Join Indices

PATRICK VALDURIEZ
Microelectronics and Computer Technology Corporation

In new application areas of relational database systems, such as artificial intelligence, the join operator is used more extensively than in conventional applications. In this paper, we propose a simple data structure, called a join index, for improving the performance of joins in the context of complex queries. For most of the joins, updates to join indices incur very little overhead. Some properties of a join index are (i) its efficient use of memory and adaptiveness to parallel execution, (ii) its compatibility with other operations (including select and union), (iii) its support for abstract data type join predicates, (iv) its support for multirelation clustering, and (v) its use in representing directed graphs and in evaluating recursive queries. Finally, the analysis of the join algorithm using join indices shows its excellent performance.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures—*graphs, trees*; E.5 [**Data**]: Files—*organization/structure*; H.2.2 [**Database Management**]: Physical Design—*access methods*; H.2.4 [**Database Management**]: Systems—*query processing*; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*indexing methods*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Abstract data types, join, multirelation clustering, recursive query, relational query, semijoin, updates

1. INTRODUCTION

Relational database technology is widely accepted as a basic support technology for evolving application areas like artificial intelligence, CAD/CAM, and so forth. Indeed, relational technology must be extended to fulfill the requirements of these new applications. Compared to conventional (business) applications, they tend to consist of large numbers of more complex queries. Efficient query processing becomes a more difficult problem since complex operations are used extensively. A first step toward the efficient processing of complex queries is the *optimization* of all primitive operations and the compatibility of these optimizations.

In this paper, we consider the join operation as a *paradigm* of basic complex operations. Although many join algorithms have been proposed [3, 4, 5, 10, 21], they are generally designed independently of the effect on other operations in the global query optimization. With new database applications, the effect can be important. For example, a transitive closure operator, which we expect to be a

Author's address: Microelectronics and Computer Technology Corporation, 3500 West Balcones Center Drive, Austin, TX 78759.

primitive operator for knowledge base support, will use joins and unions repeatedly. Thus, the join operation must be optimized for repetitive use. Also the combination of join and union must be efficient.

Join algorithms have been studied in depth in the context of business applications. With no indices, two basic algorithms based on sorting [4] and hashing [5, 10] avoid the prohibitive cost of the nested loop method. Parallel versions of these algorithms can significantly improve join performance [4, 21].

Indices on join attributes can also be employed [4, 11]. Clustered indices on the join attribute (tuples are clustered in the relation according to the join attribute) yield excellent performance, but at most one clustered index can be defined on a relation. Inverted indices are efficient only for very selective joins (producing a small result). An important result of [4] is that the sort merge join algorithm is almost always better than the join algorithm using inverted indices on join attributes. Another recent and important result shows that the availability of large main memories in database systems makes hash-based algorithms much more efficient than the sort merge join algorithm [10]. Consequently, we deduce that hash-based algorithms should be more efficient than algorithms using inverted indices.

A different concept useful for joins is the *link* [13, 15, 20]. The concept is the same as the Conference on Data Systems Languages (CODASYL) set notion. Tsichritzis [20] uses links in a model that allows the coexistence of hierarchical, network, and relational models. Similarly, [15] employs the so-called indirect join in the Pascal R environment. These two papers do not specify the data structure and join algorithms. Haerder [13] also uses the concept of link to optimize relational joins. He proposes an implementation very similar to that of CODASYL systems. Links are implemented by chaining tuples using tuples identifiers (instead of pointers as in CODASYL) mixed with the data. He also argues for a generalized access path structure that combines links on several relations and indices (called images). Although a unique access path for different access patterns can lead to high performance under particular workloads, we believe that in a highly concurrent environment, it is likely to become a highly contended resource.

Another way of optimizing joins is to prejoin all relations by storing each domain separately where each domain value associates the list of identifiers of matching tuples [17]. This storage model favors joins but at the expense of other operations.

In this paper, we propose a simple solution for optimizing joins in the context of complex queries. This solution is based on two design principles.

(1) An algorithm's performance is proportional to the amount of useful information. We thus strive to make the size of useful information for query processing as small as possible.

(2) Future computers will have a parallel processing capability and large amounts of random-access memory (RAM). We intend to take advantage of the availability of such hardware when designing our algorithms.

The application of these design principles to the problem of complex query evaluation led us to the notion of *join indices*. A join index is a particular

implementation of the concept of link (mentioned above). It is a prejoined relation, usually much smaller than the joined relation, and is stored separately from the operand relations. This will be the main reason for performance improvement. It is intended to improve the performance of complex operations by using a small and simple data structure. Furthermore, for most of the joins, updates of join indices incur very little overhead. Some properties of a join index are (i) its efficient use of memory and adaptiveness to parallel execution, (ii) its compatibility with other operations (including select, union), (iii) its support for abstract data type join predicates, (iv) its support for multirelation clustering, and (v) its use in representing directed graphs and in evaluating recursive queries.

The contribution of this paper is that it proposes an efficient implementation of join indices, gives algorithms for joins and updates, shows the compatibility of join indices with inverted indices and their value in answering complex queries, and, finally, shows the superior performance of the proposed implementation through a detailed analysis. We believe that join indices represent a carefully designed accelerator that can effectively use large RAM to increase performance and constitute an interesting alternative to hashing. Also, and perhaps more important, join indices optimize recursive as well as traditional complex queries.

The remainder of this paper is organized as follows. In Section 2, we formally define join indices and their implementation. In Section 3, we propose algorithms for supporting relational queries with join indices and managing these indices during updates. In Section 4, we analyze the performance of the join algorithm using join indices. We compare our algorithm with the hybrid-hash join algorithm [10], since the latter is efficient and makes effective use of the available main memory, like our algorithm. In Section 5, we summarize some interesting properties of join indices. Our conclusions and extensions of our ideas are given in Section 6.

2. CONCEPT OF JOIN INDEX

2.1 Definitions

Let R and S be two relations, not necessarily distinct. We consider the join of R and S on attributes A from R and B from S, giving a result relation T. Intuitively, a join index is an abstraction of the join of the two relations. We assume each tuple of a relation is uniquely identified by a *surrogate* [6, 14]. A surrogate is a system generated identifier that never changes. The surrogate of R (respectively, S) is noted r (respectively, s). The surrogate of tuple i of R is noted r_i and the surrogate of tuple j of S is noted s_j. *Tuple* r_i refers to the tuple having r_i as surrogate. More formally, the join index on R and S representing T is the set

$$\text{JI} = \{(r_i, s_j) \mid f(\text{tuple } r_i.A, \text{ tuple } s_j.B) \text{ is true}\},$$

where f is a boolean function that defines the join predicate. The join predicate can be arbitrary, and thus very general.

Thus, a join index is a relation of arity two. It is created by joining the relations R and S and projecting the result on (r, s).

CUSTOMER

csur	cname	city	age	job
1	Smith	Boston	21	clerk
2	Collins	Austin	26	secretary
3	Ross	Austin	36	manager
4	Jones	Paris	29	engineer

CP

cpsur	cname	pname	qty	date
2	Smith	jeans	2	052585
3	Smith	shirt	4	052585
1	Ross	jacket	3	072386

JI

csur	cpsur
1	2
1	3
3	1

Fig. 1. Join index for relations CUSTOMER and CP on attribute cname.

In the rest of this paper, we will often base our examples on the following database:

CUSTOMER (cname, city, age, job)
CP (cname, pname, qty, date)
PART (pname, dept, price, age)

(The age of a part is the number of years since it first appeared on the market.) Figure 1 gives an example of a join index summarizing the equi-join of the relations CUSTOMER and CP on attribute cname.

2.2 Implementation of Join Indices

A join index is a binary relation. It only contains pairs of surrogates which makes it small. However, for generality, we assume that it does not always fit in RAM. Therefore, a join index must be *clustered.* Since we may need fast access to JI tuples via either r values or s values depending on whether there are selects on relations R or S, a JI should be clustered on (r, s). A simple and uniform solution is to maintain *two* copies of the JI, one clustered on r and the other clustered on s. Each copy is implemented by a B^+-tree an efficient variation of the versatile B-tree [1, 7]. Simplicity and uniformity will lead to increased performance. The JI clustered on r (respectively on s) makes join accesses from R to S (respectively from S to R) efficient. Note that for limited access patterns, a single copy is sufficient (e.g., when one *always* goes from R to S). Figure 2 illustrates the implementation of the join index and of the relations of Figure 1, where the copy clustered on surrogate csur (respectively cpsur) is named JI_{csur} (respectively JI_{cpsur}). Figure 3 shows the use of the join index on relations R (r, A, B) and S (s, C, D) with join predicate $(R.A = S.D)$ to process a query for which the qualification is $(R.B =$ "b" and $R.A = S.D)$. For each tuple of R satisfying the selection predicate $(R.B =$ "b"$)$, its surrogate r_i permits accessing the join index

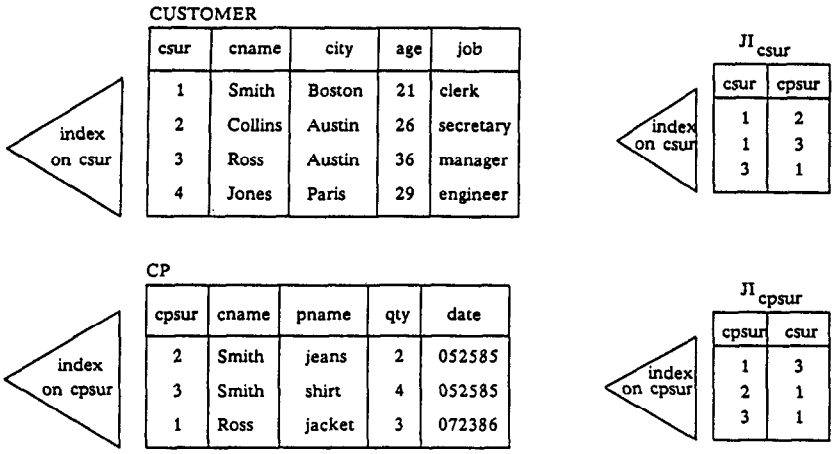

Fig. 2. Implementation of the join index on CUSTOMER and CP.

