TECHNICAL REPORT NO. 280

Join Index, Materialized View, and Hybrid-Hash Join: A Performance Analysis

by

José A. Blakeley and Nancy L. Martin

June 1989

COMPUTER SCIENCE DEPARTMENT

INDIANA UNIVERSITY

Bloomington, Indiana 47405–4101

Join Index, Materialized View, and Hybrid-Hash Join: A Performance Analysis

*José A. Blakeley** *Nancy L. Martin*
Computer Science Department,
Indiana University

Abstract

This paper deals with the problem of efficiently computing a join between two base relations in the presence of queries and updates to the base relations. We present a performance analysis of three methods: *join index*, *materialized view*, and *hybrid-hash join*. The first two methods are examples of a strategy based on data caching which represent two ends of a spectrum of possibilities depending on the attributes projected in the materialization. The third method is an example of a conventional strategy for computing a join from the base relations. The results of this study show that the method of choice depends on the database environment, in particular, the update activity on base relations, the join selectivity, and the amount of main memory available. A byproduct of this study is a strategy for incrementally maintaining a join index in the presence of updates to the underlying base relations.

1 Introduction

Improving the performance of query processing in relational database management systems continues to be a challenging area of database research. New application areas of relational systems such as engineering design require the storage of more complex objects than the ones required by conventional business applications [8,16]. In addition, designers of object-oriented database systems are choosing to build their systems on top of relational ones [7,12]. Efficient query processing in such systems becomes a more difficult problem because queries involve complex objects which may themselves be composed of complex objects and so on.

*J.A. Blakeley's current address: Information Technologies Laboratory, Texas Instruments Incorporated, P.O. Box 655474, MS 238, Dallas, Texas, 75265.

Active database systems [17] which allow users to specify actions to be taken automatically when certain conditions arise are systems that require very efficient query processing. The completion of many of the actions specified in these systems may be time-constrained in the order of a few milliseconds. In such situations, the system cannot afford to spend a lot of time performing secondary storage accesses, hence caching precomputed queries may be a good strategy.

Several caching mechanisms have recently been suggested to support efficient query processing in extensible relational database systems. Materialized views [1,3,15,21] have been suggested by Stonebraker *et al.* [22] and by Hanson [11] as an efficient alternative for the support of procedures in Postgres [23]. They have also been suggested by several researchers as an alternative approach to structuring the database at the internal level in a relational system [2,14,19,24,26]. Other forms of caching include *links* [9,20], *view indices* [18], and *join indices* [25]. Valduriez [25] has suggested a join index as a data structure to support efficient retrieval of complex objects in object-oriented systems built on top of relational systems.

As a result of these developments, customizers of relational database management systems must decide among several performance-improving mechanisms. For example, if the customizer chooses to use auxiliary relations to improve query efficiency, should he keep full tuples stored (*i.e.*, materialized views) or only the tuple identifiers from the joining relations (*i.e.*, join indices)? On the other hand, the customizer may decide to incorporate more efficient algorithms to compute joins [4,5,6] and rely exclusively on complete re-evaluation of queries.

This paper represents a step in establishing criteria for selecting among the various approaches mentioned above. Specifically, we concentrate on the performance analysis of two caching strategies: a materialized view defined as an equi-join operation between two relations and the corresponding join index. An alternative to caching is the complete computation of a join from the base relations. We have chosen the hybrid-hash join algorithm as a representative of this alternative approach because it consistently outperforms other methods of its type and because it allows us to extend and compare our results with the results obtained by Valduriez [25]. The remainder of this paper analyzes each of these three approaches and compares their costs. Section 2 presents a brief description of the methods while Section 3 describes the performance analysis. Section 4 presents the results of our study, and Section 5 presents our conclusions.

2 Methods

In this section we illustrate how each method works via an example. Consider the two relations shown below. The `Student` relation contains tuples describing student volunteers. Each tuple contains a student's name, major and native country; each tuple also has a unique identifier known

as a surrogate. The Project relation is used to store data pertaining to the on-going summer projects of a university's archeology department. It has attributes for the project title, the project supervisor and the project location as well as a surrogate.

Ssur	Name	Major	NativeCountry
010	S. Bando	Music	USA
011	G. Jetson	Art	Great Britain
012	C. Falerno	History	Italy
013	L. LaPaz	Art	Mexico
014	J. Jones	English	USA
015	P. Valens	Archeology	Mexico

Table 1: Student relation

Psur	Title	Supervisor	City	Country
030	Deforestation	N. Smith	Coba	Mexico
031	Facade Res.	E. Ruggeri	Venice	Italy
033	Mural Res.	A. Montez	Tulum	Mexico
034	Excavation	M. Cox	Lima	Peru

Table 2: Project relation

If the archeology department wished to place student volunteers on projects located in their native country, the following query would be necessary:

```
SELECT Title, Supervisor, City, Country, Name, Major
FROM Project, Student
WHERE Country = NativeCountry
```

We can now examine the auxiliary relations produced by the various proposed speed-up methods to optimize the retrieval of the above query.

2.1 Materialized View

The approach used by the materialized view method is to fully evaluate the join once and store the result for future use. Applying this method would create the relation shown in Table 3 as a result of the initial join. Subsequent evaluations of the example query would be very quick as they would merely consist of reading the materialized view from the disk. However, updating any attribute of

Title	Supervisor	City	Country	Name	Major
Deforestation	N. Smith	Coba	Mexico	L. LaPaz	Art
Deforestation	N. Smith	Coba	Mexico	P. Valens	Archeology
Facade Res.	E. Ruggeri	Venice	Italy	C. Falerno	History
Mural Res.	A. Montez	Tulum	Mexico	L. LaPaz	Art
Mural Res.	A. Montez	Tulum	Mexico	P. Valens	Archeology

Table 3: Materialized view for query

any tuple of Student or Project would necessitate examining the materialized view to determine if it should also be updated and, when necessary, performing the appropriate update.

2.2 Join Index

The join index method tries to store enough information to aid efficient join formation while minimizing the size of the auxiliary relation and the effects of subsequent updates on the additional relation. For each tuple in the join, only the surrogates of its component tuples are stored. Thus, when the join is needed, the appropriate component tuples can be efficiently fetched via a clustered or inverted index. Furthermore, only updates that change the join attributes (in the example, NativeCountry and Country) need to be checked against and possibly posted to the join index relation. The join index for the sample query is shown in Table 4.

Psur	Ssur
030	013
030	015
031	012
033	013
033	015

Table 4: Join index relation for the sample query

2.3 Hybrid Hash-Join

The hybrid-hash join algorithm fully utilizes the available main memory to do an efficient yet complete re-evaluation of the join each time the corresponding query occurs. The efficiency is gained by applying the divide-and-conquer principle to the problem of computing a join. The potentially large component relations are hashed on the join attribute into several smaller subfiles

(also called buckets) each of which will fit into memory; at the end of this stage, each subfile contains tuples from the base relations that may potentially join. The set of tuples within each subfile are then joined in the appropriate order to produce the final join. The hybrid-hash join method further takes advantage of the available main memory space by performing the first sub-join while building the subfiles for subsequent manipulation. This algorithm has the advantages of not requiring any permanent auxiliary relations and being uneffected by updates to the base relations.

