INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600
Optimizing select-project-join queries

Miller, Rebecca Anne, M.S.
University of Nevada, Las Vegas, 1990
NOTE TO USERS

THE ORIGINAL DOCUMENT RECEIVED BY U.M.I. CONTAINED PAGES WITH BLACK MARKS AND POOR PRINT. PAGES WERE FILMED AS RECEIVED.

THIS REPRODUCTION IS THE BEST AVAILABLE COPY.
OPTIMIZING SELECT-
PROJECT-JOIN
QUERIES

by

Rebecca Anne Miller

A thesis submitted in partial fulfillment
of the requirements for the degree of

Masters
in
Computer Science

Department of Computer Science
University of Nevada, Las Vegas
December 1990
The thesis of Rebecca Anne Miller for the degree of Masters in Computer Science is approved.

Kazam Taghva, Ph.D.
Chairman

Yonina Cooper, Ph.D.
Examinining Committee

Thomas Nartker, Ph.D.
Examinining Committee

Ebrahim Salehi, Ph.D.
Graduate Faculty Representative

Ebrahim Salehi, Ph.D.
Graduate Dean

University of Nevada, Las Vegas
December 1990
Optimizing Select-Project-Join Queries

Abstract

An optimal retrieval algorithm for relational database queries is a highly sought after ideal. A great amount of research has been applied in the quest for a better, faster, and cheaper universal retrieval scheme. Unfortunately there is no panacea, no single "optimal" method. With the many different database languages we find a variety of retrieval algorithms. This thesis explores heuristic methods for optimizing relational database query algorithms. It combines individual works on heuristics, cost modeling, decomposition, and join optimization. Lastly it assesses these optimization methods.
# Table of Contents

## CHAPTER 1
Introduction to Relational Database Management Systems......1  
Data Base Languages.........................................2  
Relational Algebra..........................................5

## CHAPTER 2
Optimization..................................................14  
Hashing........................................................15  
ISAM............................................................19  
B-Trees.........................................................22  
Select-Project-Join..........................................25  
The Cost Model..............................................33  
Costing.......................................................61

## CHAPTER 3
Heuristics....................................................71  
Decomposition...............................................80  
Ordering Joins..............................................88

## CHAPTER 4
Conclusion..................................................94
List of Figures

Figure 1 Examples of Relational Algebra Operators .......... 7
Figure 2 Sample Bunny Database ................................ 9-10
Figure 3 Example of Records in a Hash File .................. 17
Figure 4 Example of a Balanced Tree .......................... 23
Figure 5 Example of a Selection ................................ 26
Figure 6 Example of a Projection ................................ 27
Figure 7 Example of a Join ..................................... 28
Figure 8 One-Variable Query Evaluation Example ............ 40
Figure 9 Simple Two-Variable Query Evaluation Example .... 42
Figure 10 Sort Relations ........................................ 43
Figure 11 TID ................................................ 44
Figure 12 Algorithm Class 1 (RAJ) ............................. 47
Figure 13 Algorithm Class 2 (JAR) ............................. 48
Figure 14 Algorithm Class 3 (JRA and RJA) .................. 49
Figure 15 Algorithm Class 4 (ARJ) ............................. 51
Figure 16 Algorithm Class 5 (AJR) ............................. 53
Figure 17 Algorithm Class 6 (LAR) ............................. 54
Figure 18 Algorithm Class 7 (LRA and RLA) ................. 57
Figure 19 Storage Parameters ................................... 58
Figure 20 SEQUEL Classification ................................. 72
Figure 21 Parse Trees for Table 3 .........................74-75
Figure 22 The QUEL Statement..........................82
Figure 23 The QUEL Representation of Table 3............82
Figure 24 Multi-Variable Decomposition....................84
Figure 25 Decomposition - Example 1......................86
Figure 26 Decomposition - Example 2.......................87
Figure 27 Outer Linear Join Tree..........................90

List of Tables

Table 1 Relational Operators..............................6
Table 2 Rancho Bunieta Database Relations...............11
Table 3 Example of Relational Algebra Query.............12
Table 4 Symbols Used in Algorithm Examples..............37-39
Table 5 Storage Parameter Requirements..................59
Table 6 Cost Model Symbols...............................60
Table 7 Cost Equations for Access Operations............62-63
Table 8 Equivalences for Selections........................77
Table 9 Equivalences for Projections.......................77
Acknowledgments:

The author wishes to acknowledge Kazam Taghva for his instruction, ideas, words of wisdom, and motivation. She also thanks David Hoff for being willing to learn more than he ever wanted to know about database optimization and for sitting on the beach and arguing; Robert Woodruff for the original idea and a great deal of encouragement; and Gerald Engel, mentor, associate, and friend. A special heartfelt thank-you goes to Lisa Miller and the Rancho Bunieta, I guess we are somebody.
CHAPTER 1
Introduction to Relational Database Management Systems

A database stores information about data. The software used to store and manipulate this information is referred to as a Data Base Management System (DBMS). [Ullml] states "A DBMS is one of the most complex varieties of software in existence". In his reference he considers the variety of users and possibilities to interact with the system. Today's systems certainly have a wide variety of users and database software.

Efficient DBMSs must minimize the consumption of human and machine resources. Resources include storage space, memory buffers, CPU time, and transfer time between main memory and secondary storage or data access time. Optimization of the database system should provide the best utilization of resources. There are three areas where the DBMS can support optimization:
1) Physical Design
   How the data records are stored (size of records, types of indices, etc.)
2) Transaction Management
   Entering, deleting, modifying the data
3) Query Processing Subsystem
   Evaluating queries posed by the user [JKS]

Data Base Languages

Modern database systems do not utilize the typical programming language. A typical programming language has both the declarative and executable statements tied into one. In a database system these two features are separated into the declaration and computation levels. The reasoning behind the incorporation of two different languages into one is that the program variables in an ordinary language are active only when the program is active and need be redeclared each time the program is run, while in a database the data variables are "forever" and need to be declared only once. The separate declaration language is a convenience [Ullml].
The declaration language is used to define entities and to note relationships. It is not used to lookup, modify, or delete data. Because it must detail the design of the system, the data declaration language normally contains statements that describe the physical layout of the data. The manipulation of the database requires a specialized data execution language or query language. This language performs the lookup, modify and delete operations in the database.

Query languages for relational databases are broken down into two classes:

1) Algebraic languages: these languages express queries through the use of special algebra operations on the relations. They are procedural and interested in "how" things are done
2) Calculus languages: these languages describe the desired output by specifying a predicate which the relation's tuples must satisfy. Calculus based languages are declarative and concerned the output, not "how" things are done.

Relational Algebra falls into the first class of languages. Relational Calculus and Tuple Calculus belong to the second. Although some models of optimization give their examples using one of the calculus based languages, we will use Relational Algebra. Many of the fundamental optimization schemes deal with the algebraic manipulation of the query. Each of the three methods is equivalent in expressive power to the others, therefore, relational algebra may be easily translated for many systems. The description and examples of relational algebra presented in this paper are abstracts and are not actually implemented in a commercial system. However, most query systems available provide at least the capabilities of one of the three languages noted above plus some extras of their own.
Relational Algebra

A relation is a set of tuples. It can be represented as a two-dimensional table. A tuple is a row in the table. The components of the relation are the columns in the table. A relation is denoted by a descriptive name such as "employee" or if we are discussing some generalized theory then it is given a generic name, commonly a capital letter such as "R". We refer to the components by names such as "date-of-birth" or "address". Sometimes it's convenient to let the components stay anonymous and reference them by column number or by a lower case letter. R.a would represent the "a" component of relation "R". Arity, or degree, is the number of components in a tuple. We assume relations are finite.

The operators in relational algebra are listed in Table 1. Examples are shown in Figure 1. The operands are either constant relations or variables denoting relations. A relational algebra operation takes one or more operands and forms a new relation:
Table 1
Relational Operators

The Primary Operators:

Union - The union of two relations, denoted as R \cup S, is the set of all tuples in R or S or both. Union is applied to only those tuples with the same arity so that the resulting relation has the same number of components.

Difference - The difference between two relations, denoted as R - S, is the set of tuples in R but not in S. As in union, the tuples must have the same arity.

Cartesian Product - The cartesian product of two relations, denoted as R \times S, where R is arity i and S is arity j, is a relation with (i+j) tuples. The i..i components come from R and the i+1..i+j components come from S.

Projection - The projection of a component from a relation, denoted as \pi_y(R), where y is a component in R, is a means of restricting the arity of a relation.

Selection - The selection of a value or component from a relation, denoted as \sigma_y(R), where y is a component in R, is a means of restricting the number of tuples in a relation.

Some Additional Operators:

Intersection - The intersection of two tuples, denoted as R \cap S, is shorthand for R-(R-S).

Quotient - Let the arity of R be r and the arity of S be s where r > s and S is not empty, then the quotient, denoted as R\backslash S, is the set of (r-s) tuple t such that for all s tuples = u in S, the tuple tu is in R. An example may be found in [Ullml].

Join - Join is a combination of select and product. Join is commutative so the relations can be joined in any order but the ordering DOES make a difference in complexity.
Examples of Relational Algebra Operators

**Figure 1**
\[ C = R.a \theta B.d \]

where \( C, R \), and \( B \) are relations

\( a \) and \( d \) are components

and \( \theta \) is an operator

We present a small sample database in Figure 2 and Table 2. Table 3 is a relational algebra example using the database. An explanation of the database follows.

