An Incremental Join Attachment for Starburst

Michael Carey
Eugene Shekita
Computer Sciences Department
University of Wisconsin-Madison

George Lapis
Bruce Lindsay
John McPherson
IBM Almaden Research Center

ABSTRACT — In this paper we describe the design, implementation, and performance of an incremental join facility that has been added as an extension to the Starburst extensible DBMS. This facility provides an efficient access path for joins that materialize many-to-one relationships, and it works by maintaining hidden pointer fields embedded in related tuples. The facility was constructed for two reasons: as an experiment in using pointers in the internals of a relational DBMS, and as a stress-test of the Starburst extension architecture. In addition to describing the join facility and its performance, we also summarize what it taught us about extensibility both in Starburst and in general.

1. INTRODUCTION

In the two decades since the relational data model was first proposed, much progress has been made towards efficient implementation techniques for the model. As evidence, a number of relational database system products are now commercially available, and most new database applications developed today are based upon relational database systems. Despite this progress, however, relational database systems are still generally considered to be inferior to systems based on the hierarchical and network models in terms of their "inherent processing efficiency" [Elma89]. One reason for this is that systems based on the hierarchical and network models usually provide more flexibility in terms of storage and access structures. For example, such systems permit pointer-based (rather than just value-based) representations of relationships, and inter-record clustering is commonly available as an option for tuning performance when pairs of record types are frequently co-referenced in an application.

In the relational data model, relationships are frequently represented by foreign key attributes with values that match those of primary key attributes elsewhere in the database. Related foreign and primary keys can be declared in the schema through referential integrity constraints [Date81]. Support for referential integrity is now recognized as desirable, to the point where it is viewed today as an essential part of the complete relational model [Elma89]. Some commercial systems (e.g., DB2 and UNIFY) already provide a certain degree of referential integrity support, and it is not unreasonable to expect that all commercial relational database systems will include it in the future. This provides an opportunity to further improve the performance of relational database systems — if the system is aware of which records logically refer to which other records, pointer-based access structures can be used "under the covers" to link related records together. In addition to being useful for referential integrity maintenance, such structures can also be used as access paths for joins and as a basis for clustering. It is these latter opportunities that motivated the work described here.

In this paper, we describe a pointer-based access structure that has been added as an extension to the Starburst extensible DBMS [Schw86, Haas90]. This access structure, called the Starburst IMS[1] attachment, provides support for both pointer-based joins and inter-relation clustering. It works by adding hidden pointer fields to related records and by incrementally keeping these hidden pointer fields up-to-date as related records are inserted, deleted, or modified in the database. In other words, it provides support much like that provided in network database systems for set types where insertion is automatic and set selection is structural [DBTG71, Elma89]. The IMS attachment also provides access paths that are available to the Starburst optimizer in devising plans for processing foreign key joins.

The IMS attachment was actually added to Starburst for two reasons: (i) to experiment with the benefits offered by pointer-based access structures and inter-relation clustering in a relational context, and (ii) as a test of the extensibility of the architecture of Starburst. In the remainder of this paper we describe the design, implementation, and performance of the Starburst IMS attachment. In Section 2, we briefly review the relevant features of the Starburst extension architecture. Section 3 describes the design and implementation of the IMS attachment, including a discussion of how it was integrated with the other components of Starburst. Section 4 provides performance results for queries run with and without IMS support, providing an indication of the potential benefits of employing pointer-based access structures in a relational DBMS. Section 5 describes some lessons that we

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

[1] IMS stands for Incrementally Matched Sets. The name also reflects its main objective -- to enable a relational DBMS to compete with systems like IMS in efficiently handling many-to-one relationships!

learned in the process of building the IMS attachment, as doing so required certain changes in the Starburst extension architecture. Finally, Section 6 reviews our main conclusions.

2. EXTENSIBILITY IN STARBURST

The Starburst DBMS consists of two major components, the Corona query language processor and the Core data manager [Schw86, Haas90]. These two components correspond roughly to the RDS and RSS levels of the original System R architecture [Astr76]. One of the main extensibility-oriented features of Core is its data management extension architecture [Lind87], while extensibility at the Corona level includes the use of a rule-based query optimizer [Lohm88, Haas89]. In this section of the paper we briefly review these features of Starburst to set the stage for describing the IMS attachment. We focus mainly on Core, as it is the most relevant Starburst component for our purposes.

The Core data manager is designed to support two classes of extensions, and for each class it defines a generic interface that must be adhered to by all extensions of that class. The first class of data management extension is known as a *storage method.* A storage method provides a means of storing the records of a relation. Example storage methods include sequential files, nonrecoverable temporary files, and B+ tree files. Every storage method must provide a well-defined set of operations that include create storage method instance, destroy storage method instance, insert record, and delete record. In addition, each storage method must support the notion of a record key, and direct-by-key and key-sequential record access operations must be provided as well. The structure and meaning of keys are defined by the storage method. For example, a key could be made up of one or more field values (as in the case of a B+ tree storage method) or it could simply be a physically-oriented record identifier (as in the case of a sequential file storage method). Every relation managed by the Core data manager is physically stored as an instance of some storage method.

The other class of data management extension that Core supports is called an *attachment.* An attachment is a component that can be associated with a relation, and it is given the opportunity to react to changes in the contents of the relation.[2] Attachments are notified of changes as a side effect of relation updates. Whenever a record insertion, deletion, or modification occurs, the corresponding update routine for each of the relation's attachments is invoked so that it can take note of the change. A simple example of an attachment is a secondary index (e.g., a B+ tree), but the notion is actually somewhat more general. In addition to tracking changes to keep auxilliary data structures up-to-date, attachments are permitted to veto operations. This makes it possible to use the attachment interface to support functions such as integrity constraint checking or production rule triggering. In addition to change notification, the generic attachment interface also supports both direct-by-key and key-sequential record accesses. These are relevant for attachments such as secondary indices that serve as access paths. So, in addition to being managed by a primary storage method, every relation in Core can have zero or more instances of each of the available attachment types. Figure 1 summarizes these concepts, showing an Employee relation stored via the sequential file storage method, indexed via several instances of the B+ tree attachment type, and guarded by several integrity constraint instances.

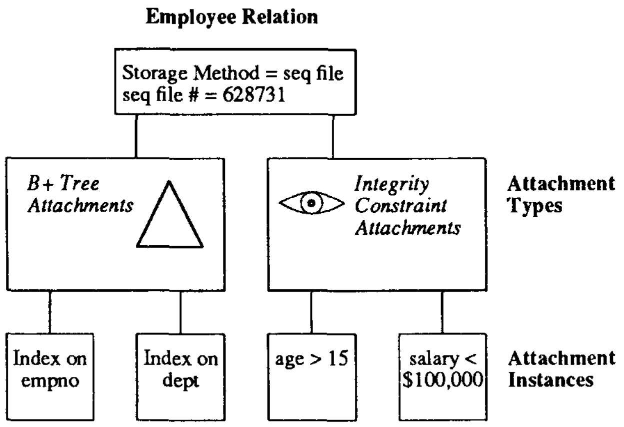

Figure 1: Storage Method and Attachment Example.

Figure 2 highlights the distinction between storage methods and attachments. It also summarizes the basic operations that each provides. The *direct operations* in the figure are those which are directly invoked by Corona when executing queries, while the *indirect operations* listed there are operations which are invoked only as a side effect of some direct (update) operation. The *common services* box at the bottom of the figure provides support for commonly needed functions such as manipulation of individual records, predicate evaluation, etc.

In addition to the operations listed in Figure 2, storage method and attachment extensions also support the creation and destruction of their instances. Since different storage method and attachment types tend to have different requirements regarding create time parameters (e.g., keys, fill factors, etc.), the Starburst data definition language supports extensible parameter lists for handling new storage methods and attachments [Haas90].

