Performance Evaluation of Incremental Materialized View Maintenance in ORDBMS

by

A.N.M. Bazlur Rashid

MASTER OF SCIENCE IN INFORMATION AND COMMUNICATION TECHNOLOGY

Institute of Information and Communication Technology
BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY
May 2010

The thesis titled “**Performance Evaluation of Incremental Materialized View Maintenance in ORDBMS**” submitted by A.N.M. Bazlur Rashid, Roll No: M10053122P, Session: October 2005 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of Master of Science in Information and Communication Technology on 8th May, 2010.

BOARD OF EXAMINERS

1. ____

   Dr. Md. Saiful Islam — Chairman
   Associate Professor — (Supervisor)
   Institute of Information and Communication Technology
   Bangladesh University of Engineering and Technology, Dhaka – 1000

2. ____

   Dr. S. M. Lutful Kabir — Member
   Professor and Director — (Ex-officio)
   Institute of Information and Communication Technology
   Bangladesh University of Engineering and Technology, Dhaka – 1000

3. ____

   Mr. Mohammad Ashraful Anam — Member
   Assistant Professor
   Institute of Information and Communication Technology
   Bangladesh University of Engineering and Technology, Dhaka – 1000

4. ____

   Dr. Hafiz Md. Hasan Babu — Member
   Professor — (External)
   Department of Computer Science and Engineering
   University of Dhaka, Dhaka – 1000

CANDIDATE'S DECLARATION

It is hereby declared that this thesis or any part of it has not been submitted elsewhere for the award of any degree or diploma.

____

A.N.M. Bazlur Rashid

DEDICATED TO MY PARENTS, BROTHER AND FAMILY MEMBERS

TABLE OF CONTENTS

Board of Examiners ii
Candidate's Declaration iii
Dedication iv
Table of Contents v
List of Figures viii
List of Tables xi
List of Abbreviations xii
Acknowledgements xiii
Abstract xiv

Chapter 1 Introduction 1-11

1.1 Overview of Database Management System 2
1.2 Object Relational Database Management System 2
1.3 Database Query, View and Materialized View 4
1.3.1 Database query 4
1.3.2 Database view 4
1.3.3 Materialized view 6
1.4 Materialized View Maintenance 6
1.5 Literature Review 7
1.6 Objective and Aims of the Research 10
1.7 Organization of the Thesis 11

Chapter 2 Materialized Views 12-43

2.1 What is Materialized View? 13
2.2 The Need for Materialized View 14
2.3 Summary Management 15
2.4 Materialized View Management Tasks 16
2.5 Materialized View Creation 17
2.6 Types of Materialized View 19
2.6.1 Read-only, updatable and writable materialized views 19
2.6.2 Primary key, object, ROWID and complex materialized views 22

2.6.3 Materialized views with aggregates, containing only joins and nested materialized views 27
2.7 Materialized View Maintenance 34
2.7.1 Incremental materialized view maintenance 36
2.7.2 Materialized view selection 39
2.8 Query Rewrite 40
2.8.1 How oracle rewrites queries? 42
2.8.2 General query rewrite method 42
2.8.3 Types of query rewrite 43

Chapter 3 View Materialization 44-64
3.1 Introduction 45
3.2 Brief Description of the Factors 46
3.3 Performance Evaluation Measurement 48
3.4 Methodology to Determine View Materialization 49
3.5 Dynamic Cost Model for Selection of Views for Materialization and Removal of Old Materialized Views 51
3.5.1 Dynamic selection of views for materialization 52
3.5.2 Dynamic removal of old materialized views 62

Chapter 4 Results and Discussions 65-110
4.1 Experimental Background 66
4.2 Experiments Results on View Materialization Determination Methodology. 68
4.2.1 Varying view selectivity 69
4.2.2 Varying view structural complexity 78
4.2.3 Varying database size 87
4.2.4 Comparison with related work 96
4.3 Experiments Results on Dynamic Selection of Views and Removal of Materialized views 97
4.3.1 Dynamic selection of views 97
4.3.2 Dynamic removal of materialized views 106
4.3.3 Comparison with related work 107

Chapter 5 Conclusion and Future Research **111-114**
5.1 Conclusion 112
5.2 Recommendation for Future Work 114

References 115

Appendix A Sales History Schema Tables with Columns Definitions 121

Appendix B Queries for Incremental Maintenance Performance Evaluation 129

Appendix C Queries for Dynamic Selection of Views 136

LIST OF FIGURES

Fig. No.	Figure Caption	Page No.
Fig. 1.1	Example of object-relational database management system	3
Fig. 2.1	Transparent query rewrite	14
Fig. 2.2	Overview of summary management	16
Fig. 2.3	Read-only materialized view in replication environment	20
Fig. 2.4	Updatable materialized view in replication environment	21
Fig. 2.5	Comparison of simple and complex materialized views	26
Fig. 2.6	A simple ROLLUP aggregation	28
Fig. 2.7	Logical CUBEs and views by different users	28
Fig. 2.8	Action of PIVOT and UNPIVOT operations	29
Fig. 2.9	Two tables join operation workflow	32
Fig. 2.10	A view materialization process	40
Fig. 2.11	Oracle SQL query rewrite mechanism	41
Fig. 3.1	Methodology to determine view materialization	49
Fig. 3.2	Sample representation of query execution frequencies	54
Fig. 3.3	Algorithm for finding access frequencies of the queries	55
Fig. 3.4	Algorithm for finding query complexities namely no. tables, joining and aggregations	56
Fig. 3.5	Algorithm for finding update frequencies of the base tables	58
Fig. 3.6	Algorithm for dynamic view selection to materialize	61
Fig. 3.7	Algorithm for dynamic removal of old materialized views	63
Fig. 4.1	Relationship of the sales history schema tables	67
Fig. 4.2	Template used to derive view selectivities	69
Fig. 4.3 (a)	View maintenance costs for joins only query	70
Fig. 4.3 (b)	View maintenance costs for aggregate query	70
Fig. 4.4 (a)	Query answering costs of view for joins only query	71
Fig. 4.4 (b)	Query answering costs of view for aggregate query	72
Fig. 4.4 (c)	Query answering costs using rewrite for joins only query	72
Fig. 4.4 (d)	Query answering costs using rewrite for aggregate query	73
Fig. 4.5 (a)	Relative cost of answering a view to the incremental propagation time for joins only query	74

Fig. 4.5 (b) Relative cost of answering a query using rewrite to the incremental propagation time for joins only query 75
Fig. 4.5 (c) Relative cost of answering a view to that of using rewrite for joins only query 75
Fig. 4.5 (d) Relative cost of answering a view to the incremental propagation time for aggregate query 76
Fig. 4.5 (e) Relative cost of answering a query using rewrite to the incremental propagation time for aggregate query 77
Fig. 4.5 (f) Relative cost of answering a view to that of using rewrite for aggregate query 77
Fig. 4.6 (a) View maintenance costs for joins only query 79
Fig. 4.6 (b) View maintenance costs for aggregate query 79
Fig. 4.7 (a) Query answering costs of view for joins only query 80
Fig. 4.7 (b) Query answering costs of view for aggregate query 81
Fig. 4.7 (c) Query answering costs using rewrite for joins only query 81
Fig. 4.7 (d) Query answering costs using rewrite for aggregate query 82
Fig. 4.8 (a) Relative cost of answering a view to the incremental propagation time for joins only query 83
Fig. 4.8 (b) Relative cost of answering a query using rewrite to the incremental propagation time for joins only query 84
Fig. 4.8 (c) Relative cost of answering a view to that of using rewrite for joins only query 84
Fig. 4.8 (d) Relative cost of answering a view to the incremental propagation time for aggregate query 85
Fig. 4.8 (e) Relative cost of answering a query using rewrite to the incremental propagation time for aggregate query 86
Fig. 4.8 (f) Relative cost of answering a view to that of using rewrite for aggregate query 86
Fig. 4.9 (a) View maintenance costs for joins only query 88
Fig. 4.9 (b) View maintenance costs for aggregate query 88
Fig. 4.10 (a) Query answering costs of view for joins only query 89
Fig. 4.10 (b) Query answering costs of view for aggregate query 90
Fig. 4.10 (c) Query answering costs using rewrite for joins only query 90

Fig. 4.10 (d) Query answering costs using rewrite for aggregate query 91
Fig. 4.11 (a) Relative cost of answering a view to the incremental propagation time for joins only query 92
Fig. 4.11 (b) Relative cost of answering a query using rewrite to the incremental propagation time for joins only query 93
Fig. 4.11 (c) Relative cost of answering a view to that of using rewrite for joins only query 93
Fig. 4.11 (d) Relative cost of answering a view to the incremental propagation time for aggregate query 94
Fig. 4.11 (e) Relative cost of answering a query using rewrite to the incremental propagation time for aggregate query 94
Fig. 4.11 (f) Relative cost of answering a view to that of using rewrite for aggregate query 95
Fig. 4.12 Relative cost of answering a view to the incremental propagation . . 96
Fig. 4.13 (a) Dynamically selected queries for materialization (Column Chart) . . 100
Fig. 4.13 (b) Dynamically selected queries for materialization (Line Chart) 101
Fig. 4.14 Query answering cost comparison for experiment 01 102
Fig. 4.15 Dynamically selected queries for materialization 104
Fig. 4.16 Query answering cost comparison for experiment 02 105
Fig. 4.17 Dynamically selected queries for materialization 110

LIST OF TABLES

Table No.	Table Caption	Page No.
Table 2.1	Materialized view refresh methods in Oracle	18
Table 2.2	Materialized view refresh modes in Oracle	18
Table 3.1	Database size example	47
Table 3.2	Different costs and factors associated with queries or views	53
Table 3.3	Access frequency count total for Fig. 3.2	54
Table 3.4	Sample table-update frequency status	58
Table 3.5	Table maintenance costs	59
Table 3.6	Dynamically selected views for materialization	62
Table 3.7	Materialized views-access frequencies matrix.	62
Table 3.8	Dynamically selected materialized views to remove	64
Table 4.1	List of database parameters and assigned values	68
Table 4.2	Database used in the experiments	68
Table 4.3	Maintenance cost calculation of the tables	97
Table 4.4	Query associated cost calculation	98
Table 4.5	Candidates query for the view materialization	100
Table 4.6	Maintenance cost calculation of the tables	102
Table 4.7	Query associated cost calculation	103
Table 4.8	Candidates query for the view materialization	104
Table 4.9	Access frequencies of existing materialized views	106
Table 4.10	Dynamically selected materialized views for removal	106
Table 4.11	Comparison of factors for selection of views with [35]	107
Table 4.12	Initial selection of views for further process	109
Table 4.13	Selected views for materialization	110
Table A.1	SALES HISTORY SCHEMA TABLES	121
Table A.2	ORDER ENTRY SCHEMA TABLES	125
Table B.1	List of queries used for selectivity experiments	129
Table B.2	List of queries used for complexity experiments	132
Table B.3	List of queries used for database size experiments	135
Table C.1	List of the queries used for dynamic selection of views in experiment no. 01	136
Table C.2	List of the queries used for dynamic selection of views in experiment no. 02	147

LIST OF ABBREVIATIONS

ADC	Australian Database Conference
BPUS	Benefit Per Unit Space
BUET	Bangladesh University of Engineering and Technology
DBMS	Database Management System
DDL	Data Definition Language
DML	Data Manipulation Language
DSS	Decision Support System
ERD	Entity Relationship Diagram
IACC	International Advance Computing Conference
ICDE	International Conference on Data Engineering
ICDT	International Conference on Database Theory
ICYCS	International Conference for Young Computer Scientist
IICT	Institute of Information and Communication Technology
IJCSNS	International Journal of Computer Science and Network Security
MVPP	Multiple View Processing Plan
ODBMS	Object Database Management System
ODMG	Object Data Management Group
OID	Object Identifier
OLAP	Online Analytical Processing
OODBMS	Object-Oriented Database Management System
OOP	Object-Oriented Programming
OQL	Object Query Language
ORD	Object-Relational Database
ORM	Object-Relational Mapping
PBS	Pick by Size
RDBMS	Relational-Database Management System
SQL	Structured Query Language

ACKNOWLEDGEMENTS

First of all, I sincerely express my gratitude to Almighty Allah for the successful completion of the thesis.

