MULTI-QUERY OPTIMIZATION IN THE DATAPATH SYSTEM

By

NIKETAN R. PANSARE

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2009

© 2009 Niketan R. Pansare

To my family, friends and professors

ACKNOWLEDGMENTS

Thanks go out to Christopher Jermaine, Alin Dobra, Subi Arumugam, Ravi Jampani and Luis Perez.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS 4

LIST OF TABLES 7

LIST OF FIGURES 8

ABSTRACT 9

CHAPTER

1 INTRODUCTION 10

1.1 Compute-Centric System 10
1.2 Data-Centric System 10
1.3 Compute-Centric Versus Data-Centric 11
1.4 Problem Statement 14

2 RELATED WORK 17

3 OVERVIEW 19

4 DESIGN 22

4.1 The Network Integrator Class 22
4.2 The Enumerator Method 22
4.3 The Cost Function 23
4.4 The Search Strategy 23

5 IMPLEMENTATION 26

5.1 Types 26
5.2 Enumeration 27
5.3 Mapping Rules 28
5.4 Bypassable Rules 29
5.5 The Cost Function 31
5.6 The Search Strategy 33

6 EXPERIMENTAL RESULTS 35

6.1 Goal 35
6.2 Setup 35
6.3 Experimental Results 37
6.4 Analysis 37

7 CONCLUSION 45

8 FUTURE WORK 46

REFERENCES 47
BIOGRAPHICAL SKETCH 49

LIST OF TABLES

Table Page

5-1 Selectivity Factor 33

6-1 Cost and Time taken by each algorithm 39

LIST OF FIGURES

Figure Page

1-1 Query plans for traditional databases 12

1-2 Path Network after query 1 13

1-3 Path Network after query 2 13

1-4 Path Network after query 3 14

5-1 Path network before bypassing 29

5-2 Final path network with bypassing 30

5-3 Final path network without bypassing 30

5-4 Example path network for bypassing 30

6-1 Framework for testing different query optimization techniques 36

6-2 Comparison of FIFO with other selectors 37

6-3 Comparison of Random selector with the cost based selectors 38

6-4 Comparison of the cost based selectors 38

6-5 Comparison of the the average time taken by the selectors 40

6-6 Path network after query 11 42

6-7 Path network after query 5 for waypoint-count selector 43

6-8 Path network after query 5 for cost-based selector 43

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

MULTI-QUERY OPTIMIZATION IN THE DATAPATH SYSTEM

By

Niketan R. Pansare

December 2009

Chair: Christopher Jermaine
Major: Computer Engineering

The Datapath system is a novel database that is implemented from the ground-up using a data-centric approach. In this thesis, I describe and evaluate a multi-query optimizer for the Datapath system. Unlike traditional multi-query optimizers that only try to overlap common sub-expressions, I propose an efficient optimization algorithm that minimizes the data (or the overall number of tuples) flowing through the system. Using this objective function, a qualitative and quantitative study is presented comparing the commonly used algorithms against the proposed multi-query optimization algorithm.

CHAPTER 1
INTRODUCTION

1.1 Compute-Centric System

Most computer systems, including databases, are compute-centric. The data is brought onto the processor through the memory hierarchy as required by the computations. For example, consider a computation `ADD A, B`. In typical computer system, the control element of the program (usually the loader) will load the computation on the processor and then figure out that it requires `A` and `B` for the computation. If `A` and `B` are not in the cache or the main memory, the control element will fetch them from the disk and load it onto the cache. Furthermore, if `A` and `B` are not stored in the same memory page, there could be additional overhead in the disk access. Though this model seems natural for most computer systems (for example, scientific and commercial applications), it does not fit well for databases. There are several reasons for this. First, the data access pattern for computations in databases is not uniform. Compared to the databases, most scientific and commercial application are able to utilize locality of the data much more efficiently. Though several algorithms [17, 19, 23] are suggested to improve the memory performance of the databases, but they can only perform as good as the data expected by the computations. This emphasis on the computation is ill-suited for the databases due to large amount of data they handle. Second, the gap between the time taken to push the data through memory hierarchy and the time taken to perform the computation on it has been increasing in the past years. This problem has been further aggravated with the advent of multi-core processors and hence transferring the data through memory hierarchy is becoming a bottleneck for the modern databases [2]. In spite of this bottleneck, computations still drive the data in current database systems.

1.2 Data-Centric System

The Datapath system is a prototype system which uses *Data-centric approach* for analytic query processing. To understand Data-centric approach, let us compare the

Datapath system to the water pipe system. Imagine the tables of the database as an active entity, like tap or some water source that keeps on generating the data, until it is turned off. This data moves through the memory hierarchy, from disk to the cache, onto the processor. We assign different cores of the processor to different relational operators or computation units called as *waypoints*. The waypoints act as a valve, which filters or merges different data flows and outputs them to other waypoints, until the result is generated. It is important to note that waypoints act only as computation units and have no control over the data they receive. Traditional database systems, however, determine which data is required based on the computation, which is then retrieved by using an access method [21] suggested by the query processor. To put it simply, in Data-centric system, data drives the computation; while in traditional database systems, computation drives the data.

1.3 Compute-Centric Versus Data-Centric

Let us consider a simple example to explain the difference between compute-centric and data-centric design for databases. Consider these three queries are issued by users of the database.

Query 1:

```
select * from nation, supplier
where n_nationkey = s_nationkey and s_acctbal > 10000
```

Query 2:

```
select * from nation, customer
where n_nationkey = c_nationkey and c_acctbal > 1000
```

Query 3:

```
select * from supplier, partsupp
where s_suppkey = ps_suppkey and ps_availqty < 500
```

Traditional databases will first find out what are the computations necessary to evaluate these queries. For example, the first query will have at least two computations

namely, *Selection* on the `supplier` table and *Join* on `nation` and `supplier`. These computations are represented as nodes (or the operators) in the query plan. The query optimizer for the traditional databases then tries to optimize these computations to produce an optimized query plan. This plan is physically realized by having one *GetNext* method for each input to the node of the query plan. The GetNext method depending upon the computation it is associated with, decides what data it should retrieve and also how that data should be retrieved. In other words, the query plans generated by the traditional databases are compute-centric.

Usually traditional databases will produce three separate query plans (see 1-1) for these three queries. This is true even for most multi-query optimizers, since these queries have no common sub-expressions [22]. The figure 1-1 ignores the physical operators such as index, sort, etc.

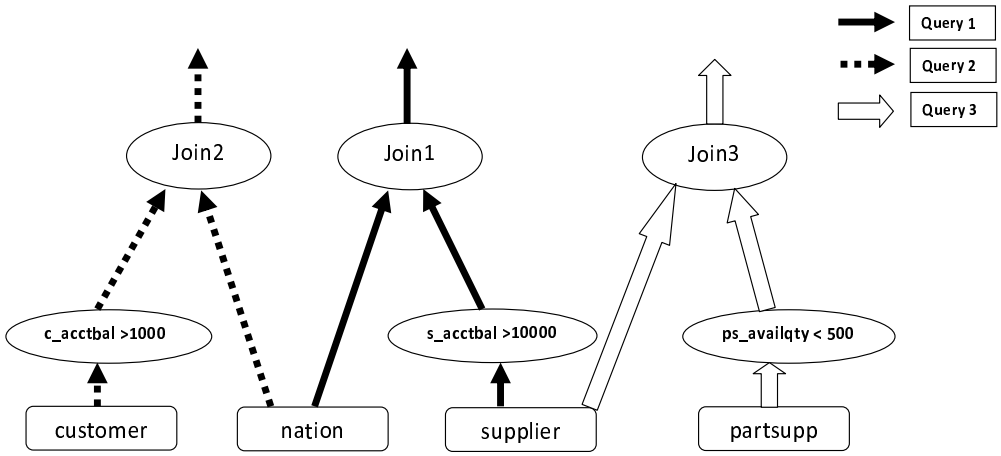

Figure 1-1. Query plans for traditional databases

The `Join1` operator will have the code to fetch the data (*GetNext method*) and also to perform the computation on it. This means that the same data (from the nation table) is brought twice onto the cache; first for `Join1` and then for `Join2`. Though some databases try to alleviate this problem by using multi-query optimizers and materialized views, it does not solve the problem.

