Introduction to Query Processing and Optimization

Michael L. Rupley, Jr.
Indiana University at South Bend
mrupleyj@iusb.edu

ABSTRACT

All database systems must be able to respond to requests for information from the user—i.e. process queries. Obtaining the desired information from a database system in a predictable and reliable fashion is the scientific art of *Query Processing*. Getting these results back in a timely manner deals with the technique of *Query Optimization*. This paper will introduce the reader to the basic concepts of query processing and query optimization in the relational database domain. How a database processes a query as well as some of the algorithms and rule-sets utilized to produce more efficient queries will also be presented. In the last section, I will discuss the implementation plan to extend the capabilities of my Mini-Database Engine program to include some of the query optimization techniques and algorithms covered in this paper.

1. INTRODUCTION

Query processing and optimization is a fundamental, if not critical, part of any DBMS. To be utilized effectively, the results of queries must be available in the timeframe needed by the submitting user—be it a person, robotic assembly machine or even another distinct and separate DBMS. How a DBMS processes queries and the methods it uses to optimize their performance are topics that will be covered in this paper.

In certain sections of this paper, various concepts will be illustrated with reference to an example database of cars and drivers. Each car and driver are unique, and each car can have 0 or more drivers, but only one owner. A driver can own and drive multiple cars. There are 3 relations: *cars*, *drivers* and *car_driver* with the following attributes:

The *vehicles* relation:

Attribute	Length	Key?
vehicle_id	15	Yes
make	20	No
model	20	No
year	4	No
owned_by	10	No

The *drivers* relation:

Attribute	Length	Key?
driver_id	10	Yes
first_name	20	No
last_name	20	No
age	2	No

The *car_driver* relation:

Attribute	Length	Key?
cd_car_name	15	Yes*
cd_driver_id	10	Yes*

2. WHAT IS A QUERY?

A database query is the vehicle for instructing a DBMS to update or retrieve specific data to/from the physically stored medium. The actual updating and retrieval of data is performed through various "low-level" operations. Examples of such operations for a relational DBMS can be relational algebra operations such as project, join, select, Cartesian product, etc. While the DBMS is designed to process these low-level operations efficiently, it can be quite the burden to a user to submit requests to the DBMS in these formats. Consider the following request:

"Give me the vehicle ids of all Chevrolet Camaros built in the year 1977."

While this is easily understandable by a human, a DBMS must be presented with a format it can understand, such as this SQL statement:

select *vehicle_id*
from *vehicles*
where *year* = 1977

Note that this SQL statement will still need to be translated further by the DBMS so that the functions/methods within the DBMS program can not only process the request, but do it in a timely manner.

3. THE QUERY PROCESSOR

There are three phases [12] that a query passes through during the DBMS' processing of that query:

1. Parsing and translation
2. Optimization
3. Evaluation

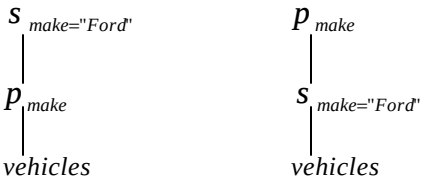
Most queries submitted to a DBMS are in a high-level language such as SQL. During the parsing and translation stage, the human readable form of the query is translated into forms usable by the DBMS. These can be in the forms of a relational algebra expression, query tree and query graph. Consider the following SQL query:

select *make*
from *vehicles*
where *make* = "Ford"

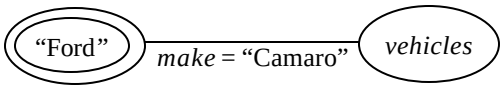
This can be translated into either of the following relational algebra expressions:

$$\sigma_{make="Ford"}(\pi_{make}(vehicles))$$
$$\pi_{make}(\sigma_{make="Ford"}(vehicles))$$

Which can also be represented as either of the following query trees:



And represented as a query graph:



After parsing and translation into a relational algebra expression, the query is then transformed into a form, usually a query tree or graph, that can be handled by the optimization engine. The optimization engine then performs various analyses on the query data, generating a number of valid evaluation plans. From there, it determines the most appropriate evaluation plan to execute.

After the evaluation plan has been selected, it is passed into the DMBS' query-execution engine [12] (also referred to as the runtime database processor [5]), where the plan is executed and the results are returned.