3 Performance Analysis

In this section we analyze the performance of three approaches for computing the join of two relations. The following scenarios will be analyzed: (a) materialized view with deferred updates to the view, (b) join index with deferred updates to the join index, and (c) complete re-evaluation using the hybrid-hash join algorithm. By "deferred updates" we mean that in case the joining base relations are updated many times between subsequent queries, updating a materialized view or a join index will be deferred until the time they are queried. Table 5 summarizes the assumptions made with respect to the storage organization of base relations, join index, and materialized view. The organization of base relations and join index follows Valduriez's assumptions [25].

Base relations R and S	clustered B^+-tree on surrogate
Base relation S	nonclustered index on join attribute
Join index JI	clustered B^+-tree on surrogate r
	nonclustered B^+-tree on surrogate s
Materialized view V	Linear hash file on join attribute

Table 5: Assumptions on the organization of base relations.

3.1 Analysis parameters

Table 6 lists the parameters we use to analyze the different scenarios. Similar notation has been used by DeWitt *et al.* [6], Hanson [10], and Valduriez [25].

Before proceeding with the following sections, the reader is invited to take a quick look at the Appendix, where all the basic formulas are introduced.

	Database dependent parameters
$\|R\|$, $\|S\|$, $\|JI\|$, $\|V\|$	Number of pages in relations R, S, join index, and materialized view, respectively
$\|\|R\|\|$, $\|\|S\|\|$, $\|\|JI\|\|$, $\|\|V\|\|$	Number of tuples in relations R, S, join index, and materialized view, respectively
JS	Join selectivity $JS = (\|\|R \bowtie S\|\|)/(\|\|R\|\| * \|\|S\|\|)$
SR	Semijoin selectivity $SR = (\|\|R \rhd\!\!< S\|\|)/\|\|R\|\|$
SS	Semijoin selectivity $SS = (\|\|S \rhd\!\!< R\|\|)/\|\|S\|\|$
T_R, T_S, T_{JI}, T_V	Size (in bytes) of a tuple of R, S, JI, and V, respectively
n_R, n_S, n_{JI}, n_V, n_{i_R}	Number of tuples per page in relations R, S, JI, V, and in the insertion (deletion) file, respectively
$N1_M, N1_J$	Number of passes for first phase of materialized view and join index algorithms, respectively
$N2_M, N2_J$	Number of passes for second phase of materialized view and join index algorithms, respectively
Pr_A	Probability that an update operation modifies the join attribute

	System dependent parameters		**System performance dependent parameters**
$\|M\|$	Number of usable pages of main memory	IO	Time to perform a random IO operation
F	Space-overhead factor for hashing	comp	Time to compare two keys in memory
P	Page size in bytes	hash	Time to hash a key
PO	Average page occupancy factor B^+-tree	move	Time to move a tuple (of any size) in memory
FO	Average fan out of an index node in a B^+-tree		
ssur	Surrogate size in bytes		
sptr	Pointer size in bytes		

Table 6: List of parameters.

3.2 Cost of materialized view with deferred updates

In this subsection we describe the cost of computing a join operation using a materialized view defined as $V = R \bowtie S$. We assume that relations R and S are joined on their common attribute A. Suppose that a transaction updates the base relations R and S. Let i_R, d_R, i_S, and d_S denote the sets of tuples inserted into and deleted from relations R and S, respectively. If $R' = R - d_R$ and $S' = S - d_S$, then the updated state of the view V' can be computed by the expression

$$V' = V \cup (i_R \bowtie S') \cup (R' \bowtie i_S) \cup (i_r \bowtie i_S) - ((d_R \bowtie S') \cup (R' \bowtie d_S) \cup (d_R \bowtie d_S)).$$

The analysis presented here assumes that only relation R is updated, thus

$$V' = (V \cup (i_R \bowtie S)) - (d_R \bowtie S).$$

Furthermore, relation R is changed by update operations only, which get translated into a deleted tuple followed by an inserted tuple, thus $||i_R|| = ||d_R||$. We defer updating the materialized view until the time the join computation is required. Computing the join using the materialized view involves: (1) maintaining the changes to R, (2) computing the changes to V from i_R and d_R, (3) updating V, and (4) reading the new view V'. Because steps (3) and (4) require reading the view, we propose performing step (3) on the fly at the time the view is read in step (4), thus saving the cost of reading V once. The sets i_R and d_R are stored on disk. Since V is stored as a linear hash file on the join attribute A (see Table 5), and since we want to perform the updates on the fly we need to have the changes to the view ordered on $hash(A)$. The next subsections describe the cost of computing each of the steps (1)–(4) above.

(1) Maintaining the sets i_R and d_R

To compute the changes to the view we need to charge the overhead of moving the sets i_R and d_R to an output buffer and writing them to disk when relation R is updated:

$$C_{1.1} = (||i_R|| + ||d_R||) * move + (|i_R| + |d_R|) * IO.$$

The sets i_R and d_R have to be read from disk to update the view for a cost

$$C_{1.2} = (|i_R| + |d_R|) * IO.$$

All algorithms discussed in this paper try to make efficient use of the main memory available. We assume that updates to R are logged in main memory as long as possible. Roughly half of the available memory is devoted to deletions while the other half is used to store insertions. The space used is not exactly half because we must also provide overhead space to sort the deletions or insertions by $hash(A)$ before writing them out to disk. The layout of memory for this part of the algorithm is shown in Figure 1. We will say that Z pages are available for insertions and Z pages are available for deletions where

$$Z \quad = \quad \max_{z \in \{Integer\}} (2 * z + SPACE_{st}(z * n_{i_R})) \leq |M| \qquad (1)$$

Thus, there will be $f = \lfloor |i_R|/Z \rfloor$ full internal sorts and $p = \lceil (|i_R| - f * Z)/Z \rceil$ partial internal sorts of each of the i_R and d_R sets. The total number of runs of this part of the algorithm is $N1_M = f + p$ and the total internal sorting cost is

$$C_{1.3} = 2 * f * CPU_{st}(Z * n_{i_R}) + 2 * p * CPU_{st}(||i_r|| - f * Z * n_{i_R}).$$

To read the sets i_R and d_R sorted by $hash(A)$ we simply need to merge $N1_M$ subfiles for each of the sets i_R and d_R. Merging is performed using a heap data structure of size $N1_M$. The cost is

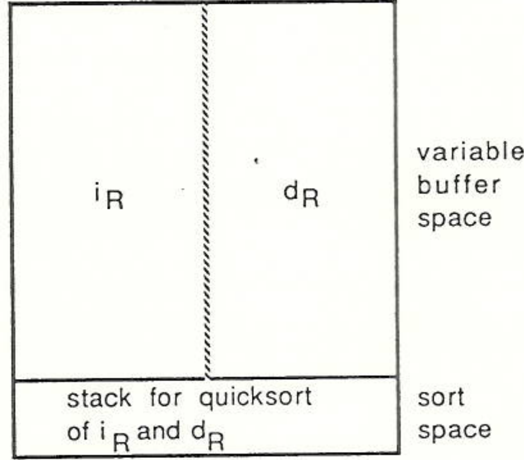

Figure 1: Memory configuration for sorting insertions and deletions

$$C_{1.4} = CPU_{mrg}(||i_R||, N1_M) + CPU_{mrg}(||d_R||, N1_M).$$

At this point we have a cost of $C_1 = C_{1.1} + C_{1.2} + C_{1.3} + C_{1.4}$.