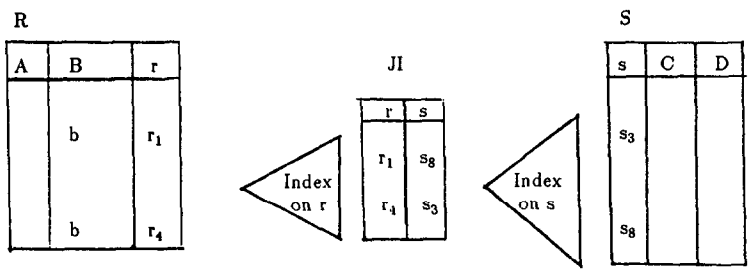

Fig. 3. Example of use of the join index clustered on r.

clustered on r. The surrogate s_j associated with r_i is in turn used to access the matching tuples of S through the index on s.

The way in which joined relations are physically clustered will generally have an impact on join performance. Surrogates contained in a join index are used for retrieving attribute values in physical relations. Therefore, a data structure that associates surrogates with page addresses is necessary. It can be implemented by either a clustered index or an inverted index (often called secondary index). In the first case, tuples having adjacent surrogate values will be in the same pages. Since the join index is clustered on surrogates, clustered access to the physical relations makes joins efficient. In the second case, tuples having surrogate values that are close will seldom be physically close. Thus, random access to tuples is necessary, which makes joins of many tuples less efficient. Effects of a clustered versus inverted index will be carefully analyzed in this paper.

A join index, like any other index, is an acceleration mechanism and should be used only for the most important joins. The efficiency of a join index will be proportional to its size. A tuple in a join index is small. For example, we consider an *impure generation* of surrogates in which a surrogate is unique within a relation and not in the whole database. Impure surrogates are sufficient to insure uniqueness in a relational database [9]. In the remainder of this paper we will

assume that the size of an impure surrogate is four bytes, which allows more than four billion tuples per relation.

The size of the join index depends on the join selectivity factor. The join selectivity factor, noted JS, is defined as follows, where $\| X \|$ indicates the number of tuples in relation X.

$$\mathrm{JS} = \frac{\| R \bowtie S \|}{\| R \| * \| S \|}.$$

The number of tuples in a join index is $\| R \bowtie S \|$. If the join has good selectivity (JS is low), the join index is small. This is a frequent case in existing databases (e.g., join on foreign key). However, a join of poor selectivity, which can be close to the Cartesian product, can make the index quite large. In this case, we claim that no good optimization is possible, and a simple nested loop join algorithm is sufficient. The effect of join selectivity is analyzed in our performance evaluation.

3. ALGORITHMS FOR RELATIONAL QUERIES

In this section, we propose algorithms that use and manage join indices. The specification of the algorithms is given independently of a machine model, although particular architectures, like parallel architectures, could improve their performance. Our only assumption is that a large amount of main memory is allocated to the operation. We denote the number of pages of main memory allocated as $| M |$. The following algorithms take advantage of the available memory space. We first present solutions for updating the join indices. We next present algorithms for semijoin and join operations using join indices. Finally, we point out the impact of join indices on query processing.

3.1 Update

Join indices, like other accelerators, incur an update overhead. A join index must reflect updates to the base relations. We limit ourselves to a JI with an equi-join predicate. Updates with more general predicates are discussed in Section 5.3. We only consider delete and insert operations. We assume that modify is done by delete followed by insert, which is a nice way of handling reliable updates, assuming a workspace model of updates.

The deletion algorithm is obvious. The surrogates of deleted tuples are retrieved when deletion is performed and are removed from the join index. Since one copy of the join index is clustered on the surrogates of the tuples deleted, this operation is efficient. For example, let us consider the following deletion in relation CP (given in Figure 1) expressed in SQL-like language:

```
delete CP where CP.cname = "Smith"
```

When performing the deletion, the tuples with cname Smith must be accessed either using an inverted index on cname if it exists or by the scanning relation CP. Thus, the set of cpsur {2, 3} would result from the deletion and be used to update the join index on the basis of the copy clustered on cpsur. In the process of deleting tuples from the JI copy clustered on cpsur, we can obtain the csur values and thus efficiently access and update the JI copy clustered on csur. The cost of deleting a tuple in a JI copy, assuming a two level B^+-tree, is an average

of three IOs (two page reads and one page write). If two copies of the join index are maintained, the average cost becomes six IOs.

We now consider updates of the join index when insertions into base relations occur. We distinguish between joins on foreign keys versus all other joins. Joins over foreign keys are the most frequent (probably more than 90 percent of the cases). In this case, the cost of updating the join index can be shared largely with the cost of performing referential integrity checks. This update optimization is done just before actually committing the updates when referential integrity checking must be applied [19]. For example, let us consider the relations CUSTOMER and CP where CP.cname is a foreign key. The insertion of a new customer does not generate any update of the join index, because referential integrity guarantees that there does not exist a corresponding CP tuple. The insertion of a new tuple in CP requires a referential integrity check, that is, that a corresponding cname actually exists. Thus, the tuple of CUSTOMER is accessed anyway, and its surrogate can be used for updating the join index. The overhead incurred is only the cost of reading and writing the join index. The cost of inserting a tuple in a JI copy is thus the same as the cost of deletion.

Joins not on a foreign key are much less frequently executed. However, in this case, updating the join index can be more costly. For example, if we want to propagate the insertion of a tuple r_i in relation R of join value A onto the join index, we need to select all tuples s_j in relation S, such that tuple $s_j.B = A$. This operation is efficient if S has a clustered index or inverted index on attribute B. Such indices are likely to exist if the join attribute is used for selections or if updates of a few tuples are frequent. The cost of inserting a tuple in a JI copy, assuming a three-level inverted index on a join attribute, is six IOs (three page reads of the inverted index to build the join index tuple and three IOs to update the JI copy). Inserting the tuple in the other join index copy is cheaper because the join index tuple has been previously built. Thus, we need just update the join index copy that incurs three IOs. If we assume the size of the two JI copies to be roughly equal to the size on an index of a join attribute (surrogates are much smaller than key values), this discussion indicates that for joins not on foreign keys, updates of join indices increase the update cost of the joined relations by a factor of approximately two.

If there is no index on the join attribute, the whole relation S must be scanned. If many tuple updates occur, the overhead can be significant. We believe that the use of a JI in this case is not worthwhile because the update overhead is too high. Join indices would only be useful in this case if the updates are done in a batch mode. Then, a simple solution can reduce the overhead by using a technique similar to view updates [18]. The update of a join index can be deferred until the next join is required. The relation updates are kept in a file containing the surrogates of updated tuples together with the update type. Thus, when the join is called, a semijoin of the joining relation with the updated relation is used to update the join indices.

3.2 Semijoin

The algorithm for semijoin, a very frequent operation, is straightforward with a join index. Consider the semijoin of R by S, denoted $R \ltimes S$. The attribute r of

the join index JI indicates which tuples of R are useful. Thus, relation S does not need to be accessed. For example, with the following semijoin query, where C stands for CUSTOMER:

select C.cname, C.age where C.cname = CP.cname

The join index in Figure 1 provides the set of surrogates {1, 3} that answer the query.

The access to tuples of R is based on the access path defined on its surrogate, that is, clustered index or inverted index. If R has a clustered index on the surrogate, using a copy of the JI clustered on r minimizes the number of IOs required to access R.

If R has an inverted index on the surrogate, additional accesses to the index occur. The inverted index on the surrogate is a set of (sur, page#) tuples sorted on sur. When accessing the index based on surrogates, the set of useful (sur, page#) is not ordered by page# and the same page in R can be read several times. To avoid reading the same page more than once, the set of pages to access is sorted on page# before reading relevant pages of R.

3.3 Join

The join algorithm accesses matching tuples of R and S using the join index in order to produce a result relation. For example, the join query

select C.cname, C.age, CP.pname where C.cname = CP.cname

will be processed by accessing a Customer tuple and a CP tuple and by producing a result tuple for each tuple in the join index. For instance, the first tuple of the join index represents the tuple ⟨Smith, 21, jeans⟩.

In this section, we concentrate on the join algorithm itself. However, as we will see in Section 3.4, since it reads the join index sequentially, this algorithm is applicable for joins preceded by selection. In this case, the join index used by the join algorithm is first reduced by the selection.

