
ABSTRACT

GUPTA, SHALU View Selection for Query-Evaluation Efficiency using Materialized Views

(Under the direction of Dr. Rada Chirkova)

The purpose of this research is to show the use of derived data such as materialized views for run time optimization of aggregate queries. In this thesis, we show the trade off between the time taken to design the views Vs the query run time. We have designed a system called Query Performance Enhancement by Tuning (QPET) which implements the idea of designing and using materialized views to answer frequent aggregate queries.

View Selection for Query-Evaluation Efficiency using Materialized Views

by

Shalu Gupta

A Thesis Submitted to the Graduate Faculty of
North Carolina State University
in Partial Fulfillment Of the Requirements for the Degree of
Master of Science

Computer Science

Raleigh

2005

APPROVED BY

Dr. Rada Chirkova
Chair of the Advisory Committee

Dr. Munindar Singh

Dr. Jaewoo Kang

Dedicated to

My Mom.

BIOGRAPHY

Shalu Gupta, was born on September 7, 1981 in Jabalpur, Madhya Pradesh, India. She received her Bachelor of Engineering in Computer Science and Engineering from Jabalpur Engineering College in 2003. She joined the Master's program at the Department of Computer Science at North Carolina State University in Fall 2003. As part of her graduate studies, she interned at IBM, RTP during Summer and Fall of 2004.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and gratitude to my advisor Dr. Rada Chirkova for her constant support, guidance and motivation. I would also like to specially thank Dr. Munindar Singh and Dr. Jaewoo Kang for being on my thesis committee.

I greatly appreciate Kyoung Hwa Kim, Rohit Ghatol and Simran Sandhu for helping me with the implementation of this project.

A warm and special thanks to my Mom, to whom I dedicate my thesis work, my sister Dr. Neha Tiwari and my brother-in-law Mr. Manu Tiwari. They have been a pillar of support for me and without them I could not have achieved this.

Finally plaudits to all my friends around specially to Anushree, Ramya, Mithun, Ravi and Radhika for putting up with me.

Table of Contents

List of Figures	vii
1 Introduction	1
1.1 Overview	1
1.2 Motivating Example	2
1.3 Contributions	5
1.4 Thesis Organization	6
2 Preliminaries and Problem Statement	7
2.1 Star Schema	7
2.1.1 Snowflake Schema	8
2.2 Materialized Views	8
2.3 Greedy Algorithm	9
2.4 Problem Statement	10
2.4.1 Considerations while choosing the set of tables	10
2.5 Related Work	12
3 System Architecture	13
3.1 Designing Views in the QPET Framework	13
3.2 Using views in the QPET framework	15
4 View Selection	17
4.1 Introduction	17
4.2 Our Approach	18
5 Query Rewriting	21
5.1 Introduction	21
5.2 Steps for Query Rewriting	21
5.2.1 Preprocessing an input query	21
5.2.2 Postprocessing an input query	22
5.2.3 Example rewriting	22
6 Experimental Setup and Results	25

List of References	33
A TPC-H Benchmarking Data	36
A.1 Introduction	36
A.1.1 CREATE Statements	36
A.2 Queries used for the Experiments	37
A.2.1 Pricing Summary Report Query (Q1)	37
A.2.2 Shipping Priority Query (Q3)	39
A.2.3 Local Supplier Volume Query (Q5)	40
A.2.4 Forecasting Revenue Change Query (Q6)	41
A.2.5 Volume Shipping Query (Q7)	43
A.2.6 National Market Share Query (Q8)	44
A.2.7 Product Type Profit Measure Query (Q9)	46
A.2.8 Returned Item Reporting Query (Q10)	47
A.2.9 Top Supplier Query (Q15)	49
A.2.10 Large Volume Customer Query (Q18)	50
B Sky Sever Astronomical Data	52
B.1 Database details	52
B.2 Queries used for the experiments	52
C Implementation Details	58
C.1 Conversion of a Query from SQL to Datalog	58
C.1.1 Introduction	58
C.1.2 Data Structure used for DataLog	61
C.1.3 Algorithm for conversion of a SQL Query to DataLog	61
C.1.4 Algorithm for translating the Operator Expression	64
C.1.5 Algorithm for translating the And Expression	65
C.2 Input to GenerateLattice	65
C.2.1 Algorithm for generating the input to GenerateLattice	65
C.2.2 Example	68
C.2.3 Steps	69
C.3 For the Greedy Algorithm:	70
C.3.1 Structure for the lattice:	71
C.3.2 Implementation Details:	71
C.4 Algorithm for Rewriting a Query using a Materialized View	73
C.5 Steps	74
C.6 Source Code for our implementation of Greedy Algorithm [17]	75

List of Figures

2.1	Star Schema	8
2.2	Lattice Structure	9
3.1	Overall system architecture [6, 7]	14
3.2	Designing derived data in QPET	15
3.3	Using derived data in QPET	16
6.1	TPC-H table sizes	26
6.2	Query Runtimes Phase I	27
6.3	View and Query Sizes Phase I	28
6.4	Time to Design Views Phase I	28
6.5	Lattice Attributes and View Design Time Phase II	29
6.6	Query Runtimes Phase II	30
6.7	Query Runtimes Phase II [19]	30
6.8	Sky Server Table Sizes	31
6.9	Lattice Attributes and View Design Time for Sky Server Database	31
6.10	Query Runtimes for Sky Server Database	32
6.11	Query Runtimes for Sky Server Database [19]	32

Chapter 1

Introduction

1.1 Overview

Answering queries using derived data is an increasingly popular method for query optimization. Derived data, such as materialized views or indexes, are routinely used in data-intensive systems to improve query-evaluation performance. In this context, the problem of *designing derived data* is as follows: Given a set of queries, a database, and a set of constraints on derived data (e.g., a storage limit), return definitions of derived data that, when materialized in the database, would satisfy the constraints and reduce the evaluation costs of the queries.

The problem of designing derived data becomes more important in case of queries with aggregation which are generally used for data warehousing and OLAP applications. For queries with aggregation, we need to consider the grouping arguments while designing views to materialize.

In this thesis, we consider the problem of designing views over a subset of base tables from the database to answer frequent aggregate queries. Our main objective is to improve the response time of queries under the given constraints using derived data. The views are materialized and stored as materialized views. Every time a query is entered in the system, it will be automatically answered using the materialized views or using a combination of the base tables and the materialized views. The users of the system are not aware whether the query is answered using a base table or a materialized

view. A user just gets a better response time for a query.

We have designed a system, which we call **Query-Performance Enhancement by Tuning (QPET)** [12] to implement the idea of designing and using materialized views to answer queries. QPET is based on an open source object-relational database management system, PostgreSQL [20]. We have mainly worked on the optimizer module of the query processor. One of the main features of QPET is that its optimizer considers materialized views in addition to base tables while generating query plan. We concentrate on the problem of designing views over a specific set of base tables.

1.2 Motivating Example

We now present two examples to show the motivation behind our work.

Consider a data warehouse with stored relations **Sales**, **Customer**, and **Time**:

```
Sales(CustID,DateID,ProductID,SalespersonID,QuantitySold,TotalAmount,Discount)
Customer(CustID,CustName,Address,City,State,RegistrDateID)
Time(DateID,Month,Year)
```

Keys of the tables are underlined. **Sales** is the fact table, and **Customer** and **Time** are dimension tables.

Let the query workload of interest have two star-schema queries, Q1 and Q2. Query Q1 asks for the total quantity of products sold per customer in the second quarter of the year 2004. Q2 asks for the total product quantity sold per year for all years after 1997 to customers in North Carolina.

```
Q1: SELECT c.CustID, SUM(QuantitySold)
      FROM Sales s, Time t, Customer c
      WHERE s.DateID = t.DateID AND s.CustID = c.CustID
      AND Year = 2004 AND Month >= 4 AND Month <= 6
      GROUP BY c.CustID;
```

```
Q2: SELECT t.Year, SUM(QuantitySold)
```

```

FROM Sales s, Time t, Customer c
WHERE s.DateID = t.DateID AND s.CustID = c.CustID
AND Year > 1997 AND State = 'NC'
GROUP BY t.Year;

```

We consider designing materialized views that could be used to answer the workload queries via equivalent *central rewritings* [3]. Intuitively, unlike the rewritings of [17, 15], central rewritings may involve *joins* of aggregate views with other relations. Here we show two such rewritings, R1 and R2, of the query Q1, and discuss the tradeoffs in using each rewriting to evaluate Q1. Consider a view V1, which returns the total quantity of products sold to each customer based on just the **Sales** data:

```

V1: SELECT CustID, DateID, SUM(QuantitySold) AS SumQS FROM Sales
GROUP BY CustID, DateID;

```

A rewriting R1 of the query Q1 uses a *join* of the view V1 with relations **Customer** and **Time**:

```

R1: SELECT c.CustID, sum(SumQS) FROM V1, Time t, Customer c
WHERE V1.DateID = t.DateID AND V1.CustID = c.CustID AND Year = 2004
AND Month >= 4 AND Month <= 6
GROUP BY c.CustID;

```

Another equivalent rewriting of Q1, R2, uses a view V2, which has *two* relations in its FROM clause:

```

V2: SELECT c.CustID AS CID,
DateID AS DID, SUM(QuantitySold) AS SumQS
FROM Sales s, Customer c
WHERE s.CustID = c.CustID
GROUP BY c.CustID, DateID;

```

```

R2: SELECT CID, sum(SumQS)
FROM V2, Time t
WHERE V2.DID = t.DateID AND Year = 2004

```

```

AND Month >= 4 AND Month <= 6
GROUP BY CID;

```

Assuming that there are typically many product IDs and product categories per sale event, and that different salespeople are responsible for selling products in different categories, the relation for each of **V1** and **V2** will likely be much smaller than the fact table **Sales**. Accordingly, the evaluation time of each of **R1** and **R2** will likely be smaller than the evaluation time of the query **Q1**; similarly, the costs of evaluating the query **Q2** could be reduced using the corresponding rewritings with **V1** or **V2**. In addition, assuming the stored data satisfy the foreign key constraints that are typical for star-schema databases, the evaluation time of the rewriting **R2** will be strictly less on typical databases than the evaluation time of **R1**, as the relations for **V1** and **V2** in this case are of the same size and as evaluating **R1** involves an extra join compared to **R2**.

We now make observations about the view **V1** and compare the set $\{ \mathbf{V1} \}$ with sets of views produced by BPUS [17], described in Section 2.3. BPUS considers designing views over the entire set of tables rather than designing views over just a subset of tables from the star schema. For any materialized view designed by the BPUS algorithm, if the view is usable in evaluating some query, then the answer to the view is *the only* relation needed in the evaluation. That is, views produced by the BPUS algorithm determine joinless rewritings of queries. In addition to joinless rewritings, in our approach we use rewritings that are computed via joins of aggregate views with other relations. All the observations we make here about the view **V1** also hold about the view **V2**. First, **V1** *by itself* provides equivalent central rewritings of both **Q1** and **Q2** and thus gives a *solution* $\{ \mathbf{V1} \}$ for the workload $\{ \mathbf{Q1}, \mathbf{Q2} \}$. Second, that BPUS would consider all subsets of a set of four grouping arguments. In contrast, to design the set $\{ \mathbf{V1} \}$ we had to look at just *two* grouping arguments, **CustID** and **DateID**. The reason is, we consider only aggregate views whose **FROM** clause has just the **Sales** relation, and **CustID** and **DateID** are the only grouping arguments of such views that are required in constructing equivalent rewritings of the workload queries.

We now show that our approach can be used to design materialized views for aggregate queries for non star-schema queries as well. Consider the database schema and query Q1 of the previous example, and let query Q3 ask for the total quantity of products sold per customer, for customers who got *registered* in the second quarter of 2004:

```
Q3: SELECT c.CustID, SUM(QuantitySold) FROM Sales s, Time t, Customer c
      WHERE s.CustID = c.CustID AND c.RegistrDateID = t.DateID AND Year = 2004
      AND Month >= 4 AND Month <= 6
      GROUP BY c.CustID;
```

As the join condition $c.RegistrDateID = t.DateID$ of Q3 differs from the condition $s.DateID = t.DateID$ of Q1, $\{ Q1, Q3 \}$ is not a workload of star-schema queries. Thus, BPUS is not applicable. At the same time, view V1 from Example 1.2 can be used to evaluate both Q1 and Q3; we give here an equivalent rewriting of Q3 using V1.

```
R3: SELECT c.CustID, sum(SumQS) FROM V1, Time t, Customer c
      WHERE c.RegistrDateID = t.DateID AND V1.CustID = c.CustID AND Year = 2004
      AND Month >= 4 AND Month <= 6
      GROUP BY c.CustID;
```

Similarly, if we add `RegistrDateID` to the list of grouping arguments of view V2 in Example 1.2, the resulting view could be used to evaluate each of Q1 and Q3.