(2) Compute the changes to V

We need only to compute $i_R \bowtie S$ as the set $d_R \rhd\!< V$ is deleted from V in step (3); this is accomplished by merely not outputting tuples in V whose R component matches a d_R tuple. As S has an inverted index on the join attribute A, we use main memory to schedule the accesses to S by ordering the inverted index pointers. We collect $|W|$ pages of i_R as they come out of the merge in the previous step. Call these pages relation W_R. Hence, computing $i_R \bowtie S$ requires $N2_M = |i_R|/|W_R|$ passes of the following steps:

2.1 sort W_R by attribute A,

2.2 compute $W_R \bowtie S$ assuming S has an inverted index on A,

2.3 sort $W_R \bowtie S$ by $hash(A)$. The relation $i_R \bowtie S$ is produced in sorted order by $hash(A)$ as the union of $W_R \bowtie S$ of each pass. Therefore, as step (2) is producing tuples of $i_R \bowtie S$, step (3) can consume them, avoiding an intermediate read/write of relation $i_R \bowtie S$.

Since $2*N1_M$ pages are used to read the different batches of i_R and d_R and we need input buffers for S and V and an output buffer for the updated V, we have $|M|-2*N1_M-3$ pages of available memory left for this step. W_R occupies $|W_R|$ pages. $W_R \bowtie S$ occupies $|W_R|*n_R*||S||*JS*(T_R+T_S)/P$ pages. In addition, the necessary merging and sorting will occupy some space. The memory configuration for this part of the algorithm is illustrated graphically in Figure 2. This memory allocation yields the following computation for $|W_R|$

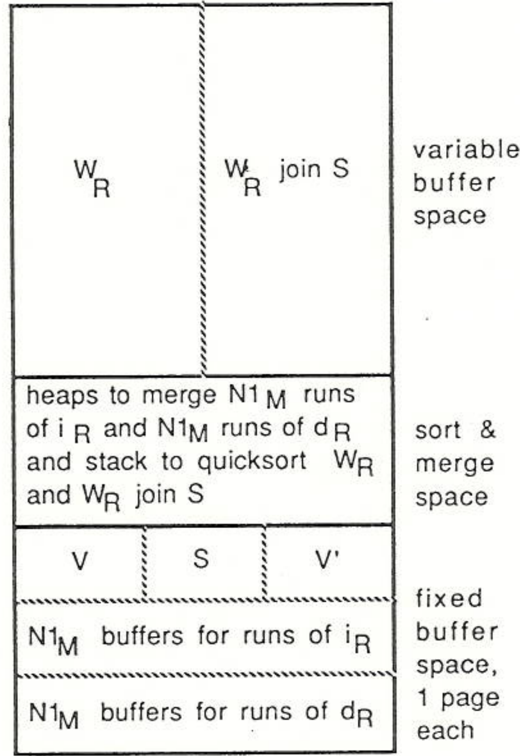

Figure 2: Memory configuration for phase 2 of materialized view algorithm

$$\begin{aligned} |W_R| \;=\; & \max_{w \in \{Integer\}, w \le |i_R|}(w + \tfrac{w * n_{i_R} * ||S|| * JS * (T_R + T_S)}{P} + 2 * SPACE_{mrg}(N1_M, T_R) \\ & + max[SPACE_{st}(w * n_{i_R}), \\ & \quad SPACE_{st}(w * n_{i_R} * ||S|| * JS)]) \le (|M| - 2 * N1_M - 3). \end{aligned}$$

Thus the costs of the steps described above are:

$C_{2.1} = CPU_{st}(||W_R||)$,

$C_{2.2} = IO_{ii}(k, |S|, ||S||) + Yao(k, |S|, ||S||) * n_S * comp + ||W_R|| * ||S|| * JS * move$,

$C_{2.3} = CPUst(||W_R|| * ||S|| * JS)$

where $k = SR * ||W_R||$. The total cost of this portion of the algorithm is

$C_2 = (C_{2.1} + C_{2.2} + C_{2.3}) * N2_M$.

(3) Update the view on the fly

This is done while reading V. Reading the whole view costs

$C_{3.1} = F * |V| * IO$.

When computing the cost of writing the updated pages of V, we need to consider the possibility that some of the $(||i_R|| + ||d_R||) * SR$ groups of adjacent tuples to be inserted or deleted may in fact extend over a page boundary and cause two writes rather than one. While this is a possibility, we assume that it does not occur. Under this assumption, writing the changed pages including inserts and deletes costs

$$C_{3.2} = F * Yao((||i_R|| + ||d_R||) * SR, F * |V|, ||V||) * IO.$$

The cost of merging the tuples is

$$\begin{aligned} C_{3.3} &= ((||i_R|| + ||d_R||) * ||S|| * JS + ||V||) * comp \\ &\quad + F * Yao((||i_R|| + ||d_R||) * SR, F * |V|, ||V||) * n_V * move. \end{aligned}$$

Thus, the total cost of this step is $C_3 = C_{3.1} + C_{3.2} + C_{3.3}$. Finally, the total cost of this scenario is $C = C_1 + C_2 + C_3$.

3.3 Cost of join index with deferred updates

This subsection analyzes the cost of displaying a join where that join is partially materialized via a join index and where updates have occurred since the join index was formed. The algorithm used is based on that of Valduriez [25] but has been extended to include incremental, on-the-fly updates of both the join and the join indices. Valduriez's algorithm exploits the available main memory to process as much as possible of JI and the corresponding $R \rhd\!\!< JI$ at a single time; if all of JI and $R \rhd\!\!< JI$ do not fit into the available memory, the processing is accomplished in several passes. Essentially, we extend the algorithm so that the available memory holds as much as possible of JI and the corresponding $R \rhd\!\!< JI$, i_R and $i_R \bowtie S$.

