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

Columbia University Technical Report CUCS-032-96
June 26, 1996

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. Each algorithm has features that make it preferable to the other depending on the context in which it is being used. 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 perform this efficiently even when the relations are much larger than main memory, as long as 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. Almost all the I/O of our algorithms is sequential, thus minimizing the impact of seek and rotational latency. The algorithms are resistant to data skew and adaptive to memory fluctuations. They can be extended to handle joins of multiple relations by using multidimensional partitioning while still making only a single pass over each input relation. They can also be extended to handle joins of relations that do not satisfy the memory bound by recursively applying the algorithms. We also show how 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 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.

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems - query processing
Keywords: Relational databases, join, query optimization, 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 [11], and data mining applications with massive amounts of transaction information [1]. 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

*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.

such as indexes. A number of techniques have been developed to perform joins in this setting [5, 6, 10, 16, 21, 28].

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 [31]. A join index between two relations maintains pairs of identifiers of tuples that 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 [24].

In [31], 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 [10]. However, when one analyzes Valduriez's algorithm, it becomes apparent that there is 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.

We propose two algorithms that significantly improve upon Valduriez's 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.

The crucial virtue of both algorithms is that they make just *a single read pass* through each input relation under lenient memory requirements. 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. Other 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 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 [2]. 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 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 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. In Section 8, we extend our techniques to joins of more than two relations. Section 9 discusses various extensions of our algorithms. In Section 10 we survey related work, and we conclude in Section 11.

2 Terminology and Assumptions

The input relations are denoted by R_1 and R_2. (In Section 8, where we consider joins of three or more relations, we also use R_3, R_4, etc.) 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 input relation records in one block have smaller tuple-id values than the records in the next 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[2] (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 [31] 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 [14], 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 [2] 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). (In Section 8, where we consider joins of three or more relations, there will be JR_3, JR_4, etc.) 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 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.[3]

[2] In Section 8, where we consider joins of three or more relations, a join index will be a set of *tuples* of tuple-ids.

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

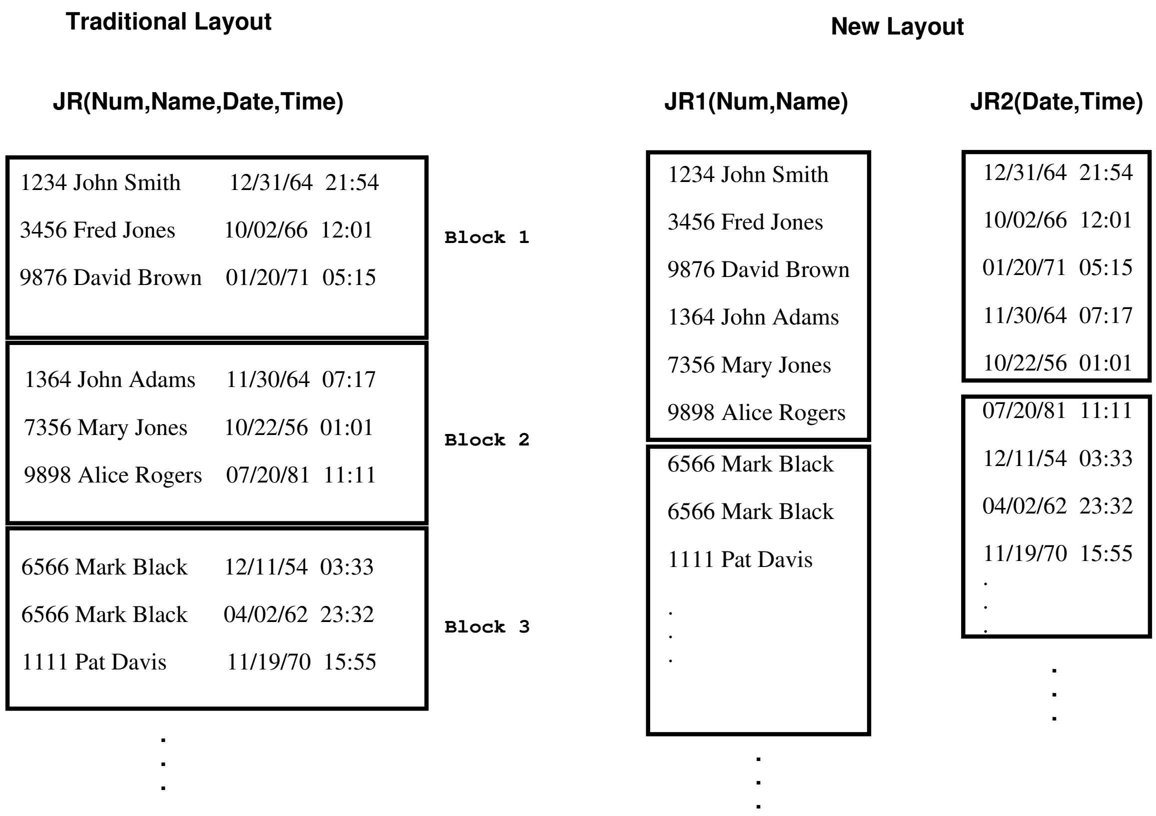


Figure 1: Physical Data Layout for the Join Result.

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.) We emphasize 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.

3 Jive-join: Range-Partition then Process

In this section we present a new algorithm called Jive-Join[4] 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:

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

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 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.

(a) id< 3		
1	1	101 Green
2	2	102 Yellow
2		

(b) 3 ≤id< 6		
4	3	103 Green
5	4	104 White
3	5	105 Evans

(c) 6 ≤id		
9	6	106 Alberts
6	7	106 Beige
7	9	109 Grey

101 Green
102 Yellow
102 Yellow

JR_2(a)

104 White
105 Evans
103 Green

JR_2(b)

109 Grey
106 Alberts
106 Beige

JR_2(c)

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 [14, 7], and correspond to the Fujitsu M2266 disk drive.

Symbol	Meaning
$\lvert R\rvert$	Number of blocks in R.
$\lvert\lvert R\rvert\rvert$	Number of tuples in R.
$\ll R \gg$	Width in bytes of a tuple in R.
r	Number of participating relations. (r is equal to 2 until Section 8.)
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 [33].
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.

Table 1: Table of symbols

Haas, Carey and Livny have proposed a detailed I/O cost model in which seek time and rotational latency are explicit [14]. 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 [14] 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.