I would like to express my cordial gratitude and deep respect to my supervisor, Dr. Md. Saiful Islam for his constant supervision, unfailing encouragement, valuable suggestions and kind helps in many ways throughout this research work.

I want to thank Dr. Abu Sayed Md. Latiful Hoque, Associate Professor, Department of Computer Science and Engineering of BUET for his valuable guidance to emphasis the thesis to a perfect direction. I also gratefully acknowledge the valuable guidance from the Director of IICT and all of my honorable teachers of IICT, BUET.

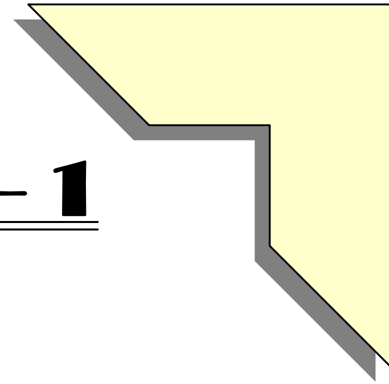
I would like to express my deepest gratitude to my beloved parents, brother and all other family members for their constant love, peaceful cooperation and encouragement.

ABSTRACT

A materialized view is a derived relation stored in the database, resembles like tables and behaves like indexes. Because of the query intensive nature of data warehousing or online analytical processing applications, materialized view is quite promising in efficiently processing queries to improve query performance. When a base relation is updated, all its dependent materialized views have to be updated in order to maintain the consistency and integrity of the database in response to the changes in the base relation. It is costly to rematerialize the view each time a change is made to the base tables that might affect it and it is desirable to propagate the changes incrementally. Hence, all of the views cannot be materialized due to the maintenance cost. So, it is necessary to evaluate the performance of incremental materialized view maintenance and to determine the circumstances in which a view is beneficial to be materialized for faster query performance. It is also necessary to dynamically select a subset of views from a set of views queried at a particular time period based on the query processing cost and view maintenance cost.

A methodology has been developed based on the performance affecting factors like - view selectivity, complexity and database size to evaluate the performance of incremental view maintenance and to determine the situations a view is profitable for materialization by computing the incremental propagation cost, query answering cost and relative costs of query answering versus propagating a materialized view. After this a dynamic cost model has been designed incorporating the above mentioned factors as well as query access frequency, execution time, table update frequency and view maintenance cost to select a subset of views from a set of views for materialization and to replace the old materialized views that are no longer in use or the materialized view access frequency is too low. A number of algorithms have been designed and mathematical equations have been developed to define the dynamic threshold level.

At the end, experimental results have been carried out for the incremental maintenance performance evaluation and on dynamic view selection and removal by using synthetic and real data sets with different characteristics in object-relational database. The outcome of the thesis reveals that the incremental maintenance is always cost effective. Finally, dynamic view selection for materialization and removal of old materialized views is explored based on dynamic threshold level.



CHAPTER - 1

INTRODUCTION

1.1 Overview of Database Management System

1.2 Object Relational Database Management System

1.3 Database Query, View and Materialized View

1.3.1 Database query

1.3.2 Database view

1.3.3 Materialized view

1.4 Materialized View Maintenance

1.5 Literature Review

1.6 Objective and Aims of the Research

1.7 Organization of the Thesis

C h a p t e r 1

INTRODUCTION

This chapter introduces the overview of database management system, database query, view, materialized view and materialized view maintenance. The chapter illustrates the review of the previous related research works, objectives and aims of the thesis and organization of the thesis.

1.1 Overview of Database Management System

A Database Management System (DBMS) is a set of computer programs that controls the creation, maintenance and the use of the database in a computer platform or of an organization and its end users. A DBMS is a system software package that helps the use of integrated collection of data records and files known as databases. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both *convenient* and *efficient*. It allows different user application programs to easily access the same database. DBMSs may use any of a variety of database models, such as the network model, relational model, object model or object-relational model. The relational data model by Codd [1] is the basis for Relational Database Management System (RDBMS). In large systems, a DBMS allows users and other software to store and retrieve data in a structured way. Instead of having to write computer programs to extract information, user can ask simple questions in a query language. It helps to specify the logical organization for a database and access and use the information within a database. It provides facilities for controlling data access, enforcing data integrity, managing concurrency controlled, and restoring database.

1.2 Object Relational Database Management System

A relational database management system (RDBMS) is a DBMS that is based on the relational model and where all data is stored and accessed via relations. A relation is usually described as a table organized into rows and columns. An object database management system (ODBMS) or object-oriented database management system (OODBMS) is a database model in which information is represented in the form of objects as used in object-oriented programming (OOP).

Object-relational database management systems (ORDBMS) grew out of research that occurred in the early 1990s. That research extended existing relational database concepts by adding object concepts. An object-relational database (ORD) or ORDBMS is a DBMS similar to a relational database, but with an object-oriented database model: objects, classes and inheritance are directly supported in database schemas and in the query language. In addition, it supports extension of the data model with custom data-types and methods. Fig. 1.1 shows an example of object-relational database management system where the entities are amends, users, shifts, departments, stores and jobs. Each entity has several attributes. The department entity has two attributes - id and name. Every attribute is defined with a type like - id is integer while name is character type field. Each of the entities is related with each other by their primary-foreign key relationship. The primary keys, foreign keys and uniquely keys are identified in the entities by P, F and U.

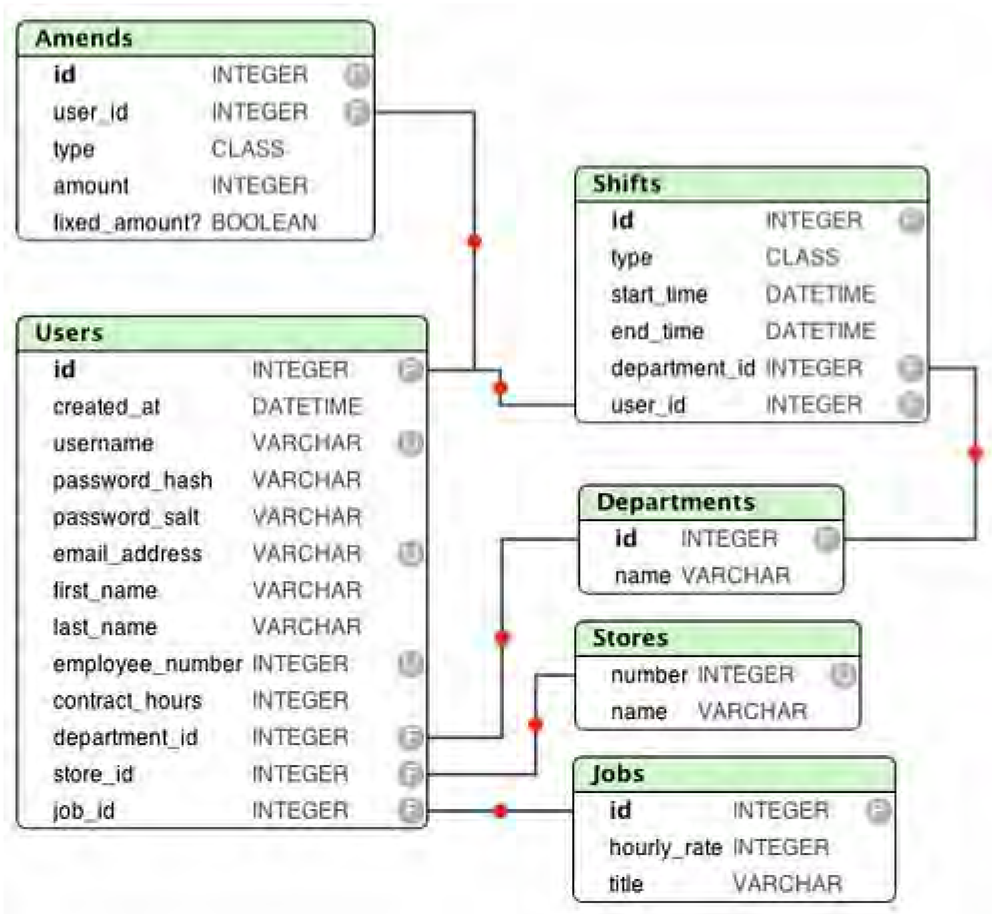


Fig. 1.1 Example of object-relational database management system

An object-relational database can be said to provide a middle ground between relational databases and object-oriented databases (OODBMS). In object-relational databases, the approach is essentially that of relational databases: the data resides in the database and is manipulated collectively with queries in a query language; at the other extreme are OODBMSs in which the database is essentially a persistent object store for software written in an object-oriented programming language, with a programming API for storing and retrieving objects, and little or no specific support for querying [2].

One aim for the Object-relational database is to bridge the gap between conceptual data modeling techniques such as Entity-relationship diagram (ERD) and object-relational mapping (ORM), which often use classes and inheritance and relational databases, which do not directly support them. Another, related aim is to bridge the gap between relational databases and the object-oriented modeling techniques used in programming languages such as Java, C++, Visual Basic .NET or C#.

1.3 Database Query, View and Materialized View

1.3.1 Database query

A database query is basically a question that is asked and answered from the database. The result of the query is the information that is returned by the database management system. Queries are usually constructed using structured query language (SQL) which resembles a high-level programming language. Object query language (OQL) is used to retrieve objects from object databases. The traditional SELECT-PROJECT-JOIN operators are the basis of an SQL query. The following syntax is an SQL select query to retrieve data from the database.

```
select <column_name>
from <table>
where <condition>;
```

1.3.2 Database view

In database theory, a view consists of a stored query accessible as a virtual table composed of the result set of a query. Unlike ordinary tables (base tables) in a relational database, a view does not form part of the physical schema: it is a dynamic, virtual table

computed or collated from data in the database. Changing the data in a table alters the data shown in subsequent invocations of the view. Views can provide advantages over tables:

- Views can represent a subset of the data contained in a table;
- Views can join and simplify multiple tables into a single virtual table;
- Views can act as aggregated tables, where the database engine aggregates data (sum, average etc) and presents the calculated results as part of the data;
- Views can hide the complexity of data; for example a view could appear as *Sales2000* or *Sales2001*, transparently partitioning the actual underlying table;
- Views take very little space to store; the database contains only the definition of a view, not a copy of all the data it presents;
- Depending on the SQL engine used, views can provide extra security;
- Views can limit the degree of exposure of a table or tables to the outer world.