Specifically, on-the-fly update of join indices involves two phases. The first phase is comprised of one or more passes where the insertions and deletions are saved in the available memory until space is exhausted; then each set is sorted on r, its surrogate for R, and written out to disk. The second phase also involves one or more passes. In each pass, "as much as possible" of JI is read into memory. A heap organization is used to merge the possibly several files of deleted tuples to produce just the deletions which correspond to the portion of JI in memory. Any join index entries in JI that match deleted tuples are "marked" so that they will not be processed further. Next a heap is used to merge the possibly several files of insertions to store in memory the pages of i_R which correspond to the memory-resident portion of the JI. These pages of i_R are subsequently sorted on the join attribute A and pages of S are accessed one page at a time to form $i_R \bowtie S$ which is in turn sorted on s, the surrogate for S. Then the necessary pages of R are read one page at a time to form $R \rhd\!\!< JI$ for the pages of JI which are memory resident. Also at this time, a pointer

is stored with the JI so that the corresponding tuple of R may be accessed quickly. Finally, JI is sorted on s and S is accessed one page at a time. As tuples of S are retrieved, they can be joined with R and merged with $i_R \bowtie S$ to give the join. Also, to keep the join index current, changed pages need to be moved to an output area and rewritten.

The assumptions made in the following analysis are exactly the same as those described in the previous section. On-the-fly-update of join indices can be partitioned into the following categories: (1) maintaining the changes to R, (2) reading and updating the JI and (3) forming the join using JI, d_R and i_R as well as R and S

(1) Maintaining the sets i_R and d_R

The method used to maintain these sets for join indices is similar to the one used for materialized views. However, there are two important differences. As a join index is a "partially materialized view," it is only effected by updates to the join attribute. Thus, if $||i_R||$ tuples are actually inserted during the update process, only $Pr_A * ||i_R||$ need to be saved for future update of the JI. Secondly, since i_R and d_R are ordered by r, no hashing needs to be done. Based on these observations and the fact that the memory configuration is exactly the same as shown in Figure 1, we merely need to slightly reformulate the cost equations of the corresponding part of the materialized view analysis. The cost of storing the pertinent insertions and deletions and then writing them to disk is

$$C_{1.1} = Pr_A * (||i_R|| + ||d_R||) * move + Pr_A * (|i_R| + |d_R|) * IO.$$

Reading the pertinent insertions and deletions from disk to update the join and the JI costs

$$C_{1.2} = Pr_A * (|i_R| + |d_R|) * IO.$$

There will be $f = \lfloor Pr_A * |i_R|/Z \rfloor$ full runs and $p = \lceil (Pr_A * |i_R| - f * Z)/Z \rceil$ partial runs of sorting for insertions and also for the deletions. This gives a total number of runs of $N1_J = f + p$ and a total internal sort cost of

$$C_{1.3} = 2 * f * CPU_{st}(Z * n_{i_R}) + 2 * p * CPU_{st}(Pr_A * ||i_R|| - f * Z * ni_R).$$

As the $N1_J$ subfiles for deletions and the $N1_J$ subfiles for insertions are read into memory, we provide two heaps of size $N1_J$ for the merging of these subfile sets. The cost of merging is

$$C_{1.4} = CPU_{mrg}(Pr_A * ||i_R||, N1_J) + CPU_{mrg}(Pr_A * ||d_R||, N1_J)$$

Thus, the total cost of maintaining the pertinent insertions and deletions is $C_1 = C_{1.1} + C_{1.2} + C_{1.3} + C_{1.4}$.

(2) Reading and updating the JI

Just like merging the sorted deletions and insertions, reading and updating the join index file is actually carried out during a series of one or more passes. However, as the cost of these operations is independent of the number of passes, we show them here in a separate section.

Reading the join index file costs

$$C_{2.1} = |JI| * IO.$$

Using the pertinent deletions to "mark" the entries in JI which correspond to deleted items costs

$$C_{2.2} = (Pr_A * ||d_R|| + ||JI||) * comp.$$

The step where the join indices for the inserted tuples are merged with the already "marked" join index is actually done as part of forming the join itself. The cost of merging and moving the newly inserted tuples to the joined result output area is

$$\begin{aligned} C_{2.3} &= (||i_R|| * Pr_A * ||S|| * JS + ||JI|| \\ &\quad - ||d_R|| * Pr_A * ||S|| * JS) * comp \\ &\quad + ||i_R|| * Pr_A * ||S|| * JS * move. \end{aligned}$$

The process of forming the join will also identify pages of JI which need to be updated by being moved to the join index output buffer and written. Once again, we make the assumption that no i_R or d_R group will overlap page boundaries. The cost of this step is

$$C_{2.4} = Yao((||i_R|| + ||d_R||) * Pr_A, |JI|, ||JI||) * (IO + n_{JI} * move).$$

Thus the cost apportioned to reading and updating the JI may be summarized as $C_2 = C_{2.1} + C_{2.2} + C_{2.3} + C_{2.4}$.

(3) Forming the join

The join is actually formed in one or more passes. Hence, many of the costs involved are determined by the number of passes required which is in turn determined by exactly how many pages of JI can be read into memory during a given pass. Let $|JI_k|$ denote this quantity. The available memory pages, M, must be able to contain one page to input S, one page to input R, one page to store a portion of $S \rhd\!\!< JI_k$, one page to store the join result, one page to form the updated JI, $2 * N1_J$ pages to read in the insertions and deletions, space to merge both the insertions and deletions, as many pages as possible to accommodate the JI and its pointers to the corresponding R tuple, enough pages to store $R \rhd\!\!< JI_k$, enough pages to store the insertions pertaining to JI_k, enough

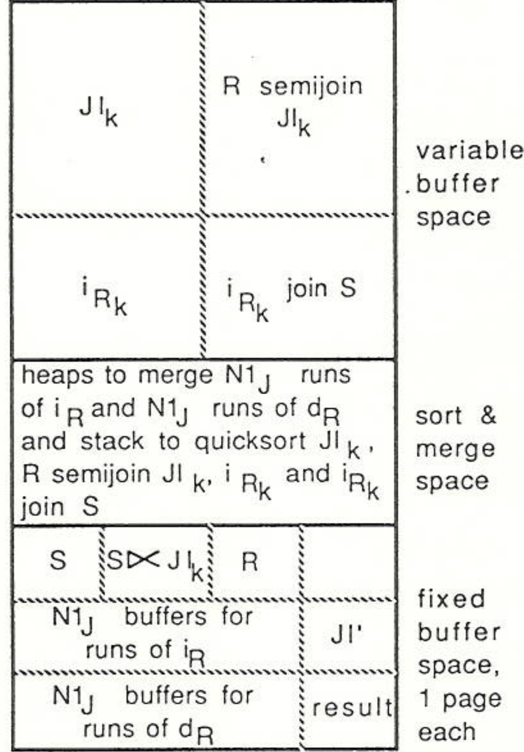

Figure 3: Memory configuration for phase 2 of join index algorithm

pages to store memory-resident $i_R \bowtie S$, and enough space to sort the largest of JI_k, memory-resident i_R and memory-resident $i_R \bowtie S$. The memory requirement for this phase of the algorithm is illustrated in Figure 3. Based on this allocation of memory, we can compute the number of memory-resident pages available for JI_k as follows

