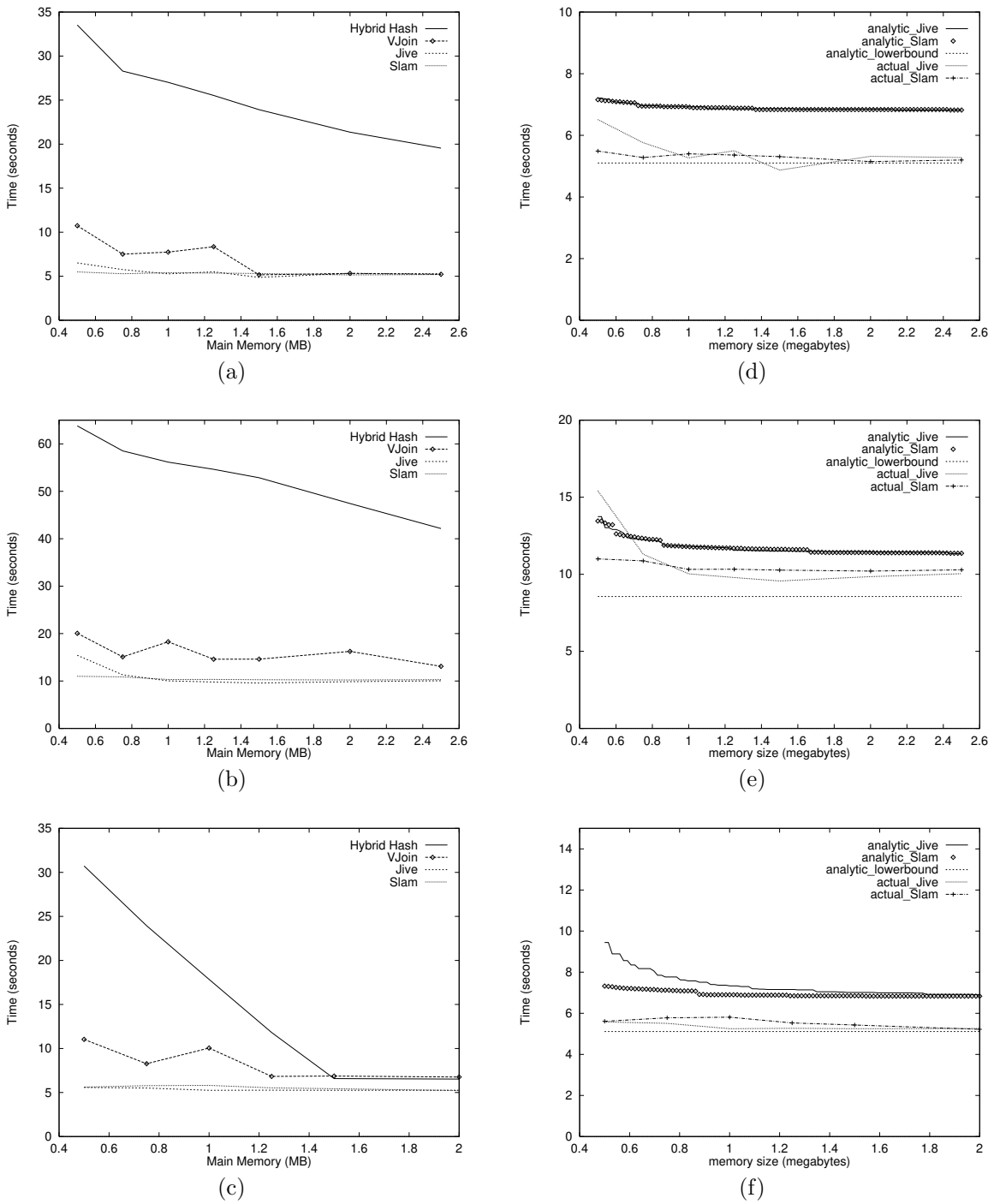


Figure 8: Performance comparison for Example 7.2.