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

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 $3|R|/D$ times the average seek cost, where D is the capacity (in blocks) of the disk unit. We count three times[6] the "average" seek cost, estimating that the average seek cost is equal to one third of the time taken to move from one edge of the disk to the other. This rough approximation assumes that seek time can be accumulated in a linear fashion, and that there are no competing accesses to the disk device. If there was contention on the disk device between cylinder accesses, then we would have to count one seek per cylinder, since the seeks between cylinders would not necessarily be small.

We assume that in-memory sorting is done in-place using an algorithm such as quicksort [15].

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 [4].

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.

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

[6] One can show analytically that for linear seek times the time taken to traverse the whole disk is, on average, three times the time taken to move from a random cylinder to another random cylinder on the disk.

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. (For larger relations, Jive-join still applies, but with higher cost; see Section 9.1.)

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.

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

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

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

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

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

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

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

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

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.

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.

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

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.

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 [22]), and use the result to semijoin the join index [31]. 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, and the ad-hoc Hybrid hash-join, presented in Appendix A.2. We use the detailed cost model of [14] 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.

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. 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. 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.

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.

For all of the examples except Example 6.2(a) and (d) below, the Jive-join and Slam-join costs are indistinguishable (and so one curve may seem hidden by the other). Jive-join does better in Example 6.2 because the first relation R_1 is the larger of the two, as discussed in Section 5.3. (Slam-join would have done better had the roles of R_1 and R_2 been switched.)

When comparing algorithms on examples that include selection conditions on nonindexed attributes we use straightforward extensions of Valduriez's algorithm and Hybrid-hash 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 abstract. 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.

(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). The gap between the optimal performance and Jive/Slam-join, is due to the relatively large size of the join index (and of the temporary file).

(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 Valduriez's algorithm perform much better than Hybrid hash join for reasonable memory sizes.

(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. Both Valduriez's algorithm and Hybrid hash-join have improved compared with part (a), but Jive/Slam-join still wins for memories smaller than 900 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. Jive/Slam-join is slightly more expensive that in part (d), but still wins for memories smaller than 400 megabytes. □

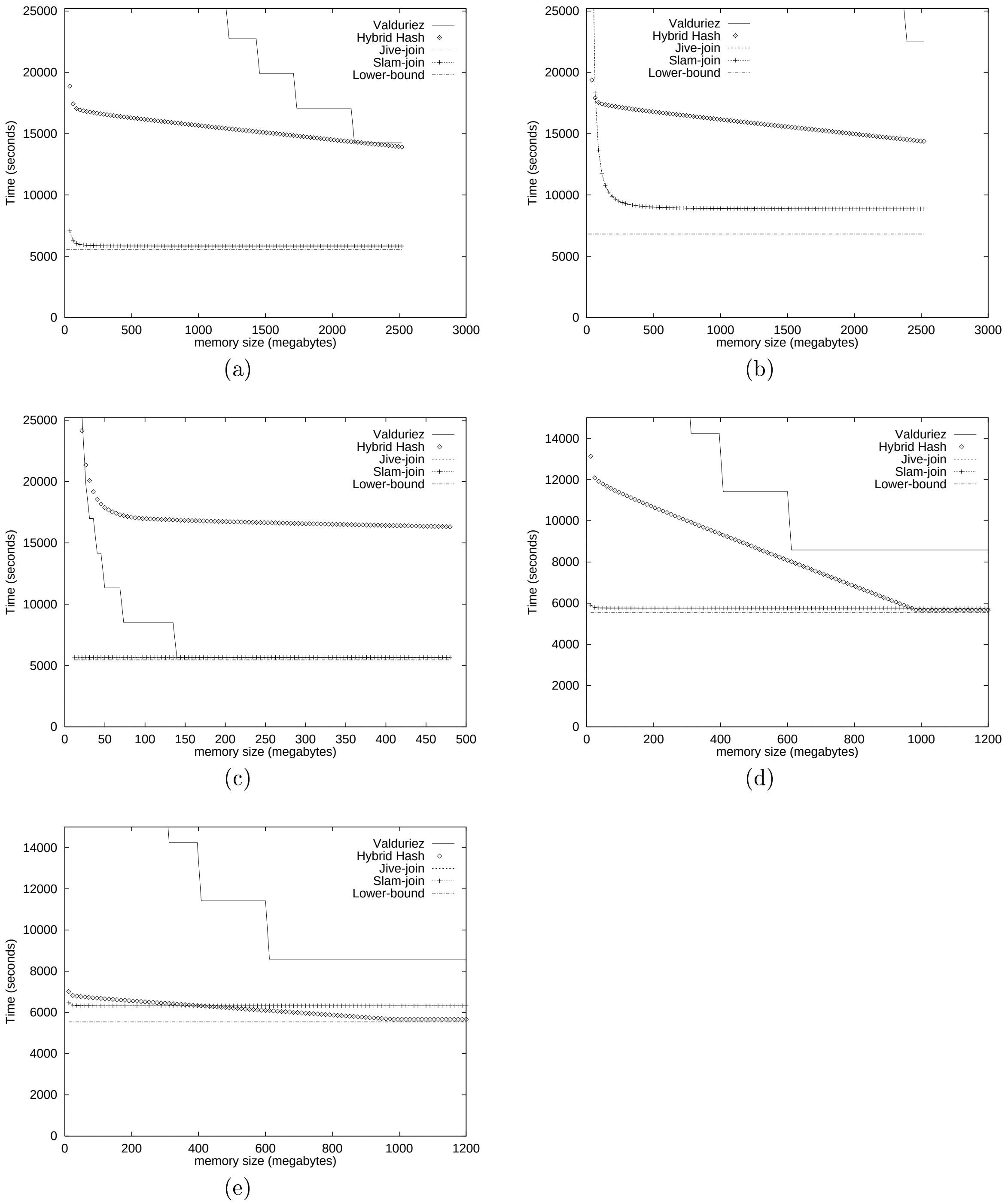


Figure 2: Performance comparison for Example 6.1.

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. We consider several scenarios.[8]

(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. Jive/Slam-join performs close to the optimal. Hybrid hash join just fits into the picture.

(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.

(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. 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 competitive with that of Jive-join. The performance of Valduriez's algorithm has also improved, but it is still not competitive on the range of the graph.