$$
\begin{aligned}
|JI_k| \; = \; & \max_{k \in \{Integer\}, k \leq |JI|}(1.5 * k + \tfrac{k * ||R|| * SR * T_R}{|JI| * P} \\
& + \tfrac{k * |i_R| * Pr_A}{|JI|} + \tfrac{k * |i_R| * Pr_A * n_{i_R} * ||S|| * JS * (T_S + T_R)}{|JI| * P} \\
& + 2 * SPACE_{mrg}(N1_J, T_R) + max[SPACE_{st}(k * n_{JI}), \\
& \quad SPACE_{st}(k * |IR| * n_{i_R} / |JI|), \\
& \quad SPACE_{st}(k * |i_R| * n_{i_R} * ||S|| * JS / |JI|)]) \leq M - 2 * (N1_J) - 5.
\end{aligned}
$$

Once $|JI_k|$ is computed, the number of passes is determined by taking $N2_J = |JI| / |JI_k|$. Likewise, the number of pages of R which are memory-resident during any pass is $|R_k| = |R| * SR / N2_J$ and the number of pages of memory-resident i_R is $|i_{R_k}| = |i_R| * Pr_A / N2_J$.

Once the $|JI_k|$ is read into memory and "marked" by the accumulated deletions and the corresponding pages of i_R are read into memory, the latter pages are sorted on the join attribute A, the corresponding tuples of S are accessed via an indirect index on A to form the join, and this portion of $i_R \bowtie S$ is sorted on s. The cost of this step is

$$
\begin{aligned}
C_{3.1} &= (CPU_{st}(|i_{R_k}| * n_{i_R}) + IO_{ii}(SR * |i_{R_k}| * n_{i_R}, |S|, ||S||) \\
&+ Yao(SR * |i_{R_K}| * n_{i_R}, |S|, ||S||) * n_S * comp \\
&+ |i_{R_k}| * n_{i_R} * ||S|| * JS * move \\
&+ CPU_{st}(|i_{R_K}| * n_{i_R} * JS * ||S||)) * N2_J
\end{aligned}
$$

Forming R_k requires reading R using a clustered index, finding which tuples match and moving these to the area reserved for R_k. These operations have an attendant cost of

$$
\begin{aligned}
C_{3.2} &= (IO_{ci}(||R|| * SR/N2_J, |R|/N2_J, ||R||/N2_J) \\
&+ Yao(||R|| * SR/N2_J, |R|/N2_J, ||R||/N2_J) * n_R * comp) * N2_J \\
&+ ||R|| * SR * move.
\end{aligned}
$$

Sorting the JI_k on s incurs the following cost:

$$C_{3.3} = CPU_{st}(|JI_k| * n_{JI}) * N2_J.$$

Finally, accessing S and moving join tuples to the output area requires

$$
\begin{aligned}
C_{3.4} &= (IO_{ci}(||S|| * SS/N2_J, |S|, ||S||) \\
&+ Yao(||S|| * SS/N2_J, |S|, ||S||)) * N2_J \\
&+ ||S|| * SS * move.
\end{aligned}
$$

The cost for forming the join is thus $C_3 = C_{3.1} + C_{3.2} + C_{3.3} + C_{3.4}$ and the full cost for the join index scenario is $C = C_1 + C_2 + C_3$.

3.4 Cost of hash join

As the hybrid-hash join algorithm has been analyzed extensively elsewhere [6] and adding the complicating factor of updates does not invalidate that analysis, we give only a brief presentation here. The algorithm consists of $B + 1$ steps where

$$B = \max(0, \frac{|R| * F - |M|}{|M| - 1}).$$

On the first step R and S are read into memory and hashed into $B+1$ compatible sets; in addition, the first sets, $R0$ and $S0$ are joined at this time while the remainder are written out to disk. The remaining B steps merely consist of processing the sets $R_1, \ldots, R_B$ and $S_1, \ldots, S_B$ by reading them into memory and joining them. q of the tuples will be processed as part of the first pass and $1-q$ will be processed during the subsequent passes. q is calculated as $|R0|/|R|$ where $|R0| = (|M| - B)/F$. The cost of the entire algorithm is

$$
\begin{aligned}
C &= (|R| + |S|) * IO + (||R|| + ||S||) * hash \\
&+ (||R|| + ||S||) * (1 - q) * move + (|R| + |S|) * (1 - q) * IO \\
&+ (||R|| + ||S||) * (1 - q) * hash + ||S|| * F * comp \\
&+ ||R|| * move + (|R| + |S|) * (1 - q) * IO.
\end{aligned}
$$

4 Results

This section presents the performance comparisons of the three methods just analyzed. The default values used for some of the parameters are shown on Table 7 and are the same as those used in previous related studies [6,25].

$\|\|R\|\|$, $\|\|S\|\|$	200,000 tuples	ssur, sptr	4 bytes
$\|M\|$	1000 pages	IO	25 msec
T_R,T_S	200 bytes	comp	3 μsec
PO	0.7	hash	9 μsec
FO	400 entries	move	20 μsec
P	4000 bytes	F	1.2

Table 7: Parameter settings.

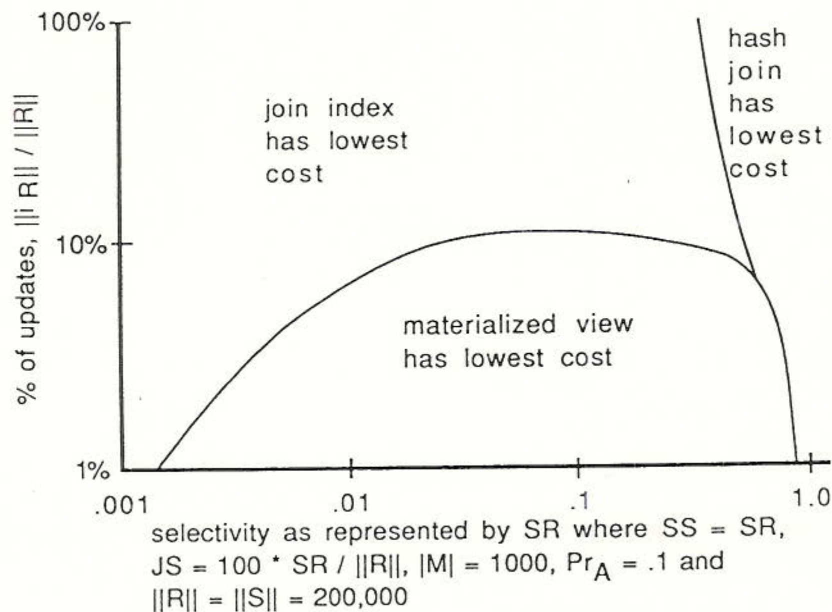

Figure 4: Cheapest method as selectivity and update activity vary.