The Datapath system has one plan for all the queries running in the system for maximal reuse of the data. This plan is called as the *path network* and is detailed enough to allow the code to be generated and executed by the execution engine. The path network is optimized to minimize the data paths and not the computations. The waypoints have no control over the data they receive and hence have no analogous GetNext method.

Assume that `query 1` is the first query and initial path network is empty. The query plan for `query 1` will form the new path network as shown in the figure 1-2.

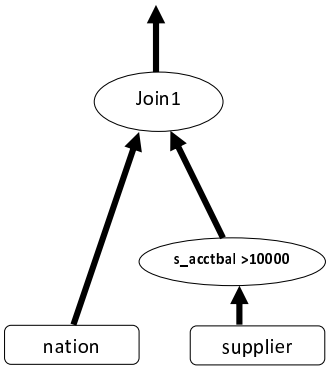

Figure 1-2. Path Network after query 1

The path manager will now try to overlap `query 2` to reduce the flow of data in the system. The figure 1-3 shows the new path network, where `Join1` and `Join2` are merged together to form `Join1-2`.

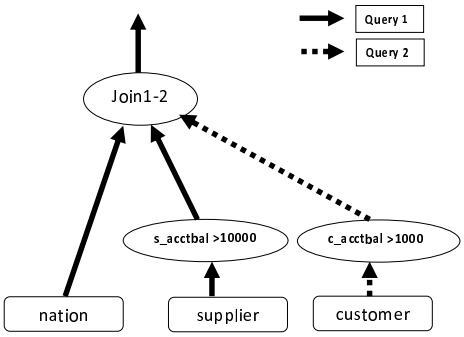

Figure 1-3. Path Network after query 2

The figure 1-4 shows a path network after integrating `query 3` into the existing path network shown in the figure 1-3. Notice that the selection waypoint of `query 1` acts as a bypass waypoint[1] for `query 3`.

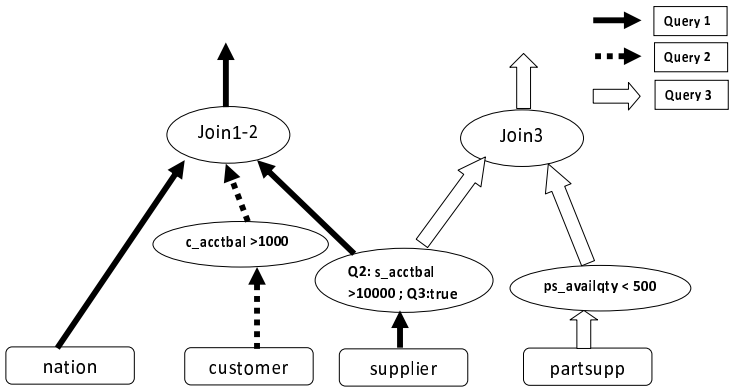

Figure 1-4. Path Network after query 3

The plan generated by traditional database 1-1 has more data paths. This means that there is much more data being transferred than in the path network. Clearly, most traditional query optimizers are not an ideal choice for the Datapath system.

Multi-query optimizers(MQO) try to alleviate this problem by sharing the result of common sub-expressions between queries [22]. The constraints on the type of data in the Datapath system are more relaxed than what most MQO assume, hence making the problem a little different from multi-query optimization. Also, multi-query optimizers like traditional query optimizers focus on optimizing the computations, while ignoring the data paths. Therefore, traditional multi-query optimizers are also not suitable for the Datapath system.

1.4 Problem Statement

The previous section explained the differences in traditional query optimization techniques and the data-centric query optimization techniques. It alse pointed out that

[1] A bypass waypoint simply forwards the data without performing any computation on the data.

latter and not the former is suitable for the Datapath system. Before discussing it further, let us define the problem of *Data-centric query optimization*:

Given an input query(Q_{n+1}) and a path network (with queries Q_i, where i = 1 to n), create a new path network (with queries Q_j, where j = 1 to n + 1) such that execution time (or the response time) of queries Q_j is minimum.

This means that the goal is to improve the overall response time of the system and not just the input query. The two intuitive approaches to solve this problem are:

1. *Create a new problem specific algorithm.* For example, create a new data-centric MQO algorithm that uses all the features supported by the Datapath system for optimization.

2. *Use a previously solved problem and transform it into your problem*: This means that we use existing query optimizers to first find an optimal query plan for the input query and then try to merge it onto the existing path network. Since the existing query optimizers has no knowledge of the path network, it will use a local optimization function which may improve the execution time of that query, but not of all the queries in the system. Hence, this is not an ideal choice for implementing the Query Planner.

Since the second approach is not feasible, I use the first approach for this thesis. Let us now discuss, how two queries share a waypoint. Every query contains one or more predicates. These predicates can be either selection, join or top [2] predicates. When we say two queries share a waypoint, it means that one or more predicates of the queries are mapped onto the same waypoint. For a predicate to be mapped onto the same waypoint, it has to satisfy two properties:

1. *Two predicates should be of the same type.* This means that a selection predicate cannot be mapped onto a waypoint with join or group-by predicate.

2. *Both predicates should work on same type of data.* There are various rules, which determine whether two predicates work on same type of data or not. I will discuss these rules later in the thesis. Since the Datapath system is expected to evolve and

[2] The predicates that is not a selection or join predicates qualifies as a top predicate. For example, group-by, projection, order-by, etc are considered as the top predicates.

include more complex queries, these rules are also expected to change over a period of time. Hence, the facility to include new rules and modify existing rules is an important requirement for the Query Planner.

In this thesis, I propose a framework that is specific to the Datapath system, but generic enough to test different strategies used in existing query optimization algorithms. This framework is modularized into four main components: namely *Enumerator*, *Search*, *Coster* and *Mapping rules*. Using this kind of modularization, we test and compare different ways to implement each module. This framework also allows us to incorporate new rules for mapping in the Path network without modifying significant amount of code.

Using the above framework, we propose a solution that would try to minimize the response time of the query. Since data-centric focuses on the data and not the computation, it is obvious that the proposed query optimizer also focuses on the data. It does this in two ways. First, the optimization function in the Coster component is *to minimize the flow of data through the system*. Second, the problem is presented in form of path network, which makes the mapping easy and intuitive. Also the design of input data structures (which will be discussed later) helps to seperate different aspects of query optimization and hence are useful for the framework. It is important to note that the problem of query optimization is NP-hard [13] and hence exhaustive solution is not feasible. For simplest case where there are no queries in the system, our problem becomes a traditional query optimization problem. Hence, we use the strategy that limits the search space by performing a look-ahead search rather than exhaustive search in the Search component. This will be discussed in depth in the chapter 5.

CHAPTER 2
RELATED WORK

Selinger et al. [21] laid the foundation for optimizing single queries in the database system. Most query optimizers use a cost model to search through the search space determined by their search strategies. Various search strategies have been proposed for single query optimization [4, 7, 10, 14–16, 27]. Moreover, different query optimization schemes were proposed to achieve different optimization goals, namely minimizing response time of the input query, minimizing the memory usage, maximizing the throughput of the system, etc. Most single query optimizers focus on trying to minimize the response time of input query, whereas multi-query optimizers [22] try to improve the throughput of the system. Instead of optimizing each query independently, multi-query optimizers try to optimize the global query plan that represents all the queries in the system to exploit common sub-expressions in multiple queries. A multiple-query graph is generally used to represent this global query plan [3, 18]. Sellis [22] proved that multi-query optimization would lead to substantial savings over single query optimization. Since multi-query optimization is a NP-hard problem, Sellis [22] suggested using an A^* search directed by a heuristic function rather than an exhaustive solution. Later, this heuristic function was replaced by a more informed cost function which improved the performance of the optimizer [24]. Roy et al. [20] suggested a greedy heuristic algorithm that tried to maximize sharing by materializing some partial results on the disk. Dalvi et al. [6] extended this algorithm by using pipelining to reduce the cost of materialization. Toroslu and Cosar [26] proposed a dynamic programming scheme for multi-query optimizers.

Most multi-query optimizers try to overlap only common sub-expressions in multiple queries. Hall [11] suggested detecting common sub-expression within single query. Chen and Dunham [5] allow for partial overlap of selection predicates by leaving all projection operations to the final stages. They argue that pushing projections up is bad for nested loop join but good for hash join [9].

Most multi-query optimization techniques are not integrated with the existing query optimizers. Hall [11] suggests evaluating common sub-expression as a pre-processing step; whereas Subramanian and Venkataraman [25] suggests it as a post-processing step of traditional query optimization. This would allow the MQO techniques to be integrated with the existing query optimizers and hence provide a practical solution. Roy et al. [20] also provide a practical algorithm by modifying the Volcano search strategy [10].