7.3 Assessment of the Results

The analytic and experimental results in Figure 7(d)–(f) and Figure 8(d)–(f) are close, mostly within 25% of each other. We do not expect an exact match because disk drives are complex mechanisms whose behavior we have only approximated. In particular, our cost model ignores track buffering, which may cause it to slightly overestimate the cost. Our cost model also assumes writes happen at the same speed as reads, which is not true in practice (see Appendix B.1), and which may cause it to slightly underestimate the cost. Finally, since our experimental relations do not fill the disk, it is possible that the cost model slightly overestimates the seek time because the actual seeks are between tracks that are close to one another.

Our results also indicate that the CPU time component of performance time is small. Thus even if CPU processing is not overlapped with I/O, our I/O cost model still does a good job predicting the relative performance of the algorithms in terms of total execution time.

7.4 Implementation Experience

We learned several lessons while implementing the various algorithms. Some of these lessons have been learned by others before us in different contexts.

- Even with substantial tuning, an implementation on top of a software layer (such as the Unix file system) rather than a raw file system cannot come close to generating optimal disk throughput. A preliminary implementation of ours on top of the Unix file system gave correct *relative* performance curves, but absolute performance numbers that were four times slower than optimal.
- Bulk I/O is important. Buffering of all inputs and outputs needs to be considered carefully in order to avoid unnecessary seek time and rotational latency.
- Disks are complex devices. For our disks, writes are more expensive than reads by a nontrivial amount. Track caching takes place in a fashion that is hard to control and difficult to explain.
- Developing a cost model and a realistic implementation together lead to more robust development. Our cost model allowed us to observe and correct performance bugs in the implementation, while our implementation also highlighted inaccuracies in the cost model, which we were then able to improve.

8 Extensions

We outline below several extensions to Jive-join and Slam-join. More detail about these extensions can be found in [Li and Ross, 1996].

Multi-level Recursion When the inputs are larger than the memory bound (Equation 3), it is possible to perform an additional partitioning step (with an associated extra I/O cost) to subsequently apply Slam-join or Jive-join.

Tapes Jive-join and Slam-join are particularly well-suited to joining inputs that are stored on tape. The input relations and the join index are accessed purely sequentially. Assuming that sufficient disk space was available for the temporary files and the output result, Jive-join and Slam-join would be good choices for performing the join with a pre-existing join index. Our cost model would have to be modified to model the characteristics of a tape drive.

Multiple Relations We can extend Jive-join (and Slam-join) to joins of more than two relations, assuming that we are given a join-index having a tuple-id column for every participating relation. Star-schema design, as advocated for decision support systems [Kimball and Strehlo, 1995], is particularly well-suited to the use of multi-way join indices. The basic idea for our extension is to create a *multidimensional* set of partitions. Under slightly more stringent (but still reasonable) memory requirements, the resulting algorithms still make only one pass through each input relation.

Parallelism Slam-join and Jive-join can be parallelized across multiple processors and multiple I/O devices. Suppose that each relation is horizontally partitioned among a number of devices, with the partitioning ranges chosen to match the partitioning tuple-id ranges chosen by Jive-join or Slam-join. Then in the first stage of Jive-join each horizontal partition could be processed independently with the corresponding

horizontal partition of the join index. The result would be a set of horizontally partitioned temporary files. The second step of Jive-join could also be parallelized. The horizontal partitions of the second relation, together with the corresponding temporary files can be processed independently. The communication overhead of such a scheme depends on the data layout and machine architecture, and is beyond the scope of this paper.

Pipelining Jive-join can be altered slightly to fit into an inter-operator pipeline by rereading the first vertical partition while the second is being generated, in order to generate a pipelined tuple stream. We call this variation "Pipelined Jive/Slam-join". The baseline TID-join algorithm performs exactly as well as Pipelined Jive/Slam-join. (In fact this equality of performance holds *always*, due to the structure of the algorithms.) However, Jive/Slam-join always outperforms both (but materializes the join result). Thus, if one needs a materialized join result, Jive/Slam-join is the method of choice.