3.1 Parsing and Translating the Query

The first step in processing a query submitted to a DBMS is to convert the query into a form usable by the query processing engine. High-level query languages such as SQL represent a query as a string, or sequence, of characters. Certain sequences of characters represent various types of tokens such as keywords, operators, operands, literal strings, etc. Like all languages, there are rules (syntax and grammar) that govern how the tokens can be combined into understandable (i.e. valid) statements.

The primary job of the parser is to extract the tokens from the raw string of characters and translate them into the corresponding internal data elements (i.e. relational algebra operations and operands) and structures (i.e. query tree, query graph).

The last job of the parser is to verify the validity and syntax of the original query string.

3.2 Optimizing the Query

In this stage, the query processor applies rules to the internal data structures of the query to transform these structures into equivalent, but more efficient representations. The rules can be based upon mathematical models of the relational algebra expression and tree (heuristics), upon cost estimates of different algorithms applied to operations or upon the semantics within the query and the relations it involves. Selecting the proper rules to apply, when to apply them and how they are applied is the function of the query optimization engine.

3.3 Evaluating the Query

The final step in processing a query is the evaluation phase. The best evaluation plan candidate generated by the optimization engine is selected and then executed. Note that there can exist multiple methods of executing a query. Besides processing a query in a simple sequential manner, some of a query's individual operations can be processed in parallel—either as independent processes or as interdependent pipelines of processes or threads. Regardless of the method chosen, the actual results should be same.

4. QUERY METRICS: COST

The execution time of a query depends on the resources needed to perform the needed operations: disk accesses, CPU cycles, RAM and, in the case of parallel and distributed systems, thread and process communication (which will not be considered in this paper). Since data transfer to/from disks is substantially slower than memory-based transfers, the disk accesses usually represent an overwhelming majority of the total cost—particularly for very large databases that cannot be pre-loaded into memory. With today's computers, the CPU cost also can be insignificant compared to disk access for many operations.

The cost to access a disk is usually measured in terms of the number of blocks transferred from and to a disk, which will be the unit of measure referred to in the remainder of this paper.

5. THE ROLE OF INDEXES

The utilization of indexes can dramatically reduce the execution time of various operations such as select and join. Let us review some of the types of index file structures and the roles they play in reducing execution time and overhead:

Dense Index: Data-file is ordered by the search key and every search key value has a separate index record. This structure requires only a single seek to find the first occurrence of a set of contiguous records with the desired search value.

Sparse Index: Data-file is ordered by the index search key and only some of the search key values have corresponding index records. Each index record's data-file pointer points to the first data-file record with the search key value. While this structure can be less efficient (in terms of number of disk accesses) than a dense index to find the desired records, it requires less storage space and less overhead during insertion and deletion operations.

Primary Index: The data file is ordered by the attribute that is also the search key in the index file. Primary indices can be dense or sparse. This is also referred to as an Index-Sequential File [5]. For scanning through a relation's records in sequential order by a key value, this is one of the fastest and more efficient structures—locating a record has a cost of 1 seek, and the contiguous makeup of the records in sorted order minimizes the number of blocks that have to be read. However, after large numbers of insertions and deletions, the performance can degrade quite quickly, and the only way to restore the performance is to perform a reorganization.

Secondary Index: The data file is ordered by an attribute that is *different from the search key* in the index file. Secondary indices must be dense.

Multi-Level Index: An index structure consisting of 2 or more tiers of records where an upper tier's records point to associated index records of the tier below. The bottom tier's index records contain the pointers to the data-file records. Multi-level indices can be used, for instance, to reduce the number of disk block reads needed during a binary search.

Clustering Index: A two-level index structure where the records in the first level contain the clustering field value in one field and a second field pointing to a block [of 2^{nd} level records] in the second level. The records in the second level have one field that points to an actual data file record or to another 2^{nd} level block.

B^+-tree Index: Multi-level index with a balanced-tree structure. Finding a search key value in a B^+-tree is proportional to the height of the tree—maximum number of seeks required is $\lceil \lg(height) \rceil$. While this, on average, is more than a single-level, dense index that requires only one seek, the B^+-tree structure has a distinct advantage in that it does not require reorganization—it is self-optimizing because the tree is kept balanced during insertions and deletions. Many mission-critical applications require high performance with near-100% uptime, which cannot be achieved with structures requiring reorganization. The leaves of the B^+-tree are used to reorganize the data file.