1.3 Contributions

This thesis makes the following contributions:

1. Implementation and experiments on the idea of materializing views for a subset of tables, thereby reducing the time taken in designing these views, while improving the response time.
2. Implement and test a system architecture for automated query-performance by periodically designing materialized views and using these views to answer queries.
3. Implement and test an approach which can be used to design views for non star schema queries as well as shown in the second example above.

1.4 Thesis Organization

Chapter 2 gives the background knowledge about some of the key concepts discussed and formally states the problem being addressed by this work. Chapter 2 also discusses the related work done in this area. Chapter 3 describes our QPET system. Chapters 4 and 5 discuss our approach in more detail. Finally Chapter 6 shows the experimental setup and results.

Chapter 2

Preliminaries and Problem Statement

2.1 Star Schema

A *star schema* is based on the *dimensional modeling principle* [1]. The schema has a fact table and multiple dimension tables. All the dimension tables satisfy foreign key constraints with the fact table. The fact table, which is the central table, contains the quantitative data whereas the dimension tables store more descriptive information about the various attributes of the fact table. The fact table links to the other relations using foreign key dependencies. A typical star schema is shown in Figure 2.1

Most data warehousing applications use a star schema database. The following is the commonly used terminology for a star schema.

1. Measure: The aggregated data is called the measure.
2. Fact Table: This is the central table in the schema. It has foreign key dependencies to all the other tables. The fact table generally has more number of attributes, than the dimension tables, which are used to establish the dependencies with the dimension tables.
3. Dimension Table: The dimension tables describe the dimension attributes.

The star schema has foreign key constraints from the fact table to the dimension tables.

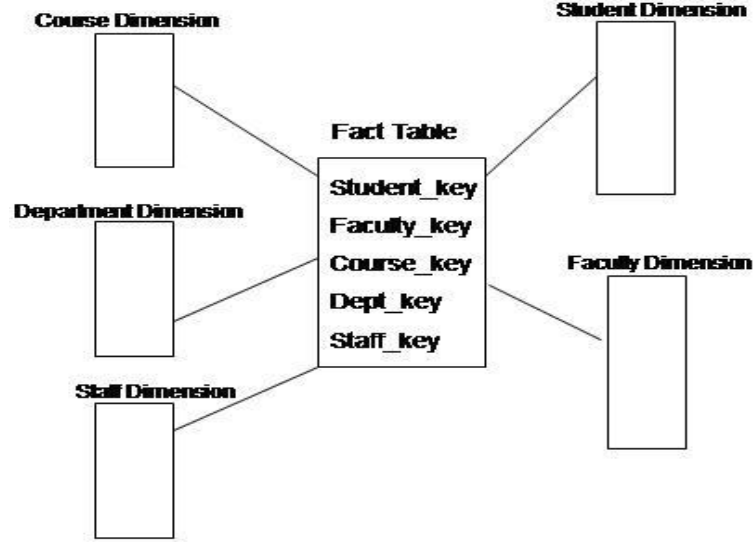


Figure 2.1: Star Schema

2.1.1 Snowflake Schema

A snowflake schema is a more complex data warehouse model than a star schema, and is a type of star schema. In a snowflake schema, dimension tables can be further decomposed into sub-dimension tables. The TPC-H database [22] is an example of the snowflake schema.

2.2 Materialized Views

A *materialized view*, in a star or snowflake data warehouse, can be defined as a pre-computed table comprising aggregated or joined data from fact and possibly dimension tables. Materialized views within our system are transparent to users, which means that views and tables can be added or dropped without affecting the validity of the data. [14] study the problems associated with maintaining a materialized view. Given the way a materialized view is designed, it may be expensive to update a view. A materialized view needs to be updated whenever the base tables on which the view is designed, are updated. We do not discuss this problem in our work.

2.3 Greedy Algorithm

Our algorithm uses the greedy algorithm BPUS [17], as a subroutine. A workload of star schema queries is considered, which has the same **FROM** clause and the same join conditions in the **WHERE** clause. The greedy algorithm considers a view lattice, of all the possible views that can be materialized over a database. It is a search space of views for the workload, with edges between views denoting which view can be evaluated using another view. Given n grouping attributes, [17] constructs a lattice of height n . The lattice can be shown in the form of the Figure 2.2.

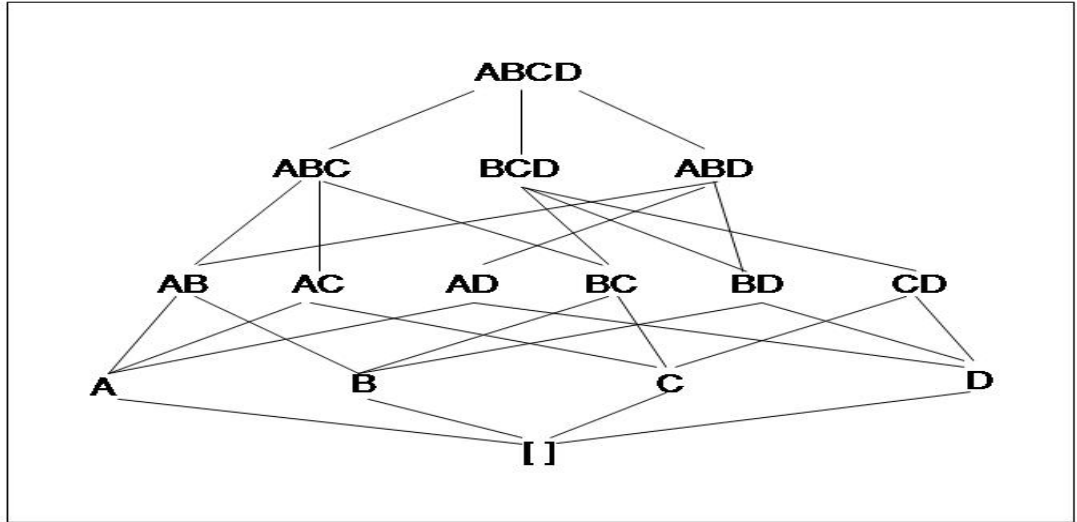


Figure 2.2: Lattice Structure

There can be a total of 2^n views in a lattice for n grouping attributes. Once the lattice structure is ready, [17] runs a greedy approach for picking the best views to materialize. At each step, [17] selects from the lattice a view with the greatest *benefit*, that is, a view that reduces the most the average cost of answering a query, per unit space. As noted earlier, views produced by the algorithm of [17] determine joinless rewritings of queries. Please refer to Appendix C for our implementation of the algorithm.

2.4 Problem Statement

The problem can be stated as follows: Given a database and a query workload, return view definitions, which when materialized, would improve the query performance while satisfying a set of constraints. With materialized views, we consider the following constraints:

1. Storage Space:

As storing a materialized view requires a lot of space, we need to set a maximum limit to the amount of space that can be used for storing views. This constraint can be set by the administrator of the database.

2. View Design: The views are designed considering a specific query workload. But this workload may change over time and the views should be redesigned for the new workload.

3. View Maintenance: The views should be maintained as well. In other words, they should be periodically updated depending on the frequency of updates on the base tables.

The focus of this work is to find an optimum set of tables over which views should be materialized under a given space constraint. We have worked on materializing views on just the fact table and [17] have materialized views over the entire set of tables. The focus of this thesis is on materializing views over a subset of the entire set of tables thereby reducing the time for view generation and the cost of updating the views.

2.4.1 Considerations while choosing the set of tables

While selecting the set of tables over which to materialize views there are a number of considerations. First, the set should contain the fact table. This is because of the way the star schema is designed. If we have a view over two or more dimension tables, we will not have a joining argument and so will not be possible. Second, the view might not contain all the arguments from the tables. It will in most cases contain only some

of the arguments from the chosen set of tables. We need to make sure that the view contains at least the arguments which are used by the base tables to join with other tables not chosen to be materialized. Also if there is an aggregated argument from one of these tables, then it should be in the view as well, otherwise the view cannot be used to answer the query. The other important point to note is that the view should also contain all the arguments from the chosen set of tables, that are also in the **WHERE** clause of the query.

Consider the following query:

```
SELECT
    l.orderkey,
    sum(l.extendedprice),
    sum(o.totalprice),
    o.orderdate,
    o.shippriority
FROM
    lineitem l,
    orders o,
customer c,
supplier s,
part p

WHERE
    l.returnflag = 'R'
AND l.shipdate >= '1995-03-15'
AND l.shipdate <= '1996-12-01'
    AND l.orderkey = o.orderkey
AND c.custkey = o.custkey
AND l.suppkey = s.suppkey
AND l.partkey = p.partkey
```

GROUP BY

```
l.orderkey,  
o.orderdate,  
o.shippriority;
```

Also consider that we are materializing views over the lineitem and the orders table. We will thus have to include all the arguments from the lineitem and the orders table that are used to join with other tables, such as o.custkey used to join with the customer table and l.partkey used to join with the part table. We will also need all the arguments in the GROUP BY clause, i.e l.orderkey, o.orderdate and o.shippriority. Besides these we check for the WHERE clause of the query which has conditions to constant values. In this example, l.returnflag and l.shipdate are the two arguments in the WHERE clause which have conditions with constant values.

2.5 Related Work

Designing and using materialized views for answering queries has long been an active research field. [16] presents a survey on answering queries using views. [10, 6, 11] have also shown some practical solutions to the problem using materialized views and indexes. [8, 9, 5] study the problem for data warehouse in particular. [13, 21] have worked on answering each query using a single view. [6, 7] also present a generic system architecture for answering queries using materialized views and indexes. We have already discussed the BPUS algorithm [17] for designing views to answer aggregate queries.

Chapter 3

System Architecture

As discussed earlier the QPET system was implemented for designing and using materialized views for answering queries. QPET is based on an open-source relational database system PostgreSQL version 7.3.4 [20]. The current version of QPET is available online [12]. QPET framework is designed within the general architecture of Figure 3.1. We will describe the system architecture in detail here.

There are two modules within the QPET Framework. The first module is used for view design, whereas the second module is used for answering user queries in terms of views.

3.1 Designing Views in the QPET Framework

The view generator, as shown in Figure 3.2 generates the view definitions of efficient views based on our implementation of the greedy algorithm. Please refer to Appendix C for a detailed explanation on the working of this module.

In the design stage, the first thing to do is to delineate the search space of views that will be considered for materialization. This function is performed by the *view format manager* in Figure 3.2, which determines the suggested format of views and rewritings and the related search space of candidate views based on the type of the workload queries and on other requirements. The *view generator* in Figure 3.2 considers one by one the elements of the resulting search space of candidate views and passes