Like the Datapath system, the StagedDB system focus of sharing the access to the data and not the computation. Both the systems group the computations (or the execution requests) of different queries that share the same data. The StagedDB uses the *stages* to group the computations, whereas the Datapath system uses the waypoints. So, the optimizer of the StagedDB is expected to solve the similar problem (if not the same problem) as the Path Optimizer. However, the decision of sharing the data is pushed down to the execution engine. The execution engine of the StagedDB system takes most of the decisions by monitoring each relational operators or the stages to detect an overlap. This makes sharing of the data opportunistic in the StagedDB system. As a result, the optimizer for the StagedDB is similar to traditional query optimizers [12]. Also, the level of sharing supported by the execution engine of the StagedDB system is less as compared to that of the Datapath system. The cooperative scans [28] also share the data in concurrent scans. This is analogous to sharing of the table-scans in the Datapath system. Apart from the tablescans, the cooperative scans do not support any sharing of the data. In essence, the cooperative scans only try to minimize the disk access and not the accesses to the cache. Though both cooperative scans and the StagedDB system focus to some extent on sharing of the data (rather than computations), they do not fully exploit the level of data-sharing as as compared to the Datapath system.

CHAPTER 3
OVERVIEW

The two main components on the Datapath system are the Query Planner and the execution engine. The Query Planner is a module that is responsible for generating a path network that can be used by the execution engine. It is very similar to traditional multi-query optimizers in the sense that both incorporate the new query into the global execution plan or the path network. However, they differ in the underlying optimization principle. The Query Planner is intrinsically a data-centric multi-query optimizer. As stated earlier, it tries to minimize the flow of the data in the system. In this thesis, I propose that the overall number of tuples transferred through the memory hierarchy characterizes the flow of the data in the system. However, the overall number of tuples transferred depends on various factors in the system, some of which are difficult to predict. These factors include the cache block size, the page size, the current state of the existing queries, swapping of pages by the operating system, competing processes for the memory bus and other resources, and some other optimization policies implemented by the compiler as well as the operating system. Modeling these factors for the optimization process is beyond the scope of this thesis. Hence, all the existing queries are assumed to have processed no tuples. Though this seems to be a pessimistic assumption, it makes sense in the case of batch query processing. Using this assumption and ignoring the operating system dynamics, I propose that *minimizing the number of tuples in the path network will reduce the flow of the data in the system.*

Since multi-query optimizers are designed for compute-centric databases, they try to overlap common sub-expressions. It is important to note that though overlapping common sub-expressions reduces the flow of the data in some cases, it may not be true for all cases. In a case where there is a plan with more flow but less computations and another plan with less flow but more computations, traditional multi-query optimizers will chose the former while the Query Planner will chose the latter. In addition, due to inherent design

principle, traditional multi-query optimizers do not exploit all the properties that data centric databases can offer. Most practical multi-query optimizers and also the Query Planner do not use an exhaustive approach, so it is difficult to prove that the Query Planner will always perform either better or same as the existing multi-query optimizer. However, I have created a simple compute-centric cost function that tries to minimize the computations but which still expects a data-centric execution engine. As a part of my thesis , I prove that my cost function outperforms the compute-centric cost function. This is discussed in more detail in the experimental results section (See 6).

The Query Planner consist of three main components, namely the Parser, the Path Optimizer and the Translator. The Parser gets a SQL query and performs type-checking and other validations. If the query is valid, it forwards the query to the Path Optimizer. The Path Optimizer first transforms the query into a graph called as *query description*. The query description contains no information about the ordering of joins. As discussed earlier, the path network is a graph that represents the overall execution plan of all the queries in the system or the global query plan. The Path Optimizer then tries to integrate the query description onto the path network. It does so incrementally by considering one predicate at a time from the query description and trying to integrate it onto the path network. It is important to note that this integration is non-destructive. This means that the edges in the existing path network are not modified. The details of this algorithm will be discussed later. The Path Optimizer uses an object called the *network integrator* to maintain the state of the algorithm. The network integrator object contains a partially integrated path network and a partial query description. The final state of the Path Optimizer is a network integrator object that contains a fully integrated path network and an empty query description.

To summarize:

1. Data-centric query optimization is *different* than compute-centric query optimization.
2. This thesis uses a data-centric approach to multi-query optimization.

3. The goal of the Path Optimizer is to minimize the number of tuples in the path network.

4. The proposed algorithm (which will be discussed in depth later) is incremental, non-destructive, non-exhaustive and modular (to seperate different aspects of query optimization).

CHAPTER 4
DESIGN

4.1 The Network Integrator Class

The design and implementation of the algorithm for adding a new query to the existing path network relies fundamentally on a class called the "NetworkIntegrator" class. The constructor for this class takes as input two objects:

1. The existing path network
2. A representation of the new query that is to be integrated into the network (Query description)

The job of this class is to integrate the new query into the path network. However, for reasons that I will discuss subsequently, this class does not encode any notion of "search". In fact, it is quite unintelligent. All this class does is to provide the machinery necessary to integrate the query into the network: the class does not guide the integration in any way. That is done via an external algorithm that makes use of the class.

The NetworkIntegrator class works as follows. At all times, an instance of this class contains a certain "state of integration". Initially, after the constructor is called, the new query is totally separate from the existing path network inside of the NetworkIntegrator object. Thus, intially, the two are totally un-integrated. Eventually, the query and the network will be totally integrated, in which case the instance encapsulates a valid path network that totally contains the new query and could be directly executed by the system. An instance of the NetworkIntegrator class may also hold an intermediate level of integration, where the new query is only partially integrated into the existing path network.

4.2 The Enumerator Method

The most important method of the NetworkIntegrator class is the "*Enumerate*" method. A call to foo.Enumerate() on a NetworkIntegrator object foo returns a set of many new NetworkIntegrator objects. Every NetworkIntegrator object bar that is in this return set is "slightly more integrated" than foo. That is, in bar some small additional

part of the new query has been inserted into the existing path network compared to the extent to which the query was in the network in foo. The fact that many different NetworkIntegrator objects are returned from a call to foo.Enumerate() allows for the Enumerate method to return many possible ways to more tightly couple the new query with the existing network in foo. In fact, a call to foo.Enumerate() generally returns all possible ways to perform one more step of the integration, regardless of how desirable those steps are.

4.3 The Cost Function

To help in differentiating among the possible ways to perform the integration, the NetworkIntegrator class also has a "Coster" method. This method measures the goodness of the current (possibly partial) integration. This method returns an integer value that denotes the number of tuples in the path network. foo.Coster() can also take into account classical query optimization considerations, such as the join ordering for the new query in the network. If the join ordering is poor, then foo.Coster() might return a larger value compared to an integration with a high-quality join ordering.

It is important to note that while costing a partially integrated path network, returning the number of tuples in partially integrated path network is not enough. If cost function only approximates the number of tuples in partially integrated path network, then the optimizer will always join the smaller tables first. This might lead to local optimum while ignoring global optimum solutions in some case. Hence, the cost function is accompanied by a *mini-search* that tries to predict the final path network with a very simple search. This predicted path network is then costed and the number of tuples for it is returned rather than the partially integrated path network.

4.4 The Search Strategy

The reason for defining the "NetworkIntegrator" class is that it totally decouples the search strategy (that is, the way in which a high-quality integration is obtained) from the integration mechanism, which is embodied by the NetworkIntegrator class. Given an

implementation of the NetworkIntegrator class, almost any search strategy can be used. For example, the following pseudo-code would implement a greedy search strategy, using a NetworkIntegrator object foo[1] :

```
while (temp <- foo.Enumerate ()) is not empty:
        bestcost = inf
        for bar in temp, do:
                if bar.GetCost () < bestCost
                        bestCost <- bar.GetCost ()
                        nextStep <- bar
                end if
        end for
        foo <- nextStep
end while
```

Or, one could extend the greedy strategy to always keep the 10 best solutions so far. This would allow for a broader search, and could be done by adding a priority queue to the loop. In the following, I assume that the declaration:

```
PriorityQ Q (10)
```

returns a priority queue that has 10 slots in it. Any time that more than an 11 item is inserted into the queue, the item with the worst score is removed from the queue. Given this, the following pseudo-code implements a slightly more intelligent search strategy:

```
PriorityQ Q (10)
temp <- foo.Enumerate ()
for bar in temp, do:
        Q.insert (bar, bar.GetCost ())
```

[1] Note that the pseudo code is intended to express the design and not the implementation.

```
end for

while (TRUE)
        PriorityQ NewQ (10)
        while (Q.Remove (foo))
                temp <- foo.Enumerate ()
                if temp is empty:
                        return foo as the best network
                end if
                for bar in temp, do:
                        NewQ.insert (bar, bar.GetCost ())
                end for
        end while
        Q <- NewQ
end while
```

CHAPTER 5
IMPLEMENTATION

5.1 Types

The network integrator consist of two objects, namely the path network and the query description. Both the path network and the query description are of type graph. The graph is a network of waypoints and is represented using the adjacency list structure. To simplify the code and interaction with the execution engine, each waypoint is identified by an identifier which is generated by the Query Manager component[1] . The waypoints are stored in a hash table with identifier as the key for faster access. Each waypoint also contains a list of predicates.