Just as functions (in programming) can provide abstraction, so database users can create abstraction by using views. In another parallel with functions, database users can manipulate nested views, thus one view can aggregate data from other views. Without the use of views the normalization of databases above second normal form would become much more difficult. Views can make it easier to create lossless join decomposition. Just as rows in a base table lack any defined ordering, rows available through a view do not appear with any default sorting. A view is a relational table and the relational model defines a table as a set of rows. The following is an example of a database view where the query selects customer name, money received and sent and balance:

```
create or replace view account_view as
select name, money_received, money_sent,
(money_received - money_sent) as balance, address
from table_customers c join accounts_table a
on a.customerid = c.customer_id;
```

1.3.3 Materialized view

A materialized view takes a different approach in which the query result is cached as a concrete table that may be updated from the original base tables from time to time. This enables much more efficient access, at the cost of some data being potentially out-of-date. It is most useful in data warehousing scenarios, where frequent queries of the actual base tables can be extremely expensive. In addition, because the view is manifested as a real table, anything that can be done to a real table can be done to it, most importantly building indexes on any column, enabling drastic speedups in query time. In a normal view, it's typically only possible to exploit indexes on columns that come directly from (or have a mapping to) indexed columns in the base tables; often this functionality is not offered at all. Materialized views were implemented first by the Oracle database [3]. The following is an example of creating a materialized view on SALES schema:

```
create materialized view sales_mv
build immediate
refresh fast on commit
as
select t.calendar_year, p.prod_id, sum (s.amount_sold) as sum_sales
from times t, products p, sales s
where t.time_id = s.time_id and p.prod_id = s.prod_id
group by t.calendar_year, p.prod_id;
```

1.4 Materialized View Maintenance

Just as a cache gets dirty when the data from which it is copied is updated, a materialized view gets dirty whenever the underlying base relations are modified. The process of updating a materialized view in response to changes to the underlying data is called view maintenance.

In most cases, it is wasteful to maintain a view by re-computing it from scratch. Often it is cheaper to use the heuristic of inertia (only a part of the view changes in response to changes in the base relations) and thus compute only the changes in the view to update its materialization. The above scenario is only a heuristic. For example, if an entire base

relation is deleted, it may be cheaper to recomputed a view that depends on the deleted relation (if the new view quickly evaluates to an empty relation) than to compute the changes to the view. Algorithms that compute changes to a view in response to changes to the base relations are called incremental view maintenance algorithms.

1.5 Literature Review

A materialized view is like a cache – a copy of data that can be accessed quickly. From a physical design point of view, materialized view resembles like tables or partitioned tables and behaves like indexes and it is used for improving query performance. Utilizing materialized views that incorporate not just traditional simple SELECT-PROJECT-JOIN operators but also complex online analytical processing (OLAP) operators (i.e., PIVOT and UNPIVOT) play crucial role to improve the OLAP query performance. Materialized views are useful in applications such as data warehousing, replication servers, data recording systems, data visualization and mobile systems [4-6].

In certain situations, it is more profitable to materialize a view than to compute the base relations every time the view is queried. Materializing a view causes it to be refreshed every time a change has been made to the base relations that it references. It can be costly to rematerialize the view each time a change is made to the base tables that might affect it. So it is desirable to propagate the changes incrementally *i.e.*, the materialized view should be refreshed for incremental changes to the base relations. In the last few years several view maintenance methods and algorithms have been designed and developed to obtain an efficient incremental view maintenance plan and view selection for materialization [7-11].

Since materialized views correspond to pre-computed and stored query results, they may become out-of-date when the underlying sources are changed. Hence, one important issue is to maintain the materialized view's consistency upon any source changes. While re-computing views from scratch in response to any source updates may be acceptable for some relatively static databases, it is unaffordable when the source changes are frequent. Hence, incremental view maintenance, as an efficient alternative, has been proposed and extensively studied [12].

Ali *et al.* (2003) reported and evaluated an algebraic incremental maintenance plan for each of the update events for incremental maintenance of materialized object query language (OQL) views in object data management group (ODMG) compliant object databases [13]. Lee *et al.* (2007) designed an optimal delta evaluation method to minimize the total accesses to relations for efficient incremental view maintenance [14]. An improved algorithm Glide* based on the incremental view maintenance algorithms is introduced by Chen *et al.* (2008) to eliminate the anomalies by using extra compensating queries [15].

A delta propagation strategy is introduced for multiple views that compute the change of a join view in a recursive manner to incrementally maintain the multiple join views efficiently [16]. Surendrababu *et al.* (2006) developed an algorithm to implement an incremental update to the schema-restructuring view that propagates the updates through the operators of the *SchemaSQL* algebra tree [17]. Hanson (1987) used a cost model to compare the performance of immediate and deferred view materialization algorithms with that of virtual views. The study reveals that the performance of materialized views and their virtual correspondents is sensitive to: selectivity of the view predicate, probability of updates, the selectivity of the query over the view and the number of tuples affected by each update [18]. Blakeley *et al.* (1990) compared the performance of materialized views against the use of join indexes and hybrid-hash joins in virtual views. Their study is based on a cost model and reveals the issues like selectivity, update activity, the probability of update to the joining attributes and the size of tables and memory [19]. Hull *et al.* (1996) evaluated the performance which reveals an impact of selectivity. They also addressed query/update issues and showed that network traffic for materialized views is proportional to the update rate [20].

The materializing of views is the most important task in data warehousing environment. It is impossible to materialize all possible views due to the large computation and huge space occupied by the materialized view. The selection of view materialization is affected by numerous factors. Thus the process of selecting the suitable views to materialize in ORDBMS or especially in data warehousing environment is a critical issue [21].

Harianarayan *et al.* [22] presented a greedy algorithm for the selection of materialized views so that query evaluation costs can be optimized in the special case of “data cubes” without addressing the cost of view maintenance and storage. Yang *et al.* [23] proposed a heuristic algorithm which utilizes a multiple view processing plan (MVPP) to obtain an optimal materialized view selection such that the best combination of good performance and low maintenance cost can be achieved. But the algorithm did not consider the system storage constraints. Gupta [24] developed a greedy algorithm to incorporate the maintenance cost and storage constraint in the selection of data warehouse materialized views. “AND-OR” view graphs introduced to represent all the possible ways to generate warehouse views such that the best query path can be utilized to optimize query response time.

Shukla *et al.* [25] proposed a simple and fast heuristic algorithm, Pick by Size (PBS), to select aggregates for pre-computation. PBS runs several orders of magnitude faster than Benefit Per Unit Space (BPUS) and is fast enough to make the exploration of the time-space tradeoff feasible during system configuration. Gupta and Mumick [26] developed algorithms to select a set of views to materialize in a data warehouse in order to minimize the total query response time under the constraint of a given total view maintenance time. Zhang *et al.* [27] proposed a completely different approach, Genetic Algorithm, to choose materialized views and it was effective compared with heuristic approaches. Agrawal *et al.* [28] presented an end-to-end solution to the problem of selecting materialized views and indexes.

Zhang *et al*. [29] explored the use of a hybrid evolutionary algorithm for materialized view selection based on multiple global processing plans for queries. An efficient solution has been proposed by Lee and Hammer [30] to the maintenance-cost view selection problem using a genetic algorithm for computing a near optimal set of views used to search for a near optimal solution. Kalnis *et al.* [31] proposed the application of randomized search heuristics, namely iterative improvement and simulated annealing which select fast a sub-optimal set of views.

Yu *et al.* [32] presented a new constrained evolutionary algorithm for the maintenance-cost view-selection problem where the constraints were incorporated through a stochastic

ranking procedure. Wang *et al.* [33] proposed a modified genetic algorithm for the selection of a set of views for materialization. Aouiche *et al.* [34] developed a framework for materialized view selection that exploits a data mining technique (clustering) in order to determine clusters of similar queries. They also proposed a view merging algorithm that builds a set of candidate views as well as a greedy process for selecting a set of views to materialize. An optimized framework has been designed by Ashadevi and Balasubramanian [35] for the selection of views to materialize for a given storage space constraints to achieve the best combination good query response, low query processing cost and low view maintenance cost. The proposed framework considered the query execution frequencies, query access costs, view maintenance costs and system's storage space constraints for materialized view selection.

From the above mentioned works, it is found that most of the research works have been focused on different methods and algorithms based on various data models and view languages to process the incremental materialized view maintenance efficiently. The research works also provided different approaches for the selection of views to materialize considering view maintenance cost and storage space. A systematic performance evaluation on incremental view maintenance for selecting a view to be materialized in different situations and the dynamic selection of views for materialization and removal of old materialized views based on dynamic threshold level is yet to be reported in ORDBMS. Thus, it is necessary to evaluate the performance of incremental maintenance of materialized views and to develop cost model for the dynamic selection of views to materialize and removal of old materialized views in ORDBMS.

1.6 Objective and Aims of the Research

The goal of this research is to evaluate the performance of incremental materialized view in ORDBMS to determine in what circumstances a view is to be materialized and which view is to be selected for materialization and which old materialized views are to be removed dynamically. To meet the goal, the following objectives have been pointed out:

- Developing a methodology to evaluate the performance of incremental maintenance;

- Developing a cost model to dynamically select views to materialize and remove old materialized views dynamically;
- Applying update events to the base tables and propagating the changes to the materialized views;
- Computing the cost of answering query, update propagation and the relative costs;
- Simulating and analyzing the performance results.

1.7 Organization of the Thesis

The thesis is organized in five different chapters. At a first glance, in **Chapter 1**, introduction of database, materialized view, literature review of related works and objectives of the thesis have been discussed.

Chapter 2 provides the detailed on materialized view, its importance, maintenance of materialized view and related tasks.

Chapter 3 includes the theoretical detailed of materialized view maintenance performance evaluation, dynamic selection of views for materialization and removal of old views.

Chapter 4 presents the experimental results and simulated output. The results are analyzed to determine the situations of view materialization and performance of dynamic selection of views for materialization and dynamic removal of old materialized views.

Finally, **Chapter 5** concludes the thesis and suggests recommendations for future research works.

CHAPTER - 2

MATERIALIZED VIEWS

2.1 What is Materialized View?

2.2 The Need for Materialized View

2.3 Summary Management

2.4 Materialized View Management Tasks

2.5 Materialized View Creation

2.6 Types of Materialized View

2.6.1 Read-only, updatable and writable materialized views

2.6.2 Primary key, object, ROWID and complex materialized views

2.6.3 Materialized views with aggregates, containing only joins and nested materialized views

2.7 Materialized View Maintenance

2.7.1 Incremental materialized view maintenance

2.7.2 Materialized view selection

2.8 Query Rewrite

2.8.1 How oracle rewrites queries?

2.8.2 General query rewrite method

2.8.3 Types of query rewrite

Chapter 2

MATERIALIZED VIEWS

This chapter describes materialized view, its importance, materialized view management tasks and materialized view creation, different types of materialized views. This chapter also illustrates materialized view maintenance and query rewrite.

2.1 What is Materialized View?

When a view is defined, normally the database stores only the query defining the view. In contrast, a materialized view is a view whose contents are computed and stored. Materialized views constitute redundant data, in that their contents can be inferred from the view definition and the rest of the database contents. However, it is much cheaper in many cases to read the contents of a materialized view than to compute the contents of the view by executing the query defining the view.

Materialized views are important for improving performance in some applications. We may consider the following view definition, which gives the total loan amount at each branch:

> create view *branch_total_loan (branch_name, total_loan)* as
> select *branch_name, sum (amount)*
> from *loan*
> group by *branch_name*;

Suppose the total loan amount at the branch is required frequently (*i.e.*, before making a new loan). Computing the view requires reading every *loan* tuple pertaining to the branch and summing up the loan amounts which can be time consuming. In contrast, if the view definition of the total loan amount is materialized, the total loan amount could be found by looking up a single tuple in the materialized view [36].

