Materialized View Design:
An Experimental Study

Diplomarbeit von Vikas Kapoor
Institut für Informatik
Fachbereich Mathematik und Informatik
Freie Universität Berlin

Betreuer: Prof. Dr. Heinz Schweppe & Dr. Agnes Voisard

April 1, 2001

Abstract

Materializing views to improve query response time is a common technique in data warehousing environments. When a query is posed, it is rewritten in terms of available materialized views which, according to the data warehousing system may enhance the execution of this query. On the other hand, if any of the base relation changes, all materialized views depending on this relation have to be updated in order to conserve the consistency of the system. The materialized view design problem is the problem of selecting a set of views to materialize in the data warehouse which answer all queries of interest while minimizing the response time remaining within a given view maintenance cost window.

This work is an experimental study of the materialized view design problem. The test environment we use is specified by the TPC Benchmark R. First, we optimize the test database using standard tuning techniques. We then study the technical aspect of materialized views and develop a naive approach to the problem. Finally, we study two algorithms for the materialized view design problem. We run our tests for each of these options and compare the results.

Contents

Chapter 1

Introduction

Data warehousing is an in-advance approach to provide integrated access to multiple, distributed, heterogeneous databases and other information sources [Wid95]. A data warehouse is a repository which stores integrated information of interest extracted in-advance from different sources, which is then available for querying and analysis. Huge data volumes and complex query types are typical for data warehouses.

Figure 1.1 illustrates the basic architecture of a data warehouse. Each source is monitored for changes. When a new source is attached to the system, or when relevant information at the source changes, these changes are propagated to the integrator. The integrator is responsible for translating the new or changed information to what the data warehouse expects and then integrates it into the data warehouse. When a query is now posed in terms of information available at remote sources, it is forwarded to the query rewriter, which transforms the query in terms of information available in the datawarehouse and answers the query locally. As the query is transformed transparently to the end user, they do not have to worry about query formulation.

The queries supported by data warehouses are often of analytical nature and fall under the notion of On-line Analytical Processing (OLAP). OLAP makes heavy use of aggregate queries. These aggregations are much more complex than in the case of On-line Transaction Processing (OLTP) and thus require local availability of data of interest. To facilitate complex analyses and visualization, the data in a warehouse is typically modeled multidimensionally. For example, in a sales data warehouse, time of sale, sales district, salesperson, and product might be some of the dimensions of interest. Typical OLAP operations include *rollup* (increasing the level of aggregation) and *drill-down* (decreasing the level of ag-

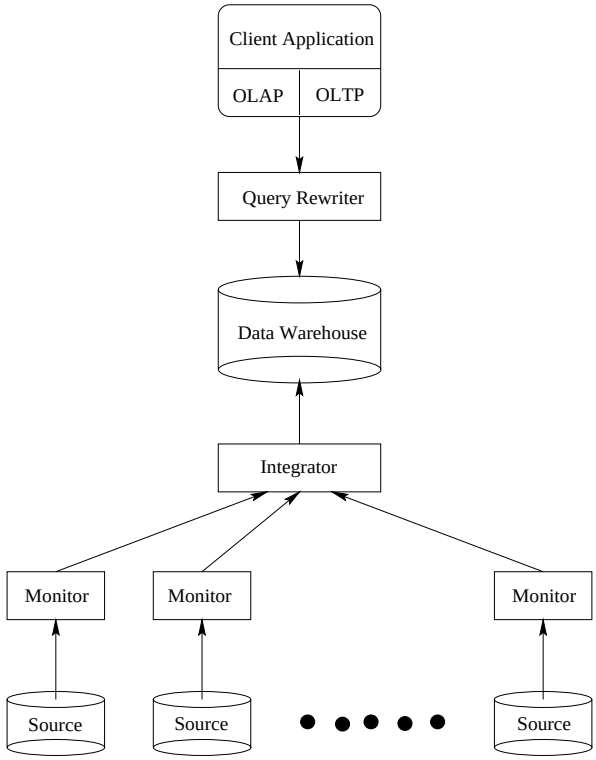

Figure 1.1: Basic data warehouse architecture

gregation or increasing detail) along one or more dimension hierarchies, *slice-and-dice* (selection and projection), and *pivot* (re-orienting the multidimensional view of data) [CD97]. Trying to execute such complex OLAP queries against a database designed to support OLTP queries would result in unacceptable system performance. Thus the requirement of special data organization, access methods, and implementation methods arises, justifying the need to address the data warehousing issues seperately.

Data warehouses might be implemented on standard or extended relational DBMSs, called Relational OLAP (ROLAP) servers. These servers assume that data is stored in relational databases, and they support extensions to SQL and special access and implementation methods to efficiently implement the multidimensional data model and operations. In contrast, multidimensional OLAP (MOLAP) servers are servers that directly store multidimensional data in special structures and implement the OLAP operations over these special data structures [MJV98].

1.1 The Materialized View Design Problem

A data warehouse can be seen as a set of materialized views defined over the sources. *The materialized view design problem* is the problem of selecting a set of views to materialize in the data warehouse which answer all queries of interest while minimizing the response time remaining within a given view maintenance cost window. This problem, being one of the most important decisions in designing a data warehouse is often addressed as the data warehouse configuration problem and is the focus of this work.

Before we formalize the materialized view design problem, we make some assumptions. First, we assume that all information needed for query processing is available in the data warehouse in form of relational tables. We call these tables *base relations*. Second, we want our system to support update operations to the base relations in parallel to query executions.[1] Naturally, the number of updates should be very small as compared to number of queries posed within a certain interval of time. In other words, the system should efficiently support query executions with occasional updates to the data warehouse. We do not assume any resource limitation such as materialization time, storage space, or total view maintenance time. The algorithms presented in [JYL97] and [TS97] match our assumptions and are thus implemented by us.

The goal is to select an appropriate set of views that minimizes total query response time and the cost of maintaining the selected views. Formally, given a set of base relations

$$R = \{r_1, r_2, .., r_n\} \text{ and}$$

a set of queries

$$Q = \{q_1, q_2, .., q_m\},$$

find a set of views defined on R

$$V = \{v_1, v_2, .., v_l\}$$

such that the operational cost

$$C(Q, V) = E(Q) + M(V) \text{ is minimal,}$$

where $E(Q) = \sum_{q \in Q} f^q E(q)$ is the query evaluation cost and $M(V) = \sum_{v \in V} f^v M(v)$ the view maintenance cost. f represents the relative frequency of the respective operation.

[1] This is in contrast to append-only and bulk update datawarehouses where there are no delete operations and inserts are done only in batches when the system is offline.

1.2 Related Work

As compared to the view maintenance problem ([YW00], [CW00], [MBM99], [PVQ99] to name some of the latest), which is the problem of updating materialized views if any base relation changes, there has been little work done in the field of materialized view design problem. In the following, we briefly describe some of these works.

A Greedy algorithm is provided in [VHU96] which they claim, performs within a small constant factor of optimal under a variety of models. They use a lattice framework to express dependencies among views. The greedy algorithm works off this lattice to determine the set of views to materialize. They then examine the most common case of the hypercube lattice in detail.

[Gup97] presents polynomial-time heuristics for selection of views to optimize total query response time, for some important special cases of the general data warehouse scenario, viz.: (i) an AND view graph, where each query/view has a unique evaluation, and (ii) an OR view graph, in which any view can be computed from any one of its related views, e.g., data cubes. Then the algorithms are extended to the case when there is a set of indexes associated with each view. Finally, they extend their heuristic to the general case of AND-OR view graphs.

[TS97] addresses the problem by formulating it as a state space optimization problem and then solves it using an exhaustive incremental algorithm as well as a heuristic one. This method is then extended by considering the case where auxiliary views are stored in the data warehouse solely for reducing the view maintenance cost.

[JYL97] suggests a heuristic which provides a feasible solution based on individual optimal query plans. They then map the materialized view design problem as 0-1 integer programming problem, whose solution can guarantee an optimal solution.

The technique proposed in [EBT97] reduces the solution space by considering only the relevant elements of the multidimensional lattice. An additional statistical analysis allows further reduction of the solution space.

In [GM99], first they design an approximation greedy algorithm for the problem in OR view graphs. They prove that the query benefit of the solution delivered by the proposed greedy heuristic is within 63% of that of the optimal solution. Second, they design a heuristic which they call a A^* heuristic, that delivers an optimal solution, for the general case of AND-OR view graphs.

DynaMat, the system presented in [KR99], unifies the view selection and view maintenance problem under a single framework. The incoming queries are con-

stantly monitored and views are materialized accordingly subject to the space constraints. During updates, the system reconciles the current materialized view selection and refreshes the most beneficial subset of it with in a given maintenance window. They compare their system experimentally against a system that is given all queries in advance and the pre-computed optimal static view selection.

1.3 Goals and Organization of this Work

This work is an experimental study of the materialized view design problem. On the one hand we want to find out the impact of view materialization on a system which has been optimized using only standard database tuning techniques. On the other hand we want to compare different view materialization strategies.

Due to wide acceptance, we use TPC Benchmark R (TPC-R) specifications as the basis of our test environment. Chapter 2 describes what TPC-R provides and is relevant for us. We also disclose which TPC-R conventions are not held by us and why. The driver, which has the responsibility of posing queries and measuring the execution times is also presented.

Chapter 3 concerns standard database tuning techniques which we have used to optimize the system. For the sake of comparison, we first present the execution times for all the queries without any tuning. These timings are then compared to the query execution times after we tune the system. Due to space constraint, We do not analyse and explain execution time improvement for each and every query, but give a general idea of the contribution of each performance tuning technique.

In Chapter 4 we look at the technical aspects of materialized views in our database management system[2]. We describe the different kinds of materialized views available in our system, how to create them, how they are maintained, how they are used by the query rewriter, and most importantly the limitations they have. We also present a naive solution to materialized view design problem.

Chapter 5 takes a detailed look at the two algorithms which we chose to implement. We have implemented the algorithms presented in [TS97] and [JYL97]. Apart from explaining, why we chose these algorithms, we analyse them along with their cost models and present the pseudocode for each of them.

Finally, in Chapter 6 we present the results of our tests. For each available option, we tested the performance of the system for the case when only one user is interacting with the system and for the case when many users are concurrently

[2]We use *Oracle 8i* as the underlying system. Whenever we refer to some technical aspect in this work, it is specific to *Oracle 8i*.

posing queries on the system. We compare the change in query execution time for each case and try to reason out these changes. The result of this analysis is then presented in Chapter 7.

Chapter 2

Test Environment

The *Transaction Processing Performance Council* [1] (TPC) provides a number of database benchmarks which serve the purpose of measuring the performance of systems having different goals. The TPC Benchmark R (TPC-R) is a decision support benchmark which evaluates the performance of a decision support system by executing a set of queries against the system database under controlled conditions. Apart from a logical database design, TPC-R provides a suite of business oriented queries and data modification operations. Data and query generating tools are also provided.

In the following we take a closer look at the TPC-R specification which forms the basis of the test environment we have decided to use for our purpose. At the end of this chapter we describe how we measure the performance of the system which is a bit different than the TPC-R performance metric. For futher details on TPC-R please refer to [Tra].

2.1 Database Schema and Queries

Figure 2.1 shows the TPC-R logical database design. It consists of eight base tables. The relations between columns of these tables are illustrated by the arrows pointing in the direction of one-to-many relationships. Attributes of all the tables have the `NOT NULL` constraint. TPC-R holds the use of any other constraint to be optional. As the DBMS we have used does not allow materialized views to be defined on tables having no primary keys, we define the primary key constraint for each table. These primary keys are the same as specified by TPC-R.

[1] http://www.tpc.org/

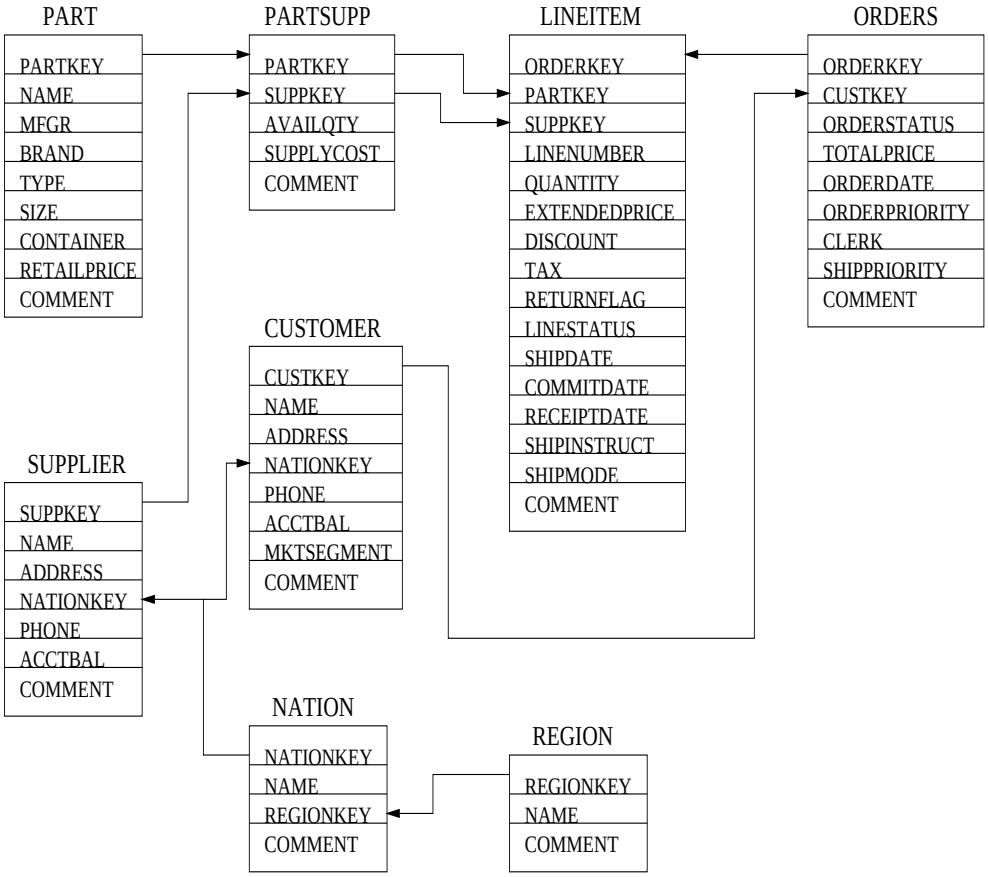

Figure 2.1: TPC-R Database Schema

TPC-R describes twenty-two decision support queries and two operations for the purpose of database refreshment. The refresh operations delete or insert rows either in the `LINEITEM` or in the `ORDERS` table, which are also the largest and most frequently used tables. The refresh operations are:

```
RF1:  LOOP (SF*1500) TIMES
        INSERT a new row into the ORDERS table
        LOOP RANDOM(1, 7) TIMES
          INSERT a new row into the LINEITEM table
        END LOOP
      END LOOP

RF2:  LOOP (SF*1500) TIMES
```

```
    DELETE FROM ORDERS WHERE O_ORDERKEY = [value]
    DELETE FROM LINEITEM WHERE L_ORDERKEY = [value]
END LOOP
```

where SF is the scale factor defined in the next section. These refresh operations simulate batch refreshes. As already mentioned, we are interested in refresh operations which can be executed when the database is operational, i.e. queries and refresh operations are executed simultaneously. Refresh operations in their above form represening bulk updates are too intensive for our system under test which has to support some degree of operational data. Apart from this, the frequency of an update to a table should be relatively low as compared to the frequency of queries, as is the case in data warehousing environments. One can naturally force the driver (see Section 2.3) to sleep after every insert or delete operation, but as we are also interested in the execution times for each single update to a table, we prefer to view each update operation to be atomic as follows:

```
RF1: DELETE FROM ORDERS WHERE O_ORDERKEY = [value]
RF2: DELETE FROM LINEITEM WHERE L_ORDERKEY = [value]
RF3: INSERT a new row into the ORDERS table
RF4: INSERT a new row into the LINEITEM table
```

From our point of view only those queries are of interest whose response times are affected by a concurrent data manipulation operation. Any query which does not make use of any of these two tables is not considered by us. So the set of queries in our case contains only eighteen of the twenty two queries provided by TPC-R. The queries are presented in Appendix A. [2]

QGEN is a utility provided by TPC to generate executable query text. We use QGEN to generate executable query text for the queries presented in Appendix A. The generation of refresh data for the refresh operations is explained in the next section.

