Concise Papers

Access to Indexed Hierarchical Databases Using a Relational Query Language

Chin-Wan Chung and Kenneth E. McCloskey

Abstract— **This paper presents an efficient means to access indexed hierarchical databases using a relational query language. The purpose of this paper is an effective sharing of heterogeneous distributed databases. We investigate 1) translation of hierarchical data definition to an equivalent relational data definition, 2) translation of a relational query language statement to an equivalent program processable by a hierarchical database management system, and 3) automatic selection of secondary indexes of hierarchical databases. A major portion of the result has been implemented and the performance of the implemented system is analyzed. The performance of the system is satisfactory for a wide range of test data and test queries. It is shown that the utilization of the secondary index significantly enhances the efficiency in accessing hierarchical databases.**

Index Terms—**Data model, data manipulation language, heterogeneous database, translation, hierarchical database, relational query language, index, performance.**

I. INTRODUCTION

An effective sharing of heterogeneous databases is essential to organizations with diverse database management systems (DBMS's). DATAPLEX [2] is a heterogeneous distributed DBMS which will provide a common view of diverse databases and a standard query language for organization-wide data sharing. In the architecture of DATAPLEX, a relational model is used to provide a common view of data and SQL is selected as a standard query language. A prototype DATAPLEX which interfaces an IMS hierarchical DBMS and an INGRES relational DBMS shows the feasibility of the DATAPLEX architecture.

An interface to IMS in the DATAPLEX environment is important because IMS has been the most heavily used mainframe DBMS in large corporations. In addition, the techniques to interface IMS can be applied to interface other nonrelational DBMS's such as network DBMS's and object-oriented DBMS's.

IMS is based on a hierarchical data model and uses DL/1 as a data manipulation language. While SQL is a high level query language, DL/1 is a set of subroutine calls that must be embedded in a host language, such as PL/1 or COBOL. Therefore, the interface to a hierarchical DBMS poses a difficult translation problem. Another major issue is the use of IMS secondary indexes. The secondary index is an integral part of data management. The secondary index management of IMS is quite different from that of the relational DBMS.

Due to the need for a rapid prototyping, the prototype DATAPLEX uses DXT [6] as an SQL to DL/1 translator. DXT is an IBM product which translates a subset of SQL to DL/1. The experience of DXT in the prototype DATAPLEX leads to a conclusion that the performance of DXT is not adequate and the subset of SQL supported by DXT is too restrictive for a full-function DATAPLEX. Therefore, we decided to develop an SQL interface to IMS. This interface is called SQL/IMS.

Based on the research results presented in this paper, the detailed design of SQL/IMS has been completed and a major portion of the design has been implemented. Benchmarks were prepared to test various features and the performance of SQL/IMS. The comparison of performance of SQL/IMS, DXT, and PL/1 programs was conducted using a large production database.

In Section II, the translation of IMS data definitions to equivalent relational data definitions is presented. The translation of an SQL query to a DL/1 program and the execution of the program are discussed in Section III. Section IV explains the automatic selection of IMS secondary indexes. Section V describes the implementaion of SQL/IMS and the result of benchmark executions.

Manuscript received December 22, 1989; revised December 7, 1990.

The authors are with the Computer Science Department, General Motors Research Laboratories, Warren, MI 48090-9057

IEEE Log Number 9205834.

II. DATA DEFINITION TRANSLATION

A. Hierarchy to Relation Mapping

Each IMS segment type is mapped to a relation. The relation corresponding to a segment type will include the key attributes of each predecessor of the segment type in the hierarchy. This is intended to produce a simple and normalized set of relations. This method is illustrated in Example 1 in Section IV. Relations defined on the dependents of nonkey segments do not include any attributes from the nonkey segments. The navigation through access paths to access multiple segments can be accomplished by equijoining relations in an equivalent relational data definition on common key attributes.

The repeating group is an exception to the general mapping rules. Repeating groups are common in IMS segment definitions. Since the relational model has no provision for repeating groups they have to be mapped to appropriate relational data structures. In general, a repeating group can be mapped to an additional relation. Consider a segment type $X(kX, fX, fG, fH)$, where kX is a list of key fields of segment type X, fX is a list of nonkey fields of X, and fG and fH are the list of fields of repeating group G and H, respectively. The equivalent relational data definition of this segment type consists of the following three relations:

$$RX(\boldsymbol{kRX}, fX), RG(\boldsymbol{kRX}, fG), RH(\boldsymbol{kRX}, fH)$$

where kRX is the key of RX including the keys of all the predecessors of X in the hierarchy. If X is a root segment type, $kRX = kX$.