So the materialized views are query results that have been stored in advance so long-running calculations are not necessary when we actually execute our SQL statements. From a physical design point of view, materialized views resemble tables or partitioned tables and behave like indexes in that they are used transparently and improve performance.

2.2 The Need for Materialized Views

Materialized views are used to increase the speed of queries on very large databases. Queries to large databases often involve joins between tables, aggregations such as SUM or both. These operations are expensive in terms of time and processing power. The way the materialized view is created that determines how the materialized view is refreshed and used by query rewrite. Materialized views improve query performance by pre-calculating expensive joins and aggregation operations on the database prior to execution and storing the results in the database. The query optimizer automatically recognizes when an existing materialized view can and should be used to satisfy a request. It then transparently rewrites the request to use the materialized view. In Fig. 2.1, a general query rewrite process is given. Queries go directly to the materialized view and not to the underlying base tables. In general, rewriting queries to use materialized view rather than base tables and thus improves response time.

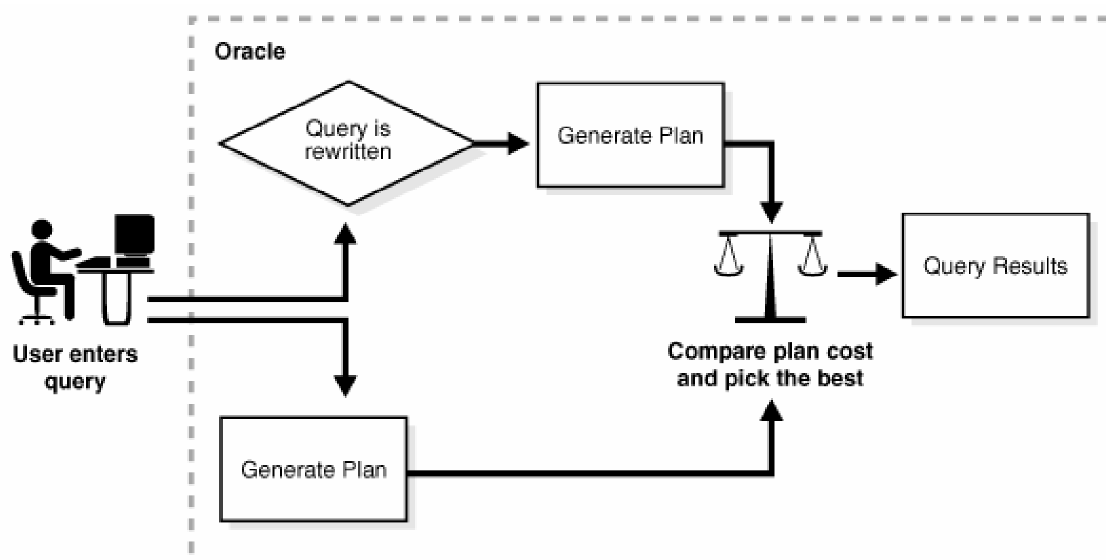


Fig. 2.1 Transparent query rewrite

For query rewrite, materialized views need to be created to satisfy the largest number of queries. For example, if 15 queries are commonly applied to the base tables, then with four or five well written materialized views can be able to satisfy them. If a materialized view is to be used by query rewrite, it must be stored in the same database as the base tables on which it is relies. Unlike indexes, materialized views can be accessed directly using a ***select*** statement. However it is recommended to avoid querying the materialized

view directly as it is difficult to change the SQL statement without affecting the application and it should use the query rewrite to use the materialized view.

Materialized views application are formed in different systems like - data warehousing, distributed computing and mobile computing etc. In data warehouses, materialized views can be used to pre-compute and store aggregated data such as the sum of sales. Materialized views in these environments are often referred to as summaries because they store summarized data. They can also be used to pre-compute joins with or without aggregations. A materialized view eliminates the overhead associated with expensive joins and aggregations for a large or important class of queries.

In distributed environments, materialized views can be used to replicate data at distributed sites and to synchronize updates at sites with conflict resolution methods. These replica materialized views provide local access to data that otherwise would have to be accessed from remote sites. Materialized views are useful in remote data marts. Materialized views can also be used to download a subset of data from central servers to mobile clients with periodic refreshes and updates between clients and the central servers [37].

Materialized views can be used to replicate data to non-master sites in a replication environment and to cache expensive queries in a data warehouse environment. In a replication environment, materialized views can be used to achieve the goals like - ease network loads, create a master deployment environment, enable data sub-setting and enable disconnected computing etc.

2.3 Summary Management

The use of summary management features imposes no schema restrictions and can enable some existing decision support system (DSS) database applications to improve performance without need to redesign the database or the application. Fig. 2.2 illustrates the use of summary management in the data warehousing cycle. After the data has been transformed, stages and loaded into the base data in the data warehouse, the summary management process can be invoked. The summary management process consists of:

- Mechanism to define materialized views and dimensions;
- A refresh mechanism to ensure that all materialized views contain the latest data;
- A query rewrite capability to transparently rewrite a query to use a materialized view.

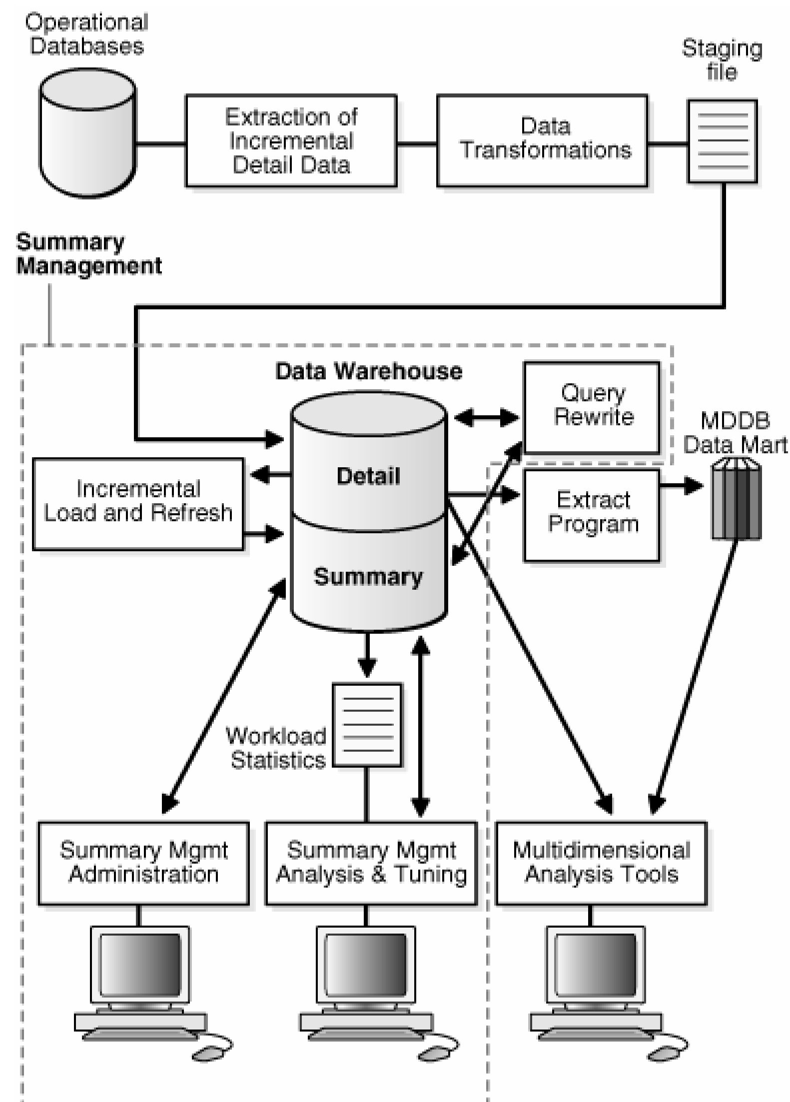


Fig. 2.2 Overview of summary management

2.4 Materialized View Management Tasks

The motivation for using materialized view is to improve query performance but the overhead associated with materialized view management can become a significant system management problem. The common materialized view management activities are:

- Identifying which materialized views to create;
- Indexing the materialized view;
- Ensuring that all materialized views and materialized view indexes are refreshed properly each time the database is updated;
- Verifying the incremental changes are correct, consistent and complete;
- Checking which materialized views have been used;
- Determining how effective each materialized view has been on workload performance;
- Measuring the space being used by the materialized views;
- Determining which new materialized views should be created;
- Determining which existing materialized views should be dropped;
- Archiving old detail and materialized view data that is no longer useful.

2.5 Materialized View Creation

The basic syntax for creating a materialized view in Oracle Database is like the following:

```
create materialized view <materialized_view_name>
tablespace <tablespace_name>
build <build_method>
refresh <refresh_method>
<refresh_mode>
<query rewrite enable/disable>
as
<select subquery>;
```

From the syntax, *<materialized_view_name>* specifies the materialized view name to be defined. *<tablespace_name>* is the tablespace in which the materialized view is to be created; if the tablespace name is unspecified then the default tablespace will be used to store the materialized view. There are two build methods for creating a materialized view in the *<build_method>* namely ***build immediate*** and ***build deferred***; build immediate method creates the materialized view and then populates it with data while build deferred method creates the materialized view but do not populate it with data. In case of build immediate, the materialized view definition is added to the schema according to the

SELECT expression and the results are stored in the materialized view; Depending on the size of the tables, this build process can take a considerable amount of time. For the build deferred method, after the materialized view is created, it should be refreshed completely for populating it with data.

The refresh option can be specified at the time of materialized view creation in <*refresh_method*>. The refresh mode can also be specified with the refresh method. There are four different kinds of refresh methods and two types of refresh modes in Oracle. The refresh methods are Complete, Fast, Force and Never. The refresh modes are On Commit and On Demand. Table 2.1 and 2.2 show the different refresh methods and modes available in Oracle database.

Table 2.1 Materialized view refresh methods in Oracle.

Refresh Method	Description
COMPLETE	Refreshes by recalculating the materialized view's defining query.
FAST	Applies incremental changes to refresh the materialized view using the information logged in the materialized view logs, or from a SQL*Loader direct-path or a partition maintenance operation.
FORCE	Applies FAST refresh if possible; otherwise, it applies COMPLETE refresh.
NEVER	Indicates that the materialized view will not be refreshed with refresh mechanisms.

Table 2.2 Materialized view refresh modes in Oracle.

Refresh Mode	Description
ON COMMIT	Refresh occurs automatically when a transaction that modified one of the materialized view's detail tables commits. This can be specified as long as the materialized view is fast refreshable (in other words, not complex). The ON COMMIT privilege is necessary to use this mode.
ON DEMAND	Refresh occurs when a user manually executes one of the available refresh procedures contained in the DBMS_MVIEW package (REFRESH, REFRESH_ALL_MVIEWS, and REFRESH_DEPENDENT).

2.6 Types of Materialized View

There are different types of materialized view: read-only, updatable and writable, primary key materialized views, object materialized views, ROWID materialized views, complex materialized views, materialized views with aggregates, materialized views containing only joins and nested materialized views [38].

2.6.1 Read-only, updatable and writable materialized views

A materialized view can be read-only, updatable or writable. Users can not perform data manipulation language (DML) statements on read-only materialized view but they can perform DML on updatable and writable materialized views.