Of course, extensions to Starburst at the Core level are only useful if the Corona query processor is aware of them and able to take advantage of them when appropriate. To enable the Corona query optimizer to be informed about newly added Core facilities, the query optimizer has a rule-based architecture [Lohm88]. Optimization rules, which are called *strategy alternative rules* (or STARs), have a structure similar to that of the production rules of a context-free grammar. To inform Corona about a new index attachment, one need only (i) add an additional option to the right hand side of the rule that describes how alternative indices can be used to select records from a relation and (ii) provide certain auxilliary functions that handle things like estimating the cost of using the new attachment. The hierarchical nature of Corona

[2] Actually, Starburst now permits an attachment to be associated with multiple relations. This extension of the original design was driven by IMS attachment needs, as Section 5 will describe.

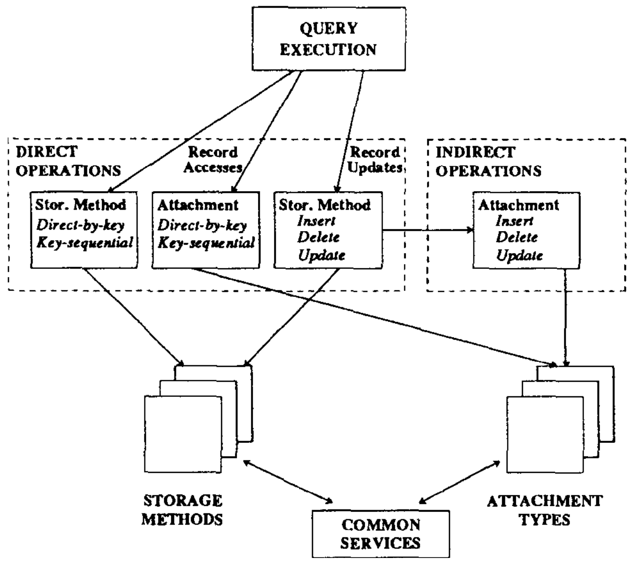
QUERY EXECUTION
DIRECT OPERATIONS
Record Accesses
Record Updates
INDIRECT OPERATIONS
Stor. Method Direct-by-key Key-sequential
Attachment Direct-by-key Key-sequential
Stor. Method Insert Delete Update
Attachment Insert Delete Update
STORAGE METHODS
COMMON SERVICES
ATTACHMENT TYPES

Figure 2: Storage Method and Attachment Interfaces.

rules ensures that, in all contexts where record selection is performed, the new attachment type will be considered as an option because all selections will be optimized using the newly extended rule.

For query execution, Corona provides a demand-driven query execution subsystem (QES) where each primitive operator consumes and produces record streams [Haas89]. Adding a new QES operator (e.g., a new join method) is similar to adding a new storage method or attachment to Core. It must adhere to the QES operator interface, and it is made known to the Corona query optimizer by extending the appropriate operator selection STAR (e.g., the STAR that describes alternative join methods).

3. THE IMS ATTACHMENT

The Starburst IMS attachment provides support for pointer-based joins, for clustering related records together on disk, and for a limited form of referential integrity. This support is provided in the context of many-to-one (i.e., referential) relationships. In this section of the paper we describe the design and implementation of the IMS attachment as well as the associated pointer-based join methods.

3.1. Creating an IMS Attachment Instance

The job of the IMS attachment can be viewed as incrementally joining related records from a pair of relations when changes are made to either relation. One of the relations, called the *child* relation, must have a field (or field set) that references by value a record of the other relation, called the *parent* relation. In other words, the reference field (or field set) of the child is a foreign key, and the field (or field set) that it matches in the parent relation is the parent's primary key.

To create a new IMS instance, the user must specify the pair of relations involved, the relevant foreign and primary keys, and two additional pieces of information. The first piece of information is the name of another attachment, which is a unique index on the primary key of the parent relation. This index is used when the IMS attachment needs to find a parent record based on its primary key. The second piece of information is the desired clustering mode. IMS provides three clustering options — NONE, where related records are linked together but are not clustered on disk; CHILD clustering, where child records with a common parent are clustered together on disk; and FULL, where a parent record *and* its associated child records are all clustered together on disk.

At this point, an example is in order. Consider the following schema for the inevitable Employee/Department database:

Department(*deptno*, dname, budget, location)
Employee(*empno*, ename, age, salary, dept)

If the query workload for this database includes frequent requests for department information to be joined with the corresponding employee information, it may be beneficial to construct an IMS instance to maintain the join and to cluster each employee with its department. To do this, one could type the following Starburst command in setting up the physical schema for the database:

```
create IMS EmpDept on Department
   childtable (Employee),
   parentkey (deptno),
   childkey (dept),
   parentindex (DnoIndex),
   clustering (FULL)
```

This would create an IMS instance called EmpDept with Department as the parent relation. The command text from the second line on is specific to the IMS attachment type, and comprises the attribute list of the attachment creation syntax [Haas90]. The second line of the command declares that Employee is the child relation. The third and fourth lines specify the join, giving the primary key of the parent relation (deptno) and the foreign key of the child relation (dept). In general, these keys may be lists of fields. The fifth line identifies DnoIndex as a unique, permanent index on the deptno field of the Department relation. The final line of the command specifies that FULL clustering is desired.

3.2. Data Structures and Updates

When a new IMS instance is created, the schemas of the parent and child relations are extended to include several hidden pointer fields.[3] Figure 3 illustrates the IMS pointer structure via an example where three Employee records (E3, E84, and E39) reference a common Department record (D11). A parent record points to the first of its children, if any, with this pointer being null if no corresponding children exist. Each child record points to its parent and to the next and previous children that share the

[3] Due to time constraints on the initial implementation, we disallowed the creation of an IMS attachment if the parent or child relation is already populated with data. However, since missing fields are defined to be null in Starburst [Haas90], it would not be difficult to relax this constraint.

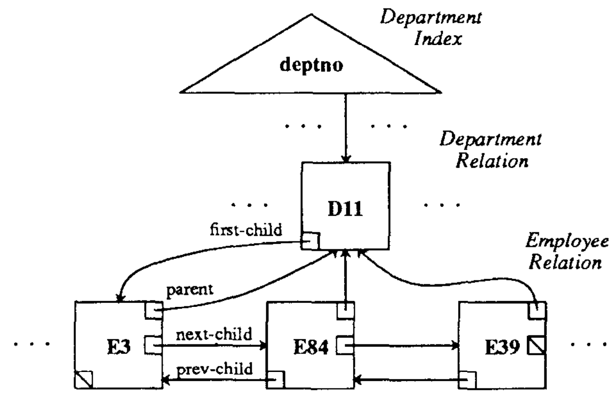

Figure 3: Example of IMS Data Structure.

same parent. This is similar to structures that are used to implement set occurrences in network database systems [DBTG71, Elma89]. We selected this particular structure for use in the IMS attachment because it allows (i) a parent to locate its first child in at most one I/O, (ii) a child to locate its parent in at most one I/O, and (iii) the space overhead for parent records to be independent of the number of children.

An IMS instance is notified when an update is made to either its parent or child relation. To simplify the initial IMS implementation, we chose to enforce the flavor of referential integrity where child records can exist only if they reference a valid parent record, and where parent record deletions and modifications are rejected (vetoed) if they would cause this constraint to be violated. Consequently, to process updates to the parent relation, an IMS instance has relatively little to do. Insertions can simply be ignored, as unspecified fields (such as the child list pointer in a new parent record) are initialized to null by default in Starburst. Deletions are permitted if the parent record's child pointer is null; otherwise the IMS instance vetos the operation. Similarly, modifications are ignored unless they affect the primary key, in which case they are also vetoed.

