A multiprocessor computer architecture for database support

Roger K. Shultz
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/rtd
Part of the Computer Sciences Commons

Recommended Citation
INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame. If copyrighted materials were deleted you will find a target note listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in "sectioning" the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.

University Microfilms International
300 N. ZEEB RD., ANN ARBOR, MI 48106
SHULTZ, ROGER K.

A MULTIPROCESSOR COMPUTER ARCHITECTURE FOR DATABASE SUPPORT

Iowa State University

Ph.D. 1981

University Microfilms International

300 N. Zeeb Road, Ann Arbor, MI 48106
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark ✓.

1. Glossy photographs or pages ☐
2. Colored illustrations, paper or print ☐
3. Photographs with dark background ☐
4. Illustrations are poor copy ☐
5. Pages with black marks, not original copy ☐
6. Print shows through as there is text on both sides of page ☐
7. Indistinct, broken or small print on several pages ✓
8. Print exceeds margin requirements ☐
9. Tightly bound copy with print lost in spine ☐
10. Computer printout pages with indistinct print ☐
11. Page(s) ________ lacking when material received, and not available from school or author.
12. Page(s) ________ seem to be missing in numbering only as text follows.
13. Two pages numbered ________. Text follows.
14. Curling and wrinkled pages ☐
15. Other ____________________________
A multiprocessor computer architecture
for database support

by

Roger K. Shultz

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of
the Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

Signature was redacted for privacy.
In Charge of Major Work
Signature was redacted for privacy.
For the Major Department
Signature was redacted for privacy.
For the Graduate College

Iowa State University
Ames, Iowa
1981
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER I. INTRODUCTION</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outline of Thesis</td>
<td>3</td>
</tr>
<tr>
<td>The Relational Model</td>
<td></td>
</tr>
<tr>
<td>Selection</td>
<td>6</td>
</tr>
<tr>
<td>Projection</td>
<td>7</td>
</tr>
<tr>
<td>Join</td>
<td>8</td>
</tr>
<tr>
<td>Division</td>
<td>9</td>
</tr>
<tr>
<td>Complex queries</td>
<td>10</td>
</tr>
<tr>
<td>Updates</td>
<td>12</td>
</tr>
<tr>
<td>Further comments</td>
<td>12</td>
</tr>
<tr>
<td>Background</td>
<td>13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER II. DATABASE COMPUTER ARCHITECTURES</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAFS</td>
<td>19</td>
</tr>
<tr>
<td>Selection</td>
<td>21</td>
</tr>
<tr>
<td>Projection</td>
<td>22</td>
</tr>
<tr>
<td>Join</td>
<td>23</td>
</tr>
<tr>
<td>CASSM</td>
<td>24</td>
</tr>
<tr>
<td>Search operations</td>
<td>29</td>
</tr>
<tr>
<td>Selection</td>
<td>29</td>
</tr>
<tr>
<td>Pointer search</td>
<td>30</td>
</tr>
<tr>
<td>Projection</td>
<td>31</td>
</tr>
<tr>
<td>Join</td>
<td>32</td>
</tr>
<tr>
<td>DBC</td>
<td>34</td>
</tr>
<tr>
<td>Selection</td>
<td>37</td>
</tr>
<tr>
<td>Projection</td>
<td>37</td>
</tr>
<tr>
<td>Join</td>
<td>38</td>
</tr>
<tr>
<td>RAP</td>
<td>38</td>
</tr>
<tr>
<td>Selection</td>
<td>40</td>
</tr>
<tr>
<td>Projection</td>
<td>41</td>
</tr>
<tr>
<td>Join</td>
<td>42</td>
</tr>
<tr>
<td>DIRECT</td>
<td>43</td>
</tr>
<tr>
<td>Selection</td>
<td>46</td>
</tr>
<tr>
<td>Projection</td>
<td>47</td>
</tr>
<tr>
<td>Join</td>
<td>47</td>
</tr>
<tr>
<td>HYPERTREE</td>
<td>48</td>
</tr>
<tr>
<td>Selection</td>
<td>50</td>
</tr>
<tr>
<td>Projection</td>
<td>50</td>
</tr>
<tr>
<td>Join</td>
<td>51</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER III. SYSTEM STRUCTURE</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database System Duties</td>
<td>53</td>
</tr>
<tr>
<td>User interface</td>
<td>55</td>
</tr>
<tr>
<td>Front-end machine duties</td>
<td>55</td>
</tr>
<tr>
<td>Back-end machine duties</td>
<td>56</td>
</tr>
<tr>
<td>Example 1: Small Selected Sets</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>CAFS</td>
<td>122</td>
</tr>
<tr>
<td>CASSM</td>
<td>123</td>
</tr>
<tr>
<td>DBC</td>
<td>123</td>
</tr>
<tr>
<td>RAP</td>
<td>125</td>
</tr>
<tr>
<td>DIRECT</td>
<td>125</td>
</tr>
<tr>
<td>REPT</td>
<td>125</td>
</tr>
<tr>
<td>HYPERTREE</td>
<td>126</td>
</tr>
<tr>
<td>Conclusions</td>
<td>126</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example 2: Larger Selected Sets</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAFS</td>
<td>128</td>
</tr>
<tr>
<td>CASSM</td>
<td>128</td>
</tr>
<tr>
<td>DBC and RAP</td>
<td>130</td>
</tr>
<tr>
<td>DIRECT</td>
<td>130</td>
</tr>
<tr>
<td>REPT</td>
<td>130</td>
</tr>
<tr>
<td>HYPERTREE</td>
<td>131</td>
</tr>
<tr>
<td>Conclusions</td>
<td>131</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Increasing Database Size</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effect of Increasing k</td>
<td>133</td>
</tr>
<tr>
<td>Direct Search Advantage</td>
<td>137</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER VII. CONCLUSIONS</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Future Work</td>
<td>143</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>REFERENCES</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>150</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACKNOWLEDGEMENTS</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>153</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>APPENDIX. PARAMETER DEFINITIONS</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>154</td>
</tr>
</tbody>
</table>
CHAPTER I. INTRODUCTION

The development of very large scale integrated (VLSI) circuit technology can provide hardware support for multiple processor computer architectures. In particular the economy associated with making large numbers of identical parts, makes it possible to investigate replacing large, fast computer systems with sizeable collections of small, cheap processor nodes. This replacement may be done whenever the application is factorable into a sequence of tasks or the application contains inherent parallelism.

For such an approach to be successful, certain principles should be followed. First, all nodes should be identical, both at the hardware and software level. This unifies inter-processor communication and decreases production costs. Second, no node is to require global knowledge. Such a restriction implies that the system will need no central controller. Third, the system should be indefinitely expandable. This implies a fixed number of ports per node to interconnect system nodes.

Principles of program and structure organization of computers have been suggested (Glushkov et al. 1974). These principles, explained below, support the three previously described guidelines. The first is a limitless level of machine languages where the system structure does not limit the complexity of the operands and operators. We believe that the application for a computer system should imply a high-level machine language. This language then guides the development of the system
architecture. The second suggested principle is that all program elements for which operands are available are to be executed. We believe this is desirable and can be compatible with restricting global knowledge. The third principle suggested is that memory structure should be re-programmable. We feel that the memory structure within each node is defined by the tasks of each node, and communication between nodes should be through messages rather then shared memory. Fourth, it is suggested that there be no limits to the number of machine elements. They state that this allows the design of a family of machines ranging from small machines to super systems and nets of computers. They also state that this allows a machine to continue functioning when certain machine elements fail. The fifth principle noted is that the system consists of a flexible re-programmable structure. We believe that the design of a machine for a particular application requires switching of messages between statically connected nodes.

A machine to directly support a relational database is an appropriate application for the design of a multiprocessor architecture. The functions performed upon a relational database contain inherent parallelism. We feel there is adequate work in the area reported in the literature to form a basis of comparison.

Trees as data structures and processors connected in tree structures have been investigated for computer applications. Tree structured machines have been applied to the support of database functions. These proposals have not fully explored the capabilities of a tree
machine to support a database. A binary tree communication structure has the potential to follow our three principles of design.

In a relational database environment, it is reasonable to assume that the database is stored on secondary memory devices. Assuming that a processing system requires a minimal number of secondary store transfers, there is no way to improve throughput beyond what the secondary store device will permit.

We wish to apply the three earlier stated principles to a particular application. This will allow us to compare this approach with other computer architectures designed for database support. Thus, we will be able to draw conclusions based on these comparisons, particularly conclusions that apply to other processing environments.

Outline of Thesis

There are seven chapters in this dissertation. The balance of Chapter I contains a description of the relational model of data and background information. Chapter II is an overview of six proposed database machine architectures. Chapters III, IV, and V detail the structure, machine language and hardware support of the example tree structure database machine architecture. The performance of the six machines described in Chapter II and the example tree machine are analyzed in Chapter VI. That chapter compares the response time of these machines to a representative query. Chapter VII summarizes the conclusions of the previous chapters and explains the effect of the design principles used to create the tree machine.
The Relational Model

One schema for the management of data is the relational model (Codd 1970 and Codd 1972). The relational model provides a means of describing data by its natural structure. The theory supporting this model provides for derivability, redundancy and consistency of data relationships. The model exhibits independence from the data. It provides a simplified common view of the data store. Database operations are raised to the set level rather than applied element by element. Here we present an overview of the relational model of data. For a more complete treatment, refer to one of the textbooks on the subject (Date 1977).

The concept of relations, the relational algebra, and normal forms were first proposed by E. F. Codd (Codd 1970). The relational algebra and the relational calculus were defined later (Codd 1972). In that paper, Codd defines the concept of the relational completeness of a data language. A language is said to be relationally complete if it possesses the property that any relation definable by means of calculus expressions may be retrieved via suitable statements in the language. In the same paper, he shows the equivalence of the relation algebra and the relational calculus. Since the relational algebra reflects the search operations performed on a relational database, we have chosen the algebra as a basis for our machine language. The concept of relation, domains, and degree may be defined as follows:

Given a collection of sets $D_1, D_2, \ldots, D_n$ (not necessarily distinct), $R$ is a relation of these $n$ sets if it is a set of ordered
$n$-tuples $(d_1, d_2, \ldots, d_n)$ such that $d_i$ belongs to $D_i$, for $1 \leq i \leq n$. Sets $D_1, D_2, \ldots, D_n$ are the domains of $R$. The value $n$ is the degree of $R$.

Figure 1.1 depicts an example relation STUDENT of degree 5, defined on domains $S\#$ (student number), SNAME (student name), MAJOR (student major), GPA (student grade point average) and CITY (student home town). The domain of GPA, for example, is the set of possible grade point averages, 0.0 to 4.0.

<table>
<thead>
<tr>
<th>S#</th>
<th>SNAME</th>
<th>MAJOR</th>
<th>GPA</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Harris</td>
<td>CSCI</td>
<td>4.0</td>
<td>Chicago</td>
</tr>
<tr>
<td>S2</td>
<td>Aller</td>
<td>ENGL</td>
<td>3.0</td>
<td>Detroit</td>
</tr>
<tr>
<td>S3</td>
<td>Carnes</td>
<td>ENGL</td>
<td>2.0</td>
<td>Colombus</td>
</tr>
<tr>
<td>S4</td>
<td>Lan</td>
<td>CSCI</td>
<td>1.0</td>
<td>Hong Kong</td>
</tr>
<tr>
<td>S5</td>
<td>Maurer</td>
<td>PHYS</td>
<td>4.0</td>
<td>Boone</td>
</tr>
<tr>
<td>S6</td>
<td>Ford</td>
<td>EEENG</td>
<td>3.0</td>
<td>Des Moines</td>
</tr>
</tbody>
</table>

Figure 1.1. The Relation STUDENT

Relations can be conveniently represented as tables. Each row of the table represents a tuple of the relation. The number of tuples in the relation is called the cardinality of the relation. Relations of degree 1 are termed unary and relations of degree 2 are termed binary. Likewise, relations of degree $n$ are $n$-ary. The actual tables chosen to store a database depends on the data being represented. The concept of normal form is used to choose the relations to be stored in the database. This will be discussed after the concept of keys and the relational algebra are presented.

Many times a relation has a single attribute which uniquely distinguishes a tuple. In the STUDENT relation, $S\#$ is such an attribute.
S# is said to be a primary key of STUDENT. Primary keys can be composed of multiple attributes. A relation can have several single attribute primary keys or several multiple attribute primary keys. Since sets do not contain duplicate entries, and relations are sets, then each relation must have a primary key. The access to a relation need not be restricted to the primary key.

Various set operations may be performed upon relations: union, intersection, difference, and the extended cartesian product. Union, intersection and difference operations may be performed upon compatible relations. The two source relations must be of the same degree and the jth attribute of one relation must be drawn from the domain of the jth attribute of the other relation. The extended cartesian product acts as a concatenation of two relations.

The special relational operations are most important. These allow specific subsets of relations to be formed and smaller relations to be combined into larger relations. These special operations are the SELECT, PROJECT, JOIN, and DIVIDE.

**Selection**

SELECT is an operator for choosing a subset of tuples from a relation. The tuples of the subset satisfy a specified predicate. The predicate is expressed as a Boolean combination of terms. Each term can be established as true or false by examining a tuple of the relation. If the Boolean expression evaluates to true, then that tuple is included in the result relation. Examples of selection are shown in Figure 1.2. These are SELECTS upon the student relation in Figure 1.1.
(a) SELECT student where CITY = 'Boone'

(b) SELECT student where GPA = 4.0

(c) SELECT student where GPA = 3.0 and CITY = 'Detroit'

Figure 1.2. Sample selections upon the STUDENT relation

Projection

PROJECT is an operator for producing a "vertical" subset of a relation. This operator selects specified attributes and removes all others. The PROJECT operation includes the removal of duplicate tuples formed when part or all of the primary key attributes are removed. To PROJECT STUDENT over the attribute MAJOR we first remove the attributes S#, SNAME, GPA, and CITY from each tuple. This produces the list in Figure 1.3(a). The duplication of the MAJOR attribute values must be eliminated. This gives the result relation in Figure 1.3(b).

Duplicate removal is potentially a time consuming operation. It has the complexity of a sort algorithm. For a relation of cardinality N, the duplicate removal would require O(N log(N)) tuple comparisons.
Join

If two relations have a common domain then they may be joined over that domain. The result of a join is a new relation in which each tuple is formed by joining together two tuples of the source relations. Two tuples of the source relation are joined only if the shared values in their common domains satisfy the join condition. A common condition is the equality of the values in the common domains of source tuples. This is termed the equi-join of two relations. An example of the equi-join of two relations in Figure 1.4(a) is shown in Figure 1.4(b).

(a) With duplicates  (b) Without duplicates

Figure 1.3. Projection of STUDENT over MAJOR with duplicate removal

Join

If two relations have a common domain then they may be joined over that domain. The result of a join is a new relation in which each tuple is formed by joining together two tuples of the source relations. Two tuples of the source relation are joined only if the shared values in their common domains satisfy the join condition. A common condition is the equality of the values in the common domains of source tuples. This is termed the equi-join of two relations. An example of the equi-join of two relations in Figure 1.4(a) is shown in Figure 1.4(b).

(a) Two relations which may be joined over PART

(b) JOIN of $R_1$ and $R_2$ where $R_1$ (part) = $R_2$ (part)

Figure 1.4. The equi-join of $R_1$ and $R_2$ over the PART attribute
The example in Figure 1.4 shows that a result of a join can have a greater cardinality than either of the two source relations. This is due to the value of the join field attribute PART in $R_1$ having more than one related value in the PART attribute of $R_2$. The cardinality of a join result can range from zero to the product of the cardinalities of the two source relations. In an equi-join, the later case would save the unlikely precondition that all values in the join field attributes of the two relations were identical.

We use the terms JOIN or 'strict' join to mean the equi-join. We use the term 'implicit' join for the matching of tuples without the concatenation step. Either operation requires $O(N_1N_2)$ tuple comparisons for relations stored in arbitrary tuple order. The execution of this type of query on a conventional machine is time consuming. This is true when no index files are maintained for the join field attributes.

Division

We consider here the simplest form of division. Division is an operation between a binary relation and a unary relation. Assume that the dividend $R_1$ has attributes $X$ and $Y$. Let the divisor $R_2$ have attribute $Z$. Assume that $Y$ and $Z$ are defined on the same underlying domain. The operation DIVIDE $R_1$ by $R_2$ over $Y$ and $Z$ produces a quotient defined on the same domain as $X$. A value $x$ will be included in the quotient if and only if the pair $(x,y)$ appears in $R_1$ for all values $y$ appearing in $Z$ of $R_2$. In other words, the quotient consists of those $X$-components of the dividend whose corresponding $Y$-components include every component of the divisor. Figure 1.5 gives examples of division.
### Figure 1.5. Three examples of division

<table>
<thead>
<tr>
<th>DIVIDEND RELATION</th>
<th>DIVISOR RELATIONS</th>
<th>RESULT RELATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>S#</td>
<td>CLASS</td>
<td>(a) CLASS</td>
</tr>
<tr>
<td>S1</td>
<td>C1</td>
<td>C1</td>
</tr>
<tr>
<td>S1</td>
<td>C2</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>C3</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>C4</td>
<td>(b) CLASS</td>
</tr>
<tr>
<td>S1</td>
<td>C5</td>
<td>C2</td>
</tr>
<tr>
<td>S1</td>
<td>C6</td>
<td>C4</td>
</tr>
<tr>
<td>S2</td>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>C2</td>
<td>(c) CLASS</td>
</tr>
<tr>
<td>S3</td>
<td>C2</td>
<td>C1</td>
</tr>
<tr>
<td>S4</td>
<td>C2</td>
<td>C2</td>
</tr>
<tr>
<td>S4</td>
<td>C4</td>
<td>C3</td>
</tr>
<tr>
<td>S4</td>
<td>C5</td>
<td>C4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C6</td>
</tr>
</tbody>
</table>

Complex queries

The result set of a relational operation is a relation. This means that the results of an operation can be the operand for another operation. This nesting of relational operations enables complex queries to be formulated.

For example, a query for the retrieval of information from the STUDENT relation of Figure 1.1, might require the return of all student names which have a home town of Chicago and major in Computer Science. This query can be formulated as the following expression:

```
PROJECT(SELECT STUDENT where MAJOR= 'CS' and CITY = 'CHICAGO') over SNAME;
```

Figure 1.6 shows a relation STATUS which records the rank and classification of each student.
### Figure 1.6. The STATUS relation

Suppose we wish to retrieve information which is contained in both the relations STUDENT and STATUS. For example, we could retrieve all numbers of students who have a full classification. This query would be formulated as the following expression:

\[
\text{PROJECT (JOIN (SELECT STATUS where CLASS = FULL)}
\text{ and STUDENT over S#) over S#;}
\]

### Figure 1.7. The relation COURSE

An example retrieval that makes use of the COURSE relation is presented next. Suppose we wish to get the names and student numbers of students who have full classification and have taken the most courses of any student. By using DIVIDE, this query can be formulated:
PROJECT (JOIN (JOIN (DIVIDE (COURSES BY
    PROJECT(COURSES over COURSE))
    and SELECT STATUS where CLASS=FULL)
    over S#
    over S#)
    over S#, SNAME;

Updates

To insert a new tuple into the STUDENT relation, we can make use of
the UNION operation. For example, to add a student JONES, with S# S8,
MAJOR IADD, GAP 3.0 and CITY CONESVILLE we might write:

STUDENT UNION {('S8', 'JONES', 'IADD', 3.0, 'CONESVILLE')};

To delete a tuple from the STUDENT relation, we can make use of the
difference operator. To remove student S3 we might write:

STUDENT MINUS {('S3', 'CARNES', 'CENG', 2.0, 'COLOMBUS')};

Given that S# is a primary key, it is not necessary to specify all
attribute values for a deletion.

Further comments

From the operations outlined previously, it is apparent that the
JOIN, PROJECT, and SELECT are very important. The PROJECT and SELECT
both narrow the database to subsets which are desired by the user. The
JOIN allows the retrieval of data relationships which are stored in
physically separate tables. Without the efficient implementation of
these operations, a large relational database is unmanageable.

The relational operations presume that the values of attributes
are not in themselves relations. Such a relation is a normalized
relation. This simplifies the data representations and the relational
operations. Relational tables are simplified further. Codd
originally defined three levels of normalization (Codd 1970). All normalized relations are in first normal form (1NF). A database may be organized further into second normal form (2NF) and third normal form (3NF). Fagin defined normal forms based on the operations which are allowed on the database. He defines projection-join normal form in which the only legal operations are projection and join (Fagin 1979).