2.2 Data Generation and Population

The TPC provides a utility, DBGEN, for the purpose of data generation. This tool has been used to generate test as well as refresh data. DBGEN is written in ANSI C and allows data to be generated either in flat files or on the fly (inline data generation). We connected DBGEN to our database via the *Oracle Call Interface*

[2]The *Oracle 8i* system has had problems generating an execution plan for query Q21 which is query Q16 in our case. So we had to simplify this query.

(OCI) in order to populate the tables inline. DBGEN expects a scale factor (SF) with which the data has to be generated. Scale factor is a reference to the database size which is approximately 1GB for SF = 1. TPC-R allows to choose between the following scale factors: 1, 10, 30, 100, 300, 1000, 3000, 10000. We run our experiments only for SF = 1. [3] One can consider our data warehousing environment to be one with minimum data volume.

The database is initially populated with 75% sparse primary keys for the ORDERS and LINEITEM tables where only the first eight key values of each group of 32 keys are used. Subsequently, the insert refresh operations use the 'holes' in the key ranges for inserting new rows. DBGEN generates refresh data sets for the refresh operations such that for the first through the 1000th execution of insert refresh operations, data sets are generated for inserting 0.1% new rows with a primary key within the second 8 key values of each group of 32 keys and for the first through the 1000th execution of delete refresh operations, data sets are generated for deleting 0.1% existing rows with a primary key within the original first 8 key values of each group of 32 keys. As a result, after 1000 executions of insert/delete pairs the database is still populated with 75% sparse primary keys, but the second 8 key values of each group of 32 keys are now used.

TPC-R asks to assure the correctness of the database during the tests. This is accomplished by keeping track of which set of inserted and deleted rows should be used by the refresh operations. TPC-R also recommends to build a qualification database for the purpose of query validation. The qualification database should not be updated. We have not built an extra qualification database. Instead, we have checked the results for all the queries each time before we run our test for any database configuration. The query validation output data is provided by TPC-R.

2.3 Driver

The driver shown in Figure 2.2 has the responsibility of activating, scheduling, and synchronizing the execution of queries and refresh operations. For each submitted query/refresh operation, the driver measures the execution time in seconds and writes it to a log file.

The driver opens four connections to the database. Three of these connections are used for submitting queries and the fourth one is used solely for refreshing

[3] Remember that SF = 1 in our case may mean that the data is much more than 1GB for a database configuration with materialized views

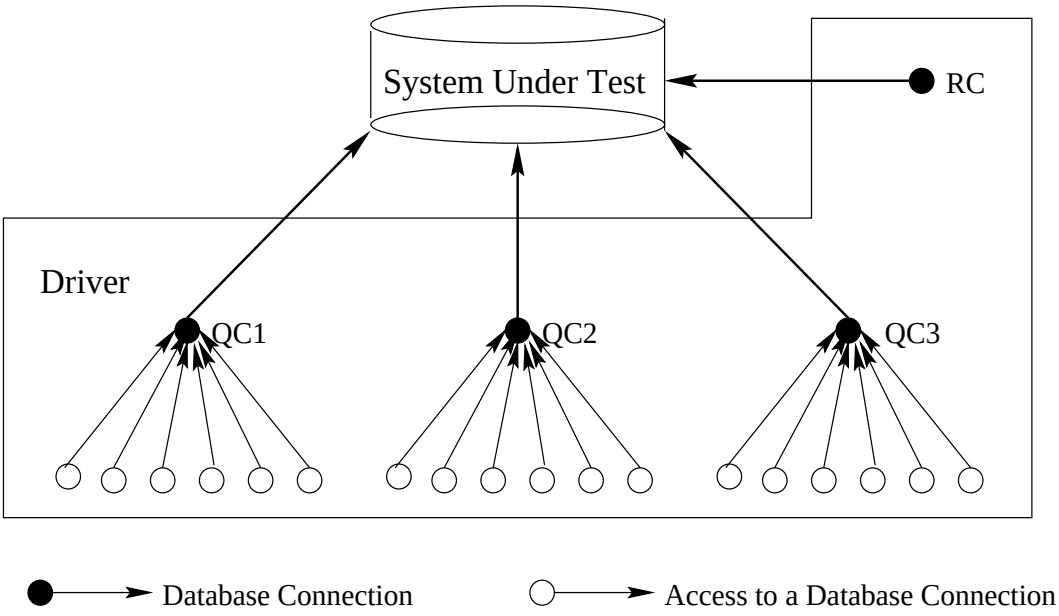

Figure 2.2: The Driver Configuration

purpose. We call these connections query connections (`QC1, QC2, QC3`) and refresh connection (`RC`), respectively. Whereas the time between two refresh operations is constant (24 minutes) throughout our experiments for any database configuration, we reduce the submission time between two queries through a single query connection to increase the workload on the system. However, each connection poses a query at the same time in parallel.

The reduction of submission time between two queries does not take place arbitrarily. The time interval has to be scheduled such that one can express the frequency of query submission with respect to refresh operations. For example, a work load of `1:N` means: during the time there is one update (insert or delete) to each of the tables `ORDERS` and `LINEITEM`, each of the eighteen TPC-R queries is submitted exactly `N` times for execution. The queries are submitted in a round robin fashion. `QC1` starts with `Q0`, `QC2` with `Q6`, and `QC3` with `Q12` and proceed to execute the next sequential query. After the submission of `Q17` by any connection, the corresponding connection executes `Q0` in the following submission.

We perform our experiments for the following workloads:

- `1:1:` each query connection submits a query every eight minutes. So after 48 minutes, `QC1` executes `Q0` through `Q5`, `QC2` executes `Q6` through `Q11`,

and QC3 executes Q12 through Q17. In the same time RC executes one refresh operation to each of ORDERS and LINEITEM tables. Hence realising a workload of 1:1. In the next 48 minutes QC1 plays the role of QC2, QC2 of QC3, and QC3 of QC1.

- 1:2: each query connection submits a query every four minutes. After 48 minutes each query is submitted exactly twice.
- 1:3: each query connection submits a query every 1 min 40 sec. After 48 minutes each query is submitted exactly thrice.

Driver Implementation: Our driver has been written in the Java programming language. The connection to the database is done through JDBC (Java Database Connectivity).[4] Each connection has a timer associated with it. The timer has the responsibility of posing the next query after a given time interval. The Timer class provided by the swing package has been used for this purpose. As the execution of a query may take quite a long time (which may be more than the time after which the timer is supposed to execute the next query), the timer sends the query execution process in the background with the help of SwingWorker class and is then ready to pose the next query for execution.

2.4 Performance Metrics

The TPC-R benchmark defines three different types of tests. Although, we do not follow the TPC-R performance metrics rules, we briefly describe these tests in the following.

- *Load test:* begins with the creation of the database tables and includes all activity required to bring the system under test to the configuration that immediately precedes the beginning of the power or throughput test.
- *Power test:* measures the raw query execution power of the system when connected with a single active user. In this test, the refresh operations are executed exclusively by a separate refresh connection and scheduled before and after the execution of the queries. The results of the power test are used to compute the TPC-R query processing power at the chosen database size.

[4] using the Oracle OCI JDBC driver

It is defined as the inverse of the geometric mean of the timing interval, and is computed as:

$$Power@Size = \frac{3600 \times SF}{\sqrt{\prod_{i=1}^{i=18} QI(i) \times \prod_{j=1}^{j=4} RI(j)}}$$

where $QI(i)$ is the timing interval, in seconds, of query Q_i and $RI(j)$ is the timing interval, in seconds, of refresh function RF_j. The units of $Power@Size$ are $queriesperhour \times ScaleFactor$.

- *Throughput test:* measures the ability of the system to process the most queries in the least amount of time. In this test, refresh operations are executed exclusively by a separate refresh connection and scheduled as defined in 2.3. The test is driven by queries submitted by the driver through two or more connections. The measurement interval T_s for the throughput test is measured in seconds. It starts when the first character of the executable query text of the first query of the first query connection is submitted by the driver, or when the first character requesting the execution of the first refresh operation is submitted by the driver, whichever happens first. It ends when the last character of output data from the last query of the last query connection is received by the driver, or when the last transaction of the last refresh operation has been completely and successfully committed and a success message has been received by the driver, whichever happens last. The results of throughput test are defined as the ratio of the total number of queries executed over the length of the measurement interval:

$$Throughput@Size = \frac{S \times 18 \times 3600}{T_s} \times SF$$

where S is the number of query connections used in the throughput test. The units of $Throughput@Size$ are the same as for $Power@Size$.

We diverge a bit from TPC-R performance metrics. The load test is not relevant in our case as we are not interested in the time needed to configure the database. The power test in our case is a sequential execution of all queries and refresh operations when no other process is accessing the database. The result is presented as a bar chart as in Figure 3.1. Power test results serve as a pre-check for query execution times. It makes sense to go for the throughput test, only if all queries have acceptable query execution times. The throughput test is the execution of the driver described in Section 2.3 for any of the three workloads. For each

database configuration and each workload we run the driver for six hours. The results are presented in a similar way as for the power test, just that the query times represent the average query time for all execution of that particular query. The number of times a query was executed during a throughput test is also presented.

The focus of our interest is the change in individual query execution times for different database configurations for the power as well as throughput test. We want to monitor these timings for each of the query and refresh operation and try to reason out the improvement or worsening of these timings as the configuration changes. We do not report $Power@Size$ and $Throughput@Size$, firstly, because in our case they are not exactly what TPC-R says and secondly, from our point of view they are not of interest for our analysis. But still if someone wants to calculate these numbers, all the values needed for the calculation are presented.

Chapter 3

Tuning for Performance with Standard Techniques

As mentioned earlier, power test serves as a pre-check for execution times for queries and refresh operations. It makes sense to run the driver only if these timings lie in an acceptable range. In Section 3.1 we present the power test results for the system when it is tested as it is, that is without using any performance tuning techniques. Section 3.2 gives an introduction to the performance tuning techniques provided by our system which we have used. In Section 3.3 we compare the results of the untuned and tuned system.

3.1 The Untuned System

The untuned system in our case is the system we have at our hand after the schema described in Section 2.1 is created in the database and populated with DBGEN for a scale factor of 1. The only constraints defined are the `NOT NULL` constraints (in this case for every column in every table) and the `PRIMARY KEY` constraints for each table. Just to remind that defining a field (or a group of fields) as primary key creates implicitly a unique index on that field (or group of fields). So these are the only indexes present in the database. Our system provides standard configuration for a data warehousing application which we have used. We have not changed anything in the proposed configuration. We do not assume that anybody would consider operating a system for data warehousing purpose without any sort of optimization. We have done so just to have something to compare with, when we tune the system.

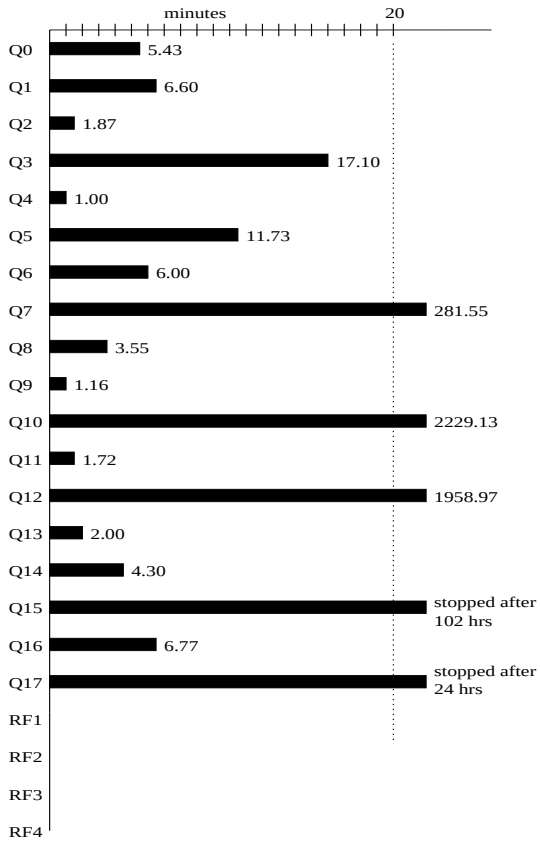

Figure 3.1: Execution Times for Untuned System

Figure 3.1 shows the result of the power test. Execution times below 1 second are not shown. This is the case for all refresh functions. As mentioned earlier, we want to run our driver for 6 hours. The execution times for the queries are clearly unacceptable for any sort of application let alone running our driver described in Section 2.3. This thus justifies an inevitable need to tune the system for performace.

3.2 Tuning the System

In the following we give a brief intoduction to the tuning concepts provided by the database management system at our hand. The directives given by us to the system to realise the corresponding tuning method are presented in boxes in each

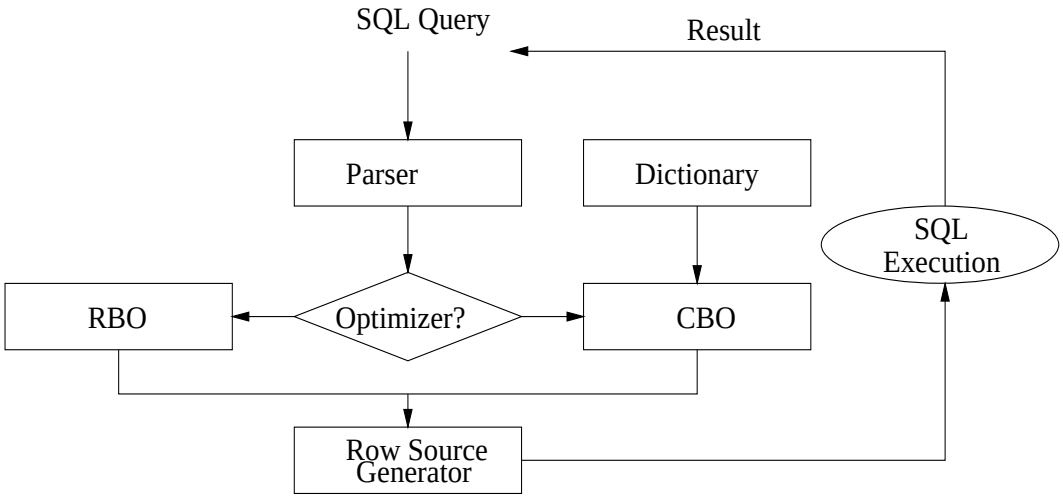

Figure 3.2: SQL Processing Architecture of Oracle 8i

section. The first line in each box tells whether the directives are initialization[1] or SQL[2] directives.