Certain repeating groups, however, cannot be mapped to additional relations as follows:

- The position of a repeating group occurrence is significant. For instance, SALES repeating group with the nth occurrence indicating the sales figure of the nth month of a year. Such a repeating group cannot be mapped to a relation because the order of tuples in a relation is immaterial.
- The relation mapped from a segment type containing a repeating group does not have a key. In case of the segment type X above, the relationship of which occurrences of the repeating group G belong to which occurrence of the segment X can be identified by joining RG and RX on kRX because kRX is the key of RX. If RX did not have a key, the relationship between the occurrences of X and G cannot be identified.

Therefore, these repeating groups are mapped in a simple linear form. For example, a repeating group G, with fields $f_1, f_2, \cdots, f_m$, which can have a maximum of n occurrences would map as $G_1, G_2, \cdots, G_n$, where $G_i = a_{i1}, a_{i2}, \cdots, a_{im}$ with relational attributes a_{ij} for $j = 1, 2, \cdots, m$.

B. Source of Mapping Information

The information on IMS data definition necessary for translation comes from various sources. In this subsection, we explain three important data description features of IMS: the database definition, the program specification block, and the program communication block. IMS databases are defined in the database definitions (DBD's). The DBD's supply all the physical description information for IMS to create the database, such as access methods, storage devices, segment size/ hierarchical position, and segment key attribute descriptions. The segment nonkey attributes are usually defined in the application programs that use the database.

Access to an IMS database is always from a procedural (i.e., COBOL, Assembler, PL/1 or "C") language program. In order for the program to communicate with IMS, an interface control block called the program specification block (PSB) must be defined. Within the PSB, access to multiple IMS databases may be provided by defining one program communication block (PCB) per database access required.

Although IMS primarily supports a hierarchical structure, network style structures are possible via IMS logical databases (which are actually physical links). Existing physical IMS hierarchies are connected using physical pointers to form new "logical" hierarchies.

This additional structure could cause a mapping problem as not all IMS structures would be hierarchies. Fortunately, there is a simple solution; mapping is always based on the PSB. Since all IMS databases described in a PSB present a consistent data structure (a hierarchy), regardless of the underlying physical data strucure, this is the most reasonable point from which to map.

The mapping information collected from the above indicated sources is stored in two files: the physical definition table (PDT) and the view definition table (VDT). The PDT contains all the IMS access information for each attribute. The VDT contains a definition of the relation as seen by the user. This provides three main features: attributes names to be aliased, attribute type coercion, and relation subsets to be formed.

III. Query Translation and Execution

Before discussing query translation and execution, the level of SQL to be considered must be defined. One of the major issues in developing an SQL interface to IMS is the incompatibility between SQL and DL/1. SQL is a set (set of records) oriented language, whereas DL/1 is a record-oriented language.

SQL has more features than DL/1 and consequently there is no DL/1 feature to which certain SQL features can be translated. These SQL features must be implemented without using DL/1 features. However, some of the SQL features are seldom used in applications. In fact, the most frequently used SQL features are the ones corresponding to relational operations selection, projection, and equijoin. These three operations can be translated to DL/1 features, and equijoin can especially be processed very efficiently using DL/1 because, in many cases, the relationship between two entities is implemented by pointers in IMS databases.

Based on the above facts, we have defined a subset of SQL to be used by SQL/IMS with the following guiding principles:

- include all SQL features that can be translated to DL/1 features to make use of efficient IMS capabilities;
- include SQL features that are frequently used in applications so that most of the applications can be covered by the defined SQL subset;
- exclude SQL features that are seldom utilized to avoid the implementation of a relational DBMS and the runtime overhead of a large system.

Included in our SQL subset are:

SELECT — with DISTINCT, ORDER BY, equijoin, attribute expressions, set functions (MAX, MIN, AVG, SUM, COUNT);
UPDATE — ANSI syntax [1] with limited search conditions (see excluded below);
INSERT — ANSI syntax from value list only, subselect is not supported;
DELETE — ANSI syntax with limited search conditions (see excluded below).

Excluded in our SQL subset are:

- HAVING, GROUP BY, nested queries and non-equijoins in SELECT statements;
- For all statements (SELECT, UPDATE, INSERT, DELETE), NULL and EXISTS are not supported in WHERE clauses.

A nested query can be transformed to an equivalent non-nested query. All the above SQL statements can be implemented for interactive and embedded interfaces. Thus transactions are supported. All the usual embedded cursor commands (e.g., DECLARE, OPEN, CLOSE, WHERE CURRENT OF, and FETCH) are included.