When a new record is inserted into a child relation that has an IMS attachment, the affected IMS instance is notified both before and after the record is physically inserted into the underlying storage method instance. In order to maximize the effectiveness of CHILD or FULL clustering, and to keep insertions simple, new child records are always added at the front of their parent's list of children. Thus, IMS pre-insertion processing involves: (i) fetching the child's parent record using the parent index, (ii) using the parent record's storage method key and first child pointer to set up the new child record's parent pointer and next child pointer, and (iii) optionally giving Core the parent or current first child's storage method key as a clustering hint (see below) to use when inserting the new child. Post-insertion processing for the affected IMS instance involves fetching and updating both the new child record's parent and the next child record in order to modify their pointers to refer to the new child. Note that the new child's storage method key is not known until the child record has actually been inserted, so these pointers cannot be set at pre-insertion time.

As indicated above, Core provides a mechanism that enables attachments to provide clustering hints to storage methods when new records are about to be inserted. The IMS attachment uses this mechanism to support its FULL and CHILD clustering modes. To support FULL clustering, the child pre-insertion routine gives either the child's parent or the parent's current first child as a "near hint" for inserting the new child record. The parent is used as the hint if it has no other children, and the current first child is used as the clustering hint otherwise. The net effect is that the first child of a parent will go on the same page, if space allows, as will subsequent children of the same parent, and this continues until that page fills up. After that, the next series of child records for this parent will be placed on another nearby page, and will accumulate there until that page fills up as well. Subsequent children will be placed on yet another nearby page, then another, and so on. CHILD clustering is handled almost identically, except that no clustering hint is given when the very first child record of a parent record is inserted (since child records are not clustered with their parent in this case).

When a record is deleted from the child relation of an IMS instance, the affected instance is again notified both before and after the record is physically deleted. The IMS attachment uses only the pre-deletion call in this case, using it to perform the obvious processing on the affected list of children. If the deleted record is the first child of its parent, then its parent record and the next child record are fetched and their pointer fields are modified. If the deleted record lies somewhere in the middle of the list, then the previous and next child records are fetched and modified. Finally, if the deleted record is at the end of the list, only its predecessor must be fetched and modified.

When a child record is updated, the affected IMS instance is notified both before and after the update operation actually takes place, as for insertions and deletions. If the update does not affect the child record's foreign key field, it is simply ignored by the IMS attachment. If it does change the child's foreign key, it is handled as a deletion from the old parent's list of children followed by an insertion into the new parent's list; the details follow from the discussions above.

3.3. Join Processing

An IMS instance can be used to process a join query if the IMS parent and child keys appear together as an equi-join predicate. The query optimizer thus considers the use of IMS join strategies whenever this condition holds. The current implementation supports three IMS-based join strategies: the parent-child (PC) join method, the child-parent (CP) join method, and the sorted child-parent (CP-SORT) join method. The PC join method, in turn, relies on a scan facility associated with the IMS attachment type. This scan facility, given the first child pointer from a particular parent record, enumerates all of the children of that parent by traversing the list of children.

The PC join method is essentially an IMS-based nested-loop join using the parent relation as the outer relation for the join. In fact, due to the way the optimizer's rule set is structured, the Starburst execution plan for an IMS PC join simply reduces to a standard nested-loop join operation with an IMS scan being used to retrieve records from the inner (child) relation. Thus, instead of using a key value from each parent record to perform an index probe of the child relation, as in a conventional nested-loop index join, each parent's first child pointer is used to perform an IMS scan of its children. This eliminates the index I/O and index page processing that would be required in a nested-loop index join.

The CP join method is another IMS-based nested-loop join method, but the roles of the parent and child relations are reversed. An IMS CP join reduces to composing an access to the child relation with a fetch of the parent record using the parent pointer field from the child. Again, compared to a nested-loop index join using the parent relation's primary key index, the CP join method saves on index I/O and page processing. To further reduce the amount of I/O required for a large CP join in cases without clustering, the records of the child relation can be sorted on their parent pointer field by inserting a sort operation before the fetch of the parent records; this is how the IMS CP-SORT join method works. To further improve performance for CP-SORT joins, repeated accesses to the same parent record are avoided by an optimization where the sort operator indicates that its current output record will be followed by another record with the same sort key.

Given the similarity of IMS joins to an index-based nested-loop join, the task of adding these join methods to Starburst was not difficult. The use of IMS pointer fields to perform equi-joins was expressed by adding several alternatives to the existing join method selection STAR of the optimizer. In this join STAR, the applicability of various join methods is conditioned on things such as the availability of access methods and the eligibility of predicates. Since IMS join plans are expressed as alternatives of the join operation STAR, no new primitive Starburst execution operators (LOLEPOPs) had to be built to support IMS joins. However, one other slight twist did arise: IMS join plans require the relevant IMS pointer fields to be included when fetching projected records from the query's base relations. This requires that the field lists for the execution plan's outer and inner relation fetches be modified to include these pointer fields when IMS joins are selected.

3.4. Related Work

In addition to its direct relationship to network database systems, the IMS attachment is also related to the implementation tactics of several relational database systems. System R supported *links* (record pointers) internally at the RSS level, but never exploited them for handling user data [Cham81]. UNIFY includes an *explicit relationship* access method [Rube87] based on a pointer structure much like that of the IMS attachment. Valduriez proposed using *join indices* to maintain the join of a pair of relations via a dual B+ tree data structure stored separately from the relations themselves [Vald87]. Join indices are more general than the IMS attachment, as they can handle joins that are not foreign key (parent-child) joins. The IMS attachment is superior for foreign key joins, though, since it avoids the overhead involved in storing and accessing the join index data structure. As for clustering, ORACLE supports multi-relation *clusters* in order to reduce paging for joins [Mart86].

4. EXPERIMENTS AND RESULTS

In order to study the join processing benefits offered by the IMS attachment, we conducted a series of experiments comparing traditional join methods with the three IMS join methods described in the previous section. Two traditional join methods were used for the comparisons: nested-loop join using an index on the inner relation (NL-INDEX) and a variant of sort-merge join where the outer relation must be sorted but the inner relation can be accessed in sorted order using a clustered index (NL-SORT). Our experiments include both full join queries as well as queries with a selection followed by a join. The data used for the experiments are the OneK (1,000 record) and TenK (10,000 record) relations from the Wisconsin benchmark database [Bitt83]. Each query was executed on test relations that were constructed using several clustering alternatives to investigate the impact of clustering on join query performance.

4.1. The Test Database

Each of the experiments that we ran involved joining a parent relation with a child relation and extracting several attributes from each. As prescribed in [Bitt83], the test relations each contain sixteen attributes — thirteen 4-byte integer attributes and three 52-byte string attributes. Each relation has two candidate key attributes, *unique1* and *unique2*, whose values range from 0 to N-1 where N is the cardinality of the relation. The other attribute relevant to our experiments is the *thousand* attribute, which contains values that range from 0 to 999. We chose *unique2* as the primary key for the parent and child relations, with the *thousand* attribute of the child serving as the foreign key for relating child records to their corresponding parent.[4] The parent and child relations are both indexed on their *unique2* attributes, and the child relation is also indexed on its *thousand* attribute. In each case, the index is an instance of the Starburst B+ tree attachment type. For query plans based on IMS joins, an IMS instance was also attached to the parent-child relation pair.

The experiments that we ran considered two different cases for the average cardinality of the parent-child relationship. In half of the tests, the average number of child records per parent record was just one (1:1). For these tests, the parent and child were each OneK relations, each containing 1,000 records and occupying 67 4K-byte pages of disk space, excluding the overhead for indices and IMS pointers. In the other half of the tests, the average number of children per parent was ten (10:1). As before, the parent was a OneK relation, but in this case the child relation was a TenK relation, containing 10,000 records and consuming 667 pages of space without overhead.