9 Related Work

Valduriez first proposed and analyzed an efficient algorithm based on join indexes [Valduriez, 1987]. One very important contribution of that work was to demonstrate that, under many circumstances, having the join index allows one to compute the join significantly faster than Hybrid hash join. Slam-join is similar to Valduriez's algorithm in the initial phase. A greedy approach is used to process as much of J and R_1 as possible during each pass. Unlike Valduriez's algorithm which reads R_2 multiple times during each pass and incurs repetitive I/Os, Slam-join processes R_2 in a lazy fashion, i.e., only after R_1 is completed processed. By adopting a vertically partitioned data organization for join results, Slam-join can make just a single pass through R_2.

An extension of the Jive-join technique presented here to object-oriented databases is given in [Ross, 1995]. In that context, there is no join index; tuple-identifiers from one relation are effectively embedded in the other. Related techniques for pointer-based joins have been presented in [Shekita and Carey, 1990]. Shekita and Carey consider pointer-based versions of standard join algorithms and demonstrate their relative performance. Their pointer based version of Hybrid hash join is sometimes an improvement of standard Hybrid hash join for large relations. However, they still access the full tuples in the first relation multiple times during an initial hashing phase and a subsequent matching phase.

Cheng et al. have described a technique that joins a relation R_1 with an index into R_2, then sorts the intermediate result by the R_2 tuple-id before dereferencing the tuple-ids [Cheng et al., 1991].

The performance analysis work closest to ours is [Blakeley and Martin, 1990]. Blakeley et. al. conducted a detailed simulation study comparing the performance of Valduriez's algorithm, materialized views and Hybrid-hash join in a centralized environment. That study includes the update cost of maintaining the join index and materialized view, so some of the results are not directly comparable to ours. However, conclusions drawn after excluding the update cost are consistent with our results.

Jive-join and Slam-join can be integrated into a query optimizer in the same way that any other join algorithm can. An example of the use of Jive-join would be in the SQLmmp system [Colby et al., 1994]. That system manipulates tuple-ids rather than full tuples to minimize the number of times that full tuples are looked up in processing a query. The final operation in the SQLmmp system is the computation of the full join given a join index, an operation they call `MAKEREL`. In [Colby et al., 1994] it is implicitly assumed that the join result fits in memory. Jive-join and Slam-join solve the problem for cases in which the join result and input relations are much larger than main memory.

Join indices have been used in [Murphy and Rotem, 1993] to define a graph between pages in two relations based on whether the pages contain a pair of matching records. This graph is then used to help in parallelizing the join. Hierarchies of join indices are used in [Xie and Han, 1994] to speed up navigation in object-oriented databases. Join indices have also been used in spatial databases [Rotem, 1991, Lu and Han, 1992].

An extension of Jive-join that is particularly useful for self-joins is presented in [Lei and Ross, 1997].

10 Conclusions

We have presented two new algorithms, called Jive-join and Slam-join, for performing joins given a join index. We have derived detailed analytic cost formulas for the performance of our algorithms. Each requires one pass through each input relation, one pass through the join index, and two passes through temporary files whose

total size is half that of the join index. We have experimentally verified the performance of our algorithms and validated our cost model.

We have shown how selection conditions can be incorporated into the algorithms. We do not rely on selection conditions to reduce the size of the required input data to the point that it fits into main memory. Thus, unlike previous algorithms, our algorithms allow the efficient solution of queries with only mildly selective selection conditions.

We have demonstrated that:

- A single pass of each input relation can be guaranteed under lenient memory requirements.
- For joins of large input relations, the performance of Jive-join and Slam-join can be significantly better than both Valduriez's algorithm and Hybrid hash-join.
- Almost all of the I/O performed is sequential.
- Skew does not affect the performance, and Slam-join can adapt to memory fluctuations.