Figure 4 illustrates the regions where each method performs best for different update activity and join selectivities. The update activity in the system is described by the ratio $||i_R||/||R||$ which represents the percentage of the tuples from the base relation R modified between two consecutive queries that involve the join. The join selectivity factor JS is proportional to the semijoin selectivities SS and SR ($SS = SR$) as $JS = 100 * SS/||R||$. This value has been chosen

to produce a resulting join relation of realistic size. For example, when $SR = 0.01$, the resulting join relation has the same cardinality as an operand relation. We have chosen a join selectivity whose proportion to the semijoin is 10 times larger than the proportion used by Valduriez [25] to best highlight differences among the three methods. Figure 4 shows that materialized views offer the fastest performance when the selectivity is neither extremely high nor extremely low and the update activity is at most moderate. When the selectivity is extremely high, *e.g.*, the join relation is much larger than the relations used to form the join, the hash join method has the lowest cost. If the selectivity is extremely low or the selectivity is moderate but the update activity is large, then the join index algorithm has the fastest execution time.

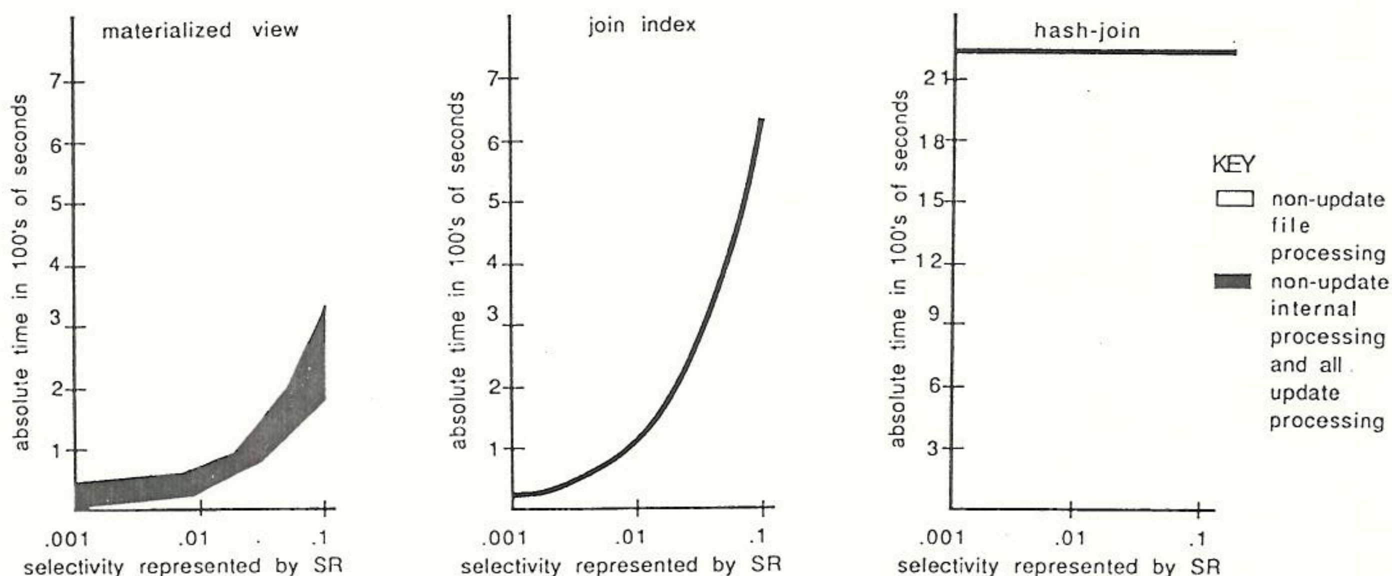

Figure 5: Cost of each method broken down into non-update-related file processing and other costs.

The effect produced in Figure 4 can be best understood by looking at a slightly more detailed cost analysis contained in Figure 5. This diagram breaks down the cost at each selectivity into the file costs that are associated with the basic algorithm and the costs for supporting updates and any non-update-related internal operations. All parameter settings are the same as those used in Figure 4 with the exception that the update activity has been fixed at 6 percent and the values for SR do not range beyond 0.1. For each method, the time associated with the non-updated-related file costs of the basic algorithm is represented by the white area under the total cost curve; the dark area under the curve represents the time associated with update operations and/or non-update-related internal processing. For the materialized view algorithm the dark area under the curve represents only update costs as this method has no internal processing associated with the basic

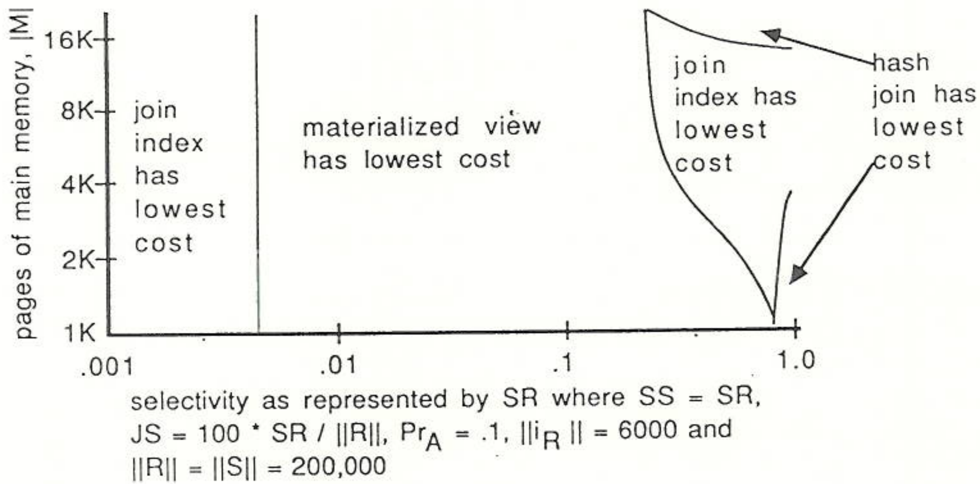

Figure 6: Cheapest method as selectivity and memory size vary.

algorithm. In the case of the join index method, the dark area under the curve represents both update costs and internal costs associated with the basic algorithm; however, the internal costs are small and never exceed 3 percent of the total time. The cost curve for the hash join method is constant with the darkened area representing only internal processing costs associated with the basic algorithm; the internal costs are approximately 1 percent of the total cost. Comparison among the three detailed analyses show that the materialized view method has a competitive advantage because the file time required by its basic algorithm is less – sometimes much less – than the other two approaches. In particular, for low selectivities, reading the relation V takes a fraction of the time to read R and S, rewrite them and then read them again as required by the hash join method. And reading V takes much less time than reading JI, randomly accessing portions of R and several runs of randomly accessing portions of S as required by the join index method. The important implication of this observation is that optimizing the internal processing of the hash join or join index algorithms or the update processing of the join index algorithm is extremely unlikely to effect the comparative advantage of the materialized view method. The only way that the hash join method can favorably compete with the materialized view approach is by drastically increasing the size of V, which is exactly what occurs for extremely high selectivities. The only way that the join index algorithm can beat the time performance of the materialized view method is when the latter method spends sufficiently more time in processing updates, which is exactly what occurs when the selectivity is extremely low or the update activity is high.