As mentioned earlier, our tests considered several alternative parent-child clustering strategies. In each of the tests, parent records were physically ordered by their *unique2* key values. The first clustering strategy is NONE, where parent and child

[4] Unlike [Bitt83], we generated the *thousand* value for each record by randomly choosing an integer (with replacement) from the integers 0 through 999; this seemed a bit more realistic for our purposes than having 1000 distinct *thousand* values.

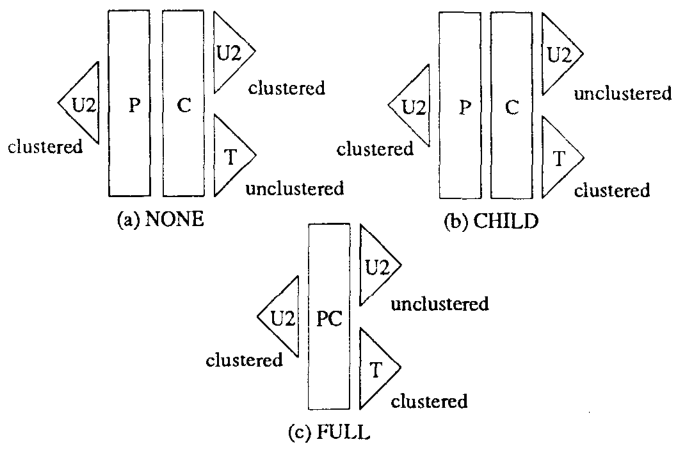

Figure 4: Test Database Clustering Alternatives.

records were stored separately and child records were clustered on their own *unique2* attribute values. The next form of clustering is CHILD, where the relations were stored in separate areas but were relatively clustered (i.e., both were clustered on the parent key). In this case, child records were clustered on their *thousand* values. The third and final clustering option is FULL. The relative record ordering in this case was the same as in CHILD, but with child and parent records clustered together on pages of a common database space. Figure 4 illustrates the three clustering alternatives by showing which indices were clustered and whether the parent and child relations were stored separately or together in each case. In the figure, *P* represents the parent relation, *C* represents the child relation, *U2* stands for the *unique2* attribute, and *T* stands for the *thousand* attribute.

Based on the options enumerated above, the test database for the full set of experiments contained 12 parent-child relation pairs, 6 with IMS attachments connecting them and 6 without. In each case, the 6 pairs are due to the two different relationship cardinalities, 1:1 and 10:1, and the three different clustering options, NONE, CHILD, and FULL. The relations and their B+ tree and IMS attachments were defined using the Starburst SQL interface. Once defined, the 12 parent-child relation pairs were then populated in accordance with the chosen clustering strategies. This was done using a driver program that repeatedly performed insertions. (While experiments with a more dynamically clustered test database would also be interesting, we have not yet run any such experiments.)

4.2. The Test Query Set

For our initial tests, we wished to investigate IMS attachment performance for both full join queries and select-join queries. The general form of the queries used in our experiments is captured by the following SQL query:

```
select P.unique1, P.unique2, C.unique2
from Parent P, Child C
where P.unique2 = C.thousand
and (P.unique2 >= ...) and (P.unique2 <= ...)
and (C.unique2 >= ...) and (C.unique2 <= ...)
```

As indicated, all of the queries used in our experiments are parent-child joins that retrieve integer attributes from each relation. The last two lines of the query, which are optional, are used to control the percentage of parent or child records that are actually selected. For our tests, we used a total of five basic SQL queries — a full parent-child join, two queries joining 1% and 10% of the parent relation with the entire child relation, and two queries joining 1% and 10% of the child relation with the complete parent relation.

Each of the five queries was run using several different query plans for both the 1:1 and 10:1 cardinalities and for all three clustering strategies. To control the join methods used in the query plans, each plan was hand-generated and then run using the Starburst query execution driver. Consequently, all measurements reported here are for compiled query plan execution (i.e., they do not include query optimization overhead). We discuss the various query plans in a bit more detail as the results are presented. All records produced by the queries were projected into a buffer and discarded in order to eliminate overheads due to writing results to the screen or to a temporary relation.

4.3. The Test Environment

The hardware used for the experiments was an IBM RT/PC with a 10MHz processor and 8MB of physical memory. The machine had three disks, including two 70MB drives and a 114MB drive. The operating system was AIX Version 2.2. The RT used for the tests was running in single-user mode throughout the tests.

The tests were run using the latest version of Starburst. The test relations were stored on one of the 70MB drives, and temporary relations were placed on the 114MB drive. Starburst currently runs on top of AIX, storing relations in the AIX file system, so we configured Starburst to have 100KB of buffer space (25 4KB pages) and configured AIX with only 50KB of I/O buffer space. This was done in hopes of minimizing the impact of AIX buffering on our test results. For sorting, Starburst does not use the global buffer pool, relying instead on a separate sort buffer; we configured its sort buffer size to be 100KB to provide a comparable amount of space for sorting as for regular I/O. Finally, the block size for AIX is 2KB, with Starburst assuming 4KB pages, so each Starburst I/O actually requires two AIX I/Os.[5] The fact that each Starburst I/O requires multiple AIX I/Os causes the execution times presented here to be somewhat higher than they would be if Starburst did its own low-level file management. Fortunately, our interest is in the relative (as opposed to absolute) performance of the different join methods.

In running the tests, we first populated the database via repeated insertions and then ran the sequence of IMS and non-

[5] Actually, each Starburst I/O involved an average of 2.73 physical AIX I/Os according to our measurements. This is because the underlying files were large enough to require indirect blocks; the additional I/Os were caused by inode and indirect block accesses. The ratio of 2.73 was quite stable, with a 95% confidence interval of 2.73 ± 0.05 over the full set of experiments.

IMS query plans one after another against the appropriate parent-child relation pairs. To prevent inter-query buffering effects from biasing our results, an unrelated copy of the TenK relation was scanned before each query to flush both the Starburst and AIX buffer pools. We measured the elapsed time for each query along with the CPU time consumed, and we measured the number of Starburst I/Os and the number of page fix operations for each query in order to characterize the page access behavior of the different join algorithms. We also measured the AIX-level I/O traffic in order to ensure that hidden AIX-level buffering did not adversely affect our measurements (though we present only the Starburst I/O numbers here). The timing information was obtained using standard AIX system calls, as was the AIX I/O information. The Starburst I/O and buffer-related measurements were obtained from internal Starburst counters.

4.4. Full Join Query Experiments

The first experiments that we present are the full joins. Figure 5 shows the results for the 1:1 relationship cardinality, where the parent and child relations were both OneK relations. The figure presents measurements under all three clustering alternatives for the three IMS join methods (PC, CP, CP-SORT) and the two non-IMS join methods (NL-INDEX, NL-SORT). Figure 5a shows the actual elapsed time measurements, and Figures 5b and 5c show the measurements for the number of Starburst I/Os and number of page fixes, respectively. Figure 6 presents the corresponding results when the child is a TenK relation, i.e., when the relationship cardinality is 10:1. Each of the elapsed time figures (here and elsewhere) is also annotated to indicate the percentage decrease in join time provided by the best IMS join method relative to the best non-IMS join method.

Throughout the full join experiments, the outer relation for the NL-INDEX and PC join methods was the parent relation, whereas the outer relation for the NL-SORT, CP, and CP-SORT join methods was the child relation. We remind the reader that NL-SORT is essentially a sort-merge join — the child relation is sorted by its foreign key field and then the parent's clustered primary key index is used to retrieve corresponding parent records in physical order. Put another way, the parent relation is clustered optimally for NL-SORT since it is in *unique2* order. Were this not the case, NL-SORT would perform significantly worse relative to CP-SORT than it does here.