The current implementation only support Select-Project-Join (SPJ) queries. It does not support sub queries, but can be extended easily by treating the sub queries as a new query and pipelining its result to the main SPJ query. Each predicate is associated with the query identifier. The predicate can be of following type:

1. **The join predicate**: It is of the form '`Table1.Attribute1 operator Table2.Attribute2`'.

2. **The selection predicate**: There are three types of selection predicates. The first type is of the form '`Table1.Attribute1 operator constant`', the second type is of the form '`Table1.Attribute1 operator Table1.Attribute2`' and the third type is an Empty selection which simply bypasses the data without any computation.

3. **The table scan predicate**: The job of the table scan waypoint is to scan the table and push the data through the memory hierarchy. There is only one table scan waypoint per table. However, the table scan waypoint can contain many table scan predicates each representing different queries.

4. **The top predicate**: This is a big waypoint which is pushed at the top of the query plan that performs aggregation, projection and other non-join operations.

[1] Each query and the waypoint in the system has an identifier associated with it. The job of the query manager is to generate and maintain these identifiers.

5.2 Enumeration

The `enumerate` method gets a network integrator object and returns a list of next possible network integrator objects. This method does not in any way affect the search strategy. For example, the search strategy such as look-ahead can enumerate more than once (depending on the look-ahead depth) before deciding which network integrator object should direct the search. To find next possible network integrator objects, the `enumerate` method gets every remaining predicate $\mathtt{P}_{QD}$ in the query description and tries to perform following three operations on every waypoint $\mathtt{W}_{PN}$ in the path network. Let $\mathtt{P}_{PN}$ be any predicate in the waypoint $\mathtt{W}_{PN}$ and $\mathtt{W}_{QD}$ be the waypoint that has the predicate $\mathtt{P}_{QD}$.

1. **Mapping**: If the predicate $\mathtt{P}_{QD}$ can be mapped onto the predicate $\mathtt{P}_{PN}$, then the predicate $\mathtt{P}_{QD}$ is added to the list of predicates of the waypoint $\mathtt{W}_{PN}$. The rules for mapping the predicates are discussed later in the section 5.3.

2. **Bypass**: If the predicate $\mathtt{P}_{QD}$ cannot be mapped onto the predicate $\mathtt{P}_{PN}$, then it tries to find out whether they are bypassable or not. The rules for bypassing a waypoint is discussed in the section 5.4. If the predicates are bypassable, then a new predicate $\mathtt{P}_{Bypassable}$, it created and added to the waypoint $\mathtt{W}_{PN}$. The Path Optimizer also recursively checks for the bypassable parents and adds the bypass predicates to them.

3. **New waypoint**: Irrespective of whether the predicates $\mathtt{P}_{QD}$ and $\mathtt{P}_{PN}$ are mappable or not, a new waypoint is created with the predicate $\mathtt{P}_{QD}$ in the path network.

The detailed algorithm for enumeration is given below (see Algorithm 1).

Algorithm 1: Enumeration Algorithm

```
Input: network integrator object (PN,QD)
Output: list of network integrator objects
Let PN = Input Path Network;
and QD = Input Query Description;
and returnList = list of network integrator objects to be returned (initially empty);
foreach predicate P_QD in the query description QD do
    foreach waypoint W_PN in the path network PN do
        Let P_PN be any predicate of W_PN;
        if isMappable(P_QD, P_PN) then
            Create a deep copy of the path network PN;
            Map P_QD onto W_PN-Copy of the copy;
            Add PN-Copy to the returnList;
        end
        else if isBypassable(P_QD, P_PN) then
            Create a deep copy of the path network PN;
            Add the bypass predicates in the child waypoints;
            Map P_QD onto W_PN-Copy of the copy;
            Add PN-Copy to the returnList;
        end
        Create a deep copy of the path network PN;
        Add a new waypoint with th predicate P_QD in the PN-Copy;
        Add PN-Copy to the returnList;
    end
end
```

5.3 Mapping Rules

The two predicates $\mathtt{P}_1$ and $\mathtt{P}_2$ can be mapped if they satisfy following criteria:

1. Both the predicates are of the same type. For example, if P_1 is a selection predicate and P_2 is a join predicate, they cannot be mapped.

2. If both the predicates are table scan or selection predicates and work on the same table, then they can be mapped else they cannot be mapped. For example, if P_1 is predicate of the form `nation.n_name = 'US'` and P_2 is predicate of the form `orders.o_orderstatus = 'F'`, then they cannot be mapped because they have different tables.

3. If both the predicates are join predicates, the left hand side table and attribute of the predicate P_1 should be same as either left or right hand side table and attribute of the predicate P_2. For example, the predicate `lineitem.l_suppkey = supplier.s_suppkey` can be mapped onto the predicate `lineitem.l_suppkey = partsupp.ps_suppkey`.

4. To keep the algorithm simple, the top predicates are not mappable.

5.4 Bypassable Rules

In some cases, bypassing a waypoint is helpful to reduce the data flow in the path network. For example, consider a path network shown in the figure 5-1, that has the join of lineitem and orders followed by the join of lineitem and supplier.

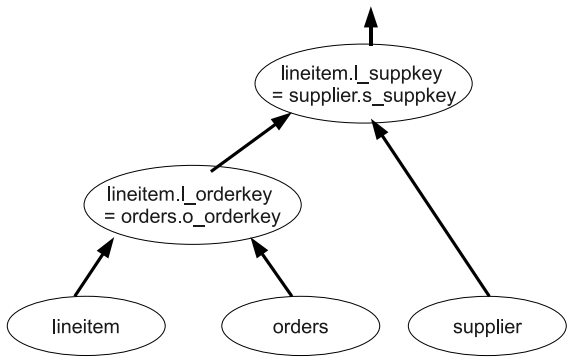

Figure 5-1. Path network before bypassing

Say, if the new query is the join of lineitem and partsupp, then the figure 5-2 shows the path network with bypass waypoints where as the figure 5-3 shows the path network without bypassing. Clearly, the former path network has less data flow than the latter path network.