The quest for the ultimate normal form continues. It is not in the scope of this explanation to pursue advanced normal forms. We assume that for a well-formed database a normal form can be found which will insure the precision of the relational algebra operations.

Background

The relational operations provide a means of retrieving any relationship from the database. The method which a conventional machine uses to store, search, and retrieve data, influences which relationships are most easily retrieved. In conventional database management systems, index files are maintained to reduce the serial search time of the database. If the database management system supports the relational schema, there may be queries for relationships which are not facilitated by index files. These queries require a single central processor to search serially through database files. Su et al. write that software techniques such as data structures and cross-reference files to some extent alleviate the speed problem. However, they go on to say software techniques introduce problems with excessive storage requirements, and require updating of pointers and index files. They
conclude that a single processor searching data in a large database of $10^{12}$ bytes cannot meet the response time required for many applications. (Su et al. 1979).

Babb puts forth the thesis that: "currently available hardware may be found fundamentally unsuited to implement relational operations" (Babb 1979). He goes on to say that overhead results from lookup of individual records to satisfy queries for sets of those records, and content addressable memory is needed to reduce this overhead.

Storing data as relational tables eliminates the need for inverted files to serve as indexes into the data. Relational tables may be stored in blocks on a secondary store. The relational operations require searches of entire relations. Thus, index files need only be maintained for the whole relation. For a moving-head-disk secondary store all that is needed is a list of the cylinders containing each relation. This eliminates the overhead of updating pointers. It also increases the need for applying multiple processors to search relations in parallel.

Currently, there is a strong interest in hardware support for storing relations directly, and for implementing relational operations. The proposed machines: RAP (Ozkarahan et al. 1974), CAFS (Babb 1979), LEECH (McGregor et al. 1976), and DIRECT (DeWitt 1979a) were all designed to store data as relational tables. Explicit hardware capabilities are supplied by these machines for performing relational algebra operations. Other machines have been designed for executions of high-level database functions. Examples of these are CASSM (Healy et al. 1972)
and DBC (Baum et al. 1976). CASSM was designed to store data in a hierarchical model. The uses of CASSM for a relational database have been examined (Lipovski 1978). The uses of the DBC for storing relational tables have also been noted (Banerjee and Hsiao 1978b). More details about CAFS, CASSM, RAP, DIRECT, DBC and HYPERTREE¹ are given in Chapter II.

Tree structure processing for database support has been suggested (Goodman and Despain 1980 and Song 1980). Of all the machines proposed, the HYPERTREE, which came out of the X-TREE project, comes the closest to our recommended principles of design. This machine is comprised of homogenous nodes interconnected into a binary tree structure. The tree has ring connections for lateral communication within each tree level. The nodes are proposed to be X-NODES (Sequin et al. 1978). Each leaf node is connected to a read/write head of a moving head disk. These disks are used as the secondary store for relational data. This gives HYPERTREE the ability to search an entire cylinder in one revolution of the disk.

Assuming that the database is stored on these special disks in HYPERTREE, the cost of expanding the database rises steeply. To add to the secondary store requires the addition of special read/write processing and a leaf node per platter of the disk. The data are distributed across disks making portability of the database difficult. The join algorithm involves routing of information from the leaf nodes

¹J. R. Goodman, Personal Communication, Department of Computer Science, University of Wisconsin, Madison, Wisconsin, 1980.
to arbitrary nodes of the tree. This random traffic requires a routing algorithm which uses global information. Because of this, the addition of nodes to expand a HYPERTREE would require modification of the software controlling the node processes.

We have followed the previously stated hardware design principles to originate hardware support for storing data directly as relational tables. It is a statically connected binary tree structure used to implement a machine language based on the relational algebra. We have used the indirect search method of reading the relations from secondary store into an intermediate memory. Except for the degree to which one can re-access relations in intermediate store such a database machine can search data no faster then it can be transferred from secondary store.

The advantage to the indirect search method is to allow expansion of the database without the cost of logic to directly read the expanded secondary store. This also stores the database on a minimal number of secondary storage devices. Such a scheme enhances the portability of the database.

We have taken a top down approach to the implementation of a machine language based on the relational algebra. Functions to execute machine instructions are distributed to the nodes. No node needs knowledge other than what node it is and what query is being implemented by the machine. The nodes are identical on the hardware level and nearly identical on the software level. This database machine architecture will be referred to as REPT (RELational Processing Tree).
CHAPTER II. DATABASE COMPUTER ARCHITECTURES

This chapter describes six of the proposed database machine architectures which were explicitly designed to support high-level database functions. A classification of such machines has been proposed (Bray and Freeman 1979). The classes are single processor direct search machines (SPDS), single processor indirect search machines (SPIS), multiple processor direct search machines (MPDS), multiple processor indirect search machines (MPIS), and multiple processor combined search machines (MPCS).

The indirect and direct search classifications describe the level at which database functions are applied to the data. An indirect search machine transfers the data to an intermediate memory. This type of machine performs comparisons upon data in the intermediate store. An intermediate store may be comprised of a fast access block addressable memory technology such as charged coupled devices (CCD) or random access memory (RAM). Queries which involve multiple access to the same data can take advantage of this cache-like intermediate storage. The number of processing units is not directly tied to the size of the database. This means that database expansion is not as costly as in a direct search machine.

Database machines which access the data at the interface to secondary store are termed direct search machines. These machines strive to perform database functions at the rate data can be transferred from secondary store. The MPDS category contains the cellular logic
machines. Typically a processor per read head of a fixed head disk is employed. This allows parallel processing of data as it is transferred from secondary store. Such an architecture allows access to all data in one rotation of the disk. This parallel access capability gives the MPDS machine an inherent advantage in the rate at which data can be transferred from secondary store. Costs rise rapidly for expansion of database within direct search architectures.

Combined search machines use both the indirect and direct search techniques. Both HYPERTREE and DBC are examples of MPCS machines in this category. HYPERTREE makes use of RAM in its nodes and a leaf node per platter of a moving head disk. DBC uses associative and RAM memory for structure information and employs a processor per platter of a moving head disk.

The remaining machines described here fall into the SPDS, MPIS and MPDS categories. The SPIS category includes conventional machines executing database management systems. Both DIRECT and RAP are MPIS machines. CASSM is a MPDS machine and CAPS is a SPDS machine. The REPT architecture described in chapters III, IV, and V is a MPIS machine. Figure 2.1 summarizes this classification of database machine architectures.

A description of these six machines follows. It includes each machines capabilities, design concepts, data models supported, instruction sets and search abilities. The search abilities of each machine are described in terms of the relational algebra operations. This provides background for the analysis in chapter VI.
The content addressed file store (CAFS) was proposed in 1979 (Babb 1979). It was developed at ICL in England. CAFS is meant to be a SPDS back-end machine. It was designed to support relational algebra operations. Multiple disk drives may be multiplexed into one CAFS processor. Projection, selection, and implicit join are completed in CAFS. The host implements the strict join of relations.

The extended CAFS hardware uses multiple bit-stores and hashed addressing to hold intermediate results. This extended hardware allows key values to be outside the store limits. Also, a precompiled, compact representation of data is not needed and the key value can be formed from several arbitrary fields.

The version of CAFS described here contains: a search evaluation unit, key registers and comparators, address filters, hash coders, bit array stores, and a retrieval unit. This architecture is diagrammed in Figure 2.2. Six key registers can contain values for comparisons against fields of tuples. Comparators send the results of tuple
Figure 2.2. The architecture of CAFS
comparisons to the search evaluation unit. This unit can evaluate the results of comparisons as logical expressions. The search unit can allow tuples to pass from the retrieval unit to the host. This decision is based on the results of comparisons between the key registers and information read from the bit array stores. An address filter removes wanted fields from a tuple. These values are sent to the hash coders. Each hash coder uses the field values to compute a hash address. The addresses are computed to be within the bit store address range. Each hash coder produces an independent hash address to access an associated bit array store. The bit array store may be written into or read. A signal from the search evaluation unit designates a read or a write operation on the bit array stores.

Selection

The selection criteria are sent to the search evaluation unit. Using this criteria the search evaluation unit initializes the key registers to contain the values for comparison against tuple attributes. Tuples from a disk file enter each key register and a latch is set if the value in a tuple is "greater than", "equal to" or "less than" the value in the key register. The output of the latches then go to latch comparators. These contain the selection conditions derived from the selection criteria. The output of the comparators are further combined in the search evaluation unit to evaluate the selection expression. If this logical expression is true then the search evaluation unit signals the retrieval unit. The retrieval unit
then passes the tuple to the host.

**Projection**

The host sends the projection description to the search evaluation unit. This unit passes the identification of the wanted attributes to an address filter and the retrieval unit. Tuples from a disk-file arrive at the address filter. This filter sends the projected attribute values of a tuple to three hash coders. These hash coders each compute a different hash function based on the projected tuples values. Each hash coder uses its hash function to create an address. Each address is used to read the RAM associated with that hash coder. The search evaluation unit receives the logical "and" of the three bit locations. If this signal is a logical "true" then it is assumed that the current tuple attribute values are duplicates of a previous tuple. If the signal is a logical "false" then the addressed bits in each bit array store are set to "true". In this case the retrieval unit allows the tuple to pass to the host.

Hashing functions are many to one mappings. Two keys with distinct values can share the same address. This means that the projection operation may lose information. Tuples with differing attribute values may "hash" to the same location in the bit stores. The second tuple would be erroneously considered a duplicate. The chance of losing information must be made extremely low. The use of multiple bit arrays can easily give a mean time until keys are lost of as great as 1,000,000 years. An analysis of the error rate has been done (Babb 1979).
Join

The search evaluation unit transmits join instructions to an address filter. The first relation is read from disk. As tuples arrive, the address filter sends the join field attribute values to three hash coders. These hash coders calculate three independent addresses from the join field values. These addresses are used to write a "true" to the bit locations in each respective bit array store. The tuples of the relations are passed to the host.

Next, the second relation is transferred to CAFS. As tuples arrive to the same address filter this unit passes the join field attribute values to the same hash coders. These coders use the same hashing functions as used for the first relation. The calculated addresses are used to read the respective three bit array stores. If all locations have "true" values then the tuple is passed to the host. This decreases the number of comparisons the host must do to complete the join operation.

As in the projection, the hashing functions can map different key values to the same address. No information is lost in the join operation. Tuples which will not participate in the join will be sent to the host. The multiple bit arrays decrease the probability that spurious tuples will be transmitted to the host.

CAFS is capable of performing the selection, projection and join operation by reading each relation only once. The projection and join can be done by using separate address filters, hash coders, and bit array stores for each operation. Selection can be done by using the
key registers. The results of the selection can allow projection to be done. The results of the projection can govern a tuples participation in the join.

CASSM

CASSM (Context Addressable Segment Sequential Store) is a multiple processor indirect search machine architecture. It was designed at the University of Florida electrical engineering department and appeared in literature in 1972 (Healy et al. 1972). Design objectives and architectural features have been well-described (Copeland et al. 1973, Bush et al. 1976, Lipovski 1978, Su et al. 1979). In addition architectural tradeoffs between associative processors, including CASSM, have been discussed (Langdon 1978, Su 1979). A prototype of a simple CASSM cell was completed in 1976. A simulator for the multiple cell configuration has been developed.

CASSM is designed to support database management functions in hardware. It supplies associative and parallel processing capabilities for retrieval and manipulation of data in a large database. Included in the implementation are non-numeric processing searches and operations.

CASSM was designed as a cellular logic machine. The architecture of CASSM is diagramed in Figure 2.3. A logic cell is permanently associated with one track of a fixed head disk. All tracks may be searched in parallel. The entire database may be accessed in one revolution of the disk.
Figure 2.3. The architecture of CASSM
Each cell is continuously reading data from and writing data to its disk track. The data words are read serially from a track. Each word passes through a pipeline of functional units. Within the cell a word can be compared to operands, replaced, marked for further searches, collected as garbage, or collected as output. Only one word comparator exists in each module. Thus, criteria for data searches involving more than one word per record require multiple revolutions. At the end of each cell logic pipeline the data words are rewritten onto the disk track by means of a separate write head. Gaps are kept on the tracks of the disk. This allows for synchronization of the timing marks that start each track. In general, the gap time grows logarithmically as the number of cells (Su et al. 1979).

CASSM has a tagged memory architecture. Distinguished bits of each word are used to identify the type of that word. Words are of type instruction, delimiter, value, or pointer.

Instructions are stored within the same store as the database. Instructions can be fetched during data searches. A stack and queue scheduling mechanism is used to control the instruction sequence.

Delimiter words are used to mark levels of data. Data are stored in a hierarchical model. This can be used to represent data in a network or relational schema. A relation can be stored as a two level tree. An example of a relation in CASSM record format is shown in Figure 2.4. The tree is then stored in a left-right linear order. The first word of a CASSM relation is a delimiter word at level zero. This delimiter word contains the name of the relation.
(a) A tabular representation of a relation STUDENT

<table>
<thead>
<tr>
<th>S#</th>
<th>SNAME</th>
<th>MAJOR</th>
<th>GPA</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Harris</td>
<td>CSCI</td>
<td>4.0</td>
<td>Chicago</td>
</tr>
<tr>
<td>S2</td>
<td>Aller</td>
<td>ENGL</td>
<td>4.0</td>
<td>Martelle</td>
</tr>
<tr>
<td>S7</td>
<td>Hurmence</td>
<td>CSCI</td>
<td>3.5</td>
<td>Chicago</td>
</tr>
</tbody>
</table>

(b) The CASSM representation of the relation STUDENT

Figure 2.4. Storage of a relation in CASSM
The tuples of the relation follow sequentially after the relation delimiter. Each tuple is begun with a delimiter word. The name of the relation to which the tuples belong is contained within these delimiter words. A bit stack in these tuple delimiter words is used to store results of search operations.

The attributes of tuples are stored sequentially after each tuple-delimiter. A value which will fit into 16 bits can be stored directly as an attribute in a name-value word. A 16 bit name identifies the attribute and a 16 bit data area contains the attribute value. The attribute S# is stored in a name-value word in the relation in Figure 2.4.

For string type attribute values, or values which are repetitious, a pointer word may be used to represent the value. The attribute name and location of the value are stored in a pointer word. The string value is stored only once. The attribute CITY is replaced by a pointer word in the relation in Figure 2.4.

A delimiter word begins a string and the next delimiter word ends the string. The string delimiter will contain the name POINT. The address of the string is the sequential position of the string-delimiter. It is calculated by counting the delimiter words from the cell zero, sequentially across cells. The address of the string-delimiter is stored in the attribute-pointer words which have that string as their value.

To reduce redundant storage, repetitious string values are stored only once. This is similar to creating a binary relation. The first
attribute uniquely identifies the string. The second attribute is the string value. Wherever a string appears as an attribute value, the identifier for that string is stored. However, by its nature, the address will change through updates and garbage collection. All pointer words containing the location of the string must be modified when the location of the string is changed. Thus, reverse pointer words are stored with every string value. These pointers give the location of the delimiter words of tuples which have the string as the value of an attribute.

Delimiter words separate relations and tuples. The attributes are represented by name-value words and pointer words. Examples of relations stored hierarchically have been published (Copeland et al. 1973 and Lipovski 1978).

Search operations

CASSM supports both content searching and context searching. Searching is accompanied by marking the words or records which satisfy the search. Only the searches suggested as useful for selection, projection and join are described here. These can be found in various papers. (Lipovski 1978 and Su et al. 1979)

Selection

The first search is termed the unordered set search. This may be used to implement a selection operation. The delimiter word of a tuple will be used to store the results of the search. A small RAM is used in each module. This search enables marking tuples which
contain data-word values satisfying Boolean conditions.

An unordered set search for a tuple with a word of value 'A' proceeds as follows. The word 'A' is stored in each module logic, a counter C and a one bit wide RAM are used. Initially C is cleared and the RAM is cleared. When a delimiter word is encountered C is incremented. A RAM must have a bit for every tuple in a cell segment. As words pass through a module's logic a 1 bit is ORED into the RAM at location C. At the end of the rotation every tuple with a word of value 'A' has its corresponding bit set in the RAM. In the next revolution the counter C is incremented as before. As tuple-delimiters are encountered, the contents of RAM location C are pushed, ORED, etc. onto the bit-stack in the delimiter word.

This second revolution may be done concurrently with the search for the next word value of a complex search criterion. Thus, the tuples satisfying the criterion ('A' and 'B') or 'C') and 'D' can be marked in five revolutions.

**Pointer search**

As explained earlier, string values are stored only once. It then becomes necessary to mark the actual string values of wanted attributes. The pointer search operates from pointer words to tuples. This operation uses the RAM, counter and bit stacks. Participating pointer words have been marked by any of various searching mechanisms.

As words flow through the module, if a word is marked, the value in that word is used to set a corresponding RAM bit. The RAM bits set by a module's logic are not necessarily accessed only by that module.
In this search all the individual RAMs are accessed as one global RAM. If two cells need to simultaneously access this global RAM, then memory contention occurs. One cell must complete the marking on an additional rotation. Thus, pointer marking in the RAM can take more than one revolution of the disk.

After every pointer is marked in the RAM, a single revolution can be used to "or", push, etc. the contents of the RAM onto the corresponding tuples bit stack. Without two global RAMs this operation cannot be done in parallel with a first step of a pointer search.

**Projection**

If projected tuples contain pointer words, then a pointer search must be used to mark the objects of the pointers. This implies that the attributes of a tuple over which the tuple is projected are marked by a search operation. Since the attributes of a tuple are all stored in identical order a burst string search can be used to mark the projected attributes (Bush et al. 1976). Making use of the fact that most CASSM tuples are of equal length and identical order can enable particular attributes to be identified by their relative position from the start of their tuple delimiter word. Such a search could mark all attributes, over which a tuple is projected, in one revolution of the disk.

Duplicate removal can be done either in CASSM or the front-end. To remove duplicates in CASSM, the individual tuples of a projected relation are used as comparator fields against the rest of the relation. Exactly one word at a time, beginning with the topmost
track, is collected. It is automatically used as a comparand, to
clear the status bit in all words which are identical to it (Lipovski
1978). This unique output requires one rotation for every word to
unmark duplicates in a relation. If the burst string search technique
is incorporated here, unique output could be reduced to one rotation
for every tuple of the projected relation.

Join

A join is implemented using the set intersection operation
(Lipovski 1978). Intersection can be used to implicitly join two
relations. A 1 bit wide RAM is used as in the CAFS implicit join. As
in CAFS it is of more general use to compute a hash address from the
join fields of a tuple then to use and maintain a precompiled list
of unique code words. The delimiter-word address gives such a unique
code to string value attributes. The position of the string delimiter
word relative to other delimiter words, counting from the first track,
uniquely identifies a string. This delimiter location could be used
as an index into the RAM during a JOIN operation if the join field is
composed of one string valued attribute. For a more general join field
made up of pointer words and name-value words, a hash function is used
to calculate a RAM address. Thus, it is assumed that the attribute of
the join field may be used as the input to a hashing address calculation
done in each module.

The calculated address is used to access two global one bit wide
RAMs. The CASSM literature does not mention using two RAMs. However,
to take advantage of the direct search capabilities, the implicit join
of two relations can be done by reading the first relation twice and marking two RAMS (Hawthorn and DeWitt 1980).

Suppose two relations $R_1$ and $R_2$ are to be joined in CASSM. The join criteria are broadcast to each cell. The tuples of $R_2$ are marked differently from those of $R_1$ by a previous search operation. In the first step when tuples of $R_2$ pass through the cell logic the join fields, which have been previously marked, are used to calculate a hash function which provides an address into RAM1. The value '1' is written to these locations. Contention for the address bus can cause extra rotations of the disk during this writing to RAM1.

Next, when tuples of $R_1$ pass through the module logic the join field is used to calculate the same hash function used on $R_2$. If the bit in RAM1 at this calculated location is set then the tuple of $R_1$ is marked for collection and the same address is used to write a value '1' to a bit in RAM2. This can take more then one revolution due to address bus contention. At the end of this the marking of the tuples of $R_1$ has implicitly joined $R_1$ to $R_2$. A reread of $R_2$ tuples, with addressing to RAM2, can mark the tuples of $R_2$ for collection. Once these three steps are finished, the marked tuples represent the implicit join of $R_1$ and $R_2$. However, some extra tuples will be marked for collection due to hashing collisions.