3.2.1 Choosing the Cost Based Optimizer

A query execution plan is a combination of steps, used by the system to execute a given SQL statement. It consists of access method for each table that the statement accesses and the join order of the tables. By varying the order in which tables and indexes are accessed, one can have many different execution plans for the same statement. Furthermore, answering a statement using a particular execution plan may be good from one point of view (for e.g., minimal memory usage), but not from the other (for e.g., minimal execution time). The optimizer has the responsibility of choosing the most efficient way to execute a SQL statement which would fulfill a given goal.

Figure 3.2 shows the SQL processing architecture of the used system. The parser performs syntax and semantic check of the SQL statement and forwards it to the optimizer. The optimizer outputs an optimal plan which forms the input to the row source generator. The row source generator structures the row sources in

[1]The initilaization parameter settings for an Oracle instance have to be entered in the init<instance name>.ora file.

[2]for example to be executed through the SQLPLUS interface provided by Oracle 8i

form of a tree which is then called an execution plan. A row source is an iterative control structure which is responsible to produce a row set. The SQL execution module then operates on the execution plan to produce the result of the query which is then forwarded to the client.

As evident from Figure 3.2 two methods of optimization are provided: rule-based optimizer (**RBO**) and cost-based optmizer (**CBO**). The goal of the CBO is to optimize for the best throughput; that is using the least amount of resources necessary to process all rows accessed by the statement. The goal of the RBO is to optimize for the best response time; i.e. using the least amount of resources necessary to process the first row accessed by a SQL statement. The execution plan produced by the optimizer can vary depending on the optimizer's goal. CBO is more likely to result in a full table scan rather than index scan, or a sort-merge join rather than a nested loop join. RBO more likely chooses an index scan or a nested loop join. It is quite evident that we need the CBO as an optimizer as we are interested in better throughput rather than response time.

Our system provides the following optimizer modes:

- CHOOSE: the system decides itself which is the better approach,
- ALL_ROWS: use CBO,
- FIRST_ROWS: use CBO but optimize for response time,
- RULE: use RBO

The CBO generates a set of potential execution plans for the SQL statement based on available access paths and estimates the cost of each of them. The cost estimation is based on statistics in the data dictionary for the data distribution and storage characteristics of the tables and indexes accessed by the statement. The cost is an estimated value proportional to the expected resource use needed to execute the statement with a particular execution plan. The optimizer calculates the cost of each possible access method and join order based on the estimated computer resources, including I/O, CPU time, and memory, that are required to execute the statement using the plan. The execution plan with the smallest cost is then chosen by the optimizer.

Gathering statistics in the data dictionary helps the CBO in deciding which query plan to choose and is one of the most important features of cost based optimization. One has to give directives to the system, so that it starts gathering statistics for a particular index or table. We have not given any such directive

to the system. The reason for this is our goal of comparison between different database configurations. To get a true picture about the quality of configuration by a particular strategy, we have to ensure that the same execution plans are selected as proposed by the strategy no matter whether a better one exists. But still we would like to use the CBO due to other advantages described in the following.

Enabling CBO enables several optimizer features, depending on the user-specified value. For example, the complex view merging and common subexpression elimination are automatically enabled. Complex view merging allows the optimizer to merge the query in the statement with that in the view, and then optimize the result. Eliminating common subexpressions allows the use of an optimization heuristic that identifies, removes, and collects common subexpressions from disjunctive branches of a query.

Pushing join predicates into the view is another feature available only with the CBO. Other features which can use only cost based optimization and are used by us are bitmap indexes and parallel query.

```
#init.ora
OPTIMIZER_MODE=ALL_ROWS
OPTIMIZER_FEATURES_ENABLED=8.1.6
COMPATIBLE=8.1.6
_PUSH_JOIN_PREDICATE=TRUE
OPTIMIZER_INDEX_COST_ADJ=TRUE
```

3.2.2 Tuning the Data Access Method

Creating indexes on fields in tables which are accessed frequently is a common technique to improve the performance of a system. Indexes improve the performance of queries that select a small percentage of rows from a table. Although cost-based optimization helps avoid the use of nonselective indexes within query execution, the system must continue to maintain all indexes defined against a table regardless of whether they are used. Creating indexes “just in case” is not a good practice as index maintenance can present a significant CPU and I/O resource demand. We have checked the usage of each index we have created by looking at the query execution plan.

Talking about indexes is almost always talking about B-Tree indexes. As a general guideline, indexes are created on tables that are queried for less than 2% or 4% of the table’s rows. This value may be higher in situations where all data can be retrieved from an index, or where the indexed columns and expressions can be

used for joining to other tables. Unfortunately, in our case either the percentage of rows which can be accessed through an index on a field in our tables is either too big, making the index structure too large and causing a worsening of execution time of the respective SQL statement due to CPU swapping or the number of distinct values of the field is too small, resulting in traversing down the B-Tree and then going through a linked list sequentially and again worsening the execution times. After much testing we have been able to create only two B-Tree indexes which have helped improving the performance of our system.

```
#SQLPLUS>
CREATE INDEX l_partkey_idx on LINEITEM (l_partkey)
CREATE INDEX o_custkey_idx on ORDERS (o_custkey)
```

An alternative to B-tree indexes are bitmap indexes. Bitmap indexes should used when the WHERE clause of a statement contains predicates on low- or medium-cardinality columns and the tables being queried contain many rows. Multiple bitmap indexes can be used to evaluate the conditions on a single table. Bitmap indexes can also provide optimal performance for aggregate queries. In a bitmap index, a bitmap for each key value is used instead of a list of rowid, as is the case with B-Tree index. Each bit in the bitmap corresponds to a possible rowid, and if the bit is set, it means that the row with the corresponding rowid contains the key value. A mapping function converts the bit position to an actual rowid, so the bitmap provides the same functionality as a regular index even though it uses a different representation internally. If the number of different key values is small, bitmap indexes are very space effecient. For columns where each value is repeated hundreds or thousands of times, a bitmap index typically is less than 25% of the size of a regular B-tree index. We have created bitmap indexes on columns of the tables which are referenced in the where clause of the queries and have cardinality less than 25.

```
#SQLPLUS>
CREATE BITMAP INDEX l_returnflag_bm_idx on LINEITEM (l_returnflag)
CREATE BITMAP INDEX l_shipmode_bm_idx on LINEITEM (l_shipmode)
CREATE BITMAP INDEX l_shipinstruct_bm_idx on LINEITEM (l_shipinstruct)
CREATE BITMAP INDEX l_linestatus_bm_idx on LINEITEM (l_linestatus)
CREATE BITMAP INDEX o_orderpriority_bm_idx on ORDERS (o_orderpriority)
CREATE BITMAP INDEX o_orderstatus_bm_idx on ORDERS (o_orderstatus)
CREATE BITMAP INDEX c_nationkey_bm_idx on CUSTOMER (c_nationkey)
CREATE BITMAP INDEX s_nationkey_bm_idx on SUPPLIER (s_nationkey)
```

The following factors have to be taken care of while using bitmap indexes:

- Bitmap create area determines the amount of memory allocated for bitmap creation. The default value is 8MB (also in our case). A larger value may lead to faster index creation.
- Bitmap merge area determines the amount of memory used to merge bitmaps retrieved from a range scan of the index. The default is 1MB, we have exceeded it to 50MB as a larger value improves performance because the bitmap segments must be sorted before being merged into a single bitmap.
- The sort area parameter being a very important parameter is discussed in detail in the next section.

```
#init.ora
CREATE_BITMAP_AREA_SIZE=8388608
BITMAP_MERGE_AREA_SIZE=52428800
```

3.2.3 Tuning Sorts

If one looks at the TPC-R queries in Appendix A, almost every query has a `group by` and/or `order by` clause. The execution of these queries require sorting of intermediate or end results. The sort area allocated to the system environment has a critical impact on the sort process. Sort area specifies the maximum amount of memory to use for each sort. There is a trade-off between performance and memory usage. For best performance, most sorts should occur in memory; sorts written to disk adversely affect performance. If sort area size is too large, then too much memory may be used. If sort area size is too small, then sorts may need to be written to disk which can severely degrade performance.

When the system writes sort operations to disk, it writes out partially sorted data in sorted runs. After all the data has been received by the sort, the system merges the runs to produce the final sorted output. If the sort area is not large enough to merge all the runs at once, then subsets of the runs are merged in several merge passes. If the sort area is larger, then there are fewer, longer runs produced. A larger sort area also means the sort can merge runs in one merge pass. On the other hand, increasing sort area size causes each sort process to allocate more memory. This increase reduces the amount of memory for private SQL and PL/SQL areas. It can also affect operating system memory allocation

and may induce paging and swapping. Before increasing the size of the sort area, one has to be sure that enough free memory is available in the operating system.

If one decides to increase the size of the sort area, as we have done, one should take care of deallocating memory once it is not needed by the corresponding sort operation. The size of retained sort area, which controls the lower limit to which the size of the sort area is reduced when the system completes some or all of a sort process, should be set to a minimum. As soon as the sending of sorted data begins, the size of the sort area is reduced.

Normally, a sort may require many space allocation calls to allocate and deallocate temporary segments. Sort performance can also be increased by specifying a tablespace as `TEMPORARY`. The system then caches one sort segment in that tablespace for each instance requesting a sort operation. As this scheme bypasses the normal space allocation mechanism and improves performance, we have allocated 1GB temporary tablespace to our database.

Increasing the sort area size has caused a big improvement in the execution time of some queries. For example, the improvement in execution time of query Q7 in Figure 3.3 is basically due to sort area tuning.

```
#init.ora
SORT_AREA_SIZE=524288000
SORT_AREA_RETAINED_SIZE=0
```

3.2.4 Tuning Parallel Execution of Individual Queries

When the system is not parallelizing the execution of SQL statements, each SQL statement is executed sequentially by a single process. With parallel execution, however, multiple processes work together simultaneously to execute a single SQL statement. By dividing the work necessary to execute a statement among multiple processes, the system can execute the statement more quickly than if only a single process executed it. Parallel execution can dramatically improve performance for data-intensive operations as in our case.

A process known as the parallel execution coordinator breaks down execution functions into parallel pieces and then integrates the partial results produced by the parallel execution servers. The number of parallel execution servers assigned to a single operation is the degree of parallelism for an operation. Multiple operations within the same SQL statement all have the same degree of parallelism. When the system starts up, it creates a pool of parallel execution servers which are available for any parallel operation. When executing a parallel operation, the

parallel execution coordinator obtains parallel execution servers from the pool and assigns them to the operation. If necessary, the system can create additional parallel execution servers for the operation. These parallel execution servers remain with the operation throughout job execution, then become available for other operations. After the statement has been processed completely, the parallel execution servers return to the pool. When a user issues an SQL statement, the optimizer decides whether to execute the operations in parallel and determines the degree of parallelism for each operation.

The maximum number of parallel execution servers has to be specified. This value cannot be more than $4 \times num\,of\,CPUs$. Setting this value anything greater than this may cause operating system errors. If all parallel execution servers in the pool are occupied and the maximum number of parallel execution servers has been started, the parallel execution coordinator switches to serial processing. One should also take care of ensuring that the degree of parallelism be reduced as the load on the system increases.

```
#init.ora
PARALLEL_MIN_SERVERS=1
PARALLEL_MAX_SERVERS=8
PARALLEL_ADAPTIVE_MULTI_USER=TRUE
```

One can parallelize almost every operation. As we are not allowed to change the SQL statements provided by TPC-R, we specify degree of parallelism only on the table level. Index level parallelism did not bring any performance gains in our case.

```
#SQLPLUS>
ALTER TABLE LINEITEM PARALLEL 3
ALTER TABLE ORDERS PARALLEL 2
```

Any value greater or less than these have resulted in performance loss in our case. Blindly setting these values to large numbers seems to worsen the execution times of our statements dramatically. These values are a result of intensive testing. Although we have taken care of reducing the degree of parallelism as the load on the system increases, this does not guarentee an improved performance through parallelized statement.

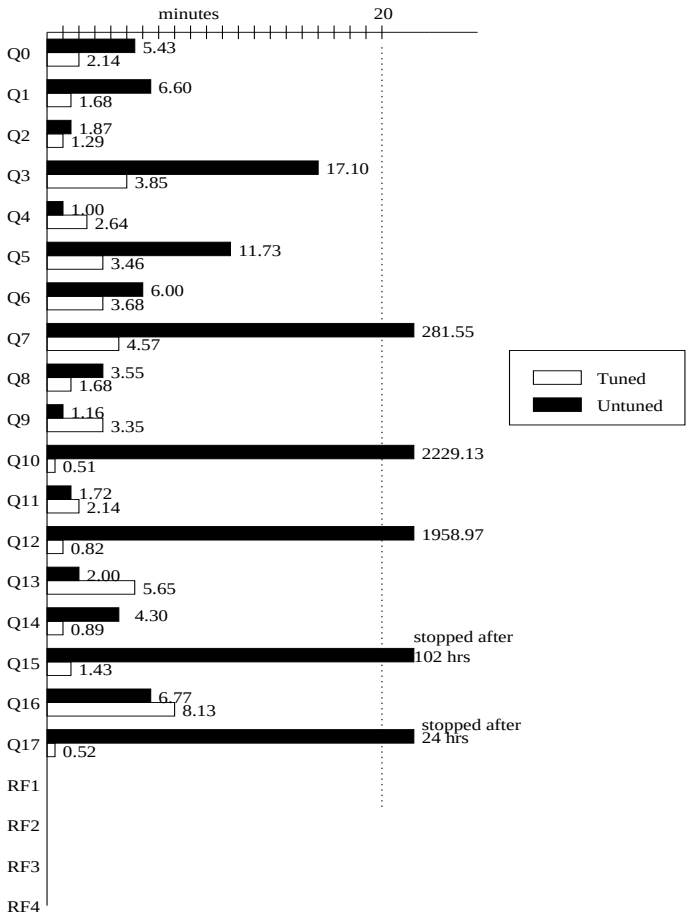

Figure 3.3: Comparison of Query Execution Times for the Tuned vs. the Untuned System

3.3 Results of Power Test

Figure 3.3 shows the result of power test for the system tuned with the techniques described in the previous section as compared to the untuned system. As can be clearly seen, tuning for performance has brought an immense improvement in the execution times of the query. One thing which also strikes is the worsening of query execution times in some cases. Instead of analyzing the change in execution times for each and every query, we just go through the queries whose execution times have worsened. The improvements in execution times have obvious reasons, so we leave it for the reader to reason it out.

The execution times of queries Q4, Q9, Q11, and Q13 have worsened. As mentioned earlier, our system has had problems parsing Q16, so we would not be analyzing this query any further. The execution plans for Q4 and Q9 have remained the same as for the untuned system. The only change is the degree of parallelism. Setting the degree of parallelism to the values as they were in the untuned system brings the execution times for these queries to their old (better) execution times. But as the degree of parallelism has brought an overall improvement, we have decided to bear this side affect. The execution plan for Q11 has changed as it now uses an index to access the l_partkey field of the LINEITEM table. This does not seem to be a better way than doing a full table scan in this case. This index has brought a drastic improvement in case of Q12 and therefore we have decided for it. In case of Q13 the index on field o_custkey of the ORDERS table has brought for a worsening of query execution time. As this index is the cause for improvements in other cases, we again decide to keep this index.