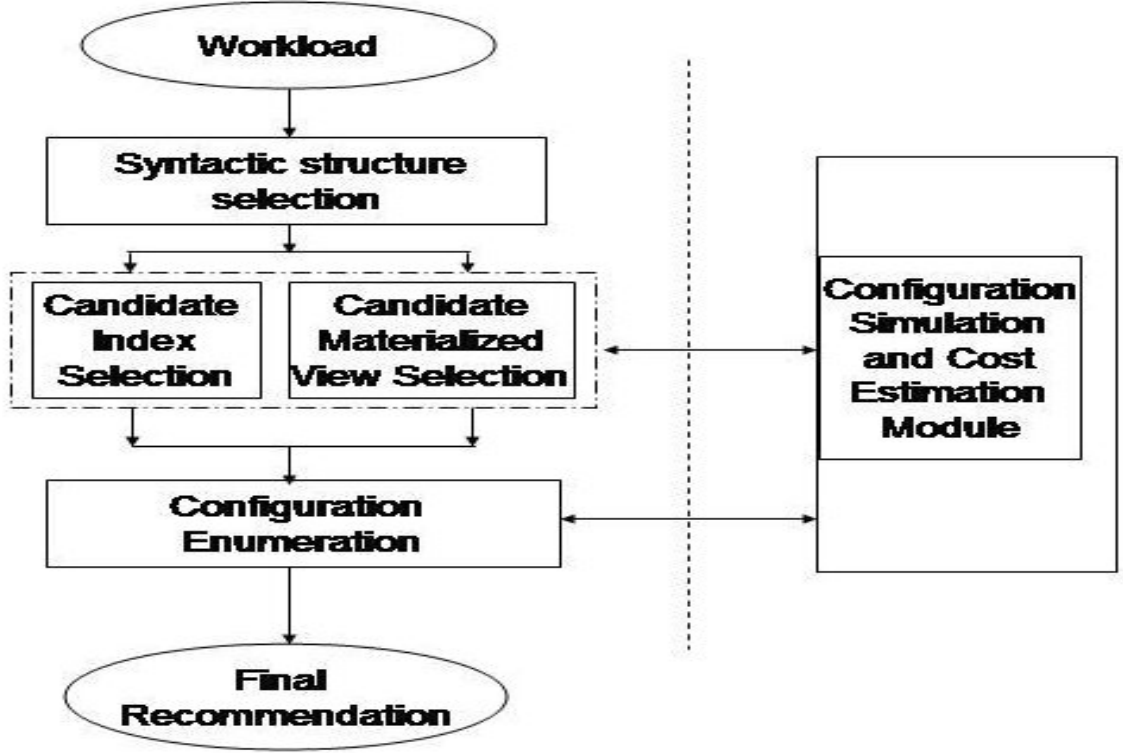


Figure 3.1: Overall system architecture [6, 7]

those of the views that satisfy the input constraints (e.g., the given storage limit) on to the next module. That module, *configuration manager* in Figure 3.2, puts together certain candidate views and tests the potential of the resulting view configurations to improve the evaluation performance of the workload queries. The tests are performed by an *enhanced query optimizer*, which estimates the costs of answering the workload queries using the configurations. We use the architecture of [6] to obtain query-cost estimates without materializing the views in the candidate configurations. In the process of estimating the query costs, the enhanced optimizer considers potential rewritings of the workload queries *within* the process of considering query plans for any single query, in the manner of [10]. After the set of constraints are satisfied, typically after several iterations of designing and testing views and view configurations, the views in the best configuration are materialized in the system, as new stored relations computed on the base relations.

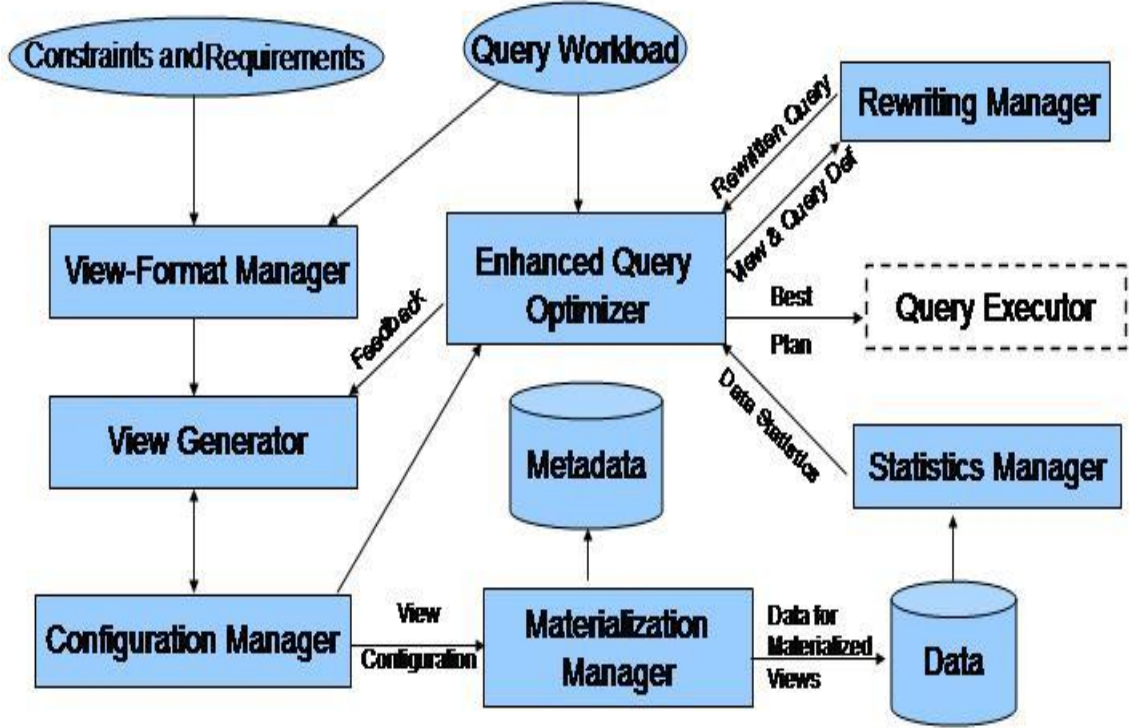


Figure 3.2: Designing derived data in QPET

3.2 Using views in the QPET framework

Once the views proposed by the view generator module are materialized the enhanced query optimizer considers them when constructing the query plan, for user queries. The enhanced query optimizer works with the rewriting module for executing the query. The query input by the user is in terms of base relations. The view generator module is shown in Figure 3.3

Figure 3.3 highlights those modules in the QPET framework that are active at the stage of *using* materialized views to answer user queries. Provided the system chooses to maximize the evaluation performance of a given input query using derived data, the process consists of two steps. First, QPET finds a “good” query plan while taking the materialized views into account, by using the enhanced query optimizer and the query-rewriting subroutine in essentially the same way the modules are used in the view-design

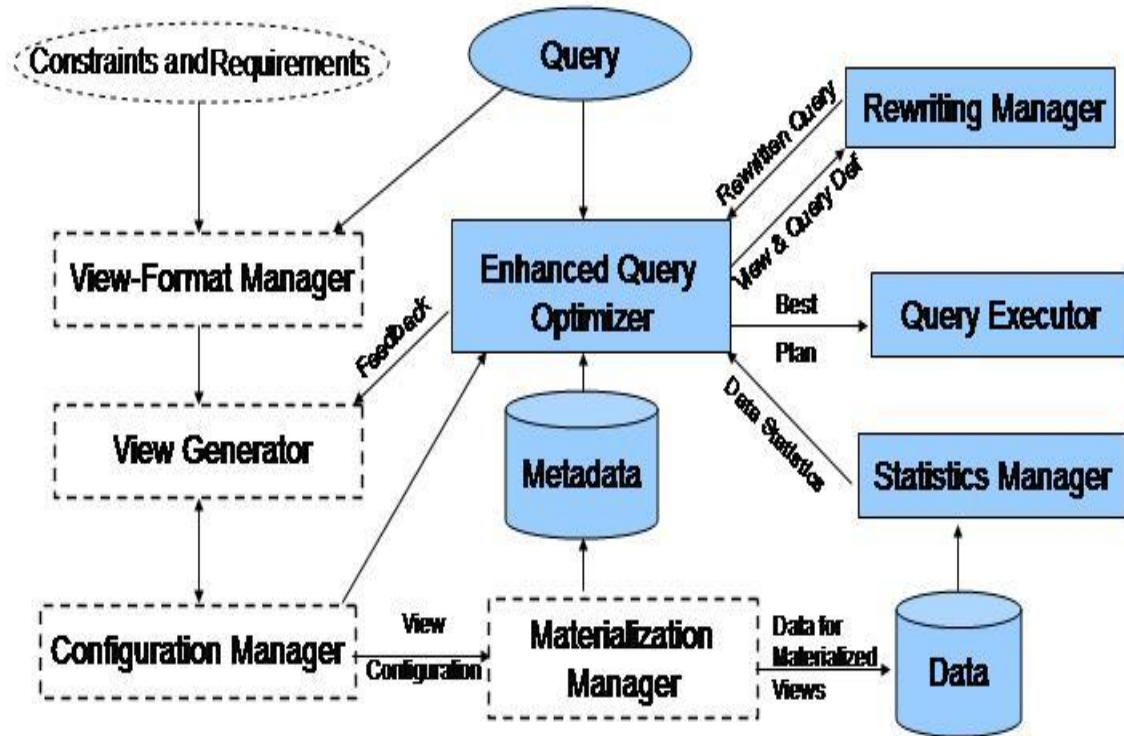


Figure 3.3: Using derived data in QPET

phase. Depending on how appropriate the currently materialized views are for answering the query, the resulting query plan may or may not refer to stored view relations. QPET then passes the resulting query plan to the query executor, which obtains an answer to the query in a standard way using the base relations and the materialized views.

Chapter 4

View Selection

4.1 Introduction

In this chapter, we focus on the problem of view selection. As discussed earlier, we are given a database, a workload of queries and a set of constraints over the database. The objective is to design a set of views to materialize so that the workload of queries can be answered efficiently, given the space and time constraints. Note that all the rewritings we obtain should be equivalent to the respective input queries, which means that we get the same result using the view as we get using the base tables — that is, there is no loss of information when using the materialized view to answer the query. For this thesis, we have considered the following two constraints:

1. Time required for designing the views.
2. Space required for materializing the views.
3. View maintenance cost.

The main intuition while selecting views is that views should be designed to efficiently answer all the queries in the workload and not individual queries from the workload.

4.2 Our Approach

We have developed our approach to design views over a subset of tables from the entire schema. As pointed out earlier, query-performance tuning is a tradeoff between the amount of system resources spent on designing views and the degree of resulting improvement in query performance. As shown in our experimental results in Chapter 6, the time spent in designing views increases as the number of tables in the FROM clause of the views increases. At the same time, we get better query response times from views which cover more base tables. In our QPET framework, the focus is on using specialized algorithms for designing and using derived data to reduce the evaluation costs for each specific class of queries. We consider the following three classes of queries with aggregation and without self-joins as mentioned in [4]:

1. *Workload of queries that aggregate the same tables.* All queries in a query workload aggregate the same tables, $T_1 \dots T_k$, if the aggregated arguments of each workload query come from same tables in $T_1 \dots T_k$. In this case, we refer to the tables that provide the arguments of aggregation for all the workload queries as the *central tables* for the workload. Our approach can handle more than one central table.
2. *Workload of queries that have the same join conditions.* A workload of queries that have the same join conditions is a workload of queries that aggregate the same tables, such that an additional condition is satisfied: For any pair of queries in the workload, if the queries share the same tables in the FROM clause then they share, in the WHERE clause, all the join conditions between the tables. As an illustration, the same-join condition is satisfied in workloads of queries where all joins are natural joins.
3. *Workload of star-schema queries.* A workload of star-schema queries is a workload of queries that have the same join conditions, such that two additional conditions are satisfied. First, the database schema is a star schema [18], with a designated fact table and dimension tables, such that each join of the fact table with a dimension table is on attributes that satisfy a referential-integrity constraint from

the fact table to the dimension table. Second, the fact table is the central table for the workload (i.e., each query aggregates an argument of the fact table). Again, as pointed out earlier, we can also handle more than one central table.

We now discuss the main steps taken to select the set of views to materialize for a given query workload:

1. Stage One:

Select a set of tables over which we can materialize the views. As discussed in our work [4], we have considered the case of materializing a view over just the fact table of the star schema. The BPUS approach of [17] has considered the case of materializing a view over the entire set of tables. The focus of this thesis is to materialize views over only a subset of tables so that we can achieve an optimum balance between the space required to store a materialized view along with the time to design the views vs the time taken to answer the query. As shown in Chapter 6, we have achieved this balance by materializing views over a subset of tables. We suggest that the space and time constraints could be a configurable value in any commercial database, where depending on the space availability and the frequency of a given query, views can be materialized over varying sets of tables. Note that in a star schema, the fact table will always be selected in the set of tables to materialize as the aggregated attribute is mostly from the fact table. As opposed to this, a snowflake schema, as defined in Chapter 2, can have aggregation from other tables as well, in which case the views will be designed at least over all the tables which supply the aggregation arguments to the query workload.

2. Stage Two:

Construct a lattice as described in [17], of all the possible views. As discussed in Chapter 2, the lattice is a representation of the entire search space of views for the input query workload. Possible views are formed by considering all subsets of the grouping arguments in the lattice. The arguments considered while creating this lattice are taken from the input query workload. They are the following arguments:

- (a) Arguments in the **GROUP BY** clause of the query
- (b) Arguments used to join with tables not included in the view
- (c) Arguments used in the **WHERE** clause with condition to constant values.

Please refer to Appendix C on the steps taken to create a lattice. Note that the arguments considered for creating a lattice depend on the set of tables chosen to materialize the view. In our QPET system, this is done by the pre-processing module in the view format manager. We need to make sure that all arguments listed above are chosen. Also the views designed are parameterized, to answer any instance of the query, which means it does not include comparisons with constants in the view definitions. A parameterized view has placeholders instead of constants. The parameters can be replaced at runtime.