(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. □

[8]Neither Valduriez's algorithm nor Hybrid Hash join showed a significant change in performance in the given range when R_1 and R_2 are switched.

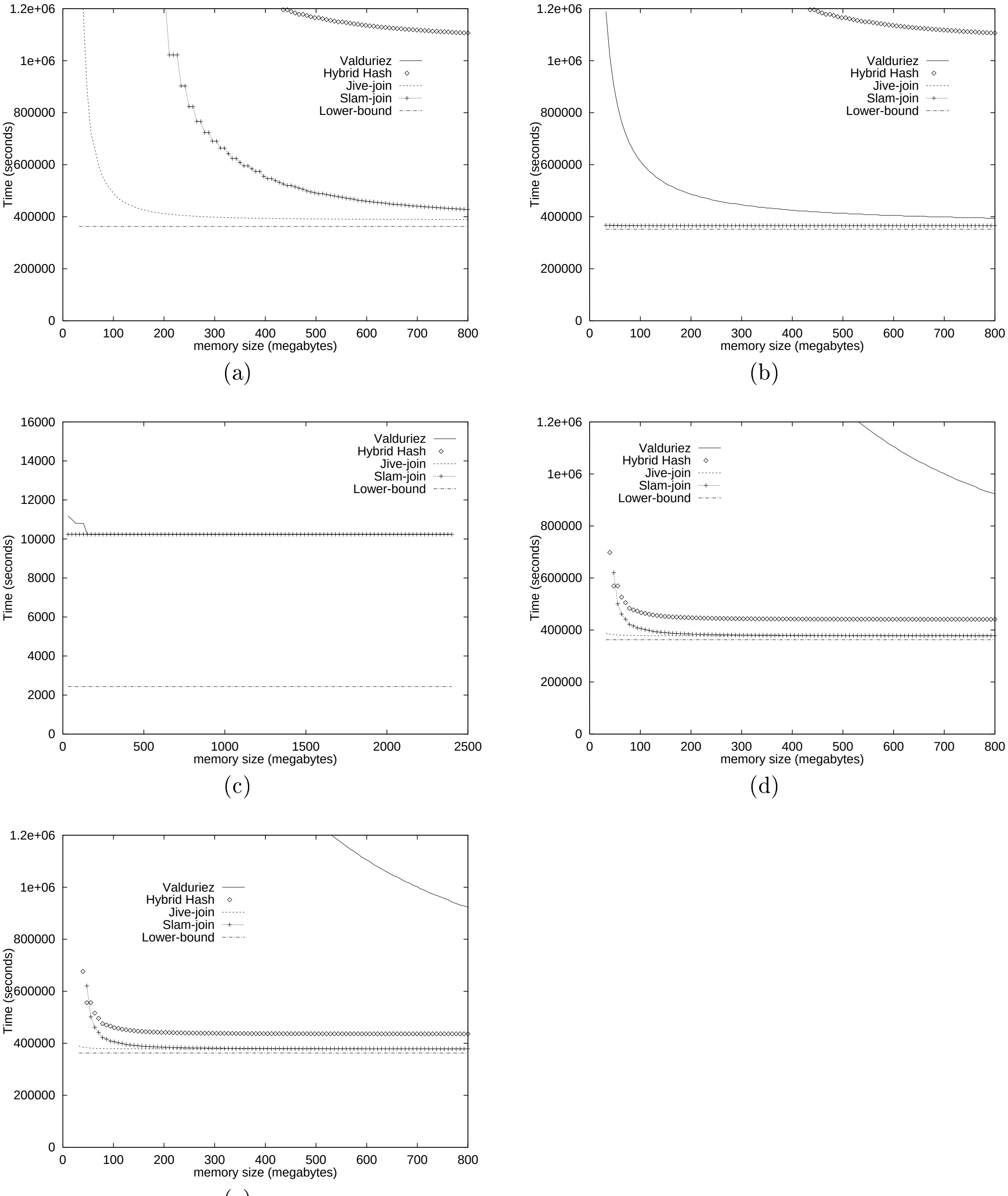
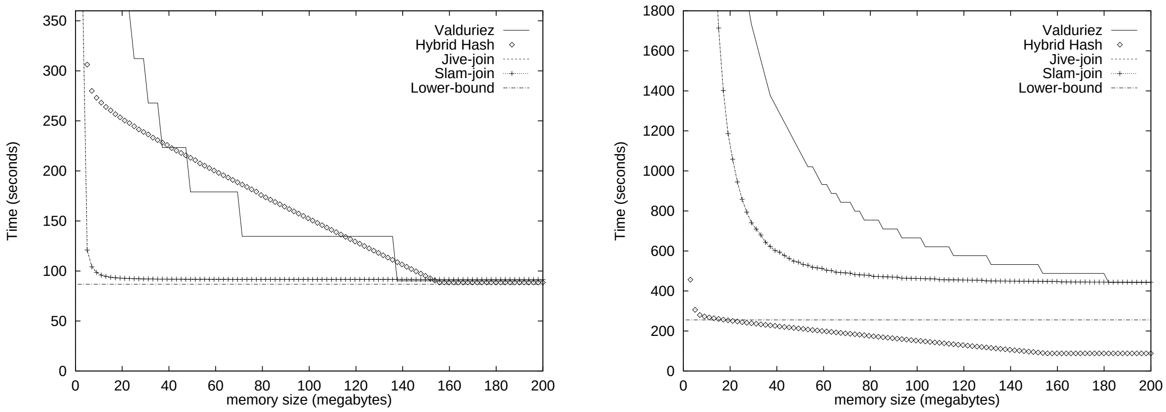


Figure 3: Performance comparison for Example 6.2.



(a) (b)

Figure 4: Performance comparison for Example 6.3.

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.

(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 that Jive/Slam-join and Valduriez's algorithm all have a cost component that is proportional to the size of the join index, which in this case is significantly larger than the input relations. This is the typical case in which Jive/Slam-join perform worse than other techniques. However, 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.

Also notice that 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. □

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, 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 5. 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 three 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.