We conclude this section with some experimental observations about the effects of various parameters which are held constant in Figure 4. As an example of what happens when these constants take on different values, consider the implications of varying the size of main memory. Figure 6 illustrates the regions where each method is better for different join selectivities and amounts of main memory available. Clearly, the join index algorithm is able to use additional main

memory more efficiently than the other two algorithms in the sense that the join index algorithm reaches the point where all processing can be accomplished in one iteration, sooner than the other two methods. Thus, moderately increasing the size of main memory in Figure 4 would enlarge the area where the join index algorithm performs best. If the memory size were increased by approximately 20K pages, the area where the hash join method is superior would be increased. Similar effects can be observed for changes in other parameters. These changes do not change the general implications of the results shown in Figure 4 but they may considerably alter the boundaries of various regions of superiority. For instance, the size of V is largely dependent on the value used for JS so varying it within the bounds established by SS and SR have a considerable effect on the cost of the materialized view algorithm. In particular, the size of the area where the materialized view algorithm performs best varies inversely with the value of JS. The join index method gains a competitive advantage from only having to process a percentage of the updates. Therefore, it is not surprising that its area of superiority varies inversely with the probability of an update altering the join attribute. Lastly, we consider the effects upon Figure 4 results when the relation size varies. This can be accomplished either by changing the tuple sizes, T_S or T_R, or the number of tuples, $||R||$ or $||S||$. Varying the relation size has an inverse effect on whatever method is doing the most file process at a given selectivity. The materialized view cost is most effected at low selectivities, the join index method is effected at moderate selectivities, and the hash join method is effected at high selectivities.

5 Conclusion

This study has raised several points regarding the effectiveness of the join index, the materialized view, and the hash join algorithm for efficiently computing an equi-join. The method of choice depends upon the values of several parameters. Our results have shown that among these parameters are the selectivity, the update activity, the probability that the joining attribute is updated, the relation and memory sizes. The effects of these parameters on the methods analyzed can be summarized as follows:

- The hash join algorithm performs well when the selectivity is extremely high. Its performance is adversely effected by an increase in relation size. Increasing the size of available main memory does not help the algorithm's performance until the memory is made extremely large. Although its performance is invariant to the update activity and the join attribute update probability, the hash join gains indirectly because increasing these parameters adversely effects the cost of the other two methods.

- The materialized view method performs well for what might be described as "typical values." Primarily, these values include selectivities that are neither extremely high nor extremely low and a low to moderate update activity. This method is only slightly slower than the join index algorithm for very low selectivities. Increasing the relation size adversely effects this algorithm at low selectivities but increases its relative goodness at moderate selectivities. The algorithm does not appear to utilize additional main memory as well as the other two approaches. The materialized view approach is itself unaffected by increasing the join attribute update probability, but it gains relatively when this occurs because the join index method becomes more costly.
- The join index algorithm performs best when the selectivity is low to moderate, the update activity is high and the join attribute update probability is low. This algorithm is favorably effected by an increase in memory and adversely effected by an increase in the attribute update probability. Increasing the relation size favors this method at high and low selectivities but decreases its relative cost effectiveness at moderate selectivities. As a byproduct of the analysis of join indices we have proposed a strategy for incrementally maintaining a join index in the presence of updates to the underlying base relations.

The results just summarized are important because complete or partial caching of joins is a relevant strategy in the following environments: (1) efficient support of procedures as data types in extensible database systems, (2) efficient support of situation monitoring in active databases, and (3) efficient support of querying through methods in object-oriented database systems. Unfortunately, database customizers working in these environments often have only incomplete or imperfect knowledge concerning the parameters which determine the best way of satisfying the efficiency constraints. We propose the following heuristics based on our results: (a) If the join relation is much larger than the two relations which form it, use the hash join algorithm; (b) If the join relation is smaller or not much larger than its base relations and the update activity is less than or equal to 10 percent, cache the join via the materialized view algorithm; and (c) If the join relation is smaller or not much larger than its base relations but the update activity is more than 10 percent, use the join index algorithm to partially cache the join relation. While these heuristics do not guarantee the quickest join, the actual times obtained will generally not be too far from the optimal time.

Although the work described here has already generated some interesting results, there is yet much to be done. There are several places where the internal processing could be optimized and the cost equations described in the paper need to be augmented to account for the projectivity of a join. In addition, the entire analysis should be generalized to investigate the feasibility of maintaining precomputed results for queries involving other additional operators like select and

aggregate functions, joins of more than two relations and arbitrary and possibly unequal sets of insertions and deletions. Eventually, the information gleaned from such an investigation could be incorporated in a system which used the designer's estimates to initially select among algorithms for efficiently supporting queries but also maintained usage statistics so that the system could automatically adapt to the appropriate structures and algorithms after a suitable period of time.

Acknowledgements

This work has benefited from ideas provided by Pedro Celis in early stages of this project.

References

[1] ADIBA, M., AND LINDSAY, B. Database snapshots. In *Proceedings of the Sixth International Conference on Very Large Data Bases* (Montreal, Canada, 1980), pp. 86–91.

[2] BLAKELEY, J., COBURN, N., AND LARSON, P.-Å. Updating derived relations: Detecting irrelevant and autonomously computable updates. To appear in TODS, Sept. 1989.

[3] BLAKELEY, J. A., LARSON, P.-Å., AND TOMPA, F. W. Efficiently updating materialized views. In *Proceedings of ACM-SIGMOD '86 International Conference on Management of Data* (Washington, D.C., May 1986), pp. 61–71.

[4] BLASGEN, M. W., AND ESWARAN, K. P. Storage and access in relational databases. *IBM Systems Journal 16*, 4 (1982), 337–344.

[5] BRATBERGSENGEN, K. Hashing methods and relational algebra operations. In *Proceedings of the Tenth International Conference on Very Large Data Bases* (Singapore, 1984), pp. 323–333.

[6] DEWITT, D. J., KATZ, R. H., OLKEN, F., SHAPIRO, L. D., STONEBRAKER, M., AND WOOD, D. Implementation techniques for main memory database systems. In *Proceedings of ACM-SIGMOD 1984 International Conference on Management of Data* (Boston, MA, June 1984), pp. 1–8.

[7] FISHMAN, D. H., BEECH, D., , CATE, H., CHOW, E., CONNORS, T., DAVIS, J., DERRETT, N., HOCH, C., KENT, W., LYNGBAEK, P., MAHBOD, B., NEIMAT, M., RYAN, T. A., AND SHAN., M. IRIS: An object-oriented database management system. *ACM Transactions on Office Information Systems 5*, 1 (Jan. 1987), 48–69.