An SQL query which uses unsupported SQL features is decomposed by DATAPLEX into two queries: $Q1$ (Query 1) contains supported SQL features, and $Q2$ is the remaining part of the original query. Then, $Q1$ is sent to SQL/IMS which translates it to a DL/1 program and submits the program to IMS. $Q2$ and the result of $Q1$ are fed to a relational DBMS (which may be at a remote location). The result of $Q2$ is the result of the original query.

Our approach to process an SQL query submitted to SQL/IMS is to decompose the query into simpler queries that can be processed by IMS and then perform remaining relational operations on the intermediate results retrieved from IMS to obtain the final result. There are two ways to translate a decomposed simple SQL query to a DL/1 program: 1) to generate DL/1 code on the fly, and 2) to generate parameter values from the SQL query and feed them to a fixed parameterized DL/1 program. We take the second approach for its simplicity and efficiency.

The process of query decomposition/translation involves breaking the query down into units simple enough to be executed by an IMS application program and converting them to an acceptable format. For the sake of brevity this process will be referred to as translation. There are seven steps of translation that all queries must pass through prior to execution. Each step either removes functions that are not processable by IMS or splits the query into subqueries that reference a smaller range of data. For each act of translation imposed on a query, a complementary recomposition query is generated to be executed once the data has been retrieved from IMS. The seven steps of translation are as follows:

1) remove operations in target lists nonsupported by IMS, such as set functions;
2) convert predicates to disjunctive normal form;
3) remove expressions from predicates of resulting conjunctive queries;
4) convert queries to graphs by representing a relation as a node and a search condition as an arc;
5) decompose by database because an IMS query (DL/1 call) may reference only one database;
6) decompose by cutting the equijoin arcs that do not correspond

to equijoins processable by navigating access paths between IMS segment types. We call this type of equijoin a pathjoin;

7) select an IMS secondary index on attributes in a subquery. This step is explained in detail in Section IV.

At this point the subqueries are processable by the IMS interface program, the fixed parameterized DL/1 program. A detailed description of steps 1)–6) can be found in [3].

After translation, the query is executed and the result recomposed. The query graph of a subquery contains the necessary parameter values for the fixed parameterized DL/1 program. The individual subqueries are passed to the IMS interface program for execution. Once all the subqueries have executed, the recomposition queries are applied against the retrieved data in the reverse order in which they were created during translation as follows:

1) non-pathjoin equijoin;
2) qualification clause expression;
3) union of results from conjunctive subqueries;
4) DISTINCT;
5) set Function;
6) target list expression;
7) sorting, i.e., ORDER BY.

IV. Index Selection

IMS provides the capability to define mulitiple indexes for a database. In general, IMS databases have at least one index called the primary index that is defined as part of the initial database definition. The primary index describes the actual physical ordering of the database segments. Additional indexes may be defined providing alternate access paths to database segments. These are called secondary index databases.

There are three problems in providing automatic index selection.

1) In order to utilize a secondary index, the IMS application must use a PCB (within the target PSB) that specifies the secondary index database for its processing sequence. In the mapping provided in the PDT, the primary index PCB is defined. Therefore, the mapping must be dynamically altered to take advantage of a secondary index.
2) IMS does not directly support composite keys. It is quite common in IMS databases for a composite key to be defined to IMS as a single character string which is then broken down into its component attributes by the application program.
3) A secondary index database that indexes a segment of the hierarchy other than the root segment will cause the hierarchy to be inverted with the indexed segment assuming the root position. The hierarchical mapping information stored in the query graph must be dynamically altered to correctly depict the new access path.

Each subquery is tested for potential substitution of an index for its component attributes. (An index and an alternate key on which the index is set are used interchangeably.) This is done by using a table of indexes, available via the target PSB, called the segment index table (SIT). In the default mapping provided in the PDT, the primary index PCB will be selected, if 1) primary key search is used or 2) no key search qualification is specified (i.e., scan the entire database).

For each index, the SIT contains the three mappings required to transform the query: PCB, composite key, and hierarchy. The PCB map gives the name of the PSB, the PCB position within the PSB, and the name of the segment type that is the target of the index. The composite key map describes the attributes that form the key of the target segment type to the key attribute name as it is defined in IMS. The hierarchy map describes the relationships of the segment types that form the structure created through the use of the index.

There are two restrictions imposed in using indexes. The first is that only one table of the subquery may be selected for index substitution. This is due to the IMS requirement that database access is managed via a PCB and that a PCB may only reference a single index. Therefore, all the tables of a subquery must be accessed through the same PCB. It is an inherent disadvantage of the hierarchical database such as IMS that multiple indexes cannot be used simultaneously. However, this disadvantage is offset by the advantage that the reduction of search for a segment using an index reduces search for all segments connected by access paths to the segment.