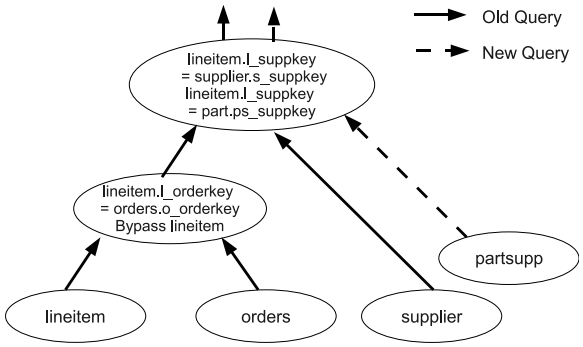

Figure 5-2. Final path network with bypassing

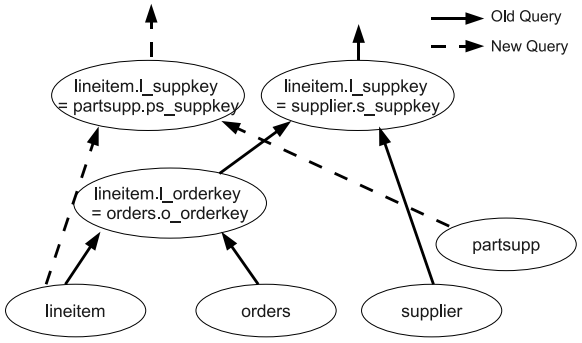

Figure 5-3. Final path network without bypassing

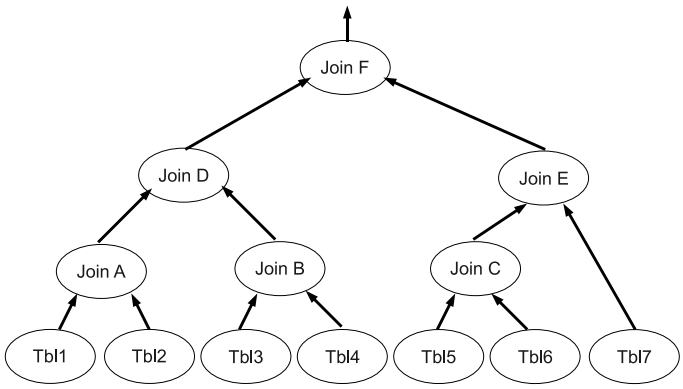

Figure 5-4. Example path network for bypassing

Only left hand side tables can be bypassed. This means if the new query had a join of orders and partsupp, then we cannot bypass. Bypassing rules are applied recursively, hence the tables involved in the new query should be on left stem of the child. For example, consider the path network shown in figure 5-4. Only a query with join of `Tbl1` and `Tbl5` can be bypassed for `Join F`. All other tables are right hand side tables for atleast one join. Though `Tbl3` is on left hand side of `Join B`, but it is on right hand side of `Join D`, hence it cannot be consider for bypassing.

5.5 The Cost Function

Given a partially integrated path network, the cost function first converts it to a fully integrated path network by performing a `mini-search`. Mini-search is a simple function that uses very simple heuristics to find the fully integrated path network. It is important to note that this fully integrated path network is only used to improve the costing and does not affect the search strategy. Once a fully integrated path network is found, the costing is performed.

The detailed algorithm for costing is given below (see Algorithm 2).

Algorithm 2: Cost Algorithm

```
Input: A Network Integrator object
Output: An Integer Cost
Let PN = Input Path Network and QD = Input Query Description;
Let Full-PN = Mini-Search(PN, QD) and H = Hashtbl of (Waypoint, Flow);
foreach waypoint W in Full-PN do
    Let I be the set of input waypoints and F be the output flow of W;
    foreach input I_k in the set I do
        if Waypoint_k not present in H then
            Add (Waypoint_k, Flow_k) in H;
        end
    end
    if W is a join waypoint then
        foreach predicate P in the waypoint W do
            Let S be the selectivity factor of P;
            if I_i and I_j are the inputs for P then
                F_i = S * I_i * I_j;
            end
        end
        F = max(F_i);
    end
    else if W is a selection waypoint then
        foreach predicate P in the waypoint W do
            Let S_i be the selectivity factor of P;
        end
        F = Input * (1 - (1 - S_1) * (1 - S_2) * ... * (1 - S_k));
    end
end
Return the sum of all the flows in the path network Full − PN.
```

Here we make a simple assumption that the tuples in the join predicate with the maximum flow subsumes the tuples in the remaining join predicates of the same waypoint. However, this assumption does not apply to all the cases, it provides a good approximation of the data flow in the system.

The selectivity factor for each predicate is calculated using the statistics provided by the `Statistics` module and the method described in [21] and [8]. The table 5-1 gives the selectivity factor for various cases[2] .

Table 5-1. Selectivity Factor

Type of predicate	Condition	Selectivity factor
Selection	$\sigma_{R.A=const}$	$\frac{1}{ValueCount(R.A)}$
Selection	$\sigma_{R.A<const}$	$\frac{1}{3}$
Join	$\bowtie_{R.A=S.B}$	$\frac{1}{max(ValueCount(R.A), ValueCount(S.B))}$

5.6 The Search Strategy

The search strategy used in this thesis is a look-ahead search with user-specified look-ahead depth. The `search` function is a recursive function that uses `enumerate` method and `cost` function to find the final path network. It is important to note that these three function are independent of each other and any of them can be replace by an equivalent function without affecting the others. For example, the look-ahead search can be replaced by a greedy or exhaustive search without affecting the enumerate or cost function.

[2] In table 5-1, ValueCount(R.A) means number of distinct values of attribute A in relation R.

The detailed algorithm for the search is given below (see Algorithm 3).

Algorithm 3: Search Algorithm

```
Input: List of Network Integrator objects: lni
Output: A fully integrated path network
foreach Path network PN in lni do
    if Is PN fully integrated then
        Return PN;
    end
    else
        Perform Look-Ahead on PN;
        Let newLNI be list of new network integrator objects from look-ahead;
        Recursively call this algorithm using newLNI;
    end
end
```

CHAPTER 6
EXPERIMENTAL RESULTS

6.1 Goal

Using the experiment, I compare the proposed algorithm that uses a look-ahead search, mini-search (while costing) and data-centric cost function with other family of algorithms. For comparison, I use an objective function that counts the number of tuples in the path network. This function has a bias towards the data-centric cost function. Infact, the better way to compare these algorithms would be to run the queries for different path networks on the Datapath system. However, the execution engine of the Datapath system is not fully implemented and hence I use the above objective function. Also, I compare the family of algorithms based on the time taken to generate the path network.

6.2 Setup

For the sake of comparing different search techniques with the proposed algorithm, I have created a framework that treats query optimization as a state-space search problem. Using this framework, the Path Optimizer searches for the solution by using a top-down approach on the search tree.

The framework takes a configuration object that specifies three important parameters:

1. *Search algorithm*: The search algorithm can be exhaustive, greedy or look-ahead. The search algorithm takes a list of network integrator objects and returns a list of next possible network integrator objects.

2. *Selector function*: The selector function takes as input a list of network integrator objects and returns the best possible network integrator objects depending on the algorithm. The current framework supports two blind selector functions (i.e FIFO and Random) and two cost based selector functions.

3. *Cost function*: The framework allows two cost based selectors, namely the proposed data-centric cost function which counts the number of tuples and a compute-centric cost function that counts the number of waypoints or the computations.

It is important to note that every search algorithm calls the `enumerate` method. The exhaustive search algorithm recursively calls enumerates on all possible network integrator

objects. The greedy search algorithm prunes the branches of the search tree based on the selector function and hence only enumerates a small subset of the possible network integrator objects. A look-ahead search however, does not take an immediate greedy decision before pruning, but enumerates until few extra levels of the search tree. This improves the quality of the result found by the look-ahead search. It is important to note that a look-ahead search with zero depth simulates a pure greedy approach, whereas a look-ahead search with an infinite depth simulates an exhaustive search. The figure 6-1 shows different modules of the framework and their interfaces.

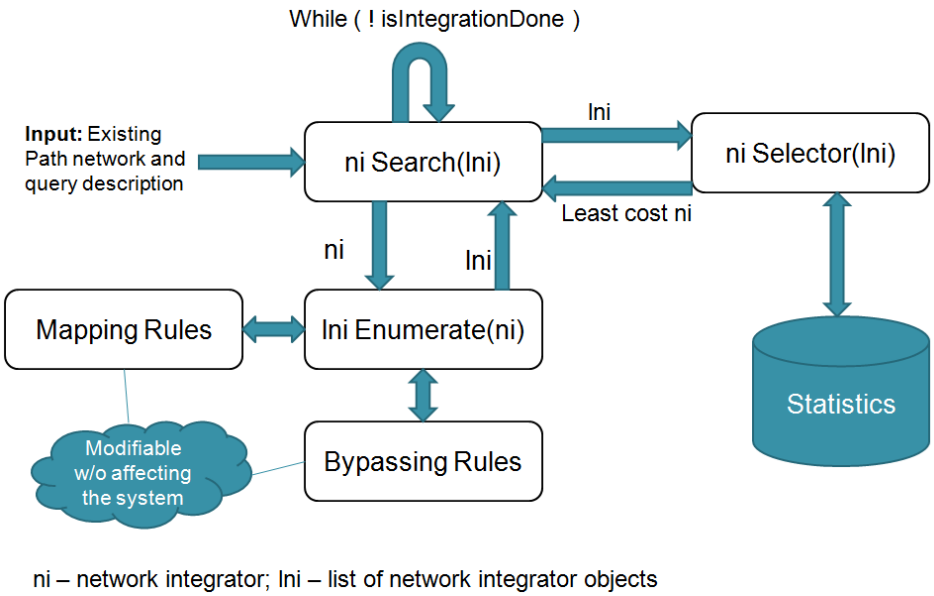

Figure 6-1. Framework for testing different query optimization techniques

The Path Optimizer is tested on 8 TPC-H queries[1] . These queries are randomly shuffled and are incrementally given to the Path Optimizer. The same sequence of queries are also given to different combinations of the search algorithms, selector functions and the cost functions. The framework is tested on ten random input orderings and the cost of final path network and also the time taken by each algorithm is recorded into a

[1] The TPC-H query 2, 3, 5, 10, 11, 18, 20 and 21 are tested using the given framework.

comma-seperated-value (csv) file. The framework also generates a PDF file which displays the final path network for each path network using GraphViz software[1].