Both of our algorithms could be used in a conventional database system. Jive-join and Slam-join can, under many circumstances, deliver better performance for decision support systems than previously proposed algorithms. We describe the benefits of using Jive-join in the context of distributed query processing in [Li and Ross, 1995].

Acknowledgements

The authors wish to thank Damianos Chatziantoniou, Shu-Wie Chen, Akira Kawaguchi, and Hui Lei for suggesting several improvements to the presentation of this paper. Hui Lei observed that techniques for multi-level recursion applied to Jive-join and Slam-join. Thanks also to Mike Carey for clarifying some of the details of [Haas et al., 1993, Haas et al., 1997].

References

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

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

[Bitton et al., 1983] Bitton, D., DeWitt, D. J., and Turbyfill, C. (1983). Benchmarking database systems: a systematic approach. In *Proceedings of the 1983 VLDB conference*, pages 8–19.

[Blakeley and Martin, 1990] Blakeley, J. and Martin, N. (1990). Join index, materialized view, and hybrid-hash join: a performance analysis. In *Proc. IEEE Int'l Conf. on Data Eng.*, pages 256–263.

[Blasgen and Eswaran, 1977] Blasgen, M. and Eswaran, K. (1977). Storage and access in relational data bases. *IBM Systems Journal*, 16(4).

[Bratbergsengen, 1984] Bratbergsengen, B. (1984). Hashing methods and relational algebra operations. In *Proceedings of the VLDB Conference*, pages 323–333.

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

[Cheng et al., 1991] Cheng, J. M., Haderle, D. J., Hedges, R., Iyer, B. R., Messinger, T., Mohan, C., and Wang, Y. (1991). An efficient hybrid join algorithm: A DB2 prototype. In *Proc. IEEE Int'l Conf. on Data Eng.*, pages 171–180.

[Chou et al., 1985] Chou, H., DeWitt, D. J., Katz, R. H., and Klug, A. C. (1985). Design and implementation of the Wisconsin storage system. *Software – Practice and experience*, 15(10):943–962.

[Colby et al., 1994] Colby, L. S., Martin, N. L., and Wehrmeister, R. M. (1994). Query processing for decision support: The SQLmmp solution. In *Proceedings of the Conference on Parallel and Distributed Information Systems*, pages 121–130.

[DeWitt et al., 1984] DeWitt, D. J. et al. (1984). Implementation techniques for main memory database systems. In *Proceedings of the ACM SIGMOD Conference*, pages 1–8.

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

[Graefe, 1992] Graefe, G. (1992). Performance enhancements for hybrid hash join. Technical Report 606, University of Colorado, Boulder.

[Graefe et al., 1994] Graefe, G., Linville, A., and Shapiro, L. (1994). Sort versus hash revisited. *IEEE Transactions on knowledge and data enginnering*, 6(6).

[Gupta and Mumick, 1997] Gupta, A. and Mumick, I. S., editors (1997). *Materialized Views*. MIT Press, Cambridge, MA. To appear.

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

[Haas et al., 1997] Haas, L. M., Carey, M. J., Livny, M., and Shukla, A. (1997). Seeking the truth about ad hoc join costs. *VLDB Journal*, 6(3):241–256.

[Hoare, 1962] Hoare, C. A. R. (1962). Quicksort. *Computer Journal*, 5(1):10–15.

[Kemper and Moerkotte, 1990] Kemper, A. and Moerkotte, G. (1990). Access support in object bases. In *Proceedings of the ACM SIGMOD Conference*, pages 364–373.

[Kim, 1980] Kim, W. (1980). A new way to compute the product and join of relations. In *Proceedings of the ACM SIGMOD Conference*, pages 179–187.

[Kimball and Strehlo, 1995] Kimball, R. and Strehlo, K. (1995). Why decision support fails and how to fix it. *SIGMOD RECORD*, 24(3):92–97.

[Kooi, 1980] Kooi, R. P. (1980). *The Optimization of Queries in Relational Databases*. PhD thesis, Department of Computer and Information Sciences, Case Western Reserve University.