Secondly, the substitution of an index for its component attributes is limited. Consider a composite index I consisting of attributes A and B. Suppose A and B take values from characters a to z for simplicity. The qualification clause (A = u AND B = v) is equivalent to (I = uv). However, the qualification clause (A > u AND B > v) is not equivalent to (I > uv). An equivalent qualification clause generated from the two conditions is (I > uz AND (au < I <= az OR bu < I <= bz OR $\cdots$ OR zu < I <= zz)). As the number of attributes comprising an index grows, the number of conditions generated in a qualification clause using the index grows exponentially, which makes the manipulation and use of the qualification clause impractical. Therefore, complete equality comparisons are allowed. All others are partially transformed except that not equal (i.e., <>) comparisons are not transformed.

The stratgey of our index selection algorithm is discussed below. For each segment type referenced in a subquery, the SIT is searched for an index to that segment. For each index found, the subquery search conditions are tested for qualifications that refer to the attributes that comprise the index. For an index to become a candidate for substitution, the left-most attribute of the index must be referenced in a qualification clause. If all the attributes of an index are referenced and the comparison operators on the attributes are all "=," then all the attributes are used for an index substitution. If i attributes from the left of an index are referenced with the comparison operator "=," then the i attributes are used. Otherwise, only the left-most attribute of the index is used. The attribute(s) usable for an index substitution are called qualified attribute(s). The index containing qualified attributes, the sum of the lengths of which is the longest (in bytes), is selected to reduce search as much as possible. If, after all the SIT's have been checked, a suitable index was found, then the query graph is modified. The query qualification is tested again for additional substitutions using the selected index. This is necessary as a user may specify a range qualification (i.e., A > 0 AND A < 100). The test and substitute process is repeated until no more substitutions are possible. The algorithm for index selection is as follows.

Algorithm I

```
Input: the query graph and SIT.
Output: revised query graph and new PCB

initialize F, the final index selected, to be
 null.
initialize FL, the length of the final index
 selected, to 0.

for each relation, Ri, referenced in the graph:
  for each element, Sj, of the SIT:
    if Sj is not an index for the target segment
     referenced by Ri
    then skip to the next Sj.
    the index represented by Sj becomes the can-
     didate index, C.
    using C, compute the sum of the lengths,
```

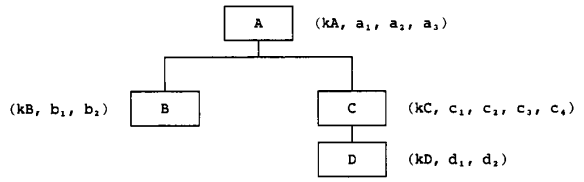

Fig. 1. Example IMS database hierarchy.

CL, of the qualified attributes.
if $CL > FL$ then assign C to F and CL to FL.

if $FL > 0$
for each relation in the query graph, rebuild the hierarchical information that describes the relative position of the segment mapped to this relation.

change the default PCB to the PCB that provides F.

while $FL > 0$
modify the query graph by substituting the index for the component attributes as follows:
;suppose F consists of component attributes $a_1, a_2, \ldots, a_n$, where
;a_i is the ith component from the left. MAX denotes the patching
;of the unused bits with binary 1 and MIN with binary 0.

Case 1: modify $(a_1 = x_1$ AND $a_2 = x_2$ AND $\cdots$ AND $a_n = x_n)$ to $(F = x_1 x_2 \ldots x_n)$ where x_j is the value of a_i.
Case 2: modify $(a_1 = x_1$ AND $a_2 = x_2$ AND $\cdots$ AND $a_i = x_i$) with $i < n$ to $(F >= x_1 x_2 \cdots x_i$ || MIN AND $F <= x_1 x_2 \cdots x_i$ || MAX) where || denotes concatenation.
Case 3: modify $(a_1 > x_1)$ to $(F > x_1$ || MAX).
Case 4: modify $(a_1 >= x_1)$ to $(F >= x_1$ || MIN).
Case 5: modify $(a_1 < x_1)$ to $(F < x_1$ || MIN).
Case 6: modify $(a_1 <= x_1)$ to $(F <= x_1$ || MAX).

if there is no qualified attribute for F in the query qualification, update FL to 0.
End Algorithm I

The following example illustrates the Algorithm I.

Example 1: Consider the IMS database hierarchy shown in Fig. 1. Segment attributes are shown in parentheses. Primary key fields are denoted as kX ("key of X").