Note that we are not gathering any statistics on tables or indexes. If we would have done so, the system would have had statistics on how much CPU time has been used by different execution plans for the same query. This would have enabled the system to make better decisions and query execution time for any query would never had worsened. But for reasons described before, we do not want to do such a favour to our system.

Chapter 4

Materialized View Design: A Naive Approach

Having exhausted the tuning techniques provided by the used system (as far as we could per our knowledge), precomputing certain intensive operations in advance and storing the results in the database seems to be a very natural alternative to achieve further gains. This is what materialized view design is all about.

Section 4.1 gives a brief introduction to the technicality of materialized view and query rewrite utilities in the used system. The important point here is the restrictions on materialized views rather than what they can do. In Section 4.2 we develop a naive solution to the materialized view design problem. Section 4.3 compares the results of the power test for the system configured with our naive approach with the earlier system.

4.1 Technical Aspects of Materialized Views

Materialized views improve query performance by precalculating expensive join and aggregate operations on the database prior to execution time and storing the results in the database. The query optimizer can use materialized views by automatically recognizing when an existing materialized view can and should (if it brings an improvement in query execution time) be used to satisfy a request. It then transparently rewrites the request to use the materialized view. Queries are then directed to the materialized view and not to the underlying tables.

The syntax of materialized view creation in the used system is as follows:

```
CREATE MATERIALIZED VIEW my_mv
BUILD DEFFERED | IMMEDIATE
REFRESH COMPLETE | FAST | FORCE | NEVER
ON COMMIT | DEMAND
ENABLE | DISABLE QUERY REWRITE
AS
{sql-query}
```

There are other parameters not mentioned here as they are not of much interest at this point. The parameter names for each option have obvious meanings, so we describe only those which we have used to create our materialized views. The `BUILD` method `IMMEDIATE` creates the materialized view and populates it immediately with data. Setting the `REFRESH` option to `FAST` allows data to be updated incrementally when any of the base tables changes. The `ON COMMIT` clause enables automatic refresh when a transaction that modified one of the base tables commits. The materialized view is available for query rewriting only if `ENABLE QUERY REWRITE` is specified. The `AS` keyword is followed by the SQL query representing the view to be materialized.

The materialized view is not eligible for query rewrite until it is populated and is consistent with data in the base tables. To enable incremental refresh, materialized view logs have to be created. A materialized view log is a schema object (a table in the used system) that records changes to a master table's data. Each materialized view log is associated with a single table.

As mentioned earlier, we are interested in restrictions on materialized views which have to be created with the options described above. Only those restrictions which have had an impact on our materialized view design are described here. For further reading please refer to the product documentation of the used system [Lib]. The restrictions are as follows:

- Materialized views with joins and aggregates cannot have `ON COMMIT` option, so they cannot be used by us.
- The `WHERE` clause can contain only joins and they must be equi-joins. All join predicates must be joined with `AND`s. No selection predicates on individual tables are allowed.
- Materialized views with join only cannot have `GROUP BY` clause or aggregates.

- Materialized views with aggregates can have only a single table in the `FROM` clause. They cannot have `WHERE` clause.

Although a materialized view with joins and aggregates cannot be realised with the options we need, one can create nested materialized views. A nested materialized view is a materialized view whose definition is based on another materialized view. So to materialize joins and aggregates, we first create a materialized view with joins only and then use it to create an aggregate materialized view. A view created in such a way can then be refreshed incrementally on commit. Another advantage of nested materialized views is that, if many aggregate materialized views are created on a single join materialized view, the join has to be performed only once if any of the base table changes. Incremental maintenance of single-table aggregate materialized views is very fast due to the self-maintenance refresh operations on this class of views.

4.2 The Naive Approach

The naive solution to the materialized view design which we have developed is based on experimental results. The basic idea is to materialize views which result in maximum savings of query execution times in seconds. The update operation time should not exceed a given time limit. In our case this value is four minutes. The pseudo-code for the algorithm is as follows:

```
 1  NaiveMVD(Q:list of queries, U:list of update ops,
             L:max update op time)

 2    if (MaxExecTime(U) > L)
         return

 3    QT = GetQueryExecTimes(Q)
 4    MV = MaterializeViews(Q)
 5    MergeJoinViews(MV)

 6    PQ = EMPTY
 7    for each q in Q
 8      t_last = 0
 9      for each m in MV(q)
10        t = QT(q) - (GetQueryExecTime(Q, m) - t_last)
```

```
11          PQ = PQ + {(t - t_last, m)}
12          t_last = t

13      while (PQ != EMPTY && MaxExecTime(U) > L)
14        DropMaterializeView(HEAD(PQ))
```

The input to the algorithm is the set of queries, the set of update operations, and the maximum allowed time for an update operation. In Step 2, if the execution time of any of the update operations is more than the maximum time allowed for an update operation, no views are materialized and the algorithm ends. If this is not the case, the execution times for each query is stored in `QT` in Step 3. Step 4 materializes all queries and stores their names in `MV`. `MV` can contain more elements than `QT` as nested materialized views may have been used to materialize a single query. In Step 5, views having the same join conditions are merged. In Step 6 through 12, the priority queue `PQ` is filled with view names from `MV` according to the savings in the query execution times they brought through their materialization for the respective queries. `MV(q)` is the ordered set of all materialized views used to materialize `q`. If `m1` and `m2` are two views materialized for query `q` and `m1` is needed to materialize `m2`, then `m1` comes before `m2` in the ordered set `MV(q)`. In Step 10, the actual saving due to the materialization of `m` is calculated. This is the execution time of query `q` when no materialized view is present (`QT(q)`) minus the execution time of query `q` when `m` and views needed to materialize `m` are present. The quantity `t_last` is the saving due to the materialized views needed to materialize `m`. In Steps 12 and 13 we drop materialized views with the least amount of savings as long as the execution time of any update operation is more than the maximum allowed time `L`.

The value of `L` has to be fixed experimentally and may be different as according to the workload a system has to support. A rule of thumb would be to have this value not exceed the desired average query execution time. But one has to take care to note that an update operation to a base table may affect the execution of a large number of queries. This adverse affect may be evident only if many queries are being executed in parallel. In such a scenario, one would like to have the update operation time not exceed half or one-fourth of the desired average query execution time.

4.3 Results of Power Test

The results of the power test for the system configured with the naive algorithm are presented in Figure 4.1. Six views have been materialized. The exact creation directives for all materialized views are presented in Appendix B. The materialized view logs and the indexes created on the materialized views are also shown. The queries materialized are `Q0, Q3,` and `Q4`. While, `Q0` and `Q4` are materialized with single aggregate views, `Q3` has been materialized using a nested view which requires two view materializations. The other two views materialize parts of `Q7` and `Q13`. In all there are two join materialized views and four single aggregate materialized views.

Some query execution times have increased (`Q1` and `Q2`), although the execution plans for these are still the same. On the other hand, query execution times have decreased for many queries, other than the ones which use materialized views (`Q5`, `Q6`, `Q8`, and `Q11`). As these changes are not execution plan dependent changes, we suppose that these affects are due to certain environment dependent variable values at the time of the execution.

The important point to note here is the increase in update operation execution times. Whereas in the tuned environment these were in milliseconds, they are now in minutes. Remember that we set the limit of the maximum time consumed by any update operation to four minutes. Materialization of any other view using the `LINEITEM` table has increased the query execution time beyond four minutes for the delete operation in this table.

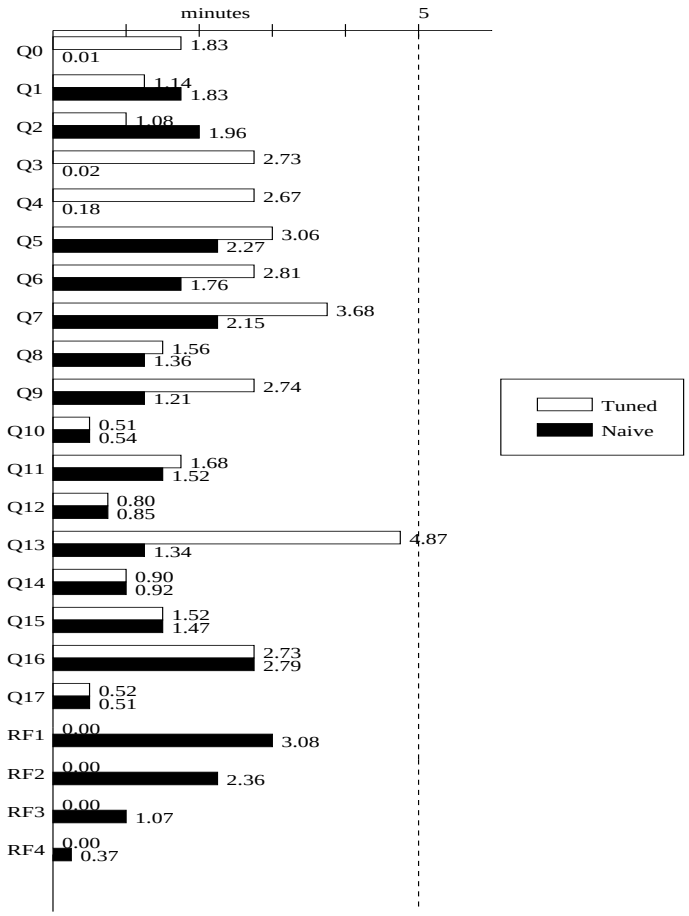

Figure 4.1: Comparison of the Query Execution Times of the Tuned System vs. the System with Naively Selected Materialized Views

Chapter 5

Algorithms for Materialized View Design

The materialized view design problem studied by us (see Section 1.1) has been addressed in [JYL97] and [TS97]. These two algorithms use completely different approaches to solve the materialized view design problem. Both of them define their own data structures which are then used in solving the given problem. The 0-1 integer programming approach presented in [JYL97] uses a *Multiple View Processing Plan* (MVPP) and the space state search in [TS97] a *Multiquery Graph* for their purposes.

In the following, we explain these algorithms with the help of examples. We briefly describe the data structures and cost models they use. Pseudo-code for each of the algorithms is also provided. Section 5.3 presents the results of the power tests for the system configured with these algorithms and the earlier configurations.

5.1 The 0-1 Integer Programming Approach

In [JYL97] the materialized view design problem is mapped to 0-1 integer programming problem. They present the problem formally using a multiple view processing plan and provide a cost model for materialized view design in terms of query performance as well as view maintenance. An optimal MVPP is generated using a 0-1 integer programming approach. Given such an MVPP and the cost model, a heuristic then selects the views to materialize.

5.1.1 Multiple View Processing Plan

An MVPP specifies the views that the data warehouse will maintain (either materialized or virtual).

An MVPP is a labeled directed acyclic graph $\mathcal{M} = (V, A, C_a^q, C_m^r, f_q, f_u)$ where V is a set of vertices, A is a set of arcs over V, such that

- for every relational algebra operation in a query tree, for every base relation, and for every distinct query, there is a vertex;
- leaf nodes correspond to the base relations and root nodes to global queries. $L \subset V$ is the set of leaf nodes and $R \subset V$ the set of root nodes.
- for every $v \in L$, $f_u(v)$ represents the update frequency of v and for every $v \in R$, $f_q(v)$ represents the query access frequency of v;
- if the base or intermediate result relation corresponding to vertex u is needed for further processing at a node v, introduce an arc $u \rightarrow v$;
- for every vertex v, $S(v)$ denotes the source nodes which have edges pointed to v and $S^*(v) = S(v) \cup \{\cup_{v' \in S(v)} S^*(v')\}$ the set of descendants of v;
- for every vertex v, $D(v)$ denotes the destination nodes to which v is pointed and $D^*(v) = D(v) \cup \{\cup_{v' \in D(v)} D^*(v')\}$ the set of ancestors of v;
- find all pairs of distinct vertices $u, v \in V$ such that $S^*(v) = S^*(u)$ and $v = u$, and merge them as they are common subexpressions.

Figure 5.1 shows how an MVPP may look like for an application with the following schema:

```
Item(I_id, I_name, I_price)
Part(P_id, P_name, I_id, number)
Sales(I_id, month, year, amount)
```

and the following queries:

```
Q1: Select   I_id, sum(amount*I_price)
    From     Item, Sales
    Where    I_name like {MAZDA, NISSAN, TOYOTA}
    And      year=1996
```

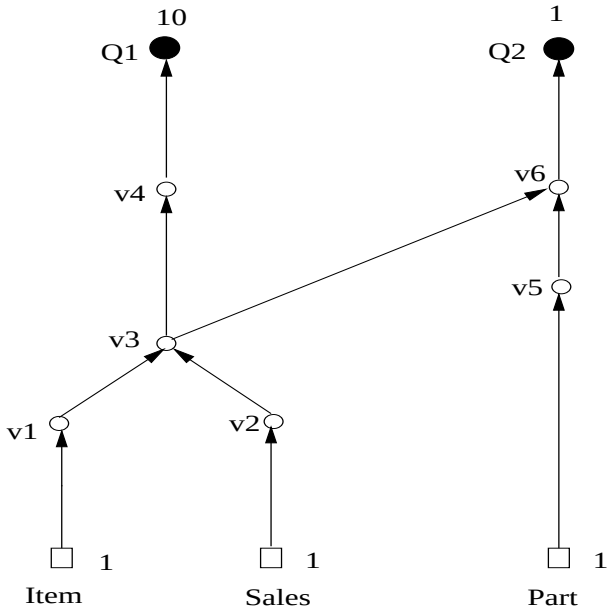

Figure 5.1: MVPP Example

```
    And      Item.I_id=Sales.I_id
    Group by I_id

Q2: Select   P_id, month, sum(amount*number)
    From     Item, Sales, Part
    Where    I_name like {MAZDA, NISSAN, TOYOTA}
    And      year=1996
    And      Item.I_id=Sales.I_id
    AND      Part.P_id=Item.I_id
    Group by P_id, month
```

The nodes $v1$ to $v6$ represent the following views:

```
v1: Select   *
    From     Item
    Where    I_name like {MAZDA, NISSAN, TOYOTA}

v2: Select   *
    From     Sales
```

```
    Where    year=1996

v3: Select   *
    From     v1, v2
    Where    v1.I_id=v2.I_id

v4: Select   I_id, sum(amount*I_price)
    From     v3
    Group by I_id

v5: Select   *
    From     v3, Part
    Where    v3.I_id=Part.P_id

v6: Select   P_id, month, sum(amount*number)
    From     v5
    Group by P_id, month
```

5.1.2 Cost Model

The cost of answering a query Q is the number of rows present in the table used to construct Q. The following cost model uses this assumption as basis for computing query processing and view maintenance costs.

Let M be a set of views in an MVPP to be materialized. For $v \in V$, $C_a^q(v)$ is the cost of query q accessing v; $C_m^r(v)$ is the cost of maintaining v based on changes to the base relation, if v is materialized.