We suppose that the number of distinct r values in the join index is greater than the number of distinct s values. We use the join index copy clustered on r. The reason will be given in the analysis. We call R the *external relation* and S the *internal relation*.

For the sake of clarity, we first present a naive version of the join algorithm using a JI that we will subsequently improve.

```
for i := 1 to | JI | do {assume | JI | pages in JI}
  begin
    read page i of JI into JI_i
    {assume k distinct surrogates from R and l surrogates from S in this page}
    read k tuples of R into R_i
    read l tuples of S into S_i
    join R_i and S_i
  end
```

This algorithm has several important shortcomings. First, the memory of | M | page frames is poorly utilized. It is likely that JI_i, R_i, and S_i require much less than | M | pages. The rest of memory should be used for optimization. Since we use the join index clustered on r, the access to R is efficient. However, the access

to S can be inefficient. The tuples of S must be randomly accessed via the index with a worst case of one IO per tuple. Furthermore we can have repeated tuple accesses to S since a page j can contain surrogates of tuples that were also in a page i where $i < j$.

We now propose an improved version of the algorithm that avoids the preceding drawbacks. The improvement is essentially based on a memory management strategy adapted to the operation. The main improvements are

(1) read as much of the JI and $R \ltimes$ JI into memory as possible,
(2) optimize the accesses to S by sorting the list of surrogates s of the subset of JI in memory, which leads to clustered accesses to S and minimizes repetitive accesses to S.

The algorithm sequentially reads the join index JI page by page, composes $R \ltimes$ JI (equal to $R \ltimes S$), and joins it with $S \ltimes$ JI (equal to $S \ltimes R$) obtained page by page. If $R \ltimes$ JI does not fit entirely in memory, the operation is divided into several passes. At each pass, a subset of $R \ltimes$ JI that fits in memory is composed, and for each (r_i, s_j) in the join index such that tuple r_i is in memory, tuple s_j is accessed and joined with r_i. At each pass k, the next subset of pages of the join index, noted JI_k, is processed.

The allocation of $|\mathrm{M}|$ page frames of main memory between operand and result relations and the join index is an important component of the algorithm. Relations R and S and the result relation T are each allocated one page for input and output data. When a page of T is full, it is written in cache memory (which can lead to an IO operation). $S \ltimes$ JI will be read page by page and thus requires one page frame. We also reserve some extra working space for internal optimizations that we will define later on. After this static allocation of memory, we denote by $|\mathrm{M}'|$ the number of remaining pages that are dynamically allocated between the join index and $R \ltimes$ JI. They are allocated in such a way that we have in memory as many tuples of R as possible with their corresponding subset JI_k of the join index. Therefore for each tuple of R in the buffer, we know which tuples of S match with it. Assuming the subset JI_k of JI in memory, we denote as R_k the set:

$$R_k = \{\text{tuple of surrogate } r_i \in R \mid r_i \in \mathrm{JI}_k\}.$$

At each pass, JI_k and R_k are such that

$$|\mathrm{JI}_k| + |R_k| \leq |\mathrm{M}'|,$$

where $|X|$ denotes the number of pages of X. The relationship between JI_k and R_k depends on the semijoin selectivity factor for R_k, which can be different for some $l \neq k$. For example, we can have $\|\mathrm{JI}_k\| \neq \|\mathrm{JI}_{k-1}\|$.

The optimized join algorithm is summarized below in the procedure JOINJI (Figure 4). The algorithm handles the general case in which the semijoin of R by a page of the join index does not fit in memory.

At each pass of the algorithm, steps (1) through (4) are executed. Step (1) reads the subsets JI_k of JI and R_k of $R \ltimes$ JI in memory. Step (2) sorts JI_k on s. The goal of step (2) is to improve the performance of the semijoin of S by JI_k by reducing the number of accesses to S. We use the working space for performing

```
JOINJI (R, S: operand relation; T: result relation; JI:join index;
        | M' | :number of memory pages)

begin
  k := 0
  read first page of JI into JIpage
  m := | M' | - 1                    {current number of pages for JI_k and R_k}
(0) while not "end JI" do
      begin
        k := k + 1                              {start a new press}
    (1) while m > 0 and not "end JI" do         {produce JI_k and R_k}
          begin
            while not "new tuples in R ⋉ JIpage" and m ≠ 0 do
              begin
                perform next R ⋉ JIpage until filling one page
                m := m - 1
              end
            if m ≠ 0 then
              begin
                read next page of JI into JIpage
                m := m - 1
              end
        end {step (1)}
    (2) sort JI_k on s
    (3) while not "new tuples in S ⋉ JI_k" do
          begin
            read next page of S containing tuples of S ⋉ JI_k into Spage
            join Spage with R_k
          end {step (3)}
    (4) if "JIpage entirely processed" then
          begin
            m := | M' | - 1
            read next page of JI into JIpage
          end
        else m := | M' | - 1
      end {step (0)}
    end {JOINJI}
```

Fig. 4. Join algorithm using a join index.

the sort internally. The access to S is based on an inverted or clustered index on s. Thus, the access to the index is in order and minimal. Step (3) performs the join of relation S with R_k. Step 4 handles the case in which the last page read from the JI in JIpage has not been entirely processed; that is, there remain tuples in JIpage whose corresponding tuples in R did not fit in RAM.

As presented, the algorithm minimizes the number of IOs. In order to minimize the central processing unit (CPU) time, we need to perform the internal join rapidly. This can be done by adding to each (r_i, s_j) of JI_k the address of tuple r_i in memory. Therefore, for each tuple s_j accessed, the matching tuple r_i is directly found. In the proposed algorithm, the same page of S may be reread at different passes. However, in most cases, the useful subset of the JI and $R \ltimes$ JI fit in memory, making the scheduling of page accesses optimal (only useful pages are read, and only once).

Join indices can be derived from classical indices on join attributes. Most relational systems use indices on join attributes to speed up joins [3]. We now give a brief comparison of the performance of a join using join indices and a join using indices on join attributes as proposed in [4]. In each method, the indices must be scanned entirely; thus the performance difference is related to the difference between the index sizes. An index on a join attribute is supposed to be of the form {(att_value, surrogate)}. It has an entry for each different surrogate value (i.e., for each tuple of the relation), where each entry contains an attribute value with its surrogate. Assuming that the cardinalities of R and S are respectively $\| R \|$ and $\| S \|$ and that each tuple of S matches with one tuple of R (e.g., a hierarchical case like Customer → CP), then the amount of extra data accessed by the method based on join attribute indices is roughly $(\| R \| + \| S \|) *$ att_size, which can be quite large. For joins of poorer selectivity, the difference would be even bigger. Also, a join index can be easily compacted using run length compression, making it even smaller. Thus, the number of IOs necessary for the method based on join attribute indices is generally much higher.

3.4 Relational Queries

We now illustrate the use of join indices and other indices for performing relational queries. For simplicity, we consider queries involving selections, joins, and projections. We assume that the compiled relational query has been optimized and thus exhibits the best possible decomposition in relational operations. The basic idea is to use indices as much as possible and to postpone access to relevant base data (generally much larger) to the very end.

We consider a first type of relational query involving a selection followed by a join. We suppose a join index exists for the join. The selection operation will produce a list of surrogates of tuples that satisfy the selection criteria. There are basically three possible access paths for selection: inverted index, clustered index, or sequential scan. An inverted index associates attribute values with surrogates. Therefore, accessing the inverted index for given attribute values produces the relevant surrogates. When using the two other access paths, the base data must be accessed. In this case, the result of the selection is an intermediate relation containing relevant tuples together with a list of their surrogates. The list of surrogates is semijoined with the join index to find the relevant surrogates of the other relation. Since as much of the join index as possible will be kept in RAM, the best way for performing the semijoin is to probe the join index for each surrogate value of the list. Remember that a join index is implemented by a B^+-tree and thus provides direct access based on surrogate. If the join index cannot fit in RAM, sorting the surrogate list first is an interesting alternative. Consider the following example:

```
select C.cname, CP.name, C.job
  where city = "Austin" and C.cname = CP.cname
```

Figure 5 illustrates the corresponding relational query tree and its operation tree when there is an inverted index on attribute City in relation C. If the access path were not an inverted index, the selection would have also produced an intermediate relation C′. C′ would be stored in a hashed table on the join attribute and used during the final join instead of C. The City_index provides

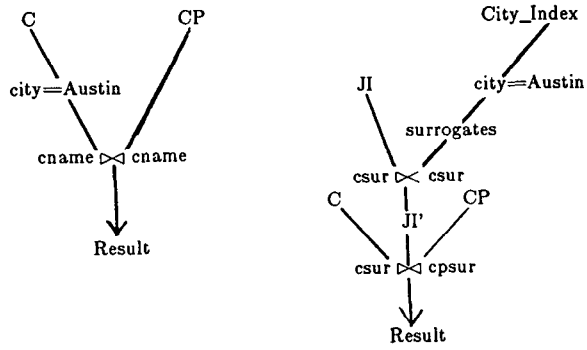

Fig. 5. A query tree and its operation tree.

the set of surrogates {2, 3}. Thus, the JI′ is {(3, 1)}. The final access to CUSTOMER with csur = 3 and to CP with cpsur = 1 enables the construction of the tuple ⟨Ross, jacket, manager⟩. The join index JI is thus accessed directly through its B-tree during the semijoin. JI′, the result of the semijoin, is stored in a sequential file, since it is accessed sequentially during the final join phase.