The relational mappings for the segment types in Fig. 1 are depicted in Fig. 2. IMS secondary indexes are denoted as $I(X)$ ("index of X"). The length of each attribute in bytes is also shown.

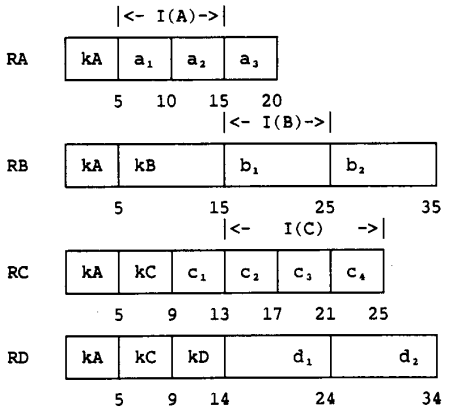

Fig. 2. Relational data definition of segment types in Fig. 1.

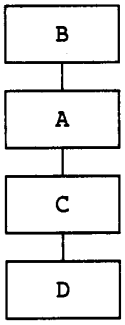

Fig. 3 Changed hierarchy for $I(B)$.

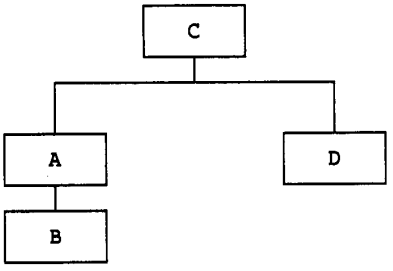

Fig. 4. Changed hierarchy for $I(C)$.

As explained previously, the use of an IMS secondary index database may alter the hierarchy. For the indexes shown in Fig. 2, the resulting hierarchies for $I(B)$ and $I(C)$ are shown in Figs. 3 and 4, respectively. $I(A)$ does not change the hierarchy.

Assume a PSB has been created with the first PCB assigned to the original database and the subsequent three PCB's assigned to indexes $I(A)$, $I(B)$, and $I(C)$, respectively. For each of the primary and secondary indexes shown previously, their respective SIT entries are given in Table I.

Suppose the following query is issued:

SELECT *
FROM RA, RC

TABLE I
SEGMENT INDEX TABLE (SIT) FOR EXAMPLE 1

#	Index	Segment	PCB	Hierarchy	Component Attributes (name, length)
1	kA	A	1	Fig. 1	$< (kA, 5) >$
2	kA, kB	B	1	Fig. 1	$< (kA, 5)(kB, 10) >$
3	kA, kC	C	1	Fig. 1	$< (kA, 5)(kC, 4) >$
4	kA, kC, kD	D	1	Fig. 1	$< (kA, 5)(kC, 4)(kD, 5) >$
5	$I(A)$	A	2	Fig. 1	$< (a_1, 5)(a_2, 5) >$
6	$I(B)$	B	3	Fig. 3	$< (b_1, 10) >$
7	$I(C)$	C	4	Fig. 4	$< (c_2, 4)(c_3, 4)(c_4, 4) >$

```
WHERE RA.kA = RC.kA
  AND a1 > x
  AND c2 = y
  AND c3 = z
  AND c4 <> u
```

After parsing, the query would be translated to a query graph that would be passed to the Algorithm I. The processing steps of the algorithm are shown in the following.

```
for the relation, RA
  SIT #1 is for segment A
    kA becomes the candidate index C
    CL becomes 0 because kA is not a qualified
          attribute
    CL is not greater than 0
  SIT #2 is for segment B
  SIT #3 is for segment C
  SIT #4 is for segment D
  SIT #5 is for segment A
    I(A) becomes the candidate index C
    CL becomes 5 because a1 is a qualified
     attribute
    CL is greater than 0 so assign I(A) to F
     and 5 to FL
  SIT #6 is for segment B
  SIT #7 is for segment C
for the relation, RC
  SIT #1 is for segment A
  SIT #2 is for segment B
  SIT #3 is for segment C
    kA, kC becomes the candidate index C
    CL becomes 0 because kA is not a qualified
     attribute
    CL is not greater than 5
  SIT #4 is for segment D
  SIT #5 is for segment A
  SIT #6 is for segment B
  SIT #7 is for segment C
    I(C) becomes the candidate index C
    CL becomes 8 because c2, c3 are qualified
     attributes
    CL is greater than 5 so assign I(C) to F
     and 8 to FL

FL is greater than 0
   rebuild the hierarchical information for RA
    and RC using the hierarchy in Fig. 4
   change the PCB from 1 to 4

by Case 2: modify (c2 = y AND c3 = z) to
             (I(C) >= yz || 0000' AND
              I(C) <= yz || 'FFFF')
```