Rancho Bunieta, a small rabbitry in Sacramento, maintains records on individual rabbits, shows, and all attempts at breeding. Our database example will limit the database to rabbits and their breeding. As we will be using the Rancho Bunieta data for our examples later some definitions now will be helpful to the non-bunny initiated. A doe is a mother, bucks are fathers, kindle means birth, dob is date of birth, nest_box is the date that the nest was placed with the doe, tested is a method of checking for pregnancy, #added means number of babies added from one doe to another, 'taken' is the number taken from one doe, 'saved' is the number that survived infancy, breed is the type of bunny such as 'mini-lop', quality is either 'pet' or 'show' (or 'meat' if we were so inclined), service is an attempt
<table>
<thead>
<tr>
<th>ear#</th>
<th>name</th>
<th>sex</th>
<th>breed</th>
<th>color</th>
<th>reg#</th>
<th>litter#</th>
<th>quality</th>
<th>b_price</th>
<th>s_price</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>AJ</td>
<td>M</td>
<td>lop</td>
<td>tan</td>
<td>54391</td>
<td>2</td>
<td>S</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>129</td>
<td>Ricky</td>
<td>M</td>
<td>lop</td>
<td>sable</td>
<td>12738</td>
<td>m</td>
<td>S</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>241</td>
<td>Macy</td>
<td>F</td>
<td>lop</td>
<td>black</td>
<td>72915A</td>
<td>13</td>
<td>S</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>242</td>
<td>Cloe</td>
<td>F</td>
<td>lop</td>
<td>black</td>
<td>72916A</td>
<td>13</td>
<td>S</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>243</td>
<td>Halston</td>
<td>F</td>
<td>lop</td>
<td>black</td>
<td>m</td>
<td>7</td>
<td>P</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>512</td>
<td>Ivy</td>
<td>F</td>
<td>dwarf</td>
<td>w/b</td>
<td>m</td>
<td>9</td>
<td>B</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>513</td>
<td>Fern</td>
<td>F</td>
<td>dwarf</td>
<td>w/b</td>
<td>m</td>
<td>9</td>
<td>B</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>514</td>
<td>Marigold</td>
<td>F</td>
<td>dwarf</td>
<td>tan</td>
<td>827910</td>
<td>20</td>
<td>S</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>516</td>
<td>Ti</td>
<td>F</td>
<td>dwarf</td>
<td>tan</td>
<td>m</td>
<td>m</td>
<td>S</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>530</td>
<td>Alfalfa</td>
<td>F</td>
<td>dwarf</td>
<td>w/b</td>
<td>621900</td>
<td>m</td>
<td>S</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>540</td>
<td>Juniper</td>
<td>M</td>
<td>dwarf</td>
<td>w/b</td>
<td>621794</td>
<td>m</td>
<td>S</td>
<td>20</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>service</th>
<th>service#</th>
<th>buck#</th>
<th>doe#</th>
<th>date_bred</th>
</tr>
</thead>
<tbody>
<tr>
<td>74</td>
<td>129</td>
<td>242</td>
<td>051589</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>129</td>
<td>242</td>
<td>053089</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>129</td>
<td>241</td>
<td>021290</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>540</td>
<td>514</td>
<td>031090</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>129</td>
<td>242</td>
<td>033090</td>
<td></td>
</tr>
<tr>
<td>85</td>
<td>540</td>
<td>530</td>
<td>040990</td>
<td></td>
</tr>
<tr>
<td>86</td>
<td>540</td>
<td>514</td>
<td>042990</td>
<td></td>
</tr>
<tr>
<td>87</td>
<td>129</td>
<td>241</td>
<td>050190</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>breeding</th>
<th>service#</th>
<th>date_due</th>
<th>tested</th>
<th>nestbox</th>
<th>kindled</th>
</tr>
</thead>
<tbody>
<tr>
<td>74</td>
<td>061589</td>
<td>false</td>
<td>-</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>063089</td>
<td>true</td>
<td>062789</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>031590</td>
<td>-</td>
<td>031290</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>041090</td>
<td>-</td>
<td>040790</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>043090</td>
<td>true</td>
<td>042790</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>85</td>
<td>051090</td>
<td>false</td>
<td>-</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>86</td>
<td>053090</td>
<td>true</td>
<td>052790</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>87</td>
<td>060190</td>
<td>true</td>
<td>053090</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Sample Bunny Database

Figure 2 (page 1 of 2)
Note: in the following relation some of the attribute names were shortened so that the record would fit on one line. These attributes are:

- bb = bucks_born
- db = does_born
- ba = bucks_added
- da = does_added
- bl = bucks_left
- dl = does_left
- bt = bucks_taken
- dt = does_taken

<table>
<thead>
<tr>
<th>litter#</th>
<th>service#</th>
<th>dob</th>
<th>weaned</th>
<th>bb</th>
<th>db</th>
<th>ba</th>
<th>da</th>
<th>bl</th>
<th>dl</th>
<th>bt</th>
<th>dt</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>75</td>
<td>063089</td>
<td>081489</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>44</td>
<td>76</td>
<td>031690</td>
<td>042090</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>45</td>
<td>77</td>
<td>041090</td>
<td>052590</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>47</td>
<td>86</td>
<td>053090</td>
<td>071590</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>48</td>
<td>87</td>
<td>060190</td>
<td>071590</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Sample Bunny Database (continued)

Figure 2 (page 2 of 2)
Table 2

Rancho Bunieta Database Relations

**bunny** (*ear#, name, sex, color, reg#, breed, litter#, quality, b_price, s_price)

**breeding** (*service#, date_due, nest_box, tested, kindled)

**litter** (*litter#, service#, dob, wean, bucks_born, does_born, bucks_added, does_added, bucks_left, does_left, bucks_taken, does_taken)

**service** (*service#, buck#, doe#, date_bred)

*item means that the variable "item" is the key for the relation.
Table 3

Example of Relational Algebra Query

English query:

Who are the "successful" mini-lop fathers?

Relational Algebra Equivalent:

\[
\text{bunny.name} \\
(\text{litter.bucks_saved} > 0 \land \text{litter.does_saved} > 0 \land \text{bunny.breed} = 'lop' \land \text{litter.service#} = \text{service.service#} \land \text{service.buck#} = \text{bunny.ear#})
\]
at breeding a buck to a doe, \( b_{\text{price}} \) is the dollars spent to buy to the bunny, \( s_{\text{price}} \) is the price fetched upon sale, \( \text{reg#} \) is an official registration number (not all bunnies are registered), and an \( \text{ear#} \) is a tattoo used for positive identification.

We've described the differences in query languages and provided a relational algebra example and a database that will be used throughout the paper. In the next chapters we discuss the complexity of block access, the basics of retrieval optimization, and [Yao]'s cost model. We also discuss heuristic methods of optimization. We introduce the work by Wong/Youssefi and Swami/Gupta on decomposing queries and ordering joins and show how this may improve on Yao's model. The final chapter is our conclusion followed by a section on other areas of interest.

We use the word optimization to describe the action of bettering the retrieval time in a query evaluation, but optimization is not necessarily a good word to use because there is no guarantee of an optimal solution. [Ullml] suggests the word "amelioration", others have
suggested "improvement", we will continue to use "optimization".

CHAPTER 2
Optimization

This chapter answers the question, "Why do we need to optimize?". Optimization involves the complexity of the query processing subsystem. Complexity is not actually finding the item in main memory but counting the number of data transfers from secondary storage. A data transfer is called a block access. This type of access transfers a block of data from secondary storage to main memory (or it writes a block from main memory back to secondary storage). The block access operation typically takes more time, and is therefore more complex, than searching an entire block of data in main memory. A good description of how to estimate block transfers when block storage is not uniform may be found in [Chris]. For complexity's sake, it is desirable not to make multiple block accesses. How a block is accessed depends on the physical storage method used. Principle database design involves the
storage of data. There are three popular storage schemes in use today. They are hashing, indexed sequential access method (ISAM), and balanced trees (b-trees). We will give brief discussions on each of these methods. Because look-up is a required transaction management operation, we will describe it along with two other important operations, insert and delete. We also outline the complexity involved with these operations. In the explanation of the storage schemes, \( O \) is the symbol used to represent "order of magnitude".

**Hashing**

Hashing is one of the most popular methods of storing data. It is generally fast to implement and use. The term "hash" comes from the seemingly mish-mashed method of determining the storage location for records. We will see where it fails to be "the best" after we describe the basics.

Records of a file (or relation) are divided among "buckets", where a bucket holds one or more physical
blocks of storage. There is a directory which consists of pointers to each bucket. If a bucket contains only one block, then the bucket's header has a null pointer. If there is more than one block then the pointer in the header points to the next block. Of course the last block in line has a null pointer. Each block has a fixed number of records that it may hold. The records in a block are referred to as sub-blocks and have deletion bits together with the data and a key.