The marked tuples of $R_1$ and $R_2$ are collected and transferred to the host computer. The output of these implicitly joined tuples of $R_1$ and $R_2$ can take more then one rotation of the disk. This is due to contention for the single output channel to the host. The host can
then complete the "strict" join of $R_1$ and $R_2$.

DBC

The database computer (DBC) was proposed at the Ohio State University in 1976 (Baum et al. 1976). Descriptions of algebra operations upon the DBC and DBC's performance have been published (Banerjee and Hsiao 1978a and Banerjee and Hsiao 1978b). These papers form the basis of the following discussion.

The DBC is capable of supporting various data models. It is intended as a back-end machine. The DBC stores the database in an on-line secondary store. The secondary store employs modified moving-head disks. An entire cylinder may be read in parallel. A microprocessor exists for each track of a cylinder. This allows the secondary store to do parallel searches upon a cylinder of data. In one revolution a complete cylinder can be read, relevant data can be found, and the data transferred out. This secondary store is referred to as a partitioned content addressable memory (PCAM).

Relations are stored in DBC, as attribute-value pairs. The attribute distinctly identifies values within a tuple. A tuple $(a_1, a_2, \ldots, a_n)$ of relation $R(A_1, A_2, \ldots, A_n)$ can be stored in the DBC as

\[
\begin{align*}
(Relation, R) \\
(A_1, a_1) \\
(A_2, a_2) \\
\vdots \\
\vdots \\
(A_n, a_n)
\end{align*}
\]
Tuples of a relation are stored in as few cylinders as possible. Clustering by relation name will always ensure that the DBC will satisfy any given one-relation query by accessing no cylinders other than those required to store the tuples of the relation.

The DBC is comprised of special purpose units. Two pipeline loops exist and are called the structure loop and the data loop. The DBC is diagramed in Figure 2.5. The structure loop is composed of four units: the keyword transformation unit (KXU), the structure memory (SM), the structure memory information processor (SMIP), and the index translation unit (IXU). These four units are designed to function concurrently. Queries in the form of conjuncts of keyword predicates are sent from the database command and control processor to the KXU. The KXU converts the keywords into the internal representation. The keywords are passed to the SMIP. The SM retrieves and updates information on the cylinder locations, clustering indices and security identifiers for keywords. The SMIP performs set intersections on this structural information supplied by the SM. The structural information from the SMIP is passed to the IXU. The IXU decodes the SMIP output. The structure loop translates user queries from keyword predicates to index terms. The index terms are used by the database command and control processor (DBCCP).

The DBCCP is connected to both the structure loop and the data loop. The two units on the data loop are the mass memory (MM) and the security filter processor (SFP). The MM is implemented by the PCAM concept discussed earlier. The MM is capable of searching for and
Figure 2.5. The architecture of DBC
retrieving records which satisfy simple criteria. The SFP does sorting and merging of result data which comes from the MM. The checking of attribute values against security sanctions is done by the SFP.

The DBCCP regulates the operations of both the structure loop and the data loop. It also interfaces with the front-end computer. Duties of the DBCCP include: scheduling of front-end commands, enforcing security on a selective basis, clustering records to be stored, and routing results of queries to the front-end system.

**Selection**

Any query is specified as a keyword predicate. Attribute names, relational operations, values, and Boolean operators make up the predicates. To select a subset of the relation \( R(A_1, A_2, A_3) \) where \( A = 'RED' \) or \( A_2 = 3 \) the query

\[
\text{Retrieve: } (A_1, A_2, A_3) \ (\text{Relation} = R) \ \text{and} \ \ (A_1 = \ 'RED') \ \text{or} \ \ (A_2 = 3)
\]

is executed.

To satisfy this query the entire relation \( R \) must be read. Each cylinder of \( R \) can be scanned in one revolution of the disk. Each processor in the MM can evaluate the predicate for the tuples accessed by its read head. The selection can be completed in a number of disk rotations equal to the number of cylinders which hold \( R \).

**Projection**

The MM is capable of removing unwanted attributes of tuples for the projection operation. There is no facility in the DBC for
duplicate removal except by the SFP. The duplicate must be removed in the host or the SFP.

**Join**

A join operation can be carried out by first retrieving the smaller relation by the host issuing a query to the DBC for every tuple of this relation. The MM would then search the second relation completely for each tuple in the first relation. This shows a lack of join facilities in the MM. The series of subqueries to complete the join is costly in terms of execution time.

**RAP**

The RAP system is a back-end database machine designed to store data as relational tables. It is a cellular logic machine where each cell contains memory and comparison logic. The memory supplies indirect search capability to the RAP architecture. RAP is a MPIS machine which can do parallel searches upon data stored within cell memories. RAP was first proposed in 1974 at the University of Toronto (Ozkarahan et al. 1974). Other publications describe the implementations of searches and the performance of RAP (Ozkarahan et al. 1977, Ozkarahan and Sevcik 1977, and Schuster et al. 1979).

A RAP device consists of a set of identical components called cells, a central controller, and a statistical arithmetic unit. RAP is diagramed in Figure 2.6. Each cell is composed of a processor and a block addressable memory. The processor can be implemented by LSI technology. It is designed for the implementation of database
Figure 2.6. The architecture of RAP
definition, insertion, deletion, update and retrieval primitives. The memory can be implemented by rotating magnetic devices, CCDS, Bubble memories, or RAMS. The statistical unit is designed for computing summary statistics over the contents of combined cell memories and is part of the controller. Additionally, the controller is responsible for decoding RAP instructions from the front-end. It broadcasts control sequences for cell execution and transfers data between the cells and the front-end.

RAP consists of multiple cells. This gives the RAP architecture capability of parallel searches of data stored in the cell memories as relational tables. The attributes are in a fixed length format determined at data definition time. Each relation and its tuples are augmented by several mark bits. These mark bits are used to aid in searching a retrieving data. Unlike the conventional relational model, a tuple in a RAP relation can have a duplicate.

Selection

The selection operation is implemented by the RAP select instruction. This instruction selects qualified tuples from the relation and sets the mark bits of these tuples. A qualification expression is included in the select instruction. This expression is a Boolean disjunctive or conjunctive expression of simple conditions each of which tests an attribute value against a specified constant.

The following is a formula for the average number of rotations to complete a selection operation (Ozkarahan et al. 1977).
.5 + \left\lfloor \frac{CL}{K} \right\rfloor \text{ for a single clause criterion}

or .5 + \left\lfloor \frac{CL}{2} \right\rfloor \text{ for a multi-clause criterion}

In the above formula CL is the number of simple conditions in the criterion list and the .5 comes from the latency of rotational storage devices. When the qualification expression is a simple conjunction or disjunction, K simple conditions can be processed in each RAP revolution since each cell has K comparator units. In the worst case at least two terms will be processed in each revolution.

**Projection**

If the source relation can fit completely into RAP cell memory then a projection may be accomplished by an iterated get-first-mark instruction (Schuster et al. 1979). The tuples to be projected are assumed to be marked by a previous operation. The first tuple is found with a specified mark bit set. One of its attribute values is used to do a new marking on the entire relation. Finally, the tuple originally selected has its mark bit reset so it will be bypassed by the next execution of the same get-first-mark instruction. This implies 3 rotations of cell memory per execution of this instruction.

It is apparent from the get-first-mark method that if the entire relation can not be stored completely in cell memory then duplicate removal must be done in the front-end.

The only intercell communication is for the acquisition of the multiplexed output bus to the controller unit. Output of tuples to the controller may take more then one rotation of all cell memories. This is due to the contention between cells for the single output bus.
The number of rotations for output of a RAP relation depends on the average number of tuples, marked for output, which occupy the same relative locations in different cell memories. This output rotation function has been simulated (Ozkarahan et al. 1977).

**Join**

RAP performs an implicit join of two relations. This can be done by a cross-mark operation. The cross-mark uses the mark bits and values of a first relation, $R_1$, to mark tuples in a second relation, $R_2$. For $K$ cells in the system the cross-mark instruction takes more than $N_1/K$ revolutions to complete. Where $N_1$ is the number of tuples in $R_1$. The cross-mark can be done if both relations do not fit into the cell memories.

When the source relations will not completely fit into cell memories, a subset of RAP cells could be locating the next tuple of $R_1$ and transferring it to the controller unit. The rest of the cells could do cross-marks based upon the $R_1$ tuples. These other cells contain the tuples of $R_2$ as they are transferred from secondary store. To keep the overhead low, a series of cross-mark searches should be executed upon the tuples of $R_2$ currently in cell memories. After all tuples of $R_1$ have been transferred from secondary store into cell memories and transferred to the controller, then new blocks of $R_2$ tuples are stored in cells and the transfer of $R_1$ is repeated. Rather than repeatedly fetching $R_1$ from secondary store $R_1$ could be stored in the front-end. Then separate selection searches can be done for each tuple of $R_1$ with each transfer of $R_2$ to cell memories.
A back-end database machine has been proposed at the University of Wisconsin in 1978 (DeWitt 1979a). The information in this section comes from various publications (DeWitt 1979a, DeWitt 1979b, and Hawthorn and DeWitt 1980). This machine is called DIRECT. It consists of a back-end controller, a set of query processors, a set of charge-coupled device (CCD) memory modules and an interconnection matrix between the query processors, CCD memories, and one or more mass storage devices. DIRECT is diagramed in Figure 2.7.

The host processor executes a version of INGRES. User queries are compiled by INGRES into a sequence of relational algebra operations called query packets. These query packets are sent to the back-end controller.

The back-end controller interacts with the host processor and commands the query processors. It determines the number of query processors to execute a query packet. If relations referenced by a query packet are not in the CCD memory modules, the back-end controller will page portions of the relation into the memories.

The function of each query processor is to execute query packets assigned by the back-end controller. The assigned packets are transmitted over a parallel word interface to the query processor.

The interconnection matrix allows a query processor to read any CCD memory module. It also allows multiple query-processors to read one CCD memory module. This gives DIRECT the ability to execute different queries simultaneously and apply multiprocessing to a single
Figure 2.7. The architecture of DIRECT
query.

The interconnection matrix can be implemented by a cross-bar switch. This switch would have the memories act as producers and the processors act as consumers. Each CCD memory continuously broadcasts its contents. Whenever a processor wishes to examine the page of a relation, it instructs the cross-bar switch to select the proper CCD memory line.

The CCD page frames are stepped with a common clock. This requires only one address line. Any query processor can begin to examine data in a CCD at the next tuple boundary. Thus, nearly no latency time exists.

Data are stored as relational tables. Each relation is divided into a number of fixed sized pages. Each page contains tuples in sequential order from one relation. Each tuple is allocated a fixed number of bytes. This eliminates the need for special characters to delimit tuples within the page and attributes within the tuple.

The fixed page and tuple format cause internal relation fragmentation. As tuples are deleted from relations, unused bytes appear throughout a page. As result relations are produced, parts of pages may not be used. DIRECT has a compress command which reduces to a minimum the number of pages which a relation occupies.

The query processor instruction set contains the relational algebra operations of restrict, project, and join. Set operations also are instructions. These are union, difference, intersection, and cross-product. The update commands of modify and insert allow for changing
the database. Compress gives the query processors a command to reduce unused bytes within pages. Aggregate operators are also included for such functions as maximum, minimum, count and average.

DIRECT does not use mark bits. Instead the instructions restrict project, join, union, difference, intersection, and cross-product produce temporary relations. These temporary relations can be stored in CCDS or returned to the user.

Selection

DIRECT has a restrict command in its query processor instruction set. The back-end controller can command any number of processors to perform a restriction. The controller specifies which module contains the source relation.

The restriction selects a subset of the source relations. The qualification of source tuples is calculated from a Boolean expression included with the restrict command.

The architecture of DIRECT allows one processor to perform a restriction on the complete relation. It also permits many processors to produce the same restricted set. Another alternative is to have multiple processors each restrict a subset of the source relation. The last method is the most efficient method for large source relations. The second implementation can be used for a restriction proceeding a join. In such a case the same restricted set would be needed in all query-processors for completing the join.
Projection

The project command instructs a query processor to remove unwanted attributes from a relation. The result relation may contain duplicate tuples. No high level facilities are described for duplicate removal in DIRECT. The duplicate removal operation is assumed to be done by the host.

Join

The join operation has been described and analyzed for performance in an explicit example (Hawthorn and DeWitt 1980). That example has been used as a guide for the following description of the join. Suppose the join of two relations, R1 and R2, is to be performed by DIRECT.

The smaller relation, assumed to be R2, is paged into the CCD memory modules. One or more query processors each read the entire relation R2. If R2 cannot be completely stored in a single query processor memory then R2 must be divided into segments which can be stored in that memory space. In that case the entire join operation must be repeated for each segment of R2.

R1 is paged into CCD memories. The participating query processors each can join all of R2 to a subset of R1. This is done by each query processor reading a separate memory module containing pages of R1. The join operation uses one or more query processors each producing a subset of the result relation. Matching tuples of R1 and R2 are concatenated and stored in an output buffer associated with each query processor. The output buffers are transferred to CCD memory modules
when they become full.

The transfer of \( R_1 \) to CCDS, the comparison of the pages of \( R_1 \) in CCDS to the tuples of \( R_2 \) stored in the query processors, and the output of joined result tuples forms a three stage pipeline. The longest stage will govern the overall response time of the join in DIRECT.

HYPERTREE

The HYPERTREE configuration of X-TREE nodes has been proposed as a database machine architecture.\(^2\) It is a development from the X-TREE project (Despain and Patterson 1978 and Sequin et al. 1978). An X-TREE node contains a processor of substantial computational power. Each node contains a memory of considerable size. The HYPERTREE is a binary tree structure composed of X-TREE nodes with regularly placed cross links at each level of the tree. HYPERTREE is diagramed in Figure 2.8. This configuration has been shown to be capable of various duplicate removal operations (Goodman and Despain 1980).

Each leaf node of HYPERTREE is attached to a read head of a moving head disk. This gives this machine the capability of accessing a cylinder of the database in one revolution of the disk. The memories in each node are used to store intermediate tables and relations. This places HYPERTREE into the MPCS machine category.

\(^2\)J. R. Goodman, Personal Communication, Department of Computer Science, University of Wisconsin, Madison, Wisconsin, 1980.
Figure 2.8. The architecture of HYPERTREE
Various relational operations have been described for different clustering fields and index files. We are considering the capability of machines where the tuples of a relation are not ordered. The processing of queries is not assumed to have any advantage of the clustering of tuples in secondary store. The overhead of maintenance of index files is avoided. Thus, we consider the selection, projection, and join implementations of HYPERTREE which do not make use of index files or data clustering.

**Selection**

The criteria for selection are transmitted to each leaf node. The relations are searched a cylinder at a time. The selection of tuples proceeds in parallel among the leaf nodes. The selected tuples are transmitted as they are chosen. We must assume that some node serves as the interface to the user. The distributed routing algorithm of X-TREE can be used to send tuples from each leaf node to the user interface node.

**Projection**

Each leaf node can remove unwanted attributes from the source relation. The entire relation is read and subsets of projected tuples stored in each leaf node. The leaf nodes can use a variety of duplicate removal operations. Among the possible methods are the perfect shuffle, the n-cube merge, and the tree merge. The tree merge duplicate removal proceeds by sorting each set of tuples in the leaves. Each leaf sends the tuples to its parent node. The parent node merges the two sorted
streams of tuples form its descendant. In this process it removes
duplicates and sends the results to its parent node. This continues
until the root node receives a duplicate free result relation.

Join

HYPERTREE makes use of one bit wide hash tables to implicitly
join two relations. Only the implicitly joined tuples are allowed to
enter the HYPERTREE structure where they are strictly joined.

First, the leaf nodes read one of the source relations, $R_2$. A
bit array is marked as in the CAPS join operation. Each leaf node
contains a "local" bit array after $R_2$ is completely read. Each leaf
node transmits its bit array to its parent node. The parent node does
a logical "or" of the two descendant bit arrays. This "ored" array is,
in turn, sent to its parent. This proceeds until the root stores a bit
array which contains information from every tuple of $R_2$.

Next, this procedure is repeated for the other source relation,
$R_1$. Now the root contains a bit array for both $R_1$ and $R_2$. The root
"ands" these two arrays together and transmits this combined array to
all leaf nodes. The leaf nodes re-read each relation. If the join
fields of a tuple "hit" in this array then the tuple is sent to a
chosen node.

The routing of implicitly joined tuples from the leaves to a
node makes use of the X-TREE distributed routing algorithm. The
destination node is chosen so that tuples of both $R_1$ and $R_2$ are sent
to the same node if their join fields will be matched in the strict
To minimize hashing collisions multiple bit arrays can replace the single bit arrays described above. As in CAFS the use of more than one bit array reduces the number of erroneous tuples sent to arbitrary nodes in the last step of the join. The number of bits in the array is based on the number of unique values in an average relation.
CHAPTER III. SYSTEM STRUCTURE

Database System Duties

A database system serves as an interface between a large shared database and multiple users. Users are supplied with a high-level interactive query and update facility. User queries and updates are translated into database primitives which are interpreted either by a database machine or database management software.

With multiple users accessing data simultaneously, queries must be scheduled. The scheduling can depend upon the subset of data accessed by each query. To supply a consistent view of data careful ordering of updates and queries must be maintained. The support of simultaneous access to data by multiple users is important. However, this has been left for future research.

Security of data requires maintenance of user identifications and capabilities. The control of data access can be a function of both the users capabilities and the data contents. The security of data may be partially maintained by a back-end database machine. We have not considered such applications and we have currently left security functions out of the back-end design.

The system envisioned here is divided into three major components. The front-end computer, the back-end machine and the secondary store. The front-end translates queries into the back-end machine language. The back-end machine performs the commands upon the data in secondary store. A proposed database system is diagramed in Figure 3.1.
Figure 3.1. The architecture of REPT
User interface

The user interface allows retrieval and update of subsets of data which are designated by their attribute relationships. The languages which are presented by the front-end machine must be interpreted by the back-end machine.

Updates and queries are expressed by users as high level language statements. Such statements must be interpretable by the back-end machine. In this design any query language may be used which can be interpreted as expressions of relational and set operations. Such languages as INGRES (Held et al. 1975), QBE (Zloof 1975), and SEQUEL (Chamberlin and Boyce 1974) have been implemented for database management systems. These would be suitable as high-level user query languages in this system.

Various logical views of data have been found to be useful for specific classes of database access problems. Three common schemas are the network, hierarchical, and relational model. A query language based on any of those schemas can still be interpreted by a machine language based on the relational algebra.

Front-end machine duties

The front-end, general purpose host computer serves as the interface between the user and the back-end machine. System software translates high-level queries into relational and set operations.

Security and user identification may be handled by the front-end. Thus, illegal queries can be halted away from the data operations. Although not studied here, the implementation of database security
functions could be separated from the front-end and/or included in the back-end.

Validity and integrity may be checked by the front-end machine. The monitoring of attribute values for updates is one example. Another is the validation of relation names in query translation.

Other functions such as system logs and report generation can be done by the front-end. The front-end is also the interface to the system manager. The initialization, definition and extension of the internal relational model may be done by interaction between the front-end and the database designer.

**Back-end machine duties**

The principal task of the back-end machine is to implement high-level database functions. It serves as an intermediary between secondary store and the front-end.

A protocol for communication with the host must be established. Queries are transmitted in the form of relational and set operations. Expressions of this type comprise the high-level back-end language.

The back-end machine communicates with secondary store to transfer relations. The unique identification of the relations is supplied at systems design time. The location of the actual relational table on secondary store is the responsibility of the secondary store unit. Thus, the back-end machine may ask for data from secondary store by name.

Once relations enter the back-end, the relational or set operations can be performed. Parallel processing needs to be generously applied.
The efficient transfer of result tuples is important. Result relations may be operands in future operations. A multiple join query is one example. In such a query the result set of the first join is an operand to a second join. Storage for intermediate relations can be included in the back-end machine. Operations may be executed upon intermediate result relations as they are produced. This can reduce the demand for storage space for these temporary relations.

Given the history of previous queries, ordering of operations within expressions can eliminate unneeded transfers of relations between secondary store and the back-end machine. Relations within the back-end machine should not be over written or transferred to secondary store if they can be used as source relations. A back-end process can monitor transfers to and from secondary secondary store. This process could commute operations within a query or change query scheduling order to efficiently use source relations currently stored within the back-end machine.