6. QUERY ALGORITHMS

Queries are ultimately reduced to a number of file scan operations on the underlying physical file structures. For each relational operation, there can exist several different access paths to the particular records needed. The query execution engine can have a multitude of specialized algorithms designed to process particular relational operation and access path combinations. We will look at some examples of algorithms for both the select and join operations.

6.1 Selection Algorithms

The Select operation must search through the data files for records meeting the selection criteria. The following are some examples of simple (one attribute) selection algorithms [13]:

S1. Linear search: Every record from the file is read and compared to the selection criteria. The execution cost for searching on a non-key attribute is b_r, where b_r is the number of blocks in the file representing relation r. On a key attribute, the average cost is $b_r / 2$, with a worst case of b_r.

S2. Binary search on primary key: A binary search, on equality, performed on a primary key attribute (file ordered by the key) has a worst-case cost of $\lceil \lg(b_r) \rceil$. This can be significantly more efficient than the linear search, particularly for a large number of records.

S3. Search using a primary index on equality: With a B^+-tree index, an equality comparison on a key attribute will have a worst-case cost of the height of the tree (in the index file) plus one to retrieve the record from the data file. An equality comparison on a non-key attribute will be the same except that multiple records may meet the condition, in which case, we add the number of blocks containing the records to the cost.

S4. Search using a primary index on comparison: When the comparison operators ($<, \leq, >, \geq$) are used to retrieve multiple records from a file sorted by the search attribute, the first record satisfying the condition is located and the total blocks before ($<, \leq$) or after ($>, \geq$) is added to the cost of locating the first record.

S5. Search using a secondary index on equality: Retrieve one record with an equality comparison on a key attribute; or retrieve a set of records on a non-key attribute. For a single record, the cost will be equal to the cost of locating the search key in the index file plus one for retrieving the data record. For multiple records, the cost will be equal to the cost of locating the search key in the index file plus one block access for each data record retrieval, since the data file is not ordered on the search attribute.

6.2 Join Algorithms

Like selection, the join operation can be implemented in a variety of ways. In terms of disk accesses, the join operations can be very expensive, so implementing and utilizing efficient join algorithms is critical in minimizing a query's execution time. The following are 4 well-known types of join algorithms:

J1. Nested-Loop Join: This algorithm consists of a inner for loop nested within an outer for loop. To illustrate this algorithm, we will use the following notations:

r, s	Relations r and s
t_r	Tuple (record) in relation r
t_s	Tuple (record) in relation s
n_r	Number of records in relation r
n_s	Number of records in relation s
b_r	Number of blocks with records in relation r
b_s	Number of blocks with records in relation s

Here is a sample pseudo-code listing for joining the two relations r and s utilizing the nested-for loop [12]:

```
for each tuple t_r in r
    for each tuple t_s in s
        if join condition is true for (t_r, t_r)
            add t_r+t_s to the result
```

Each record in the outer relation r is scanned once, and each record in the inner relation s is scanned n_r. times, resulting in $n_r * n_s$ total record scans. If only one block of each relation can fit into memory, then the cost (number of block accesses) is $n_r * b_s + b_r$ [12]. If all blocks in both relations can fit into memory, then the cost is $b_r + b_s$ [12]. If all of the blocks in relation s (the inner relation) can fit into memory, then the cost is identical to both relations fitting in memory: $b_r + b_s$ [12]. Thus, if one of the relations can fit entirely in memory, then it is advantageous for the query optimizer to select that relation as the inner one.

Even though the worst case for the nested-loop join is quite expensive, it has an advantage in that it does not impose any restrictions on the access paths for either relation, regardless of the join condition.

J2. Index Nested-Loop Join: This algorithm is the same as the Nested-Loop Join, except an index file on the inner relation's (s) join attribute is used versus a data-file scan on s—each index lookup in the inner loop is essentially an equality selection

on *s* utilizing one of the selection algorithms (ex. S2, S3, S5). Let *c* be the cost for the lookup, then the worst-case cost for joining *r* and *s* is $b_r + n_r * c$ [12].