3. Stage Three:

Once we have created the lattice, we run the *generatelattice* command to select the best views to materialize. The *generatelattice* command is our implementation of the greedy algorithm BPUS [17]. For more details about the implementation of the *generatelattice* command please refer to Appendix C.3. The *generatelattice* command returns the best view to materialize given the input queries and the space constraint.

4. Stage Four: After a set of views is obtained, the input query can be rewritten in terms of the materialized view rather than the base tables. This is discussed more in Chapter 5 on query rewriting.

Chapter 5

Query Rewriting

5.1 Introduction

In this chapter we discuss how to rewrite a query using a materialized view. In the previous chapters we have discussed how to obtain the best views to materialize. Once we have the best views to materialize, and we have actually materialized the views in the database, they can be used to answer the queries. Once a view is materialized, it becomes a relation stored in the database similar to a base table.

Note that all our rewritings are equivalent to the input queries: each original query and its rewriting return the same results on all databases.

5.2 Steps for Query Rewriting

5.2.1 Preprocessing an input query

The pre-processing step scans an input query to get a list of attributes that are required in the view. The list of attributes depends on the tables over which views are being designed, in other words, it depends on the tables that we wish to replace with a view. As pointed out earlier, in our approach we can replace any combination of tables with a view. The required attributes are essentially the attributes in the `GROUP BY` clause of the query and the attributes that are used in the `WHERE` clause with either constant values,

or to join with other tables. The pre-processor is executed at the stage of designing views. Please refer to Appendix C.2 for the implementation details.

5.2.2 Postprocessing an input query

Once the views are designed, the postprocessor selects the view which will be used to rewrite a given input query. The postprocessor is implemented in the rewriting manager in our QPET framework. To rewrite a query using a view we first of all need to check if any view exists that can be used to rewrite the query efficiently. To rewrite a query, we start by selecting a view from the set of all materialized views. The set of materialized views is obtained as discussed in the previous chapter. From this set of views, we need to match the attributes in the views with those required by the preprocessing step. Only those views with all the required attributes can be considered for rewriting. The postprocessor picks the cheapest view which has all the required attributes.

As part of rewriting, there may also be a need for re-aggregating the view. This happens when the grouping arguments in the view are a proper superset of the arguments in the query. In this case we need to re-aggregate the already aggregated attribute in the view. This is explained with the help of an example in Section 5.2.3.

5.2.3 Example rewriting

Consider the *Pricing Summary Report Query (Q1)* from our modified TPC-H queries.

1. Objective: Reports the amount of business that was billed, shipped, and returned.
2. Modified query Q1 for the experiments, in SQL:

```
SELECT
    l_returnflag,
    l_linestatus,
    SUM(l_extendedprice),
    COUNT(*)
```

```

FROM
    lineitem
WHERE
    l_shipdate = date '[DATE]'
GROUP BY
    l_returnflag,
    l_linestatus;

```

3. Modified query Q1 for the experiments, in Datalog:

$q_1(RF, LS, sum(EX), count) :-$
 $L(OK, PK, SK, LN, QT, EX, DS, TX, RF, LS, 'date', CD, RD, SI, SM, C8).$

If we consider designing views over the fact table, which is the Lineitem table, then the required attributes will be: RF,LS,SD.

4. Consider the view obtained for this rewriting:

In SQL:

```

SELECT returnflag, linestatus, shipdate,
SUM(extendedprice) AS sumex, count(*) AS totalcount
INTO _112
FROM lineitem
GROUP BY returnflag, linestatus, shipdate;

```

In Datalog:

$V112(RF, LS, SD, sum(EX), count(*)) :-$
 $L(OK, PK, SK, LN, QT, EX, DS, TX, RF, LS, SD, CD, RD, SI, SM, C8).$

5. In this case, the query can be rewritten as follows:

In SQL:

```
SELECT returnflag, linestatus, SUM(sumex), SUM(totalcount) AS count
FROM _112
WHERE shipdate = date '1998-12-01'
GROUP BY returnflag, linestatus;
```

In Datalog:

$$Q1(RF, LS, SUM(sumex), SUM(totalcount)) : - \\ V112(RF, LS, '1998 - 12 - 01', sumex, totalcount).$$

Chapter 6

Experimental Setup and Results

In this chapter, we discuss our experimental results in detail. We conducted experiments to evaluate the system architecture and techniques presented earlier in this thesis. All the experiments were done on a machine with a 2.8GHz Intel P4 processor, 512MB RAM, and 80GB of internal hard drive running RedHat Linux 9 with kernel version 2.4.20-8. Our implementation of the QPET framework is written in C and is based on an open-source relational data-management system PostgreSQL [20] version 7.3.4. The current version of QPET is available online [12]; it incorporates support for answering SQL queries using materialized views, as described in [10], and for generating aggregate views using the BPUS algorithm of [17]. We have used the standard TPC-H data [22] and queries as well as the Sky Server Astronomical Database [2] for the purpose of our experiments.

We conducted our experiments in three phases:

1. Phase I:

In phase I, we used the modified TPC-H data and queries. The table sizes used for the TPC-H database are shown in Figure 6.1

In this phase we compared our approach with the BPUS algorithm of [17]. For our approach in this phase of experiments, we chose to design and materialize *fact-table views*, that is, views whose **FROM** clause has just the fact table in the star schema. We considered both simple and range queries and obtained the following

TPC Tables	
Table Name	Size
Lineitem	6,001,215
Part	200,000
Supplier	10,000
PartSupp	800,000
Customer	150,000
Orders	1,500,000
Nation1	25
Nation2	25
Region	5

Figure 6.1: TPC-H table sizes

results.

Figure 6.3 shows the sizes of the views materialized in both cases.

The results clearly validate our approach. Also the space required to store these views is within acceptable limits. The BPUS views give us better response times but the time taken to generate the view lattice and design the views far exceeds the time taken in case of the views designed over the fact table. The number of nodes in the view lattice increase exponentially and so does the computation time. Figure 6.4 shows the time taken for designing the views in the two cases.

2. Phase II:

The results from phase I lead us to experimenting with varying the number of relations in the FROM clause of these views. The results obtained are shown in Figure 6.6. The results clearly show that there is a tradeoff between the time and space needed to compute the view lattice and design the view versus the response time of the query. Figure 6.5 shows the tables considered in each case along with the number of nodes in the search space lattice and the time taken to design the

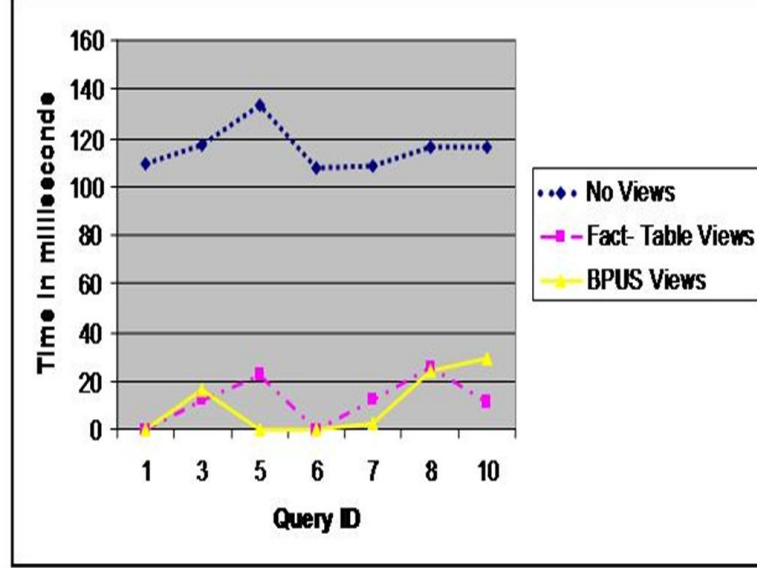


Figure 6.2: Query Runtimes Phase I

views. Figure 6.7 shows the query runtimes obtained from the approach discussed in [19].

3. Phase III:

In the third phase of our experiments, we worked with the Sky Server Astronomical Database [2] Figure 6.8 shows the tables used for the sky server astronomical database. Figure 6.9 shows the lattice information and the time taken to design the views.

Figure 6.10 shows the results obtained from the astronomical database and Figure 6.11 shows the results obtained using the approach discussed in [19]

Query ID	Rows Returned	Fact-Table Views Used		BPUS Views Used	
		View ID	View Size	View ID	View Size
1	1	112	3,817	112	3,817
3	1,281	113	5,927,453	5105	5,986,375
5	5	5	5,999,989	6656	60,150
6	1	120	41,969	120	41,969
7	1	101	6,001,207	6496	1,515,213
8	3	39	6,001,204	7936	5,918,807
9	752,916	39	6,001,204	3074	752,916
10	505	17	2,070,731	5008	4,537,104

Figure 6.3: View and Query Sizes Phase I

	Time to Generate Views (seconds)	Number of Nodes In the Lattice
Fact-Table Lattice	40	128
BPUS Lattice	1,920	8,192

Figure 6.4: Time to Design Views Phase I

	Tables	No. of Nodes	Time to Generate Views (seconds)
FROM 1	Lineitem	128	38
FROM 2	Lineitem, Orders	2048	56
FROM 3-1	Lineitem, Orders, Customer	4096	197
FROM 3-2	Lineitem, Orders, Supplier	4096	197
FROM 4-1	Lineitem, Orders, Customer, Supplier	8192	1836
FROM 4-2	Lineitem, Orders, Customer, Part	2048	56
FROM ALL	Lineitem, Orders, Customer, Part, Supplier	65536	8900

Figure 6.5: Lattice Attributes and View Design Time Phase II

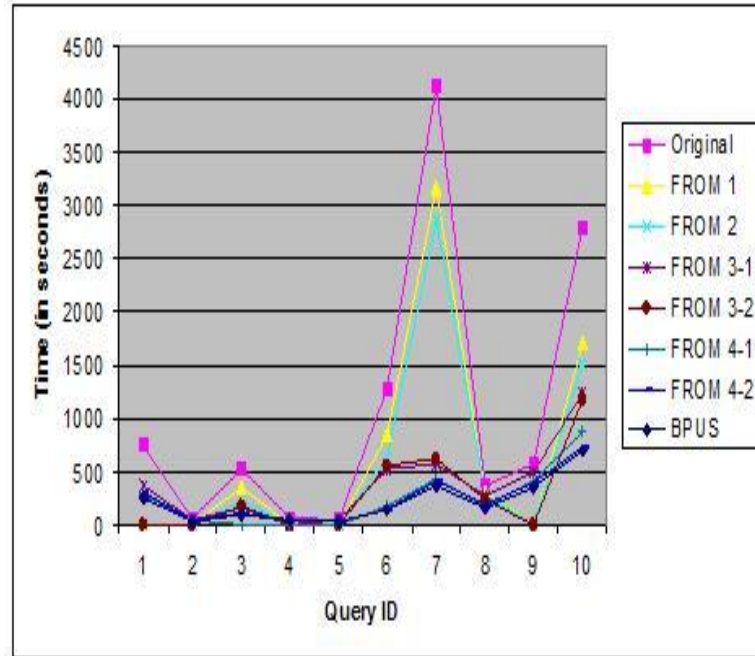


Figure 6.6: Query Runtimes Phase II

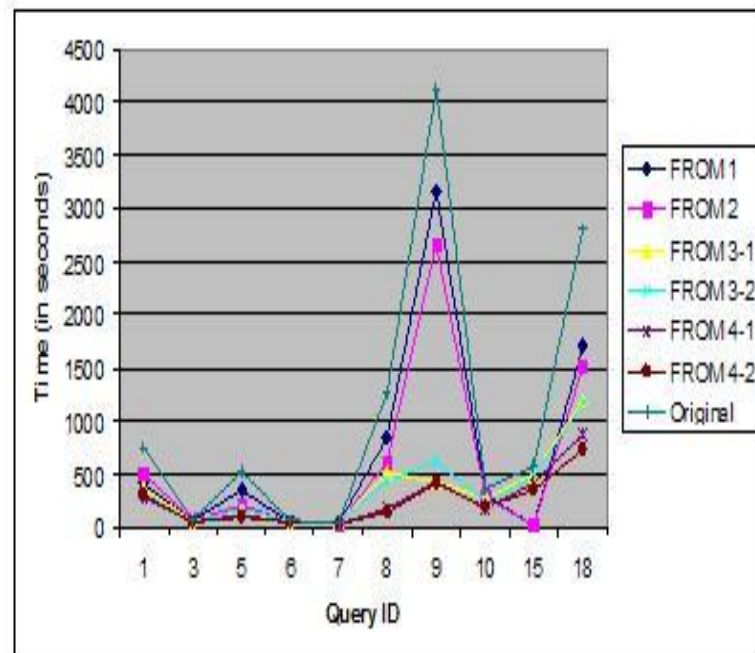


Figure 6.7: Query Runtimes Phase II [19]