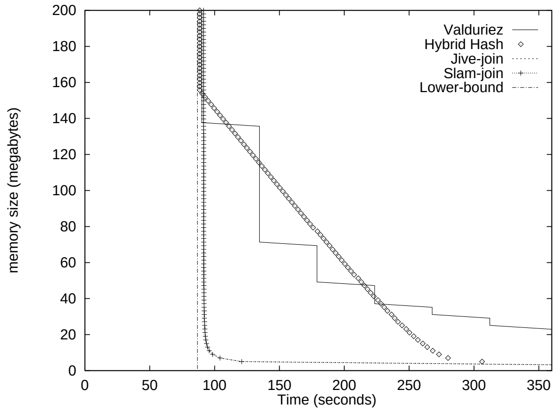


Figure 5: Main memory usage comparison.

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.

Our comparison has focused on large joins because it is in those cases that we obtain the largest speedup. However, our algorithms are still competitive when the join index is very small compared to the input relations, as would result from a selective selection condition on an indexed attribute being used to reduce the join index to a small set of pairs of tuple-ids. In this case, our algorithms take time proportional to the size of the reduced join index, irrespective of the size of the input relations.

The fundamental conclusions to be drawn from these results are:

(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. Our purpose is to both validate the cost model used, and to demonstrate the relative performance of the various algorithms in practice.

7.1 Test Environment

The four algorithms are handcoded 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 [8], 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 [30]. Similar to [14], 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 [14], 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 [31].

Two joining relations are produced using the Wisconsin synthetic database generator [3]. 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 6(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 6(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 6(c).

Comparisons of the experimental and analytic graphs for each case are given in Figure 6(d)–(f). $\square$

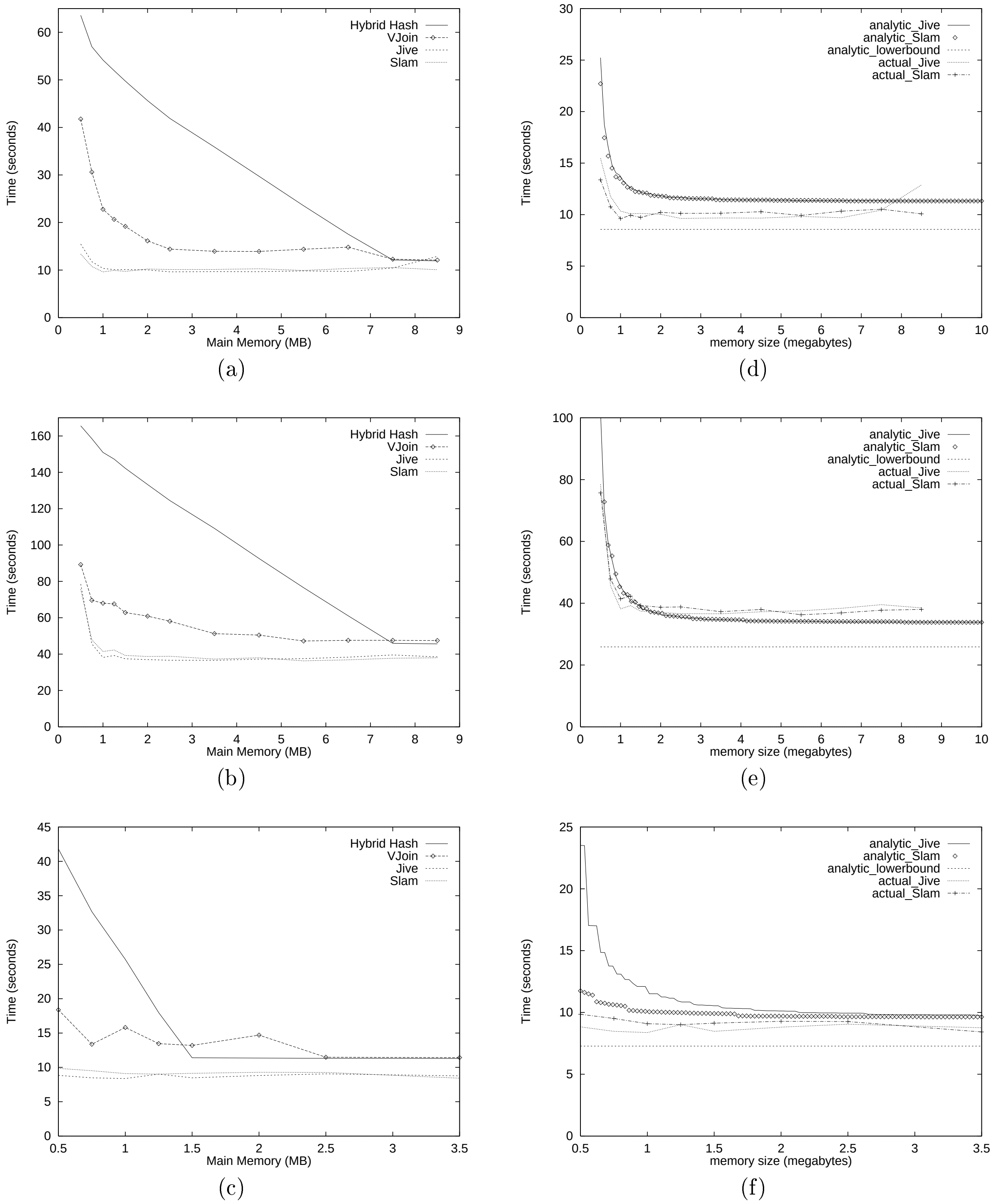


Figure 6: Performance comparison for Example 7.1.

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 7(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 7(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 7(c).