J3. Sort-Merge Join: This algorithm can be used to perform natural joins and equi-joins and requires that each relation (*r* and *s*) be sorted by the common attributes between them ($R \cap S$) [12]. The details for how this algorithm works can be found in [5] and [12] and will not be presented here. However, it is notable to point out that each record in *r* and *s* is only scanned *once*, thus producing a worst and best-case cost of $b_r + b_s$ [12]. Variations of the Sort-Merge Join algorithm are used, for instance, when the data files are in un-sorted order, but there exist secondary indices for the two relations.

J4. Hash Join: Like the sort-merge join, the hash join algorithm can be used to perform natural joins and equi-joins [12]. The hash join utilizes two hash table file structures (one for each relation) to partition each relation's records into sets containing identical hash values on the join attributes. Each relation is scanned and its corresponding hash table on the join attribute values is built. Note that collisions may occur, resulting in some of the partitions containing different sets records with matching join attribute values. After the two hash tables are built, for each matching partition in the hash tables, an in-memory hash index of the smaller relation's (the build relation) records is built and a nested-loop join is performed against the corresponding records in the other relation, writing out to the result for each join.

Note that the above works only if the required amount of memory is available to hold the hash index and the number records in any partition of the build relation. If not, then a process known as *recursive partitioning* is performed—see [5] or [12] for details.

The cost for the hash join, without recursive partitioning, is $3(b_r + b_s) + 4n_h$ where n_h is the number of partitions in the hash table [12]. The cost for the hash join with recursive partitioning is $2(b_r + b_s)\lceil \log_{M-1}(b_s) - 1 \rceil + b_r + b_s$ where *M* is the number of memory blocks used.

7. QUERY OPTIMIZATION

The function of a DBMS' query optimization engine is to find an evaluation plan that reduces the overall execution cost of a query. We have seen in the previous sections that the costs for performing particular operations such as select and join can vary quite dramatically. As an example, consider 2 relations *r* and *s*, with the following characteristics:

10,000 = n_r = Number of tuples in *r*
1,000 = n_s = Number of tuples in *s*
1,000 = b_r = Number of blocks with tuples in *r*
100 = b_s = Number of blocks with tuples in *s*

Selecting a single record from *r* on a non-key attribute can have, a cost of $\lceil \lg(b_r) \rceil = 10$ (binary search) or a cost of $b_r / 2 = 5{,}000$ (linear search). Joining *r* and *s* can have a cost of $n_r * b_s + b_r = 1{,}001{,}000$ (nested-loop join)[13] or a cost of $3(b_r + b_s) + 4n_h = 73{,}000$ (hash-join where n_h = 10,000)[13].

Notice that the cost difference between the 2 selects differs by a factor of 500, and the 2 joins by a factor of ~14. Clearly, selecting lower-cost methods can result in substantially better performance. This process of selecting a lower-cost mechanism is known as cost-based optimization. Other strategies for lowering the execution time of queries include heuristic-based optimization and semantic-based optimization.

In heuristic-based optimization, mathematical rules are applied to the components of the query to generate an evaluation plan that, theoretically, will result in a lower execution time. Typically, these components are the data elements within an internal data structure, such as a query tree, that the query parser has generated from a higher level representation of the query (i.e. SQL).

The internal nodes of a query tree represent specific relational algebra operations to be performed on the relation(s) passed into them from the child node(s) directly below. The leaves of the tree are the relation(s). The tree is evaluated from the bottom up, creating a specific evaluation plan. In section 3, we saw that a query's query tree can be constructed in multiple, equivalent ways. In many instances, there will be at least one of these equivalent trees that produces a faster, "optimized" execution plan. Section 7.2 will illustrate this concept.

Another way of optimizing a query is semantic-based query optimization. In many cases, the data within and between relations contain "rules" and patterns that are based upon "real-world" situations that the DBMS does not "know" about. For example, vehicles like the Delorean were not made after 1990, so a query like "Retrieve all vehicles with make equal to Delorean and year > 2000" will produce zero records. Injecting these types of semantic rules into a DBMS can thus further enhance a query's execution time.