Storing a record involves hashing the record's key in a hash function. This provides a bucket in which the record is placed. One method of hashing the key is to add the ASCII codes for the characters in the key and mod by the number of buckets available. Another method is to add the number of letters in the key and mod that by the number of buckets. There are some very elaborate hash functions in use, but these simple functions also work. It has been determined that a prime number of buckets works best. A pictorial example of inserting a record in a hash table may be seen in Figure 3. The insert function must actually start with a look-up to insure that the record doesn't
key = name
hash(key) = length of key mod 5
hash(halston) = 7 mod 5 = 2 = bucket 2

Example of Records in a Hash File

Figure 3
already exist. Then the new record may be added in the proper bucket.

Look-up in hashing involves finding the bucket that holds the desired record. The hash function used to store all of the data records is applied to the key of the record to be found. The associated bucket is then located by looking up the pointer in the directory. The first block in the bucket is searched for the key. If it holds the record then the look-up is done else the next block is found by tracing the pointer in the header. This searching of blocks is continued until the record is found or there are no more blocks in the bucket.

Delete involves finding the record (look-up) and setting the "delete" bits. No reassignment of the pointers is necessary because the record is not really deleted.

The best case hashing function is look-up (find and bring-in) which equals O2. Worst case is insert/delete (find, bring-in, and write) at O3
Hashing is a bad choice if queries are range queries; such as "all the rabbits with names beginning with A-C". This is because hashing is random and many, many buckets/blocks may need to be searched in order to make sure that one of the names is not missed. Something else to keep in mind when using hashing, every few months the data need to be rehashed to apply a garbage disposal routine for the deleted records and to make sure that the distribution of records is fairly even. But, despite this, hashing is generally a fast good method to store data.

ISAM

Indexed Sequential Access Method, or ISAM, is another extremely popular method for storing database records. The idea behind ISAM is similar to that of a phone book. People with the same last name share a common key and the user finds a phone number by first locating the page with the desired last name on it by looking at the header on the page and then by searching the entries on the associated page. In ISAM, a record is found by first searching the index for the key and then
by searching the associated block. Both the index and the blocks are sorted according to their keys.

The index for the ISAM records is actually just the first key and address of each block of data in the main file. The main file consists of blocks, each holding sorted records. The big picture is that the index stores only the address of each block and the key of the first record in each block.

To look-up a record, we find the key in the index that is just before or equal to our key. This requires finding two keys, one that is less than or equal and one that is greater than our key. We find the greater key so that we know we have the most immediate smaller key. The key immediately less than (or equal to) our key is called the cover. Now, we search the block that is associated with the cover. The search method may be any one of a number of popular searches; linear, binary, etc.

To insert a record, find the cover and search for the record. If it is not there then check for enough empty space in the block to hold the new record. If there is
no room then the next block in line needs to be checked. If there is no room there then a new block needs to be picked up and its address/key needs to be inserted in the index (sorted of course).

Deleting a record in an ISAM file involves looking up the record and setting the delete bits in the main file record. If the key record of the block is being deleted, then the index needs to be updated with the key of the next record in the block.

The order of magnitude for all three operations is the same. They all involve a search \((1 + \log_n)\) and bringing in one block (+1). Worst case involves inserting a record into a full block. This requires looking at the next block in line to check to see if there's room in it. If there isn't then another block for the main file must be brought in and we add two accesses (+2). Updating the index would require an additional write (+1). This makes for a total of \(5 + \log_n\) which is \(O\log_n\). This order is worse than hashing but it handles the range questions that kill hashing.
B-Trees

A Balanced Tree (b-tree) is a data structure where every path from the root to a leaf node is of the same length.

The lowest level of a b-tree consists of the sorted records in the database. In a tree structure these are referred to as leaves. The records are identified by their keys and packed into blocks. The index for a b-tree is comprised of a number of index blocks. In the index blocks, the smallest key in each path is stored with a pointer to the first block in the path. This may be another index block. We assume that the number of records in the main file per block is odd, and is \( 2e-1 \); where \( e \) is the number of pointers per index. We also assume that the number of index blocks is odd, and is \( 2d-1 \); where \( d \) is the depth of the tree. An example of a b-tree may be seen in Figure 4. As a side note: in the index blocks of a b-tree, the key value in the first block may be omitted to save space. It is assumed that if the look-up value is less than the second value in the index then it is covered by the first key value.
Example of a Balanced Tree

Figure 4
To look-up a value in a b-tree, first search for an index with the key value, $k$. This is done in a manner similar to look-up in ISAM. We must find a cover for the key by examining the keys in the first index record. Remember, the cover is the value $k_1$ where $k_1 < k < k_2$. Keep checking the keys in the greater indices until a cover is found. Trace this path down to the record block. Examine this block for a record with the key value.

Inserting a new record in a b-tree means that the new key value must be sorted into the existing keys. First perform a look-up to determine into which block the new record is to go. If there are fewer than $2e-1$ records in this block, then insert the new record in sorted order in the block. Update the index key if the new key is smaller than the old. If there are already $2e-1$ records in the block, create a new block and divide the records from the first block and the new record into two groups of $e$ records each. The parent index for the block must be updated in a similar add-divide-update maneuver if a new block is added.
To delete a record from a b-tree, first apply look-up. Set the delete bits in the header of the record and check to see if this delete unbalances the block. If it does, adjust adjacent blocks to re-balance the tree.

Look-up for b-trees has a maximum order of $\approx 1 + \log_d(n/e)$. Insert and delete average $\approx 2 + \log_d(n/e)$. This places b-trees somewhat ahead of ISAM. B-trees, like ISAM, do better than hashing when it comes to range queries.

Select-Project-Join

The look-up operation of the Query Processing System is used to access data in order to answer a question asked by a user. The question, or query, must be assessed so that the relations with the desired data records are accessed. This is referred to as query evaluation. Query evaluation operators include select, project, and join. These relational algebra operators were introduced in Table 1. Figures 5-7 show examples of these operators on our sample database.
The request to select Macy's record would look like:

one_bun = $bunny.name = "Macy"

This would pull up the record:

<table>
<thead>
<tr>
<th>ear#</th>
<th>name</th>
<th>sex</th>
<th>breed</th>
<th>color</th>
<th>reg#</th>
<th>litter#</th>
<th>quality</th>
<th>b_price</th>
<th>s_price</th>
</tr>
</thead>
<tbody>
<tr>
<td>241</td>
<td>Macy</td>
<td>F</td>
<td>lop</td>
<td>black</td>
<td>72915A</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example of a Selection

Figure 5
The request to project out just the tuples for "saved" bunnies and the service numbers would be:

\[ \text{total} = \pi \text{litter.bucks.left} \land \text{litter.does.left} \land \text{litter.service#} \]

This produces a small relation called total that looks like:

<table>
<thead>
<tr>
<th>service#</th>
<th>bucks.left</th>
<th>does.left</th>
</tr>
</thead>
<tbody>
<tr>
<td>75</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>76</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>77</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>80</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>86</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>87</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

Example of a Projection

Figure 6
The request to join two records would look like:

Final_outcome = service \& breeding

This would produce a new relation called Final_outcome:

```
<table>
<thead>
<tr>
<th>service#</th>
<th>buck#</th>
<th>doe#</th>
<th>date_bred</th>
<th>date_due</th>
<th>tested</th>
<th>nestbox</th>
<th>kindled</th>
</tr>
</thead>
<tbody>
<tr>
<td>74</td>
<td>129</td>
<td>242</td>
<td>051589</td>
<td>061589</td>
<td>false</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>75</td>
<td>129</td>
<td>242</td>
<td>053089</td>
<td>063089</td>
<td>true</td>
<td>062789</td>
<td>3</td>
</tr>
<tr>
<td>76</td>
<td>129</td>
<td>241</td>
<td>021290</td>
<td>031590</td>
<td>-</td>
<td>031290</td>
<td>4</td>
</tr>
<tr>
<td>77</td>
<td>540</td>
<td>514</td>
<td>031090</td>
<td>041090</td>
<td>-</td>
<td>040790</td>
<td>1</td>
</tr>
<tr>
<td>80</td>
<td>129</td>
<td>242</td>
<td>033090</td>
<td>043090</td>
<td>true</td>
<td>042790</td>
<td>4</td>
</tr>
<tr>
<td>85</td>
<td>540</td>
<td>530</td>
<td>040990</td>
<td>051090</td>
<td>false</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>86</td>
<td>540</td>
<td>514</td>
<td>042990</td>
<td>053090</td>
<td>true</td>
<td>052790</td>
<td>2</td>
</tr>
<tr>
<td>87</td>
<td>129</td>
<td>241</td>
<td>050190</td>
<td>060190</td>
<td>true</td>
<td>053090</td>
<td>3</td>
</tr>
</tbody>
</table>
```

Example of a Join

Figure 7
In the general sense, the larger and more numerous the relations involved in the evaluation of the query, the more accesses to secondary storage must be performed in order to bring in the desired data.