Now we can summarize the duties of the back-end machine. A high level machine language based on the relational algebra is interpreted. Parallel processing is applied to the execution of these operations. Source relations are retained in back-end machine storage as efficiently as possible. Transfer of relations to or from secondary store is done on command form the back-end. When possible, intermediate result relations are operated upon as the tuples are produced.
**Back-end machine architecture**

The back-end machine architecture consists of interconnected nodes. Each node is capable of concurrent communication and processing. The VLSI X-TREE node is an example of such a building block. Unlike the general purpose X-TREE configuration, the communication topology studied here is designed explicitly for implementation of relational and set operations upon a database.

The uses of a binary tree composed of X-TREE like nodes are investigated here. This machine will be referred to as REPT. The nodes are statically connected in a binary tree structure. Communication between nodes is asynchronous. If an input buffer of a node port is full, then no information is accepted until space is opened in the buffer.

Fault tolerance although a major concern, has not been considered. The issues of alternate routes and fault detection have been left for future research.

As described in Chapter II, HYPERTREE has been investigated as a database machine. Unlike HYPERTREE this investigation has taken the indirect search approach. It then does not grow in cost with the increase of disk storage as quickly as HYPERTREE. Cross linking of nodes is not needed for relational operations since data travels either down or up the tree. Another difference is that routing of data within a node is simplified. The choice of output ports does not need the general purpose X-TREE routing algorithms.
Some interesting abilities of a tree structure have been identified here. The transmission of commands and data to all nodes can be simply implemented. The root begins transfer of the information to both descendants output ports. This process is repeated for all intermediate nodes. The leaf nodes all receive this information from their ascendant input ports and do not transmit it to any output ports. This pattern of transmission will be called DDT (dual descendant transmission).

Another ability is for tuples of a relation to be distributed across leaf node memories. The tuples enter the root (from secondary store). The root node alternates sending one tuple to the left descendant output port and the next tuple to the right descendant output port. The intermediate nodes continue this alternate descendant routing of tuples. The result is to distribute tuples evenly to all leaf nodes. We will refer to this operation as distributing a relation.

The DDT capability is useful for propagating machine commands to all of the nodes. It is also useful for transmitting tuples of a relation to all leaf nodes in an operation such as a join. The distribution of tuples across leaf nodes is used for parallel processing of tuple comparisons such as in the join operation.

A reverse operation to the above can easily remove tuples from leaf nodes. Each leaf node sends its tuples to the ascendant output port. The intermediate nodes first route a tuple from left descendant input port to the ascendant output port. They then route a tuple from the right descendant input port to the ascendant output port. These
two transfers continue alternately until an end of relation marker is received at both descendant input ports. The root follows the same procedure and thus outputs the tuples from the back-end machine.

All the above system operations execute in the time it takes to transmit the tuples. The tree is filled in $dt$ time where $d$ is the depth of the tree and $t$ is the tuple transfer time. After this fill time the tuples arrive to leaves (or the root) at the rate $1/t$. The execution time for any one of these three operations is $dt + Nt$ for $N$ tuples in the transmitted relation.

Secondary store is interfaced to the root node. Bulk storage devices are connected to an input port on the root via a multiplexed channel. It is the duty of the secondary store to keep track of absolute locations of relations. The root can request that the secondary store transfer a particular relation to or from the root. The root is also in a position to monitor the transfer of relations to or from secondary store. This information can be used for the scheduling of queries and the commuting of operations within queries to take advantage of source relations currently stored within the back-end machine.

Data format

The distributed computing of X-TREE implies a certain format for relations. Relations travel serially over the node links. Each relation needs to be identified uniquely. Thus, a unique internal relation name is assigned as relations are created. The number of distinct relations will not be large so that the unique criteria is
not hard to meet. This relation name always leads the tuples of a relation through the tree.

Machine instructions prepare a node in the tree for arrival of a relation. Thus, the instructions contain the names of the source relations. Other information about tuple format is needed. The relation, S, described in Figure 3.2 is shown in a serial format in Figure 3.3. For execution of queries the node processors need information about a relation. The number of attributes per tuple, the attribute domains, the length of the attributes, and the data types of the attributes are needed to execute the comparisons for set and relational operations. This information precedes the tuples as a relational header. The indefinite length of an attribute allows for such data types as strings. These are delimited by a special end of attribute marker.

Since arbitrary length tuples are assumed, the tuples are delimited by a special end-of-tuple marker. For the same reason an end of relation marker delimits the last tuple of a relation. This marker serves as a special character for the system programs controlling the node processes.
(a) An example relation \( S \)

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Domain</th>
<th>Length (bytes)</th>
<th>Data type</th>
<th>Type Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>S#</td>
<td>( D_1 )</td>
<td>2</td>
<td>Integer</td>
<td>( t_1 )</td>
</tr>
<tr>
<td>SNAME</td>
<td>( D_2 )</td>
<td>I</td>
<td>String</td>
<td>( t_2 )</td>
</tr>
<tr>
<td>MAJOR</td>
<td>( D_3 )</td>
<td>4</td>
<td>Symbol</td>
<td>( t_3 )</td>
</tr>
</tbody>
</table>

\( I \): an indefinite attribute length \( \geq 1 \)

(b) The description of the attributes of relation \( S \)

Figure 3.2. Description of the relation \( S \)
The table below shows the header for relation S:

<table>
<thead>
<tr>
<th>Relation Identifier</th>
<th>Number of Attributes</th>
<th>Attribute Descriptors</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>3</td>
<td>((D_1,2,t_1), (D_2,1,t_2), (D_3,4,t_3))</td>
</tr>
</tbody>
</table>

(a) The header for relation S

(b) A serial format for relation S

```
tuple 1
3 'Moses' enda RLGN endt

tuple 2
2 'Gipper' enda PHED endt

tuple 3
7 'Cahn' enda PLSI endt endr
```

enda: end of attribute marker
endt: end of tuple marker
enadr: end of relation marker

Figure 3.3. Representation of relation S in REPT
CHAPTER IV. BACK-END MACHINE LANGUAGE

This chapter describes the instruction set for the REPT machine. It consists of high-level database operations based upon the relational algebra. It includes a description of set operations, relational operations, and system operations. The individual instruction implementations upon a tree structure are discussed.

The basic back-end machine operations can be combined into queries in the form of relational expressions. These query expressions are then transferred to all the nodes of REPT by DDT. A discussion of the basic REPT operations follows.

The set operations which are included are union, intersection, and difference. These are the usual set operations. For example the union, $R_1 \cup R_2$, combines the two relations, $R_1$ and $R_2$, into one duplicate free relation. The intersection, $R_1 \cap R_2$, is the set of tuples which are members of both $R_1$ and $R_2$. The set difference, $R_1 - R_2$, produces a set of all tuples which are in $R_1$ and are not in $R_2$. Union, intersection, and difference require that the corresponding attributes of the two source relations have identical domains.

The relational algebra based instructions are SELECT, PROJECT, JOIN, and DIVIDE. These operate on relations as previously described. The order in which the operations are executed can depend upon the condition of the system. For example, the order of execution can depend on what operand relations are currently stored within REPT. Queries are in the form of relational expressions. Each expression, when received, is formatted by the root in a form which is easily
interpreted by any node. For example, a postfix form could be used for query expressions. An example of a query expression is shown in Figure 4.1.

Delineating queries from each other and from data can be done by special separation markers. This allows the nodes to anticipate the arrival of source relations and the destination of intermediate results in complex queries.

The assumption is made that query expressions arrive to a node before the arrival of the source relations. The exception would be for relations already stored at the leaf nodes.

When a query is received at the root the appropriate instruction expression is transmitted to all nodes. This can be accomplished through routing instructions from ascendant input ports to both descendant output ports. Also, the root must request from secondary store relations which are not already in the leaf nodes.

### Implementation of Operations

#### Selection

The first phase of the SELECT operation is for the root to decide whether the source relation is in secondary store or spread across the leaves. This information is available because relations pass through the root on the way to the leaves.

If selection is the complete query and the relation is in secondary store, then the root can select tuples as they are transferred from secondary store. The root node processor can send qualifying
JOIN (PROJECT (SELECT S where CITY = 'Des Moines')
    over SNAME
  and SELECT T where GPA = 3.0)
over SNAME

(a) A user query

S SELECT S# = 3 end PROJECT SNAME end

T SELECT GPA = 3.0 end JOIN SNAME end endq

end: end of operation marker
endq: end of query marker

(b) The corresponding query expression in REPT

Figure 4.1. A query expression format
tuples of the relation to the host input port.

If the source relation is shared in the leaves, then the selection instruction is sent to all nodes of the system. Since queries are delimited, the result of the selection will be transferred to the root. The nodes know that the selection results are not used in further operations.

Upon receipt of a selection instruction, which makes up one entire query, the leaf nodes will find that part of the source relation is contained within each leaf node memory. The instruction is of arbitrary length. This allows for inclusion of any selection criterion with the instruction. The criterion is a Boolean expression which tests the specified attribute values of each tuple of the source relation (stored in leaf node memory). The qualified tuples are sent to parent nodes.

Upon receipt of data from a descendant, the parent node can react according to the instructions it has received for that data. The result relation is uniquely identified at the time the query is broadcast from the root. The intermediate nodes pass selected tuples upward. The root can then pass the tuples serially to the host.

For very large relations, all tuples may not fit into the leaf nodes. A combination of the root selection and the leaf node selection may then be implemented. Both subsets of results are then transmitted to the host.

For the selection of a source relation \(R_i\) within secondary store, the operation takes primarily as long as the transfer of the tuples
from secondary store. This time is DT BIL (see the Appendix for parameter definitions).

The selection of a relation stored in the leaf nodes takes the maximum of the time to do the parallel selection and transfer the last tuples to the host, or the time to transfer all result tuples to the host. When selection reduces a source relation minimally, then the transfer of results in a serial fashion will take the maximum time. The time for this SELECT can be expressed as:

\[ \text{maximum } (\frac{tC N_l}{K} + dt + kt, dt + N_l F_k t) \]

**Projection**

The PROJECT operation can be implemented by two methods. The bit map hash encoding of the projected attributes could be done in the root. This would provide for a duplicate removal operation like CAFS. The source relation could enter the root from descendant nodes or from secondary store. Result tuples can then be transmitted to the host.

If the small probability of information loss can not be tolerated, then the tree structure can be used to remove duplicates. The source relation must be in the leaf node memories or transmitted down the tree to the leaf nodes.

In this instance the subset in each leaf node is sorted, duplicates removed and sent to its parent node. Each intermediate node merges the tuples from its descendant ports and eliminates duplicates. This continues up the tree to the root. An end of relation marker must follow each tuple subset as it is combined and flows up the tree. The root then transfers a duplicate free projected subset of the source relation.
to the host. This ability of a tree structure has been compared against other multiple processor communication topologies for duplicate removal (Goodman and Despain 1980).

The time to project and use a tree-merge to remove duplicates is given below. It is the time to project relation $R_1$ and have the results arrive at the root. $R_1$ is in secondary store.

$$dt + DT BL1 + \text{SORT}(N_1 / k) + (d + 2)t + N_1 t$$

Join

The JOIN operation can be performed between relations in secondary store or relations which are stored in leaf node memories. The first condition requires that the smaller relation be distributed across the leaf node memories. The two source relations will be called $R_1$ and $R_2$.

For the first case we consider $R_2$ stored across the leaf nodes and sorted. $R_1$ is transmitted from the root to all the leaf nodes. As the tuples of $R_1$ arrive at the leaf node processors, comparisons are made to implement the join. Results are produced for matches and transferred from the leaf nodes up to the host.

This constitutes a three step pipeline. The first step is the transmission of the relation $R_1$ from secondary store to the leaf nodes. The second is the comparison of each arriving tuple of $R_1$ to the stored subset of $R_2$. Last, the results are transferred to the root. Once the depth of the tree is traversed, the rate at which result tuples are received depends upon the time of the maximum length step.
The comparison time per arriving tuple of $R_1$ depends on the numbers of tuples of $R_2$ stored at a particular leaf node. For a comparison time of $t_c$, $k$ leaf nodes, and $N_2$ tuples of $R_2$ we have $N_2 t_c / (2k)$ average search time per leaf node. This time could be reduced by use of a binary search or a hashing method. As the cardinality of $R_2$ grows, so does the time for comparison of each tuple of $R_1$.

The average number of tuples produced within a leaf node during the comparison of one tuple of $R_1$ to the stored subset of $R_2$ is $N/N_1$. $N$ is the cardinality of the result relation. Refer to the Appendix for parameter definitions. The response time for a join operation can be expressed as:

$$dt + DT BL2$$
$$+ \text{ maximum } (DT BL1 + Nt / N_1, \text{ to read } R_1$$
$$N_1 N_2 t_c / (2k) + Nt / N_1, \text{ to compare } R_1 \text{ tuple}$$
$$N_2 t_c / (2k) + Nt) \text{ to output results.}$$

As the cardinality of the source relations increases, one of the last two expressions becomes maximum. This is because the rate of transfer of tuples from secondary store remains constant. However, for increases in $N_1$ and $N_2$, on the average, the comparison time and the number of results increase.

It is not desirable to have $DT / (QE) < t$. Increases in results produced per tuple transferred from secondary store can cause response time increases unless $t < DT / (QE)$. This is shown in the examples of Chapter VI.
If the entire relation $R_2$ can not be completely stored in $K$ leaf node memories, then part of $R_2$ is distributed. The join operation is executed as above. Then the remaining tuples of $R_2$ are stored in the leaf nodes and the join operation is repeated. The entire operation is repeated for each subset of $R_2$ distributed to the leaf nodes.

**Division**

This is the implementation of $R_1 / R_2$ over $y$ and $z$. $R_1$ is a binary relation with attributes $x$ and $y$. $R_2$ is a unary relation with attribute $z$. The attributes $y$ and $z$ have the same underlying domain.

First $R_1$ is distributed across the leaf nodes. At each leaf node the subset of $R_1$ is sorted by $y$ values within $x$ values. For small numbers of unique values in $x$ a table of first occurrences of each unique value can help the next search step.

The tuples of $R_2$ are broadcast. As each $R_2$ tuple arrives at a leaf node it is compared against the tuples of $R_1$. Matching values in the $y$ field of $R_1$ cause that tuple to be marked.

When the end of relation markers for $R_2$ reaches a leaf node, that node sends (in sorted order) all values of attribute $x$ which appear in a matched tuple. Duplicate values of $x$ can be sent from the same leaf node. The $X$-values are sorted and merged as they travel to the root. No duplicates are removed however.

This sorted series of $X$-values is examined by the root. The root has counted the number of tuples of $R_2$ sent down the tree in a previous step. For a value of $X$ to be in the quotient set it must have as
many consecutive duplicate values as tuples of R₂. This makes the final decision for the root easy. It counts consecutive duplicate values of X and includes that value in the quotient set if the count is equal to the cardinality of R₂. The time for this operation is:
\[ dt + DT BL1 + \text{SORT } (N_1 / k) + DT BL2 + 2dt + N_1 t \]

**Union**

The UNION operation requires all source operations to have identical attributes. This implies that the UNION operation will be used after projection upon one or more source relations. The results of the projection must have identical attributes. For example:

\[(\text{PROJECT } R_j \text{ over } a^1) \cup (\text{PROJECT } R_j \text{ over } a^1)\]

where \(a_1\) is an attribute of \(R_1\) and \(R_2\).

Another example would be the union of two selected sets from the same source relation.

\[(\text{SELECT } R_1 \text{ where } a_1 = v_1) \cup (\text{SELECT } R_1 \text{ where } a_2 = v_2)\]

However, this may be implemented as one SELECT with the more complex criterion of:

\[\text{SELECT } R_1 \text{ where } (a_1 = v_1 \text{ or } a_2 = v_2)\]

This shows that a tradeoff exist between processing \(R_1\) twice and doing the UNION of the two results versus processing \(R_1\) once with a longer comparison time per tuple selection.

The UNION operation is also useful for insertion of tuples into a relation. To add the tuples, \(t\), to relation \(R_1\) the query expression \(R_1 \cup \{t\}\) can be executed.

The UNION operation can be implemented by combining the source
relations and removing duplicates. Various situations for the use of the UNION operation can have source relations on secondary store or within the leaf nodes.

If two relations are stored in leaf node memory, the tree-merge duplicate removal used for projection can complete the UNION operation. If the source relations are in secondary store, then a bit map duplicate removal such as in CAPS can be completed by the root. If this is insufficient then the overhead of distributing both relations across the leaf nodes must be incurred.

The response time for the bit map method where $R_1$ and $R_2$ are in secondary store follows:

$$DT_{BL1} + DT_{BL2} + t$$

For the removal of duplicates using the REPT tree-merge ability and with $R_1$ and $R_2$ on secondary store the response time can be expressed:

$$dt + DT_{BL1} + DT_{BL2} + \text{SORT} \left(\frac{N_1 + N_2}{k}\right) + 2dt + (N_1 + N_2)t$$

Intersection

The intersection of two relations may be implemented by the SELECT or JOIN operations. The intersection of two selections of the same relation can be implemented as the selection of the logical "and" of each criterion. For example:

$$\text{SELECT (R where } a_1 = v_1) \cap \text{SELECT (R where } a_2 = v_2)$$

can be implemented:

$$\text{SELECT (R where } a_1 = v_1 \text{ and } a_2 = v_2)$$
Suppose two different relations $R_1$ and $R_2$ are projected over the same attributes. The results of the two projections is then intersected. That is suppose:

$\text{(PROJECT } R_1 \text{ over } a_i) \cap \text{(PROJECT } R_2 \text{ over } a_i)$

must be implemented.

To do this every tuple of the projected $R_1$ subset must be compared to the projected $R_2$ subset. This can be done by an equi-join on the two subsets, where the projection of $R_1$ and the projection of $R_2$ are joined over all attributes in the tuples.

**Difference**

The set DIFFERENCE operation requires two source relations with identical attributes. The DIFFERENCE operation can be used to delete tuples from a relation. To delete tuple $t$ from relation $R_1$, the REPT can implement $R_1 - \{t\}$.

To implement the DIFFERENCE operation $R_1 - R_2$, the following steps can be executed. The relation $R_1$ is stored within the leaf nodes. It is then sorted at each leaf node. DDT of $R_2$ then is executed. As tuples of $R_2$ arrive at the leaf nodes they are compared to the tuples of $R_1$ stored in each leaf node memory. Matching tuples are removed from the stored subset of $R_1$. This can be accomplished by marking an associated mark bit in a removed tuple or by setting a bit in a table which holds a bit entry for each tuple in that leaf node.

When a leaf node receives the end of $R_2$ marker, then it routes the unmatched tuples of $R_1$ to its ascendant output port. These tuples are passed up the tree. The root receives this relation which is the
result of \( R_1 - R_2 \).

The DIFFERENCE operation with both \( R_1 \) and \( R_2 \) in secondary store executes in the following time:

\[
dt + DT_{BL1} + \text{SORT}(N_1 / k) + \text{maximum}(N_1 N_2 t_C / (2k), DT_{BL2}) + dt + N_t
\]

System commands

System commands are necessary to store relations in secondary store which have been updated. This is the interpretation of the STORE command. It uses the ability of the tree to transfer tuples from the leaves to the root.

A similar command can be used to transfer relations from the back-end to the host. This will be called the RETRIEVE command.

Complex Queries

Set and relational operations may be combined to form complex queries. The closure property of the relational algebra lets the results of one operation be the operand for the next. A node must be aware of the full query expression. The query expressions with multiple operations may vary the locations of source relations and the destinations of result relations.

Join, selection, and projection

Many times a selection and projection can reduce the number of tuples which will participate in a join operation. The following query is such an example.
JOIN PROJECT (SELECT $R_1$ where $a_1 = v_1$) over $a_2$

and (SELECT $R_2$ where $a_3 = v_2$) over $a_4$

over $a_5$ and $a_6$

We assume that $R_1$ and $R_2$ are in secondary store. Also, we assume that $N_2 < N_1$. First $R_2$ is read from secondary store and selected by the root node processor. The qualifying tuples of $R_1$ are either projected by a bit map technique or by the tree-merge duplicate removal. As an example we will assume the latter method of projection.

As the duplicate free sorted tuples are received at the root they are evenly distributed across the leaf nodes. This guarantees that the leaf nodes receive subsets of $R_2$ in sorted order. The join field should be used as the sort key.

The selection of $R_1$ is done as the tuples of $R_1$ are transferred off of secondary store. The root removes unwanted attributes from the tuples of $R_1$. These tuples are then distributed across the leaf nodes. This requires enough leaf node memory to hold $N_2F_2D_2 + N_1F_1$ tuples. A sort merge duplicate removal of these $R_1$ tuples is next. The root transmits the results of the selection and projection of $R_1$ to all leaf nodes.