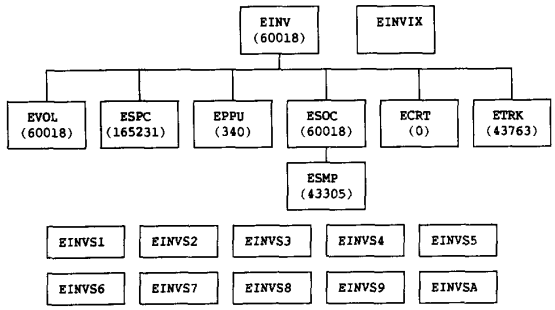

Fig. 5. Structure of MMIS database (number of occurrences).

```
there are no more qualified attributes for
 I(C), change FL to 0.
```

The modified query is as follows:

```
SELECT *
  FROM RA, RC
 WHERE RA.kA = RC.kA
   AND a1 > x
   AND I(C) >= yz || '0000'
   AND I(C) <= yz || 'FFFF'
   AND c4 <> u
```

V. IMPLEMENTATION AND TESTING

The architecture of SQL/IMS consists of the query decomposer/translator, the IMS interface program, and the subresult recomposer. In addition, SQL/IMS includes modules to interface a database protocol such as RDA [8]. These modules are used to make a connection between a database protocol and SQL/IMS, and to exchange commands and data between a client process and SQL/IMS through the database protocol.

Currently, the retrieval part of the SQL subset described in Section III has been implemented, except for the embedded query interface. SQL/IMS is mostly in C [5] with some assembler. SQL/IMS is installed on an IBM 4381 S91E (Model 23) which runs MVS/ESA. A large production IMS database is used to test the performance of SQL/IMS. It is a part of the production Maintenance Management Information System (MMIS) obtained from a plant automation development group. The MMIS database contains maintenance scheduling information for plant equipment.

The data structure diagram of the MMIS database is shown in Fig. 5 with the number of occurrences for each segment type. The segment types, which are not in the hierarchy, are for index databases. EINVIX is the primary index for EINV. All other indexes are the secondary indexes for EINV, except for EINVSA which is the secondary index for EPPU. A part of the equivalent relational data definition of the MMIS database is provided in Appendix A.

Sixteen SQL queries are formulated to test the performance of SQL/IMS for various types of access to IMS databases. The queries 1 to 8 can be found in [3]. Some representative queries are listed in Appendix A. In addition, it is compared with the performance of DXT and PL/1. PL/1 programs equivalent to the test SQL queries were written to access the MMIS database using DL/1. The comparison of the CPU time for SQL/IMS, DXT, and PL/1 is shown in Table II. Due to the lack of functionality, DXT could not process some of the queries. The corresponding entries are marked with "*." For the queries that are successfully executed by all of the systems, the three systems produced the same number of result records for each request.

TABLE II
COMPARISON OF PERFORMANCE OF SQL/IMS, DXT, AND PL/1 PROGRAM AGAINST A LARGE PRODUCTION DATABASE

Request	CPU Time (in seconds)			# Records in Result
	SQL/IMS	DXT	PL/1	
1	36.049	35.382	25.026	5
2	35.823	36.893	25.141	539
3	0.741	4.142	0.280	165
4	65.263	99.028	47.429	5882
5	46.666	82.687	35.715	978
6	1.584	6.087	0.906	244
7	0.802	*	0.210	91
8	59.763	*	41.951	5033
9	0.458	36.901	0.204	1
10	0.459	40.588	0.205	1
11	1.237	36.367	0.734	224
12	0.440	36.204	0.214	7
13	25.708	53.051	15.930	7459
14	0.432	39.454	0.212	4
15	0.481	54.837	0.206	3
16	0.511	59.495	0.205	3

TABLE III
COMPARISON OF PERFORMANCE OF TWO VERSIONS OF SQL/IMS

SQL Query	SQL/IMS with Index Support		SQL/IMS without Index Support	
	CPU	DL/1 Calls	CPU	DL/1 Calls
1	36.049	60 019	35.712	60 019
2	35.823	60 019	36.448	60 019
3	0.741	166	0.744	166
4	65.263	43 306	65.688	43 306
5	46.666	43 764	47.302	43 764
6	1.584	245	1.552	245
7	0.802	92	0.814	92
8	59.763	60 019	61.133	60 019
9	0.458	2	37.753	60 019
10	0.459	2	51.834	60 019
11	1.237	225	36.259	60 019
12	0.440	8	35.681	60 019
13	25.708	7460	46.946	60 019
14	0.432	8	38.283	58 689
15	0.481	4	19.747	341
16	0.511	4	19.802	341

The queries 9 to 16 are used to test the access by using secondary indexes. These queries reference attributes on which IMS secondary keys are defined. An index is created on a secondary key which may be a composite key. For example, for Query 9 in Appendix A, a composite IMS key "xeinvsi1" is defined on attribtes "plantsi1," "system," "equip," and "comp." In processing this query, SQL/IMS modifies the qualification clause to xeinvsi1 = "to 111-10 sta 01" so that IMS can search using an index on xeinvsi1.