The brute force approach to evaluating a query is to first access and join the records of the relations involved, then select the records that meet all the selection criteria, and finally project out the desired components. In the traditional system much depended on how the query was written because this determined how the relations were joined which in turn determined how the records were accessed from memory. The best query would join the relations at least one attribute in common BUT with the identical values. This would keep the size of the final relation manageable. It was up to the programmer to guess at the best, or optimal query. What is wrong with this system? Not everyone is efficient at posing "optimal" queries. A person requesting data had to be experienced not only with the query system but also with the data records themselves. These systems had little tolerance for ignorance.
DBMS designers added to the complexity problem when they began to implement more user-friendly query interface systems allowing the users more flexibility in the way the queries were formulated. The user was replacing the programmer as it "easier" to formulate queries. Soon, a large amount of the "guess-work" of how to join the relations was lifted from the shoulders of the human and was being pushed onto the evaluation software. Systems considered "user-friendly" were more concerned with how easy it was for the user to formulate a query than with how easy it was for the software to evaluate the query. Naive users, those inexperienced with the use of computers, desired a "push-button" system, one that required minimal interaction. They did not customarily have a clear understanding of the file structure or how to manipulate it. New systems being implemented began to show up with a variety of query evaluation algorithms to help the situation; software processors that re-wrote the users' queries into something that the query processor could understand.

In these new and improved systems more than one algorithm could effectively solve the query. The
problem now was which one was the best? Obviously, the algorithm which made fewer block accesses and took less time would be the best. But costing all of the available options took more time than actually answering the query using the original representation. The search for one great omni-powerful costing model began. The most popular model to meet the challenge is that developed by Yao. Since the increase in retrieval time is due mainly to the increased complexity of block accesses, we will concentrate our optimization efforts in the area of the query processing subsystem.

Two types of query processors are being used in commercial DBMSs; algebraic manipulation and cost-estimation strategies. Manipulating the algebraic representation of a query may be accomplished through the use of heuristics. Heuristic optimization methods are well known and well used. We cover the most popular in Chapter 3. Cost estimating is more involved and less used than heuristic methods. It considers physical attributes of the data such as indices and links. Yao's cost model covers many different possible algorithms. We describe this model in the next sub-chapter.
Three abstract data models are currently used to represent and manipulate real world data. They are the relational, network, and hierarchical models. The majority of query systems today use the relational model. It provides the most user-friendly interface and a wide degree of design freedom. Remember, this ease of use has its price, complexity, which translates to a slow retrieval.

We have restricted our study to the optimization of select-project-join queries. We are using join and not cartesian product because although cartesian product uses more time (and therefore would seem to be more interesting), join is more popular. We use select and project because they are the two basic operations of query processing. We have also limited our study to queries on local relational databases. Distributed systems pose other, equally difficult problems, but also contain some interesting decomposition and parallel processing strategies [Ullm1] [Ullm2] [King] [Wong2].
The Cost Model

As covered in the previous chapter, one cost model needs to be used in order to effectively evaluate the performances of the large number of query algorithms available. We chose the cost model developed by [Yao]. We decided to use this model because it is the standard on which most new study in optimization is based.

In this model, the queries must be decomposed or restricted to two variable queries. This involves applying some decomposition algorithm to the queries. [WoYo]'s method for breaking multi-variable queries into smaller components is covered in Chapter 3.

The cost equations of Yao's model consider storage structures and use a small set of access operators. The next paragraph lists some necessary definitions. The algorithms that Yao used as examples are introduced after the definitions. His work with a large variety of algorithms is presented. The next chapter improves on the works of [Yao] by explaining how [WoYo]
decomposes queries and how [SwGu]'s method of ordering queries may help to optimize the retrieval.

The following definitions are used in the explanations involving storage structures.

**Indices**: An index on an attribute is a set of pointers to the records containing values of the attribute. Indices defined over attributes supply direct access to a relation's information. An "index scan" provides all the pointers in an index ordered by the attribute value. The pointers of an index are normally ordered within each attribute value. Indices are used in hash tables, b-trees, and ISAM files. When a relation, R, has an index on an attribute, a, then the number of blocks accessed to perform selection \( s_{a=c}(R) \) is approximately the number of blocks on which we find tuples for R where \( a=C \). The number of blocks accessed to examine the index is comparable (or smaller) than the number of blocks containing R's tuples. Therefore, in costing, we usually negate the cost of looking at the index when we must also access the full records.[Ullml].
Link: A link is a storage representation of a one-to-many relationship between two compatible relations, i.e. \( R.a \rightarrow S.b \) where \( R \) is the parent, \( S \) is the child, and \( a \) and \( b \) are components of \( R \) and \( S \) respectively. A link may be represented as a linked list or an array.

Compatible Relations: Relations whose domains are identified over the same set of values are considered compatible relations.

Chain Link: A chain link is a link where each parent record in \( R \) is the head of a linked list of all its child records in \( S \).

Clustering Link: Child records stored after their parent record are grouped in a clustering link. Orphan children are stored separately. Clustering links are pre-order traversal trees. B-trees use a non-clustering link.

Clustering Index: A clustering index is where the index is ordered on both the attribute values of the index and the pointers values of the index. ISAM uses this
method. The number of blocks needed to hold tuples with a certain value, C, is the number of blocks on which the tuples can be packed.

Alternatives to the brute force evaluation method described previously differ in the way that they use indices and links, in the order that they perform operations, and in their applicability. The alternate query evaluation algorithms described by Yao use four basic tasks in implementing the ordering, indexing, and links. These operations are Restriction (R), Join (J), Record Access (A), and Projection (P). Restriction selects rows, Projection selects columns, Record Access retrieves records from storage, and Join cross references and matches compatible relations. Different algorithms correspond to different sequences of these operations. Some algorithms take advantage of existing indices and some take advantage of parent-child links. Table 4 describes the symbols used in the following examples. Figure 8a shows an example of the basic one-variable query evaluation. Figure 8b shows the same query but with indexing added. In 8b, the query Q is divided into its indexed and non-indexed components:
Table 4
Symbols Used in Algorithm Examples (page 1 of 3)

(\v)
Join Value
All output of previous operation pertinent to the join value must be obtained before the next operation starts

(\O)
Complete Hold
All output of previous operation must be obtained before the next operation starts

\{p\}
Set of Pointers

\{r\}
Set of records

\{r\} \rightarrow \text{RF} \rightarrow \{r\}
Restriction Filter
Input = restriction predicate and a set of records
Output = records satisfying the predicate

\rightarrow \text{RF} \rightarrow \{p\}
Restriction Indexing
Input = restriction predicate
Output = set of pointers to records satisfying the predicate
Table 4 continued (page 2 of 3)

\[
\begin{align*}
\{v, \{p\}\} \text{ or } \\
\{v, \{r\}\} & \quad \rightarrow \quad \text{Join Filter} \\
\rightarrow \quad \text{Input} = \text{pointers (or records) grouped by join values and} \\
\text{the join values processed by the other join filter} \\
\text{Output} = \text{pointers (or records) grouped by join values} \\
\text{which also appear in the other join filter}
\end{align*}
\]

\[
\begin{align*}
\text{Join Indexing} & \\
\text{Input} = \text{join index} \\
\text{Output} = \text{record pointers grouped by join value}
\end{align*}
\]

\[
\begin{align*}
\{p\} \text{ or } \\
\{v, \{p\}\} & \quad \rightarrow \quad \text{Record Access} \\
\rightarrow \quad \text{Input} = \text{set of pointers, may be grouped by join value} \\
\text{Output} = \text{records grouped by join values}
\end{align*}
\]

\[
\begin{align*}
\{p\} \quad \rightarrow \quad \text{Intersection} \\
\rightarrow \quad \text{Input} = \text{two sets of pointers, one of them grouped by join values} \\
\text{Output} = \text{intersection of the two input sets, grouped by join values}
\end{align*}
\]
Table 4 continued (page 3 of 3)

\[ v, \{p\} \xrightarrow{\text{LK}} \{r\} \xrightarrow{\text{SORT}} \{v,\{r\}\} \]

**Link Scan**

*Input* = pointers to records containing a particular join value  
*Output* = all records containing the join value

**Sorting**

*Input* = records and the sorting (join) attribute  
*Output* = records grouped by ascending join values

\[ \{r\} \xrightarrow{\text{SORT}} \{v,\{r\}\} \]

**Sequential Scan**

*Input* = relation name  
*Output* = records in the relation

\[ \text{relation name} \xrightarrow{\text{S}} \{r\} \]

**Projection**

*Input* = set of records  
*Output* = records containing only the projected attributes

\[ \{v,\{r_1\}\} \xrightarrow{\text{p}} \{r\} \]

**Concatenation**

*Input* = two sets of records grouped by join values  
*Output* = concatenation of records with matching join values

\[ \{v,\{r_2\}\} \xrightarrow{\text{X}} \{v,\{r_1 \cdot r_2\}\} \]
\[ Q \rightarrow A \rightarrow P \rightarrow \{r\} \]

\[ Q_1 \rightarrow RI \{p\} \rightarrow RA \{r\} \rightarrow RF \rightarrow \{r\} \]

\[ P = P_1 \land P_2 \]

\[ P_1 = \text{indexed clauses only} \]
\[ P_2 = \text{partially or non-indexed clauses} \]

One-Variable Query Evaluation Example

Figure 8
\[ Q = Q_1 \land Q_2. \] The symbol \( \{p\} \) represents a set of pointers from the index. The symbol \( \{r\} \) represents the set of records in memory. The predicate, \( P \), is broken down into its indexed and non-indexed components, \( P_1 \) and \( P_2 \). If there are no indices, then the restriction index is not needed and may be eliminated.