6.3 Experimental Results

The table 6-1 shows the cost and the time taken by each algorithm.

The figure 6-2 compares the average cost of the FIFO selector with that of other selectors.

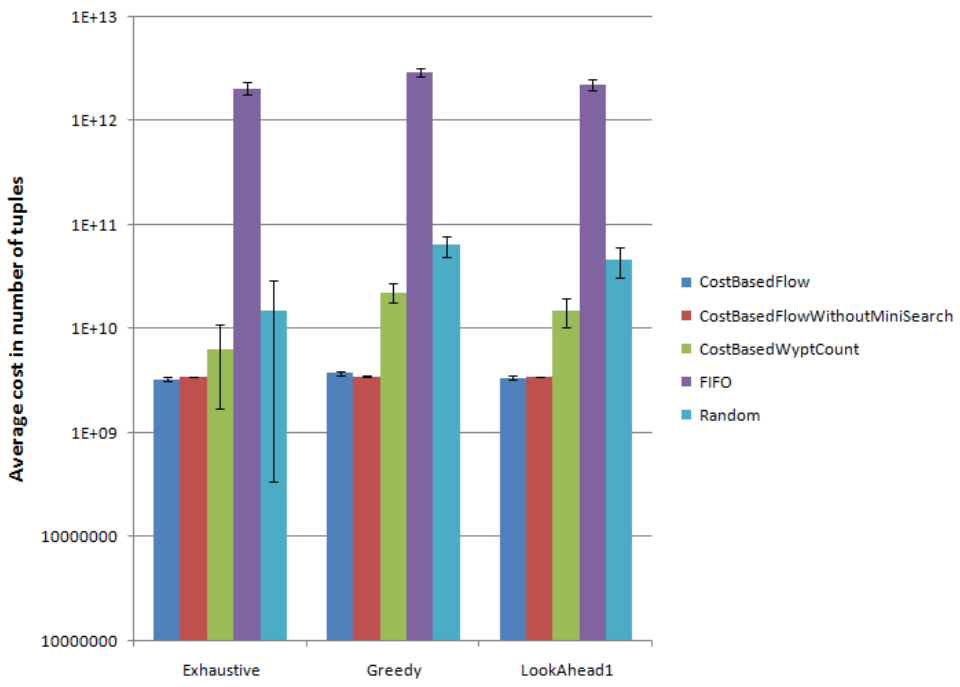

Figure 6-2. Comparison of FIFO with other selectors

The figure 6-3 compares the average cost of the Random selector with the average cost of the cost-based selectors.

The figure 6-4 compares the average cost of the cost-based selectors.

The figure 6-5 compares the average time taken by all the selectors.

6.4 Analysis

The above results show that the exhaustive search always gives the best results, while greedy search usually gives the worst results. Also, the look-ahead search gives the results very similar to the exhaustive search. For less than eight input queries, the

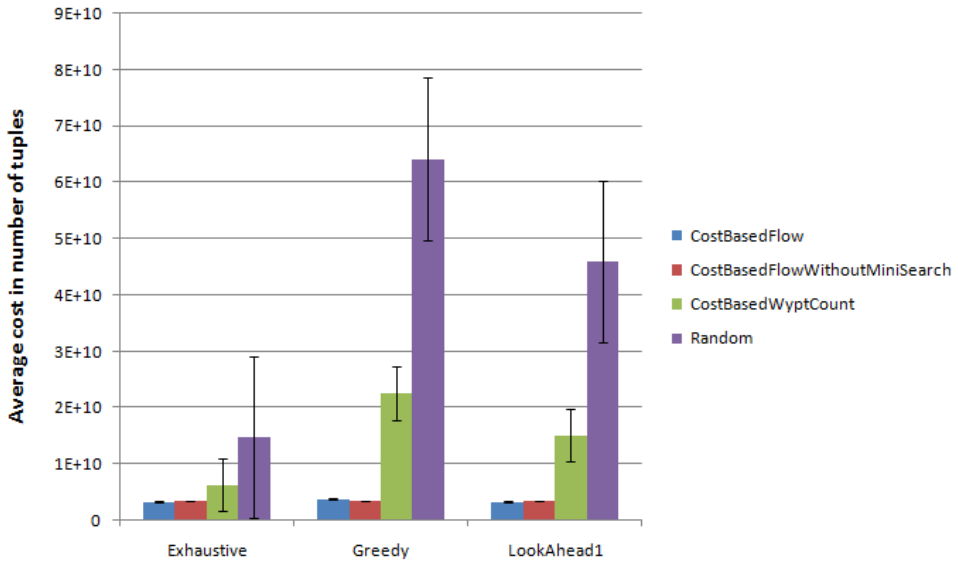

Figure 6-3. Comparison of Random selector with the cost based selectors

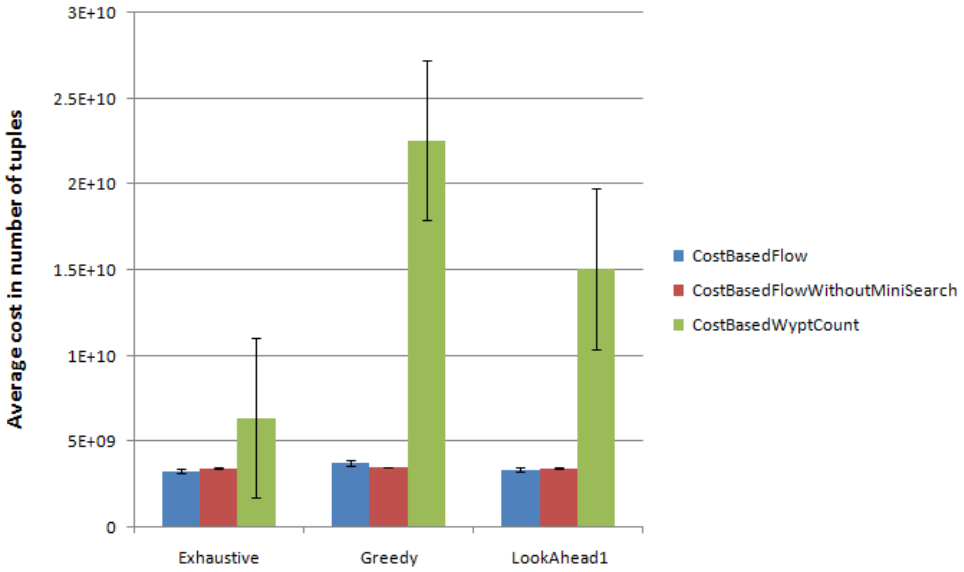

Figure 6-4. Comparison of the cost based selectors

Table 6-1. Cost and Time taken by each algorithm

Search	Selector	Avg cost	Max cost	Min cost	Avg time	Max time	Min time
Exhaustive	Costbased with mini-search	3308260720	4339400090	2146320100	180.13	312.22	80.91
Exhaustive	Costbased without mini-search	3449952840	4339400095	2912333459	162.5	312.27	85.99
Exhaustive	Waypoint count	4395383130	8972215000	1440000400	25.903	73.17	3.37
Exhaustive	FIFO	2.05728E+12	8.10005E+12	1440000555	33.904	104.25	4.17
Exhaustive	Random	14768863045	74218667890	1520010980	41.712	119.29	3.98
Greedy	Costbased with mini-search	3746172230	4769866765	3348026765	36.827	74.72	10.85
Greedy	Costbased without mini-search	3507063135	3640986771	3251026765	36.997	75.28	10.87
Greedy	Waypoint count	22552508025	65898400180	2572666700	0.493	0.56	0.305
Greedy	FIFO	2.96523E+12	1.60001E+13	4.91044E+11	0.692	0.8	0.407
Greedy	Random	64181229885	94817668050	16000632860	0.643	0.84	0.409
Look-ahead	Costbased with mini-search	3385487380	4470200090	2629520100	71.268	109.94	33.27
Look-ahead	Costbased without mini-search	3444960100	4554400090	2713720100	71.061	110.68	33.18
Look-ahead	Waypoint count	15058316550	32314505050	8640000000	0.426	0.52	0.281
Look-ahead	FIFO	2.23372E+12	9.6E+12	6480604050	0.536	0.63	0.322
Look-ahead	Random	45932428745	79600156560	28000056210	0.612	0.69	0.372

look-ahead depth of 1 is sufficient in most cases and performs as good as the depth of 2 or 3. The time taken by an algorithm depends upon the number of network integrator object it enumerates. Hence, exhaustive search takes a lot more time than the look-ahead