The query processing cost is

$$C_{queryprocessing}(v) = \sum_{q \in R} f_q C_a^q(v)$$

The materialized view maintenance cost is

$$C_{maintenance}(v) = \sum_{r \in L} f_u C_m^r(v)$$

The total cost of materializing a view v is

$$C_{total}(v) = \sum_{q \in R} f_q C_a^q(v) + \sum_{r \in L} f_u C_m^r(v)$$

The total cost of materializing all $v \in M$ is

$$C_{total} = \sum_{v \in M} C_{total}(v)$$

For every view v in individual query access plan, an $Ecost(v)$ function is introduced which represents the benefit of sharing a view among multiple views, and is defined on each view as follows:

$$Ecost(v) = \sum_{q \in R} f_q C_a^q(v)/n_v$$

where n_v is the number of queries which can share view v.

For the example in Section 5.1.1, let the $Item$ relation have 1000 rows and the $Sales$ relation 12 million rows. Then $v3$ has 36 million rows as the result of filtering the `Item` table has three rows. Let us assume that $v3$ is materialized. Then the query processing cost for $Q1$ is 10 * 36 million and the view maintenance cost for $v3$ is 2*(36 million + 12million + 1000).

The total cost for an MVPP is the sum of all query processing and view maintenance costs.

5.1.3 Generating Optimal Multiple View Processing Plan

In [JYL97], the MVPP generation problem is modeled as 0-1 integer programming problem. For simplicity they assume that all the select, project and, aggregate operations have been pushed up and they only consider join operations. We stick to this assumption in our implementation too.

In the following we first define some notations used in the algorithm and then present the algorithm for optimal multiple view processing plan generation.

- *join plan tree* for a query is a binary tree with join operations as nodes. A query may have more than one join plan trees. Figure 5.2 shows the join plan trees for our example Q1 and Q2 from Section 5.1.1. Let $p(q)$ denote the set of all possible join plan trees of query q and let $P = \cup_{i=1}^{k} p(q_i)$ where k is the total number of queries. For our example we have:
  $p(Q1) = \{(It, Sal)\} = \{p1\}$
  $p(Q2) = \{((It, Sal), Pt), ((It, Pt), Sal)\} = \{p2, p3\}$
  $P = \{p1, p2, p3\}$

- *join pattern* for a join plan tree p is a subtree of p. Let $s(p)$ denote the set of all patterns of join plan tree p and let $S = \cup_{i=1}^{l} s(p_i)$ be the set of all

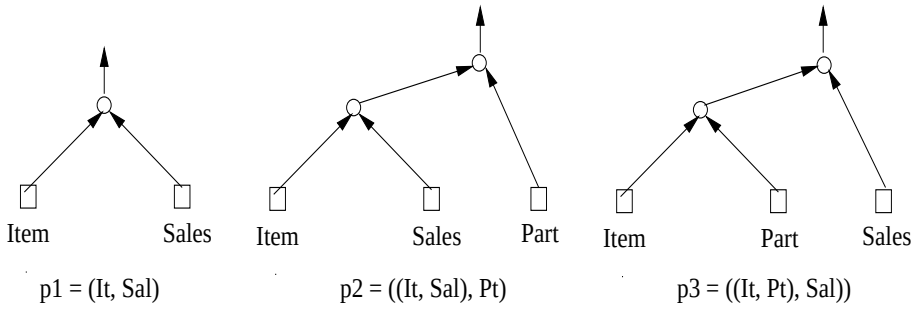

Figure 5.2: Join Plan Trees

possible join patterns for all the join plan trees. For our example we have:
$s(p1) = \{(It, Sal)\} = \{s1\}$
$s(p2) = \{(It, Sal), ((It, Sal), Pt)\} = \{s1, s2\}$
$s(p3) = \{(It, Pt), ((It, Pt), Sal)\} = \{s3, s4\}$
$S = \{s1, s2, s3, s4\}$

The following algorithm takes a set of queries and the cost function defined in Section 5.1.2 as input and returns a set of join plan trees representing the optimal MVPP:

```
MVPPGenerator(Q: list of queries, Ecost: cost function)
  P = get_join_plan_trees(Q)
  S = get_join_patterns(P)
  K = |Q|
  L= |P|
  M = |S|
  min_cost = infinity
  A: K * L Matrix of Bits
  B: M * L Matrix of Bits
  Unit, X, Y : L * 1 Matrix of Bits

  for i = 1 to L
    Unit[i, 1] = 1
    X[i, 1] = 0
    Y[i, 1] = 0
  for i = 1 to K
    for j = 1 to L
```

```
    if (P[j] is a join plan tree for Q[i])
      A[i, j] = 1
    else
      A[i, j] = 0
for i = 1 to M
  for j = 1 to L
    if (S[j] is a join pattern for P[i])
      B[i, j] = 1
    else
      B[i, j] = 0
for each permutation of Y
  if (A*Y == Unit)
    cost = 0
    for i = 1 to M
      for j = 1 to L
        cost = cost + B[i, j] * Y[j, 1]
      cost = cost * Ecost(S[i])
    if (cost < min_cost)
      X = Y
for i = 1 to L
  if (X[i, 1] == 0)
    delete P[i] from P

return P
```

The algorithm constructs two matrices, $A(K \times L)$ whose element a_{ij} is 1 if query q_i can be answered by join plan tree p_j, and $B(M \times L)$ whose element b_{ij} is 1 if pattern s_i is contained in the join plan tree p_j. Then the algorithm selects a subset of join plan trees and stores their indexes in the $L \times 1$ matrix X such that $\sum_{i=1}^{i=M} Ecost(s_i) * \{\sum_{j=1}^{j=L} b_{ij} * x_j\}$ is minimum and for every query exactly one join plan tree is selected; that is, $\sum_{j=1}^{j=L} a_{ij} * x_i = 1$ for every query i.

The matrices A and B for our example look like these:

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

5.1.4 The Algorithm

Given an MVPP, the algorithm described in this section finds a set of views to be materialized such that the total cost for query processing and view maintenance is minimal.

Apart from the notations defined in Section 5.1.1, the algorithm uses the following notations:

- $O(v)$ denotes the global queries which use v, $O(v) = R \cap D^*v$.
- $I(v)$ denotes the base relations which are used to produce v, $I(v) = L \cap S^*v$.
- $w(v) = \sum_{q \in O_v} f_q(q) * C_a^q(v) - \sum_{r \in I_v} f_u(r) * C_m^r(v)$ denotes the weight of the node. The first part of this formula indicates the saving if node v is materialized, the second part indicates the cost for materialized view maintenance.
- LV is the list of nodes based on descending order of $w(v)$.

Although this algorithm does not directly has anything to do with 0-1 integer programming (but the MVPP given as parameter to this algorithm), we name this algorithm `ZeroOneInt()` so as to identify the approach used in solving the materialized view design problem.

```
ZeroOneInt(mvpp: MVPP)
  M = {}
  LV = all_nodes(mvpp)
  sort(LV) /* according to the descending order of
              weight of each node */

  while (LV not empty)
    v = first(LV)
    cost = 0
    for each node q in O(v)
      cost = cal_cost(v)
    if (cost > 0)
      M = M + {v}
      delete(LV, v)
    else
      delete(LV, O(v)+v)
```

```
for each v in M
  if D(v) is a subset of M
    delete(M, v)
```

The algorithm sorts all the nodes of the given MVPP according to their weights in descending order and considers them for materialization. The function `cal_cost()` called for each node in the algorithm calculates

$$C_s = \sum_{q \in O_v} f_q(q) * (C_a^q(v) - \sum_{u \in S_v \cap M} C_a^q(u)) - \sum_{r \in I_v} f_u(r) * C_m^r(v)$$

where $\sum_{u \in S_v \cap M} C_a^q(u)$ is the replicated saving in case some descendants of v are already chosen to be materialized. $C_s > 0$ for a node n means that the materialization of n brings an overall saving to the excution times of the queries and should hence be materialized. Otherwise, n and all ancestors of n having weight less than n (i.e. those which have not yet been considered for materialization), should not be materialized.

The materialized views suggested by this algorithm for the workload of `1:1` (see Section 2.3) are shown in Chapter 6.

5.2 The State Space Search Approach

[TS97] formulates the problem of materialized view design as a state space optimization problem and then solves it using an exhaustive incremental algorithm. A state space search algorithm searches for states having minimal operational cost. A state in this case represents a possible solution to the materialized view design problem. A multiquery graph serves the purpose of defining their notion of state. A set of rules can be applied on a state which transform one state into another. Given an initial state and the transition rules, the algorithm searches for states with minimal operational cost.

5.2.1 States

In the following, we first define a multiquery graph and then present their definition of a state.

Let V be a set of views and R the set of relations appearing in V, then the corresponding multiquery graph G^V is defined as follows:

- for each $r \in R$ create a node in G^V.

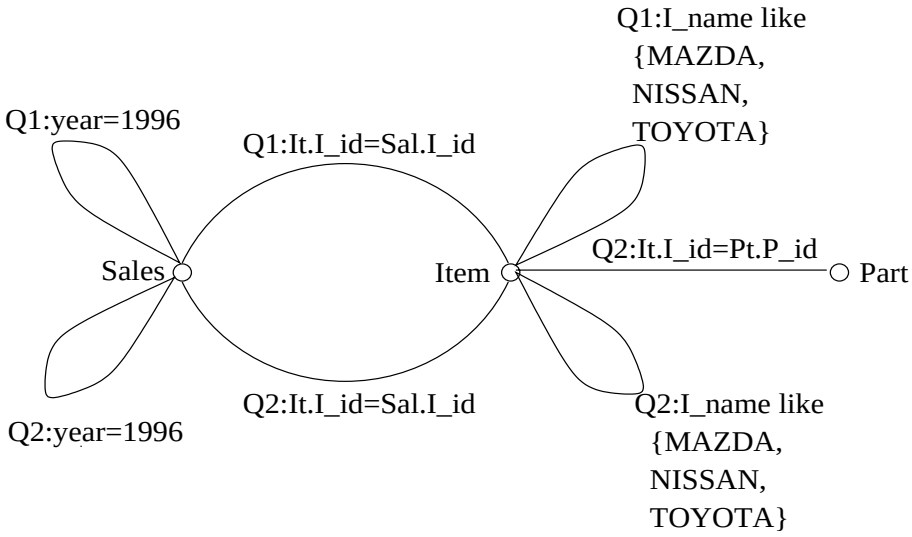

Figure 5.3: Multiquery Graph

- for each join predicate p in $v \in V$ involving attributes of $r_i, r_j \in R$, introduce an edge labeled $v : p$ between r_i and r_j and mark the type of such an edge as *join edge*.
- for each selection predicate p in $v \in V$ involving attributes of $r \in R$, introduce an edge from r to itself labeled $v : p$ and mark the type of such an edge as *select edge*.
- if for some $v \in V$ and $r \in R$, $v = r$, then intoduce an edge from r to itself labeled as $v : *$ and mark the type of such an edge as select edge.

Given the queries $Q1$ and $Q2$ of Section 5.1.1, Figure 5.3 depicts the corresponding multiquery graph $G^{\{Q1,Q2\}}$.

Given a set of views V, a *state* in [TS97] is defined as a pair $< G^V, Q^V >$ where G^V is a multiquery graph and Q^V is a rewriting of Q over V.

5.2.2 The Transitions

A transition from one state to another occurs whenever one of the following state transformation rules is applied to a state $< G^V, Q^V >$:

- *select edge cut:* if e is a select edge in G^V of a node R labeled as $v : p$, construct a new multiquery graph as follows: (a) if e is the unique edge of R, then replace its label by $u : *$, where u is a new view name (b) else remove e from G^V and replace any occurence of v in G^V by a new view name u. Replace any occurence of v in Q^V, by the expression $\sigma_p(u)$. Figure 5.4 shows the multiquery graph of figure 5.3 after this rule is applied on the edge labeled $Q1 : year = 1996$. The query $Q1$ transforms to:

```
Q1: Select   I_id, sum(amount*I_price)
    From     Item, v1
    Where    I_name like {MAZDA, NISSAN, TOYOTA}
    And      Item.I_id=v1.I_id
    Group by I_id

v1: Select *
    From Sales
    Where year=1996
```

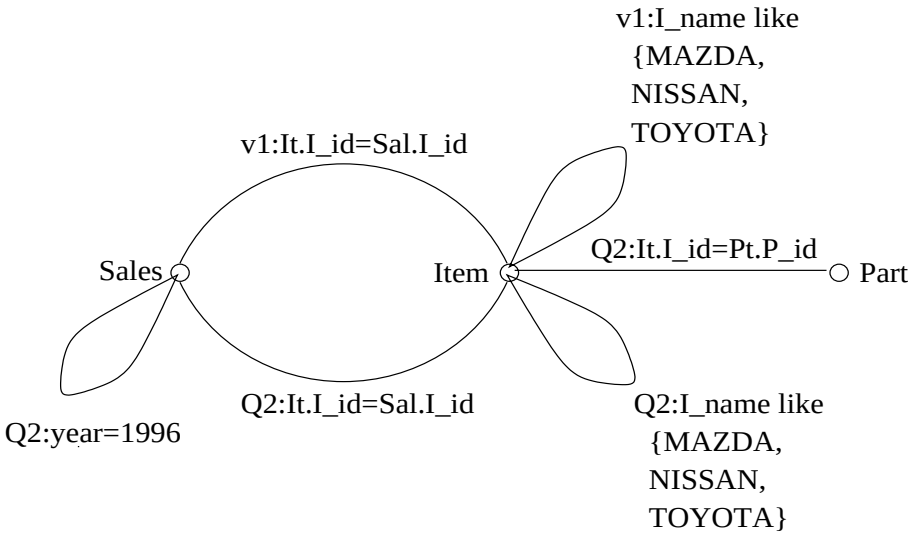

Figure 5.4: Select Edge Cut

- *join edge cut:* if e is a join edge labeled as $v : p$ in G^V, remove e from G^V and construct a new multiquery graph, as follows: (a) if the removal of e

does not divide the query graph of v in G^V into two disconnected components, then replace every occurence of v in G^V by a new view name u. (b) otherwise, replace every occurence of v in G^V by a new view name u in the one component and by a new view name w in the other component. If a component is a single node without edges, then add in G^V a selection edge to this node labeled as $u : *$. In this case u is a base relation. Replace any occurence of v in Q^V, in case (a) above, by the expression $\sigma_p(u)$ and in case (b), by the expression $\sigma_p(u \times w)$. Figure 5.5 shows the multiquery graph of figure 5.4 after this rule is applied on the edge labeled $Q2 : It.I_id = Pt.P_id$. The query $Q2$ transforms to:

```
Q2: Select   P_id, month, sum(amount*number)
    From     v2, v3
    Where    v2.I_id=v3.P_id
    Group by P_id, month

v2: Select *
    From   Sales, Item
    Where  I_name like {MAZDA, NISSAN, TOYOTA}
    And    year=1996
    And    Item.I_id=Sales.I_id

v3: Select *
    From Parts
```

- *view merging:* If query graphs of two views v and u in G^V have the same set of nodes and each predicate in their query graphs is either implied by a predicate of the other view or implies a predicate of the other view, then construct a new multiquery graph as follows: remove from G^V each edge labeled by a predicate of one of the views that imply a predicate of the other view and is not implied by a predicate of the first view. Replace any occurence of v and u in G^V by a new view name w. Replace any occurence of v in Q^V by the expression $\sigma_P(v)$, where P is the conjuction of the predicates p in v such that: p implies a predicate in u and is not implied by a predicate in v. Do similarly for u. Figure 5.6 shows the multiquery graph of Figure 5.5 after select edge cut is applied on the edge $v2 : year = 1996$ and then view merging is applied. The queries transform to:

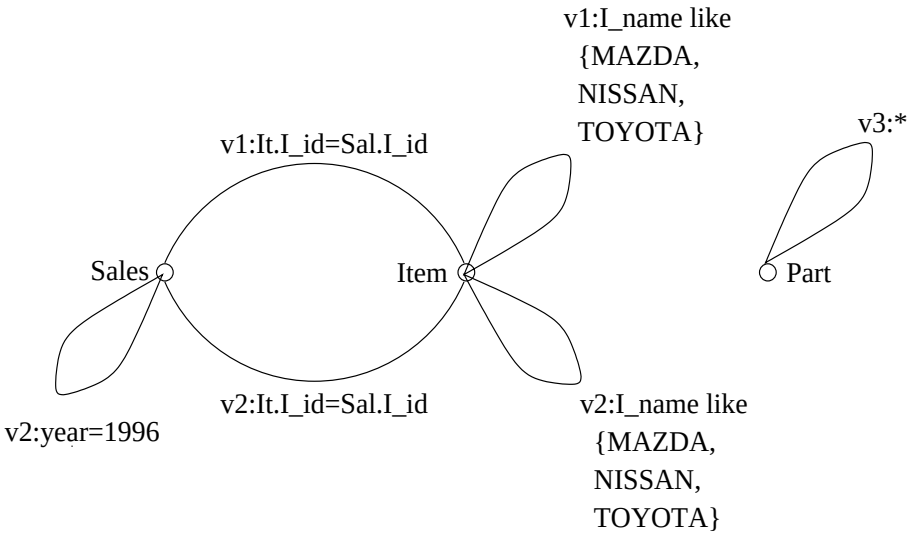

Figure 5.5: Join Edge Cut

```
Q1: Select   I_id, sum(amount*I_price)
    From     v5
    Group by I_id

Q2: Select   P_id, month, sum(amount*number)
    From     v3, v5
    Where    v5.I_id=v3.P_id
    Group by P_id, month

v5: Select *
    From   Sales, Item
    Where  I_name like {MAZDA, NISSAN, TOYOTA}
    And    Item.I_id=Sales.I_id
```

5.2.3 Cost Model

The operational cost of a state $s =< G^V, Q^V >$ is given by

$$Cost(s) = E(Q^V) + cM(V)$$

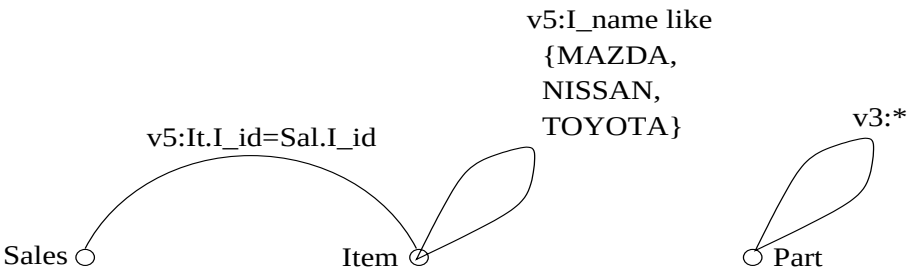

Figure 5.6: View Merging

where $E(Q^V) = \sum_{q \in Q} f^q E(q)$ is the query evaluation cost and $M(V) = \sum_{v \in V} f^v M(v)$ the view maintenance cost. f represents the relative frequency of the respective operation. The parameter c is the relative importance of the maintenance operation. In our case c is always one meaning that the updates to the base relations are to be integrated in the data warehouse immediately.

The cost model presented in [TS97] relies on the cost estimated by the optimizer of the used system. So $E(q)$ is the cost of query q estimated by the cost based optimizer (see Section 3.2.1). The view maintenance cost $M(v)$ can not be estimated by the optimizer. So we take it to be the sum of costs of updating a base relation and that of updating all materialized views depending on this base relation. For example, let t be a table with $n1$ rows and the cost of updating t be c, then cost of updating a materialized view mv with $n2$ rows and depending on t is $n2 * (n1 \times c)$.

5.2.4 The Algorithm

The algorithm gets an initial state $s =< G^Q, Q^Q >$ as input. For a set of queries, the initial state is constructed as described in Section 5.2.1. It then produces all subsequent states using the state transformation rules of Section 5.2.2. In the end, the state with minimum operational cost is returned.

```
StateSpaceSearch(s: State)
  open: Set of State
  closed: Set of Pair(State, Cost)
  open = {s}

  while (open != empty)
    curr_state = first(open)
```

```
    for every transition of curr_state to new_state
      if (new_state not in open or closed)
        open = open + {new_state}
    open = open - {curr_state}
    closed = closed + {(curr_state, Cost(curr_state))}
  return min_cost(closed)
```

The initial state `s`, given as input to the algorithm represents the case when all queries are materialized. The set `open` keeps track of all states which should be considered for the application of the transformation rule and the set `closed` stores all states which have already been considered. The states in `open` are worked upon one after another. Successful application of any of the trasformation rules results in a new state, which is then stored in `open` for further processing. In the end, one has all possible states with their costs in the `closed`. The state with minimum cost is then returned.

The materialized views suggested by this algorithm for the workload of `1:1` (see Section 2.3) are shown in Chapter 6. As already mentioned, materialized views in the used system cannot have selection predicates. So we do not consider any selection edge in the above algorithm. To reduce the search space, we consider only those join edges which have the node representing at least one of `LINEITEM` and `ORDERS` tables as adjacent node. Remember these are the only two tables subject to update operations. As there are no update operations to other tables, we materialize joins between these tables wherever it brings some performance gains.

5.3 Results of Power Test

Figure 5.7 presents the results of power test for all the configurations we tend to compare. We shall do a thorough analysis and comparison of the execution times for each operation in Chapter 6. In this section we just make note of the changes which are evident at first sight of Figure 5.7.

The points to be noted are:

- There is not much difference between execution times for the whole system configured with any of the algorithms. If one has better execution time in one case, then the other is better in the other case. The same is true when comparing the execution times with the tuned system.
- The query execution times for the system with naive configuration are the best, apart from some minor worsenings.

- The execution times for delete operations for the system configured with any of the algorithms are very high.
- The naive approach is the only one resulting in measurable execution times for insert operations.

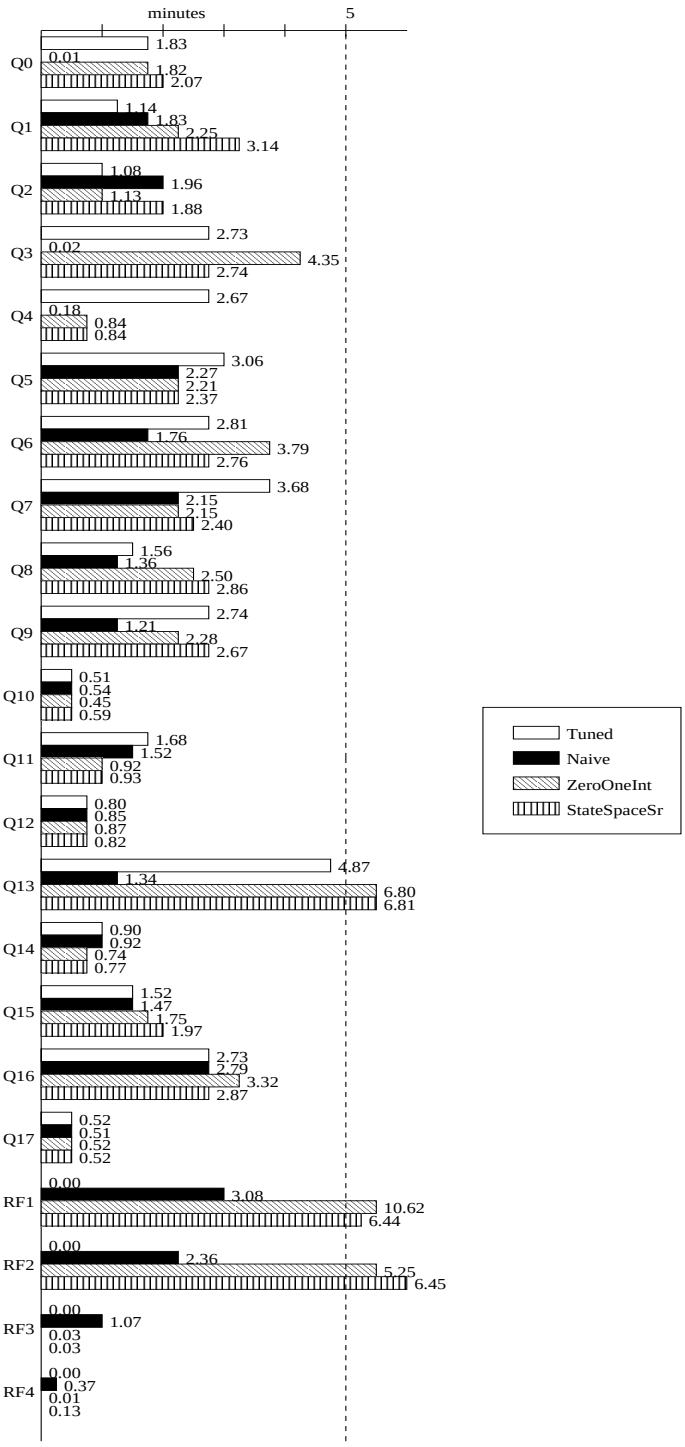

Figure 5.7: Power Test Results for all Configurations

Chapter 6

Results, Analysis and Reasoning

In Section 6.1 we present the views which have been materialized in case of both the algorithms and compare the results of the power as well as throughput test for the different configurations. In Section 6.2, we try to find answers to the questions raised due to analysis of the results.

6.1 Results and Analysis

```
ZMVD1:  Q1, Q2, Q9, Q13
   select * from lineitem, orders where l_orderkey = o_orderkey
ZMVD2:  Q8
   select * from customer, nation where c_nationkey = n_nationkey
ZMVD3:  Q15, Q7
   select * from lineitem, partsupp where l_suppkey = ps_suppkey
     and l_partkey = ps_partkey
ZMVD4:  Q11, Q12, Q14
   select * from lineitem, part where l_partkey = p_partkey
ZMVD5:  Q3, Q6, Q16
   select * from lineitem, supplier, orders, nation
     where l_suppkey = s_suppkey
     and l_orderkey = o_orderkey
     and s_nationkey = n_nationkey
ZMVD6:  Q7, Q15, Q16
   select * from supplier, nation where s_nationkey = n_nationkey
```

Before we proceed to analyse the results presented in Figures 5.7 and 6.2, we present the views materialized in case of both the algorithms. Each materialized view is given a name followed by the queries (see Appendix A) using this materialized view and the SQL query materialized by the respective view. The box above shows the materialized views suggested by `ZeroOneInt()` algorithm and the box below those suggested by the `StateSpaceSearch()` algorithm for a workload of `1:1` (see Section 2.3).

```
SMVD1:  Q2, Q9, Q16
   select * from lineitem, orders where l_orderkey = o_orderkey
SMVD2:  Q10, Q17
   select * from customer, orders where c_custkey = o_custkey
SMVD3:  Q15
   select * from lineitem, partsupp where l_suppkey = ps_suppkey
     and l_partkey = ps_partkey
SMVD4:  Q11, Q12, Q14
   select * from lineitem, part where l_partkey = p_partkey
SMVD5:  Q7
   select * from lineitem, supplier, orders, partsupp, part
     where l_suppkey = s_suppkey
     and l_orderkey = o_orderkey
     and l_partkey = p_partkey
     and l_suppkey = ps_suppkey
     and l_partkey = ps_partkey
SMVD6:  Q1, Q8, Q13
   select * from customer, orders, lineitem
     where c_custkey = o_custkey
     and l_orderkey = o_orderkey
SMVD7:  Q3, Q5, Q6
   select * from customer, orders, lineitem, supplier
     where c_custkey = o_custkey
     and l_orderkey = o_orderkey
     and l_suppkey = s_suppkey
```

Having presented the results of the power test (see Figure 5.7) for each configuration and analyzed them briefly in the respective chapters, Figure 6.2 presents the results of the throughput test for a workload of `1:1` (see Section 2.3). We have been able to run the throughput test with a workload of `1:2` only for the naive approach. For the other approaches, the tests were aborted in the middle as either

the process memory was exhausted or no more processe to start the execution of a new query was available[1].

Let us first look at the throughput test results of Figure 6.2 and note the points evident at first sight as we did in the case of power test results in Section 5.3.

- As in case of power test, there is not much difference between execution times for the system configured with any of the algorithms.
- In contrast to the results of power test, the same is not true when comparing the execution times with that of the tuned system.
- The query execution times for the system with naive configuration are the best, apart from minor worsening in case of `Q14`, when compared to the tuned case.
- The execution times for update operations for the tuned system are the best (almost negligible).
- The execution times for delete operations for the system configured with any of the algorithms are very high.
- The naive approach is the only one resulting in measurable execution times for insert operations.

Now let us start comparing the execution times for each query individually for both power and throughput tests for the different configurations. The execution times for query `Q0` are almost same in case of power test for the system configured with any of the algorithms and the tuned system. `Q0` is fully materialized in case of naive approach and thus has almost negligible execution time in this case. In fact, `Q0` being an aggregate query on a single table, is not considered for materialization by `ZeroOneInt()` algorithm at all. In case of `StateSpace-Search()` algorithm, the technical limitation on materialized views (see Section 4.1) do not allow a selection edge to be considered and thus leaving `Q0` not to be materialized. Apart from the naive configuration, all other configurations follow the same execution plan. In case of throughput test, there is a worsening of execution times for `Q0` for the system configured with any of the two algorithms as compared to the tuned system. This clearly is a result of system overloading due to parallel execution of operations, not to mention the delete update operations

[1]These are error messages ORA-04030 and ORA-27142 in the used system

which take substantial amount of time in the case of both algorithms. The same observation is actually true in the case of Q4, except that the execution times for power test for the system configured with the two algorithms are somehow far better than that for the tuned system. We do not have an explanation for this as the execution plans for these cases are exactly the same.

In case of power test results for query Q1, it seems that the more parts of Q1 you materialize, the more time it takes to execute the query. Q1 is supported by materialized view ZMVD1 and SMVD6 in the respective configurations. Whereas, ZMVD1 materializes just one join condition, SMVD6 materializes an additional join condition. But the results are exactly the opposite. Execution of Q1 takes longer when it uses SMVD6 as compared to using ZMVD1. The best results are achieved with the tuned system, i.e. not materializing any part of Q1. The discrepancy in the execution times for the tuned and the system configured with the naive approach can only be blamed on the change in some environment variables at the time of execution as they follow the same execution plans. Rather unusual is the shooting up of execution time of Q1 for the tuned system in case of throughput test. Apart from such environment dependent inexplicable differences in query execution times (which follow the same execution plan), other queries showing the same affect as that in the case of Q1 are: Q2, Q6, Q8, Q13, and Q15. In all these cases, the two algorithms suggest to materialize one part or the other of the queries leading to the worsening of query execution times as compared to the tuned system. In case of naive configuration, the same execution plan is followed as that for the tuned system except in case of Q13 which is partly materialized by the naive approach (see Section 4.3).