The join is then completed as described in the section on join implementation. This operation takes the following time to complete:

$$(d + 1)t + DT BL2 + SORT(N_2F_2 / k) + 2dt + N_2F_2t$$

$$+ dt + DT BL1 + SORT(N_1F_1 / k) + 2dt + dt$$

$$+ \text{maximum}(N_1F_1t + N_t / N_1, N_1F_1D_1N_2F_2D_2 \cdot t_c + N_t / N_1,$$

$$N_2F_2D_2t_c / (2k) + N_t)$$
The ratios $F_1, F_2, D_1$ and $D_2$ reduce the number of transfers and comparisons to complete the join (see Appendix A).

REPT nodes have separate processors for control of output from node processor memory, control of input to node processor memory, and node processing. This enables the transfer of the tuples of $R_1$ to ascendants for duplicate removal, the receipt of duplicate free tuples being transmitted, and the comparisons for the join to be conducted concurrently.

**Projection, selection, and intersection**

For the query shown below, $R_1$ is assumed to be in secondary store. The subsets formed by two separate selections are projected and the result sets intersected.

\[
\text{PROJECT \ (SELECT } R_1 \text{ where } a_1 = v_1 \) over } a_2 \\
\cap \text{ PROJECT \ (SELECT } R_1 \text{ where } a_2 = v_2 \) over } a_2
\]

All of $R_1$ is distributed across the leaf nodes. The selections can be implemented as one operation where each tuple qualifies if $a_1 = v_1$ and $a_2 = v_2$. This is done in parallel in the leaf nodes as tuples arrive at ascendant input ports. The result sets are sorted at each leaf node. A tree-merge is then used to remove duplicate tuples. That is, the leaf nodes remove unwanted attributes before sending the tuples up to the root for the tree-merge. This query takes the following time to execute:

\[
dt + DT B_{11} + t_c + \text{SORT}(N_1 F_1 / k) + 2dt + N_1 F_1 t
\]
Multiple joins

The following is a query demonstrating three joins. All relations are assumed to be initially in secondary store.

\[
\text{JOIN (SELECT } R_1 \text{ where } a_1 = v_1 \text{) and} \\
\text{(JOIN } R_2 \text{ and (SELECT } R_3 \text{ where } a_2 = v_2 \text{) over } a_3 \text{ and } a_4 \text{) over } a_3 \text{ and } a_4
\]

First \( R_3 \) is distributed across the leaf nodes. As \( R_3 \) is transferred from secondary store the tuples are selected for distribution by the criterion \( a_1 = v_1 \). One tuple of \( R_2 \) is transmitted to the leaf nodes. This tuple may be joined to the tuples of \( R_3 \) in the leaves. The results of that join are stored locally at each leaf node as relation \( R_{23} \).

Next, all tuples of \( R_1 \) can be transferred from secondary store, selected at the root and transmitted to the leaf nodes. The leaf nodes perform the join comparisons of the arriving \( R_1 \) tuples and \( R_{23} \). Copies of the selected tuples of \( R_1 \) are stored within the tree. Any available node memory can store these tuples as long as no two nodes save copies of identical tuples.

The result tuples are transferred to the root. A "begin next cycle" symbol is sent from all leaf nodes as they finish the transfer of results to their ascendant output ports. This special symbol starts the transfer of the selected subset of \( R_1 \) up to the root. The root has already transmitted the next tuple of \( R_2 \). The root transmits the selected subset of \( R_1 \). This repeats for all tuples of \( R_2 \).

This implementation of a triple join query, transfers \( R_1 \) from
secondary store only once. It then stores the selected subset of \( R_1 \) in the tree. This \( R_1 \) subset is transmitted following each tuple of \( R_2 \). In this way only part of the first join of \( R_2 \) and \( R_3 \) is completed. The partial results are saved in leaf nodes and joined to the arriving tuples of \( R_1 \).

A balance between the portion of intermediate results stored and the number of repeated transmissions of \( R_1 \) must be found. The number of tuples produced from joining one tuple of \( R_2 \) to a subset of \( R_3 \) is data dependent. It can range from zero to the cardinality of the selected subset of \( R_3 \).

For a low average number of results produced, more than one tuple of \( R_2 \) can be transmitted between the DDT of \( R_1 \) tuples. This increases the number of intermediate result tuples to compare against the arriving tuples of \( R_1 \). In general \( R_1 \) is transmitted \( j \) times (for \( j \geq 1 \)). The time to complete the join can be expressed as follows:

\[
dt + \text{DT BL1} \quad \text{to distribute } R_3 \text{ across the leaves} \\
+ \text{DT BL2} \quad \text{to select and transmit the tuples of } R_2 \\
+ \text{DT BL1} + (j - 1)(N_t + dt) \quad \text{to transmit } R_1 \text{ } j \text{ times} \\
+ N_t \quad \text{to retrieve the result tuples for the front-end}
\]

**Conclusions**

Selection can be done as data enters the tree or concurrently within the leaf nodes. Projection can be done by a bit map method or making use of the tree-merge duplicate removal capability of the REPT. Join can be accomplished by distributing one relation across the leaves and transmitting the other relation from the root to all leaf nodes.
Parallel activities include the transmission of source relations, the comparisons of tuples, and the transmissions of result tuples. The tree-merge duplicate removal operations can be pipelined with the DDT of tuples for a join. This allows some concurrency between a projection followed by a join. The selection operations may produce tuples which are immediately transmitted for a join operation. This allows concurrency between selection and join.
CHAPTER V. NODE STRUCTURE

This chapter describes the X-TREE project and the design of the X-NODE. The information presented in the first section is meant as a summary of the X-TREE project. Following that a description of the use of X-NODES to support a relational database is given. The intent of the second section is to present the duties of each process within a node of REPT and to describe the differences between those processes and the general purpose ones of X-TREE.

X-TREE

The X-TREE project examined the problem of organizing multiple microprocessors into a general purpose computer architecture (Despain and Patterson 1978). A tree structure with extra interconnections was found to be attractive. These interconnections "ring" together processors. Each ring establishes horizontal communication across one level of the tree. The nodes of X-TREE, X-NODES were designed to meet the communication and processing needs of a general purpose, X-TREE machine (Sequin et al. 1978). The following is a summary of some of the X-TREE research.

X-TREE communication

In X-TREE a message may have to travel through many nodes to reach its destination and many messages will be traversing the network at the same time. Results of simulations lead the X-TREE researchers to the selection of a ringed binary tree structure. This structure
supports an efficient addressing scheme and allows a routing algorithm of relatively low complexity.

All X-NODES occupy a distinct location within a binary tree. The root is assigned address "1" and its two descendant nodes are assigned addresses "10" and "11". This address scheme continues down the tree to the leaves. An address is of variable length. This type of addressing allows for expansion of the tree without changing existing addresses.

Routing of messages from an input port of a node to an output port can be controlled by an algorithm based upon the destination address and a lookup table stored within the node. The relative position of the current node and the target node is classified into "higher", "lower" or "the same level". Horizontally six cases exist: "far left" or "far right", "near left", "near right" and "left or right line" (horizontal distance = 0). By examining the current and destination node addresses the proper output port can be chosen by consulting a lookup table under the proper horizontal and vertical entry.

One goal of the X-TREE project was to provide data paths throughout the tree that have at least the bandwidth of a modern disk. They found it necessary to transmit several bits in parallel. In analyzing projected pin constraints a byte-parallel transmission over a link was selected. Communication between nodes was asynchronous with a handshaking protocol.

Links are time multiplexed in X-TREE. Links must support concurrent traffic in both directions and also more than one message channel.
in the same direction. Time slots on links are assigned to channels which are actively transmitting messages. Each message using a particular link is identified by its "slot address".

The slot address is a special byte which has meaning for one particular link. It precedes any portion of the message which passes over that link. In each link which the message passes a new slot address is assigned by the local routing controller.

X-NODE structure

Each X-NODE contains a switching network and a node processor. This allows concurrent data processing and message transmission within a node. A X-NODE is diagramed in Figure 5.1. The switching network is comprised of the input buffers, input controllers, output buffers, output controllers, a multiplexed communication bus and a routing controller. The bus interconnects the buffer controllers and the routing controller. Communication links connect the external ports of a node to the ports of other nodes. A single link is multiplexed between carrying information to an input buffer controller or from an output buffer controller. The node processor is interfaced to buffer controllers as if it were another node link.

The X-NODE design separates message switching from node data processing. The processor at each node is not bothered by traffic which is only passing through the node. The switching network uses a time multiplexed bus. This is designed to be entirely on one VLSI circuit. This will enable a bandwidth of an order of magnitude higher within the chip then through the package pins. This is the
Figure 5.1. Logic diagram of a X-NODE
right ratio for the bus to serve five links, the main processor, and the routing controller. At each external port there will be a set of I/O buffers and two finite state machines. One of these finite state machines controls incoming messages and the other controls outgoing messages.

Several channels may be established through one input port. When a slot address arrives belonging to an established channel it is routed to the corresponding buffer. This implies that buffer space is managed for each existing channel.

Each input port maintains a look-up table indexed by slot address value. An arriving slot address which is not found in this table is sent to the routing controller. The routing controller assigns a slot address and designates the output port for this new message.

The output port controller monitors the bus. When the output port controller reads a slot address and data byte it enters the data byte into the proper buffer. Separate buffers must be maintained for each output channel. The output port immediately signals the input port either that the transfer was successful, that the buffer was full, or that a parity error occurred.

The output port concurrently selects a channel for transmission over the link. It strives to maintain the longest possible continuous byte stream.

The routing controller receives a message headed by a new slot address. The proper output port is determined from the routing algorithm. A new slot address is assigned. This new slot address is
used for transmission on the internal bus and across the proper link. The routing controller returns this assigned slot address and the output port designation to the input port which received the message. This input port enters this information into a look-up table. From then on the intra-node communication for this channel can be handled exclusively by the input and output port controllers.

Two other duties of the routing controller are to monitor the success rate of traffic over each link and to remove stale messages. If a particular channel has not transmitted any bytes within a certain time limit, its slot address is invalidated. The proper way to remove a channel is to transmit a "teardown" byte along the message channel. The corresponding slot address is removed from each look-up table as this byte is received.

Use of X-NODES in REPT

The bandwidth goal of X-TREE is identical to that of REPT. The relational operations can not proceed faster then tuples can be transferred from secondary store. The rate at which the tuples can be transferred from input port to output port and over the link must be at least as great as the rate which tuples are transferred from secondary store. Otherwise, an operation such as a join could proceed no faster then the second relation can be transmitted to the leaves and secondary store buffering would be required.

The instructions of REPT transfer relations either upward to the root or downward to the leaves. Multiple relations do not need to be
transmitted simultaneously across a link in the same direction. This reduces the overhead of assigning new slot addresses and increases the relative bandwidth within a node. The order of magnitude increase in bandwidth within a VLSI circuit implementation of an X-NODE over the bandwidth through the pins should allow tuples to move through nodes as fast as they are transmitted over links.

Implementing a machine language based on the relational algebra changes the method which the routing controller uses to assign output ports to relations. The assignment is made based on these three qualifications: the name of the relation which has arrived at the input port, the instruction involving the relation, and the identity of the current node. In contrast a routing controller of X-TREE bases its assignment of output port upon the current node address and destination node address of the message.

The format of relations and queries must be distinguishable by the routing controller. This can be accomplished by a special begin relation character and begin query character. The slot addresses identify the arrival of either a new relation or a new instruction to a node. The input port sends the new slot address and the next identifying character to the routing controller.

The routing controller responds to the arrival of a query at an input port by assigning a slot address and designating the processor buffer as the output port. The entire query is placed in the instruction queue of processor memory. The processor decodes the functions which the routing controller must perform for the query. These include the
channel assignment for source relations when they first come to an input port and the routing of result relations, if any, from the processor input port. The query format identifies the names of result relations. The operations to be performed upon a relation defines the destinations of source and result relations. The names and destinations of source and result relations are sent to the processor input port. This port identifies them as new slot addresses and sends the information to the router. The router responds with a no channel needed signal to the processor input port. In this way the router can build a look-up table of relation names and corresponding output ports.

Node processors

The duties of the node processor will always include decoding queries. What this processor does when particular relations enter its data queue depends upon the current query being executed and the nodes responsibilities in REPT. The current query can be defined as the most recently received query which uses a relation arriving or stored in processor memory. A particular node processor's responsibility is determined from the classification of the node. It may be an intermediate node or a leaf node. Each of the classes performs a different set of functions for the same query.

One method of organizing a node-processors activity is to designate three types of queues within the processors memory. One is the incoming relation queue where the tuples of incoming relations can be queued in memory. A second queue is for incoming queries. The third queue is for output from the node processor. Implementation of producer consumer
processes for input and output from a node processor memory would allow
concurrent pipelining of tuple arrival, comparison, and result tuple
output.

The node process is assumed to be continuously searching for
something to do with realtions which are in its relation queues. It
perpetually searches its relation queues from front to back searching
for a relation which can be used as a source operand. For unary
relational operations such as select or project, an entire relation
can be scanned in its queue. For binary relations, each relation
involved is treated as a separate queue. Thus, the join of two
relations proceeds from the front to rear of both relation queues
simultaneously. In this manner operations are performed when operands
arrive to the node processor memory.

Intermediate nodes

As an example suppose the query that follows is received at the
ascendant input port of an intermediate node in REPT:

JOIN R_1 and R_2 over a_1, a_2 producing R_3.

The query could be encoded:

BQ R_1 R_2 JOIN a_1 a_2 endj R_3 RETRIEVE EQ

where BQ and EQ are special "begin query" and "end query" characters
respectively. The new slot address assigned by the parent node would
cause the ascendant port controller to transmit the new slot address
and the BQ character to the routing controller. The routing controller
establishes a channel for the query. The query is transmitted to the
processor output port. The query is then entered into the instruction
queue of the processor.

The processor decodes the query. As an intermediate node the switching network must transfer \( R_1 \), when it arrives, from ascendant input ports to alternating descendant output ports. This is a special output port designation for \( R_1 \). This special designation, when entered in the look-up table for the ascendant input port will cause the opening of two channels for the incoming relation \( R_1 \). The tuples of \( R_1 \) can be alternately sent from the ascendant input port to the left and right descendant ports. This special output port designation requires the descendant input port to maintain a left-right bit for transmission. This bit is toggled each time the end of tuple marker for \( R_1 \) is received. This bit may then constitute part of the bus address.

The processor places the relation name \( R_1 \) and the alternate-descendant output port specification into its output queue and marks it with a new slot address. It then does the same for \( R_2 \) by designating both descendant ports for output. The result relation \( R_g \) is assigned the ascendant output port. The processor will not participate further in this query.

The end of relation markers for \( R_1 \), \( R_2 \) and \( R_g \) remove the look-up table entries in the routing controller for their respective channels. They also remove the look-up table entries in any input port receiving the relation.

**Leaf node processor**

The leaf nodes perform the same type of routing and query decoding as intermediate nodes. The interpretation of set and relational
operations in a leaf node processor differs from the interpretation in an intermediate node. The algorithm in Figure 5.2 describes the leaf node processor functions.

```
LEAF_NODE_PROCESS
Repeat
  If the query queue is not empty then begin
    Decode the query;
    Store the query in internal form;
    Send the output port designation to the router controller;
  End;
  If the relation queue is not empty then begin
    Find the first relation in the queue which can be used as a source operand;
    If the operation is binary then
      wait for the second relation to be stored in memory;
      Perform the operations on the relations;
      Store any results in the relation queue or the output queue;
  End;
Forever;
```

Figure 5.2. Description of a leaf node process

The internal storage of queries should facilitate the finding of any query using a specific relation. It also can replace instruction opcodes with routine addresses for the specific node routines to be executed. An inverted list of relation names used in queries can be maintained. However, a linear scan of the queue would probably be sufficient since large numbers of queries accumulating in a node processor is unlikely.

The order of execution of each operation within a query can depend on the order of arrival of relations to the processor memory. In general, the order in which the operations appear in the query can
reflect the order in which relations will be available in processor memory.

**Input port controller**

The algorithm in Figure 5.3 describes an input port process. It describes the actions of the input port controller for any node.

The input port process acts similarly to its counterpart in X-TREE when a new slot address is transmitted over the link. One difference is that the router can send back an output port designation for the new message which can require alternating tuples between two destination output ports. It can also designate that the tuples of a relation be sent to both descendant output ports. The increased overhead to alternate tuples between two descendant ports consists of an extra access to the left-right bit stored in the look-up table and the calculation of the output port using this bit. Both these operations can be done with little time increases.

**Output port controller**

The algorithm in Figure 5.4 describes an output port process. It describes the actions of any node output port.

The two tasks of the output buffer controller could be concurrent producer, consumer processes. One process reads the data bus and stores bytes in buffers. The other removes the bytes from the buffers and transmits them across the link. Independent processes could be implemented in hardware but they are not proposed as such in the X-TREE literature.
INPUT_PORT_PROCESS;
Repeat
If a byte B is received over the link then
Case B of
New_Slot_Address: begin
Wait until the next three bytes are received;
Transmit these four bytes to the routing controller;
Wait until the controller sends the assigned slot address and output port designation;
Enter the output port and slot address into the look-up table;
Transmit the new slot address and the two bytes received to the output port;
Establish a buffer for the new channel;
End;
Old_Slot_Address:
Designate the current input buffer to correspond to the Old_Slot_Address;
Otherwise:
If the buffer is not full then store the byte into the current input buffer;
Else send a retransmit signal over the link;
End Case;
If some input buffer is non-empty then
Begin
Choose the next input buffer for transmission;
Retrieve the Output_Port_Designation for the chosen input buffer slot address;
Case Output_Port_Designation of
Simple_Port: begin
Remove the next byte from the chosen input buffer;
Send this byte to the designated output port;
End;
Alternate_Descendants: begin
Retrieve the left-right bit for the chosen input buffer slot address;
Remove the next byte from the chosen input buffer;
If this byte is an end of tuple character then toggle the left-right bit and replace it in the look-up table;
If the left-right bit is set then transmit the byte to the left descendant;

Figure 5.3. Description of an input port process
Else transmit the byte to the right descendant;
End;
Both_Descendants: begin
  Remove the next byte from the chosen input buffer;
  Transmit the byte to the left descendant output port;
  Transmit the byte to the right descendant output port;
End;
End Case;
If the byte is an end of relation character then begin
  Remove the relation entry from the look-up table;
  Send the end of relation character and the slot address to the router;
End;
End;
Forever;

Figure 5.3. Continued
OUTPUT_PORT_PROCESS:
    Repeat
        If a slot address appears on the control bus which is
designated for this output buffer then begin
            If the buffer for that slot address is not full
            then begin
                Read the byte from the data bus;
                If a parity error occurred then transmit
                "error" to the input port;
            Else begin
                Place the byte into the buffer;
                Transmit "received" to the input port;
            End;
        End;
    End;
    If some buffer is non-empty then begin
        Choose the next buffer for transmission;
        Transmit the next byte in this buffer over the link;
    End;
Forever;

Figure 5.4. Description of an output port process

Choosing the next buffer for transmission can involve checking for
link transmission errors. The error signal arrives on the input link
time slot. The same buffer should be chosen if an error occurred. The
previously sent byte should not be removed from the buffer until this
error signal has been read.

Routing controller process

The algorithm in Figure 5.5 describes a routing controller
process. It describes the action of any node router controller.

In any node the router look-up table is used to control the
routing algorithm. No decisions need be made based upon addresses.
The node processor creates the entries for this look-up table by
decoding the query. The entries for this table are transmitted from
the processor to the router. For this transmission the message is transmitted labeled as the type Output_Port_Designation.