Read-only materialized view: A materialized view can be made *read-only* during creation by omitting the FOR UPDATE clause. Read-only materialized views use many of the same mechanisms as updatable materialized views except they do not need to belong to a materialized view group. In addition, using read-only materialized view eliminates the possibility of a materialized view introducing data conflicts at the master site or master materialized view site, although this convenience means that updates can not be made at the remote materialized view site. The following is an example of creating a read-only materialized view:

create materialized view <*mv_name*> as
*select * from hr. employees*;

Fig. 2.3 shows an example of a read-only materialized view in a replication environment where from the client the materialized view can be locally queried but can not be updatable.

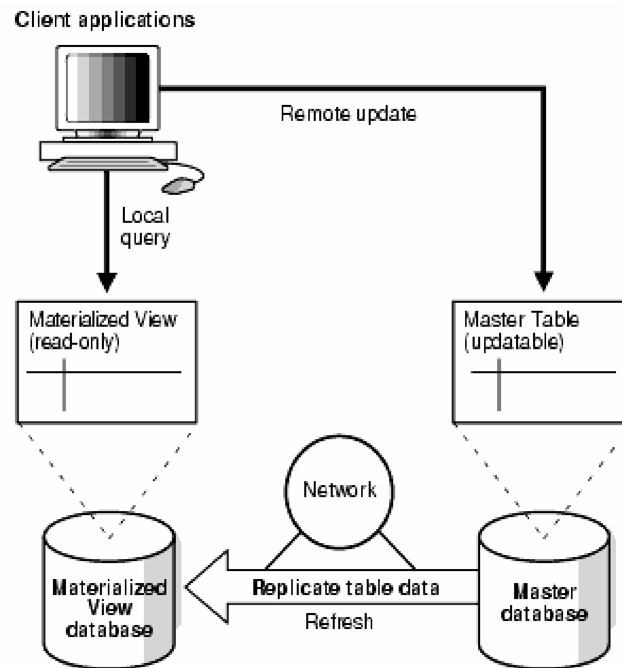


Fig. 2.3 Read-only materialized view in a replication environment

Updatable materialized view: A materialized view can be made *updatable* during creation by including the FOR UPDATE clause. For changes made to an updatable materialized view to be pushed back to the master during refresh, the updatable materialized view must belong to a materialized view group. Updatable materialized views enable to decrease the load on master sites because users can make changes to the data at the materialized view site. The following is an example of creating updatable materialized view:

```
create materialized view hr.departments for update as
select * from hr.departments@orcl.world;
```

The following statement creates a materialized view group:

```
begin
    dbms_repcat.create_mview_repgroup (
        gname => 'hr_repg',
        master => 'orcl.world',
        propagation_mode => 'ASYNCHRONOUS');
end;
/
```

The following statement adds the *hr.departments* materialized view to the materialized view group, making the materialized view updatable:

```
begin
    dbms_repcat.create_mview_repobject (
        gname => 'hr_repg',
        sname => 'hr',
        oname => 'departments',
        type => 'SNAPSHOT',
        min_communication => TRUE);
end;
/
```

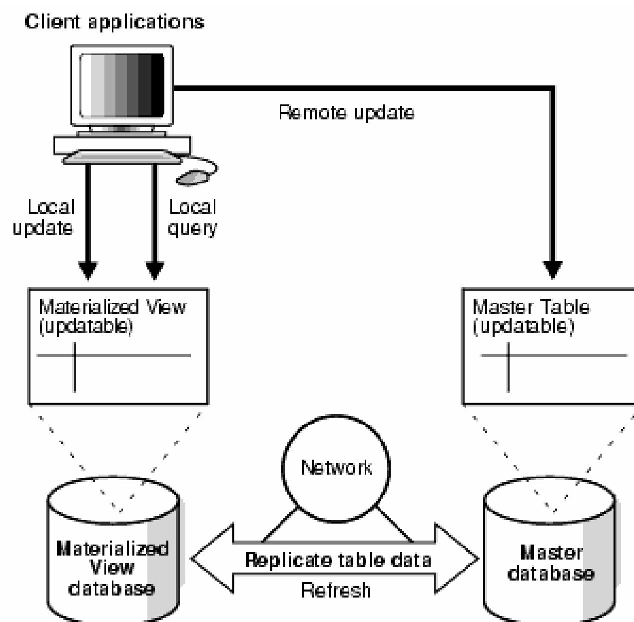


Fig. 2.4 Updatable materialized view in a replication environment

Fig. 2.4 shows an example of an updatable materialized view in a replication environment where from the client the materialized view can be locally queried as well as it can be locally updatable.

***Writable materialized view*:** A *writable* materialized view is one that is created using the FOR UPDATE clause but is not part of a materialized view group. Users can perform DML operations on a writable materialized view, but refreshing the materialized view,

these changes are not pushed back to the master and the changes are lost in the materialized view itself. Writable materialized views are typically allowed wherever fast-refreshable read-only materialized views are allowed.

2.6.2 Primary key, object, ROWID and complex materialized views

In Oracle database, it offers several types of materialized views to meet the needs of many different situations like data warehousing and replication. The examples of different types of materialized views include primary key materialized views, object materialized views, ROWID materialized views and complex materialized views. The materialized views with aggregates, containing only joins and nested materialized views are fall under the category of complex materialized views.

Primary key materialized views: Primary key materialized views are the default type of materialized view. These kind of materialized views are based on the primary key of the underlying table. They are updatable if the materialized view is created as part of a materialized view group and FOR UPDATE is specified when defining the materialized view. Changes are propagated according to the row-level changes that have occurred as identified by the primary key value of the row (not the ROWID). The following is an example of a SQL statement for creating a primary key materialized view:

> create materialized view *employees_mv* as
> select * from *emp_user.employee*;

The following is an example of a SQL statement for creating an updatable, primary key materialized view:

> create materialized view *oe.customers_mv* for update as
> select * from *oe.customers*;

Object materialized views: If a materialized view is based on an object table and is created using the OF *type* clause, then the materialized view is called an object materialized view. An object materialized view is structured in the same way as an object table. That is, an object materialized view is composed of row objects and each row object is identified by an object identified (OID) column. If a materialized view that is based on an object table is created without using the OF *type* clause, then the materialized view is

read-only and is not an object materialized view. That is such a materialized view has regular rows, not row objects. To create a materialized view based on an object table, the types on which the materialized view depends must exist at the materialized view site and each type must have the same object identifier as it does at the master site. The following SQL statements create the *oe.categories_tab* object table at the *orcl.world* master site:

```
create type oe.category_typ as object
(category_name VARCHAR2(50),
category_description VARCHAR2(1000),
category_id NUMBER(2));

create table oe.categories_tab OF oe.category_typ
(category_id PRIMARY KEY);
```

Now an object materialized view can be created based on the *oe.categories_tab* object table using the OF *type* clause as in the following SQL statement:

```
create materialized view oe.categories_objmv OF oe.category_typ
refresh fast for update as
select * from oe.categories_tab;
```

ROWID materialized views*:* A ROWID materialized view is based on the physical row identifiers (rowids) of the rows in a master site. ROWID materialized view can be used for materialized views based on master tables that do not have a primary key or for materialized views that do not include all primary key columns of the master tables. The following is an example of a SQL statement that creates a ROWID materialized view:

```
create materialized view oe.orders
refresh with ROWID as
select * from oe.orders;
```

Complex materialized view: Generally, a materialized view is considered complex when the defining query of the materialized view contains:

- A CONNECT BY clause;
- An INTERSECT, MINUS, or UNION ALL set operation;
- The DISTINCT or UNIQUE keyword;
- An aggregate function;
- Joins other than those in a subquery;
- A UNION operation;
- More than 1 table is involved.

The following examples create complex materialized view:

To select the employees those are manager with their level and email address, the following complex materialized view can be created which uses CONNECT BY clause:

```
create materialized view hr.emp_hierarchy as
select LPAD (' ' ,  4 * (level - 1)) || email USERNAME
from hr.employees start with manager_id is null
connect by prior employee_id = manager_id;
```

Find the old and new employee's IDs and email addresses and then combine all the employees' ID and email by using the UNION ALL set operation:

```
create materialized view hr.mview_employees as
select employees.employee_id, employees.email
from hr.employees
union all
select new_employees.employee_id, new_employees.email
from hr.new_employees;
```

Find the unique department ids from the employees table and sort the result in ascending order:

```
create materialized view hr.employee_depts as
select distinct department_id from hr.employees
order by department_id;
```

Find the average salary of the employees:

```
create materialized view hr.average_sal as
select AVG (salary) "Average" from hr.employees;
```

Find the name of the employees who work in a department:

```
create materialized view hr.emp_join_dep as
select last_name from hr.employees e, hr.departments d
where e.department_id = d.department_id;
```

Find the orders of the customers whose credit limit equals to 30 or greater than 50:

```
create materialized view oe.orders as
select order_total
from oe.orders o
where exists (select cust_first_name, cust_last_name
    from oe.customers c
    where o.customer_id = c.customer_id and c.credit_limit > 50)
union
select customer_id
from oe.orders o
where exists (select cust_first_name, cust_last_name
    from oe.customers c
    where o.customer_id = c.customer_id and c.account_mgr_id = 30);
```

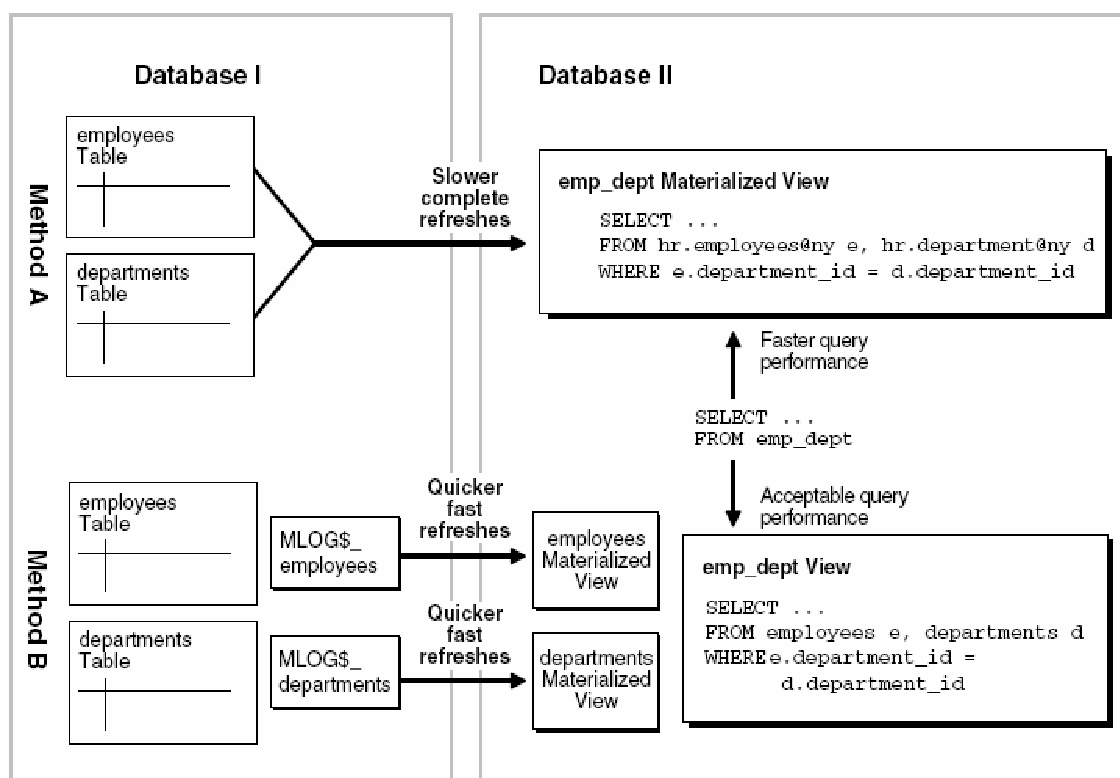


Fig. 2.5 Comparison of simple and complex materialized views

Fig. 2.5 shows the comparison between the simple and complex materialized views. The complex materialized view (Method A) in Database II exhibits efficient query performance because the join operation was completed during the materialized view's refresh. However, complete refreshes must be performed because the materialized view is complex and these refreshes will probably be slower than incremental refreshes. On the other hand, a virtual view performs the join operation between the simple materialized views (Method B) in Database II. Query performance against the virtual view would not be as good as the query performance against the complex materialized view in Method A. However, the simple materialized views can be refreshed more efficiently using incremental refresh. So, if the refresh occurs rarely and it needs faster query performance then complex materialized view is better than the simple materialized view where the refresh may occur regularly and query performance may be sacrificed.