[Lei and Ross, 1997] Lei, H. and Ross, K. A. (1997). Faster joins, self-joins and multi-way joins using join indices. In *Proceedings of the Workshop on Next Generation Information Technologies and Systems*, pages 89–101. Full version to appear in Data and Knowledge Engineering.

[Li and Ross, 1995] Li, Z. and Ross, K. (1995). Federated query processing on the Net using information servers. manuscript.

[Li and Ross, 1996] Li, Z. and Ross, K. (1996). Fast joins using join indices. Technical Report CUCS-032-96, Columbia University.

[Lu and Han, 1992] Lu, W. and Han, J. (1992). Distance-associated join index for spatial range search. In *Proc. IEEE Int'l Conf. on Data Eng.*, pages 284–292.

[MacNicol, 1997] MacNicol, R. (1997). personal communication.

[Mishra and Eich, 1992] Mishra, P. and Eich, M. (1992). Join processing in relational databases. *ACM Computing Surveys*, 24(1).

[Mohan et al., 1990] Mohan, C., Haderie, D., Wang, Y., and Cheng, J. (1990). Single table access using multiple indexes: optimization, execution and concurrency control techniques. In *Proc. International Conference on Extending Data Base Technology*.

[Murphy and Rotem, 1993] Murphy, M. C. and Rotem, D. (1993). Multiprocessor join scheduling. *IEEE Transactions on knowledge and data enginnering*, 5(2):322–338.

[O'Neill and Graefe, 1995] O'Neill, P. and Graefe, G. (1995). Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3).

[Ross, 1995] Ross, K. A. (1995). Efficiently following object references for large object collections and small main memory. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, pages 73–90. Springer LNCS 1013.

[Rotem, 1991] Rotem, D. (1991). Spatial join indices. In *Proc. IEEE Int'l Conf. on Data Eng.*, pages 500–509.

[Shapiro, 1986] Shapiro, L. (1986). Join processing in database systems with large main memories. *ACM Transactions on Database Systems*, 11(3).

[Shekita and Carey, 1990] Shekita, E. and Carey, M. J. (1990). A performance evaluation of pointer-based joins. In *Proceedings of the ACM SIGMOD Conference*, pages 300–311.

[Silberschatz et al., 1997] Silberschatz, A., Korth, H., and Sudarshan, S. (1997). *Database System Concepts*. McGraw Hill, third edition.

[Stonebraker, 1981] Stonebraker, M. R. (1981). Operating system support for database management. *Communication of the ACM*, 24(7):412–418.

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

[Xie and Han, 1994] Xie, Z. and Han, J. (1994). Join index hierarchies for supporting efficient navigations in object-oriented databases. In *Proceedings of the 1994 VLDB conference*, pages 522–533.

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

A Other Join Algorithms

A.1 The Algorithm of Valduriez

In this section we describe the algorithm of Valduriez to compute joins using a join index. We then derive a detailed cost measurement for this algorithm using the formula of Section 4.1. Valduriez makes several assumptions about the availability of indexes. In our model, where it is easy to compute the location of a tuple given its tuple-id, we shall omit the index lookup step from Valduriez's algorithm.

Step V1

As much of J and R_1 as will fit in memory is read in (sequentially) from secondary storage. We read a block from R_1 only if it contains a tuple whose tuple-id is in J. Those tuples of R_1 that match a tuple in J are kept in memory, but the actual join is not computed at this stage. When a tuple from R_1 is read into memory, its memory address is appended to the corresponding entries in the in-memory segment of the join index. 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 in-memory fragment of the join index $\overline{J}$.

Step V2

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

Step V3

If J and R_1 have been exhausted, then we are finished. If not, we repeat steps V1 and V2 until all of J and R_1 are consumed.