Figure 9 shows the graphic representation of a simple two variable evaluation. The two variables, called files in the figure, may be processed with or without indexing. Figure 9a shows the breakdown without indexing. Figure 9b includes a join index. Note that in 9b, pointers, not records are sent to the join filter. This is a benefit of indexing.

Yao used examples of two popular, if more complex, two variable algorithms, Sort Relations and TID. The strategies to evaluate these algorithms are shown in Figures 10 and 11. The TID algorithm makes use of a feedback filter. This filter is valid iff \( \{v_1\} = \{v_2\} \); the value(s) for the join from the first relation equal the value(s) for the join of the second relation. It also must have the first relation processed at least up to the Restriction Filter (RF) before the processing
Simple Two-Variable Query Evaluation Example

Figure 9
Sort Relations

Figure 10
for the second relation begins. This exception withstanding, both TID and Sort Relations are adaptable to pipeline processing.

The four operations, R, J, A, and P, provide for $4! = 24$ combinations or different algorithms. Sort Relations is a sequence of RAPJ/RAPJ while the TID algorithm falls into the frame of RJAP/RJAP. In a two variable query we must take into account the required interaction at the join filters, the concatenations operations, and the projection operation. With sequencing alone, two variable query combinations are at least $12 \times 12 = 144$.

Yao's model considers seven different classes of algorithms using RAJ operations. Because projection must be the last step, it is not considered as a main operation in the sequencing. Yao does provide variations within the classes using projection and concatenation in various positions. The projection may be inserted anywhere after the access operator. Note that if the projection is used before the join, then the projection must be a modified version of the original. We will call it $P^*$. This $P^*$ includes all of
the attributes needed for the original projection plus the attributes (if different) that are required for the join. The use of \( P^* \) holds true for all classes described below. Also, the original projection must still occur after the access. Why would we want to add this seemingly complicated operation? Sometimes \( P^* \) can act as a restriction and limit the size of the search space. Table 4 explains the symbols for the figures accompanying the classes.

Class 1 RAJ. The Restrict-Access-Join combination may be seen in Figure 12. This class has three options for placement of the projection operation. Sort Relations is represented in version 3, where \( P^* \) follows the access and proceeds the join.

Class 2 JAR. This class swaps the restriction and join of class 1. It too, has three variations for projection. Figure 13 shows this class and its versions.

Class 3 JRA and RJA. Two combinations share this class. As seen in Figure 14, it is immaterial whether the restriction is performed first or the join is
Algorithm Class 1 (RAJ)

Figure 12
Algorithm Class 2 (JAR)

Figure 13
Algorithm Class 3 (JRA and RJA)
performed first (or if they are performed in parallel). The important factor is that the complete-hold operator (all values must be present to continue) follows the restriction and the join-value-hold (all output pertinent for the join value must be present to continue) follows the join filter. Both of these conditions are required in order for the intersection to take place. Note the catenation of records where the join values matched (the $x$ operator) at the end. This is necessary to "join" the relations. This class has four versions. The TID algorithm falls into version 2.

Class 4 ARJ. When there are no indices, it is necessary to first access the records, then perform the restriction and join. This class and the next class represent the access-first algorithms. There are two obvious methods of accessing the records, a sequential scan (S) or if the records are distributed then it may be preferable to access an existing index serially to obtain record pointers. Notice the similarity between this class and class 1; their back ends are the same. There are three versions to this class. Figure 15 shows this class.
Figure 15

Algorithm Class 4 (ARJ)
Class 5 AJR. As with the previous class, there are two methods for accessing the required records. The reversal of the join with the restriction gives one more version, a projection before the sort. This makes a total of four variations. Figure 16 shows this class.

Class 6 LAR. This class, as well as the next, takes advantage of existing links between the two files (variables) being scanned. The link must be defined over the join attributes. Links are stored as pointers from one file to the other. To obtain these links, the first file must be accessed. We will call the first file F and the second file G. One of the algorithms from class 1, 4, or 5 needs to be used on file F to obtain the links to file G. We may then use these links, as seen in Figure 17, to access the second file. There are only two versions to this class. We cannot project sooner than the (v) operator because not all of the pointers are available prior to the link. Note that the link-access has replaced the join-access of class 2.
Algorithm Class 5 (AJR)

Figure 16
Algorithm Class 6 (LAR)

Figure 17
Class 7 LRA and RLA. In this class the link-traversal replaces the join and the access of class 3. Note that all of the join-values must be present for the intersection to take place. This would require a 1:many link of parent to children records. There are three variations to this class as may be seen in Figure 18.

The classes defined above are performed per variable. Because we deal with two variable queries we must further define rules to combine these algorithms.

Let $i$ denote the algorithm class $i$ and $i/j$ denote the combination of two algorithms (called a "type") from classes $i$ and $j$.

The combination rules are as follows:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>L</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
<td>A/A</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>7</td>
<td>B/L</td>
</tr>
</tbody>
</table>
Example: Type 1/6 is a combination of a class 1 algorithm for file F and a class 6 algorithm for file G. The application of a class 6 algorithm follows after traversing the link between the first file and the second.

The rules allow for $5 \times 5 + 3 \times 2 = 25 + 6 = 31$ types of algorithms. This does not include variations within each class. The use of versions are denoted as $ik/jl$ where $i$ and $j$ are classes and $k$ and $l$ are versions within their classes. Now, it is easy to see that there are $17 \times 17 + 10 \times 5 = 339$ different types of algorithms covered in this cost model. Of course, not every algorithm may be used in every database application due to varying requirements between classes and applications. Using the storage parameters described in Figure 19 we may define the storage structures required for each algorithm class. These requirements are listed in Table 5. Figure 19 explains the symbols, except for $r'$, which is in Table 6.
Figure 18

Algorithm Class 7 (LRA and RLRA)
i = relation
j = relation
m = number of relations
n_i = number of attributes in relation i

1 ≤ i ≤ m
1 ≤ j ≤ m
1 ≤ k ≤ n_i

\[ d_{ij} = \text{clustering link} \quad \begin{cases} 1 & \text{link between relations } i \text{ and } j \\ 0 & \text{otherwise} \end{cases} \]

\[ d'_{ij} = \text{parent link} \quad \begin{cases} 1 & \text{link between relations } i \text{ and } j \\ 0 & \text{otherwise} \end{cases} \]

\[ d''_{ij} = \text{child link} \quad \begin{cases} 1 & \text{link between relations } i \text{ and } j \\ 0 & \text{otherwise} \end{cases} \]

\[ d'''_{ij} = \text{chain link} \quad \begin{cases} 1 & \text{link between relations } i \text{ and } j \\ 0 & \text{otherwise} \end{cases} \]

\[ c_{ik} = \text{clustering index} \quad \begin{cases} -1 & \text{no index for } k^{\text{th}} \text{ attribute of the } i^{\text{th}} \text{ relation} \\ 0 & \text{non-clustering index} \\ 1 & \text{clustering index} \end{cases} \]

Storage Parameters

Figure 19
## Table 5

### Storage Parameter Requirements

<table>
<thead>
<tr>
<th>Class</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Restriction indices</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c_{i1} + c_{i2} + \cdots + c_{ir} &gt; r')</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| **Join index** |   |   |
| \(c_{ik} \geq 0\) | X | X |

| **Link type** |   |   |
| \(d_{ij} + d'_{ij} + d''_{ij} + d'''_{ij} > 0\) | X | X |

| **Link type** |   |
| \(d'_{ij} + d''_{ij} > 0\) | X |
### Table 6

**Cost Model Symbols**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>set of projection attributes</td>
</tr>
<tr>
<td>b</td>
<td>page size</td>
</tr>
<tr>
<td>c_i</td>
<td>join index (0=\text{non-clustering}) (1=\text{clustering})</td>
</tr>
<tr>
<td>d_{ij}</td>
<td>clustering link (1=\text{both parent } F_i \text{ &amp; child } F_j \text{ are clustered}) (0=\text{otherwise})</td>
</tr>
<tr>
<td>e_i</td>
<td>number of pages in the storage area</td>
</tr>
<tr>
<td>f_i</td>
<td>record size</td>
</tr>
<tr>
<td>n</td>
<td>number of attributes in the query</td>
</tr>
<tr>
<td>p_i</td>
<td>number of pages containing the file</td>
</tr>
<tr>
<td>q</td>
<td>restriction indexing selectivity</td>
</tr>
<tr>
<td>r_i</td>
<td>number of records in relation i</td>
</tr>
<tr>
<td>r'_i</td>
<td>number of records accessed (r_i \times w)</td>
</tr>
<tr>
<td>r''_i</td>
<td>number of records accessed to satisfy the join (r_i \times t_i)</td>
</tr>
<tr>
<td>s_k</td>
<td>index selectivity</td>
</tr>
<tr>
<td>t_i</td>
<td>join selectivity</td>
</tr>
<tr>
<td>u_k</td>
<td>number of values accessed</td>
</tr>
<tr>
<td>w</td>
<td>restriction selectivity</td>
</tr>
<tr>
<td>y</td>
<td>clustering index selectivity</td>
</tr>
<tr>
<td>z</td>
<td>tree degree</td>
</tr>
<tr>
<td>E</td>
<td>non-clustering selectivity</td>
</tr>
<tr>
<td>F_i</td>
<td>parent file (0=\text{primary key in projection}) (1=\text{otherwise}) (note: Yao uses &quot;file&quot; and &quot;variable&quot; to mean the same thing)</td>
</tr>
<tr>
<td>F_j</td>
<td>child file</td>
</tr>
<tr>
<td>P_{ij}</td>
<td>distance between the parent and the first child</td>
</tr>
</tbody>
</table>
Costing

Determining which algorithm is the most efficient requires the calculation of the access costs. The cost of an algorithm depends on the storage method used. Figure 19 and Table 5 describe the storage requirements for the seven algorithm classes in Yao's model. We assume the data storage is arranged in pages or blocks. We will refer to accesses in terms of pages. The cost of each algorithm and each version within the algorithm follows. Generally the cost is computed by adding the cost of each access operation involved.