2.6.3 Materialized views with aggregates, containing only joins and nested materialized views

The SELECT clause in the materialized view creation statement defines the data that the materialized view is to contain. Any number of tables can be joined together. Views, inline views (subqueries in the FROM clause of a SELECT statement), subqueries and materialized views can all be joined or referenced in the SELECT clause. The SELECT clause of the materialized view may retrieve data by aggregating; joining of tables from more than two tables or from remote locations and materialized views can itself be nested.

Materialized views with aggregates: Aggregation is a fundamental part of data warehousing. In data warehouses, materialized views normally contain aggregates. There are lots of aggregates and the functionally of each of the aggregates distinguishes from each other. The CUBE, ROLLUP and GROUPING SETS extensions to SQL make querying and reporting easier and faster. CUBE, ROLLUP and GROUPING SETS produce a single result set that is equivalent to a UNION ALL of differently grouped rows. ROLLUP calculates aggregations such as SUM, COUNT, MAX, MIN and AVG at increasing levels of aggregations from the most detailed up to a grand total. CUBE is an extension similar to ROLLUP, enabling a single statement to calculate all possible combination of aggregations. Computing a CUBE creates a heavy processing load, so replacing cubes with grouping sets can significantly increase performance. Fig. 2.6 shows a ROLLUP aggregation where the individual order total price is aggregated first and then the total price of that customer aggregated and finally the total price for all customers is aggregated by the ROLLUP operation.

	MyCustomerID	MyOrderID	price
1	(Total)	-1	469771.3400
2	ALFKI	10952	491.2000
3	ALFKI	10835	851.0000
4	ALFKI	11011	960.0000
5	ALFKI	-1	2302.2000
6	ANATR	10926	514.4000
7	ANATR	-1	514.4000
8	ANTON	10856	660.0000
9	ANTON	-1	660.0000
10	AROUT	10864	282.0000
11	AROUT	10920	390.0000
12	AROUT	11016	491.5000
13	AROUT	10953	4675.0000
14	AROUT	-1	5838.5000
15	BERGS	10875	729.5000

Fig. 2.6 A simple ROLLUP aggregation

Fig. 2.7 shows a logical data CUBE and how it can be used differently by various groups. The CUBE stores sales data organized by the dimensions of product, markets, sales and time.

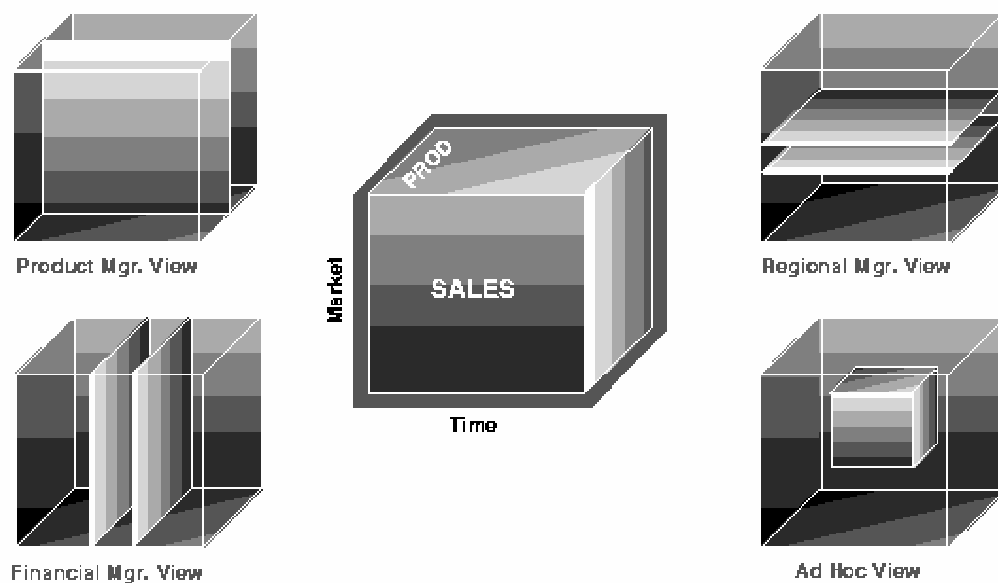
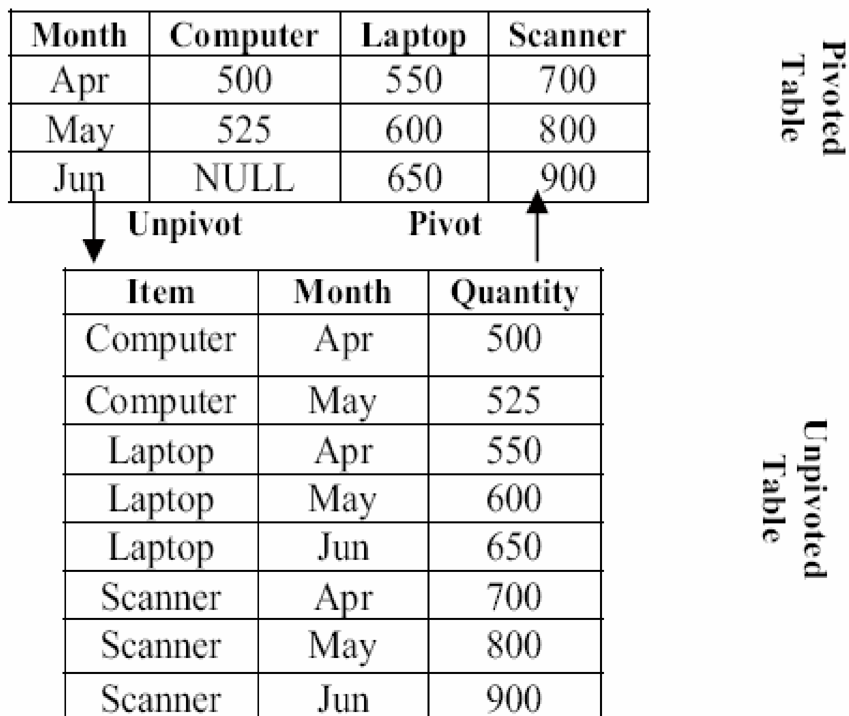


Fig. 2.7 Logical CUBEs and views by different users

PIVOT transforms a series of rows into a series of fewer rows with additional columns. Data in one source column is used to determine the new column for a row and another source column is used as the data for the new column. UNPIVOT provides the inverse operation, removing a number of columns and creating additional rows that capture the column names and values from the pivoted form. The pivoted form can be considered as a matrix of column of values while the unpivoted form is a natural encoding of a sparse matrix [39]. Fig. 2.8 shows the pivot and unpivot operations.



Month	Computer	Laptop	Scanner
Apr	500	550	700
May	525	600	800
Jun	NULL	650	900

Item	Month	Quantity
Computer	Apr	500
Computer	May	525
Laptop	Apr	550
Laptop	May	600
Laptop	Jun	650
Scanner	Apr	700
Scanner	May	800
Scanner	Jun	900

Fig. 2.8 Action of PIVOT and UNPIVOT operations

The followings are examples of materialized views with different aggregations:

Find the product wise total sales amount and quantity of the store:

create materialized view *product_sales_mv*
tablespace *demo*
build immediate
refresh fast
enable query rewrite as

```
select p.prod_name, sum(s.amount_sold) as dollar_sales,
count(*) as cnt, count(s.amount_sold) as cnt_amt
from sales s, products p
where s.prod_id = p.prod_id group by p.prod_name;
```

Find the total sales amount, number of sales, total quantity sold and number of quantity sold in the store:

```
create materialized view sum_sales
parallel
build immediate
refresh fast on commit
as
select s.prod_id, s.time_id, count(*) as count_grp,
sum(s.amount_sold) as sum_dollar_sales,
count(s.amount_sold) as count_dollar_sales,
sum(s.quantity_sold) as sum_quantity_sales,
count(s.quantity_sold) as count_quantity_sales
from sales s
group by s.prod_id, s.time_id;
```

Find the daily total sales amount of each of the total channel, month and country standard code wise sales:

```
create materialized view sales_mv
as
select channels.channel_desc, calendar_month_desc,
countries.country_iso_code,
to_char(sum(amount_sold), '9,999,999,999') sales$
from sales, customers, times, channels, countries
where sales.time_id=times.time_id
and sales.cust_id=customers.cust_id
and customers.country_id = countries.country_id
```

```
and sales.channel_id = channels.channel_id
and channels.channel_desc in ('direct sales', 'internet')
and times.calendar_month_desc in ('2000-09', '2000-10')
and countries.country_iso_code in ('gb', 'us')
group by rollup (channels.channel_desc, calendar_month_desc,
countries.country_iso_code);
```

Find the daily total sales amount of each of the total channel, month and country standard code wise detail sales whose country standard code is ‘BD’ and ‘US’:

```
create materialized view sales_mv as
select channel_desc, calendar_month_desc, countries.country_iso_code,
to_char(sum(amount_sold), '9,999,999,999') sales$
from sales, customers, times, channels, countries
where sales.time_id=times.time_id and sales.cust_id=customers.cust_id and
sales.channel_id= channels.channel_id
and customers.country_id = countries.country_id
and channels.channel_desc in
('direct sales', 'internet') and times.calendar_month_desc in
('2000-09', '2000-10') and countries.country_iso_code in ('bd', 'us')
group by cube(channel_desc, calendar_month_desc,
countries.country_iso_code);
```

Find the daily total sales amount of each of the total channel, month and country standard code wise sales whose country standard code is ‘BD’ and ‘US’:

```
create materialized view sales_mv as
select channel_desc, calendar_month_desc, country_iso_code,
to_char(sum(amount_sold), '9,999,999,999') sales$, grouping(channel_desc)
as ch,
grouping(calendar_month_desc) as mo, grouping(country_iso_code) as co
from sales, customers, times, channels, countries
where sales.time_id=times.time_id
```

and sales.cust_id=customers.cust_id

and customers.country_id = countries.country_id

and sales.channel_id= channels.channel_id

and channels.channel_desc in ('direct sales', 'internet')

and times.calendar_month_desc in ('2000-09', '2000-10')

and countries.country_iso_code in ('bd', 'us')

group by rollup(channel_desc, calendar_month_desc, countries.country_iso_code);

Materialized views containing only joins: A join is a means for combining fields from two or more tables by using values common to each. But the joining of tables is expensive as it incurs a lot of steps to do the operation. Fig. 2.9 shows sample tables join operation between two tables. From the table join workflow, it is found that there are lots of processing steps for the join and it is much expensive. So if the table join can be pre-computed and stored in the database before the actual query occurs, the query performance will improve as it is not needed re-computing the join on the runtime of the query. The materialized view with the table joins serves the expensive table joins by pre-computing the join and storing the result in the database.