7.1 Statistics of Expression Results

In order to estimate the various costs of query operations, the query optimizer utilizes a fairly extensive amount of metadata associated with the relations and their corresponding file structures. These data are collected during and after various database operations (such as queries) and stored in the DBMS catalog. These data include [5, 12]:

- n_r Number of records (tuples) in a relation *r*. Knowing the number of records in a relation is a critical piece of data utilized in nearly all cost estimations of operations.
- f_r Blocking factor (number of records per block) for relation *r*. This data is used in calculating the blocking factor, and is also useful in determining the proper size and number of memory buffers.
- b_r Number of blocks in relation *r*'s data-file. Also a critical and commonly used datum, b_r is calculated value equal to n_r / b_r.
- l_r Length of a record, in bytes, in relation *r*. The record size is another important data item used in many operations, particularly when the values differ significantly for two relations involved in an operation. For variable-length records, the actual length value used—either the average or the maximum—depends on the type of operation to be performed.
- d_{Ar} Number of distinct values of attribute *A* in relation *r*. This value is important in calculating the number of resulting records for a projection operation and for aggregate functions like **sum**, **count** and **average**.
- x Number of levels in a multi-level index (B^+-tree, cluster index, etc.). This data item is used in estimating the number of block accesses needed in various search algorithms. Note that for a B^+-tree, *x* will be equal to the height of the tree.
- s_A Selection cardinality of an attribute. This is a calculated value equal to n_r / d_{Ar}. When *A* is a key attribute, $s_A = 1$. The selection cardinality allows the query optimizer to determine the "average number of records that will satisfy an equality selection condition on that attribute"[5].

The query optimizer also depends on other important data such as the ordering of the data file, the type of index structures available and the attributes involved in these file organization structures. Knowing whether certain access structures exist allows the query optimizer to select the appropriate algorithm(s) to use for particular operations.

7.2 Expression and Tree Transformations

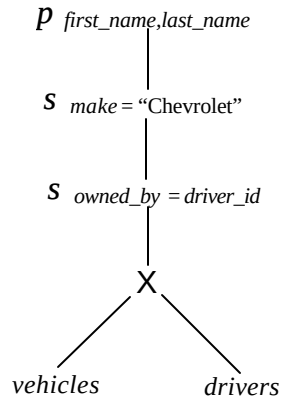
After a high-level query (i.e. SQL statement) has been parsed into an equivalent relational algebra expression, the query optimizer can perform heuristic rules on the expression and tree to transform the expression and tree into equivalent, but optimized forms. As an example, consider the following SQL query:

select *first_name, last_name*
from *drivers, vehicles*
where *make* = "Chevrolet" and *owned_by* = *driver_id*

A corresponding relational algebra expression is:

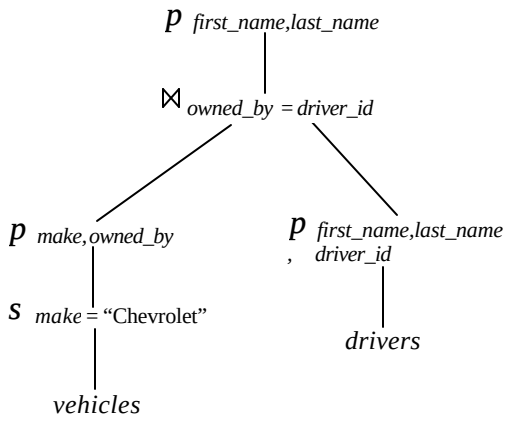
$\pi_{first\_name,last\_name}((\sigma_{make = \text{"Chevrolet"}}(\sigma_{owned\_by = driver\_id}(vehicles \times drivers)))$

And the corresponding canonical query tree for the relational algebra expression:



Suppose the *vehicles* and *drivers* relations both have 10,000 records each and the number of Chevrolet vehicles is 5,000. Note that the Cartesian product resulting in 10,000,000 records can be reduced by 50% if the $\sigma_{make = \text{"Chevrolet"}}$ operation is performed first. We can also combine the $\sigma_{owned\_by = driver\_id}$ and Cartesian product operations into a more efficient join operation,

as well as eliminating any unneeded columns before the expensive join is performed. The diagram below shows this better, "optimized" version of the tree:



In relational algebra, there are several definitions and theorems the query optimizer can use to transform the query. For instance, the definition of equivalent relations states that the set of attributes (domain) of each relation must be the same—because they are sets, the order does not matter. Here is a partial list of relational algebra theorems from the Elmasri/Navathe textbook [5]:

1. Cascade of σ : A select with conjunctive conditions on the attribute list is equivalent to a *cascade* of selects upon selects: $\sigma_{A_1 \wedge A_2 \wedge \ldots \wedge A_n}(R) \equiv \sigma_{A_1}(\sigma_{A_2}(\ldots(\sigma_{A_n}(R))\ldots))$

2. Commutativity of σ : The select operation is commutative: $\sigma_{A_1}(\sigma_{A_2}(R)) \equiv \sigma_{A_2}(\sigma_{A_1}(R))$

3. Cascade of π : A *cascade* of project operations is equivalent to the last project operation of the cascade: $\pi_{AList_1}(\pi_{AList_2}(\ldots(\pi_{AList_n}(R))\ldots)) \equiv \pi_{AList_1}(R)$

4. Commuting σ with π : Given a π 's and σ 's attribute list of A_1, A_2,…, A_n, the π and σ operations can be commuted: $\pi_{A_1,A_2,\ldots,A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1,A_2,\ldots,A_n}(R))$

5. Communativity of $\bowtie$ or X: The join and Cartesian product operations are communative: $R \bowtie S \equiv S \bowtie R$ and $R \times S \equiv S \times R$

6. Commuting σ with $\bowtie$ or X: Select can be commuted with join (or Cartesian product) as follows:
   a. If all of the attributes in the select's condition are in relation R then $\sigma_c(R \bowtie S) \equiv (\sigma_c(R)) \bowtie S$
   b. Given select the condition c composed of conditions c1 and c2, and c1 contains only attributes from R, and c2 contains only attributes from S, then $\sigma_c(R \bowtie S) \equiv (\sigma_{c1}(R)) \bowtie (\sigma_{c2}(S))$

7. Commutativity of set operations $(\cup, \cap, -)$: Union and intersection operations are commutative; but the difference operation is not: $R \cup S \equiv S \cup R$, $R \cap S \equiv S \cap R$, $R - S \neq S - R$

8. Associativity of $\bowtie$, X, $\cup$ and $\cap$: All four of these operations are individually associative. Let θ be any *one* of these operators, then: $(R \theta S) \theta T \equiv R \theta (S \theta T)$

9. Commuting σ with set operations $(\cup, \cap, -)$:Let θ be any *one* of the three set operators, then: $\sigma_c(R \theta S) \equiv (\sigma_c(R)) \theta (\sigma_c(S))$

10. Commuting π with $\cup$: Project and union operations can be commuted: $\pi_{A_{List}}(R \cup S) \equiv (\pi_{A_{List}}(R)) \cup (\pi_{A_{List}}(S))$

Using these theorems, an algorithm can be defined to transform the original query expression/tree created by the parser into a more optimized query. A detailed example of such an algorithm can be found in the Elmasri/Navathe textbook [5]—some of the key concepts can be summarized as follows:

1. One primary objective is to reduce the size of the intermediate relations, both in terms of bytes per record as well as number of records, as soon as possible so that subsequent operations will have less data to process and thus execute quicker.

2. Operations, such as conjunctive selections, should be broken down into their equivalent set of smaller units to allow the individual units to be moved into "better" positions within the query tree.

3. Combine Cartesian products with corresponding selects to create joins—utilizing optimized join algorithms like the sort-merge join and hash join can be orders of magnitude more efficient.

4. Move selects and projects as far down the tree as possible, as these operations will produce smaller intermediate relations that can be processed more quickly by the operations above.

7.3 Choice of Evaluation Plans

The query optimization engine typically generates a set of candidate evaluation plans. Some will, in heuristic theory, produce a faster, more efficient execution. Others may, by prior historical results, be more efficient than the theoretical models—this can very well be the case for queries dependent on the semantic nature of the data to be processed. Still others can be more efficient due to "outside agencies" such as network congestion, competing applications on the same CPU, etc. Thus, a plethora of data can exist from which the query execution engine can probe for the best evaluation plan to execute at any given time.