If the selection is very strong for this type of query, classical indexing can outperform join indices. With the classical approach of [4], the example query of Figure 5 is processed by first accessing the C tuples that satisfy the select predicate, and, for each tuple selected, the matching CP tuples are accessed through an index on the join attribute (cname). If the selection is very strong (e.g., one C tuple is selected), then the indirect access to the join index will incur one or two additional IOs.

We now consider relational queries with multiple joins possibly combined with selections. If every join can be processed using a join index, then all the join indices are first joined. They produce subsets of join indices that store the surrogates of relevant tuples. Then, the joins on base data are applied at the very end using join index subsets. Therefore, only useful base data are accessed.

If not every join can be processed using a join index, then we must combine joins using join indices with more classical join algorithms. Examples of classical joins in which the base data are accessed are the sort merge join algorithm [4] or some hash join algorithms [5, 10]. We consider the query whose optimized decomposition is $(R \bowtie S) \bowtie T$, where the second join involves an attribute of S and an attribute of T. We assume only one join index exists. Two cases can occur: There is JI(s, t) for $(S \bowtie T)$ or JI(r, s) for $(R \bowtie S)$. In the first case, the classical join precedes the join using the join index. Thus, the classical join produces a result relation U together with a set of surrogates $\{s\}$ of tuples in U. A tuple in U is of the form $\langle s, \text{att}_1, \text{att}_2, \ldots \rangle$. However, since U is a temporary relation, its tuples do not have their own surrogates. Therefore, s cannot be considered a unique surrogate of a tuple in U; that is, there can be several tuples in U having the same s value. The set of surrogates is then semijoined with JI(s, t), giving a subset of the join index JI′(s, t) in a sequential file. The join algorithm using the join index is then applied on T and U using JI′. In the second case, the join using the join index JI precedes the classical join. The join using

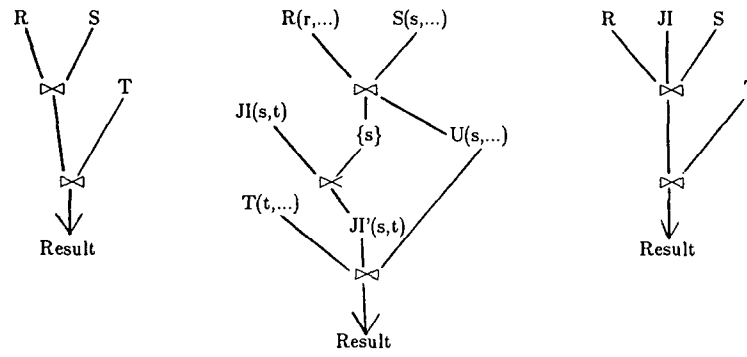

Fig. 6. Operation trees for different join indices.

the JI is applied giving a result relation U that can be used by the subsequent join $U \bowtie T$. Figure 6 shows the two different operation trees for the same query tree where classical joins and joins using join indices are combined.

4. PERFORMANCE ANALYSIS

In this section, we analyze the performance of the join algorithm presented in Section 3. We first define the analysis parameters and then derive the execution times of our join algorithm for different clusterings of the operand relations. Our algorithm is then compared with the hybrid-hash join algorithm [10]. The latter is one of the most efficient that we know of and, like our algorithm, makes use of all the available main memory. We choose it also because it is much more efficient than the sort merge join algorithm, which in turn is generally better than join algorithms using secondary indices on join attributes.

As in [4], we do not include the update overhead incurred by join indices for two reasons. The first reason is that we showed in Section 3.1 that, for the most frequent joins, the update overhead is very little. It should be acceptable and conditional upon a significant gain against other efficient algorithms. The second reason is that an analysis including a cost factor for updates in the join algorithm cost is not realistic because this factor depends on update/retrieval frequencies. Such an analysis would be too complex and is beyond the scope of this paper. Instead, we plan to measure the update cost of join indices through implementation. Initial results are given in [23].

4.1 Analysis Parameters

4.1.1 *Database Dependent Parameters.* The following notation will be used to evaluate algorithms:

$\|R\|$, $\|S\|$	Number of pages in relations R and S, respectively
$\|\|R\|\|$, $\|\|S\|\|$	Number of tuples in relations R and S, respectively
JS	Join selectivity factor defined by JS $= \|\|R \bowtie S\|\| / \|\|R\|\| * \|\|S\|\|$
SR	Semijoin of (R by S) selectivity factor defined by SR $= \|\|R \ltimes S\|\| / \|\|R\|\|$

SS	Semijoin of (S by R) selectivity factor defined by SS = $\| S \ltimes R \| / \| S \|$
$\| JI \|$	Number of pages in the join index
$\| JI \|$	Number of tuples in the join index given by $\| JI \| = (\| R \| * \| S \|) * JS = \| R \bowtie S \|$
T_R, T_S, T_{JI}	Size (in bytes) of a tuple of R, S, and JI, respectively

Join selectivity is related to the semijoin selectivities, since each tuple of the join is constructed from two tuples, each belonging to a semijoin by the following inequality,

$$\max\left(\frac{SR}{\| S \|}, \frac{SS}{\| R \|}\right) \leq JS \leq SS * SR.$$

The lower bound is attained when each tuple in the larger semijoin is matched exactly once. The upper bound is attained when the two semijoined relations produce the join using a Cartesian product.

4.1.2 *System Dependent Parameters.* We use the following parameters to describe the capabilities of the database system:

$\| M \|$	Main memory size (in number of pages) allocated to the operation
F	Universal fudge factor for hashing
PO	Average page occupancy factor
P	Page size (in number of bytes)
FO	Fan-out of a node in a B-tree
ssur	Surrogate size (in number of bytes)

4.1.3 *System Performance Dependent Parameters.* The following performance parameters, dependent on computer capabilities, are used in measuring the IO and CPU times of the algorithms:

IO	Time to perform an IO operation
comp	Time to compare two keys
hash	Time to hash a key
move	Time to move a tuple

4.2 Analysis of Join Algorithms

In this section, we analyze the performance of the join algorithm using join indices when relations are stored separately. We also summarize the performance of the hybrid-hash join algorithm. We make a number of simplifying assumptions. The basic assumption is that there is no overlap between IO and CPU. Since we will compare the algorithms on a conventional architecture, we believe that the effect of overlap between IO and CPU would be the same for both algorithms (this may not be true for parallel architectures). We suppose that the operand relations are stored on magnetic disks. Also, we ignore the cost for writing the result relation, since it is roughly the same for all algorithms. We assume that the root page of a B^+-tree index is always kept in memory. The small size of a surrogate implies a high fan-out of B-tree nodes. Assuming that a page# has the same size as a surrogate and considering that the page occupation factor is high

(we use the classical factor 70 percent), then there are about 400 (sur, page#) tuples per 4K bytes page. According to the values that we will use for relation sizes, we assume that the B-tree of an index based on a surrogate will always have two levels.

4.2.1 *Join Using Join Indices.* In this section, we evaluate the execution time of the join algorithm presented in Section 3. The performance of this algorithm is greatly influenced by the join and semijoin selectivity factors. To simplify our evaluation, we assume that the participation in the join is uniformly distributed among matching tuples; that is, there is no tuple that participates in the join more times than any other tuple. This implies that each subset JI_k of the join index processed in a pass is of same size. Thus, we have $\| JI_1 \| = \| JI_2 \| = \cdots \| JI_k \| \cdots$. Relaxing this assumption implies that the proportion of JI and $R \ltimes$ JI that fit in memory would vary from one pass to another. Therefore, the number of tuples of $S \ltimes$ JI accessed would vary over passes. In several experiments we found that the performance becomes significantly better under the nonuniformity assumption only when the variations in the number of tuples accessed from $R \ltimes$ JI are high. Thus, our assumption is pessimistic. Also, we do not take into account the effect of run-length compression of the join index, although it would make the algorithm more efficient. Finally, we note that our assumptions favor the hybrid-hash join algorithm.

We divided the analysis into four steps corresponding to phases in the algorithm:

(1) read the join index,
(2) perform $R \ltimes$ JI,
(3) internally sort the join index JI_k on s,
(4) perform $S \ltimes$ JI.

4.2.1.1 *Number of Passes in the Algorithm.* The join algorithm allocates three memory pages for operand and result relations and one page for reading $S \ltimes R$. The amount $| M' |$ memory dynamically allocated for $R \ltimes S$ and the join index is thus

$$| M' | = | M | - 4.$$

If $R \ltimes S$ and the join index do not fit in $| M' |$ pages of memory, the four steps of the algorithm must be repeated during several passes. Assuming a uniform distribution of the semijoin selectivity among tuples of R, the number of passes is simply the sum of the sizes of $R \ltimes S$ and the join index divided by $| M' |$. The size of $R \ltimes S$ is

$$\| R \| * SR * T_R/P.$$

The size of JI is $| JI |$. We also provide for some amount of working memory space for internal optimizations (sorting and direct access to tuple by a surrogate). This amount is proportional to $| JI | / | M' |$ and is measured by a factor denoted by PR. The size of the working space is thus PR $*$ $| JI | / | M' |$. We will need as much space as the join index in memory to internally sort it. Thus, PR = 1. Furthermore, for each surrogate r_i in memory, we add the internal address (= size of surrogate) of tuple i. Thus, PR = PR + 0.5 = 1.5. The number of

passes, denoted by N, is

$$N = \max\left(\frac{|\,\mathrm{JI}\,| + |\,\mathrm{JI}\,| * \mathrm{PR} + (\|\,\mathrm{R}\,\| * \mathrm{SR} * \mathrm{T}_R)/\mathrm{P}}{|\,\mathrm{M}'\,|}\right).$$

4.2.1.2 *Basic Formulas.* The following basic formulas will be commonly used in evaluating the different steps of the algorithm.