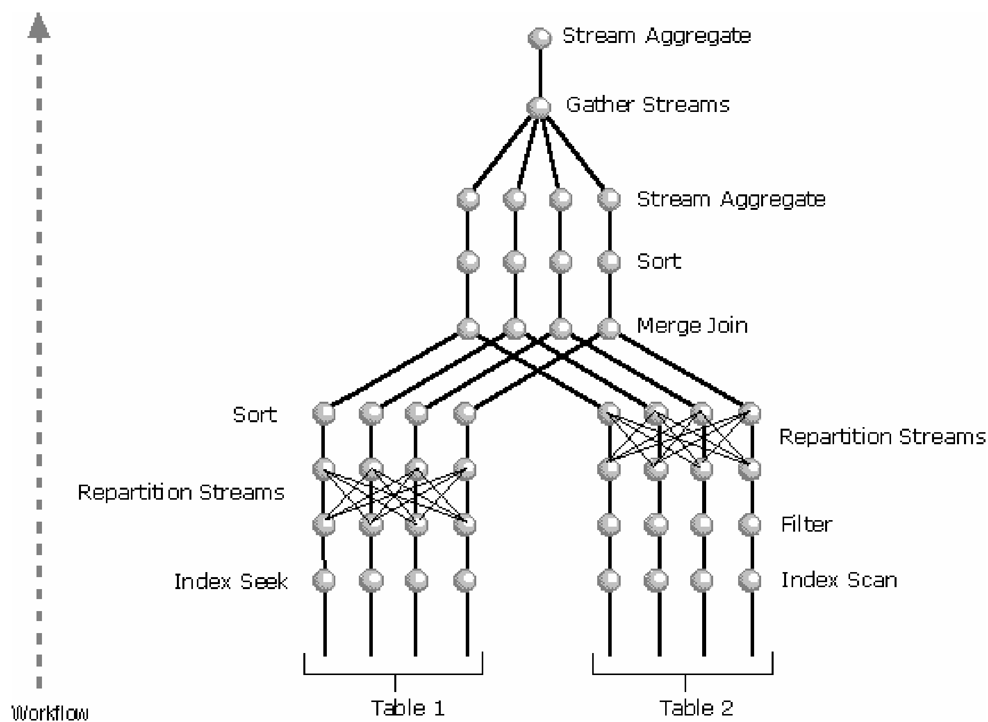


Fig. 2.9 Two tables join operation workflow

The following example shows the materialized view creation containing only the table joins:

```
create materialized view detail_sales_mv
parallel build immediate
refresh fast
as
select s.rowid "sales_rid", t.rowid "times_rid", c.rowid
"customers_rid",
c.cust_id, c.cust_last_name, s.amount_sold, s.quantity_sold, s.time_id
from sales s, times t, customers c
where s.cust_id = c.cust_id(+) and s.time_id = t.time_id(+);
```

Nested materialized views: A nested materialized view is a materialized view whose definition is based on another materialized view. A nested materialized view can reference other relations in the database in addition to referencing materialized view. Incrementally maintaining the distinct materialized aggregate views on the single join can take a long time because the underlying join need to perform many times. Using nested materialized views, multiple single-table materialized views can be created based on a joins only materialized view and the join is performed just once. In Oracle database, for creating a nested materialized view on materialized views, all parent and base materialized views must contain joins or aggregations. The following example creates a nested materialized view on another materialized view *join_sales_cust_time*:

```
create materialized view join_sales_cust_time
refresh fast on commit
as
select c.cust_id, c.cust_last_name, s.amount_sold, t.time_id,
t.day_number_in_week, s.rowid srid, t.rowid trid, c.rowid crid
from sales s, customers c, times t
where s.time_id = t.time_id and s.cust_id = c.cust_id;
```

```
create materialized view sum_sales_cust_time
refresh fast on commit
as
select count(*) cnt_all, sum(amount_sold) sum_sales,
count(amount_sold)
cnt_sales, cust_last_name, day_number_in_week
from join_sales_cust_time
group by cust_last_name, day_number_in_week;
```

2.7 Materialized View Maintenance

Materialized views must be kept up-to-date when the data used in the view definition changes. For instance, if the *amount* value of a loan is updated, the materialized view would become inconsistent with the underlying data and must be updated. The task of keeping a materialized view up-to-date with the underlying data is known as *materialized view maintenance*.

Materialized views can be maintained is several ways. One way to materialized views maintenance can be a manually written code: That is, every piece of code that updates the *amount* value of a loan can be modified to also update the total loan amount in the corresponding branch. Another option for maintaining materialized views is to define triggers on insert, delete and update of each relation in the view definition. The triggers must modify the contents of the materialized view, to take into account the change that caused the trigger to fire. A simplistic way of doing so is to completely recompute the materialized view on every update i.e., rematerializing the view for every update on the base relations. A better option is to modify only the affected parts of the materialized view, which is known as incremental materialized view maintenance. Several incremental materialized view maintenance techniques, algorithms and methods have been designed and developed in the decades and methods have been optimized for efficient incremental maintenance of the materialized views.

The existing view maintenance algorithms can be classified into two categories, namely, algorithmic and algebraic. Given a view and source updates, algorithmic view maintenance algorithms derive a program (a program can be a collection of deductive

rules or SQL statements) whose evaluation maintains the view. The first proposal is the finite differencing algorithm, for incremental view maintenance under a functional data model. The output of the maintenance algorithm adds several lines of code into the source update transaction in order to also update the view. It assumes set semantics of all base tables and a key is required to exist in the view. A counting algorithm for maintaining views under bag semantics essentially keeps track of the multiplicity of each view tuple, or in other words, the number of derivations of each view tuple. A delta is a count of rows or data that appears in the query but not in the materialized view. The insert deltas have a positive count while the delete deltas have a negative count. A view tuple is deleted from the view if its count becomes 0.

The main issues with algorithmic view maintenance algorithms are that (1) the correctness of the algorithms is hard to prove, especially when the view language is extended, it is unclear and hard to prove if the existing algorithms will still work; (2) the output of maintenance algorithms (a program as mentioned above) is also hard to optimize. Hence algebraic solutions have been proposed to address these limitations. More specifically, an algebraic approach pre-defines a set of primitive change propagation rules for each operator. The maintenance plan can then be constructed by propagating changes through each algebra operator in the view query algebra tree and recursively applying those primitive rules. The output of such algorithms, namely, the maintenance plan, can be optimized by a cost-based query optimizer. Also since the algorithm is algebra-based, the result is not tied to any particular query language.

Due to these benefits mentioned above, algebraic view maintenance algorithms have been extensively explored. Most existing work builds upon such an algebraic maintenance framework by considering more types of operators or considering different underlying data models. Algebra-based maintenance work has also been studied beyond the relational data model, e.g., maintaining *XQuery* views based on an XML algebra. The extensibility of such an algebraic maintenance framework lies in the fact that for each algebra operator, its change propagation is independent of its application context. Hence the existing change propagation rules can be reused for the same operator in more complex language constructs.

Incremental view maintenance techniques are applicable to many other applications, such as trigger/constraint processing, cache/replica maintenance etc. The view self-maintenance problem can also be considered as an application of the view maintenance techniques. That is, given a view and source updates, after generating the maintenance plan, we can easily determine if we need to query the sources or not. Some recent emerging applications, such as continuous query processing over data streams, are also closely related to the incremental view maintenance techniques.

2.7.1 Incremental materialized view maintenance

To incrementally maintain a materialized view, the changes need to be tracked regularly and only the changes applied to the materialized view. The changes to a relation that cause a materialized view to become out-of-date are inserts, deletes and updates. The changes (say inserts and deletes) to a relation or expression are referred to as its differential. The incremental maintenance process undergoes through several operations like joining of old materialized view and to the changes, selection operations, aggregation operations etc.

Join operation: Consider the materialized view $v = r \bowtie s$. Suppose the relation r is modified by inserting a set of tuples denoted by i_r. If the old value of r is denoted by r^{old}, and the new value of r by r^{new}, $r^{new} = r^{old} \cup i_r$. Now, the old value of the view, v^{old} is given by $r^{old} \bowtie s$, and new value of v^{new} is given by $r^{new} \bowtie s$. So $r^{new} \bowtie s$ can be rewritten as $(r^{old} \cup i_r) \bowtie s$, which can again be rewritten as $(r^{old} \bowtie s) \cup (i_r \bowtie s)$. In other words,

$$v^{new} = v^{old} \cup (i_r \bowtie s)$$

Thus, to update the materialized view v, it simply needs to add the tuples $i_r \bowtie s$ to the old contents of the materialized view. Inserts to s are handled in an exactly symmetric fashion.

Now suppose r is modified by deleting a set of tuples denoted by d_r. Using the same reasoning as above,

$$v^{new} = v^{old} - (d_r \bowtie s)$$

Deletes on s are handled in an exactly symmetric fashion.

Selection and projection operations: Consider a view $v = б_{\theta}\ (r)$. If r is modified by inserting a set of tuples i_r, the new value of v can be computed as

$$v^{new} = v^{old} \cup б_{\theta}\ (i_r)$$

Similarly, if r is modified by deleting a set of tuples d_r, the new value of v can be computed as

$$v^{new} = v^{old} - б_{\theta}\ (d_r)$$

Projection is a more difficult operation with which to deal. Consider, a materialized view $v = \Pi_A(r)$. Suppose the relation r is on the schema $R = (A, B)$, and r contains two tuples $(a, 2)$ and $(a, 3)$. Then, $\Pi_A(r)$ has a single tuple (a). If the tuple $(a, 2)$ is deleted from r, the tuple (a) cannot be deleted from $\Pi_A(r)$: If it did so, the result would be an empty relation, whereas in reality $\Pi_A(r)$ still has a single tuple (a). The reason is that the same tuple (a) is derived in two ways, and deleting one tuple from r removes only one of the ways pf deriving (a); the other is still present.

This reason also gives the intuition for solution for each tuple in a projection such as $\Pi_A(r)$, a count of how many times it was derived will be kept.

When a set of tuples d_r is deleted from r, for each tuple t in d_r the following can be done. Let $t.A$ denote the projection of t on the attribute A. $(t.A)$ is found in the materialized view, and the count is decreased stored with it by 1. If the count becomes 0, $(t.A)$ is deleted from the materialized view.

Handling insertions is relatively straightforward. When a set of tuples i_r is inserted into r, for each tuple t in i_r the following can be done. If $(t.A)$ is already present in the materialized view, the count is increased stored with it by 1. If not, $(t.A)$ is added to the materialized view, with the count set to 1.

Aggregation operations: Aggregation operations proceed somewhat like projections. The aggregate operations in SQL are count, sum, avg, min, max etc. Here on the aggregate operation sum is discussed.

Sum: Consider a materialized view $v = Ag_{sum(B)}(r)$. When a set of tuples i_r is inserted into r, for each tuple t in i_r the following can be done. The group $t.A$ is to be looked in the materialized view. If it is not present, ($t.A$, $t.B$) is added to the materialized view; in addition, a count of 1 is stored associated with ($t.A$, $t.B$), just as did for the projection. If the group $t.A$ is present, the value of $t.B$ is added to the aggregate value for the group, and 1 is added to the count of the group.