[8] HAAS, L. M., FREYTAG, J., LOHMAN, G., AND PIRAHESH, H. Extensible query processing in starburst. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data* (Portland, OR, May 1989), pp. 377–388.

[9] HAERDER, T. Implementing a generalized access path structure for a relational database system. *ACM Transactions on Database Systems 3*, 3 (1978), 285–298.

[10] HANSON, E. A performance analysis of view materialization strategies. In *Proceedings of ACM-SIGMOD 1987 Annual Conference* (San Francisco, CA, May 1987), pp. 440–453.

[11] HANSON, E. Processing queries against database procedures. In *Proceedings of ACM-SIGMOD 1988 International Conference on Management of Data* (Chicago, IL, June 1988), pp. 295–302.

[12] KACHHAWAHA, P., AND HOGAN, R. LCE: An object-oriented DBMS for the research laboratory. In *Spring DECUS U.S. Symposium 1987 (Refereed Papers Journal), Nashville, Tenn.* (Apr. 1987), pp. 43–55.

[13] KNUTH, D. E. *The Art of Computer Programming: Sorting and Searching*, vol. 3. Addison-Wesley, Reading, Ma, 1973.

[14] LARSON, P.-Å., AND YANG, H. Z. Computing queries from derived relations. In *Proceedings of the Eleventh International Conference on Very Large Data Bases* (Stockholm, Sweden, Sept. 1985), pp. 259–269.

[15] LINDSAY, B., HAAS, L., MOHAN, C., PIRAHESH, H., AND WILMS, P. A snapshot differential refresh algorithm. In *Proceedings of ACM-SIGMOD '86 International Conference on Management of Data* (Washington, D.C., 1986), pp. 53–60.

[16] LORIE, R., KIM, W., MCNABB, D., PLOUFFE, W., AND MEIER, A. *Supporting Complex Objects in a Relational System for Engineering Databases.* Query Processing in Database Systems. Springer-Verlag, 1985, pp. 145–155.

[17] MCCARTHY, D. R., AND DAYAL, U. The architecture of an active database management system. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data* (May 1989), pp. 215–224.

[18] ROUSSOPOULOS, N. View indexing in relational databases. *ACM Transactions on Database Systems* 7, 2 (June 1982), 258–290.

[19] SCHKOLNICK, M., AND SORENSON, P. The effects of denormalization on database performance. Tech. Rep. RJ 3082, IBM San Jose Research Center, 650 Harry Road, San Jose, CA 95120, 1981.

[20] SCHMID, H. A., AND BERNSTEIN, P. A. A multi-level architecture for relational data base systems. In *Proceedings of the International Conference on Very Large Data Bases, Framingham* (1975).

[21] SHMUELI, O., AND ITAI, A. Maintenance of views. In *Proceedings of ACM-SIGMOD 1984 International Conference on Management of Data* (Boston, MA, June 1984), pp. 240–255.

[22] STONEBRAKER, M., ANTON, J., AND HANSON, E. Extending a database system with procedures. *ACM Transactions on Database Systems 12*, 3 (Sept. 1987), 350–376.

[23] STONEBRAKER, M., AND ROWE, L. A. The design of postgres. In *Proceedings of ACM-SIGMOD '86 International Conference on Management of Data* (Washington, D.C., 1986), pp. 340–355.

[24] TOMPA, F. W., AND BLAKELEY, J. A. Maintaining materialized views without accessing base data. *Information Systems 13*, 4 (1988), 393–406.

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

[26] YANG, H. Z., AND LARSON, P.-Å. Query transformation for psj-queries. In *Proceedings of the Thirteenth International Conference on Very Large Data Bases* (Birghton, England, Sept. 1987), pp. 245–254.

[27] YAO, S. B. Approximating block accesses in database organizations. *Commun. ACM 20*, 4 (Apr. 1977), 260–261.

Appendix

Basic formulas used in the analysis

Although the formulas used throughout the analysis are similar to or compatible extensions of those used by Valduriez [25], we give a very brief explanation of them. Initial experiments showed that both quicksort and heap merge possess favorable time-space characteristics for sorting and merging, respectively. Costs for these algorithms are based on average case analyses by Knuth [13]. The CPU time to quicksort n tuples is defined by

$$\begin{aligned} CPU_{st}(n) &= 2*(n+1)*\ln((n+1)/11)*comp \\ &\quad +2/3*(n+1)*\ln((n+1)/11)*move \end{aligned}$$

if the sort is on a key that is not hashed, or

$$\begin{aligned} CPU_{st}(n) &= 2*(n+1)*\ln((n+1)/11)*(comp+2*hash) \\ &\quad +2/3*(n+1)*\ln((n+1)/11)*move \end{aligned}$$

if the sort is on a key that must be hashed. The number of overhead pages required to quicksort n items already stored in memory is

$$SPACE_{st}(n) = 2*sptr*\lg(n)/P.$$

Merging n items of size s in a heap of size z requires time and space as shown below. (The n items are assumed to be in a main memory buffer before they are moved to the heap which contains entire items as well as pointers into corresponding buffers.)

$$\begin{aligned} CPU_{mrg}(n,z) &= ((2*n-1)*\lg(z)-3.042*n)*comp \\ &\quad +(n*\lg(z)+1.13*n+\lfloor n/2\rfloor-4)*move \end{aligned}$$

if the keys are not hashed, or

$$\begin{aligned} CPU_{mrg}(n,z) &= ((2*n-1)*\lg(z)-3.042*n)*(comp+2*hash) \\ &\quad +(n*\lg(z)+1.13*n+\lfloor n/2\rfloor-4)*move \end{aligned}$$

if the merge keys are hashed. The space required is given by the formula

$$SPACE_{mrg}(z,s) = z*(s+sptr)/P.$$

The number of page access required to get k records randomly distributed in a file of n records stored in m pages given that a page is accessed at most once is given by Yao's formula [27]

$$Yao(k,m,n) = m - m*\prod_{i=1}^{k}\frac{n-(n/m)-i+1}{n-i+1}.$$

Based on this formula, we can now calculate the IO time for accessing k tuples in a relation having m pages and n tuples via a clustered (IO_{ci}) or inverted index (IO_{ii}) using the following equations.

$$IO_{ci} = [Yao(k, m, n) + Yao(Yao(k, m, n), m/FO, m)] * IO.$$

$$\begin{aligned} IO_{ii} = & [Yao(k, m, n) + Yao(k, n/FO, n) \\ & + Yao(Yao(k, n/FO, n), n/(FO * FO), n/FO)] * IO. \end{aligned}$$

These formulas assume B^+-tree indices with two and three levels of index pages when used as clustered and inverted indices, respectively. The root node is assumed to be permanently stored in main memory.