(1) *Number of tuples per page.* The number of tuples per page of relation X, where X will be R, S, or JI is

$$n_X = \frac{\mathrm{P} * \mathrm{PO}}{\mathrm{T}_X}.$$

(2) *Number of page accesses.* For accessing k records randomly distributed in a file of n records stored in m pages, a formula for the expected optimal number of page accesses is given in [24]:

$$Y(k, m, n) = m^*\left[1 - \prod_{i=1}^{k} \frac{n - (n/m) - i + 1}{n - i + 1}\right].$$

This formula assumes that the scheduling of page accesses is optimal; that is, the same page is not accessed more than once.

(3) *IO time for accessing k tuples.* We determine the IO time incurred in accessing k tuples of relation X based on a surrogate, where X will be R or S. The access to relation X is based either on a clustered index or an inverted index. In both cases, the k tuples are supposed to be randomly distributed because the relation is not clustered on the join attribute (tuples with the same join attribute value are not physically close). Thus, the number of page accesses to relation X is given by $K = Y(k, m, n)$, where $m = |\,X\,|$ and $n = \|\,X\,\|$.

In the first case, relation X has a clustered index, denoted by CX, on a surrogate. We assume that the index is maintained as a two-level B^+-tree whose leaves associate a surrogate with a page# in X. We also assume that the root resides in cache memory and thus ignore the cost of accessing it. Thus, we need just evaluate the number of accesses, denoted by $\mathrm{nba}_{\mathrm{CX}}$, to the last level of the clustered index. $\mathrm{nba}_{\mathrm{CX}}$ is the number of page accesses for locating, for example, x records randomly distributed in the index. One tuple of the index must be accessed for each page accessed in relation X. Since there are K pages accessed in relation X, we have $x = K$. The total number of pages in the second level of the index depends on the node fan-out and is m/FO. Since there is one index tuple per page of relation X, the total number of tuples in the last level of the index is m. Therefore, the number of page accesses to the index is

$$\mathrm{nba}_{\mathrm{CX}} = Y\left(K, \frac{m}{\mathrm{FO}}, m\right).$$

Finally, the IO time for accessing k tuples in relation X using a clustered index is

$$\mathrm{IO}_{\mathrm{ci}}(k, m, n, X) = (K + \mathrm{nba}_{\mathrm{CX}}) * \mathrm{IO}.$$

In the second case, the access to pages of relation X is based on an inverted index on a surrogate. An inverted index is larger than a clustered index. We assume that the inverted index is a three-level B^+-tree whose leaves associate a surrogate with a page# in X. Let us first evaluate the number of accesses to the index on X, noted IX, for retrieving k records. Assuming that the size of a page# equals the size of a surrogate, the number of pages of the last level of the index is

$$|\,\mathrm{IX}\,| = \frac{\|X\| * 2 * \mathrm{ssur}}{\mathrm{P} * \mathrm{PO}}.$$

For each tuple of X, one tuple of the inverted index must be accessed. The number of accesses to k records randomly distributed in the last level of the index, where $m' = |\,\mathrm{IX}\,|$ is

$$\mathrm{nba}_{\mathrm{IX}} = Y(k, m', n).$$

The number of accesses, noted $\mathrm{nba}'_{\mathrm{IX}}$, to the second level of the index can be evaluated similarly to $\mathrm{nba}_{\mathrm{CX}}$ above. Therefore, we have

$$\mathrm{nba}'_{\mathrm{IX}} = Y\left(\mathrm{nba}_{\mathrm{IX}}, \frac{m'}{\mathrm{FO}}, m'\right).$$

The IO time using an inverted index is thus

$$\mathrm{IO}_{\mathrm{ii}}(k, m, n, X) = (K + \mathrm{nba}_{\mathrm{IX}} + \mathrm{nba}'_{\mathrm{IX}}) * \mathrm{IO}.$$

(4) *CPU time for semijoin.* We now derive the CPU time incurred in performing the semijoin of relation X by the join index, where X means R or S. The semijoin is done in N passes where, at each pass, k tuples of X must be accessed. The number of page accesses to relation X is given by $K = Y(k, m, n)$. Each tuple of each page of X is compared with the JI, which requires $N * K * n_X *$ comp. The semijoin selectivity factor is noted SX and will be SR or SS. The $\|X\| *$ SX tuples of the semijoin must be moved in memory from their input page to the buffer allocated to the semijoin, which needs $\|X\| *$ SX $*$ move. Thus, the CPU time for the semijoin of relation X by N passes of K pages is

$$\mathrm{CPU}_{\mathrm{sj}}(k, m, n, X) = N * K * n_X * \mathrm{comp} + \|X\| * \mathrm{SX} * \mathrm{move}.$$

(5) *Internal sort of n tuples.* Sorting internally n tuples [16] requires a time

$$\mathrm{CPU}_{\mathrm{st}}(n) = n * \log_2 n * \mathrm{comp} + n * \mathrm{move}.$$

4.2.1.3 *Analysis of Step* 1. Step 1 consists of reading the join index one page at a time. The execution time of step 1 is

$$t_1 = |\,\mathrm{JI}\,| * \mathrm{IO}.$$

4.2.1.4 *Analysis of Step* 2. Step 2 consists of accessing relation R on the basis of surrogate values found in the join index and producing $R \ltimes \mathrm{JI}$. The access to relation R depends on whether R has a clustered index or an inverted index on a surrogate.

(1) *R is clustered on a surrogate.* At each pass, the number of tuples of the semijoin to retrieve in R is

$$k = \frac{\| R \| * \mathrm{SR}}{N}.$$

The tuples are randomly distributed because the relation has a clustered index on a surrogate (not on a join attribute). Since the JI is clustered on a surrogate r, and relation R is also clustered on r, then at each pass i, the subset R_i of R corresponding to the subset JI_i of the join index in memory is accessed. Therefore, we have

$$m = \frac{| R |}{N} \quad \text{and} \quad n = \frac{\| R \|}{N}.$$

For N passes, the IO time of step 2 incurred with a clustered relation is

$$t_{2\mathrm{cIO}} = N * \mathrm{IO}_{\mathrm{ci}}(k, m, n, R).$$

The CPU time consists of doing the semijoin of R by JI, which is

$$t_{2\mathrm{cCPU}} = \mathrm{CPU}_{\mathrm{sj}}(\mathrm{nba}_{2\mathrm{c}}, R).$$

The total time of step 2 when R is clustered on r is

$$t_{2\mathrm{c}} = t_{2\mathrm{cIO}} + t_{2\mathrm{cCPU}}.$$

(2) *R is indexed on a surrogate.* The join index is read by sets JI_k of pages, where for each JI_k, the subset R_k of $R \ltimes S$ is retrieved. Even in the case of nonuniform distribution of join selectivity among tuples of R, we are always able to determine JI_k such that R_k fits in memory, since the join index tells us which tuples of R to retrieve. At each pass, the number of tuples of the semijoin to retrieve in R is

$$k = \frac{\| R \| * \mathrm{SR}}{N}.$$

For each semijoin subset, the entire relation R must be accessed because the tuples are randomly spread over R (R is not clustered on a surrogate); thus

$$m = | R | \quad \text{and} \quad n = \| R \|.$$

The IO time to access R is thus

$$t_{2\mathrm{iIO}} = N * \mathrm{IO}_{ii}(k, m, n, R).$$

We now evaluate $t_{2\mathrm{iCPU}}$. The main difference with the previous case is that the inverted index provides the relevant tuples (r, page#) sorted on r but not on page#. In order to optimally schedule the disk accesses to R, the set of couples (r, page#) is sorted on page#. The number of pages of JI_k is in the average $| \mathrm{JI} | / N$. Thus, the number of tuples in JI_k, n_{JIK}, is

$$n_{\mathrm{JIK}} = \frac{n_{\mathrm{JI}} * | \mathrm{JI} |}{N}.$$

The tuples (r, page#) obtained from the index for each JI_k must be sorted on page#, which requires for all passes

$$N * \mathrm{CPU}_{st}(n_{JIK}).$$

By adding the semijoin time, we obtain

$$t_{2iCPU} = N * \mathrm{CPU}_{st}(n_{JIK}) + \mathrm{CPU}_{sj}(k, m, n, R).$$

The total time of step 2 when R is indexed on r is

$$t_{2i} = t_{2iIO} + t_{2iCPU}.$$

4.2.1.5 *Analysis of Step* 3. Step 3 consists of internally sorting each subset JI_k on s. Assuming that all JI_k are of the same size (this is consistent with our basic assumption of uniform distribution of join selectivity among the tuples), we get

$$\| JI_k \| = \frac{\| JI \|}{N}.$$

For N passes, the time of step 3 is

$$t_3 = N * \mathrm{CPU}_{st}\left(\frac{\| JI \|}{N}\right).$$

4.2.1.6 *Analysis of Step* 4. At each pass of the join algorithm, step 4 accesses page by page the semijoin of S by JI for tuples of R in memory. The analysis is similar to that of step 2. The main difference is that the join index is not clustered on surrogate s. Therefore, for each subset of the semijoin (S by JI), we have to access the whole relation S. Therefore, we have

$$k = \frac{\| S \| * \mathrm{SS}}{N}, \qquad m = | S |, \qquad \text{and} \qquad n = \| S \|.$$