We compute the cost of accesses for one relation. It is a simple matter to add two relation's costs together in order to determine the total cost. Let \( C_{ij} \) denote the cost of access for an algorithm in class \( i \) version \( j \). Tables 6 and 7 explain the symbols used in this section.

Class 1 - Figure 12 shows that the access components of this class are the Restriction Index (RI), the Access (A), the SORT J (SORT), and the catenation (x). These may be summed up as:
Cost Equations for Access Operations

Restriction Indexing:
\[ R(r_i) = \sum_{k=1}^{m} u_k (\log_2 r_i + (r_is_k)/b) \]

Join Indexing:
\[ I(r_i) = r_i/b \]

Record Access:
\[ A(\alpha, \beta) = x + (p_i - x)((\alpha r_i p_i)/p_i) \]
where \( x = p_i(1-1/p_i) r_i \beta \)

if used after restriction:
\[ \alpha = (\gamma \text{ clustering restriction index selectivity}) \]
\[ \beta = (\epsilon \text{ nonclustering restriction index selectivity}) \]

if used after join:
if used after join and restriction:
if used for relation scanning:

Sequential Scan:
\[ S(e_i) = e_i \]

Sorting:
\[ T(r,f_i) = 2((r^2 f_i)/b) \log_2 ((r^2 f_i)/b) \]

Joining:
\[ J(r,f_i) = (r^2 f_i)/b \]

Catenation:
\[ N(r,f_i) = (r^2 f_i)/b \]

Catenation (of pointers):
\[ N_1(r,f_i) = (r^2 s_i t_i)/b \]
Table 7 (page 2 of 2)

Cost Equations for Access Operations

<table>
<thead>
<tr>
<th>Cost Equation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1(r) = T(r, f_1) + (\delta/s_i)(T(r s_i, f_i))$</td>
<td>Projection before SORT: $P_1(r)$</td>
</tr>
<tr>
<td>$P(r) = (1/s_i)(T(r s_i, f_i))$</td>
<td>Projection after SORT: $P(r)$</td>
</tr>
<tr>
<td>$H(r) = \min (r; \prod_{a \in A} (1/s_j))$</td>
<td>File Size after projection: $H(r)$</td>
</tr>
<tr>
<td>$L(q) = d_{ij} r_j t_j (P_{ij} + (r_i t_i q - 1)P_i)$ + $(1-d_{ij}) r_j t_j (c_i (1 + (r_i t_i q - 1)P_i) + (1-c_i) r_i t_i q)$</td>
<td>Link Access(1:m): $L(q)$</td>
</tr>
<tr>
<td>$L'(q) = d_{ij} r_i t_i P_{ij} + (1-d_{ij}) r_i t_i q$</td>
<td>Link Access(m:1): $L'(q)$</td>
</tr>
</tbody>
</table>

$r$ is the size of the file being accessed
\[ C_{11} = R(r_i) + A(\alpha, \beta) + T(r'_i, f_i) + J(r'_i, f_i) + N(r''_i, f_i). \]

When the projection is performed before the join, as it is in version 2, the size of the relation is reduced. This requires the use of a "new" file size, \( H \). Also, the record size is reduced. We use 'g' to denote this:

\[ C_{12} = R(r_i) + A(\alpha, \beta) + T(r'_i, f_i) + J(H(r'_i), g_i) + N(H(r''_i), g_i). \]

If the projection is moved before the sort operation, as in version 3, a new sort is required to remove duplicate records. But the cost of the original sort is reduced. This gives us:

\[ C_{13} = R(r_i) + A(\alpha, \beta) + P_1(r'_i) + T(H(r'_i), g_i) + J(H(r'_i), g_i) + N(H(r''_i), g_i). \]

Class 2 - Figure 13 shows that the main cost of this class is obtaining the record pointers, accomplished by the join index (I). Also, the pointers are further filtered by the join (J) before they are accessed (A) and catenated (N). For version 1 of class 2 we have:
\[ C_{21} = I(r_1) + J(r_1,1) + A(\alpha, \beta) + N(r''_1, f_1). \]

By moving the projection in front of the catenation, we have version 2. Of course moving the projection changes the size of the records:

\[ C_{22} = I(r_1) + J(r_1,1) + A(\alpha, \beta) + P(r'_1) + N(H(r''_1), g_1). \]

Version 3 of this class is obtained by performing the catenation operation on the record pointers before the records are accessed:

\[ C_{23} = I(r_1) + J(r_1,1) + N_1(r_1) + A(\alpha, \beta). \]

Class 3 - This class involves indices on both the join and the restriction. Figure 14 is the reference for this class. The cost for accessing records using version one of this algorithm is:

\[ C_{31} = R(r_1) + I(r_1) + J(r_1,1) + A(\alpha, \beta) + N(r''_1, f_1). \]

By performing the projection before the catenation, we have version 2:
$C_{32} = R(r_i) + I(r_i) + J(r_i, l) + A(\kappa, \theta) + P(r''_i) + N(H(r''_i), g_i)$. 

Version 3 performs the catenation before the record access. This is to take advantage of pointers for the restriction and join indices:

$C_{33} = R(r_i) + I(r_i) + J(r_i, l) + N_1(r_i') + A(\kappa, \beta)$. 

If we move the catenation after the intersection operation, as in version 4, we have:

$C_{34} = R(r_i) + I(r_i) + J(r_i, l) + N_1(r_i) + A(\kappa, \beta)$. 

Because the only difference between $C_{33}$ and $C_{34}$ is the record access in the catenation, and one costs more than the other, we may eliminate the more expensive one, $C_{34}$.

Class 4 - Figure 15 shows us that we need to calculate the best cost of one of the sequential retrieval methods for the records ($S$, or $I$ plus $A$). Then we need to add in the sorting ($T$), join ($J$), and catenation ($N$). This gives us:
$C_{41} = \min[S(e_1); I(r_1) + A(\alpha, \beta)] + T(r'_i, f'_i) + J(r'_i, f'_i) + N(r''_i, f'_i)$.

Version 2 performs the projection before the join filter, giving:

$C_{42} = \min[S(e_1); I(r_1) + A(\alpha, \beta)] + T(r'_i, f'_i) + P(r'_i) + J(H(r'_i), g_1) + N(H(r''_i), g_1)$.

When we move the projection even further up the processing stream to be placed after the sort then we have:

$C_{43} = \min[S(e_1); I(r_1) + A(\alpha, \beta)] + P_1(r'_i) + T(H(r'_i), g_1) + J(H(r'_i), g_1) + N(H(r''_i), g_1)$.

Class 5 - This class, like class 4, requires the determination of the best sequential scan. However, each version in this class is slightly more expensive than those in class 4. Since $C_{51} > C_{41}$, $C_{52} > C_{42}$, $C_{53} > C_{43}$, $C_{54} > C_{43}$ and they are applicable in identical situations we may effectively eliminate this class from further consideration. Viewing Figure 16 will show the
similarities between this class and the previous. For the exercise, the cost equations for class 5 are:

\[ C_{51} = \min \{ S(e_i); I(r_i) + A(\alpha, \beta) \} + T(r_i, e_i) + J(r_i, e_i) + N(r''_i, e_i) \].

\[ C_{52} = \min \{ S(e_i); I(r_i) + A(\alpha, \beta) \} + T(r_i, e_i) + J(r_i, e_i) + P(r_i) + N(H(r''_i), g_i) \].

\[ C_{53} = \min \{ S(e_i); I(r_i) + A(\alpha, \beta) \} + T(r_i, e_i) + P(r_i) + J(H(r_i), g_i) + N(H(r''_i), g_i) \].

\[ C_{54} = \min \{ S(e_i); I(r_i) + A(\alpha, \beta) \} + P_1(r_i) + T(H(r_i), g_i) + J(H(r_i), g_i) + N(H(r''_i), g_i) \].