Table Name	Size
SpecObjAll	511209
PhotoObj	11913
Plate	10002
Segment	90
Field	74
Neighbors	64

Figure 6.8: Sky Server Table Sizes

	Tables	No. of Nodes	Time to Generate Views (seconds)
Lattice A	SpecObjAll	128	38
Lattice B	SpecObjAll, Plate	128	40
Lattice C	SpecObjAll, PhotoObj	512	123
Lattice D	SpecObjAll, PhotoObj, Plate	512	123
Lattice E	SpecObjAll, PhotoObj, Field	1024	210
Lattice F	SpecObjAll, PhotoObj, Field, Segment	1024	210
Lattice G	SpecObjAll, PhotoObj, Field, Segment, Plate	1024	211

Figure 6.9: Lattice Attributes and View Design Time for Sky Server Database

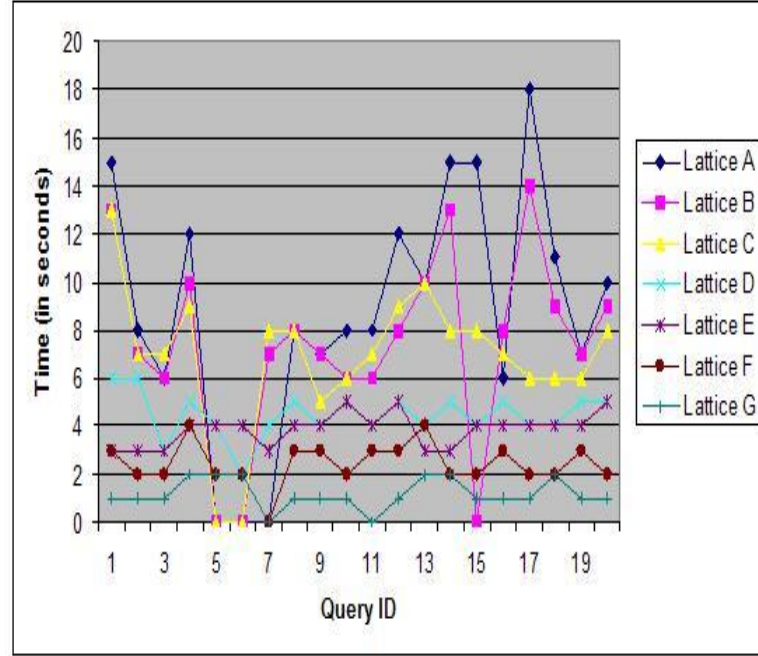


Figure 6.10: Query Runtimes for Sky Server Database

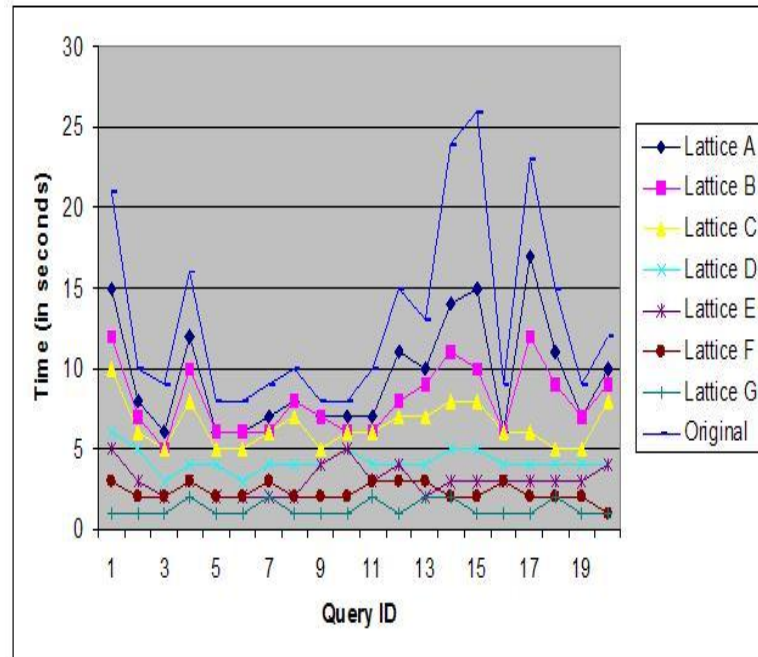


Figure 6.11: Query Runtimes for Sky Server Database [19]

List of References

- [1] Dbms magazine. <http://http://www.dbmsmag.com/9708d15.html/>.
- [2] Sky Server Astronomical Database. <http://www.sdss.org/>.
- [3] F. Afrati and R. Chirkova. Selecting and using views to compute aggregate queries. Proceedings of ICDT, 2005.
- [4] F Afrati, R Chirkova, S Gupta, and Loftis C. Designing and using views to improve performance of aggregate queries. In *Proceedings of The 10th International Conference on Database Systems for Advanced Applications* , 2005.
- [5] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proceedings of VLDB*, pages 506–521, 1996.
- [6] S. Agrawal, S. Chaudhuri, and V.R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *Proceedings of VLDB*, pages 496–505, 2000.
- [7] S. Agrawal, S. Chaudhuri, and V.R. Narasayya. Materialized view and index selection tool for Microsoft SQL Server 2000. In *Proceedings of ACM SIGMOD*, 2001.
- [8] G. Chan, Q. Li, and L. Feng. Design and selection of materialized views in a data warehousing environment: A case study. In *Proceedings of DOLAP*, pages 42–47, 1999.

-
- [9] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, 1997.
 - [10] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of the Eleventh International Conference on Data Engineering (ICDE)*, pages 190–200, 1995.
 - [11] S. Chaudhuri and V.R. Narasayya. AutoAdmin 'What-if' index analysis utility. In *Proc. SIGMOD*, pages 367–378, 1998.
 - [12] R. Chirkova, S. Gupta, K.-H. Kim, and S. Sandhu. Extensible framework for query-performance enhancement by tuning. Code downloads and documentation are available from <http://research.csc.ncsu.edu/selftune/>, 2005.
 - [13] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proceedings of VLDB*, pages 358–369, 1995.
 - [14] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques and applications. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing*, 18(2):3–18, 1995.
 - [15] H. Gupta, V. Harinarayan, A. Rajaraman, and J.D. Ullman. Index selection for OLAP. In *Proceedings of ICDE*, pages 208–219, 1997.
 - [16] Alon Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.
 - [17] V. Harinarayan, A. Rajaraman, and J.D. Ullman. Implementing data cubes efficiently. In *Proc. SIGMOD*, pages 205–216, 1996.
 - [18] R. Kimball and M. Ross. *The Data Warehouse Toolkit (second edition)*. Wiley Computer Publishing, 2002.

- [19] J Li, Z Talebi, R Chirkova, and Fathi Y. A formal model for the problem of view selection for aggregate queries. In *Proceedings of ADIBS, Tallinn, Estonia*, 2005.
- [20] PostgreSQL. Open source database-management system. <http://www.postgresql.org/>.
- [21] D. Srivastava, S. Dar, H.V. Jagadish, and A.Y. Levy. Answering queries with aggregation using views. In *Proc. VLDB*, pages 318–329, 1996.
- [22] TPC-H:. TPC Benchmark H (Decision Support). Available from <http://www.tpc.org/tpch/spec/tpch2.1.0.pdf>.

Appendix A

TPC-H Benchmarking Data

A.1 Introduction

The TPC-H benchmarking data has been used for all our experiments. It can be downloaded from [22] (We would like to thank Jia Li from University of California, Irvine for her help in setting up the TPC-H data)

A.1.1 CREATE Statements

The following SQL Statement were used to create tables in Postgresql to load the TPC-H data.

1. create table region (regionkey text, name char(25), comment varchar(152), primary key (regionkey));
2. create table nation (nationkey text, name char(25), regionkey text references region(regionkey), comment varchar(152),primary key (nationkey));
3. create table part (partkey text, name varchar(55), mfgr char(25),brand char(10), type varchar(25),size int, container char(10), retailprice decimal, comment varchar(23),primary key (partkey));
4. create table supplier (suppkey text, name char(25), address varchar(40), nationkey text references nation(nationkey), phone char(15),acctbal decimal, comment

- varchar(101), primary key (suppkey));
5. create table partsupp (partkey text references part(partkey), suppkey text references supplier(suppkey), availqty int, supplycost decimal, comment varchar(199), primary key (partkey, suppkey));
 6. create table customer (custkey text, name varchar(25), address varchar(40), nationkey text references nation(nationkey), phone char(15), acctbal decimal, mktsegment char(10), comment varchar(117), primary key (custkey));
 7. create table orders (orderkey text, custkey text references customer(custkey), orderstatus char(1), totalprice decimal, orderdate date, orderpriority char(15), clerk char(15), shippriority int, comment varchar(79), primary key (orderkey));
 8. create table lineitem (orderkey text references orders(orderkey), partkey text references part(partkey), suppkey text references supplier(suppkey), linenumber integer, quantity decimal, extendedprice decimal, discount decimal, tax decimal, returnflag char(1), linestatus char(1), shipdate date, commitdate date, receiptdate date, shipinstruct char(25), shipmode char(10), comment varchar(44), primary key (orderkey, linenumber));

A.2 Queries used for the Experiments

A.2.1 Pricing Summary Report Query (Q1)

1. Objective: Reports the amount of business that was billed, shipped, and returned.
2. Modified query Q1 for the experiments, in SQL:

```
SELECT
    l.returnflag,
    l.linestatus,
    SUM(l.extendedprice),
```

```

COUNT(l.orderkey)
FROM
    lineitem l,
    orders o,
    customer c,
    supplier s,
    part p

WHERE
    l.shipdate >= '1995-03-15'
AND l.shipdate <= '1998-12-01'
    AND l.orderkey = o.orderkey
AND c.custkey = o.custkey
AND l.supkey = s.supkey
AND l.partkey = p.partkey

GROUP BY
    l.returnflag,
    l.linestatus;

```

3. Modified query Q1 for the experiments, in Datalog:

$q_1(RF, LS, sum(EX), count(OK))$:-

$L(OK, PK, SK, LN, QT, EX, DS, TX, RF, LS, ' [date]', CD, RD, SI, SM, C8),$
 $O(OK, CK, OS, TR, OD, OP, CL, SP, C7),$
 $C(CK, N4, A4, NK2, P4, B4, MS, C4),$
 $S(SK, N2, A2, NK1, P2, B2, C2),$
 $P(PK, N1, MF, BR, TP, SZ, CN, RT, C1).$

A.2.2 Shipping Priority Query (Q3)

1. Objective: Retrieves the ten unshipped orders with the highest value.
2. Modified query Q3 for the experiments, in SQL:

```
SELECT
    l.orderkey,
    sum(l.extendedprice),
    sum(o.totalprice),
    o.orderdate,
    o.shippriority
FROM
    lineitem l,
    orders o,
    customer c,
    supplier s,
    part p

WHERE
    l.returnflag = 'R'
AND l.shipdate >= '1995-03-15'
AND l.shipdate <= '1996-12-01'
    AND l.orderkey = o.orderkey
AND c.custkey = o.custkey
AND l.suppkey = s.suppkey
AND l.partkey = p.partkey

GROUP BY
    l.orderkey,
    o.orderdate,
```

o.shippriority;

3. Modified query Q3 for the experiments, in Datalog:

$q_3(OK, OD, SP, sum(EX), sum(TP))$:-

$L(OK, PK, SK, LN, QT, EX, DS, TX, '[rf]', LS, '[sd]', CD, RD, SI, SM, C8),$
 $O(OK, CK, OS, TR, OD, OP, CL, SP, C7),$
 $C(CK, N4, A4, NK2, P4, B4, MS, C4),$
 $S(SK, N2, A2, NK1, P2, B2, C2),$
 $P(PK, N1, MF, BR, TP, SZ, CN, RT, C1).$

A.2.3 Local Supplier Volume Query (Q5)