A.1.1 Measuring the Cost

We now calculate the values of N_S, N_L, and N_X in order to measure the cost of Valduriez's algorithm. Let u denote the number of passes made in the algorithm, i.e., the number of iterations of steps V1 and V2. We can calculate u as

$$\frac{|J| + |J|/2 + \tau_1|R_1|}{m}.$$

We can compute K, the number of blocks of R_2 accessed in one pass, as

$$K = \beta Y((1 - \left(\frac{u-1}{u}\right)^g)||R_2 \ltimes R_1||, |R_2|/\beta, ||R_2||).$$

where $g = \frac{||J||}{||R_2 \ltimes R_1||}$. [9]

The number of seeks in Step V1 is $3|J|/D$ for J (since it is read sequentially) and $u+3|R_1|/D$ for R_1 (since R_1 and R_2 are on the same disk, and so we need a full seek for R_1 every pass). We now compute the number of seeks needed to process the blocks of R_2. This is where the sorting phase of Step V2 helps. Since access is sequential, the disk head will move monotonically from the start of the relation on disk to the end, rather than alternating between different parts of the disk. We count the seek cost as the total seek cost for moving from the start of the relation to the end. The seek time coefficient for the R_2 blocks is then $u + 3u|R_2|/D$, since we also need a seek to the beginning of R_2 for each pass. Thus, we obtain the formula

$$N_S \quad = \quad 2u + \tfrac{3}{D}\left(|J| + |R_1| + u|R_2|\right)$$

The number of I/O requests in Step V1 is $|J|/\beta + Y(||J||, |R_1|/\beta, ||R_1||)$. The number of accesses to R_2 is $u * Y(||J||/u, |R_2|/\beta, ||R_2||)$. We also need to count $2u$ I/Os because we alternate between R_1 and R_2 which are on the same disk. Thus, we obtain the formula

$$N_{I/O} = 2u + |J|/\beta + Y(||J||, |R_1|/\beta, ||R_1||) + u * Y(||J||/u, |R_2|/\beta, ||R_2||).$$

Unless the size of the join index is very small, $N_{I/O}$ is $|J|/\beta + |R_1|/\beta + u * |R_2|/\beta$.

The number of block transfers in Step V1 is $|J|$ for J and $\beta Y(\tau_1|R_1|, |R_1|/\beta, ||R_1||)$ for R_1. The number of block transfers in Step V2 is $u * K$. Thus, we obtain the formula

$$N_X = |J| + \beta Y(\tau_1|R_1|, |R_1|/\beta, ||R_1||) + u * K.$$

When there is full participation by both relations, when the number of passes is relatively small, and when the join index is small compared with R_1, the expression for N_X is well-approximated by $|J| + |R_1| + |R_1| * |R_2|/m$.

A.2 The TID-join Algorithm

In this section we describe another baseline algorithm which we refer to as "TID-join". It is essentially a non-pipelined version of Valduriez's algorithm, and is based on the work of Kooi [Kooi, 1980].

Step T1

As much of J and R_1 as will fit in memory is read in (sequentially) from secondary storage. We read a block from R_1 only if it contains a tuple whose tuple-id is in J. Those tuples of R_1 that match a tuple in J are kept in memory, but the actual join is not computed at this stage. When a tuple from R_1 is read into memory, its memory address is appended to the corresponding entries in the in-memory segment of the join index. 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 in-memory fragment of the join index $\overline{J}$.

Step T2

$\overline{J}$ is sorted by the tuple-id value from R_2. Then the sorted run, containing both the R_1 records together with matching R_2 tuple-ids, is written to a single temporary file on disk.[10] Repeat steps T1 and T2 until both J and R_1 have been exhausted.

[9] This is not the estimate derived by Valduriez in [Valduriez, 1987]. See [Li and Ross, 1996] for an explanation of why this is a better estimate of K than that given by Valduriez.

[10] Replacement selection can be used to increase the length of the runs, and hence reduce the number of runs.

Step T3

The sorted runs produced after step T2 are merged in memory (in R_2 tuple-id order) and joined with R_2. Only matching R_2 tuples are read based on the tuple-ids in the sorted runs.