In order to evaluate the effect of supporting the secondary index, the performance of two versions of SQL/IMS are compared: 1) one supporting the secondary index (the same version used to produce the content in Table II), and 2) the other which does not support the secondary index. The comparison of these two versions is given in Table III. Table III shows that the queries run much faster by using the secondary index, except for Query 13. Since Query 13 requires 7460 DL/1 calls in retrieving records using an index from EINV segment with a total 60 018 records, the use of index is not much faster than the scan, and it incurs an overhead of index search. However, it is unlikely that realistic queries retrieve a large portion of a large database.

VI. CONCLUSIONS

Translation methods for data definitions and queries were developed to access hierarchical databases using a relational query language. Another major issue settled was an automatic selection of IMS secondary indexes. Based on the methods developed, an SQL interface to IMS, called SQL/IMS, was implemented.

The performance of SQL/IMS was analyzed using a large production database. The performance of SQL/IMS was compared with those of DXT, which is IBM's SQL to DL/1 translator, and the PL/1 program. The host language program (e.g., PL/1, COBOL) is the fastest way to access IMS data and thus the performance of the PL/1 program is the theoretical limit of the performance of any query language interface to IMS. Compared with the PL/1 program, SQL/IMS incurs fixed overhead and variable overhead which is a function of the size of the database being accessed. For accessing a large IMS database, the percentage of the overhead in the processing time is small.

Two versions of SQL/IMS, with and without the secondary index support, were compared to examine the effect of the secondary index support. It was shown that the ability to use the IMS secondary indexes was extremely important.

APPENDIX A

Relational Data Definition and SQL Queries for the MMIS Database

1) An equivalent relational data definition:

Since relations contain a large number of attributes, only the relations and attributes referenced in queries are listed.

```
EINV (EINVKEY, PLANTSI1, PLANTSI3, SYSTEM,
      EQUIP, COMP, ORIG_COST,
      BOOK_VALUE, DRAWING_SET, ACCOUNT,
      ACQUIRE_DDATE, COMMON, MAKE, MODEL)

EVOL (EINVKEY, PRD_DOWNYTD)

EPPU (EINVKEY, COMMON, MODELYEAR, MODELUS-
      AGE, MODELCOST, PLANT,
      SYSTEM, EQUIPMENT, COMP)
```

2) Representative SQL queries to the MMIS database:

```
8. select equip, orig_cost - book_value,
    prd_downytd
      from einv i, evol v
     where i.einvkey = v.einvkey
       and i.drawing_set >= '0015640'
     order by 2;

9. select einvkey, account, acquire_ddate,
    book_value
     from einv
    where plantsi1 = 'TO'
      and system   = '111-10'
      and equip    = 'STA 01'
      and comp     = ' ';

14. select einvkey, make, model, orig_cost,
     book_value
      from einv
     where plantsi3 = 'TO'
       and common   = '402-0823F'
       and einvkey >= 'TO 0001331';
```

```
16. select i.einvkey, book_value, orig_cost,
      modelusage, modelcost
       from eppu p, einv i
      where p.plant     = 'TO'
        and p.common    = 'FL000419'
        and p.system    = 'LAMB TRANS 1'
        and p.equipment = 'STA 04'
        and p.comp      = 'LH MECH DRIVE'
        and p.einvkey   = i.einvkey;
```

ACKNOWLEDGMENT

The authors would like to thank Jeff Hollar and Mike McGreevy for their participation in the development of SQL/IMS. They acknowledge the helpful comments of Relational Technology, Inc. personnel about the design of SQL/IMS.