For NONE clustering, where the parent and child relations were unclustered relative to one another, Figure 5a indicates that CP-SORT was by far the best join method. NL-SORT was the next best method, followed by the CP and PC methods, with NL-INDEX performing the worst of all. As indicated in the figure, the best IMS join method (CP-SORT) was 48% faster than the best non-IMS method (NL-SORT). The reason for these results is clear from Figures 5b and 5c. Figure 5b shows that CP-SORT and NL-SORT did far fewer I/Os than the other methods, with CP-SORT doing slightly fewer I/Os than NL-SORT. Both of these methods scan the child relation, project out the fields needed, sort the result, and then scan the parent relation in physical order (directly in CP-SORT and through the *unique2* index in NL-SORT). In contrast, the other methods scan one relation and randomly access corresponding records of the other relation. This random access behavior naturally leads to many more I/Os. Figure 5c shows why CP-SORT performed so much better than NL-SORT — CP-SORT and the other IMS join methods required far fewer page fix operations than the non-IMS join methods. While the IMS methods use pointers to fetch inner relation records, the non-IMS methods must use index lookups to fetch them; the resulting page fixes and correspondingly higher CPU times contributed significantly to the elapsed times of the NL-SORT method (and the NL-INDEX method).[6]

For CHILD clustering, where the parent and child relations were relatively clustered with respect to each other, Figure 5a shows that all three IMS join methods ran significantly faster than the non-IMS methods, with CP-SORT performing the best of the group. Figures 5b and 5c indicate why this was the case. CHILD clustering allows all of the join methods to compute the join in roughly the same number of I/Os, as shown in Figure 5b. This is because child records are stored in parent-record order; also, the two sort-based methods are able to sort the projected child data in main memory. In terms of page fixes, Figure 5c shows that CP-SORT does the least page fixes, followed by the other two IMS join methods, with the two non-IMS methods doing significantly more fix operations due to inner relation index processing. CP-SORT does the fewest fixes because of the optimization that was noted in Section 3; it does one fix per parent page rather than one per parent record since a run of references to a given page can be served by a single fix operation. The important point is that the IMS join methods again enjoy significant performance benefits because they follow pointers rather than probing an index to retrieve referenced records.

For FULL clustering, where the parent and child relations were stored together on disk, Figure 5a again shows that all three IMS join methods performed significantly better than the non-IMS methods. CP performed the best, then PC, then CP-SORT, with NL-SORT and NL-INDEX taking significantly more time to perform the join. Figure 5b shows that the two sort-based methods did more I/Os in this case since both of them scan, project, and then sort child relation data which is now spread over twice as many disk pages due to the parent-child clustering. And, of course, there is no advantage to sorting in this case, as evidenced by the increased elapsed time for the sort-based methods as compared to CHILD clustering. The CP join method was slightly faster than PC join because it involved slightly less CPU processing overhead. This is because a CP join does a direct parent record fetch operation for each child record, while a PC join must open, use, and then close a child record scan for each parent record processed.

Let us now examine Figure 6, the 10:1 case, where the child relation is ten times larger. A number of the results are similar to those of Figure 5, and for similar reasons, so we concentrate on the differences and their causes.

[6] Reading index pages also involves locking them; this contributes to the CPU time advantage of IMS methods over non-IMS methods as well.

For NONE clustering, Figure 6a shows the same relative join method ordering as obtained in the 1:1 case, with the IMS CP-SORT join method being the best performer by a significant margin. One difference here is that the PC join method is significantly worse than the CP method due to more I/Os, as shown in Figure 6b. This is because CP scans a large relation while randomly probing a small relation, which leads to more buffer hits than scanning a small relation while randomly probing a large one as PC does. Another difference affecting the relative performance of all of the join methods here is that methods which use the child as the outer relation require relatively more fix operations, as shown in Figure 6c. This is because of the increased child relation size, with a parent page being fixed for each child record processed (except in CP-SORT).

For CHILD clustering, Figure 6a shows that the IMS methods outperform the non-IMS methods by quite a bit, as in the 1:1 case. The PC join method does the best in this case. The CP-SORT method performs worse here because more data must be retrieved and sorted, as the Starburst I/O counts of Figure 6b indicate. The CP method performs worse relative to the PC join method because of a larger number of page fixes, as Figure 6c shows. The PC join method scans the parent relation, fixing each parent page once, while the CP method fixes a parent page once for each of ten thousand child records.

The results for FULL clustering are similar to those of CHILD clustering, with PC join outperforming the other IMS methods, especially CP-SORT, and beating the non-IMS methods by a large margin. The reasons are similar to those just given for CHILD clustering and to those given earlier for FULL clustering in the 1:1 case.

4.5. Select-Join Query Experiments

The next results that we present are for select-join queries. Figure 7 presents measurements for both the 1:1 and 10:1 relationship cardinalities for the query that joins a 10% selection of the parent relation with the full child relation. Two join methods

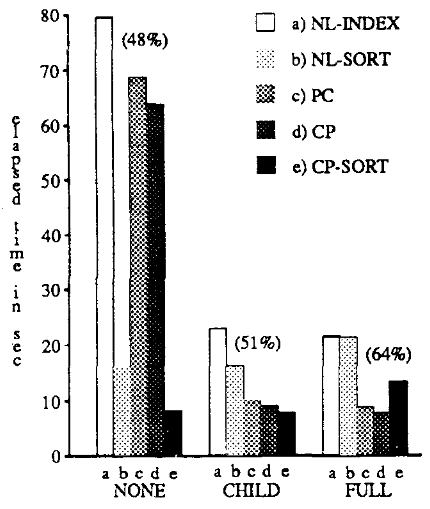

Figure 5a: Full Join Time (1:1).

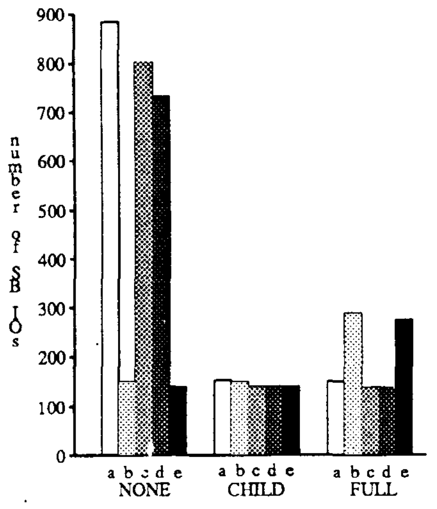

Figure 5b: Full Join I/O (1:1).

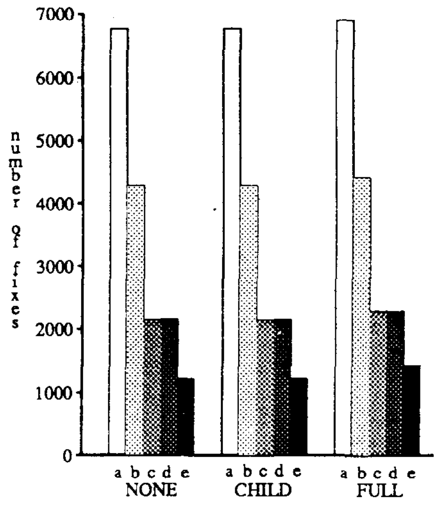

Figure 5c: Full Join Page Fixes (1:1).

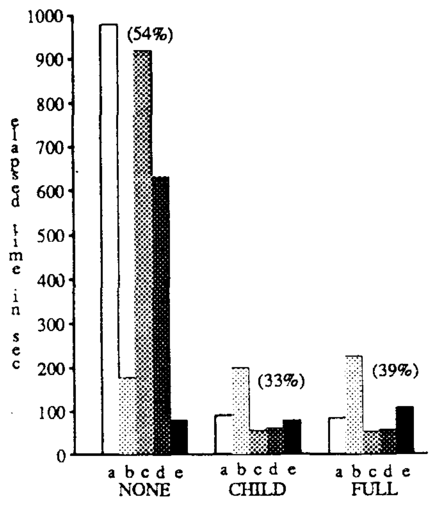

Figure 6a: Full Join Time (10:1).