1. Objective: Lists the revenue volume done through local suppliers.
2. Modified query Q5 for the experiments, in SQL:

```
SELECT
    n.name,
    SUM(l.extendedprice),
    SUM(o.shippriority)
FROM
    customer c,
    orders o,
    lineitem l,
    supplier s,
    part p,
    nation n,
    region r
WHERE
    o.orderdate < '1995-01-01'
    AND o.orderdate > '1992-12-31'
```

```
AND r.name = 'AFRICA'
AND c.custkey = o.custkey
AND l.orderkey = o.orderkey
AND l.supkey = s.supkey
AND s.nationkey = n.nationkey
AND n.regionkey = r.regionkey
AND l.partkey = p.partkey

GROUP BY
    n.name;
```

3. Modified query Q5 for the experiments, in Datalog:

$q_5(N51, SUM(EX), SUM(SP))$

$L(OK, PK, SK, LN, QT, EX, DS, TX, RF, LS, SD, CD, RD, SI, SM, C8),$

$O(OK, CK, OS, TR, 'date', OP, CL, SP, C7),$

$C(CK, N4, A4, NK2, P4, B4, MS, C4),$

$S(SK, N2, A2, NK1, P2, B2, C2),$

$P(PK, N1, MF, BR, TP, SZ, CN, RT, C1),$

$Nat1(NK1, N51, RK, C51),$

$R(RK, 'region', C6).$

A.2.4 Forecasting Revenue Change Query (Q6)

1. Objective: Quantifies the amount of revenue increase that would have resulted from eliminating certain companywide discounts in a given percentage range in a given year. Asking this type of “what if” query can be used to look for ways to increase revenues.
2. Modified query Q6 for the experiments, in SQL:

```
SELECT
```

```
        o.orderkey,  
        l.linestatus,  
        SUM(l.extendedprice)  
FROM  
        lineitem l,  
        orders o,  
customer c,  
supplier s,  
part p  
  
WHERE  
        l.shipdate < '1992-12-12'  
        AND l.discount >= 0.01  
        AND l.discount <= 0.05  
        AND l.returnflag = 'R'  
        AND l.orderkey = o.orderkey  
        AND c.custkey = o.custkey  
        AND l.supkey = s.supkey  
        AND l.partkey = p.partkey  
  
GROUP BY  
        l.linestatus,  
        o.orderkey;
```

3. Modified query Q6 for the experiments, in Datalog:

$$q_6(OK, SUM(EX)) \quad :-$$
$$L(OK, PK, SK, LN, QT, EX, ' [ds]', TX, ' [rf]', LS, ' [date]', CD, RD, SI, SM, C8),$$

$O(OK, CK, OS, TR, OD, OP, CL, SP, C7),$
 $C(CK, N4, A4, NK2, P4, B4, MS, C4),$
 $S(SK, N2, A2, NK1, P2, B2, C2),$
 $P(PK, N1, MF, BR, TP, SZ, CN, RT, C1).$

A.2.5 Volume Shipping Query (Q7)

1. Objective: Determines the value of goods shipped between certain nations to help in the renegotiation of shipping contracts.
2. Modified query Q7 for the experiments, in SQL:

```
SELECT
    n1.nationkey AS supp_nation,
    n2.nationkey AS cust_nation,
    SUM(l.extendedprice) AS revenue,
    SUM(o.totalprice)
FROM
    lineitem l,
    orders o,
    customer c,
    supplier s,
    part p,
    nation n1,
    nation n2
WHERE
    n1.name = 'CHINA'
    AND n2.name = 'GERMANY'
    AND l.shipdate >='1995-01-01'
    AND l.shipdate <= '1999-12-12'
    AND s.supkey = l.supkey
```

```
AND o.orderkey = l.orderkey
AND c.custkey = o.custkey
AND l.partkey = p.partkey
AND s.nationkey = n1.nationkey
AND c.nationkey = n2.nationkey
```

```
GROUP BY
    supp_nation,
    cust_nation;
```

3. Modified query Q7 for the experiments, in Datalog:

```
q7(NK1, NK2, SUM(EX), SUM(TP))
    L(OK, PK, SK, LN, QT, EX, DS, TX, RF, LS, '[date]', CD, RD, SI, SM, C8),
    O(OK, CK, OS, TR, OD, OP, CL, SP, C7),
    C(CK, N4, A4, NK2, P4, B4, MS, C4),
    S(SK, N2, A2, NK1, P2, B2, C2),
    P(PK, N1, MF, BR, TP, SZ, CN, RT, C1),
    Nat1(NK1, '[nation1]', RK1, C51),
    Nat2(NK2, '[nation2]', RK, C52).
```

A.2.6 National Market Share Query (Q8)

1. Objective: Determines how the market share of a given nation within a given region has changed over two years for a given part type.
2. Modified query Q8 for the experiments, in SQL:

```
SELECT
    n1.name,
    n2.name,
```

```
SUM(l.extendedprice),
SUM(o.shippriority)
FROM
    part p,
    supplier s,
    lineitem l,
    orders o,
    customer c,
    nation n1,
    nation n2,
    region r
WHERE
    r.name = 'ASIA'
    AND s.nationkey = n1.nationkey
    AND o.orderdate >= '1995-01-01'
    AND o.orderdate < '1997-12-30'
    AND p.type = 'ECONOMY BURNISHED TIN'
    AND p.partkey = l.partkey
    AND s.supkey = l.supkey
    AND l.orderkey = o.orderkey
    AND o.custkey = c.custkey
    AND c.nationkey = n2.nationkey
    AND n1.regionkey = r.regionkey

GROUP BY
    n1.name,
    n2.name;
```

3. Modified query Q8 for the experiments, in Datalog:

$q_s(NK1, NK2, SUM(EX), SUM(TP))$
 $L(OK, PK, SK, LN, QT, EX, DS, TX, RF, LS, SD, CD, RD, SI, SM, C8),$
 $O(OK, CK, OS, TR, '[date]', OP, CL, SP, C7),$
 $C(CK, N4, A4, NK2, P4, B4, MS, C4),$
 $S(SK, N2, A2, NK1, P2, B2, C2),$
 $P(PK, N1, MF, BR, '[type]', SZ, CN, RT, C1),$
 $Nat1(NK1, N51, RK1, C51),$
 $Nat2(NK2, N52, RK, C52),$
 $R(RK, '[region]', C6).$

A.2.7 Product Type Profit Measure Query (Q9)

1. Objective: Determines how much profit is made on a given line of parts, broken out by supplier nation and year.
2. Modified query Q9 for the experiments, in SQL:

```

SELECT
    n.name,
    p.name,
    SUM(l.extendedprice),
    SUM(o.shippriority)
FROM
    lineitem l,
    orders o,
    customer c,
    supplier s,
    part p,
    nation n
WHERE
    s.supkey = l.supkey

```

```
AND p.partkey = l.partkey
AND o.orderkey = l.orderkey
AND s.nationkey = n.nationkey
AND o.custkey = c.custkey
```

```
GROUP BY
  n.name,
  p.name;
```

3. Modified query Q9 for the experiments, in Datalog:

```
q9(N51, N1, SUM(EX), SUM(TP))
  L(OK, PK, SK, LN, QT, EX, DS, TX, RF, LS, SD, CD, RD, SI, SM, C8),
  O(OK, CK, OS, TR, OD, OP, CL, SP, C7),
  C(CK, N4, A4, NK2, P4, B4, MS, C4),
  S(SK, N2, A2, NK1, P2, B2, C2),
  P(PK, N1, MF, BR, TP, SZ, CN, RT, C1),
  Nat1(NK1, N51, RK1, C51).
```

A.2.8 Returned Item Reporting Query (Q10)

1. Objective: Identifies customers who might be having problems with the parts that are shipped to them.
2. Modified query Q10 for the experiments, in SQL:

```
SELECT
  c.custkey,
  c.name,
  c.acctbal,
  n.name,
  c.address,
```

```
        c.phone,  
        SUM(l.extendedprice)  
  
FROM  
        lineitem l,  
        orders o,  
        customer c,  
        supplier s,  
        part p,  
        nation n  
  
WHERE  
        c.custkey = o.custkey  
        AND l.orderkey = o.orderkey  
        AND o.orderdate >= '1991-01-01'  
        AND o.orderdate <= '1996-12-12'  
        AND l.returnflag = 'R'  
        AND c.nationkey = n.nationkey  
        AND p.partkey = l.partkey  
        AND s.supkey = l.supkey  
  
GROUP BY  
        c.custkey,  
        c.name,  
        c.acctbal,  
        c.phone,  
        n.name,  
        c.address;
```

3. Modified query Q10 for the experiments, in Datalog:

$q_{10}(CK, N4, B4, N51, A4, P4, SUM(EX))$
 $L(OK, PK, SK, LN, QT, EX, DS, TX, 'rf', LS, SD, CD, RD, SI, SM, C8),$
 $O(OK, CK, OS, TR, 'date', OP, CL, SP, C7),$
 $C(CK, N4, A4, NK2, P4, B4, MS, C4),$
 $S(SK, N2, A2, NK1, P2, B2, C2),$
 $P(PK, N1, MF, BR, TP, SZ, CN, RT, C1),$
 $Nat1(NK1, N51, RK1, C51).$

A.2.9 Top Supplier Query (Q15)

1. Objective: Determine the top supplier so it can be rewarded, given more business, or identified for special recognition.
2. Modified query Q15 in SQL:

```

SELECT
    l.supkey,
    sum(l.extendedprice)

FROM
    lineitem l,
    orders o,
    customer c,
    supplier s,
    part p

WHERE
    l.shipdate >= '1994-01-01'
    AND l.shipdate < '1996-12-12'
    AND c.custkey = o.custkey
    AND o.orderkey = l.orderkey

```

AND p.partkey = l.partkey

AND s.supkey = l.supkey

GROUP BY

l.supkey;

3. Modified query Q15 in Datalog:

$q_{15}(SK, SUM(EX)) \quad :-$

$L(OK, PK, SK, LN, QT, EX, DS, TX, RF, LS, ' [date]', CD, RD, SI, SM, C8),$

$O(OK, CK, OS, TR, OD, OP, CL, SP, C7),$

$C(CK, N4, A4, NK2, P4, B4, MS, C4),$

$S(SK, N2, A2, NK1, P2, B2, C2),$

$P(PK, N1, MF, BR, TP, SZ, CN, RT, C1).$

A.2.10 Large Volume Customer Query (Q18)

1. Objective: Find a list of the top 100 customers who have ever placed large quantity orders.
2. Modified query Q18 in SQL:

SELECT

c.name,

c.custkey,

o.orderkey,

o.orderdate,

o.totalprice,

sum(l.quantity)

FROM

customer c,

```
orders o,
lineitem l,
supplier s,
part p
```

WHERE

```
o.orderkey > 1000
AND o.orderkey < 5000
AND c.custkey = o.custkey
AND o.orderkey = l.orderkey
AND p.partkey = l.partkey
AND s.supkey = l.supkey
```

GROUP BY

```
c.name,
c.custkey,
o.orderkey,
o.orderdate,
o.totalprice;
```

3. Modified query Q18 in Datalog:

$$\begin{aligned}
 q_{18}(N4, CK, OK, OD, TR, \neg SUM(QT)) \\
 & C(CK, N4, A4, NK1, P4, B4, MS, C4), \\
 & O('ok', CK, OS, TR, OD, OP, CL, SP, C7), \\
 & L(OK, PK, SK, LN, QT, EX, DS, TX, RF, LS, SD, CD, RD, SI, SM, C8). \\
 & S(SK, N2, A2, NK1, P2, B2, C2), \\
 & P(PK, N1, MF, BR, TP, SZ, CN, RT, C1).
 \end{aligned}$$

Appendix B

Sky Sever Astronomical Data

For the purpose of our experiments, we worked with modified sky server astronomical database [2].