REFERENCES

[1] ANSI X3H2-86-141, Database Language SQL2, Nov. 1986.
[2] C. W. Chung, "DATAPLEX:An access to heterogeneous distributed databases," *Commun. ACM*, vol. 33, pp. 70-80, Jan. 1990.
[3] C. W. Chung and K. E. McCloskey, "A relational query language interface to a hierarchical database management system," in *Proc. Second Int. Conf. on Data and Knowledge Systems for Manufacturing and Engineering*, Oct. 1989, pp. 105-112.
[4] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, pp. 377-387, June 1970.
[5] *C Language Manual (SC19-1128-0)*, International Business Machines Corporation, 1986.
[6] Data Extract Version 2.3: General Information (GC26-4241), International Business Machines Corporation, May 1988.
[7] *IMS/VS 1.3 General Information Manual (GH20-1260)*, International Business Machines Corporation, Mar. 1984.
[8] ISO/JTC 1/SC 2/WG 3N, Information Systems - Open Systems - Generic Remote Database Access Service and Protocol, Sept. 1988.
[9] T. Landers and R. L. Rosenberg, "An overview of Multibase," *Distributed Databases*, pp. 153-184, 1982.
[10] C. Zaniolo, "Design of relational views over network schemas," in *Proc. ACM SIGMOD Con.*, May 1979, pp. 179-190.

Join and Data Redistribution Algorithms for Hypercubes

Chaitanya K. Baru and Sriram Padmanabhan

***Abstract*— An important aspect of database processing in parallel computer systems is the use of data parallel algorithms. This paper presents several parallel algorithms for the relational database join operation in a hypercube multicomputer system. The join algorithms are classified as *cycling* or *global partitioning* based on the tuple distribution method employed. The various algorithms are compared under a common framework, using time complexity analysis as well as an implementation on a 64 node NCUBE hypercube system. In general, the global partitioning algorithms demonstrate better speedup. However, the cycling algorithm can perform better than the global algorithms in specific situations, viz., when the difference in input relation cardinalities is large and the hypercube dimension is small. We also study the usefulness of the data redistribution operation in improving the performance of the join algorithms, in the presence of uneven data partitions. Our results indicate that redistribution significantly decreases the join algorithm execution times for unbalanced partitions.**

***Index Terms*—Join algorithms, parallel algorithms, hypercubes, database systems, all-to-all communication.**

I. INTRODUCTION

Recently, the idea of using coarse-grained "shared-nothing" multicomputer architectures for database processing has been gaining momentum. In particular, the hypercube architecture [16] is an attractive candidate for many parallelization projects, including database processing, due to several useful properties of the hypercube topology and the availability of commercial systems based on this topology. Several parallel database projects have used or assumed the hypercube interconnection scheme. For example, the Bubba [2] project assumes that a hypercube system is capable of providing the communication bandwidth required for database processing, the HC16-186 [3] project uses a 16 node hypercube system, while the Gamma [5] database software has been ported to a hypercube system.

In a hypercube of N nodes, each node is connected to $\log(N)$ neighbors by direct links. One of the key issues in a hypercube system is to use the interconnection topology effectively for parallel operations. The parallel join operation requires *all-to-all* communication [8] in which each participating node sends a part or all of its data to every other node in the hypercube. Even with the advent of direct-connect communication technology, hypercube nodes must use efficient, coordinated algorithms to perform all-to-all communication [7], [14]. Allowing each node to route its data independently results in link contention and poor performance. In this paper, we describe join algorithms for the hypercube in which the nodes cooperate to perform the data communication efficiently while eliminating link and buffer contention problems.

Join algorithms have been proposed and evaluated for general parallel architectures by a number of researchers including [13], [20]. For hypercube systems, cycling-, broadcast-, and hash-based join algorithms have been proposed [1], [11], [19]. An algorithm based on the semi-join operation has been described and compared with broadcast join algorithms in [12]. We describe three hypercube join algorithms, each of which uses a different method to distribute the tuples of the joining tables. The *global sort* algorithm uses a parallel sorting method to distribute the tuples of both joining tables. The *global hash* algorithm uses a hashing (partitioning) function for this purpose, while the *cycling* method broadcasts the tuples of one table to all sites of the other table by embedding a ring in the hypercube. We compare the hypercube join algorithms under a common framework using simple O-notation analysis. Our analysis and experiments assume memory resident data. However, the same techniques for tuple distribution are applicable (and should be used) when data are stored in disks. Note that the emphasis in this paper is on the tuple distribution methods employed by the hypercube join algorithms.

The join algorithms have been implemented on a 64-node hypercube and results from a variety of experiments are presented. The experiments compare the performance of the join algorithms for different table cardinalities, unequal table sizes, and unbalanced table partitions. The results from these experiments indicate the trends in

Manuscript received January 19, 1990; revised March 14, 1991 and October 7, 1991. This work was supported in part by the National Science Foundation under Grant IRI-8710855.

The authors are with the Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, MI 48109.

IEEE Log Number 9207082.