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

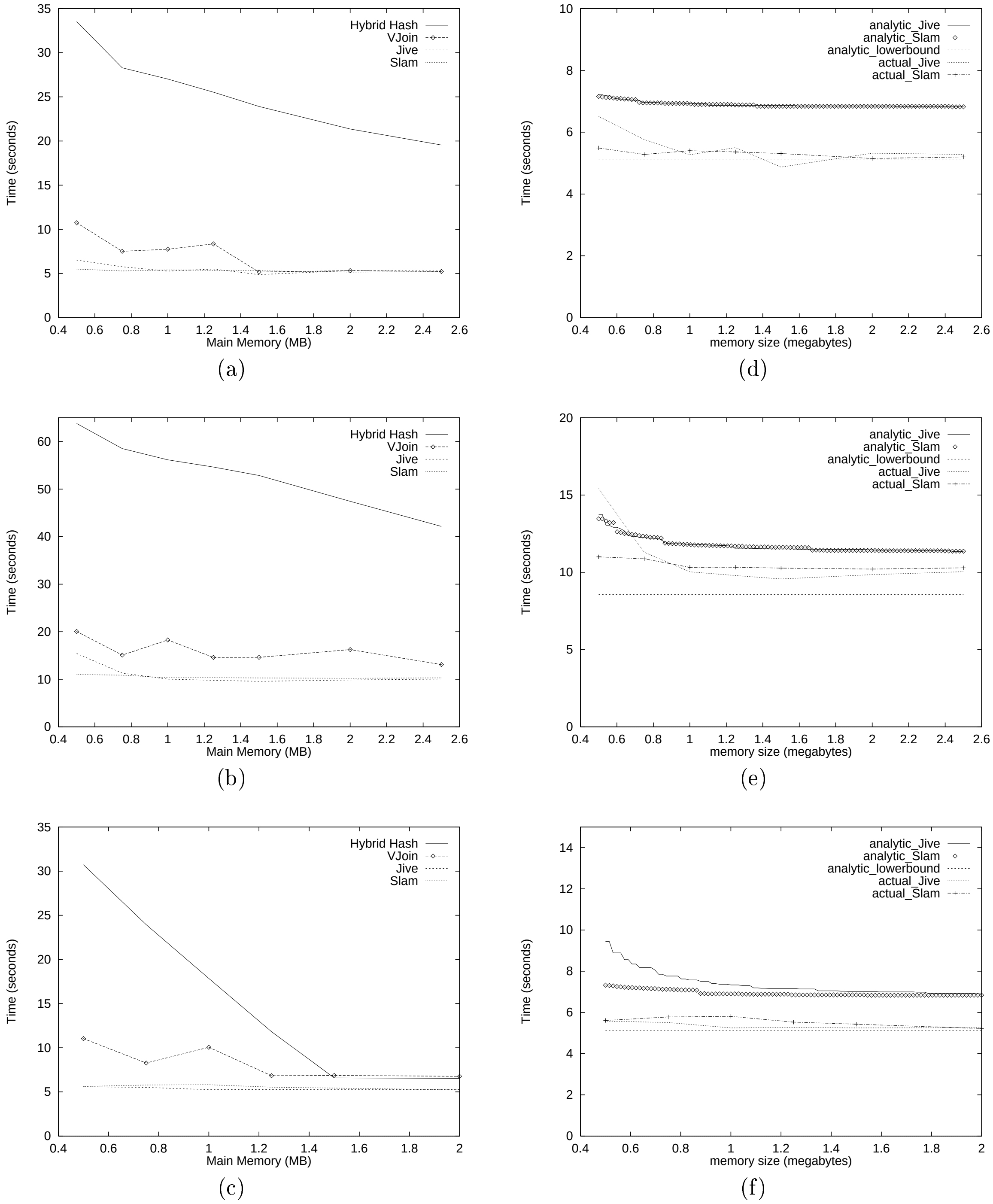


Figure 7: Performance comparison for Example 7.2.

7.3 Assessment of the Results

The analytic and experimental results in Figure 6(d)–(f) and Figure 7(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 Joins of More Than Two Relations

One sometimes needs to take the join of more than two relations. Traditionally, one would take a pair of joining relations, compute the join result, and then use the result as the input for the next join needed. One could either store the intermediate join result or pipeline it into the subsequent join phase. One chooses a join order that minimizes the total cost.

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 [17], is particularly well-suited to the use of multi-way join indices. Multi-way join indices could be maintained by the system. As outlined in Section 6, it can sometimes be more efficient to use a join-index based join method (compared with ad-hoc join methods) even when the cost of computing the join index is included.

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.

In this section we extend Jive-Join to the multi-relation case. (A similar extension can be made to Slam-join.) We will derive memory constraints, measure the analytic cost of the algorithm, and compare the analytic cost with that of competing algorithms.

8.1 Multi-Way Jive-Join

Rather than partitioning just R_2 into disjoint ranges, as done in the Jive-join of two relations, we partition each of $R_2, \ldots, R_r$ into disjoint ranges. We divide the R_i tuple-ids into k_i disjoint ranges, for $i = 2, \ldots, r$. Each tuple $(t_2, \ldots, t_r)$ of tuple-ids can then belong to one of $y = k_2 k_3 \cdots k_r$ partitions. This is illustrated in

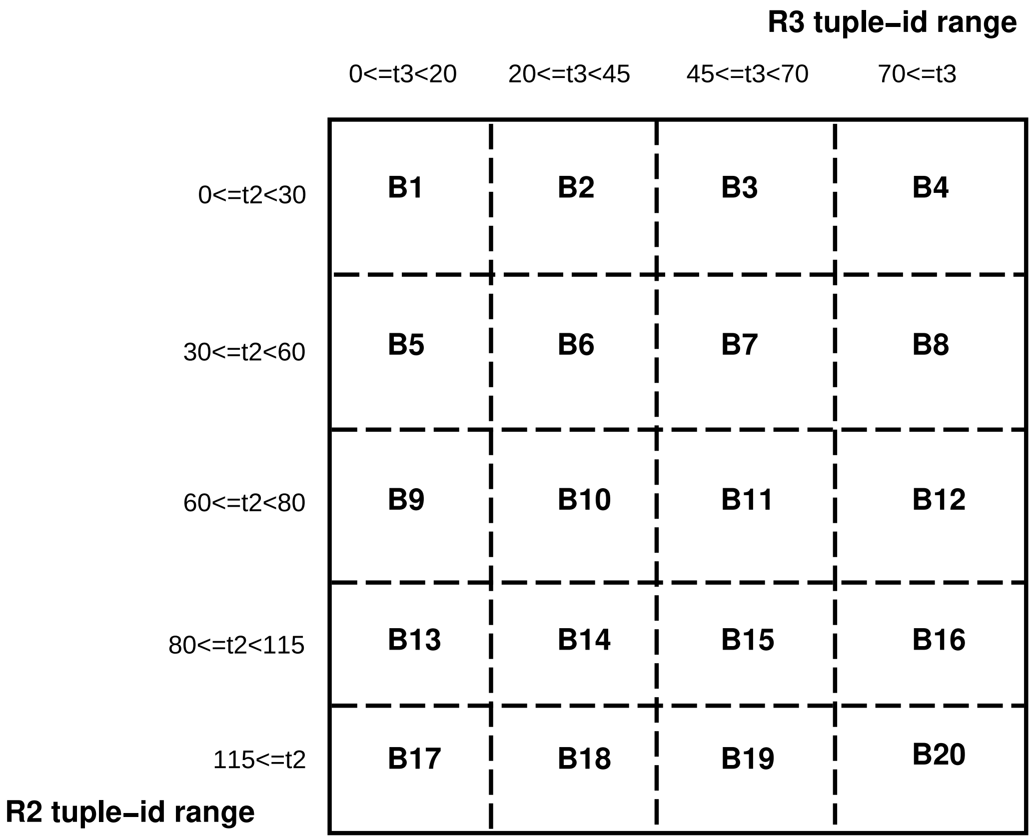


Figure 8: An example partitioning for $r = 3$.