10. CONCLUSION

One of the most critical functional requirements of a DBMS is its ability to process queries in a timely manner. This is particularly true for very large, mission critical applications such as weather forecasting, banking systems and aeronautical applications, which can contain millions and even trillions of records. The need for faster and faster, "immediate" results never ceases. Thus, a great deal of research and resources is spent on creating smarter, highly efficient query optimization engines. Some of the basic techniques of query processing and optimization have been presented in this paper. Other, more advanced topics are the subjects of many research papers and projects. Some examples include XML query processing [3, 11], query containment [2], utilizing materialized views [13], sequence queries [9, 10] and many others.

11. EXTENDING MY MINI-DB ENGINE

My primary goal in enhancing my "mini" database engine application is to speed up the processing of queries. Before we get in the details behind the implementation plan for accomplishing this goal, let's look at what data structures, file structures and algorithms are currently in place. Only the most significant ones will be discussed.

11.1 Current Implementation

File Structures

- Record-number-ordered index. This index is "low-level" and is not "seen" by the relational algebra methods (i.e. select, project, etc.). It's purpose is to provide a primary access mechanism to the data-file records. It is also the "owner" of the state of each data-file record (i.e. active or deleted).
- Hash-based, unordered, single-level, secondary index. This index structure provides single-seek access to data-file records. Because it is unordered, only equality-based comparisons can be utilized in locating records.
- Sequential, unordered data file. Due to the fact that the current indices are unordered, this file will not be able to be put into a physically sorted format, even after a reorganization.
- Meta-Data file: This file stores all of the information, in standard XML-formatted text, pertaining to its corresponding relation. This includes: table name; number of fields (attributes); list of fields with their name, size and type; primary key field identifier; foreign key identifiers; list of indices with the index name, index field name and type of index (unique/key or clustered/secondary). Also, the access algorithm for this file is flexible enough to handle any additional data (singular or nested).

Data Structures

- Each of the file structures has a corresponding class with methods that handle the file access (open, close, rename, etc.) as well as the reading, updating, inserting and deleting of records.
- Wrapping around the file structure classes is the DBRelation class with methods that handle the creation, opening and updating of the associated files. This class also has high-level methods one would associate with single-relation operations such as: insert, delete, update and search.
- Wrapping around the DBRelation class is the Mini_Rel_Algebra class with methods that perform the following relational algebra operations: select, project, Cartesian product, union, intersection difference and join.

Algorithms

- DBRelation.Search() This method performs the actual search for record(s) in a given relation. It can accept multiple search fields, conditions and search values for performing the equivalent of a conjunctive select. The search values can be either constant string values or references to a particular field's value. The search algorithm itself can operate in two ways: 1. Linear search,

where every record in the relation's data file is scanned and compared to the condition. 2. Index lookup, by equality, on an attribute in the condition list. The index lookup is essentially equivalent to the S5 search algorithm detailed in section 6.1. Note that if the record is found using the index, then the remaining search conditions, if any, are evaluated for the current record. If the record is not found, then the condition with the indexed attribute is false, and the remaining conditions do not have to be evaluated.

- Mini_Rel_Algebra.Join() The current implementation of the join operation is performed by executing a select operation followed by a Cartesian product operation.

11.2 Proposed Enhancements

There are 6 main query execution speed enhancements I plan on implementing:

1. Implement a "record generator" so that a large number (>100,000) of records can be populated into the database. This will allow performance comparisons to be made between the current and new implementations.
2. Replace the current join algorithm with the J4 hash join algorithm discussed in section 6.2. I expect a very significant boost in performance, and, so that speed comparisons can be made, I will keep the old join method (rename it to OldJoin). The syntax of the new Join() call will be the same:

   Join(**string** *relationName1*, **string** *relationName2*, **string** *joinField1*, **string** *joinField2*)
3. Create a new HJIndex class to handle the hash table file structure that will be needed by the new hash join algorithm. Since most all of the current DBIndex class' existing access methods (add, delete, modify) will most likely not need any changes (even if they do, the changes will be minor), the HJIndex will be inherited from the DBIndex class.
4. Finish the implementation of the cluster index. This new index structure will allow multiple records to be retrieved utilizing the concepts of the S5 search algorithm in section 6.1.
5. Modify the DBRelation.Search() method to utilize the new cluster index.
6. Create a new DBQuery class. This will allow the user to build and execute a query consisting of a sequence of relational operations. This class will be a simplified version in that it will only handle a sequential list of operations. If time permits, I may create a "true" query tree structure. The following instance variables and methods will be implemented:

   ArrayList opList : Instance variable holding the sequential list of relational operations.