Class 6 - Figure 17 shows this class and its versions. The first two versions are for a 1:many link. The second two versions are used with a many:1 link. The costing for this class is simple; add the link access cost \( L \) and the catenation cost \( N \):

\[ C_{61} = L(1) + N(r''_i, e_i) \].
Putting the projection before the catenation adds one more element to the cost, but it also reduces the size of the records:

\[ C_{62} = L(1) + P(r''_i) + N(H(r''_i),g_i). \]

These two versions are very similar to the previous two; the difference being the type of link.

\[ C_{63} = L'(1) + N(r''_i,f_i). \]

\[ C_{64} = L'(1) + P(r''_i) + N(H(r''_i),g_i). \]

Class 7 - Restrictions indices are the main feature of the algorithms in Figure 18. They are used to reduce the access. Note that the first four versions are comparable to those of class 6. However this class has one additional 1:many version and one many:1 version. All six are:

\[ C_{71} = R(r_i) + L(q) + N(r''_i,f_i). \]

\[ C_{72} = R(r_i) + L(q) + P(r''_i) + N(H(r''_i),g_i). \]
\[ C_{73} = R(r_1) + L'(q) + N(r''_1, f_1). \]

\[ C_{74} = R(r_1) + L'(q) + P(r''_1) + N(H(r''_1), g_1). \]

\[ C_{75} = R(r_1) + N_1(r''_1) + L(q). \]

\[ C_{76} = R(r_1) + N_1(r''_1) + L'(q). \]

It is relatively easy to calculate the minimum equations of all the various versions/classes. First we find the minimum for each version within a class. Since we do not need to include version 4 of class 3 and the entire class 5, they are not listed below.

\[ C_1 = \min\{C_{11}; C_{12}; C_{13}\} \]
\[ C_2 = \min\{C_{21}; C_{22}; C_{23}\} \]
\[ C_3 = \min\{C_{31}; C_{32}; C_{33}\} \]
\[ C_4 = \min\{C_{41}; C_{42}; C_{43}\} \]
\[ C_6 = \min\{C_{61}; C_{62}; C_{63}; C_{64}\} \]
\[ C_7 = \min\{C_{71}; C_{72}; C_{73}; C_{74}; C_{75}; C_{76}\} \]

Then we determine the minimum of these classes:

\[ C = \min\{C_1; C_2; C_3; C_4; C_6; C_7\} \]
Not all algorithm classes are relevant for a particular database. Storage structures play a large role in determining applicability. Therefore, not all algorithm classes and versions need to have all costs estimated for a particular query.

Current optimization algorithms have been classified into this system. SEQUEL is of type 11/31 with the join index and records for the second relation being accessed multiple times. Figure 20 shows this.

CHAPTER 3

Heuristics

Currently the most common method used by optimization algorithms is that of heuristics. Heuristics are also known as the "trial and error" method. We use heuristics because, simply stated, they work. They work for a large class of problems. Heuristics are the best known method for solving NP-Complete problems such as shortest path and knapsack. Other methods may prove better, but always for some sub-set of the problem. Heuristics work, in a general sense, for a large set of
SEQUEL Classification

Figure 20
problems. Heuristics are divided into algebraic manipulation and physical storage strategies. Algorithms in Yao's classes 6 and 7 rely on physical storage information, specifically on links between two files.

The general principle behind heuristic optimization is that as the size and number of relations grow, so do the time and space required to perform the operations. The most popular heuristic strategy used is to push selections down the parse tree as far as possible. This automatically reduces the size of the relation involved. Figure 21a shows the parse tree for Table 3. Figure 21b depicts the same parse tree with selections pushed down. Pushing projections down the parse tree also reduces the relation size. As a general rule we perform selections before projections because projections tend to require more work, consistently in the area of removing duplicate values. Also, if a join follows then the projection must be modified to include the attributes needed in the join.

To manipulate the parse tree, we start with an expression tree and using the equivalences given in
Parse Tree for Table 3

Figure 21 (page 1 of 2)
Table 8 to transform the graph. As an example: if $R$ and $S$ are relations and we have a condition $C$ that can be expressed as a series of conditions $c_1 \land c_2 \land c_3$ where the attributes of $c_2$ belong to $R$ and those in $c_3$ belong to $S$ we can write the expression as:

$$\sigma_{c_1 \land c_2 \land c_3} (R \bowtie S) \equiv \sigma_{c_1} (\sigma_{c_2} (R) \bowtie \sigma_{c_3} (S)).$$

To push projections down the parse tree we use the equivalences in Table 9.

Other common heuristics are to combine selections with accompanying cartesian products to produce equi-joins, combine sequences of unary operations such as selections and projections, preprocess files appropriately by sorting and developing secondary indices, and evaluate options before processing [Ullml]. Other considerations are to re-use intermediate results by determining common subexpressions.

Common subexpression isolation creates a shareable resource. A common subexpression is a part of a query.
For the next two tables the following definitions hold:

- $C$ is a condition to be met by the query
- $R$ is a Relation
- $S$ is a Relation
- $x$ is an attribute of a relation
- $y$ is the smallest set of attributes
- $xy$ contains all attributes mentioned in $C$

### Table 8

**Equivalences for Selections**

- $\sigma_C(\pi_x(R)) \equiv \pi_x(\sigma_C(R))$
- $\sigma_C(R \cup S) \equiv \sigma_C(R) \cup \sigma_C(S)$
- $\sigma_C(R \circ S) \equiv \sigma_C(R) \circ \sigma_C(S)$
- $\sigma_C(R - S) \equiv \sigma_C(R) - \sigma_C(S)$

### Table 9

**Equivalences for Projections**

- $\pi_x(\sigma_C(R)) \equiv \pi_x(\sigma_C(\pi_x(R))$
- $\pi_x(R \cup S) \equiv \pi_x(R) \cup \pi_x(S)$
that defines an intermediate result. Determining those expressions that are used more than once and separating them so that they are performed only once allows us to realize substantial savings in time. Removing common subexpressions involves a bottom-up merge procedure in an operator graph. [Hall] describes this process. [Maie] explains common subexpression simplification.

The ordering of joins may prove to make a difference in the speed and use of memory. We discuss this under the sub-chapter "Ordering Joins".

These heuristic methods are referred to as algebraic manipulation. This type of simplification improves the query without needing knowledge of the actual data or the physical structures. But, to get good "improvements" one can't rely solely on universal algebraic heuristics; one needs information on the physical database structure or at least decent estimates [JKS]. Quantitative information about the storage requirements of a database is heavily relied upon in a few query algorithms. Unfortunately these estimates are difficult to obtain. This is especially true for sizes of intermediate data structures being
constructed [JKS]. For this reason, many systems stick with this type of optimization. [Rich] has a parameter system that computes the result size for any sequence of relational algebra operations under the assumptions that the size of all universes and each possible projection is known. Actual estimates of these can be found in [Gel]. The size of the intermediate results and the actual secondary storage access depends on physical storage structures involved, the size of the buffers, and how many elements may be accessed at one time [JKS].

Each of the heuristics mentioned above has been incorporated into numerous different algorithms. Each algorithm performs its services well for its specific tasks. But no one algorithm could possibly be used for every query system on the market. Each system requires its own optimized algorithm. Many use more than one algorithm. In these instances the query subsystem must decide which algorithm is best. But, the costing of multiple algorithms has been shown to be combinatorial hard [SwGu]. One method of restricting the costing function is to work with only two variables at a time.
Decomposition

The multi-variable decomposition method described in [WoYo] is simple to understand and is used by many systems requiring queries to be decomposed into one or two variable queries. Their method assembles small pieces rather than pares down the cartesian product. Because this method uses no cartesian product there is no geometric growth. The number of tuples to be scanned is kept as small as possible by using restrictions and substitutions and by working with the knowledge of pages.

A two variable query is defined as a predicate of two variables where each variable can have the form \((V.a \: v)\) or \((V_1.a \: \theta \: V_2.b)\) where \(V\) is a relation, \(a\) and \(b\) are attributes, \(v\) is an actual value of \(a\), and \(\theta\) is an operation in the set \(\{=, <, >, \geq, \leq\}\) [Yao]. The example in Table 3 is a three variable query, it involves the ranges of three different relations, "bunny", "litter", and "service".

The decomposition method in [WoYo] will break queries down to one variable in most instances. This is more
resolution then we need, but we may still use the method because it must decompose the queries into two variables before finalizing the reduction to one variable. The decomposition is achieved through the repeated use of two steps:

(1) Detachment, breaking up of a query into two queries such that the queries have only one variable in common and

(2) Tuple substitution, an n-variable query is replaced by a family of (n-1) variable queries resulting from substituting for one of its variables, tuple by tuple.

Tuple substitution or instantiation is used to reduce a two variable query into one variable. Our work needs only the detachment step. We do not need the tuple substitution step so it will not be discussed in this paper.

Our example of detachment uses the notation from QUEL, a database language. Figure 22 shows the syntax of a QUEL statement. Figure 23 is our database example from
RANGE OF (variable) IS (relation)
SELECT (attribute) RETRIEVE INTO result_name (target_list)
WHERE (qualifications)

The SELECT and WHERE are optional.

The QUEL Statement

Figure 22

RANGE OF (L,S,B) IS (litter, service, bunny)
SELECT B.name WHERE (L.bucks_saved > 0)
   AND (L.does_saved > 0)
   AND (B.breed = 'lop')
   AND (L.service# = S.service#)
   AND (S.buck# = B.ear#)

The QUEL Representation of Table 3

Figure 23
...Table 3 written in QUEL. Figure 24 is Figure 23 decomposed into a series of two variable queries using detachment.