(1) *S is clustered on s.* The time of step 4 is simply the time for reading the semijoin of S by the JI using the clustered index, and the CPU time for performing the semijoin, which gives

$$t_{4c} = N * \mathrm{IO}_{ci}(k, m, n, S) + \mathrm{CPU}_{sj}(k, m, n, S).$$

(2) *S is indexed on s.* The IO time is incurred in reading the semijoin of S by the JI using the inverted index. The tuples (s, page#) obtained from the index must be sorted on page#. Thus, the CPU time is the time of sorting at each pass a set of $\| JI \|/N$ tuples (s, page#) plus the time for performing the semijoin, which leads to

$$t_{4i} = N * \mathrm{IO}_{ii}(k, m, n, S) + N * \mathrm{CPU}_{st}\left(\frac{\| JI \|}{N}\right) + \mathrm{CPU}_{sj}(k, m, n, S).$$

In conclusion, the time of the join algorithm using the join index is as follows, where | means "exclusive or":

$$t(\mathrm{JOINJI}) = t_1 + t_{2c} \mid t_{2i} + t_3 + t_{4c} \mid t_{4i}.$$

4.2.2 *Hybrid-Hash Join Algorithm.* The hybrid-hash join algorithm is proposed and analyzed in [10]. It makes use of all available memory ($| M |$). We briefly

review the algorithm and its performance. Relations R and S are sequentially read from disk and partitioned into sets on the basis of the same hash function applied to the join attributes. Let

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

The algorithm consists of $B + 1$ steps where R and S are partitioned into compatible sets $R_0, R_1, \ldots, R_B$, and $S_0, S_1, \ldots, S_B$. Furthermore R_0 has $|\mathrm{M}| - B$ pages and is processed at the same time that R and S are being partitioned. The join is then divided into B joins of $R_1 \bowtie S_1$, $R_2 \bowtie S_2$, $\ldots$, $R_B \bowtie S_B$. Let

$$|R_0| = \frac{|\mathrm{M}| - B}{\mathrm{F}} \quad \text{and} \quad q = \frac{|R_0|}{|R|}.$$

The time of the hybrid-hash join algorithm using our model is

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

4.3 Performance Comparisons

This section presents performance comparisons of the join algorithm using join indices (JOINJI) and the hybrid-hash join algorithm (JOINHH) using the previous cost formulas. We fix the values of the analysis parameters assuming a conventional system as in [10]. Other comparisons have been run with different parameter settings, and similar results have been found. The parameter values are set as follows:

F	Universal "fudge" factor for hashing is 1.2
PO	Average page occupancy factor is 0.7
P	Page size is 4000 bytes
FO	B-tree node fan-out is 400
ssur	Surrogate size is 4 bytes
IO	Time of an IO operation is 25 milliseconds
comp	Time to compare two keys is 3 microseconds
hash	Time to hash a key is 9 microseconds
move	Time to move a tuple is 20 microseconds

From the cost formulas previously derived, we see that the performance of JOINJI relative to JOINHH is strongly influenced by the join and semijoin selectivity factors and by the fact the operand relations are clustered or indexed on a surrogate. Relation sizes and memory size will also significantly affect both algorithms.

Figure 7 illustrates the performance ratio of JOINJI versus JOINHH for varying semijoin selectivity factors (with SS = SR). HH denotes the execution time of JOINHH, whereas JI denotes the execution time of JOINJI. The join

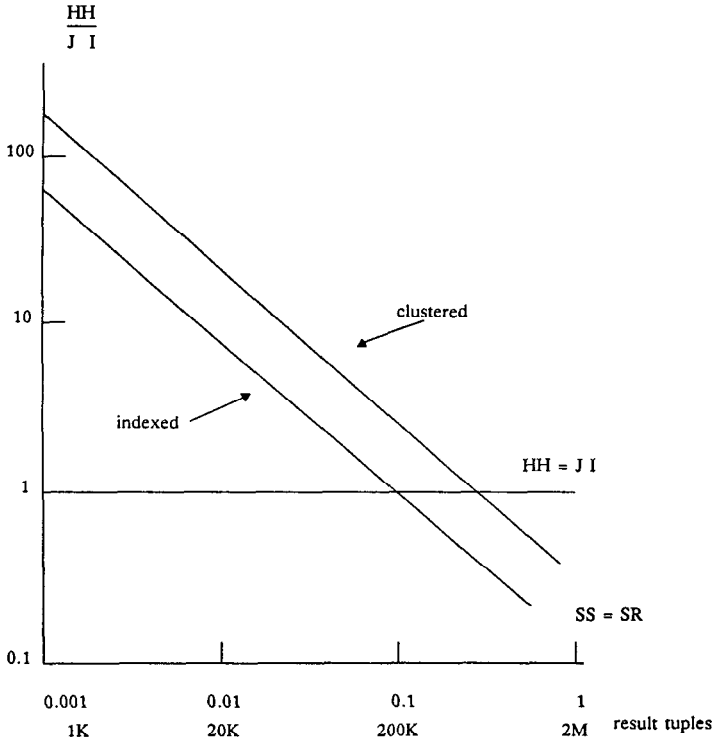

Fig. 7. Performance ratio of JOINJI versus JOINHH ($|R| = |S| = 10{,}000$, $\|R\| = \|S\| = 200{,}000$, $|M| = 1{,}000$, $JS = 10 * SS/\|R\|$).

selectivity factor is proportional to SS. It has been chosen in order to produce a result-joined relation of realistic size. For example, when SS = 0.1, the result relation is as big as an operand relation. The graph shows clearly the influence of join selectivity. The curve *clustered* (respectively *indexed*) means that the operand relations are clustered (respectively indexed) on a surrogate, which impacts on the performance of JOINJI. The difference between the curves remains constant on the graph, which means that the performance difference between JOINJI clustered and JOINJI indexed decreases significantly as SS increases. JOINHH outperforms JOINJI only in the presence of poor join selectivity, that is, when producing a very large result relation.

Figure 8 presents the execution times of JOINHH and JOINJI versus the semijoin selectivity factor. The graph illustrates in a different way the influence of join selectivity on JOINJI. The two curves of JOINJI correspond to the minimum and maximum join selectivity factors that can be derived from the semijoin selectivity factors. For the fixed values of parameters, all possible joins using join indices would show performance curves between these two extremes.

Figure 9 shows the effect of varying the semijoin selectivity factors on the performance of JOINJI for relations of equal size clustered on a surrogate. The X axis represents SS for the curve of SR and SR for the curve of SS. The graph tells us what we can intuitively guess. The relation that produces the biggest

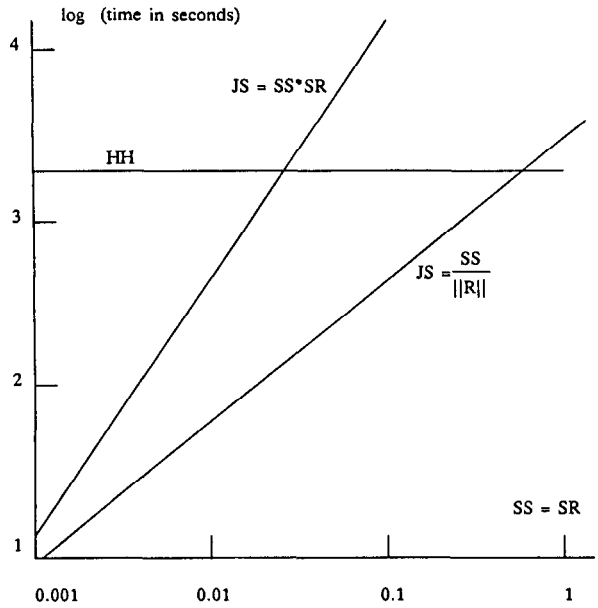

Fig. 8. Performance of JOINJI and JOINHH versus join selectivity ($|R| = |S| = 10{,}000$, $|M| = 1{,}000$); R and S are clustered on surrogate.