B.1 Database details

The modified database we used had the following tables:

1. SpecObjAll : 511209 tuples
2. PhotoObj : 11913 tuples
3. Plate : 10002 tuples
4. Field : 74 tuples
5. Segment : 90 tuples
6. Neighbors : 64 tuples

B.2 Queries used for the experiments

Q0:

```
SELECT *
```

```
FROM
specobjall,plate, photoobj, field, segment
WHERE
specobjall.plateid = plate.plateid AND
specobjall.objid = photoobj.objid AND
photoobj.fieldid = field.fieldid AND
field.segmentid = segment.segmentid;
```

Q1:

```
SELECT N.neighbortype, primtarget, max(zerr) from Q0, Neighbor N
WHERE N.objid = Q0(P0).objid AND zstatus = 1 AND specclass >= 1
AND specclass <=3
AND sprerun = 1 AND ngalaxy > 3
GROUP BY N.neighbortype, primtarget;
```

Q2:

```
SELECT zwarning, ngalaxy, count(*)
FROM Q0
WHERE sectarget > 200 AND
Q0(P0).type = 'GALAXY'
GROUP BY zwarning, ngalaxy;
```

Q3:

```
SELECT zstatus,zwarning,sprerun, max(veldisperr)
FROM Q0
WHERE specclass <=3
GROUP BY zstatus, zwarning,sprerun;
```

Q4:

```
SELECT N.type,N.neighbortype, ngalaxy, primtarget, max(zerr)
```

```
FROM Q0, Neighbor N
WHERE N.objid = P0.objid
AND camcol > 2
AND camcol < 5
AND sectarget = 3
GROUP BY N.type, N.neighbortype,ngalaxy, primtarget;
```

Q5:

```
SELECT specclass, zwarning, sprerun, P0.type,ngalaxy, max(veldisperr)
FROM Q0
WHERE zstatus = z
AND primtarget < 200
AND camcol = 6
GROUP BY specclass, zwarning, sprerun, P0.type, ngalaxy;
```

Q6:

```
SELECT specclass, zwarning, sprerun, P0.type, max(veldisperr)
FROM Q0
WHERE zstatus = z
AND primtarget < 200
AND camcol = 6
GROUP BY specclass, zwarning, sprerun, P0.type;
```

Q7:

```
SELECT specclass, zwarning, sprerun, N.neighbortype, P0.type, max(veldisperr)
FROM Q0, Neighbor N
WHERE N.objid = P0.objid
AND zstatus = z
AND primtarget < 200
AND camcol = 6
GROUP BY specclass, zwarning, sprerun, P0.type, N.neighbortype;
```

Q8:

```
SELECT P0.type, PL.sprerun,ngalaxy, camcol, count(*)
FROM Q0
WHERE zstatus = 1
AND zwarning < 26:
GROUP BY P0.type, PL.sprerun,ngalaxy, camcol;
```

Q9:

```
SELECT P0.type, PL.sprerun,ngalaxy, count(*)
FROM Q0
WHERE zstatus = 1
AND zwarning < 2
GROUP BY P0.type, PL.sprerun,ngalaxy;
```

Q10:

```
SELECT P0.type,ngalaxy, camcol, count(*)
FROM Q0
WHERE zwarning < 2
GROUP BY P0.type,ngalaxy, camcol;
```

Q11:

```
SELECT P0.type, PL.sprerun,ngalaxy,N.neighbortype, count(*)
FROM Q0, Neighbor N
WHERE N.Neighborobjid = P0.objid
AND zstatus = 1
AND zwarning < 2
GROUP BY P0.type, PL.sprerun,ngalaxy,N.neighbortype;
```

Q12:

```
SELECT P0.type,ngalaxy, camcol, N.Neighbortype, count(*)
FROM Q0, Neighbor N
WHERE N.Neighborobjid = P0.objid
AND zwarning < 2
GROUP BY P0.type,ngalaxy, camcol, N.Neighbortype;
```

Q13:

```
SELECT N.type, N.Neighbortype, primtarget, max(zerr)
FROM Q0, Neighbor N
WHERE N.objid = P0.objid
AND camcol > 2 and camcol < 5
GROUP BY N.type, N.Neighbortype, primtarget;
```

Q14:

```
SELECT zstatus, max(veldisperr)
FROM Q0
WHERE specclass <=3
GROUP BY zstatus;
```

Q15:

```
SELECT zwarning, ngalaxy, count(*)
FROM Q0
WHERE sectarget > 200
GROUP BY zwarning, ngalaxy;
```

Q16:

```
SELECT ngalaxy, count(*)
FROM Q0
WHERE sectarget > 200
AND P0.type = GALAXY
GROUP BY ngalaxy;
```


Q17:

```
SELECT ngalaxy, count(*)
FROM Q0
WHERE sectarget > 200
GROUP BY ngalaxy;
```

Q18:

```
SELECT zwarning, ngalaxy, N.Neighbortype, count(*)
FROM Q0, Neighbor N
WHERE N.Neighborobjid = P0.objid
AND sectarget > 200
GROUP BY zwarning, ngalaxy, N.Neighbortype;
```

Q19:

```
SELECT ngalaxy, N.Neighbortype, count(*)
FROM Q0, Neighbor N
WHERE N.Neighborobjid = P0.objid
AND sectarget > 200
AND P0.type = GALAXY
GROUP BY ngalaxy, N.Neighbortype;
```

Q20:

```
SELECT ngalaxy, N.Neighbortype, count(*)
FROM Q0, Neighbor N
WHERE N.Neighborobjid = P0.objid
AND sectarget > 200
GROUP BY ngalaxy, N.Neighbortype;
```

Appendix C

Implementation Details

C.1 Conversion of a Query from SQL to Datalog

C.1.1 Introduction

In this section we discuss the conversion of an SQL Query to Datalog. We assume only simple queries with no sub-queries. The where clause can have multiple AND clauses. NOT and OR clauses, in the where clause are not considered.

All queries are represented as a parse tree by the parser module in Postgresql. The parse tree is an internal representation of an SQL statement. The query is parsed into the Query data structure. Please refer to the appendix for the complete structure. The relevant elements of the Query data structure are as follows:

(defined in /src/include/nodes/parsenodes.h)

{

CmdType commandType; -indicates whether it is a select/create/delete/update statement

Node *utilityStmt; -non null if its a non-optimizable statement

RangeVar *into;

List *rtable; -list of range table entries

FromExpr *jointree; -represents a from where construct

List *targetList; -a list of projection variables along with their result domains

–internal to planner–

```
List *base_rel_list;    -list of the base relations used in the query
}
```

For now, we will only be looking at the following attributes of the structure:

- List *rtable
- List *targetList
- FromExpr *jointree

The lists which are marked as internal to planner are used for the RelOptInfo structure and so we may need to look into it.

List *rtable:

The range table is a list of base relations that are used in the query. Every range table entry identifies a table or a view and tells by which name it is called in other parts of the query. In the parse tree range table entries are referenced by index rather than by name. In a SELECT statement these are the relations given after the FROM keyword. Each base relation is assigned a relid which is also stored in the range table. Each attribute in a relation is assigned an attribute number and a table number.

For eg: If we have a table A with m attributes. Suppose A is the nth entry in the range table. If we want to refer to the kth attribute of the table we can reference it as varattno k of varno n, or the kth attribute of the nth entry in the table.

List *targetList:

The target list is a list of expressions that define the result of the query. In case of a Select statement, they are the expressions that build the final output of the query. All attributes mentioned in the select clause are in this list.

Every entry in the target list contains an expression that can be a constant value, a variable pointing to an attribute of one of the relations in the range table, a parameter, or an expression tree made of function calls, constants, variables, operators etc. The variables

are in terms of there corresponding range table entries as explained above.

FromExpr *jointree

FromExpr is another primitive node type defined in `/src/include/nodes/primnodes.h`. It has a `fromList` and a node to represent the qualifiers on joins. The query's qualification is an expression which corresponds to the WHERE clause of an SQL statement. The qualifier has an operator, and arguments which could be variables or constants. The operator is assigned an operator Id.

C.1.2 Data Structure used for DataLog

(/src/include/proposeview/unify.h Section 1)

Each DataLog Query has the following structure:

1. A list of sub goals that form the RHS of the datalog query
2. A target sub goal which is the LHS of the datalog query

A Sub goal list is a linked lists of all the subexpressions in the query. Each subgoal in turn has the following structure

1. A sub goal head or the name of the sub goal
2. A linked list of attributes

An attribute has the following structure:

1. Attribute value
2. Flag 'type' which shows whether it is a variable or a constant
3. Flag 'proj' which shows whether it is a projected attribute or not

C.1.3 Algorithm for conversion of a SQL Query to DataLog

(/src/backend/parser/convert.c Section 1)

Input: SQL Query

Output: Datalog Query

Procedure:

1. Check the query whether it is a Simple Select Statement.
 - (a) Check the CmdType field. It is set to 1 in case of a select statement

{ command=1 for select statements }

- (b) For now we only consider simple queries with no sub queries or having clause so the following two attributes should be false
 - {
 - hasSubLinks=false (this is true if there are any subqueries)
 - hasAggs=false (this is true if some aggregate function is used)
 - }
2. Write sub goals along with the attribute names
 - (a) For each entry in the rtable list create a subgoal
 - { create subgoal rtable->eref->aliasname }
 - (b) Add the subgoal to the subgoal list for the query
 - (c) For each subgoal add the attributes from the colnames field in the corresponding rtable entry
 - (d) Add the created subgoal to the goal list
3. Check for the where clause in the SQL Query
 - (a) Check if a join condition exists
 - {
 - jointree->quals is not null
 - }
 - i. If a join exists check for the operator type.
 - If it is an Operator Expression i.e there is only clause with an equality condition, call the opclause function
 - If it is an AND Expression call the andclause function
4. Create the left hand side of our datalog view.
Refer to the TargetList

- (a) If the target list entry refers to a variable, then get the attribute from the goal list
- (b) If the target list entry refers to a constant, then store the value of the constant

C.1.4 Algorithm for translating the Operator Expression

(/src/backend/parser/convert.c Section 2)

Input: Expression Node

Output: Modified Goal List for the query

Procedure:

1. There can be two cases:
 - Equality between two variables
 - Equality between a variable and a constant
2. Case 1: Set variable on the right of the equality equal to the one on the left and set the appropriate flags. In case the 2nd variable has already been set to a constant value as a result of some previous condition, then set variable on the left equal to the one on the right.
3. Case 2: Set the variable equal to the constant and set the appropriate flags. Constants are stored as Datums in Postgresql, change the datum to its corresponding value using functions defined in Postgresql such as DatumGetInt32 etc depending on the data type.

C.1.5 Algorithm for translating the And Expression

(/src/backend/parser/convert.c Section 3)

Input: Expression Node

Output: Modified Goal List for the query

Procedure:

1. Use the canonicalize qual function (defined in /optimizer/prep/prequal.c) to convert the tree of AND clauses into a simple list of AND clauses.
2. Scan the list, to first translate the AND clauses having equality conditions with constant values.
3. Call the opclause function for each such clause
4. Now translate the AND clauses having equality between two variables.
5. Call the opclause function for each such clause

Note: If we first check for equality conditions between two variables, and if there exists more than one conditions on the same variable we might reset the values. For Eg: Consider the query select * from R,S where a=c and c=10; If we first set c equal to a, then we might not be able to translate the clause c=10. On the other hand if we first translate c=10, then when we translate a=c we know that c is already set to a constant and so we will set a =10 giving us the correct query.

C.2 Input to GenerateLattice

C.2.1 Algorithm for generating the input to GenerateLattice

Assumption: We consider that we have a pre-chosen set of tables from the star schema over which we will materialize the views. These tables will include the Fact table and zero or more Dimension Tables.

Input:

1. Chosen set of tables, say Set A of tables.
2. Input Query in Datalog.

Output:

1. The algorithm will output the arguments which should be feed to the Generate-Lattice Command. The arguments to be feed will only be from the chosen set of tables as per our assumption above.
2. Nodes IDs to be marked as query nodes.

Procedure:

1. For each query repeat steps 2 to 6:
2. First pick the best view from the chosen set of Views.
3. From the input query, retrieve list of arguments used to join the fact table with the dimension tables that are not selected in Set A.
 - (a) Retrieve the list of arguments in the body of the datalog structure for the fact table sub goal. This can be done by parsing the linked list of arguments.
 - (b) Check if the arguments appear in the other sub goals as well. For each argument in the fact table, traverse the arguments in the subgoals not in Set A.
 - (c) Save the arguments that appear twice or more and are not in Set A
4. Retrieve list of arguments from Set A that are in the where clause of the input query with a condition to a constant value.
 - (a) Traverse the linked list for the datalog structure for the body of the query.
 - (b) If the subgoal is in Set A, the traverse the linked list of arguments of the sub goal.