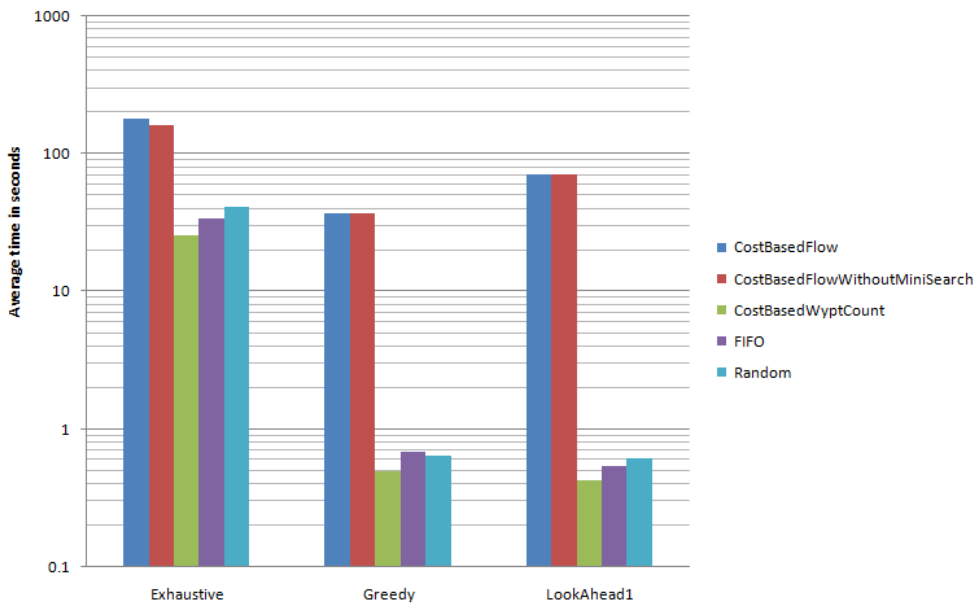

Figure 6-5. Comparison of the the average time taken by the selectors

or greedy search. Though greedy search takes less time, it does not perform as good as the look-ahead search (see table 6-1). Therefore, the proposed algorithm uses the look-ahead search.

Also, the costing of a network integrator object is a time-consuming operation. In fact, time taken by blind-selectors and exhaustive search is almost equal to the time taken by the data-centric cost-based selectors using a greedy search.

Though cost based selectors take more time than the blind selectors, they usually provide the path network with orders of magnitude less number of tuples than the blind selectors. Due to the ordering of folding function, the FIFO selector tries to select the path network with extra waypoints. In fact, FIFO acts as a single query optimizer because it always tries to introduce new flows in the network and hence produces worst results. The figure 6-2 shows that mapping waypoints provides significant gain over single query optimization. Random and WaypointCount (compute-centric) cost function are both bad. However, compute-centric (or the waypoint count) cost-based selector performs well for exhaustive search (but not better than data-centric function). This is because

the exhaustive search takes the decision at the end after all the enumeration has been completed. Also, lower number of waypoints generally have less flow, especially for TPCH queries where joins are usually done on similar tables and only on primary keys. For look-ahead search, compute-centric (or the waypoint count) cost-based selector does not perform well (see figure 6-4). Hence, the proposed algorithm uses the data-centric cost function rather than blind selectors or compute-centric (or the waypoint count) cost-based selector.

The above results show that statistically data-centric cost-based selector performs better than waypoint-count cost based selector. The example below explains the reason for this behaviour. Consider the test case where TPC-H query 11 is the first query and TPC-H query 5 is the second query.[2]

```
Query 11:
select *
from partsupp, supplier, nation
where
ps_suppkey = s_suppkey and s_nationkey = n_nationkey
and n_name = '[NATION]'
Query 5:
select *
from customer, orders, lineitem, supplier, nation, region
where
c_custkey = o_custkey and l_orderkey = o_orderkey
and l_suppkey = s_suppkey and c_nationkey = s_nationkey
and s_nationkey = n_nationkey and n_regionkey = r_regionkey
```

[2] The query 5 and 11 are simplified to work for the optimizer. For example, the projection and aggregation operators are ignored.

```
and r_name = '[REGION]' and o_orderdate >= 99990101
```

For query 11 both data-centric and waypoint-count selectors produce same path network. This path network is shown in the figure 6-6. But when query 5 is integrated onto the path network with query 11, the path network generated by waypoint-count selector is shown in the figure 6-7 and that generated by the data-centric selector is shown in the figure 6-8[3] . Note that both path network have same number of waypoints. So, waypoint-count selector treats both of them equally good and choses 6-7. In the path network 6-7, orders and customer tables are joined after lineitem. It is clear that having this join lower down the query plan is a better choice as it produces less flow. The data-centric selector is cognizant of this fact and hence choses 6-8. Also, the above experimental results attest that the path network selected by data-centric selector has lower flow than the waypoint-count selector.

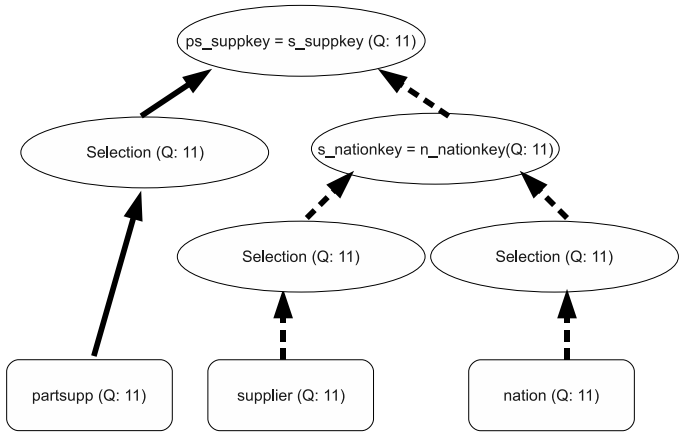

Figure 6-6. Path network after query 11

Mini-search performs well for exhaustive and look-ahead but not for greedy.This is because it tries to predict the future join ordering and does not simply join the smaller

[3] The dotted edges represents less flow than the dashed edges and the dashed edges has less flow than the solid edges.