ROUTING_CONTROLLER_PROCESS;
Repeat
    If the address bus designates the routing controller then begin
        If the routing controller input buffer is not full then begin
            Store the byte on the data bus in the buffer;
            Store the address of the sending process in the buffer;
            Signal the sending process that the data has been received;
        End;
        Else send "buffer full" to sending process;
    End;
    If the routing controller input buffer is not empty then begin
        Remove slot address and sender process addresses from the buffer;
        Wait for the next three bytes from the sender process;
        Save these three bytes as Message Type, Relation_Name, and Output_Prot;
        Case message type of
            Query: begin
                Assign new slot address;
                Send slot address to sender process;
                Designate the processor-output-port as the destination;
                Send this designated output port to the sender process;
                Notify the processor-output-port of this new channel;
            End;
            Relation: begin
                Assign a new slot address;
                Send this slot address to the sender process
                Look-up the output port designation by the relation name;
                Send this designation to the sender process;
                Notify the designated output port of this new channel;
            End;

Figure 5.5. Description of a router process
Output_Port_Designation: begin
  Enter the output port by the relation name in
  the look-up table;
  Transmit "no channel needed" to the sender
  process;
  End;
End_of_Relation:
  Remove relation entry from look-up table;
End Case;
End;
Forever;

Figure 5.5. Continued

Conclusions

The logical design of the X-NODES is suitable for the nodes in
REPT. The support of a relational algebra based machine language
changes the algorithms used for routing data through a node. It does
not require VLSI circuit design changes.

The system software controlling the input port processors and the
routing controller would reflect the specialized purpose of REPT. The
routing controller maintains a look-up table on which it bases its
routing decisions for relational data. The entries for this look-up
table are created by the node processor. These entries are based
upon the decoding of queries by the processor.

The input ports are given the duty of implementing special output
port designations. The ascendant input port uses a left-right hit to
transmit alternate tuples of a relation first to the left-descendant
port then to the right-descendant port. The ascendant input port
also can transmit tuples of a relation twice. Once to each descendant
port. The addition of these tasks does not greatly increase the
overhead associated with transmission of tuples through a node.

The transmission of tuples from a descendant port of the node processor, comparisons of the tuples, and sending of result tuples to output ports can be implemented as a three stage pipeline. Thus, the analysis of response times includes a delay of twice the depth of the tree when executing a duplicate removal and merge operation. The time to transmit a tuple from a descendant port to the processor and the result tuple out of the ascendant port should take no longer then twice the time to transmit tuples from one node to another.
CHAPTER VI. PERFORMANCE ANALYSIS

The purpose of the following analysis is to compare the response times of CAFS, CASSM, DBC, RAP, DIRECT, REPT and HYPERTREE. The query chosen for analysis uses the projection, selection, and join operations. Assume a database with relations \( R_1 \) and \( R_2 \) stored on secondary store. The query, which every machine processes, can be stated as the following relational algebra expression:

\[
\text{JOIN} \left( \text{PROJECT} \left( \text{SELECT} \ R_1 \text{ where } a_1 = v_1 \right) \text{ over } a_2 \\
\quad \text{and} \ \text{PROJECT} \left( \text{SELECT} \ R_2 \text{ where } a_3 = v_2 \right) \text{ over } a_5 \right) \text{ over } a_2
\]

It is assumed that \( a_1, a_2, \) and \( a_5 \) are attributes (or sets of attributes) of relation \( R_1 \), and that \( a_3, a_4, \) and \( a_6 \) are attributes (or sets of attributes) of \( R_2 \). Also \( v_1 \) is in the domain of \( a_1 \) and \( v_2 \) is in the domain of \( a_2 \).

The above query requires processing of the important relational operations: SELECTION, PROJECTION, and JOIN. It is the same query used to analyze various JOIN implementations on a database management system (Blasgen and Eswaran 1977). Such a query was also included in the analysis of database machines for an explicit example based on data from an INGRES system (Hawthorn and DeWitt 1980).

A discussion of the operations involved to implement the query on each machine follows. After that examples are presented using realistic parameter values.
Response Time Expressions

This section is a description of the operations performed by each machine as it implements the previously stated query. A response time formula is then determined for each machine. Every response formula is based on parameters which are defined in Appendix A. These expressions are the basis for comparisons of the machines.

**CAPS**

CAPS first reads the smaller relation $R_2$. This requires the following time to transmit the subset of $R_2$ to the host:

$$DT \cdot BL_2 + t_C + t$$

The selection and projection are done in parallel with the read of other tuples of $R_2$ from secondary store.

The subset of $R_2$ in the host is now sorted while $R_1$ is read from secondary store. Once the selected, projected subset of $R_2$ is sorted, the host can compare arriving selected, projected, and implicitly joined tuples of $R_1$. The time to complete the join in CAPS is:

$$r_1 = \text{SORT}(N_2 \cdot F_2 \cdot D_2) + N_1 \cdot F_1 \cdot D_1 \cdot P_1 \cdot N_2 \cdot F_2 \cdot D_2 \cdot t_C / 2 \text{ if } N_2 \cdot F_2 \cdot D_2 \cdot t_C / 2 > DT / (QEF_1 \cdot P_1)$$

$$r_2 = DT \cdot BL_1 + N_2 \cdot F_2 \cdot D_2 \cdot t_C / 2 \text{ otherwise}$$

In the first instance the time to compare a tuple of $R_1$ to the subset of $R_2$ (in the host) is longer than the time to find a qualifying tuple of $R_1$ to compare. In the second instance the join comparisons take less time than reading $R_1$ tuples from secondary store.

The response time for CAPS to complete the query is:
CAPS = DT BL2 + tC + t
+ r1 if \[\frac{N_2 F_2 D_2 t_C}{2} > DT / (QEF_1 P_1)\]
+ r2 otherwise

**CASSM**

Numerous scans of R1 and R2 are needed to select, project and join the tuples. Extra rotations occur due to contention among cells for the RAM address bus or the output channel. The number of rotations, \(R(k, j)\), is a function of the number of cells and the amount of tuples responding to the search. \(R(k, j)\) has been estimated for RAP using Monte Carlo simulation (Ozkarahan et al. 1977). We will use these estimates for CASSM.

First, CASSM uses an unordered set search to mark tuples qualifying for selection. For \(i_2\) attributes in the selection criteria, \(1 + i_2\) rotations are required. Next, one more rotation will mark the attributes of the qualifying tuples which will constitute the join field.

For \(i_1\) attributes in the selection criteria of \(R_1\), \(1 + i_1\) rotations are needed to select the qualifying tuples. A further rotation is needed to mark the join field attributes of the selected \(R_1\) tuples.

Next, the intersection method is used to mark tuples in the implicit join. This assumes two RAMS are used. To read \(R_2\) and write a "true" to the hash encoded address in RAM1, takes \(1 + R(k, F_2)\) rotations. CASSM reads \(R_1\), hash encodes \(R_1\), and marks the tuples matching "true" at the hash address in RAM1. Now RAM2 is marked for intersection with \(R_2\). This operation requires \(1 + R(k, F_1)\) rotations. Finally, \(R_2\) is re-read and tuples are marked which have join fields that hash
encode to RAM2 addresses containing "true". This operation requires $1 + R(K,F_2)$ rotations.

This marks the tuples of $R_1$ and $R_2$ which will participate in the join. The total time discussed so far is:

$$r_1 = (2 + i_2) \ DTK \ BL2 + (2 + i_1) \ DTK \ BL1 + (2t_{cyl} / BC + (2 + 2R(K,F_2) t_b / k) \ BL2 + (t_{cyl} / BC + (1 + R(K,F_1)) t_b / k) \ BL1$$

A projection is accomplished by marking the projected fields of the subsets of $R_1$ and $R_2$ participating in the join. This is accomplished by a burst mode search. The time for this operation is $DTK \ BL1 + DTK \ BL2$.

A pointer search must be conducted for any character string attributes. This requires $r_2 = (1 + R(K,F_1,F_2)) \ DTK \ BL1 + (1 + R(K,F_2,F_2)) \ DTK \ BL2$ seconds.

The tuples which are to be sent to the host are now marked. Duplicates from the projection operations have not been removed. Three different approaches can be taken to complete the query. Unique output of tuples removes duplicates in CASSM. Alternately, duplicates may be removed in the host either before or after the join is completed. The alternative formulas CASSM1, CASSM2, and CASSM3 each include the times for execution of different duplicate removal methods.

**CASSM1**

To complete the query duplicates may be removed in CASSM. Each tuple of $R_2$ now marked can be compared against the rest of $R_2$ to unmark
duplicates in secondary store. This takes the time of DTK BL2
+ N₂F₂P₂πDTK BL2 where π increases the number of tuples in N₂F₂P₂ to
account for hashing collisions. Similarly, the unique output of R₁
takes DTK BL₁ + N₁F₁P₁π DTK BL₁ time units to unmark duplicates of the
selected, projected, implicitly joined subset of R₁.

While R₁ duplicate tuples are removed, the host is sorting the
subset of R₂. As tuples of R₁ arrive from CASSM, the host completes
the join. Since the search of R₁ is lengthy, the comparisons for a
join are done concurrently with the repeated reads of R₁. The response
time for CASSM using the unique output operation can be expressed:

CASSM₁ = r₁ + r₂ + (1 + N₂F₂P₂π) DTK BL₂ + (1 + N₁F₁P₁π) DTK BL₁

CASSM₂

This variation outputs the tuples of R₂ to the host. It then
sorts this subset and removes duplicates. This last operation is done
concurrently with the output of tuples of R₂ to the host. This
operation requires the following time:

(t_cyl / BC + (1 + R(k,F₂P₂π))tₜ / k) BL₂
+ maximum ((t_cyl / BC + (1 + R(k,F₁P₁π))tₜ / k) BL₁,

SORT(N₂F₂P₂D₂π) + N₂F₂P₂D₂TN₁F₁P₁πt_c / 2)

The final step is to sort the result tuples to remove duplicates
from the projection of R₁. This requires the time SORT(N) where N is
the cardinality of the result set. The response time for CASSM₂ can be
expressed:
CASSM2 = \( r_1 + r_2 + \left( \frac{t_{cy1}}{BC} + \frac{1 + R(k,F_2^2)}{k} \right) t_b / k \) BL2

+ \text{maximum} \left( \left( \frac{t_{cy1}}{BC} + \frac{1 + R(k,F_1^1)}{k} \right) t_b / k \right) \text{BL1},

\text{SORT}(N_2^2R_2D_2^2 + N_2^2R_2D_2^2TN_1F_1P_1\pi t_c / 2)

+ \text{SORT}(N)

CASSM3

The host does the duplicate removal of both subsets of \( R_2 \) and \( R_2 \), individually, before completing the join. The subsets of \( R_2 \) is transferred first.

\( \left( \frac{t_{cy1}}{BC} + \frac{1 + R(k,F_2^2)}{k} \right) t_b / k \) BL2

Both duplicate removal of the subset of \( R_2 \) in the host and transmission of the subset of \( R_1 \) to the host can be done simultaneously. This operation takes:

\text{maximum} \left( \left( \frac{t_{cy1}}{BC} + \frac{1 + R(k,F_1^1)}{k} \right) t_b / k \right) \text{BL1},

\text{SORT}(N_2^2R_2^2)

Next duplicates are removed from the subset of \( R \) in the host:

\text{SORT}(N_1F_1P_1\pi)

Finally, the join is completed:

\( N_2^2R_2D_2^2TN_1F_1P_1D_1\pi t_c / 2 \)

The response time formula for CASSM3 is:

CASSM3 = \( r_1 + r_2 + \left( \frac{t_{cy1}}{BC} + \frac{1 + R(k,F_2^2)}{k} \right) t_b / k \) BL2

+ \text{maximum} \left( \left( \frac{t_{cy1}}{BC} + \frac{1 + R(k,F_1^1)}{k} \right) t_b / k \right) \text{BL1},

\text{SORT}(N_2^2R_2^2)

+ \text{SORT}(N_1F_1P_1\pi)

+ N_2^2R_2D_2^2TN_1F_1P_1D_1\pi t_c / 2 \)
The DEC has no facilities for implementing a join operation within the microprocessor per head PCAM. The assumption is made that the DEC would first transfer a selected subset of $R_2$ to the host. The host would project this subset of $R_2$ and remove duplicates. This operation would take:

$$DTK \ BL2 + kF_2H_2t + SORT(N_2F_2)$$

where $kF_2H_2t$ is the time to transfer the last selected tuples from the PCAM to the host.

Next, a subquery is executed for every tuple of the subset of $R_2$ in the host. This time is calculated by:

$$N_2F_2D_2 \ DTK \ BL1 + kF_1H_1t$$

Duplicates are removed from this subset in the front-end taking time:

$$SORT(N_1F_1P_1)$$

Finally, the join is completed in the host in the time:

$$N_1F_1P_1D_1H_1N_2F_2P_2D_2H_2t_C / 2$$

The response time formula for DEC is:

$$DEC = DTK \ BL2 + kF_2H_2t + SORT(N_2F_2)$$

$$+ N_2F_2D_2 \ DTK \ BL1 + kF_1H_1t$$

$$+ SORT(N_1F_1P_1) + N_1F_1P_1D_1H_1N_2F_2P_2D_2H_2t_C / 2$$

RAP

RAP uses the cells for selection. It can have contention for output like CASSM (Ozkarahan et al. 1977). First RAP must transfer $R_2$ from secondary store to the CCDS in the time:
Then it selects and marks tuples for each CCD taking the time:

$$2.5 \frac{t_m c_2}{k}$$

The get-next-mark operation could not be used for READ out to the host. This is because all the relation $R_2$ may not fit in the CCDS. A simple output procedure is used which has contention for the output channel:

$$t_m \frac{c_2}{k} R(K_1 F_2)$$

The host projects this selected subset of $R_2$ which requires the time:

$$\text{SORT}(N_2 F_2)$$

While the above sort is being done in the host, the first transfer to the CCDS may be accomplished. The following time is taken for $R_1$ to be transferred to CCDS:

$$\text{DT} (BL1 - k C / Q)$$

Here $C / Q$ is the number of blocks per CCD. The selection and marking of tuples of $R_1$ while they are in the CCDS requires the time:

$$2.5 \frac{t_m c_1}{k}$$

A subquery is made for each tuple of the selected, projected subset of $R_2$ in the host. This requires the following time:

$$N_2 F_2 D_2 \frac{t_m c_2}{k}$$

The marked tuples of $R_1$ are readout which takes the time:

$$C_1 \frac{t_m}{k} R(K, F_1 P_1)$$

The projection and join are completed in the host which takes the time:

$$\text{SORT}(N_1 F_1 P_1) + N_2 F_2 D_2 N_1 F_1 P_1 D_1 t_c / 2$$

The response time formula for RAP follows:
\[
\begin{align*}
\text{RAP} &= DT \ BL2 + kF_2H_2t + \text{SORT}(N_2F_2) \\
&\quad + DT (BL1 - kC / Q) + 2.5 t_m C_1 / k \\
&\quad + N_2F_2D_2t_m C_2 / k + C_1R(k, F_1P_1)t_m / k \\
&\quad + \text{SORT}(N_1F_1P_1) + N_2F_2D_111111t_c / 2
\end{align*}
\]

**DIRECT**

Direct has two main steps in executing the query. First, the smaller relation, \( R_2 \), is read from secondary store to the CCDS. Assuming single port CCDS, the transfer of tuples from CCDS to query processors is not done concurrently with the transfer of tuples from secondary store to CCDS. The query processors all read the same CCDS. They do the selection and projection of \( R_2 \). Each query processor then contains the same subset of \( R_2 \). This first step takes the following time:

\[
x_1 = DT \ BL2 + t_m C_2 + 2t_c + \text{SORT}(N_2F_2)
\]

The second step is to read \( R_1 \) from secondary store into CCDS and complete the join in the query processor. Each query processor may compare the subset of \( R_2 \) to a different subset of \( R_1 \). The results are transferred to output buffers for each query processor. The tuples in the output buffers are transferred to CCDS when they become full. The tuples in the CCDS are transferred to secondary store and from secondary store to the host. Various phases of this three step pipeline can dominate: the reading of \( R_1 \) from secondary store, the comparison of each tuple of \( R_1 \) to the subset of \( R_2 \) in the host, or the output of the result tuples.
For the case where fewer results are produced than tuples read from secondary store we have \( N < N_1 \). In this case either the comparison time or the input of tuples dominates. This gives the following times for \( N < N_1 \):

\[
\begin{align*}
    r_2 &= t_c + N_1 F_1 N_2 F_2 D_2 / (2k) + t_m + k t_c / Q \\
    &\quad \text{for } N_2 F_2 t_c / 2 > DT / (QEF_2) + t_m / (CEF_2) \\
    r_3 &= DT B_L + t_m C / k + t_m + k t_c C / Q \\
    &\quad \text{otherwise}
\end{align*}
\]

In both expressions \( t_m + k t_c C / Q \) is the time to output the last result tuple to the secondary store.

When the number of result tuples is larger than the number of tuples of \( R_1 \) we have \( N > N_1 \). The join step takes one of two times depending on whether the comparison phase or output phase dominates. First when \( N > N_1 \) it can take the following time to output result tuples:

\[
\begin{align*}
    r_4 &= t_c / (k F_2) + N_2 F_2 t_c / 2 + N_1 t_m / N + \lceil N / (CE) \rceil t_m / k \\
    &\quad + \lceil N / (QE) \rceil DT \quad \text{where } N_2 F_2 / 2 t_c < (DT / (QE) + t_m / (CE)) N / N_1
\end{align*}
\]

Here the output of results takes longer than the comparison of tuples of \( R_1 \) to the subset of \( R_2 \) stored in the query processors. The term \( t_c / (k F_2) \) is the time until the next selected tuple is found in the CCDS. \( N_2 F_2 t_c / 2 \) is the time for the first join comparison. \( N_1 t_m / N \) is the time until the first result tuple is produced for output. The last two terms represent the time to output the results.

The next expression represents the time for the join when the comparison of tuples takes longer than the output of results. The time
in this case for \( N > N_1 \) can be expressed:

\[
 r_5 = t_c + N_1 F_1 N_2 F_2 D_2 t_c / (2k) + t_m + k t_b C / Q
\]

where \( N_2 F_2 t_c / 2 > (DT / (QEF_1)) + t_m / (CEF_1) N / N_1 \)

The total response time for \( \text{DIRECT1} \) also includes the time to transfer the results from secondary store to the host and the time for duplicate removal. An expression for this time follows:

\[
 DT \left\lceil N / (QE) \right\rceil + \text{SORT}(N)
\]

The response time formula for \( \text{DIRECT1} \) follows:

\[
 \text{DIRECT1} = r_1 \\
 + r_2 \text{ for } N < N_1 \text{ and } N_2 F_2 t_c / 2 > (DT / (QEF_1)) + t_m / (CEF_1) \\
 + r_3 \text{ for } N < N_1 \text{ and } N_2 F_2 t_c / 2 < (DT / (QEF_1)) + t_m / (CEF_1) \\
 + r_4 \text{ for } N_1 < N \text{ and } N_2 F_2 t_c / 2 < (DT / (QEF_1)) + t_m / (CEF_1) ) N / N_1 \\
 + r_5 \text{ for } N_1 < N \text{ and } N_2 F_2 t_c / 2 > (DT / (QEF_1)) + t_m / (CEF_1) ) N / N_1 \\
 + DT \left\lceil N / (QE) \right\rceil + \text{SORT}(N)
\]

\( \text{DIRECT2} \)

An alternative in \( \text{DIRECT} \) is to have the host remove duplicate tuples and complete the join. An implicit join is done by \( \text{DIRECT} \). The same \( r_1 \) time is included as for the \( \text{DIRECT1} \) implementation. This is for the transfer of \( R_2 \) to the query processor for selection and projection.

Transfer of results to the host may go on in parallel with input of \( R_1 \) tuples to the query processors and the comparisons for the join. When the input phase dominates the response time we have:

\[
 r_2 = DT BL1 + t_m C_1 / k + t_b k C / Q
\]

for \( N_2 F_2 D_2 t_c / 2 < (DT / (QEF_1)) + t_m / (CEF_1) \)
When the comparison of tuples takes the longest time we have:

\[ r_3 = N_1F_1N_2F_2D_2t_C + t_m + k t_b C / Q \]

for \( N_2F_2D_2t_C / 2 > DT / (QEF_1) + t_m / (CEF_1) \)

The final step transfers the result tuples to the host, removes duplicate tuples and completes the join. The time for these operations follows:

\[ r_4 = DT N_1F_1P_1 / (QE) + \text{SORT}(N_1F_1P_1) + N_1F_1P_1D_1N_2F_2P_2D_2t_C / 2 \]

The response time formula for DIRECT2 can be expressed:

\[
\text{DIRECT2} = r_1 + r_2 \text{ for } N_2F_2D_2t_C < DT / (QEF_1) + t_m / (CEF_1) + r_3 \text{ otherwise} + r_4
\]

The input of \( R_2 \) into \( \text{REPT} \) requires the following time:

\[(d + 2)t + DT BL2\]

The extra two transfers are required by the selection and projection of the tuples in the root. The selected and projected subset of \( R_2 \) is distributed evenly over the leaves. The time to sort the individual subsets is given below:

\[
\text{SORT}(N_2F_2D_2 / k)
\]

Three alternative expressions can be found for the response time of \( \text{REPT} \). The first is the reading of \( R_1 \) from secondary store and the selection and projection of \( R_1 \) at the root. This time can be expressed:

\[ r_1 = DT BL1 + N_2F_2D_2t_C / (2k) + dt + h \]

where \( 0 \leq h \leq N_2F_2P_2t \)
The parameter $h$ is the time to transfer the remaining result tuples to the front-end.