A.2.1 Measuring the Cost

We now calculate the values of N_S, N_L, and N_X in order to measure the cost of the TID-join algorithm. Note that the cost components of TID-join and Slam-join are similar except that the latter doesn't read and write JR_1, and the former doesn't incur the extra seek costs of writing to the TID temporary files, and has a smaller number of I/O requests. One can calculate the size of the temporary file used by TID-join as $z = |JR_1| = ||J||(t+ \ll R_1 \gg)/b$, . (We count $z = 0$, however, if $|R_1| + |J| \leq m$, since no runs need to be written.) Thus we obtain the formulas

$$N_S = \frac{3}{D}\left(|J| + |R_1| + |R_2| + 2z\right).$$

$$N_{I/O} = |J|/\beta + Y(\tau_1||R_1||, |R_1|/\beta, ||R_1||) + Y(\tau_2||R_2||, |R_2|/\beta, ||R_2||) + 2z/\beta.$$

Unless the size of the join index is very small, $N_{I/O}$ is $(|J| + |R_1| + |R_2| + 2z)/\beta$.

$$N_X = |J| + \beta Y(\tau_1||R_1||, |R_1|/\beta, ||R_1||) + \beta Y(\tau_2||R_2||, |R_2|/\beta, ||R_2||) + 2z.$$

In the event that all input relation blocks participate in the join, we can simplify the equation above to $N_X = |J| + |R_1| + |R_2| + 2z$.

A.3 Hybrid Hash Join

Hybrid hash join is described in [DeWitt et al., 1984] and we do not describe it here. The cost model below is a slight modification of the one from [Haas et al., 1993, Haas et al., 1997], which doesn't take into account potential fine-tuning optimizations such as those proposed in [Graefe, 1992], but does consider output buffering. As described previously, we ignore input buffering for relations that are read sequentially. In this section F denotes a hashing "fudge factor" conventionally given the value 1.2.

Let $|R_0| = m/F - x\frac{|R_1|-m/F}{m-x}$, where x is the size of an output buffer in the algorithm. Then we let $|R_1'| = |R_1| - |R_0|$, and $|R_2'| = |R_2| \times |R_1'|/|R_1|$.

$$N_X = |R_1| + |R_2| + 2|R_1'| + 2|R_2'|$$

$$N_{I/O} = |R_1'|/x + |R_2'|/x + |R_1|/\beta + |R_2|/\beta + \frac{|R_1|F - m}{m - x}$$

$$N_S = 2 + |R_1'|/x + |R_2'|/x + 2\frac{|R_1|F - m}{m - x}$$

One can use some elementary calculus to choose the optimal value of x to minimize the cost.

B Physical I/O Performance

B.1 Disk Benchmarking

I/O performance is highly dependent on the underlying disk drive technology, disk controller scheduling and caching mechanisms. Therefore, it is imperative to achieve a good understanding of the disk I/O characteristics when conducting performance studies.

It has been recognized [Haas et al., 1993, Haas et al., 1997, Graefe et al., 1994] that different choice of buffer size per disk I/O request can have a significant impact on the join performance, especially when large data sets are accessed sequentially. For instance [Graefe et al., 1994], using an 8KB I/O buffer instead of a 4KB I/O buffer could reduce the execution time of sort-merge join by almost a factor of 2.

We measured the cost of I/O by changing the buffer size used for each I/O request, and making many repeated requests to the disk device. Random I/O cost is measured by inserting a random seek between

two consecutive I/O requests. The results are shown in Figure 9. Note that by amortizing the rotational latency overhead across multiple blocks, sequential bulk I/Os significantly increase the effective disk bandwidth utilized. However little improvement can be observed once the read buffer size exceeds four blocks (32KB, which happens to be close to our 3.5-inch disk track size).[11] Sequential writes are more expensive than sequential reads because sequential reads can take advantage of track buffering (for instance, a factor of 2 difference was observed when 32KB is used as the buffer size). This read/write performance disparity could have an impact on the optimal memory management scheme of each algorithm. (The model of [Haas et al., 1993, Haas et al., 1997] does not distinguish between read costs and write costs.) Random read and write have similar costs because random reads (even of multiple blocks) do not benefit from track caching: the track cache is reset when the disk seeks to a new track. The large performance gap between sequential and random I/O costs, and between bulk I/O and non-bulk I/O confirms the need for an I/O model that examines costs at this level of detail.