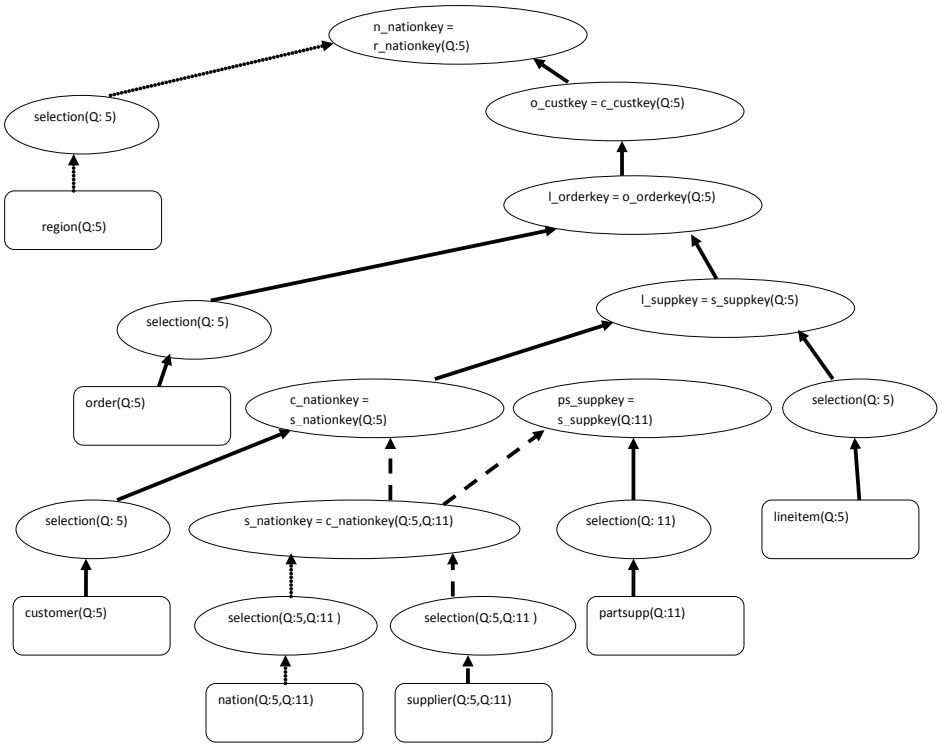

Figure 6-7. Path network after query 5 for waypoint-count selector

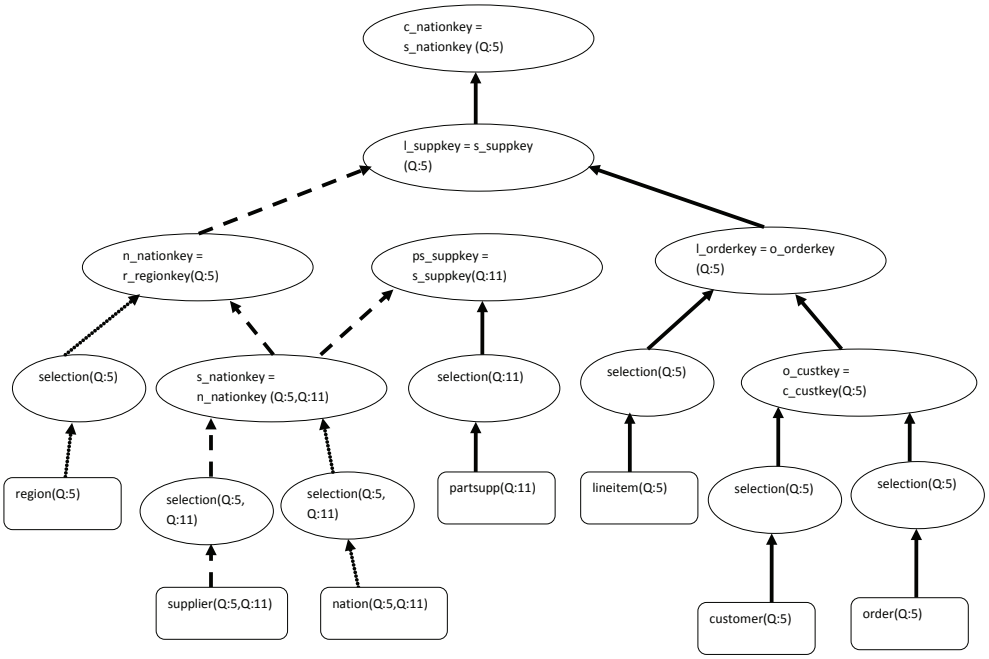

Figure 6-8. Path network after query 5 for cost-based selector

tables first. However, since this is only a prediction based on some simple heuristics, it does not always work for greedy strategies. Also, mini-search does not incur any significant overhead with respect to time. Hence, the proposed algorithm uses mini-search.

CHAPTER 7
CONCLUSION

Since no conventional multi-query optimizers are suitable for the data-centric databases (like the Datapath system), I have proposed an algorithm that optimizes the queries for the Datapath system. I have also tested and compared various search strategies against the proposed algorithm using a data-centric cost function. The experimental results show that the proposed algorithm produces a good path network (or global query plan) in reasonable amount of time.

CHAPTER 8
FUTURE WORK

The Path Optimizer only supports the mapping of join and selection predicates. I plan to introduce mapping rules for the top predicate (i.e group-by and order-by) and also modify the path optimizer to support sub-queries.

Also, the Path Optimizer assumes that all the queries start their execution at the same time. Though this assumption simplifies the optimization process, it does not account for the state of execution engine. Also, the Path optimizer relies only on the `Statistics` module for the cost function and hence is not adaptive. The Path Optimizer could be improved by considering the feedback regarding the state of execution engine and also the execution time for each waypoint from the execution engine.

The cost function assumes that the predicate with maximum number of tuples subsumes the predicates with fewer tuples. Hence, the cost function ignores the extra tuples that are not part of the larger predicate but increases the flow of data through the path network.

Also, I intend to improve the performance of the algorithm so that it can produce a reasonable path network in less time. The current algorithm only integrates one query at a time. This could easily be modified to produce a good path network for batch of queries by reordering the input queries.

REFERENCES

[1] Graphviz - graph visualization software. http://www.graphviz.org/, July 2009.

[2] A. Ailamaki, D. J. Dewitt, M. D. Hill, and D. A. Wood. Dbmss on a modern processor: Where does time go? In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 266–277, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[3] U. S. Chakravarthy and J. Minker. Multiple query processing in deductive databases using query graphs. In *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*, pages 384–391, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[4] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43, New York, NY, USA, 1998. ACM.

[5] F.-C. F. Chen and M. H. Dunham. Common subexpression processing in multiple-query processing. *IEEE Trans. on Knowl. and Data Eng.*, 10(3):493–499, 1998.

[6] N. N. Dalvi, S. K. Sanghai, P. Roy, and S. Sudarshan. Pipelining in multi-query optimization. In *PODS '01: Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 59–70, New York, NY, USA, 2001. ACM.

[7] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Found. Trends databases*, 1(1):1–140, 2007.

[8] H. Garcia-Molina, J. Ullman, and J. Widom. *Database Systems: The Complete Book*. Pearson Education, 2008.

[9] G. Graefe. Sort-merge-join: An idea whose time has(h) passed? In *Proceedings of the Tenth International Conference on Data Engineering*, pages 406–417, Washington, DC, USA, 1994. IEEE Computer Society.

[10] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 209–218, Washington, DC, USA, 1993. IEEE Computer Society.

[11] P. Hall. Optimization of a single relational expression in a relational database system. *IBM Journal of Research and Development*, pages 244–257, 1976.

[12] S. Harizopoulos and A. Ailamaki. A case for staged database systems. In *In Proceedings of 1st Conference on Innovative Data Systems Research*, 2003.

[13] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984.

[14] Y. E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.

[15] M. Jarke and J. Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16(2):111–152, 1984.

[16] Y. C. Kang. *Randomized algorithms for query optimization.* PhD thesis, Madison, WI, USA, 1991.

[17] P.-A. Larson and G. Graefe. Memory management during run generation in external sorting. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 472–483, New York, NY, USA, 1998. ACM.

[18] T. Neumann. *Efficient Generation and Execution of DAG-Structured Query Graphs.* PhD thesis, 2005.

[19] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. Alphasort: a cache-sensitive parallel external sort. *The VLDB Journal*, 4(4):603–628, 1995.

[20] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. *SIGMOD Rec.*, 29(2):249–260, 2000.

[21] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, I. A. Lorie, and T. G. Price. Access path selection in a relational database management system. pages 23–34, 1979.

[22] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13:23–52, 1988.

[23] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 510–521, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[24] K. Shim, T. Sellis, and D. Nau. Improvements on a heuristic algorithm for multiple-query optimization. *Data Knowl. Eng.*, 12(2):197–222, 1994.

[25] S. N. Subramanian and S. Venkataraman. Cost-based optimization of decision support queries using transient-views. In *ACM SIGMOD Conf*, pages 319–330, 1998.

[26] I. H. Toroslu and A. Cosar. Dynamic programming solution for multiple query optimization problem. *Inf. Process. Lett.*, 92(3):149–155, 2004.

[27] L. B. Warshaw and D. P. Miranker. Rule-based query optimization, revisited. In *ACM International Conference on Information and Knowledge Management (CIKM)*, pages 267–275. ACM Press, 1999.

[28] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: dynamic bandwidth sharing in a dbms. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 723–734. VLDB Endowment, 2007.

BIOGRAPHICAL SKETCH

Niketan R. Pansare received his Bachelor of Engineering degree in Information Technology from Veermata Jijabai Institute of Technology in 2006. He then received his Master of Science degree in Computer Engineering from the University of Florida in Fall 2009. His primary research is focused on Database and Machine Learning.