- (c) If an argument is a constant, add it to the list. Note: Will have to retrieve the original argument name and not the constant value which can be obtained from the original schema.
 - 5. Retrieve list of arguments from Set A that are the projected grouping arguments in the input query.
 - (a) Traverse the linked list for the datalog structure for the body of the query.
 - (b) If the subgoal is in Set A, the traverse the linked list of arguments of the subgoal.
 - (c) Compare each argument with the arguments in the target sub goal.
 - (d) If an argument matches with an argument in the target sub goal, then add it to the list.
 - 6. The aggregated arguments will always be in the output list of arguments.
 - (a) Traverse the argument list for the target sub goal.
 - (b) If the flag for aggregated argument is set to true add it to the list.
 - 7. The above steps will form a list of grouping arguments to be feed to GenerateLattice.
 - 8. Once the list of grouping arguments is obtained, for each query get the corresponding node id to be marked as a query node.
 - (a) Assign bitwise positions to each of the grouping arguments in the list. Let the first argument be 0 and let the last argument be $(n - 1)$ where n are the total number of arguments.
 - (b) Get the grouping arguments of the query by scanning the projected arguments with the flag for aggregation set to 1.
 - (c) Calculate the equivalent decimal number by adding the grouping arguments present in a query.
-

C.2.2 Example

We will consider the following example for better explanation of the algorithm. Consider Table 1 to be the Fact table and Tables 2,3,4,5,6 to be the dimension tables. Also consider that they have the following schema.

1(A,B,C,D,E,M)

2(A,R)

3(B,S)

4(C,T)

5(D,U)

6(E,V)

Query: Select A,B,U, max(M) from 1,3,5 where 1.B=3.B and 1.D=5.D and B=4 group by A,B,U;

Eg: Consider that tables 1,3 and 4 are chosen for the same. The worst case could be when all the arguments are chosen, that is A,B,C,D,E,S,T,M. Note that M being the measurement argument will always be chosen. Besides M, the following arguments should be chosen:

1. Arguments from the fact table that are used to join with other dimension tables, i.e the tables which are not in the chosen set of tables.

Eg: D should be chosen because it is used to join with 5, which is not in the list of chosen tables.

2. Arguments from the chosen list of tables which are also the projected arguments in the query.

Eg: A and B should be chosen because they are the projected arguments in the query.

3. Arguments from the chosen list of tables which have an equality condition with a constant in the where clause of the query.

Eg: B should be chosen because it is in the where clause (B=4).

4. Final List: A,B,D

Once we have the list of arguments, we execute the GenerateLattice Command to get the set of views to materialize.

C.2.3 Steps

1. Input: Given a set of grouping attributes n
2. Output: A lattice of views (Set S of Views)
3. Procedure:
4. Create vertical nodes for the lattice. This will be equal to the number of grouping attributes. This will help us in adding horizontal nodes in the next step.
5. For $i=2^n$ down to 0
 - (a) Create a node, malloc space for a new view
 - (b) View \rightarrow viewname = string concatenation of V and i
 - (c) View \rightarrow size = `get_view_size()`
 - (d) View \rightarrow grouping_attributes = Binary representation of i
 - (e) View \rightarrow level = `Count_1(binary representation of i)`
 - (f) View \rightarrow flag = false
 - (g) View \rightarrow next_level = NULL
 - (h) Check the vertical level at which to add the node. Add the node at the appropriate level.
 - (i) Each time a new view is added to the lattice, we will check all existing views for the presence of any parent node at the next higher level only. If a parent node exists then initialize the next_level array of pointers to point to this new view added.

Note:

1. The above algorithm has been implemented as `generatelattice()` function in `back-end/commands/generatelattice.c`
2. `Count_1` is a function which counts the number of 1's in the binary number.
3. `get_view_size()` returns the sizes of the views.

C.3 For the Greedy Algorithm:

- Input: Given a lattice of views
- Output: A selected set S of views to materialize
- Procedure:
- Select the top view, i.e the view with all 1's in its binary representation.
 1. Selecting a view would mean setting the flag to true
- For $i=1$ to k , where k is the number of views to choose repeat the following steps:
 1. For each node in the lattice:
 2. Check if `flag = false`
 3. Clear the mark in the lattice. Mark is a boolean variable used by the depth first search function. For each node that is considered, the `dfs()` function is called to get a list of all dependent nodes by tracing the *next_level* pointer recursively. As and when a descendent node is found, MARK is set to true.
 4. Find the set of all views which are dependent on V. Call `dfs()` for View V.
 5. For all marked nodes (which means for all dependent nodes of the node under consideration)
 6. Look for the cheapest view in the set of already selected views. Let this be $C(u)$

7. If $C(v) < C(u)$, then $B_w = C(u) - C(v)$, where B_w is the benefit for w
8. Sum the values of B_w to get $B(V, S)$
 - Choose the view with the maximum value of $B(V, S)$
 - Add V to set S
1. Set flag to true

C.3.1 Structure for the lattice:

We can store the lattice as a linked list or array of structures. We can have the following attributes for a view:

1. Viewname: character type
2. Size: float type
3. List of grouping attributes: list of integers (0's and 1's)
4. Level: integer type, to indicate at which level is the view in the lattice
5. Flag : boolean type to show whether the view is already in the set of selected views or not
6. Next_Level: An array of pointers to point to the child nodes or dependent views in our case.

C.3.2 Implementation Details:

To implement the above the following files have been created/modified:

1. backend/commands/generatelattice.c— This has the functionality for the command.
2. include/generatelattice/lattice.h— Defines all the data structures used in the command execution.

3. backend/nodes/copyfuncs.c— Postgresql advises creating functions to copy any data structure you extend.
4. backend/nodes/equalfuncs.c— Postgresql advises creating functions to compare any data structure you extend.
5. backend/parser/analyze.c— Postgresql analyzer that maps statements to corresponding data structures. It transforms the parse tree into a query tree.
6. backend/parser/gram.y— Postgresql parser; this is where the syntax of the statement is determined. Bison parser makes gram.c from this.
7. backend/parser/keywords.c— The parser determines the reserved words from this file.
8. backend/parser/parse.h— The parser assigns case numbers to strings representing language tokens here.
9. backend/tcop/postgres.c— The tcop(traffic cop) for the backend determines what tag to assign to the query structure root node.

In backend/commands/generatelattice.c, three main functions have been defined, `GenerateLattice()`, `greedy()` and `dfs()`.

GenerateLattice () This function creates the lattice as discussed in Section 1. The only input it takes is the number of grouping attributes.

Greedy() This function implements the greedy algorithm as discussed in Section 2.

DFS() This function does a search similar to the depth first search. It is called on a specific node. So the function will trace the *nextlevel* pointer recursively down the lattice and mark all such nodes.

C.4 Algorithm for Rewriting a Query using a Materialized View

Input:

1. Views selected by the GenerateLattice Command
2. Original Queries in Datalog.
3. Chosen Set of Tables, Set A, which will be replaced by a View.

Output: Rewritten Queries in Datalog

Procedure:

1. Check if the input queries has one or more subgoals from Set A. Continue if it does.
2. Sort the selected views returned by GenerateLattice by view sizes.
3. Until a view is chosen, scan each view from the above list to check if it can be used for the query.
 - (a) All the projected arguments of the query that are from Set A of tables, should also be in the view. Scan the attribute list of the target sub goal for each argument and check the following:
 - i. If the argument is from a table in Set A
 - ii. If it is, then check if it is also in the view.
 - (b) All the arguments from Set A used in the where clause of the query should also be in the view. We will have the following two cases:
 - i. When the argument is used as a join condition.
 - ii. When the argument is used with a condition to a constant.
4. If a view is found the query should be rewritten using the view.

- (a) Modify the from clause of the query
 - i. From the list of subgoals in the body of the query, remove all subgoals that belong to Set A.
 - ii. Add a subgoal for the selected view.
- (b) Modify the where clause of the query
 - i. Conditions which are in terms of the subgoals from Set A have to be translated in terms of the view. The subgoal added above for the selected view should represent all the conditions.
- (c) Modify the projected arguments of the query.
 - i. The aggregated argument will change depending on the arguments in the group by clause.

C.5 Steps

The following steps are followed for the preprocessing of the GenareteLattice Command

1. Files used (/usr/local/pgsql/data):
 - (a) `input_file.txt` This file saves information about the chosen set of tables. It is of the format `fact-table;table1,table2,...`
 - (b) `datalog_query.txt` This file saves the query converted in a datalog format. It is used as an intermediate processing file.
 - (c) `grouping_arguments.txt` This file stores the grouping arguments as the queries are fired. In a full system, this will act as the query analyzer which will scan the input queries to generate a list of grouping arguments over which the views can be generated. The first grouping argument in the file is assigned bit position 0 and so on.
 - (d) `generatelattice_output.txt`

- (e) `view_size.txt` This file saves the output of the `GenerateLattice` Command. It saves the view number followed by the view size.

C.6 Source Code for our implementation of Greedy Algorithm [17]

```
void new_greedy(int attr)
{
    int index,j,super,count,select_view,count1;
    float cv,cu,bv,bw,maxbv;
    struct horizontal *ghtemp1,*ghtemp2,*ghtemp3,*ghtemp4,*ghtemp5;
    struct vertical *gvtemp1, *gvtemp2,*gvtemp3,*gvtemp4,*gvtemp5;
    float totalcost=0;
    vhead->hhead->flag=1;
    index=0;

    /* The selected set of views are written to an output file.
    * The file has view number with a tab delimited view size. The file is read by the
    * rewrite_query module.
    */
    FILE *out_file = fopen("/usr/local/pgsql/data/viewsize.txt", "wb");
    for (index=0;index<=10;index++)
    //while(totalcost <=86600550 | i<=20 )
    {
        maxbv=0;
        select_view=-1;
        for (gvtemp2=vhead; gvtemp2!=NULL; gvtemp2=gvtemp2->vnext)
        {
            for (ghtemp2=gvtemp2->hhead; ghtemp2!=NULL; ghtemp2=ghtemp2->next)
```

```
{
    if(ghtemp2->flag==0)
{
    bv=bw=cu=0;
    cv=ghtemp2->cost; //Cost of view under consideration

    //First set the mark to 0 for the entire lattice.
    for (gvtemp1=vhead; gvtemp1!=NULL; gvtemp1=gvtemp1->vnext)
    {
        for (ghtemp1=gvtemp1->hhead; ghtemp1!=NULL; ghtemp1=ghtemp1->next)
{
    ghtemp1->mark=0;
}
    }
    //Call the dfs function
    dfs(ghtemp2);

    //Check for superset relation
    for (gvtemp3=vhead; gvtemp3!=NULL; gvtemp3=gvtemp3->vnext)
    {
        for (ghtemp3=gvtemp3->hhead; ghtemp3!=NULL; ghtemp3=ghtemp3->next)
{
    if(ghtemp3->mark ==1 )
    {
        if(ghtemp3->query ==1)
        {
count=0; //Reset count to 0
//Again traverse the lattice but check for flag = true.
for (gvtemp4=vhead; gvtemp4!=NULL; gvtemp4=gvtemp4->vnext)
```

```
{
    for (ghtemp4=gvtemp4->hhead; ghtemp4!=NULL; ghtemp4=ghtemp4->next)
    {
    if(ghtemp4->flag ==1 )
    {
        super=0;
        if(count ==0)
            cu=ghtemp4->cost;
            count++;
        for(j=0;j<attr;j++)
        {
            if(ghtemp3->grp[j]==1)
        {
            if (ghtemp4->grp[j]==1)
                super=1;
            else
            {
                super=0;
                break;
            }
        }
    }
    //For the base node , all other nodes will be super nodes.
    if (ghtemp3->viewno == 0)
super=1;
        if (super ==1)
        {
if (ghtemp4->cost <cu)
            cu=ghtemp4->cost;
```

```
    }
    }
    }
}
if(cv < cu)
    bw=cu-cv;
else
    bw=0;
    bv=bv+bw;
    }
    }
    }
}
    }
if(bv>maxbv)
{
maxbv=bv;
select_view=ghtemp2->viewno;
}
}
}

for (gvtemp5=vhead; gvtemp5!=NULL; gvtemp5=gvtemp5->vnext)
{
    for (ghtemp5=gvtemp5->hhead; ghtemp5!=NULL; ghtemp5=ghtemp5->next)
    {
        if(ghtemp5->viewno==select_view)
        {
            ghtemp5->flag=1;

```

```
        fprintf(out_file, "%d\t%d", ghtemp5->viewno, vcost[ghtemp5->viewno]);
    }
}
}
}
}

/*
 *
 * DFS Function
 */
void dfs(struct horizontal *dfs_str)
{
    struct next_lev *dfstemp;

    dfs_str->mark=1;

    for(dfstemp=dfs_str->h_next; dfstemp!=NULL; dfstemp=dfstemp->next)
    {
        dfs(dfstemp->next_level);
    }
    if (dfs_str->h_next ==NULL)
    {
        dfs_str->mark=1;
    }
}
```