Figure 8 for $r = 3$. In this figure, $k_2 = 5$, $k_3 = 4$, and there are 20 partitions $B_1, \ldots, B_{20}$. We call the union of those partitions with the same range in the R_i tuple-id an *i-segment* of the temporary file. Each i-segment comprises y/k_i partitions. In Figure 8 for example, $B_1B_2B_3B_4$ is a 2-segment, while $B_3B_7B_{11}B_{15}B_{19}$ is a 3-segment. The idea will be to partition the tuples in the join evenly among the i-segments, for each i. (It is not necessary that each *partition* contain the same number of tuples.) The choice of ranges is made with the even division of the tuples in the join index in mind. As illustrated in Figure 8, the ranges do not necessarily divide the whole tuple-id range evenly, since some tuples may participate more often in the join than others.

The join result is stored in r vertical fragments, one for each input relation, using our partitioned data structure. The algorithm consists of three steps:

Step M1

For $i = 2, \ldots, r$ we choose $k_i - 1$ tuple-ids as partitioning elements for R_i. Each partition has (in memory) an associated *output file buffer* of length x blocks, and $r - 1$ associated *temporary file buffers* each of length v blocks.

Step M2

We scan J and R_1 sequentially as before. On each match, we first identify the partition to which this tuple belongs, based on the $R_2, \ldots, R_r$ tuple-ids in the join index. 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 $R_2, \ldots, R_r$ tuple-ids are each written to a corresponding temporary file buffer for the partition. There is one temporary file for each of $R_2, \ldots, R_r$.

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

After finishing Step M2, we have generated part of the output, namely JR_1. The partitions of JR_1 are linked together into a single file. We have also generated several temporary files that are used in Step M3 below to generate the other parts of the output, namely $(JR_2, \ldots, JR_r)$.

Step M3

We perform the following steps for each i in $\{2, \ldots, r\}$.

For each i-segment (in order) of the temporary file we perform the following operations. We read into memory the whole i-segment, storing each partition in the i-segment separately. Additionally, we copy all of the tuple-ids from the partitions into one large array H, which we then sort into ascending order, eliminating duplicates. We then retrieve tuples from R_i in order, retrieving only blocks that contain a matching record according to our sorted array H.

We keep sequentially reading records from R_i until some record from R_i has a higher tuple-id than the largest tuple-id in the i-segment. At that point we write the segment's portion of JR_i. For each of the partitions among the y/k_i in the segment we proceed as follows.

We look at the original version of the temporary file for the partition, and write the corresponding R_i tuples in that order to a partition of JR_i, the join-result output. (We could use binary search to locate the tuples in order, or alternatively store the tuples in a hash table by hashing on the tuple-id.) We then continue with the next partition, and so on, until we have completed the segment.

By the time we have finished with the final i-segment, we have generated all of JR_i. The partitions of JR_i can be linked together into a single file. With JR_1 generated in Step M2, and $JR_i, \ldots, JR_r$ generated in Step M3, we have the required join.

Remarks

We can ignore a relation completely if we know that it does not contribute to the join-result. For example, the attributes of a relation R_j that are selected may all be join attributes that are also available from other relations. Thus we can treat the join index as if it has one less dimension, and perform the join as above without touching R_j.

The definition of multi-way Slam-join is analogous to that of multi-way Jive-join. We read in as much of R_1 as fits into memory in the first phase, and sort those records in the order of the R_r tuple-ids. Each in-memory chunk of R_1 is treated as a 1-segment; 1-segments are not defined in terms of partitioning ranges. The relations $R_2, \ldots, R_{r-1}$ are partitioned as described for multi-way Jive-join. R_r is processed by merging all of the corresponding temporary files (which are in R_r tuple-id order) as in the two-way Slam-join case. Jive-join and Slam-join are still duals: the analysis below applies also to multi-way Slam-join with the roles of R_1 and R_r reversed.

8.2 Requirements for Multi-Way Jive-Join

We need Step M1 and Step M2 to fit in main memory. The analog of Equation 1 is

$$y(x + (r-1)v) \leq m. \tag{6}$$

Step M3 must also fit in main memory. The analog of Equation 2 is

$$|J|/r + \tau_i |R_i| \leq k_i m. \tag{7}$$

We let the variable L_i denote $|J|/r + \tau_i |R_i|$. Combining Equations 6 and 7 with the constraint $x + (r-1)v \geq r$,[9] yields

$$m \geq (r L_2 \cdots L_r)^{1/r} = \left(r \prod_{i=2}^{i=r} (|J|/r + \tau_i |R_i|) \right)^{1/r}. \tag{8}$$

Equation 8 specifies the minimum amount of memory necessary for Jive-join to be applicable. This is often a very reasonable condition. If we assume that $|J|/r \ll |R_i|$, and that $\tau_i = 1$ (so that all tuples participate in the join) then Equation 8 can be rewritten as

$$m \geq r \left(\frac{|R_2|}{m} \right) \cdots \left(\frac{|R_r|}{m} \right).$$

In other words, the products of the fractions by which the input relations exceed memory should not itself exceed a fraction of $1/r$ of the number of blocks of memory. So, if $R_2, \ldots, R_r$ all had the same size, then we could handle relations of size $m(\sqrt[r-1]{m/r})$ blocks. With the parameter settings of Section 4.1, for a memory of size 128MB, this translates into the following values for the size of the participating (equal-sized) relations $R_2, \ldots, R_r$.

r	Size of $R_2, \ldots, R_r$
2	1,048,576MB
3	9,459MB
4	2,048MB
5	968MB
6	623MB
10	291MB
50	144MB

Note that R_1 may be arbitrarily large. Also note that if some relations R_i are small, or have a small number of participating tuples, then there is more leeway for the other relations to be large.