Another parallel activity is the comparison of tuples at the leaf nodes. The time for this can be expressed:

$$r_2 = N_1F_1D_1N_2F_2D_2t_C / (2k) + dt + h$$

The parameter $h$ is the same as in the previous expression.

The third parallel activity is the output of result tuples. The execution time for this phase can be expressed:

$$r_3 = N_2F_2D_2t_C / (2k) + Nt + dt$$

The total response time for REPT can be expressed:

$$REPT = (d + 2) t + DT B2 + SORT(N_2F_2D_2) + \text{maximum}(r_1, r_2, r_3)$$

**HYPERTREE**

The first step involves reading both relations and creating the merged map at the root. The merging of the map from the relation $R_2$ goes on while $R_1$ is being read. The time for this activity follows:

$$r_1 = DTK BL1 + DTK BL2 + dt + 2ydte / BP$$

where $y = \text{maximum}(\text{minimum}(T_1N_1, T_2N_2), \text{minimum}(F_1N_1, F_2N_2))$

The parameter $y$ is the number of bits in each bit map. The $dt + 2ydte / BP$ is the time spent merging the second table up to the root. Each table is $2y$ bits in size. The time to transmit the result table to the leaves can be expressed:

$$2ydte / BP$$
The two relations must be read again to determine the subset of tuples to be sent to arbitrary nodes. This requires the following time:

\[ r_2 = DTK BL1 + DTK BL2 \]

The transfer of tuples evenly over the nodes can be entirely done in parallel with the reading of \( R_1 \) and \( R_2 \), or it may take longer. This is expressed in the following:

\[ \text{maximum}(r_2, r_3) \]

where \( r_3 = d_{ave} t (N_1F_1P_1D_1\pi + N_2F_2P_2D_2\pi)(2k - 2) / (2k - 1) \)

The expression, \( r_2 \), is a best case possibility. The tuples matching at the end of reading \( R_2 \) may still be in transit to a chosen node when all \( R_2 \) has been read at the leaves. As tuples of \( R_1 \) and \( R_2 \) are read from disk the leaf nodes perform the selection. The unwanted attributes are removed at this time. Duplicate tuples are removed in the destination nodes of the implicitly joined tuples. The expression, \( r_3 \), is the time spent to transfer the implicitly joined subsets of \( R_1 \) and \( R_2 \) to chosen nodes. The average length path traversed by one of these tuples can be expressed:

\[ d_{ave} = 5d / 4 - 4 / 3 + 4 / 3 2^{-d} - d \text{ mod } 2 / 12 \]

The next step is to remove duplicates. This step is done by all nodes at once and requires the execution time:

\[ r_4 = \text{SORT}(N_1F_1P_1\pi) / (2k - 1) + \text{SORT}(N_2F_2P_2\pi) / (2k - 1) + (N_1F_1P_1\pi + N_2F_2P_2\pi)t_c / (2k - 1) \]

Finally, all nodes complete the join and transmit the results to the root node. This takes the following time:
The total response time for HYPERTREE follows:

\[ H_{-\text{TREE}} = r_1 + 2ydtE / BP + \text{maximum}(r_2, r_3) + r_4 + r_5 \]

After studying the join algorithm previously described it became of interest to analyze an alternative join implementation in HYPERTREE. The alternative would be to keep the larger relation, \( R_1 \), stationary and move the tuples of the smaller relation, \( R_2 \), to the leaf nodes which require them. HYPERTREE can have a ring interconnection of the leaf nodes to facilitate such communication. The next section describes the response time of such an implementation.

**HYPERTREE alternative join implementation**

Suppose HYPERTREE is to join two relations \( R_1 \) and \( R_2 \). It could proceed as follows. First, the larger relation, \( R_1 \), is read by the leaf nodes. As before a one bit wide array is marked with "true" values to store information about the join fields of tuples of \( R_1 \). Each leaf node creates an independent table from its subset of \( R_1 \) tuples. A table is retained by each leaf node. A merged table is created as previously described. This table is transmitted to every leaf node. Each leaf node now contains a table of global \( R_1 \) information and a table of local \( R_1 \) information.

Next, the tuples of \( R_2 \) are read by the leaf nodes. In parallel leaf nodes hash encode the join fields of tuples of \( R_2 \). The resultant address is used to read the local table of \( R_1 \) information. A tuple whose join field hash encodes to a local bit array location which con-
tains a "true" value will be stored immediately. Every tuple of \( R_2 \) is also used to access the global table of \( R_1 \) information. If a "true" value is retrieved the tuple is circulated around a ring connecting the leaf nodes. To expedite circulation, information is concatenated to each tuple. Each tuple of \( R_2 \) is sent to adjacent leaf nodes accompanied by the node address of its origin and the hash encoded join field values. A leaf node receiving a tuple on a ring link accepts the tuple if it did not originate at that node. The leaf node uses the accompanying hash encoded join field values as an address to read the local bit array for \( R_1 \). A "true" value in that location causes the tuple of \( R_2 \) to be stored in that leaf node memory. This implies that a leaf node will store tuples of \( R_2 \) which will be joined to tuples of \( R_1 \) stored in the disk platter interfaced to that leaf node.

The tuples of \( R_2 \) stored at each leaf node are sorted. The tuples of \( R_1 \) are transferred from secondary store into the leaf nodes. Each tuple of \( R_1 \) is compared to the tuples of \( R_2 \) stored in the local leaf node memory. Tuples with matching join fields are concatenated and either stored or forwarded to the user. Since the result relation is created in the leaf nodes it is very easy to save it in secondary store. Although in that case there is no guarantee that each track of a cylinder would be totally utilized. In this join method the result tuples are also in position for an additional join operation using the result relation as an operand.
Now consider the response time for the alternative join algorithm previously described. First the relation $R_1$ is read, the merged table is created and this table is transmitted to all leaf nodes. The time for this operation follows:

$$r_1 = DTK \cdot BL1 + dt + 4ydtE / BP$$

The time to read $R_2$ and circulate the implicitly joined tuples of $R_2$ to all leaf nodes can be expressed:

$$\max(r_2, r_3)$$

where $r_2 = DTK \cdot BL2 + h_1$ for $0 \leq h_1 \leq kt$

and $r_3 = DT / (QUF_2P_2) + N_2F_2P_2^{4kt}$

The first expression is maximum when the time to transfer $R_2$ from disk takes longer then circulating the tuple of $R_2$. The second expression represents the opposite case. The parameter $h_1$ is the time to circulate tuples of $R_2$ after $R_2$ has been read from secondary store. The first term of the second formula represents the average time until the first joinable tuple of $R_2$ is transferred from secondary store.

To complete the join the subsets of $R_2$ are sorted and $R_1$ is re-read. The time of the sort phase depends on the average number of tuples of $R_2$ stored at each leaf node. This depends on the distribution of $R_1$ join field values across the leaf nodes.

The number of tuples stored at some leaf node is a function of the number of tuples in $R_1$ and $R_2$, the number of leaf nodes, and the distributions of the join field values in $R_1$ and $R_2$. The cardinality
of the selected, implicitly joined subset of $R_2$ is $N_2F_2P_2$. We now need to know the average number of tuples of this relation which are stored at some leaf node.

First we derive the average number of leaf nodes storing some $x$-valued tuple from the implicitly joined subset of $R_2$. An $x$-valued tuple is defined to be a tuple which has $x$ as its join field attribute value. This means that a $x$-valued tuple of $R_2$ will be stored at the $i^{th}$ leaf node if $x$ is hash encoded to the same bit array locations as the join field value of some tuple of $R_1$ stored in a disk track accessible by the $i^{th}$ leaf node.

We will assume that the probability of an $x$-valued tuple being stored in $R_1$ is $P(x)$. The average number of $x$-valued tuples in $R_1$ is given by $E(N_1, x)$. We assume that there are $k$ leaf nodes and each leaf node accesses portions of secondary store which contain identical numbers of $R_1$ tuples. The tuples of $R_1$ are randomly chosen for storage at any particular leaf node. This means each leaf node accesses $N_1 / k$ tuples of $R_1$.

The probability, $P_1$, that a particular leaf node contains all of the $x$-valued tuples of $R_1$ can be expressed:

$$P_1 = \frac{\binom{N_1 / k}{E(N_1, x)}}{\binom{N_1}{E(N_1, x)}}$$

This is the ratio of the total number of combinations of $E(N_1, x)$ $x$-valued tuples chosen from $N_1 / k$ tuples to the total number of combi-
nations of $E(N_1,x)$ x-valued tuples chosen from $N_1$ tuples. The cardinality of the subset of $R_1$ tuples accessed by one leaf node is assumed to be $N_1 / k$.

The probability that all x-valued tuples of $R_1$ are stored only in the secondary store accessed by two particular leaf nodes can be expressed:

$$P_2 = \frac{2N_1 / k - 2\left(\frac{N_1}{k}\right)}{E(N_1,x)} - \frac{1}{E(N_1,x)}$$

The same probability for three leaf nodes can be expressed:

$$P_3 = \frac{3N_1 / k - 3\left[2\left(\frac{N_1}{k}\right) - \frac{1}{E(N_1,x)}\right]}{E(N_1,x)} - \frac{1}{E(N_1,x)}$$

The probability that all x-valued tuples in $R_1$ are stored only in the secondary store accessed by $i$ particular leaf nodes can be expressed:

$$P_i = \frac{iN_1 / k - \left(\begin{array}{c} i \\ 1 \end{array}\right)P_{i-1} - \left(\begin{array}{c} i \\ 2 \end{array}\right)P_{i-2} - \cdots - \left(\begin{array}{c} i \\ i \end{array}\right)P_1}{E(N_1,x)}$$

For analysis purposes we assume a uniform distribution of all the possible values in the domains of $X$, over the tuples of $R_1$. Then for $U_1$ unique combinations of values from the domains of $X$ we have
P(x) = 1 / U_1 for all join field values x. This means that E(N_1, x) = N_1 / U_1.

The mean number of leaf nodes storing a particular x-valued tuple in R_2, L(x), can be calculated by the expression:

$$L(x) = \sum_{i=1}^{k} \binom{k}{l} P_i$$

Note that L(x) is a function of k, N_1, and U_1. If we assume that L(x) is the same for all x-values in R_2 we can calculate the number of tuples of R_2 stored in all leaf nodes (including copies). That total is below:

$$N_2 F_2 P_2 L(x) \pi$$

Since each leaf node will store, on the average, the same number of tuples of R_2 we have the following number of tuples of R_2 stored at each leaf node:

$$N_2 F_2 P_2 L(x) \pi / k$$

In the worst case each tuple of R_2 must be stored at every leaf node. In this case L(x) = k and \( \pi = 1.0 \). The last formula reduces to N_2P_2 tuples. It can be seen from Figure 6.1 that L(x) = k is a likely condition.

To complete the join operation each leaf node sorts its set of R_2 tuples. This takes the time shown below:

$$r_4 = \text{SORT}(N_2 F_2 P_2 L(x) \pi / k)$$

The comparisons to complete the join of the time to read R_1 will dominate the final step. This time is expressed:

$$\text{maximum}(r_5, r_6)$$

where

$$r_5 = DTK BL1 + N_2 F_2 P_2 L(x) \pi \tau_c / (2k)$$
<table>
<thead>
<tr>
<th>Number of x-valued tuples in $R_1$</th>
<th>$L(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>15.9</td>
</tr>
<tr>
<td>500</td>
<td>16</td>
</tr>
<tr>
<td>1000</td>
<td>16</td>
</tr>
<tr>
<td>5000</td>
<td>16</td>
</tr>
<tr>
<td>10000</td>
<td>16</td>
</tr>
<tr>
<td>50000</td>
<td>16</td>
</tr>
<tr>
<td>120000</td>
<td>16</td>
</tr>
</tbody>
</table>

$N_1 = 120000$

$k = 16$

Figure 6.1. The mean number of leaf nodes storing an x-valued tuple of $R_2$
and \( r_6 = DT / (QEF_1P_1) + \frac{N_1F_1P_1}{k} \frac{N_2F_2P_2L(x)}{k} t_c / (2k) \)

Following is the response time formula for the alternative implementation of the join in HYPERTREE:

\[
H\text{-TREE}2 = r_1 + \text{maximum}(r_2, r_3) + r_4 + \text{maximum}(r_5, r_6)
\]

Response Time Comparisons

This section strives to compare the various machine responses to the query previously defined. Various graphs depict the response times for machines given specific values of the data dependent parameters, \( F, P, N_2 \) and \( k \).

Parameter values

Specific values have been chosen for the machine dependent parameters. This is to create an example situation in which the various machines can be compared. The following values were chosen:

\[
\begin{align*}
t &= .000321 \text{ seconds} \\
t_p &= .0008 \text{ seconds} \\
t_{cyl} &= .01 \text{ seconds} \\
t_c &= .00002 \text{ seconds} \\
t_m &= .012 \text{ seconds} \\
Q &= .125 \text{ pages/block} \\
C &= 4 \text{ pages/CCD} \\
E &= 20 \text{ tuples/pages} \\
BC &= 418 \text{ blocks/cylinder} \\
BP &= 32,768 \text{ bits/page}
\end{align*}
\]
These secondary store times and sizes are based upon an IBM 3330 series 11 disk pack. The tuple transfer time is identical to the rate at which tuples are transferred from secondary store. The comparison time is calculated from a 1 microsecond/byte comparison rate. The estimate of 20 bytes participating in the comparison is made for this example.

The relationship between $t$, $t_C$, and $DT$ has been found to be critical to the response of REPT and HYPERTREE. For purposes of demonstration $t = DT / QE$ and $t_C < t$ in the examples discussed in the following sections.

The rotation time for CCDS and $t_m$ was taken from a paper about DIRECT (Hawthorn and DeWitt 1980). The CCD size of 16K bytes was also selected from that paper. We consider the following parameters to be fixed:

$$H_1 = H_2 = .5$$
$$D_1 = D_2 = 1$$
$$T_1 = T_2 = .01$$

The following simplifications have been made:

$$F_1 = F_2 = F$$
$$p_1 = p_2 = p$$
$$N_1 = 4N_2$$

The graphs are plotted on single cycle log paper. The y-axis represents the $\log_{10}$ (seconds to respond to the query). Thus a value of 2 on the vertical axis represents a response time of 100 seconds. A value of 4 represents a response time of 10000 seconds.
Each graph shows an ideal line. This represents the time to transfer the source relations from secondary store. This is an ideal time for an indirect search machine to complete the join. On a whole, indirect search machines have a greater response time than the ideal time. This is due to comparisons and transmissions of tuples which cannot be executed concurrently with the transfer of data from secondary store.

An ideal line for a MPDS or MPCS machine with k cells lies below ten seconds. For this reason it is not plotted. The MPDS and MPCS machines have the inherent advantage of reading the data in \( \frac{1}{k^{th}} \) the number of revolutions as an MPIS, SPDS or SPIS machine.

A great saving in time could occur by using an MPCS or MPDS machine instead of the others. This can be seen by examining the ratio of the ideal times:

\[
\frac{DTK (BL1 + BL2)}{(BL1 + BL2)}
\]

When evaluated using the parameter values for our example this ratio evaluates to .09.

Example 1: Small Selected Sets

Figure 6.2 graphs response time versus \( P \) with \( P \) varying from .01 to 1.0, \( N_2 = 30,000, F = .01 \) and \( k = 16 \). In this case \( R_1 \) and \( R_2 \) fill 16 platters of an IBM 3330 disk pack. Thus, the MPDS and MPCS systems are operating near capacity with 16 cells.

The small \( F \) reduces the number of tuples qualifying for the selection to 1% of the original cardinality. That is \( N_2 F_2 = 300 \) tuples
and $N_1F_1 = 1200$ tuples. With $F = .01$ most of the comparison time can be absorbed into the concurrent reading of source tuples from disk. This is why CAFS, REPT-B, DIRECT-B and DIRECT2 are so near the ideal time. A suffix of -B implies a best case number of result tuples. A suffix of -W implies the worst case. That is $N = N_1F_1P_1D_1$ tuples in the best case and $N = N_1F_1P_1D_1N_2F_2P_2D_2$ tuples in the worst case.

**CAFS**

In Figure 6.2 CAFS does not increase in response time with increasing $P$. The sort and comparison in the host may be done in parallel with the search for the next qualifying tuple of $R_j$ on secondary store. In this instance many tuples must be searched by CAFS before one is sent to the host. This allows the host to complete its work while CAFS accesses secondary store.

**CASSM**

The large response time for the implementation, CASSM1, is due to the duplicate removal operation done within CASSM. The numerous subqueries are not as efficient as duplicate removal in the host. This is seen in the CASSM3 implementation. For $P = 1$, CASSM1 is 100 times slower than CASSM3.

The last step of CASSM2 is a sort of the result tuples to remove duplicates. This is done in the host. For a large number of result tuples this is costly. The lines CASSM2-B and CASSM2-W demonstrate the increase in response time for large result sets. For the best case it is advantageous to use CASSM2 over any other CASSM algorithm.
Figure 6.2. Response times for p = 0.01

Log₁₀ (Seconds to Response)
This is because the comparison of the tuples can begin as the subset of $R_1$ arrives at the host.

CASSM3 removes duplicates from both source relation subsets in the host. Comparisons cannot be done as tuples of $R_1$ arrive from CASSM. That is why CASSM2-B does better than CASSM3. For large results CASSM2-W demonstrates the advantage of duplicate removal before results of the join are produced.

**DBC**

The database computer's response time is dominated by the numerous subqueries made to the secondary store. The need for one subquery per tuple of $R_2$ in the host causes DBC to respond in nearly twenty minutes. This is due to the lack of join facilities in the DBC.

**RAP**

$N_2 F_2 D_2$ subqueries must be made to complete the join. A subquery searches the CCD's once for every tuple of $R_2$ in the host. The CCD's reduce the total response time as compared to DBC. Since the cylinder access time is included only for the transfer of secondary store to the CCD's, the subqueries consume less time.

**DIRECT**

DIRECT is dominated by the output of result tuples in the worst case, DIRECT-W. In the best case DIRECT-B response is comprised mostly of the transfer of tuples from secondary store. With $F = .01$, the comparison of arriving tuples can be done faster than either the transfer of $R_1$ tuples from secondary store or the output of result
tuples.

An alternative is to do only the implicit join in DIRECT. The time for completing the join in the host increases with P. The line DIRECT2 in Figure 6.2 represents such a compromise. This implementation works well for small P.

REPT

In the best case response for REPT is near the ideal time. For the worst case the output of results causes an increase in response time for increasing P. The comparison time is less then the time to transfer tuples of $R_1$ into the tree. For REPT-W the time to output result tuples to the host is longer then the time to read tuples from $R_1$ or make the comparisons in the leaf node.

HYPERTREE

The best case for HYPERTREE, H-TREE1-B, response time increases for increasing P. This is due to the increased number of tuple transmissions within the HYPERTREE. H-TREE-B remains below ten seconds so it is not shown in Figure 6.2. The worst case, H-TREE-W, is dominated by result output. The advantage of reading the data directly is lost by the internal transmission of tuples to arbitrary nodes and the linear output of a large amount of results.

In this case by using the host to complete the join the response time of HYPERTREE is reduced. This is depicted by H-TREE3.

The alternative join implementation in H-TREE2 responds faster then H-TREE1. Both the worst and best cases of H-TREE2 respond faster.
H-TREE2-B is less than ten seconds so it is not shown in Figure 6.2. The faster response is due to the fewer reads of the two relations on the disk.

Conclusions

It may appear that burdening the host with duplicate removal and completing the join can be a reasonable compromise. This is shown in Figure 6.2 for CASSM3, DIRECT2 and H-TREE3. However, this is true only when the implicit join substantially reduces the relations. Duplicate removal for a single processor will require the time of $N \log_2(N) t_c$ for an unordered set of cardinality $N$. Comparing subsets of cardinality $N$ for the join, will require a time of $N^2 t_c$. Because of this, the gap between these compromise algorithms and their corresponding best cases grows greater as the selected, implicitly joined subsets increase.