   DBOperation opObj : Object holding a relational operation. DBOperation will either be a structure or class that holds all of the possible parameters involved in the various types of relational operations.

   DBQuery(string queryName) The constructor method.

   bool AddOp(string opName, string param1, string param2, …) Adds an operation object to the opList array. Will be overloaded to handle the various parameter lists of the different relational operations. For instance, a project operation needs three parameters: string opName, string relationName, string attributeList.

   void Clear() Clears the current query—opList.Clear()

   string Execute() Executes the current query. The returned string will be the name of the resulting relation.

   string ToString() Returns a multi-line string of the current list of operations and their parameters.

11.3 Query Performance Comparisons

Time permitting, I will build a test database consisting of several thousand records. This test database will then be used to time the execution speeds of "identical" queries in the existing and new version of the Mini DBEngine application. The test results will be compiled into a comparison table and included in the report for the final version of the application.

REFERENCES

[1] Henk Ernst Blok, Djoerd Hiemstra and Sunil Choenni, Franciska de Jong, Henk M. Blanken and Peter M.G. Apers. Predicting the cost-quality trade-off for information retrieval queries: Facilitatiing database design and query optimization. *Proceedings of the tenth international conference on Information and knowledge management*, October 2001, Pages 207-214.

[2] D. Calvanese, G. De Giacomo, M. Lenzerini and M. Y. Vardi. Reasoning on Regular Path Queries. *ACM SIGMOD Record*, Vol. 32, No. 4, December 2003.

[3] Andrew Eisenberg and Jim Melton. Advancements in SQL/XML. *ACM SIGMOD Record*, Vol. 33, No. 3, September 2004.

[4] Andrew Eisenberg and Jim Melton. An Early Look at XQuery API for Java™ (XQJ). *ACM SIGMOD Record*, Vol. 33, No. 2, June 2004.

[5] Ramez Elmasri and Shamkant B. Navathe. Fundamentals of Database Systems, second edition. Addison-Wesley Publishing Company, 1994.

[6] Donald Kossmann and Konrad Stocker. Iterative Dynamic Programming: A new Class of Query Optimization Algorithms. *ACM Transactions on Database Systems*, Vol. 25, No. 1, March 2000, Pages 43-82.

[7] Chiang Lee, Chi-Sheng Shih and Yaw-Huei Chen. A Graph-theoritic model for optimizing queries involving methods. *The VLDB Journal — The International Journal on Very Large Data Bases*, Vol. 9, Issue 4, April 2001, Pages 327-343.

[8] Hsiao-Fei Liu, Ya-Hui Chang and Kun-Mao Chao. An Optimal Algorithm for Querying Tree Structures and its Applications in Bioinformatics. *ACM SIGMOD Record* Vol. 33, No. 2, June 2004.

[9] Reza Sadri, Carlo Zaniolo, Amir Zarkesh and Jafar Adibi. Expressing and Optimizing Sequence Queries in Database Systems. *ACM Transactions on Database Systems*, Vol. 29, Issue 2, June 2004, Pages 282-318.

[10] Reza Sadri, Carlo Zaniolo, Amir Zarkesh and Jafar Adibi. Optimization of Sequence Queries in Database Systems. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, May 2001, Pages 71-81.

[11] Thomas Schwentick. XPath Query Containment. *ACM SIGMOD Record*, Vol. 33, No. 1, March 2004.

[12] Avi Silbershatz, Hank Korth and S. Sudarshan. *Database System Concepts*, 4th Edition. McGraw-Hill, 2002.

[13] Dimitri Theodoratos and Wugang Xu. Constructing Search Spaces for Materialized View Selection. *Proceedings of the 7th ACM international workshop on Data warehousing and OLAP*, November 2004, Pages 112-121.

[14] Jingren Zhou and Kenneth A. Ross. Buffering Database Operations for Enhanced Instruction Cache Performance. *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, June 2004, Pages 191-202.