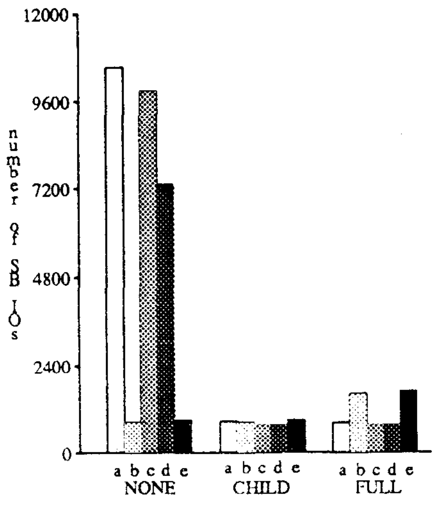

Figure 6b: Full Join I/O (10:1).

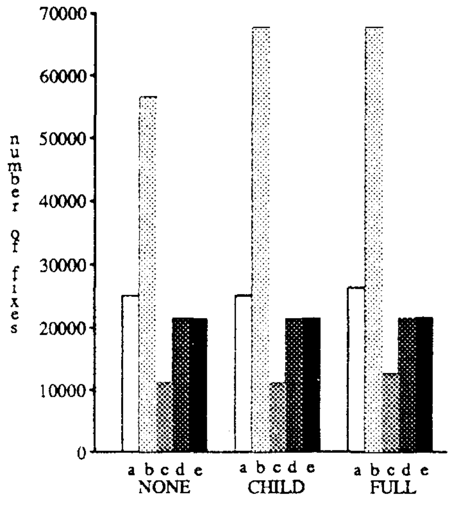

Figure 6c: Full Join Page Fixes (10:1).

were tested, an IMS method (PC) and a non-IMS method (NL-INDEX). In both cases the selected parent relation was the outer relation for the join. The other three methods were not tested here since they require the child to serve as the outer relation. Figure 7 includes results for all three clustering alternatives.

Figure 7a shows the elapsed time results for the 1:1 case. As shown, the PC join method outperformed the NL-INDEX method under all three clustering alternatives. As shown in Figure 7b, the PC join method required fewer Starburst I/Os in each case. The page fix counts are not shown there, but the PC method needed only 40% of the page fix operations performed by the NL-INDEX method. The reason that the PC method outperforms the NL-INDEX method by more when the relations are relatively clustered is clear from Figure 7b; fewer I/Os are involved in those cases, so the page processing savings of PC over NL-INDEX contributes more heavily to their performance differences there. Figures 7c and 7d show the corresponding results for the 10:1 case, where the child relation is a TenK relation instead of a OneK relation. The results are similar to those for the 1:1 case, but the elapsed times are higher due to the larger child relation size.

Figure 8 presents measurements for the 1:1 and 10:1 relationship cardinalities for the query that joins a 10% selection of the child relation with the full parent relation. Four join methods were tested in this case, including two IMS methods (CP and CP-SORT) and two non-IMS methods (NL-INDEX and NL-SORT). In all cases, the selected child relation acted as the outer relation for the join. As usual, Figure 8 contains results for all three of the clustering alternatives.

Figure 8a presents the performance results from the 1:1 case. With NONE clustering, the CP-SORT join method performed the best, followed by the NL-SORT and CP methods, with NL-INDEX doing the worst here. Figure 8b shows the underlying Starburst I/O results. The measured fix counts (not shown) were as one would expect by now: CP-SORT performed the fewest page fixes, CP performed about 20% more fixes than CP-SORT, and the two non-IMS methods each performed more than twice as many fix operations as the CP method. This information explains the ordering of the algorithms in the NONE case. The CP-SORT method required the least number of I/Os since it accesses the parent records directly and in the right physical order, and it also did the fewest page fixes, so it performed the best. The NL-SORT method required somewhat more I/Os due to index page accesses, and its fix count was much higher. The CP method had a low fix count but did more I/Os due to random probing of the parent relation. The NL-INDEX method shares the disadvantages of both the CP and NL-SORT methods without their corresponding advantages, resulting in both the most I/Os and the most page fixes.

The CHILD clustering results in Figure 8a are fairly similar in nature to those of the NONE case. This is because the child relation's *unique2* index was used to perform the 10% child selection, and it is an unclustered index in both the CHILD and FULL cases. As a result, despite the relative ordering implied by the CHILD clustering alternative, parent records are accessed in random order (i.e., in the unclustered order in which child records are selected). Note that another result of this is that the overall performance for all of the methods is worse in the two relatively clustered cases. This is because child records are clustered according to their *thousand* values instead of their *unique2* values as they are in the NONE case.

The FULL clustering results in Figures 8a and 8b show behavior for the join methods that is somewhat similar to the CHILD case. Here, though, the two sort-based algorithms perform worse because they sort the selected child relation unnecessarily, while the remaining algorithms both benefit from the clustering of child records with their parent in the FULL case. The CP join method performs the best here, followed by CP-SORT, NL-INDEX and lastly NL-SORT.

Figures 8c and 8d present the corresponding results for the 10:1 case. The relative performance of the algorithms for the three clustering alternatives is close to that of the 1:1 case, but the differences between the four join methods are somewhat more pronounced due to the larger child relation size. In particular, since the join is larger here, both sort-based methods gain more of an advantage by optimizing the order in which parent records are accessed. A related effect is evident for the non-IMS methods in the FULL case in Figures 8c and 8d. NL-SORT slightly outperforms NL-INDEX there because, although parent records are clustered with their children, an index I/O advantage is obtained by probing the *unique2* index of the parent in order (like NL-SORT) rather than randomly (like NL-INDEX). This was a non-issue in the 1:1 case because there each parent record was retrieved by only one child instead of an average of ten child records.

As discussed earlier, in addition to the results that we presented, we also ran each of the select-join queries with a 1% selectivity. Those results added no new insight beyond what we have described, and their execution times were a bit less stable because they were closer to the limits of our measurement tools and test environment. Thus, in the interest of space, we simply summarize them here. All of the trends in the 1% selectivity runs matched those of the corresponding 10% queries. The IMS join methods provided similar fix count savings and even better I/O savings in each case tested. The elapsed time differences were slightly less pronounced for the 1% queries, however. This was because the startup cost for queries began playing a measurable role in determining overall performance, thus reducing the net impact of the I/O and fix count savings. Just as we found in the 10% tests, the best performance for each of the 1% tests was provided by one of the IMS join methods.

5. LESSONS ON STARBURST EXTENSIBILITY

In addition to being a study of pointer-based access path support for joins in a relational DBMS, the IMS attachment effort was also a first attempt by "outsiders" (the first two authors) to extend the Starburst system. In this section of the paper, we reflect on lessons about storage-level extensibility that came out of this work.

As described in Section 2, the Starburst extension architecture is based on a model where each relation is physically stored by an instance of some particular storage method and can have zero or

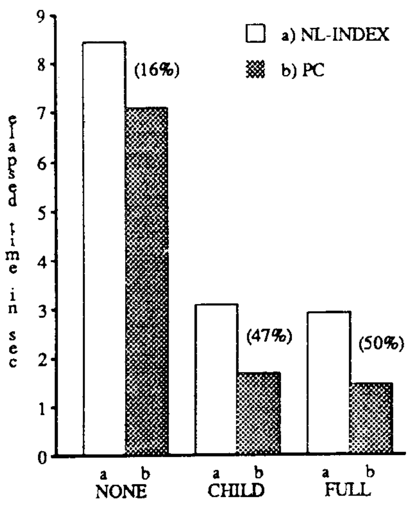

Figure 7a: 10% Parent Select Time (1:1).