A better solution to the output problem is to increase the bandwidth of result tuple transfer to the host. This can be accomplished for REPT and HYPERTREE by decreasing $t$. The tuple transfer time is $t = (t_{cy} / BC + t_b) / QE$ for these examples. This means that the result tuples can leave the system as fast as tuples of $R_1$ enter from secondary store. To remove the effect on response time of large sets of result tuples in REPT, change $t = (t_{cy} / BC + t_b) N_1 / (NQE)$. This would enable REPT to output to the front-end result tuples as fast as they could be produced. With this same $t$ value HYPERTREE could output tuples as fast as $R_1$ would be read, transferred to arbitrary nodes and compared for the join. In example 1, $t = .000107$ seconds would eliminate the output response problem. To decrease $t$ requires an increase in
bandwidth through the X-NODE links. It may also require an increase in bandwidth within the X-NODE.

It should be noted that an increase in output rate for DIRECT requires the same increase in input rate. This is because of the same storage medium for R₁ and result tuples. The CCD's can be written or read at a set rate. Thus, in this model, the change in t does not effect DIRECT.

Example 2: Larger Selected Sets

Next we assess the effect on response time of increasing F to .5. This is depicted in Figure 6.3. The values of N₁ and k are the same as Figure 6.2. The differences between example 1 and example 2 is due completely to the increase of F from .01 to .5.

The increase in F represents an increase in the cardinality of the selected subsets of R₁ and R₂. This decreases the time between locating qualifying tuples on secondary store. Increased F causes an increase in comparisons per tuples of R₁ to complete the join. It also increases the time for duplicate removal.

On the average, more results will be produced for example 2 then in example 1. For machines producing joined tuples the time to output these results increases. This influences the response in the worst case response times.

CAFS

CAFS is not constant for increasing P. The response time grows because of the time to complete the join in the host. This host join
time is no longer trivial for $F = .5$. It accounts for the difference of the CAPS line between Figure 6.2 and Figure 6.3. In example 2 the join cannot be completed in the time it takes to read $R_j$ from secondary store.

CASSM

CASSM2-W has now become the longest response time for any CASSM implementation. This reflects the duplicate removal in the host on the entire result set. The subqueries of CASSM1 finish before the comparisons of CASSM3.

As $P$ increases only CASSM3 and CASSM2-B remain under a 10,000 second response time. CASSM2-B does duplicate removal on the best case result set. CASSM3 removes duplicates from both source subsets before joining them.

DBC & RAP

Both DBC and RAP behave as in example 1. However, the increase in $F$ causes an increase in response times. More subqueries must be performed. DBC and RAP move from below one hour to more then three hours as $F$ was increased from .01 to .5.

DIRECT

DIRECT1-W is dominated by the comparisons of tuples for the join when $P < .3$. For $P > .3$, DIRECT1-W is primarily outputing result tuples.

DIRECT1-B is dominated by the comparison phase for all values of $P$. This makes this plot nearly one minute for $F = .01$ and nearly ten minutes for $F = .5$. 
DIRECT2 has a larger comparison phase. It lies between the other two lines because of the host completing the join.

**REPT**

The REPT response time is dominated by the join comparison when result production is low. This is due to the increased subset of $R_2$ stored at the leaves. Thus, REPT-B is greater then the same algorithm for $F = .01$. When the number of result tuples is large then the output phase dominates the response time. This is exhibited by the REPT-W line.

**HYPERTREE**

H-TREE1 increases response time in the best case from ten seconds when $F = .01$ and $P = 1$ (in example 1) to 90 seconds when $F = .5$ and $P = 1$ (in example 2). This is due to the increased number of comparisons and the larger number of results to output. The same is true for H-TREE-W where the output of a large result set dominates the response time.

H-TREE2-B responds slower than H-TREE1-B. This is due to the increased number of comparisons required at each leaf node in H-TREE2.

**Conclusions**

The increase of $N_2F$ from the 300 of example 1 to the 15,000 tuples of example 2 increases each machine's response time. The worst cases for DIRECT, REPT, and HYPERTREE show an increased sensitivity to the change in $P$. The difference between the worst and best cases for these machines is much greater then for $F = .01$. DIRECT2 and H-TREE3 increase the use of the host. They lie between the worst and
best cases for those machines. However, the difference between these compromise algorithms and the best case has increased. For example, with $F = .01$ and $P = 1$, $\text{DIRECT2} - \text{DIRECT1-B} = 38$ seconds. For $F = .05$ and $P = 1$, the difference is $3005$ seconds. The use of the host to complete the join is becoming infeasible. The better alternative is to reduce $\text{DIRECT1-W}$ by decreasing the tuple transfer time. This will mean increasing the cost of CCD's for faster access.

Comparing Figure 6.2 and Figure 6.3 shows that in the latter response time there is a greater difference from the ideal time. This is due to the increased comparison times of larger subsets of the source relations. The best case times of REPT and DIRECT1 are an order of magnitude greater than the ideal line. In this instance an increase in leaf nodes would bring REPT-B down toward the ideal time.

For REPT, an increase in $k$ would reduce the $R_2$ subsets in every leaf. An increase in query processors for DIRECT would also decrease the response time for DIRECT1-B, but not as effectively as for REPT-B. This is due to the complete copies of the selected $R_2$ subsets stored in each query processor.

For example 2 the response times are beginning to show the influence of the join comparison time. For CAFS, CASSM, RAP, and DBC, the completion of the join in the host is becoming infeasible for $F = .5$.

For DIRECT, REPT and HYPERTREE the decrease in tuple transfer time can decrease response. For example 2, $t < .000001$ with $P = 1$ would minimize the effect of result tuple output for REPT-W and
H-TREE1-W.

HYPERTREE shows the best possibility of feasible response time if technology can permit high speed transfer of results to the front-end. This is due to the direct search of data, the filtering of unused tuples at the secondary store interface and the distribution of comparisons throughout the tree.

Increasing Database Size

An IBM 3330 series 11 disk pack connected to a HYPERTREE with 16 leaf nodes cannot store a database larger than $N_2 = 30,000$ and $N_1 = 120,000$ tuples without expanding the number of leaf nodes. The machines CAFS, DIRECT, REPT and RAP are not restricted to this size.

The graphs in Figure 6.4, Figure 6.5 and Figure 6.6 demonstrate the increase in response for fixed values of $F$ and $P$ as $N_2$ (and $N_1$) increases.

For Figure 6.4 $F = .01$, $P = .01$ and $k = 16$. As $N_2$ increases, the increase in response time is due to the number of cylinders which are occupied by $N_2$ and $N_1$. The machines DIRECT, REPT, and CAFS are near the ideal response line. Their responses are not increased by join comparisons nor by result tuple output. This is due to the decrease in cardinality of source relations by the selection and projections.

Figure 6.5 and Figure 6.6 demonstrate that $F = .5$ and $P = .01$ have a larger affect on response then $P = .5$ and $F = .01$. In the case of REPT and DIRECT, the larger sets left after selection cause the join comparison phase to cominate the response time. A smaller selection
Figure 6.4 Response times for $F = .01$, $P = .01$ and $30,000 \leq N_2 \leq 100,000$
Figure 6.5. Response times for $F = .5$, $P = .01$ and $30,000 \leq N_2 \leq 100,000$
Figure 6.6. Response times for $F = .01$, $P = .5$ and $30,000 \leq N_2 \leq 100,000$
qualification causes input or output to dominate the response times. Thus, Figure 6.5 shows a greater response time for the identical machines in Figure 6.6. The holding of $F = .01$ and $P = .5$ (Figure 6.6) does not cause much change in responses over $F = .01$ and $P = .01$ (Figure 6.4). This demonstrates the desirability of performing selections which eliminate large portions of the source relations before the join comparisons are done.

With $F = .5$ increasing $P$ can cause the DIRECTl-W and REPT-W implementations to rise considerable. This is due to the output of results. This effect can be seen in comparing Figure 6.5 and Figure 6.6. For large $F$ a change in $P$ can make a significant difference in the response of the worst case and in the implementations such as DIRECT2 which complete the join in the front-end.

For low $F$, the indirect search machines can maintain low increases in response time with increasing database size. This is accomplished for large $F$ only if the comparison times are decreased. It is done for large $P$ if the output rate is increased.

**Effect of Increasing k**

Figure 6.7 depicts the changes in response time caused by increasing the number of cells in multiple processor machines from 16 to 64. This is done with $F = .5$ and $N_2 F_2 P_2 = 15,000$ tuples.

For the worst cases, very little change occurs. This is due to the output of result tuples. Increasing processors in the back-end does not decrease the time to output a result tuple to the host.
Figure 6.7. Response times for $F = .5$, $P = 1.0$ and $16 \leq k \leq 64$
In the best case for DIRECT, H-TREE, and REPT a reduction occurs due to the large amount of comparison time spent with \( N_1^P_1^F_1 = 15,000 \). DIRECT uses the same subset of \( R_2 \) in all query processors. This technique does not decrease comparison time as rapidly as for REPT when \( k \) is increased. REPT stores a different subset of \( R_2 \) in each leaf node. Thus, an increase in \( k \) decreases the comparison time for each arriving tuple of \( N_1 \).

HYPERTREE is not constrained by the rate at which the tuples can be read from the disk. Increases in leaf nodes do not help as much as they do in REPT. HYPERTREE has an internal communication phase. In its last step it may have many tuples crossing the same links. If no link congestion occurs, then the response time for HYPERTREE will be less then depicted in this and all the other examples. Simulation is needed to determine the time to simultaneously send tuples from leaf nodes to all other nodes in HYPERTREE.

Direct Search Advantage

In Figure 6.8 and Figure 6.9 the disk transfer parameters \( t_{cy} \) and \( t_b \) have been increased from their values used in the previous example. \( t_{cy} \) and \( t_b \) are \( k \) times larger. Thus, this graph depicts the response time for HYPERTREE as if the bandwidth for data transfer from secondary store into HYPERTREE were equivalent to an indirect search machine with one multiplexed channel rather then \( k \) parallel read heads.

Besides the above adjustment in disk transfer rate, Figure 6.8 is an identical situation to Figure 6.2. The worst and best cases for
Figure 6.8. Response times for $F = .01$ with $t_{cy1}$ and $t_b$ increased $k$ times.
Figure 6.9. Response times for $F = .5$ with $t_{cyl}$ and $t_b$ increased $k$ times.
HYPERTREE in Figure 6.8 are nearly the same as the REPT-W and REPT-B lines in Figure 6.2. For \( F = .01 \) these machines have nearly the same response time if HYPERTREE loses its advantage of directly searching the data with \( k \) parallel read heads.

The same comparison between HYPERTREE and REPT can be made with Figure 6.3 and Figure 6.9. In this case HYPERTREE is at an advantage for small \( P \) in the best case (H-TREEl-B). This is because the REPT-B response time is dominated by the comparison time in the leaf nodes. H-TREEl filters out much of the extraneous comparisons by the use of bit maps at the leaves. For small \( P \), H-TREEl does less comparisons. This cannot be done in REPT without reading both source relations twice. That is a costly process for an indirect search machine.
CHAPTER VII. CONCLUSIONS

To demonstrate the effect of our computer architecture design principles, we have designed REPT. These principles, which were followed, are 1) the use of identical processing nodes to construct the REPT architecture, 2) restricting node processes to require no global knowledge, and 3) the ability to expand REPT without modification of the node processes.

A discussion of current database machine architectures was presented. This served as a basis for the development of a response time analysis. In this analysis, CAPS, CASSM, DBC, RAP, DIRECT, REPT, and HYPERTREE were compared.

The conclusions in this chapter are based upon our design of the REPT architecture. This included the logical structure, high-level language implementation, and the description of the node processes.

The example tree machine is comprised of multiple, homogenous processing nodes. These nodes communicate in a static binary tree structure. The individual algorithms executed by each node require no global knowledge for the entire structure to execute query operations. There exists two classes of nodes, the leaf node processors and the intermediate node processors. Except for certain class functions, all nodes execute identical software. These characteristics allow expansion of the tree without changes to the existing node processes. Such expansion capability is valuable to meet needed increases in secondary store transfer rates. If faster storage technology is employed and the link
communication rate is not surpassed, then the addition of more processing nodes can increase the throughput of queries.

The binary tree communication topology facilitates the distribution of a source relation evenly across leaf node memories. The tree structure easily communicates relations and queries to every node of the system. The flow of relations is primarily upward or downward. Although the root is a natural constriction point for traffic flow, this can be countered by application of high bandwidth communication technology. Overall, the tree structure supports a high degree of parallel processing across the leaf nodes and pipelining of source and result relations.

Transmission of source relations, the comparison of tuples, and the transmission of result relations are concurrent activities. For example, the tree-merge duplicate removal operation can proceed in parallel with a join operation. The duplicate free tuples arriving at the root can be transmitted down the tree to participate in a join operation. This is one example of the concurrency among selection, projection, and join which the tree supports.

The hardware design of the X-NODE can support the capabilities of the tree structured machine. Each node of REPT can be implemented upon an X-NODE. This supplies homogeneity to the REPT processors. It also supplies REPT with a concurrent relation transfer and searching capability.

Queries are the sole basis for routing decisions. The assignment of output ports to an arriving relation is done from a router look-up
The entries for the look-up table are created by the node processor.

The processor within a node has two tasks. The first is to translate queries into internal node functions. The second is to execute those functions when the source relations arrive in its memory.

The operations performed by a node to execute a query fall into two classes. The intermediate nodes execute identical functions. The leaf nodes each act identically to the same query. Outside of the entries in the router table and the functions performed by the node processor during query execution, the algorithms performed by all nodes of the system are identical.

This process homogeneity and the requirement of no global knowledge by the nodes, contributes to the expandability of REPT. For example, an additional tree may be linked beside the existing one by attaching it to the root, or creating a new root, of the tree. This effectively doubles the processing power of the machine. However, no changes in the software controlling the nodes need be made. This reflects that there exists no central controller node in the tree structure unlike CAFS, RAP, DIRECT, CASSM and DBC.

REPT is in the MPIS class along with RAP and DIRECT. Both RAP and DIRECT have been shown to execute queries with an order of magnitude improvement over conventional database management systems (Ozkarahan et al. 1977 and Hawthorn and DeWitt 1980). REPT has been shown to be competitive to both DIRECT and RAP in executing multiple relation queries. In addition, the cost of expanding DIRECT is proportional to the
square of the number of query processors. This is due to the cost of the interconnection matrix. This is no worse than the cost of \( k - 1 \) additional intermediate nodes to add \( k \) leaf nodes to REPT.

The direct search capabilities of HYPERTREE give it an inherent response time advantage. Given \( k \) leaf nodes, it has an effective bandwidth of \( k \) times that of a \( k \)-cell MPIS machine. The expansion of a HYPERTREE requires additional read/write logic and the expansion of a leaf (and the accompanying intermediate nodes) per platter of the moving head disks. HYPERTREE could be used as an intermediate storage or pseudo-cache system where the database is loaded from a secondary store into the leaf node disks. This would decrease the cost of database expansion and allow parallel search of the data in the leaf node disks.

Expansions of REPT can be accomplished with little difficulty due to the lack of requirement for global knowledge. This is not the case for HYPERTREE. The leaf nodes must know the size and topology of the entire HYPERTREE structure in order to distribute the implicitly joined tuples evenly across its nodes.

REPT is limited by the rate at which tuples may be transferred into and out of the machine. Except for the re-access of tuples stored in the leaf nodes, REPT can process tuples only as fast as they can be transferred from secondary store. When at least one source relation of a join has a small cardinality, REPT will respond depending on either the rate the tuples can be read from secondary store, or the rate at which result tuples can be transferred up the tree and out of the
root.

The application of high bandwidth communication links is needed to lower response time. In the worst case, the transfer rate of tuples over links must be greater than the transfer rate of tuples from secondary store. This is due to the potentially large result set.

The effect of the design principles for the REPT architecture are encouraging. Parallel processing is applied to the relational and set operations. Data pipelining is useful for concurrent operations. The X-NODES offer technology suitable for implementation of homogeneous nodes. Except for basic node processor functions the node software is identical between nodes. The X-NODES also allow the concurrent transfer and searching of relations. REPT responds to multiple relation queries competitively. Along with these desirable traits, REPT is extensible. The node processes require no global knowledge and may be easily incorporated to enlarge the REPT structure.

Based on this example, we conclude that the design principles of REPT are worthy of future computer architectures. Following such design principles produces machines which make use of VLSI technology. Machines to support particular applications can be produced from multiple, identical nodes. Such machines will be low in production cost and easily expanded.

Future Work

Certain areas of database support have been left for future work. Hardware support for multiple user updates and security are to be
investigated. A tree structured front-end machine may be suited to support security and scheduling of user requests.

The leaf nodes of a front-end machine could be the interfaces to each user. The root node may correspond to the root of the back-end machine. The front-end and back-end may then be physically combined into one tree. The leaves serving as both data processing nodes and user interfaces. The intermediate nodes could perform the back-end tasks along with scheduling and security duties.

Simulation of REPT can be used to produce results to support the response time analysis. Such a simulation project if general enough can also be used to investigate other multiple node machine architectures such as the suggested front-end machine.

We wish to connect hardware prototypes of a few nodes to an existing computer. Such experiments can demonstrate timing and switching capabilities of REPT.

Other communication connections of nodes may be well-suited for database functions and fault tolerance. Different geometric communication topologies need to be analyzed and simulated. Possibilities exist for structures which are statically connected but logically reconfigurable.

Different applications will require other machine structures. It will be beneficial to investigate applications in numerical processing, graphics and semantic networks. Using the design criterion supported by this thesis, we hope to generalize our results to other application
areas. This will produce a set of specialized multiprocessor machine architectures.
REFERENCES


ACKNOWLEDGEMENTS

I thank professor Roy Zingg for the many hours spent in directing my research and his insights into the design of computer architectures.

Without the patience and love of my wife, Rosemary, this research would be uninspired. I thank you Rosemary.
APPENDIX. PARAMETER DEFINITIONS

BC  the number of blocks per secondary store cylinder

BL1  the number of secondary store blocks occupied by R1
       (BL1 = \lceil M_1 / Q \rceil)

BL2  the number of secondary store blocks occupied by R2
       (BL2 = \lceil M_2 / Q \rceil)

BP  the number of bits per memory page

C  the number of pagers per CCD memory

C_x  the number of CCDS to contain R_x
       (C_x = \lceil M_x / C \rceil)

d  the depth of a binary tree with k leaf nodes
      (root at level 1)

dave  the average path length from a HYPERTREE leaf node to an
       arbitrary node

D_x  the ratio of a projected relation cardinality to the original
       cardinality

DT  the average secondary store block transfer time
       (DT = t_{cyl} / BC + tb)

DTK  the average secondary store block transfer time
       with k parallel read heads
       (DTK = t_{cyl} / BC + tb / k)

E  the average number of tuples per memory page

F_x  the ratio of the selected subset cardinality to the original
       relation cardinality

H_x  the ratio of wanted bytes of a tuple to total bytes of a tuple
       (result of a projection)

k  the number of processing cells in a multiple processing
       architecture

L(x)  the average number of leaf nodes storing a x-valued tuple

M_x  the number of memory pages needed to hold relation R_x
       (M_x = \lceil N_x / E \rceil)

N_x  the cardinality of R_x
$P_x$  
the ratio of the number of tuples of $R_x$ participating in a join to the cardinality of $R_x$

$Q$  
the number of memory pages per block of secondary store

$R_x$  
a relation name

$R(k,j)$  
the number of rotations to output tuples from $k$ RAP cells where $j\%$ of the tuples in the same relative slot need to be outputed

$\text{SORT}(j)$  
the average time to sort $j$ tuples ($\text{SORT}(j) = j \log_2(j) t_c$)

$t$  
the average tuple transfer time within a machine

$t_b$  
the time to transfer a block from secondary store

$t_c$  
the average time to compare two tuples

$t_{cyl}$  
the average secondary store cylinder access time

$t_m$  
the CCD rotation time

$T_x$  
the number of unique tuples in $R_x$

$U_x$  
the number of unique value combinations possible from the domains of a join field of $R_x$

$y$  
the number of bits per hash table in HYPERTREE

$\pi$  
the ratio of the number of tuples including hash collision "phantoms" to the cardinality of the "phantom-less" set.