8.2.1 What if the Input Relations are Too Big?

Equation 8 specifies how big the memory must be given the sizes of the input relations. If this condition is violated, what should we do?

One possible answer is that one can perform the first two steps of Jive-join, but partitioning only on $R_1, \ldots, R_j$ where $j < r$. Step M3 can then be performed just for $R_1, \ldots, R_j$. The relations $R_{j+1}, \ldots, R_r$ must be handled separately. For each of these relations we have (from Step M2) the temporary file column corresponding to the order in which the output tuples are to be generated. The temporary file column has no ordering properties, since the partitioning was done only on $R_1, \ldots, R_j$.

[9]This constraint can be weakened by allowing temporary file buffers to contain fractions of a block. If we then only partially fill blocks, we would need to allow for larger temporary files in our cost model.

The join result column corresponding to R_{j+1} can be derived in $\lceil \log_m |R_{j+1}| \rceil$ passes through R_{j+1}, which for realistic databases is at most two passes, i.e., two reads and two writes. This observation applies to $R_{j+2}, \ldots, R_r$ as well. Thus the incremental cost (beyond that needed by Jive-join when Equation 8 is satisfied) is a read and a write of each of $R_{j+1}, \ldots, R_r$. One should choose $R_{j+1}, \ldots, R_r$ to be as small as possible in total size, subject to ensuring that $R_1, \ldots, R_j$ would satisfy a version of Equation 8 with j substituted for r.

8.3 Measuring the Cost

We now calculate the values of N_S, $N_{I/O}$, and N_X in order to measure the cost of our algorithm. The analysis is similar to that of Section 4.4, so we give just the formulas here.

$$N_S = \tfrac{3}{D}\left(|J| + |R_1| + \cdots + |R_r| + (r-1)z\right) + (r-1)z/v + n_1/x.$$

$$N_{I/O} = 2(r-1)y + |J|/\beta + Y(\tau_1||R_1||, |R_1|/\beta, ||R_1||) + \cdots + Y(\tau_r||R_r||, |R_r|/\beta, ||R_r||) + (r-1)z/v + n_1/x.$$

$$N_X = |J| + 2(r-1)z + \beta Y(\tau_1||R_1||, |R_1|/\beta, ||R_1||) + \cdots + \beta Y(\tau_i||R_i||, |R_i|/\beta, ||R_i||).$$

The optimal values of x, y and v turn out to be

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

where $L = L_2 L_3 \cdots L_r$. Since our algorithm will work best when x and v are large, it is clear from the formulas above that R_1 should be the relation with the most participating blocks since R_1 is the only relation not contributing to L. Usually this will happen when R_1 is the largest relation.

8.4 Comparison With Other Algorithms

As far as we are aware, there are no previously proposed algorithms to efficiently handle the join of more than two relations that are larger than main memory using a join index. Valduriez's algorithm [31] does not seem to be extendible to join indexes with more than two columns. Thus, we cannot compare the performance of Jive-join with other join-index based algorithms for $r > 2$.

For joins without a join index, many join algorithms have been proposed, with joins of multiple relations being performed as a sequence of pairwise joins. For comparison purposes, one could consider a sequence of pairwise joins each using a conventional algorithm such as Hybrid hash join [10]. However, there is some difficulty asociated with measuring the cost of such a join sequence because it is necessary to optimize the order in which the component joins are performed [27].

The development of a join-ordering framework for conventional joins is beyond the scope of this paper. Instead, we simply highlight three reasons why we expect our multi-way Jive-join (or Slam-join) to outperform *any* ordering of conventional two-way joins when participating relations and intermediate results are substantially larger than main memory.

The first point is that our algorithm requires just a single pass through each participating relation, under the memory conditions stated in Section 8.2. Single-pass behavior can sometimes be achieved by a sequence of two-way joins, such as when one of the participating relations is small, and when all of the intermediate joins involving this relation are also small. However, if several of the base relations and intermediate joins are substantially larger than main memory, then a single pass of each input relation will usually not suffice. Examples of this behavior in the two-way case are given in Section 6; similar examples could be constructed for $r > 2$.

The second point is that our algorithm *does not need to store full tuples as intermediate results.* Our intermediate structures consist of just tuple-ids, which are likely to be much smaller than tuples from the base relations or intermediate joins. Further, the cost of our intermediate structures grows linearly with r, the number of relations being joined.

In contrast, sequences of large pairwise joins require the writing and reading of intermediate join tuples. Reading and writing intermediate join results to and from disk represents a significant additional I/O overhead. Further, the total size of the intermediate join results can be nonlinear in r.

The third point is that traditional multi-way join algorithms need to represent tuples in intermediate results that do not contribute to the final result because they do not have matching records in subsequent relations. The multi-way versions of Jive-join and Slam-join do not retrieve or store tuples that don't contribute to the join result.

9 Extensions

We outline below several extensions to Jive-join and Slam-join.

9.1 Multi-level Recursion

Both Jive-join and Slam-join can use standard multi-level recursion [12] when the inputs are larger than the memory bound (Equation 3). The term "multi-level recursion" refers to techniques that are applied in a divide-and-conquer fashion.

To apply multi-level partitioning for Jive-join, one creates many memory-sized partitions, and a small number of large partitions in the first phase (Step J1). The large partitions are themselves sub-partitioned (both the R_1 tuples and the R_2 tuple-ids are repartitioned) into memory-sized units. Similar techniques can be used for Slam-join to merge multiple runs into a single run if too many runs were generated in the first phase.[10] As a result, both algorithms can manage with one pass over the larger relation and three passes over the smaller relation (two reading, one writing) when the smaller relation (say R_2) satisfies the following condition.

$$m \geq \sqrt{2}\sqrt[3]{|J| + 2\tau_2|R_2|}. \tag{9}$$

For memory of size 4000 blocks (32MB), R_2 can be as big as 88 terabytes! In practice one would apply multi-level partitioning before hitting the memory bound of Equation 3 in order to reduce the number of seeks for small I/O units.

9.2 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.

9.3 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.

In [19], a scheme is presented where the join index is vertically partitioned among a number of processors, each of which is responsible for one of the input relations.