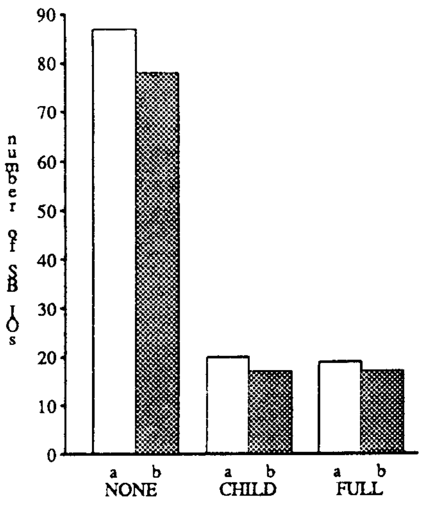

Figure 7b: 10% Parent Select I/O (1:1).

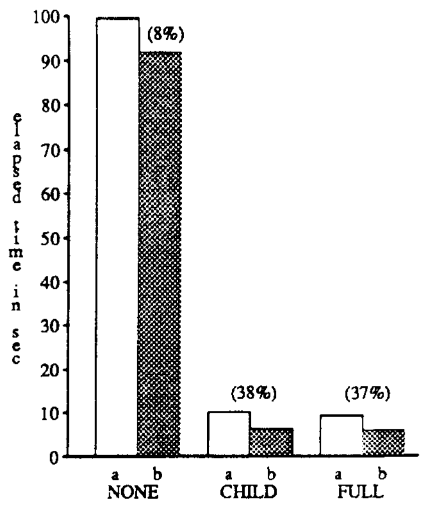

Figure 7c: 10% Parent Select Time (10:1).

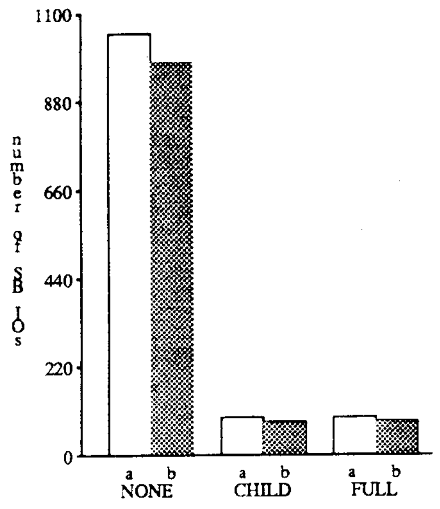

Figure 7d: 10% Parent Select I/O (10:1).

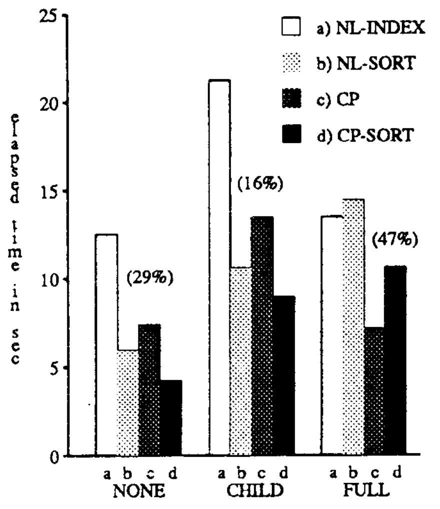

Figure 8a: 10 % Child Select Time (1:1).

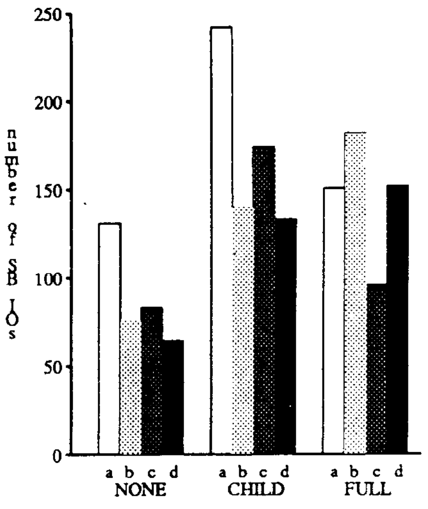

Figure 8b: 10% Child Select I/O (1:1).

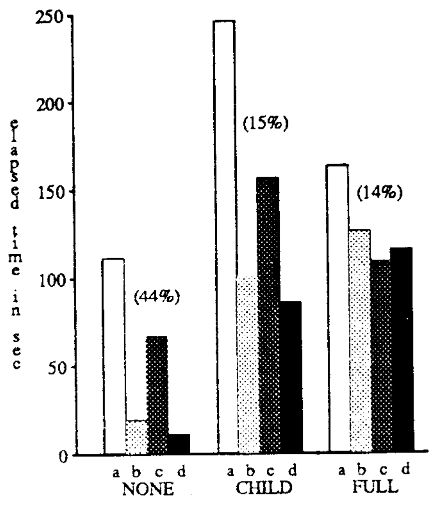

Figure 8c: 10% Child Select Time (10:1).

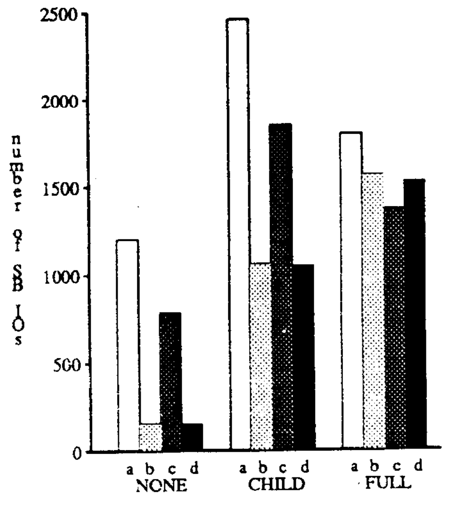

Figure 8d: 10% Child Select I/O (10:1).

more attachment instances associated with it. The model of an attachment, at least as implemented, was that of a component attached to a single relation; each time a storage method was called to update a relation, each of the relation's attachments was given an opportunity to react (after the fact) to the change. Also, when Core was designed, storage methods and attachments were assumed to be independent of one another. The needs of the IMS attachment violated both of these assumptions in ways that required certain changes to be made to the generic portion of the Core architecture.

The IMS attachment type is an example of an attachment that spans a pair of relations rather than being attached to a single (or distinguished) relation. It is not difficult to think of additional functions that one might wish to add to Starburst where multi-relation attachments would be needed. Examples include attachments to implement a join index, to enforce general N-relation integrity constraints, to monitor condition satisfaction for N-relation production rules, to incrementally maintain materialized join views, or even to manage replicated data in a distributed DBMS context. Unfortunately, the Starburst data definition facilities currently recognize just one relation as the one to which a given attachment instance is "officially" attached; this is the attachment point visible in the user-level catalog that keeps track of attachments. Fortunately, Core contained no barriers preventing more than one attachment point from actually being used (as long as just one was the "official" one).

The IMS routines for processing child relation updates need to take certain actions before the updates actually take place, as described in Section 3. In some cases this is an efficiency issue, such as avoiding the need to reread and then modify a child record that was just inserted or updated. In other cases, it is truly a necessity. For example, IMS clustering can only be supported if IMS attachments can provide hints prior to storage method insertions. Other attachments with similar requirements are easy to imagine. Any attachment that wishes to influence clustering, such as a clustered index or complex object manager, must be allowed to provide pre-insertion clustering hints. Any attachment that needs to set or modify record fields before they are written, or perhaps even when they are read, would also benefit from pre-operation action support. Examples here could include attachments to perform data compression or encryption on the contents of a relation, or an attachment that extends the records of a relation with additional hidden columns for purposes such as materializing virtual fields or replicating data [Shek89]. When we began work on the IMS attachment, Core provided only post-operation opportunities for attachments to be notified of changes. Corresponding pre-operation calls were added in response to the needs of the IMS effort.