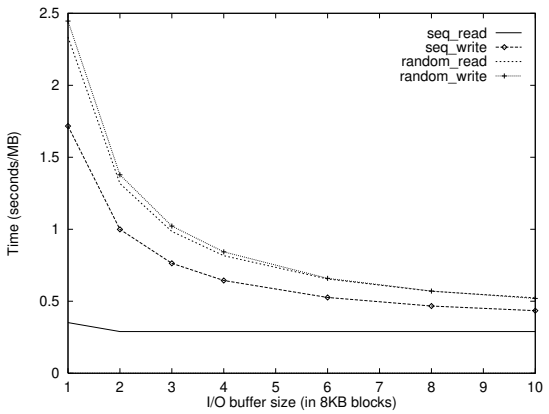


Figure 9: I/O performance versus buffer size

B.2 Reading a Vertically Partitioned Relation

Our algorithms are most efficient when the join result is stored separately in two vertical fragments. Compared with reading a traditionally represented join result, the overhead incurred when reading a vertically partitioned join result for subsequent processing consists of any additional seek time and rotational latency when reading from the two fragments, and the CPU cost of combining two record streams into a single join result record stream. We stated in Section 2.1 that this overhead can be made insignificant by using moderately large input buffers for the two fragments.

An experiment was run to substantiate this claim. Three testing relations each consisting of $100,000$ tuples were used. The join result relation JR tuple width is 164 bytes. The two fragments JR_1 and JR_2 tuple width are 104 bytes and 60 bytes, respectively. The same total amount of buffer space is allocated to read the join result in either representations. The following buffer allocation strategy works better for the vertical representation than other strategies: allocate 4 blocks to read JR_1, and allocate the remaining blocks as a buffer to read JR_2. One representative performance graph is shown in Figure 10.

First notice that the CPU cost for each representation is independent of how the input buffers are allocated, and takes only a small percentage (less than 20%) of the total elapsed time. The CPU overhead incurred by merging is about 0.7 seconds. Secondly, notice that the cost of scanning JR stays the same once the buffer space exceeds 4 blocks. This has also been demonstrated in Appendix B.1. Thirdly notice that the I/O overhead incurred by merging is made negligible when the JR_2 buffer space reaches 32 blocks. Merging actually incurs *less* I/O time when the JR_2 buffer space exceeds 32 blocks![12] The corresponding elapsed time difference is insignificant, especially when compared with the actual join cost (about 10 seconds as in Figure 7(d)).

[11] We couldn't control the disk controller caching based on the standard SCSI specification. So the effects of track caching and prefetching are included in all of the performance measurements.

[12] This unexpected effect is at least partially due to track caching.

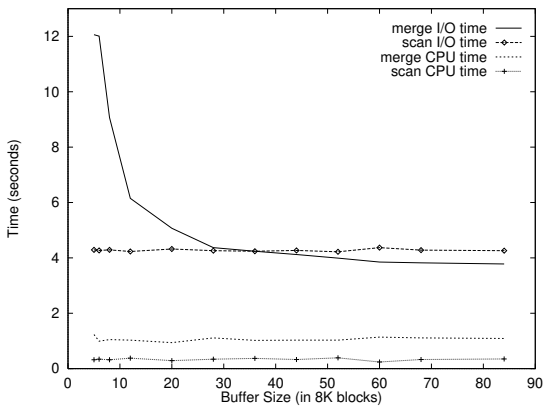


Figure 10: Cost of reading a vertically partitioned relation.