Query Q3 has been fully materialized with the help of a nested materialized view (see Section 4.3) in case of naive configuration reducing query execution time to almost zero. Both the algorithms suggest to materialize a large number of join conditions (ZMDV5 and SMVD7 respectively). Whereas, the state space approach does not show any gains in the case of power test, the execution time for Q3 in case of 0-1 integer approach suffers clearly. The throughput test results show hardly any difference in execution times for this query for all the configurations, naturally except for the naive configuration.

Materializing views suggested by both the algorithms has not always had a negative affect on the execution times fo the queries. Queries Q5, Q7, Q9, Q11, and Q14 record improvements in query execution times (at least in the case of power test), whereas in case of Q10, Q12, and Q17 all the configurations

are at par.[2]

Coming to the update operations, the tendency is the same in case of power and throughput tests for all the configurations. The naive and 0-1 integer programming approaches materialize six views, whereas state space approach seven. Still, there is a big difference in the execution times for the system configured with the naive and the other two approaches. The reasons for this and the above observations are given in the following section.

6.2 Questions & Answers

Having analysed the results for all the queries and update operations, we can now proceed to find answers to the questions raised in the previous section. For that matter, we first formulate the question and then try to give a satisfactory answer to it.

Question 1: When does view materialization have a negative impact on query execution, even when there are no parallel operations?
Answer: This question can be best answered by looking at execution plan of a query which has a better execution time for the tuned system than for the system configured with any of the algorithms. Let us have a look at the execution plans for query `Q1` for all the three configurations.

```
The tuned system:
-------------------------------------------------------------
| Operation               |  Name    | Rows | Bytes| Cost |
-------------------------------------------------------------
| SELECT STATEMENT        |          |   18G| 2207G|    1G|
|  SORT ORDER BY          |          |   18G| 2207G|    1G|
|   SORT GROUP BY         |          |   18G| 2207G|    1G|
|    SORT GROUP BY        |          |   18G| 2207G|    1G|
|     HASH JOIN           |          |   18G| 2207G| 18277|
|      TABLE ACCESS FULL  |LINEITEM  |  440K|   20M| 16366|
|      HASH JOIN          |          |    4M|  317M|  1642|
|       TABLE ACCESS FULL |CUSTOMER  |    4K|  128K|   814|
|       TABLE ACCESS FULL |ORDERS    |   97K|    4M|   826|
-------------------------------------------------------------
```

[2]We do not analyse `Q16` because of the reasons already mentioned.

```
The 0-1 integer programming approach:
-------------------------------------------------------------
| Operation               |  Name   | Rows | Bytes| Cost |
-------------------------------------------------------------
| SELECT STATEMENT        |         |  955K|  102M| 51105|
|  SORT ORDER BY          |         |  955K|  102M| 51105|
|   SORT GROUP BY         |         |  955K|  102M| 51105|
|    SORT GROUP BY        |         |  955K|  102M| 51105|
|     HASH JOIN           |         |  955K|  102M| 17033|
|      TABLE ACCESS FULL |CUSTOMER |    4K|  128K|   814|
|      TABLE ACCESS FULL |ZMVD1    |   21K|    1M| 16217|
-------------------------------------------------------------

The state space search approach:
-----------------------------------------------------
| Operation             |  Name | Rows | Bytes| Cost |
-----------------------------------------------------
| SELECT STATEMENT      |       |   542|   46K| 40356|
|  SORT ORDER BY        |       |   542|   46K| 40356|
|   SORT GROUP BY       |       |   542|   46K| 40356|
|    SORT GROUP BY      |       |   542|   46K| 40356|
|     TABLE ACCESS FULL |SMVD6  |   542|   46K| 40236|
-----------------------------------------------------
```

Let us just concentrate on the first two columns forming the execution plans. The other columns will be dealt with later. The first four rows of all three execution plans are the same. These rows represent sort operations. The tuned system executes two join operations, the system configured with 0-1 integer approach just one, and the system configured with state space approach none. The execution times for each of these configurations are exactly the opposite of what one would expect (see Figure 5.7).

Figure 6.1 takes a closer look at the three execution plans. The figure shows execution plans prior to the sort operations. Each node represents an operation, labeled on the left of the node and the number of rows in the table resulting after the execution of this operation, labeled on the right of the node. It seems that the gains in execution time due to materialized joins are undermined by the fact that the resulting materialized views are large tables which result in memory swaping and paging affects. On the contrary, when no views are materialized, the result of early filtering on tables is small enough (at least in the case of customer table) that it can remain resident in the memory for further operations.

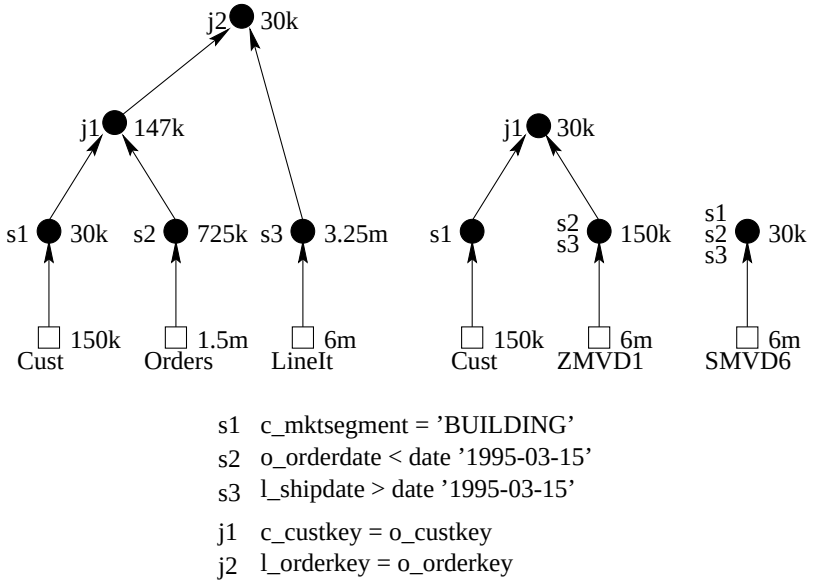

Figure 6.1: Execution Plans for Query `Q1` for different Configurations

Question 2: Why do the algorithms choose views to materialize which have such a negative impact?
Answer: Clearly, the answer of this question lies in the cost model. The cost models of the two approaches have been presented in Sections 5.1.2 and 5.2.3, respectively.

The 0-1 integer programming approach considers the number of rows in a table to be proportional to the time required to answer any query on this table. The maintenance cost is the cost of completely refreshing both the tables. Taking into account the technical stand of query optimizers, such a cost model is outdated. The query optimizer, as is evident from the above execution plans, tries to execute a query in such a way that the intermediate results are minimal. Two big tables, such as the `LINEITEM` and `ORDERS` tables in our case, are normally joined after they have been filtered to small sizes. On the contrary, the cost model of this approach tries to materialize views which are very big in size. Moreover, in case of update operations, this approach supposes a complete refresh. Such a refresh, with the views materialized for this approach, takes about 3-4 hours after a single

update operation (operation RF1). This is certainly not feasible.

The state space approach relies on the cost estimated by the optimizer. Here the case seems to be opposite than that for the 0-1 integer programming approach. The query optimizer, if not collecting statistics (see Section 3.2.1), do not seem to be estimating the costs correctly. This is also evident from the above execution plans where the estimated cost sould have been minimal in case of the tuned system.

Question 3: Why is there such a difference in the execution times for the update operations, although the number of materialized views do not vary much?
Answer: The reason for this lies solely in the technicalities of materialized views. An incremental delete in join materialized views takes much longer than in an aggregate materialized view. On the other hand, insert operations take longer to complete in an aggregate materialized view. These affects are evident from Figure 5.7. The naive approach materializes three aggregate views and only two join views, whereas in both the other configurations, only joins are materialized.

Question 4: Why does the naive approach make a better choice?
Answer: The naive approach has the advantage of relying only on experimental results. This makes it independent of the technical aspects of materialized views as well as theoretical aspects of the cost models. A very significant point to note is that execution times of queries which use aggregate materialized views are negligible. This leads to a sort of regeneration break in case of throughput test, where the system is heavily loaded and the queries get stocked in the system. This is also why only the naive approach has been able to bear a workload of 1:2 (see Section 2.3). The joins have also brought some relief but the execution time has never been negligible. This is basically because of the inability of materialized views in the used system to support select conditions in the WHERE clauses of the SQL statements.

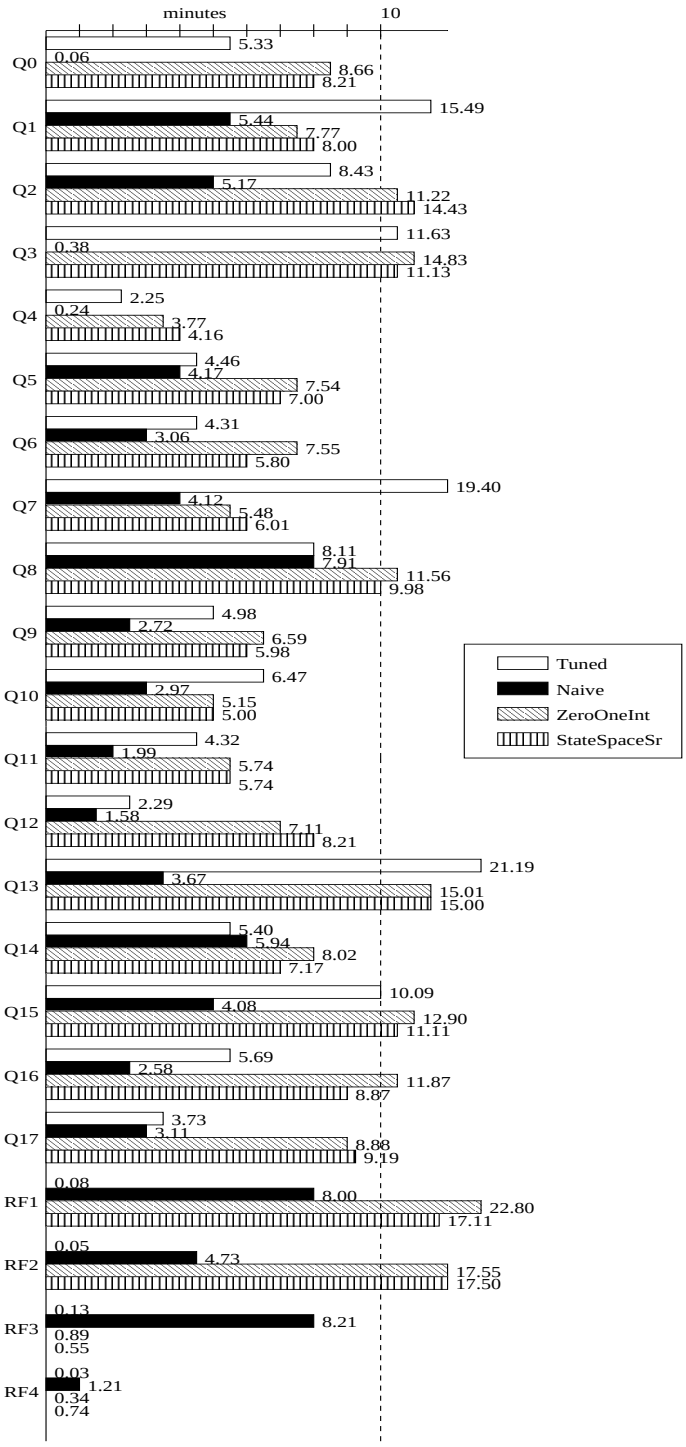

Figure 6.2: Throughput Test Results for all Configurations

Chapter 7

Conclusion

In this thesis we have presented an experimental study of materialized view design problem for the case when the data warehouse has to support occasional update operations. The underlying experimental environment has been provided by TPC-R. We have run our tests for the following four cases: without any materialized views (the system tuned with standard database tuning techniques), a naive approach to materialized view design, and materialized view design suggested by two different algorithms.

The conclusion we draw from our experiments are as follows:

- The algorithms in the literature are concentrated on join conditions. Aggregates are more or less neglected. As aggeregate operation costs are comparable with join operations and materializing aggregates brings far more savings in execution times than materializing joins (at least in the system used by us), a good materialized view design is bypassed by these algorithms. This is proven by our naive approach.

- There is a gap between theoretical and technical cost models. On the one hand the theoretical model fails to estimate the query and update operation costs correctly (0-1 integer programming) and on the other hand the cost estimates provided by the optimizer (state space approach) do not mirror the actual query execution times. A cost model depending on experimental results, as is the case in our naive approach, makes the algorithm free from any theoretical or technical flaws and provides a realistic cost measure.

- In contrast to materialized view design problem, the view maintenance problem is a well studied problem. The cost models provided by both the algo-

rithms do not refer to how the views should be maintained. In our opinion, materialized view design problem cannot be studied independent of the view maintenance problem, although one can talk about view maintenance without refering to the materialized view design problem.

Thanks

Thanks to Prof. Dr Heinz Schweppe for supporting me in every sense and believing in me that i would end the thesis some fine day.

Thanks to Dr. Agnes Voisard for satellite motivation.

Thanks to Dr. Daniel Faensen and Dr. Lukas Faulstich for their correction and suggestion work.

Thanks to the work group Databases and Information Systems for bearing my talks on my thesis from time to time and giving costructive suggestions.