Notice that in Figures 23-24 the decomposition of the query's relations into sub-queries simply involves first separating out all of the one variable relations and second breaking up the remaining relations into two-variable relations. We save only the attributes needed in the following sub-queries. They, in essence, become intermediate results. The effect is that the search space may be reduced. The order in which the relations originally appear is usually the order in which they are decomposed. If we wish to possibly improve on our retrieval time then we need to order the relations in the query in such a way as to minimize the search space. In the query from Table 3, the order does not appear to be very important. And indeed this may be the case. So, to prove that ordering joins does make a difference, we introduce a new more complex query, "Get the names of bucks who have sired litters containing only female kits". This is represented in Figures 25 and 26.
Q_1 = RANGE (L) IS (litter)
    RETRIEVE INTO litter1 (L.service#)
    WHERE (L.bucks_saved > 0)
    AND (L.does_saved > 0)

Q_2 = RANGE (B) IS (bunny)
    RETRIEVE INTO bunny1 (B.ear#)
    WHERE (B.breed = 'lop')

Q_3 = RANGE (L,S) IS (litter, service)
    RETRIEVE INTO servicel (S.buck#)
    WHERE (L.service# = S.service#)

Q_4 = RANGE (S,B) IS (service, bunny)
    RETRIEVE INTO bunny2 (B.name)
    WHERE (S.buck# = B.ear#)

Multi-Variable Decomposition

Figure 24
Figure 25 is probably the worst ordering that we could have used. Notice we have sub-query with more than one selection attribute. Whereas Figure 26 is a better ordering. Not only do we have smaller sub-query selections but also, if the actual size of the intermediate queries are taken into account, those in Figure 26 are generally smaller.

So, the question remains, how do we decide on an optimal ordering? First costs must be assigned to each possible pairing. These costs are calculated as the number of accesses needed to pull in all necessary tuples. Then the possible combinations need to be generated. Finally each total solution needs to be checked to find the least cost solution. If this sounds familiar, it is. This is an NP-Complete problem, similar to shortest path or traveling salesman. It is well known that a great effort has been applied to solving this class of problems. Swami and Gupta have done extensive work in a sub-area with which we are concerned. This area is the determining (quickly) of an efficient ordering of joins.
RANGE OF (L, Br, S, Bu) IS (litter, breeding, service, bunny)
SELECT Bu.name WHERE (Bu.ear# = S.buck#)
   AND (Br.service# = S.service#)
   AND (Br.kindled = L.does_born)

Q1 = RANGE OF (Bu, S) IS (bunny, service)
SELECT Bu.name, S.service# WHERE (Bu.ear# = S.buck#)

Q2 = RANGE OF (Br, S) IS (breeding, service)
SELECT Br.kindled WHERE (Br.service# = S.service#)

Q3 = RANGE OF (Br, L) IS (breeding, litter)
SELECT L.service# WHERE (Br.kindled = L.does_born)

\[ Q \]
\[ Q_1 \]
\[ Q_2 \]
\[ Q_3 \]

\textbf{Decomposition - Example 1}

\textbf{Figure 25}
RANGE OF (L, Br, S, Bu) IS (litter, breeding, service, bunny)
SELECT Bu.name WHERE (Br.kindled = L.does_born)
    AND (Br.service# = S.service#)
    AND (Bu.ear# = S.buck#)

Q1 = RANGE OF (Br, L) IS (breeding, litter)
SELECT L.service# WHERE (Br.kindled = L.does_born)

Q2 = RANGE OF (Br, S) IS (breeding, service)
SELECT Br.kindled WHERE (Br.service# = S.service#)

Q3 = RANGE OF (Bu, S) IS (bunny, service)
SELECT Bu.name, S.service# WHERE (Bu.ear# = S.buck#)

Decomposition - Example 2

Figure 26
Ordering Joins

The work presented in [SwGu] tries to solve the problem of finding the best join order for relations in algebraic queries. The join method used hash-join. The authors state that other methods could be employed but that hash-join gives satisfactory results for the effort involved.

The work by Swami and Gupta was presented as an investigation into optimizing queries with a large number of joins. A large number has been defined as greater than 10. However, their methods are well suited for smaller queries such as those presented in Figures 25-26.

The techniques studied include two well known methods of determining best cost, Local Search [Papa] and Simulated Annealing [NSS]; plus two lesser known methods, Perturbation Walk and Quasi-random Sampling.

To apply any of the methods described in [SwGu], the query must be represented as a join tree. More specifically, an outer linear join tree (OLJT). Our
examples in Figures 25-26 are already OLJT's. But they arrived in that form because we knew ahead of time that it was necessary and biased the examples. So we will provide a short explanation of how to create an OLJT. An OLJT is a binary join tree where non intermediate result is on the right-hand side, or outside. Figure 27 shows the progression of a four variable, three join query from a general query into an OLJT.

Out of the four methods described by Swami and Gupta, two stand out. One has proven to be the most efficient while the other is certainly the most interesting. These two methods are local search and simulated annealing. The local search method was further broken down into two sub-methods, iterative improvement and sequence heuristic. All methods involve searching a state space where all possible join trees are the states. When the algorithms finish, either by design or by time constraints, then the join tree with the least cost becomes the solution.

Local search is referred to as local optimization in [SwGu]. It is a technique with many variations. The initial state is chosen at random. Following states
Query = $a \times b \times c \times d$

Join Tree =

Linear Join Tree =

Outer Linear Join Tree =

Figure 27
are selected at random from adjoining states. The move to a new state is accepted only if it is of less cost than the current state. Local minima are of interest in this scheme. A local minimum is detected though various means. One mean is though the use of a timeout; if there is no more time left to search more states then the current state is the minimum and therefore the solution. Another is to approximate a minimum based on a random sampling of neighbors. The distinguishing factor between iterative improvement and sequence heuristic is in the determination of a new start state once a local minimum is achieved and time has not run out. In iterative improvement the new state is generated in the same manner as the start state, at random. In sequence heuristic the new state is determined by making a known number of moves from the current minimum.

Simulated annealing is definitely an interesting method. It was originally designed as an analogy to the process of annealing crystals from liquid solution. The terms "freezing condition" and "temperature" used in the popular algorithms originate from the analogy. Simulated annealing is really just a variation on
local search. Where it differs is in the area of determining the next move. Rather than pick an always lower cost new state at random or from adjoining neighbors, simulated annealing can accept any new state. The deciding factor is the use of a probability function. The probability function is derived from the mathematical model used in the annealing process. The goal in simulated annealing is to achieve a "frozen system", one in which no more moves are possible. The probability functions come into play when the initial states keep the temperature "high". The higher the temperature the more chances that a state will be accepted. A total explanation of the freezing condition and determination of initial temperature may be gathered from [JAMS], [Kirk], [SwGu], and [NSS].

Of an interesting note: simulated annealing is used in neural nets and artificial intelligence applications. The systematic rearrangement of the parameters involved in this process has inspired great research [Kirk].

The comparative experiments performed by Swami and Gupta show, time and again, that iterative improvement is the best method for determining the "optimal" join
order. But the costs associated with each state in the state space did not take into account intermediate results, such as our used in the decomposition. They used simple costs, not costs recalculated to take into account the possible variations upon changing intermediate results. To include these costs drives up all total costs, including those computed for iterative improvement.

The work by Swami and Gupta is clearly most exhaustive in determining a comparison of models. When we first started our research into the combination of Yao's, Wong/Youssefi's and Swami/Gupta's works we thought that we had a great combination. But, we discovered that even though Swami and Gupta's Local Search, specifically iterative improvement, proved the average best strategy for determining a good join order of queries, the costing of the pairing of intermediate results increased the total evaluation time beyond any value. It is quite possible to determine the best join sequence using iterative improvement, even with our intermediate results. This is because their work was designed with large join queries in mind. However, we are interested in the average query, which does not
afford us the time allotted and tolerated for large queries. We therefore decided that it is not feasible to determine a best join order if we use Yao's and Wong/Youssefi's works.

CHAPTER 4

Conclusion

Yao's cost estimating modeling is essentially a complete search effective for many algorithms. An advantage this model has over strict heuristics is that it is effective and efficient. It takes into account storage structures, unlike general heuristics. Also, this model would be suited for consideration in parallel processing and dedicated processing. The use of Wong and Youssefi's decomposition strategy provides a tried and true method for obtaining the two variable queries required by Yao's model. The inclusion of Swami and Gupta's ordering of joins, although intended to improve upon this work by insuring that queries are paired in an "optimal" order, did not enhance the model.
The numerous papers on optimizing only improve expressions under some cost criteria but do not claim to produce a least cost expression [SAGI]. These optimization theories sound good on paper but have yet to prove their worth when implemented. Currently, commercial systems use a variety of heuristic methods in one or another optimization schemes. Yao's model, together with the preprocessing method of Wong and Youssefi allow for the efficient estimation of a wide range of available algorithms. However, the use of simple heuristics will remain the method of choice for most relational optimization systems.
Other Areas:

There has been study in the area of changing the query-human interface of SQL type queries to be more human-like [Meln]. As the big problem with the interface is that the non-procedural query systems already produce sub-optimal queries, the Natural Language Interface, in breaking the user's queries down into SQL, may potentially create more complex and less optimal queries than current user interface systems produce. This seems to demand a decent query optimizer for those SQL systems. In a different light, this new interface could be used to produce "optimal" queries if designed with optimization in mind.