In designing the IMS attachment, we quickly discovered that storage methods and attachments are not always as independent as they were initially thought to be. We found two ways that a new type of attachment can depend on the storage method of a relation or another of its attachments. The first way is for an attachment to depend on certain properties of the storage method or attachments. These can either be properties of the storage method or attachment type, or they can be properties of one particular instance. The second way is for an attachment to depend on the continued existence of another attachment on the same relation. Examples of such dependencies that we encountered were:

(1) In order to create an IMS attachment on a relation with CHILD clustering, it is necessary for the child relation's storage method to support record clustering based on the Core "near hint" facility. This is a property of the child relation's storage method type that is checked whenever creation of a new IMS instance with CHILD clustering is attempted.

(2) In order to create an IMS attachment on a relation with FULL clustering, the parent and child storage methods must permit records from the two relations to be clustered together on disk. Whenever creation of a new IMS instance with FULL clustering is attempted, it is thus necessary to verify that the two relations are stored together in a common Starburst database space and that the storage method for this shared database space indeed supports clustering. These illustrate both instance-specific ("are these two relations stored together?") and type-specific ("is clustering supported?") checks of storage method properties.

(3) For the IMS attachment to function properly, it requires the parent relation to have a permanent attachment (the **parentindex** in the syntax for creating an IMS attachment) that supports efficient lookups on the parent key and ensures that the parent key is unique over the parent relation (to ensure many-to-1-ness). Another create-time IMS check thus verifies that the specified parent index meets these requirements. In addition, the IMS attachment needs to guarantee the continued existence of the parent index by not allowing it to be dropped for as long as the dependent IMS instance exists.

To deal with these dependencies, the Core architecture had to be extended with a mechanism to permit storage methods and attachments to communicate relevant type and instance specific properties. As discussed in [Haas90], the Core implementation uses vectors of function pointers to make it easy to hook in new storage methods and attachments — once the code for a new component is functional, one simply makes entries for the new component in each of a number of function vectors (two per operation, e.g., pre- and post-operation routines for insert record, delete record, or fetch record). Following this model, we added a set of storage method and attachment property vectors to permit properties to be registered with the system. For type-specific properties, there is a vector of boolean values for each property of interest; the value entered for a particular storage method or attachment type is either true or false depending on whether or not it has the property. For instance-specific properties, the vectors contain pointers to functions with boolean return values. Finally, to handle the fact that an attachment can depend on the continued existence of another attachment instance, we added a facility whereby all of a relation's attachments are notified when any of them are dropped, thereby giving them an opportunity to veto a drop operation.

In light of the above discussions, one might begin to wonder whether or not the Starburst extension architecture was initially well-designed. It is the authors' opinion that the IMS attachment effort is in fact quite a strong testimonial in favor of the architecture. Despite the issues raised above, the IMS attachment was designed and implemented by a team of two "outsiders" in less than 12 weeks — even though they had no prior exposure to Starburst aside from reading conference papers about the system. Moreover, several of the changes needed in the generic portion of the Starburst code were made by the outsiders themselves, including the addition of an "alter table" facility for adding the hidden pointer fields needed for the IMS data structure. Finally, virtually all of the necessary changes had a fairly localized impact on the existing Core code.

6. CONCLUSIONS

In this paper, we have described the design and implementation of the Starburst IMS attachment type. The significance of this work is two-fold. First, it is a demonstration of how pointer-based join access paths and inter-relation record clustering can be naturally incorporated into a relational DBMS in the context of referential integrity information. Second, it is the first objective demonstration of the extensibility provided by the Starburst data management extension architecture.

In describing the IMS attachment's design and implementation, we described how updates are handled and we outlined three simple join algorithms that take advantage of the IMS pointer structure. Two were simply nested-loop join variants, and the third was a variant of the sort-merge join method. As demonstrated by our initial performance experiments, these algorithms can significantly reduce both the I/O cost and the page processing overhead required for computing referential joins as compared to the corresponding value-based join methods. In addition to this empirical evidence favoring the use of pointers for join computation, we have also studied pointer-based join methods from an analytical perspective in [Shek90]. There we propose a wider range of pointer-based join algorithms (including a hash-based join method) and provide a detailed analytical comparison of pointer- versus value-based joins. Compared to the empirical results presented here, the results of that analysis were similarly favorable.

Our experience with the IMS attachment turned out to be quite useful from the standpoint of testing the Starburst architecture. It pointed out the need to support multi-relation attachments, demonstrated a need for pre-operation routines to complement the post-operation routines that notify attachments when storage method updates occur, and pointed out the reality that storage methods and attachments are not always as independent as one might like them to be. Despite the minor changes that were required to accommodate the IMS attachment type, we were pleased to discover that such an attachment could indeed be added to Starburst in a relatively short period by "outsiders."

ACKNOWLEDGEMENTS

The work of the University of Wisconsin authors was partially supported by the Defense Advanced Research Projects Agency under contract N00014-88-K-0303 and by the National Science Foundation under grant IRI-8657323.

REFERENCES

[Astr76] M. Astrahan et al., "System R: A Relational Approach to Database Management," *ACM Trans. on Database Systems* 1(2), June 1976.

[Bitt83] D. Bitton, D. DeWitt, and C. Turbyfill, "Benchmarking Database Systems: A Systematic Approach," *Proc. 9th VLDB Conf.*, Florence, Italy, Nov. 1983.

[Rube87] W. Rubenstein, M. Kubicar, and R. Cattell, "Benchmarking Simple Database Operations," *Proc. 1987 ACM SIGMOD Conf.*, San Francisco, CA, June 1987.

[Cham81] D. Chamberlin et al., "A History and Evaluation of System R," *Comm. ACM* 24(10), Oct. 1981.

[Date81] C. Date, "Referential Integrity," *Proc. 7th VLDB Conf.*, Cannes, France, Sept. 1981.

[DBTG71] *Report of the* CODASYL *Data Base Task Group*, ACM, April 1971.

[Elma89] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*, Benjamin-Cummings Publishing Co., Redwood City, CA, 1989 (pp. 350-351).

[Haas89] L. Haas et al., "Extensible Query Processing in Starburst," *Proc. 1989 ACM SIGMOD Conf.*, Portland, OR, June 1989.

[Haas90] L. Haas et al., "Starburst Mid-Flight: As the Dust Clears," *IEEE Trans. on Knowledge and Data Engineering* 2(1), March 1990.

[Lind87] B. Lindsay, J. McPherson, and H. Pirahesh, "A Data Management Extension Architecture," *Proc. 1987 ACM SIGMOD Conf.*, San Francisco, CA, June 1987.

[Lohm88] G. Lohman, "Grammar-Like Functional Rules for Representing Query Optimization Alternatives," *Proc. 1988 ACM SIGMOD Conf.*, Chicago, IL, June 1988.

[Mart86] D. Martin, *Advanced Database Techniques*, MIT Press, Cambridge, MA, 1986.

[Shap86] L. Shapiro, "Join Processing in Database Systems with Large Main Memories," *ACM Trans. on Database Systems* 11(3), Sept. 1986.

[Schw86] P. Schwarz et al., "Extensibility in the Starburst Database System," *Proc. 1986 Int'l. Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, Sept. 1986.

[Shek89] E. Shekita and M. Carey, "Performance Enhancement Through Replication in an Object-Oriented DBMS," *Proc. 1989 ACM SIGMOD Conf.*, Portland, OR, June 1989.

[Shek90] E. Shekita and M. Carey, "A Performance Evaluation of Pointer-Based Joins," *Proc. 1990 ACM SIGMOD Conf.*, Atlantic City, NJ, June 1989.

[Vald87] P. Valduriez, "Join Indices," *ACM Trans. on Database Systems* 12(2), June 1987.