Appendix A

TPC-R Queries

In this chapter, we present the TPC-R query set used by us. As already mentioned, we have left out some queries. The query numbers here are used as refrences in the thesis. For details on these queries please refer to [Tra]

```
Q0:
select
  l_returnflag,
  l_linestatus,
  sum(l_quantity) as sum_qty,
  sum(l_extendedprice) as sum_base_price,
  sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
  sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
  avg(l_quantity) as avg_qty,
  avg(l_extendedprice) as avg_price,
  avg(l_discount) as avg_disc,
  count(*) as count_order
from
  lineitem
where
  l_shipdate <= date '1998-12-01' - interval ':1' day (3)
group by
  l_returnflag,
  l_linestatus
order by
  l_returnflag,
  l_linestatus;
```

```
Q1:
select
  l_orderkey,
  sum(l_extendedprice * (1 - l_discount)) as revenue,
  o_orderdate,
  o_shippriority
from
  customer,
  orders,
  lineitem
where
  c_mktsegment = ':1'
  and c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate < date ':2'
  and l_shipdate > date ':2'
group by
  l_orderkey,
  o_orderdate,
  o_shippriority
order by
  revenue desc,
  o_orderdate;

Q2:
select
  o_orderpriority,
  count(*) as order_count
from
  orders
where
  o_orderdate >= date ':1'
  and o_orderdate < date ':1' + interval '3' month
  and exists (
    select
      *
    from
      lineitem
    where
      l_orderkey = o_orderkey
```

```
      and l_commitdate < l_receiptdate
  )
group by
  o_orderpriority
order by
  o_orderpriority;

Q3:
select
  n_name,
  sum(l_extendedprice * (1 - l_discount)) as revenue
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
  and r_name = ':1'
  and o_orderdate >= date ':2'
  and o_orderdate < date ':2' + interval '1' year
group by
  n_name
order by
  revenue desc;

Q4:
select
  sum(l_extendedprice * l_discount) as revenue
from
  lineitem
where
  l_shipdate >= date ':1'
```

```
  and l_shipdate < date ':1' + interval '1' year
  and l_discount between :2 - 0.01 and :2 + 0.01
  and l_quantity < :3;

Q5:
select
  supp_nation,
  cust_nation,
  l_year,
  sum(volume) as revenue
from
  (
  select
    n1.n_name as supp_nation,
    n2.n_name as cust_nation,
    extract(year from l_shipdate) as l_year,
      l_extendedprice * (1 - l_discount) as volume
  from
    supplier,
    lineitem,
    orders,
    customer,
    nation n1,
    nation n2
  where
    s_suppkey = l_suppkey
    and o_orderkey = l_orderkey
    and c_custkey = o_custkey
    and s_nationkey = n1.n_nationkey
    and c_nationkey = n2.n_nationkey
    and ( (n1.n_name = ':1' and n2.n_name = ':2')
       or (n1.n_name = ':2' and n2.n_name = ':1'))
    and l_shipdate between date '1995-01-01' and date '1996-
12-31'
  ) as shipping
group by
  supp_nation,
  cust_nation,
  l_year
order by
```

```
  supp_nation,
  cust_nation,
  l_year;

Q6:
select
  o_year,
  sum(case
    when nation = ':1' then volume
        else 0
    end) / sum(volume) as mkt_share
from
  (
    select
      extract(year from o_orderdate) as o_year,
      l_extendedprice * (1 - l_discount) as volume,
      n2.n_name as nation
    from
      part,
      supplier,
      lineitem,
      orders,
      customer,
      nation n1,
      nation n2,
      region
    where
      p_partkey = l_partkey
      and s_suppkey = l_suppkey
      and l_orderkey = o_orderkey
      and o_custkey = c_custkey
      and c_nationkey = n1.n_nationkey
      and n1.n_regionkey = r_regionkey
      and r_name = ':2'
      and s_nationkey = n2.n_nationkey
      and o_orderdate between date '1995-01-01' and date '1996-
12-31'
      and p_type = ':3'
  ) as all_nations
group by
```

```
  o_year
order by
  o_year;

Q7:
select
  nation,
  o_year,
  sum(amount) as sum_profit
from
  (
    select
      n_name as nation,
      extract(year from o_orderdate) as o_year,
      l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as a
    from
      part,
      supplier,
      lineitem,
      partsupp,
      orders,
      nation
    where
      s_suppkey = l_suppkey
      and ps_suppkey = l_suppkey
      and ps_partkey = l_partkey
      and p_partkey = l_partkey
      and o_orderkey = l_orderkey
      and s_nationkey = n_nationkey
      and p_name like '%:1%'
  ) as profit
group by
  nation,
  o_year
order by
  nation,
  o_year desc;

Q8:
select
```

```
  c_custkey,
  c_name,
  sum(l_extendedprice * (1 - l_discount)) as revenue,
  c_acctbal,
  n_name,
  c_address,
  c_phone,
  c_comment
from
  customer,
  orders,
  lineitem,
  nation
where
  c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate >= date ':1'
  and o_orderdate < date ':1' + interval '3' month
  and l_returnflag = 'R'
  and c_nationkey = n_nationkey
group by
  c_custkey,
  c_name,
  c_acctbal,
  c_phone,
  n_name,
  c_address,
  c_comment
order by
  revenue desc;

Q9:
select
  l_shipmode,
  sum(case
      when o_orderpriority = '1-URGENT'
      or o_orderpriority = '2-HIGH'
        then 1
        else 0
      end) as high_line_count,
```

```
  sum(case
      when o_orderpriority <> '1-URGENT'
      and o_orderpriority <> '2-HIGH'
        then 1
        else 0
      end) as low_line_count
from
  orders,
  lineitem
where
  o_orderkey = l_orderkey
  and l_shipmode in (':1', ':2')
  and l_commitdate < l_receiptdate
  and l_shipdate < l_commitdate
  and l_receiptdate >= date ':3'
  and l_receiptdate < date ':3' + interval '1' year
group by
  l_shipmode
order by
  l_shipmode;

Q10:
select
  c_count,
  count(*) as custdist
from
  (
    select
      c_custkey,
      count(o_orderkey)
    from
      customer left outer join orders on
        c_custkey = o_custkey
      and o_comment not like '%:1%:2%'
      group by
        c_custkey
  ) as c_orders (c_custkey, c_count)
group by
  c_count
order by
```

```
  custdist desc,
  c_count desc;

Q11:
select
  100.00 * sum(case
    when p_type like 'PROMO%'
      then l_extendedprice * (1 - l_discount)
      else 0
      end) / sum(l_extendedprice * (1 - l_discount)) as promo_revenue
from
  lineitem,
  part
where
  l_partkey = p_partkey
  and l_shipdate >= date ':1'
  and l_shipdate < date ':1' + interval '1' month;

Q12:
select
  sum(l_extendedprice) / 7.0 as avg_yearly
from
  lineitem,
  part
where
  p_partkey = l_partkey
  and p_brand = ':1'
  and p_container = ':2'
  and l_quantity < (
    select
      0.2 * avg(l_quantity)
    from
      lineitem
    where
    l_partkey = p_partkey
  );

Q13:
select
  c_name,
```

```
  c_custkey,
  o_orderkey,
  o_orderdate,
  o_totalprice,
  sum(l_quantity)
from
  customer,
  orders,
  lineitem
where
  o_orderkey in (
    select
      l_orderkey
    from
      lineitem
    group by
      l_orderkey having
      sum(l_quantity) > :1
    )
  and c_custkey = o_custkey
  and o_orderkey = l_orderkey
group by
  c_name,
  c_custkey,
  o_orderkey,
  o_orderdate,
  o_totalprice
order by
  o_totalprice desc,
  o_orderdate;

Q14:
select
  sum(l_extendedprice* (1 - l_discount)) as revenue
from
  lineitem,
  part
where
  (
    p_partkey = l_partkey
```

```
    and p_brand = ':1'
    and p_container in ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
    and l_quantity >= :4 and l_quantity <= :4 + 10
    and p_size between 1 and 5
    and l_shipmode in ('AIR', 'AIR REG')
    and l_shipinstruct = 'DELIVER IN PERSON'
  )
  or
  (
    p_partkey = l_partkey
    and p_brand = ':2'
    and p_container in ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
    and l_quantity >= :5 and l_quantity <= :5 + 10
    and p_size between 1 and 10
    and l_shipmode in ('AIR', 'AIR REG')
    and l_shipinstruct = 'DELIVER IN PERSON'
  )
  or
  (
    p_partkey = l_partkey
    and p_brand = ':3'
    and p_container in ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
    and l_quantity >= :6 and l_quantity <= :6 + 10
    and p_size between 1 and 15
    and l_shipmode in ('AIR', 'AIR REG')
    and l_shipinstruct = 'DELIVER IN PERSON'
  );

Q15:
select
  s_name,
  s_address
from
  supplier,
  nation
where
  s_suppkey in (
    select
      ps_suppkey
    from
```

```
      partsupp
    where
      ps_partkey in (
        select
          p_partkey
        from
          part
        where
          p_name like ':1%'
      )
      and ps_availqty > (
        select
          0.5 * sum(l_quantity)
        from
          lineitem
        where
          l_partkey = ps_partkey
          and l_suppkey = ps_suppkey
          and l_shipdate >= date ':2'
          and l_shipdate < date ':2' + interval '1' year
    )
  )
  and s_nationkey = n_nationkey
  and n_name = ':3'
order by
  s_name;

Q16:
select
  s_name,
  count(*) as numwait
from
  supplier,
  lineitem l1,
  orders,
  nation
where
  s_suppkey = l1.l_suppkey
  and o_orderkey = l1.l_orderkey
  and o_orderstatus = 'F'
```

```
  and l1.l_receiptdate > l1.l_commitdate
  and exists (
    select
      *
    from
      lineitem l2
    where
      l2.l_orderkey = l1.l_orderkey
      and l2.l_suppkey <> l1.l_suppkey
  )
  and not exists (
    select
      *
    from
      lineitem l3
    where
      l3.l_orderkey = l1.l_orderkey
      and l3.l_suppkey <> l1.l_suppkey
      and l3.l_receiptdate > l3.l_commitdate
  )
  and s_nationkey = n_nationkey
  and n_name = ':1'
group by
  s_name
order by
  numwait desc,
  s_name;

Q17:
select
  cntrycode,
  count(*) as numcust,
  sum(c_acctbal) as totacctbal
from
  (
    select
      substring(c_phone from 1 for 2) as cntrycode,
      c_acctbal
    from
      customer
```

```
    where
      substring(c_phone from 1 for 2) in
      (':1', ':2', ':3', ':4', ':5', ':6', ':7')
      and c_acctbal > (
        select
          avg(c_acctbal)
        from
          customer
        where
          c_acctbal > 0.00
          and substring(c_phone from 1 for 2) in
              (':1', ':2', ':3', ':4', ':5', ':6', ':7')
      )
      and not exists (
        select
          *
        from
          orders
        where
          o_custkey = c_custkey
      )
  ) as custsale
group by
  cntrycode
order by
  cntrycode;
```

Appendix B

Materialized Views

This chapter presents the views materialized for the naive approach. The purpose is to give the exact syntax of view materialization in the used system. The views materialized in the other cases (see Section 6.1) have also been created with the same options as follows.

```
create materialized view log on lineitem
with rowid (l_orderkey, l_returnflag, l_linestatus,l_quantity, l_extendedp
l_discount, l_shipdate, l_tax) including new values;

create materialized view log on orders with rowid;

create materialized view NMVDV1
parallel
build immediate
refresh fast on commit
enable query rewrite
as
select
  l_orderkey,
  sum(l_quantity),
  count(*) as count_order,
  count(l_quantity) as count_1
from
  lineitem
group by
  l_orderkey;
```

```
create materialized view NMVD2
parallel
build immediate
refresh fast on commit
enable query rewrite
as
select
  part.rowid "p_rid",
  orders.rowid "o_rid",
  lineitem.rowid "l_rid",
  supplier.rowid "s_rid",
  partsupp.rowid "ps_rid",
  o_orderdate,
  l_extendedprice,
  l_discount,
  ps_supplycost,
  l_quantity,
  p_name,
  s_nationkey
from
  part,
  supplier,
  lineitem,
  partsupp,
  orders
where
  s_suppkey = l_suppkey
  and ps_suppkey = l_suppkey
  and ps_partkey = l_partkey
  and p_partkey = l_partkey
  and o_orderkey = l_orderkey;

create materialized view NMVD3
parallel
build immediate
refresh fast on commit
enable query rewrite
as
select
```

```
  customer.rowid "c_rid",
  orders.rowid "o_rid",
  lineitem.rowid "l_rid",
  supplier.rowid "s_rid",
  nation.rowid "n_rid",
  region.rowid "r_rid",
  n_name,
  l_extendedprice,
  l_discount,
  l_partkey,
  r_name,
  o_orderdate,
  s_nationkey,
  c_nationkey,
  n_nationkey
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;

create materialized view log on NMVD3
with rowid ( n_name, r_name, o_orderdate, l_extendedprice, l_discount)
including new values;

create materialized view NMVDV3_1
parallel
build immediate
refresh fast on commit
enable query rewrite
as
```

```
select
  n_name,
  r_name,
  o_orderdate,
  sum(l_extendedprice * (1 - l_discount)) as revenue,
  count(*) as count_order,
  count(l_extendedprice * (1 - l_discount)) as count_1
from
  NMVD3
group by
  n_name,
  r_name,
  o_orderdate;

create materialized view NMVD4
parallel
build immediate
refresh fast on commit
enable query rewrite
as
select
  l_shipdate,
  l_quantity,
  l_discount,
  sum(l_extendedprice * l_discount) as revenue,
  count(*) as count_order,
  count(l_extendedprice * l_discount) as count_1
from
  lineitem
group by
  l_shipdate,
  l_quantity,
  l_discount;

create materialized view NMVD5
parallel
build immediate
refresh fast on commit
enable query rewrite
as
```

```
select
  l_returnflag,
  l_linestatus,
  l_shipdate,
  sum(l_quantity) as sum_qty,
  sum(l_extendedprice) as sum_base_price,
  sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
  sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
  avg(l_quantity) as avg_qty,
  avg(l_extendedprice) as avg_price,
  avg(l_discount) as avg_disc,
  count(*) as count_order,
  count(l_quantity) as count_1,
  count(l_extendedprice) as count_2,
  count(l_extendedprice * (1 - l_discount)) as count_3,
  count(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as count_4,
  count(l_discount) as count_5
from
  lineitem
group by
  l_returnflag,
  l_linestatus,
  l_shipdate;
```

Bibliography

[CD97] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. In *ACM SIGMOD Record 26(1)*, March !997.

[CW00] Y. Cui and J. Widom. Practical lineage tracing in data warehouses. In *Proc. 16th Intl. Conf. on Data Eng.*, Feb. 2000.

[EBT97] S. Paraboschi E. Baralis and E. Teniente. Materialized view selection in a multidimensional database. In *Proc. VLDB*, pages 156–165, 97.

[GM99] H. Gupta and I. S. Mumick. Selection of views to materialize under a maintenance cost constraint. In *Intl. Conf. on Database Theory*, January 1999.

[Gup97] H. Gupta. Selection of views to materialize in a data warehouse. In *Proc. Intl. Conf. on Database Theory*, pages 136–145, January 1997.

[JYL97] K. Karlapalem J. Yang and Q. Li. Algorithms for materialized view design in data warehousing environment. In *Proc. VLDB*, pages 136–145, 1997.

[KR99] Y. Kotidis and N. Roussopoulos. Dynamat: A dynamic view management system for data warehouses. In *Proc. ACM SIGMOD*, 99.

[Lib] Oracle Documentation Library. *Oracle 8i Server and Datawarehousing*.

[MBM99] F. Fabret M. Bouzeghoub and M. Matulovic. Modeling data warehouse refreshment process as a workflow application. In *Proc. Intl. Workshop on Design and Management of Datawarehouses*, June 1999.

[MJV98] Y. Vassiliou M. Jarke, M. Lenzerini and P. Vassiliadis. *Fundamentals of Data Warehouses*. Springer, 1998.

[PVQ99] M. Bouzeghoub P. Vassiliadis and C. Quix. Towards quality-oriented data warehouse usage and evolution. In *Proc. Intl. Conf. on Advanced Information Systems Eng.*, 1999.

[Tra] Transaction Processing Performance Council. *TPC Benchmark R*, 1.2.0 edition.

[TS97] D. Theodoratos and T. Sellis. Data warehouse configuration. In *Proc. VLDB*, pages 126–135, 1997.

[VHU96] A. Rajaraman V. Harinarayan and J. D. Ullman. Implementing data cubes efficiently. In *Proc. ACM SIGMOD*, pages 205–216, 96.

[Wid95] J. Widom. Research problems in data warehousing. In *Proc. CIKM*, pages 25–30, Nov. 1995.

[YW00] J. Yang and J. Widom. Making temporal views self-maintainable for data warehousing. In *Proc. 7th Intl. Conf. on Extending Database Tech.*, March 2000.