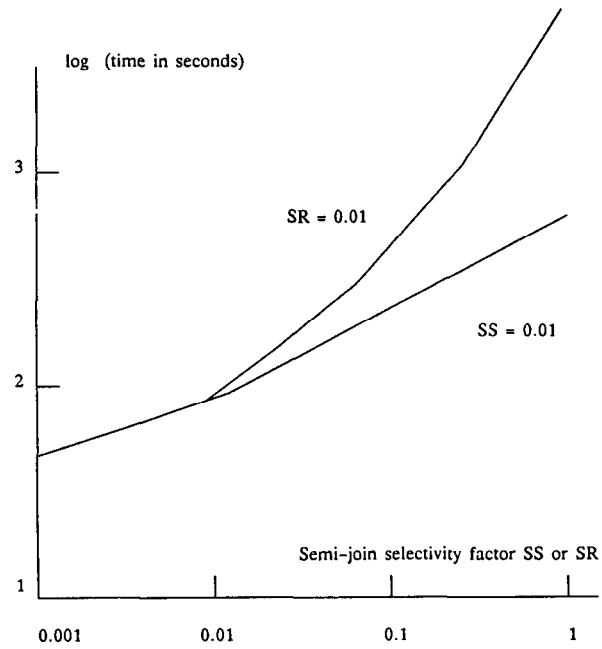

Fig. 9. Effect of varying the semijoin selectivity for JOINJI ($|R| = |S| = 10{,}000$, $|M| = 1{,}000$, $JS = 10 * \max(SS/\|R\|, (SR/\|S\|))$; R and S are clustered on surrogate.

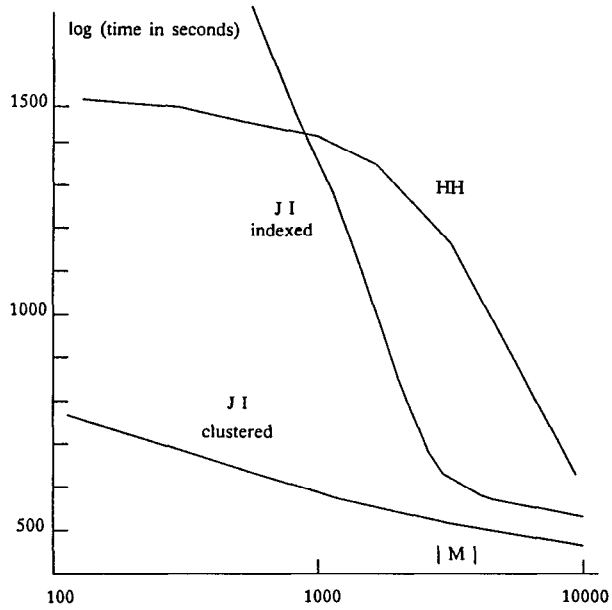

Fig. 10. Performance of JOINJI and JOINHH versus memory size ($|R| = |S| = 10{,}000$, SR = 0.5, SS = 0.1, 20,000 result tuples).

semijoin should be used as an external relation in the JOINJI algorithm. This minimizes the number of accesses to the internal relation.

Figure 10 illustrates the behavior of the algorithms versus memory size. Both algorithms take advantage of the available memory. When the operand relations are clustered on a surrogate, the performance of JOINJI is linear in $|M|$. This is not true when relations are indexed on a surrogate because the performance of JOINJI degrades significantly as the number of passes of the algorithm increases. The performance difference between the algorithms versus relation sizes (Figure 11) remains constant provided that the memory size is proportional to relation sizes.

In conclusion, the performance comparisons show that except for the high join selectivity factor JS, the algorithm using the join index outperforms the hybrid-hash join algorithm. The performance difference between the algorithms increases as the size of the joined result decreases. For some joins of the low-join selectivity factor (JS) values, JOINJI can be 100 times more efficient than JOINHH (Figure 7). The cost formula used for JOINHH is independent of JS. This assumes that JS is low, which is true for most of the joins. However, when JS is high, hashing becomes inefficient [21], since the number of duplicate join attribute values is very high and leads to many collisions (the number of useful buckets becomes very low). Also, when JS is high, the number of duplicate surrogates in the index is high, making run-length compression very effective. Taking into account the deficiency of hashing for high JS and considering the run-length compression of the join index, we predict that the algorithm using the

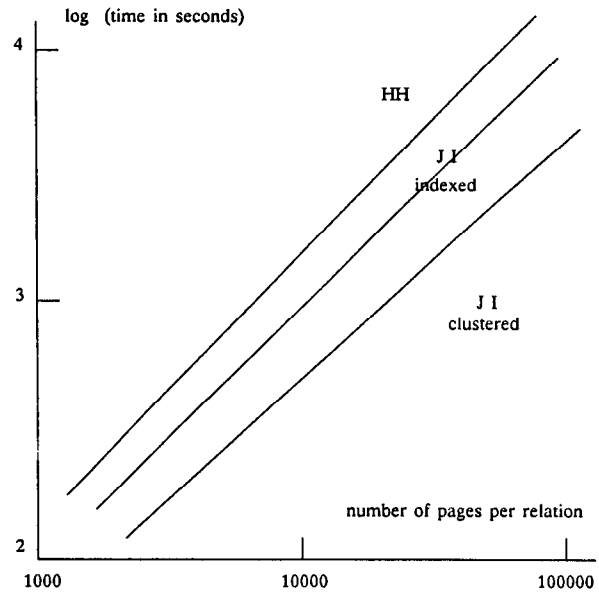

Fig. 11. Performance of JOINJI and JOINHH versus relation size ($|M| = |R| * 0.1$, SR = 0.5, SS = 0.1, JS = $5 * 10^{-7}$).

join index would always outperform the hybrid-hash join algorithm. However, note that the hybrid-hash join algorithm does not incur the index update overhead.

5. PROPERTIES OF JOIN INDICES

This section describes several properties of join indices. All these properties result from the fact that a join index is stored in a simple and separate data structure.

5.1 Algorithms Exploiting Hardware Availability

Future computers will have large amounts of RAM and a parallel processing capability. The join algorithm using the join index presented in Section 3 takes advantage of all available RAM. As shown in the performance evaluation, the algorithm's performance is proportional to the memory size. The adaptation of the join algorithm to parallel execution is easy. The join index can be divided into independent subsets, each being carried out by a different processing unit. The algorithm would guarantee that a page of the external relation is not read by more than one processor. However, the same page of the internal relation might be read by several processors.

5.2 Compatibility with Other Operations

The real value of join indices becomes apparent when they are combined with other indices for processing relational queries. A complex relational query is divided into two steps. The first step applies the query to indices (join indices, select indices, and others), producing an abstract of the result (the set of

surrogates of relevant tuples). In the second step, the base data that satisfy the query are accessed. Therefore, the most complex operations are performed using data structures smaller than the base relations. To the maximum extent, these data structures should be kept in memory.

5.3 ADT Join Predicate

The definition of a relational equi-join predicate is restricted to the equality of the join attributes. We can generalize the notion of conventional joins by defining the join predicate in terms of abstract data types (ADT) [12]. The motivation is that functionality as well as performance can be significantly improved if users are allowed to define join predicates according to the ADT operations that are most often used.

By storing separately the relationships existing between tuples, join indices can support ADT join predicates. For example, if a frequent query is to list the information about customers of Austin and parts, such that the customer was twenty years old when the part appeared on the market, the following ADT predicate can be used to specify a join index between CUSTOMER, noted C, and PART, noted P:

C.City = "Austin" and C.age − 20 = P.age

The ADT operations used for the join predicate must be computed only for updating the join index according to updates of the joined relations. Therefore, the update overhead incurred with ADT join predicates is significantly greater than that of equi-join predicates. On the other hand, the retrieval of tuples satisfying the ADT join predicate does not require reexecution of the ADT operation. It uses the join algorithm defined in Section 3.

5.4 Multirelation Clustering

The ideal clustering for improving the performance of joins places matching tuples of different relations physically close to each other. Below, we propose a multirelation clustering scheme used in combination with join indices. The multirelation scheme is similar to the CODASYL set implemented via a pointer array. A physical multirelation is organized as a B^+-tree on the basis of a multikey. The multikey is defined on surrogates, where the number of keys in the multikey is equal to the number of joins. The tuples in the leaves of the B^+-tree are ordered according to the tuples of the join index. An example of a multirelation is given in Figure 12, where the surrogates in leaves are the actual tuples. For a multirelation with three relations R, S, and T, the multikey might be (r, s).

The classical algorithms for B^+-trees must be slightly modified to keep the tuples of an internal relation sorted on a surrogate. Access (for any purpose) to only tuples of an internal relation requires access to the join indices defined on relations external to it for finding the parent tuples. For example, the access to tuple s_8 in Figure 12 implies accessing the join index on R and S to find r_3, which is used to locate the correct page in the multirelation. The join of relations clustered together is obviously performed by a sequential scan of the multirelation. Therefore, the join index does not need to be accessed. However, if the join is preceded by a selection on an internal relation, the access to the join index

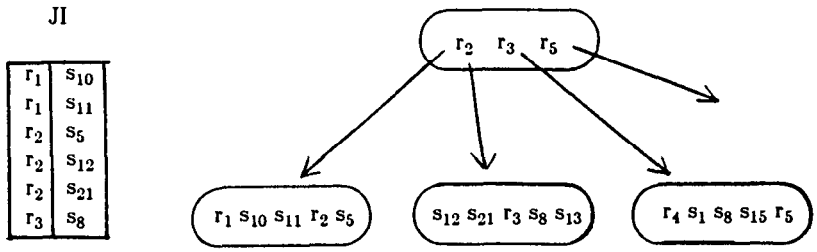

Fig. 12. Multirelation for R and S.