10 Related Work

Valduriez first proposed and analyzed an efficient algorithm based on join indexes [31]. 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 [25]. In that context, there is no join index; tuple-identifiers from one relation are effectively embedded in the other.

[10] There are certain optimizations to this process that are beyond the scope of this paper.

Related techniques for pointer-based joins have been presented in [29]. 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.

The performance analysis work closest to ours is [4]. 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 [9]. 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 [9] 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 [23] 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 [32] to speed up navigation in object-oriented databases. Join indices have also been used in spatial databases [26, 20].

11 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.
- Recursive application of the algorithms is possible when the relations are larger than the stated bound.
- An extension to join multiple relations is possible, retaining the single-pass property of the inputs.

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 [18].

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 [14].

References

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

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

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

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

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

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

[7] Brown, K., et al. Resource allocation and scheduling for mixed database workloads. Tech. Rep. 1095, University of Wisconsin, Madison, 1992.

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

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

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

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

[12] Graefe, G. Performance enhancements for hybrid hash join. Tech. Rep. 606, University of Colorado, Boulder, 1992.

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

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

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

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

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

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

[19] Li, Z., and Ross, K. A. A new client-server architecture for distributed query processing. Tech. Rep. CUCS-014-94, Columbia University, 1994.

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

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

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

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

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

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

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

[27] Selinger, P. G., Astrahan, M., Chamberlin, D., Lorie, R., and Price, T. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD Conference* (June 1979).

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

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

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

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

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

[33] Yao, S. B. Approximating block accesses in database organizations. *Communications of the ACM 20*, 4 (1977), 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 $g = \frac{||J||}{||R_2 \ltimes R_1||}$ is defined in Section A.1.2.[11]

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.1.2 Estimating the Number of Tuples Accessed in Multiple Passes

Suppose that the join of R_1 and R_2 is such that tuples from R_2 participate multiple times (with different R_1 tuples) in the join. In Step V2 of Valduriez's algorithm, the tuple-ids of the R_2 tuples are sorted and then accessed in order. If there is only one pass, then only one copy of each participating R_2 tuple is accessed. However, if there are multiple passes, then the same tuple may be accessed in more than one of the passes. Unfortunately, Valduriez's analysis does not account for this fact. Valduriez gives an estimate of

$$||R_2 \ltimes R_1||/u$$

[11] This is not the estimate derived by Valduriez in [31]. See Section A.1.2 for an explanation of why this is a better estimate of K than that given by Valduriez.

of the number of tuples accessed in each pass, where u is the number of passes. This estimate makes the implicit assumption that no tuple is accessed in more than one pass.

In this section we analyze the situation more carefully, and obtain a better estimate for the number of tuple accesses in Valduriez's algorithm. Let us assume that the multiplicity of tuples in R_2 is uniform, i.e., that all tuples in R_2 have the same multiplicity in the join. (This is actually a pessimistic assumption, and will give us a higher estimate than if we were to assume some skew. We do need to make such an assumption, because otherwise the problem is underspecified.)

Let g denote $\frac{||J||}{||R_2 \ltimes R_1||}$, i.e., the average multiplicity of tuples from R_2 in the join. Then the expected number of passes with a copy of a given R_2 tuple is $Y(g, ||J||, u)$. If $||J||$ is large relative to g, then $Y(g, ||J||, u)$ is well approximated by

$$u - u\left(\frac{u-1}{u}\right)^g$$

Thus, a better estimate of the number of accesses to R_2 on each pass is

$$(1 - \left(\frac{u-1}{u}\right)^g)||R_2 \ltimes R_1||.$$

To appreciate the difference, let us consider an example in which $||R_2 \ltimes R_1|| = 10^6$, $||J|| = 3*10^6$, $u = 25$, and $g = 3$. Then Valduriez's estimate would be 40,000 tuples per pass, while the more accurate estimate would give 115,000 tuples per pass. This leads to a significant difference in the measured cost of the algorithm. Valduriez's estimate will always be an underestimate when tuples in R_2 appear more than once in the join result.

A.2 Hybrid Hash Join

Hybrid hash join is described in [10] and we do not describe it here. The cost model below is a slight modification of the one from [14], which doesn't take into account potential fine-tuning optimizations such as those proposed in [12], 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 [14, 13] 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 [13], 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).[12] 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 [14] 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.

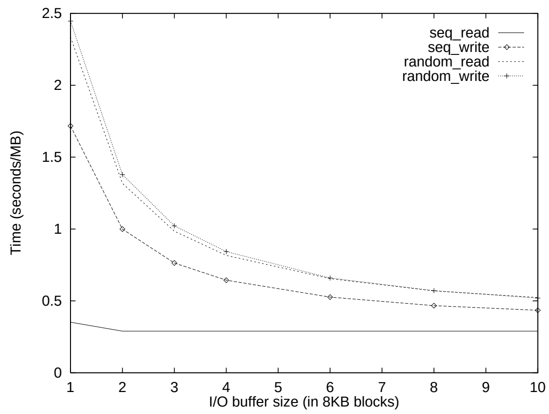


Figure 9: I/O performance versus buffer size

B.2 Reading a Vertically Partitioned Relation

Our algorithms are most efficient when the join result is stored seperately 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.[13]

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![14] The corresponding elapsed time difference is insignificant, especially when compared with the actual join cost (about 10 seconds as in Figure 6(d)).

[12] 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.

[13] A similar analysis to that presented here would also apply when there are more than two fragments, as in Section 8.

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

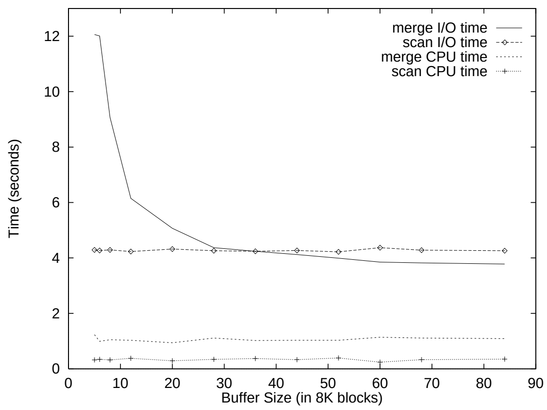


Figure 10: Cost of reading a vertically partitioned relation.