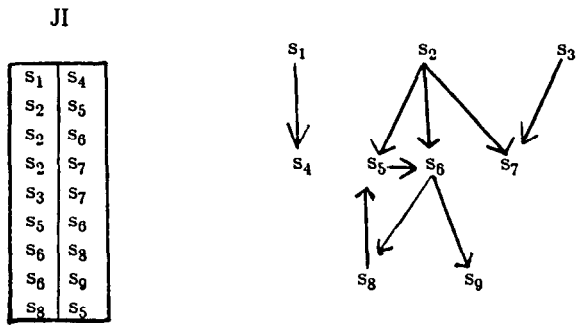

Fig. 13. Join index coding a directed graph.

is necessary for determining the subset of the multirelation that must be accessed.

Multirelation clustering is clearly the most efficient scheme for particular cases of joins. For example, the join of tuple r_4 (perhaps after selection) with s_1, s_8, s_{15} results in an access to a single data page, which is optimal. However, this scheme presents the shortcomings known in CODASYL systems that preclude its systematic application. Indeed, this scheme is well suited for hierarchies (1–m relationship), a very frequent case.

5.5 Coding of Graphs

A join index captures the semantic links that exist between tuples. If we represent the join of two tuples by an arc connecting those tuple surrogates, the join index can represent directed graphs in a very compact way. Therefore, it will be very useful for graph operations such as transitive closure. Figure 13 gives an example of a graph encoded as a join index. The index is clustered on the first attribute and depicts the parent–child relationship. Therefore, it is well suited for traversals in the parent–child direction. A join index clustered on the second attribute allows efficient traversals that follow the child–parent direction.

Applied to a single relation, a join index can capture its self-join as shown in Figure 14, where the join predicate is X.advisee = Y.advisor. For example, if we want to list the genealogy of advisee Ross, we may compute the transitive closure of relation Ph.D. Applying the operation on the join index that is smaller than the base relation is shown to be very efficient in [22]. Transitive closure preceded by select is also handled by our scheme.

	Ph.D				JI	
sur	advisee	advisor	university	year	sur	sur
1	Doe		Harvard	1970	1	2
2	Smith	Doe	Harvard	1976	2	3
3	Ross	Smith	MIT	1980	4	5
4	Hayes		Stanford	1978		
5	James	Hayes	Princeton	1984		

Fig. 14. Join index coding a self-join.

6. CONCLUSION

In this paper, we have proposed a simple data structure, called a join index, for optimizing semijoin and join operations in the context of complex queries. We presented algorithms for update, semijoin, and join and illustrated the use of join indices in relational queries. The overhead incurred in updating join indices appears to be small for the most frequent joins (joins on a foreign key). For all other joins, updates of join indices can increase the update cost of joined relations by a factor of approximately two. The join algorithm using a join index takes advantage of all available memory and is easily adaptable to parallel execution. The interesting features of join indices derive from the fact that they are stored separately, usually in a small data structure. Join indices support ADT join predicates, are independent of the storage model, and are helpful for multirelation clustering. They also represent directed graphs succinctly and thus can serve as a basic tool for recursive queries.

The analysis of the join algorithm using a join index shows its excellent performance. It generally outperforms the hybrid-hash join algorithm. Except for joins of very poor selectivity in which no optimization seems possible (the join becomes close to a Cartesian product), or for joins preceded by a strong selection (in which case classical indexing can be better), we claim that join indices should be employed. We limited our analysis to the join algorithm itself, since it is the most critical operation. However, the real value of join indices increases as queries become complex because the most complex operations are done on small data structures (select indices, join indices, etc).

For most of the joins, the join index whose size is proportional to the join selectivity factor will be small. Transitive closure, which appears to be a basic operator for supporting recursive queries, can be realized using a loop of joins and unions, two complex operations. In [22], we propose and analyze two algorithms for the transitive closure. It is shown that, for various values of parameters, applying either algorithm to a join index rather than the base data yields better performance. Again, the idea is to apply all complex operations (join, union) on join indices and to access the data at the very end. Thus, join indices appear very attractive in the evaluation of relational queries as well as in recursive queries.

In order to obtain empirical data to verify the analysis in [22] and in this paper, join indices have been implemented as part of a decomposition storage model [8] and a complex-object storage model [23]. In the latter paper, the concept of a join index is extended to capture the structure of complex objects,

and initial performance measurements using the Wisconsin benchmark [3] are given, which so far confirm our analysis. More thorough performance measurements are ongoing.

ACKNOWLEDGMENTS

The author is grateful to Haran Boral, David DeWitt, Ravi Krishnamurthy, and Marc Smith for their helpful comments and encouragement. The author wishes also to thank the anonymous referees for excellent comments that have improved the quality of this paper.

REFERENCES

1. BAYER, R., AND MCCREIGHT, E. Organization and maintenance of large ordered indexes. *Acta Inf. 1*, 3, 1972.
2. BITTON, D., BORAL, H., DEWITT, D. J., AND WILKINSON, W. K. Parallel algorithms for the execution of relational database operations. *ACM Trans. Database Syst. 8*, 3 (Sept. 1983), 324–353.
3. BITTON, D., DEWITT, D. J., AND TURBYFILL, C. Benchmarking database systems: A systematic approach. In *International Conference on Very Large Databases* (Florence, Nov. 1983), VLDB Endowment Ed., 1983, pp. 8–19.
4. BLASGEN, M. W., AND ESWARAN, K. P. Storage and access in relational databases. *IBM Syst. J. 16*, 4, (1977).
5. BRATBERGSENGEN, K. Hashing methods and relational algebra operations. In *International Conference on Very Large Databases* (Singapore, Aug. 1984), VLDB Endowment Ed., 1984, pp. 323–333.
6. CODD, E. F. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst. 4*, 4 (Dec. 1979), 397–434.
7. COMER, D. The ubiquitous *B*-tree. *Comput. Surv. 11*, 2, (June 1979), 121–137.
8. COPELAND, G. P., AND KHOSHAFIAN, S. A decomposition storage model. In *Proceedings of ACM-SIGMOD International Conference on Management of Data* (Austin, Tex., May 28–31, 1985). ACM, New York, 1985, pp. 268–279.
9. DEEN, S. M. An implementation of impure surrogates. In *International Conference on Very Large Databases* (Mexico City, Sept. 1982), VLDB Endowment Ed., 1982, pp. 245–256.
10. DEWITT, D. J., KATZ, R. H., OLKEN, F., SHAPIRO, L. D., STONEBRAKER, M. R., AND WOOD, D. Implementation techniques for main memory database systems. In *SIGMOD '84: Proceedings of Annual Meeting* (Boston, June 18–21, 1984). ACM, New York, 1984, pp. 1–8.
11. GOTLIEB, L. R. Computing joins of relations. In *ACM-SIGMOD International Conference on Management of Data* (San Jose, Calif., May 14–16, 1975). ACM, New York, 1975, pp. 55–63.
12. GUTTAG, J. Abstract data types and the development of data structures. *Commun. ACM, 20*, 6 (June 1977), 396–404.
13. HAERDER, T. Implementing a generalized access path structure for a relational database system. *ACM Trans. Database Syst. 3*, 3, (Sept. 1978), 285–298.
14. HALL, P., OWLETT, J., AND TODD, S. J. P. Relations and entities. In *Modelling in DBMS*, G. Nijssen, Ed. North-Holland, Amsterdam, 1976, pp. 201–220.
15. JARKE, M., AND SCHMIDT, J. Query processing strategies in the Pascal/*R* relational database management system. In *ACM SIGMOD International Conference on Management of Data* (Orlando, Fla., June 2–4, 1982). ACM, New York, 1982, pp. 256–264.
16. KNUTH, D. *The Art of Computer Programming: Sorting and Searching*. Addison-Wesley, Reading, Mass., 1973.
17. MISSIKOFF, M. A domain based internal schema for relational database machines. In *ACM SIGMOD International Conference on Management of Data* (Orlando, Fla., June 2–4, 1982). ACM, New York, 1982, pp. 215–224.
18. ROUSSOPOULOS, N. View indexing in relational databases. *ACM Trans. Database Syst. 7*, 2 (June 1982), 258–290.

19. SIMON, E., AND VALDURIEZ, P. Design and implementation of an extendible integrity subsystem. In *SIGMOD 84: Proceedings of Annual Meeting* (Boston, June 18–21, 1984). ACM, New York, 1984, pp. 9–17.
20. TSICHRITZIS, D. LSL: A link and selector language. In *ACM SIGMOD International Conference on Management of Data* (Washington, D.C., June 2–4, 1976). ACM, New York, 1976, pp. 123–134.
21. VALDURIEZ, P., AND GARDARIN, G. Join and semijoin algorithms for a multiprocessor database machine. *ACM Trans. Database Syst. 9*, 1 (Mar. 1984), 133–161.
22. VALDURIEZ, P., AND BORAL, H. Evaluation of recursive queries using join indices. In *Proceedings of the 1st International Conference on Expert Database Systems* (Charleston, S.C., Apr. 1986), Benjamin/Cummings, Menlo Park, Calif., 1986, pp. 197–208.
23. VALDURIEZ, P., KHOSHAFIAN, S., AND COPELAND, G. Implementation techniques of complex objects. In *International Conference on Very Large Databases* (Kyoto, Aug. 1986), VLDB Endowment Ed., 1986, pp. 197–208.
24. YAO, S. B. Approximating block accesses in database organizations. *Commun. ACM 20*, 4 (Apr. 1977), 260–261.

Received September 1985; revised July 1986; accepted September 1986