When a set of tuples d_r is deleted from r, for each tuple t in d_r, the following can be done. The group $t.A$ is to be looked in the materialized view, and $t.B$ is to be subtracted from the aggregate value for the group. Also 1 can be subtracted from the count for the group, and if the count becomes 0, the tuple for the group $t.A$ is deleted from the materialized view.

Without keeping the extra count value, it would not be able to distinguish a case where the sum for a group is 0 from the case where the last tuple in a group is deleted.

Other operations: The set operation *intersection* is maintained as follows. Given materialized view $v = r \cap s$, when a tuple is inserted in r, it is checked if it is present in s, and if so it is added to v. If a tuple is deleted from r, it is deleted from the intersection if it is present. The other set operations, *union* and *set difference*, are handled in a similar fashion as with the intersection set operation.

Outer joins are handled in much the same way as joins, but with some extra work. In the case of deletion from r tuples in s have to be handled that no longer match any tuple in r. In the case of insertion to r, tuples in s have to be handled that did not match any tuple in r.

Handling expressions or statements: To handle an entire expression, expressions can be derived for computing the incremental change to the result of each subexpression, starting from the smallest subexpressions. For example, suppose a materialized view $E_1 \bowtie E_2$ is to

be updated incrementally when a set of tuples i_r is inserted into relation r. Let assume r is used in E_1 alone. Suppose the set of tuples to be inserted into E_1 is given by expression D_1. Then the expression $D_1 \bowtie E_2$ gives the set of tuples to be inserted into $E_1 \bowtie E_2$.

Query optimization: Query optimization can be performed by treating materialized views just like regular relations.

- *Rewriting queries to use materialized views:* Suppose a materialized view $v = r \bowtie s$ is available, and a user submits a query $r \bowtie s \bowtie t$. Rewriting the query as $v \bowtie t$ may provide a more efficient query plan than optimizing the query submitted. Thus, it is the job for the query optimizer to recognize when a materialized view can be used to speed up a query.

- *Replacing a use of a materialized view by the view definition:* Suppose a materialized view $v = r \bowtie s$ is available, but without any index on it, and a user submits a query $\sigma_{A=10}(v)$. Suppose also that s has an index on the common attribute B, and r has an index on attribute A. The best plan for this query may be to replace v by $r \bowtie s$, which can lead to the query plan $\sigma_{A=10}(r) \bowtie s$; the selection and join can be performed efficiently by using the indices on $r.A$ and $s.B$, respectively. In contrast, evaluating the selection directly on r may require a full scan of v, which may be more expensive.

2.7.2 Materialized view selection

Materialized view selection is an optimization problem, namely, “what is the best set of views to materialize?” This decision must be made on the basis of the system workload, which is a sequence of queries and updates that reflects the typical load on the system. One simple criterion would be to select a set of materialized views that minimizes the overall execution time of the workload of queries and updates, including the time taken to maintain the materialized views.

Typically view selection is under a space constraint, and / or a maintenance cost constraint. Unlike answering queries using views that need to handle ad-hoc queries, in view selection scenarios, the queries are known. Hence, most view selection algorithms

start from identifying common sub-expressions among queries. These common sub-expressions serve as the candidates of the materialized views. One fundamental practical issue with view selection is that there are many possibly competing factors to be considered during the view selection phase, such as view selectivity, query complexity, database size, query performance, update performance etc.

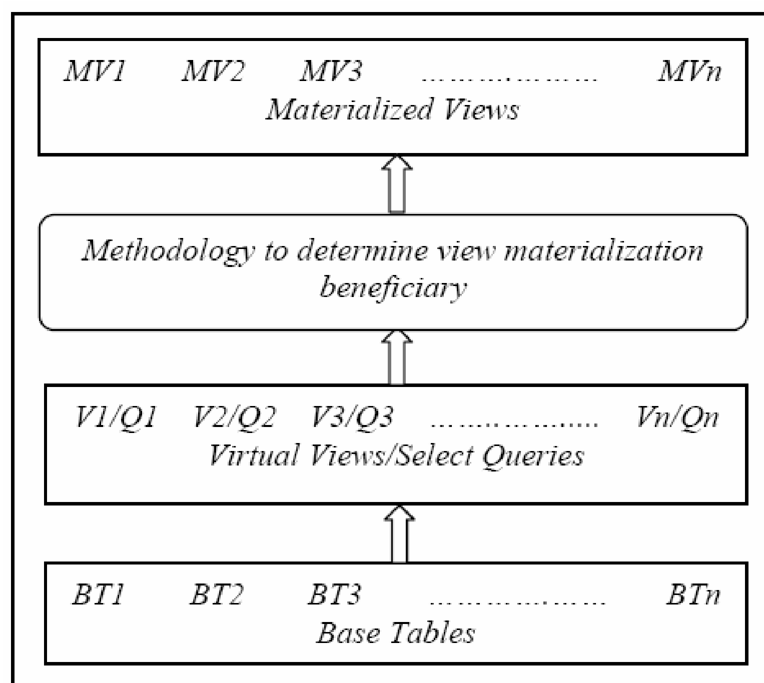


Fig. 2.10 A view materialization process

Fig. 2.10 shows a typical view materialization process where the methodology determines what kind of views is beneficial to materialize under the conditions like view selectivity, complexity, database size, view maintenance cost, access frequency etc.

2.8 Query Rewrite

Query rewrite transforms a SQL statement expressed in terms of tables or views into a statement accessing one or more materialized views that are defined on the detail tables. The transformation is transparent to the end user or application, requiring no intervention and no reference to the materialized view in the SQL statement. Because query rewrite is transparent, materialized views can be added or dropped just like indexes without invalidating the SQL in the application code.

A query undergoes several checks to determine whether it is a candidate for query rewrite. If the query fails any of the checks, then the query is applied to the detail tables rather than the materialized view. This can be costly in terms of response time and processing power.

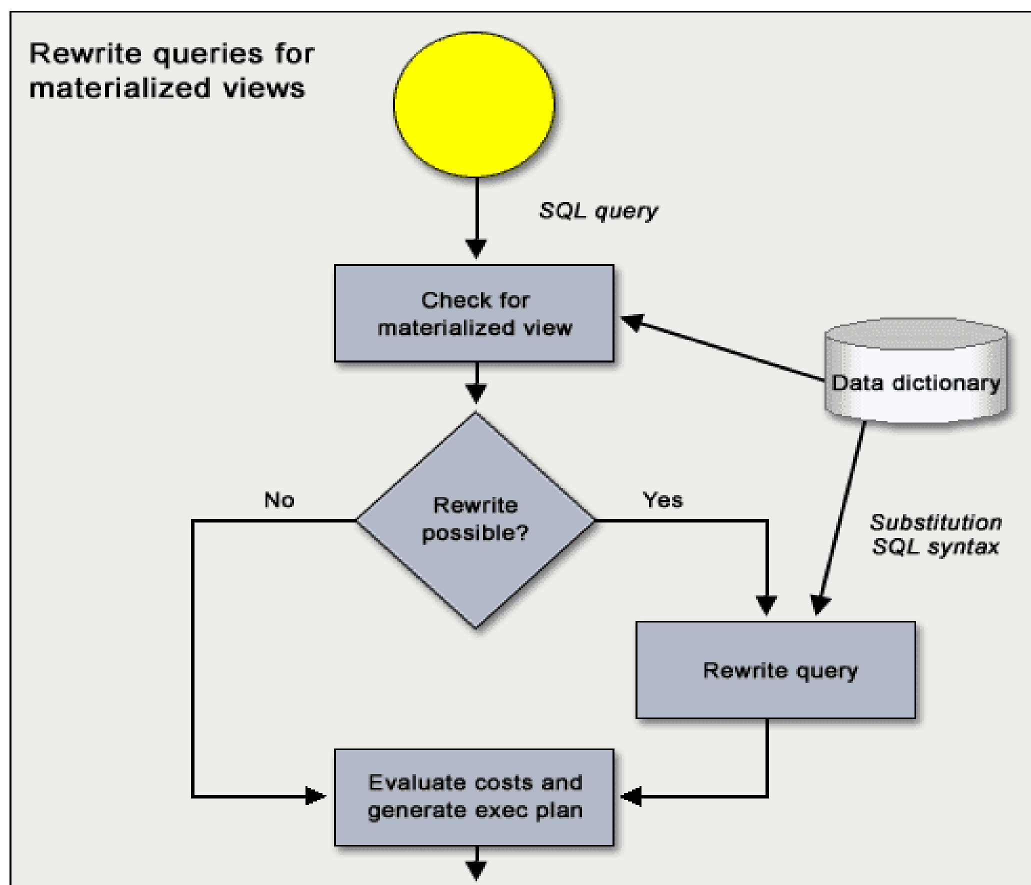


Fig. 2.11 Oracle SQL query rewrite mechanism

Figure 2.11 shows how the Oracle SQL optimizer checks the Oracle data dictionary for the presence of a materialized view whenever a new SQL statement enters the Oracle library cache.

The optimizer uses two different methods to recognize when to rewrite a query in terms of a materialized view. The first method is based on matching the SQL text of the query with the SQL text of the materialized view definition. If the first method fails, the optimizer uses the more general method in which it compares joins, selections, data columns, grouping columns and aggregate functions between the query and materialized views.

2.8.1 How oracle rewrites queries?

The optimizer uses a number of different methods to rewrite a query. The first step in determining whether query rewrite is possible is to see if the query satisfies the following prerequisites:

- Joins present in the materialized view are present in the SQL.
- There is sufficient data in the materialized view(s) to answer the query.

After that, it must determine how it will rewrite the query. The simplest case occurs when the result stored in a materialized view exactly matches what is requested by a query. The optimizer makes this type of determination by comparing the text of the query with the text of the materialized view definition. This text match method is most straightforward but the number of queries eligible for this type of query rewrite is minimal.

When the text comparison test fails, the optimizer performs a series of generalized checks based on the joins, selections, grouping, aggregates, and column data fetched. This is accomplished by individually comparing various clauses (SELECT, FROM, WHERE, HAVING, or GROUP BY) of a query with those of a materialized view.

2.8.2 General query rewrite method

The optimizer has a number of different types of query rewrite methods that it can choose from to answer a query. When text match rewrite is not possible, this group of rewrite methods is known as general query rewrite. The advantage of using these more advanced techniques is that one or more materialized views can be used to answer a number of different queries and the query does not always have to match the materialized view exactly for query rewrite to occur.

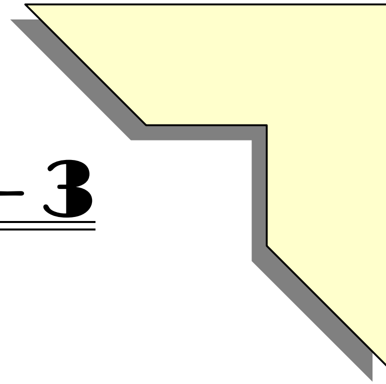
When using general query rewrite methods, the optimizer uses data relationships on which it can depend, such as primary and foreign key constraints and dimension objects. For example, primary key and foreign key relationships tell the optimizer that each row in the foreign key table joins with at most one row in the primary key table. Furthermore, if there is a NOT NULL constraint on the foreign key, it indicates that each row in the

foreign key table must join to exactly one row in the primary key table. A dimension object will describe the relationship between, say, day, months, and year, which can be used to roll up data from the day to the month level.

Data relationships such as these are very important for query rewrite because they tell what type of result is produced by joins, grouping, or aggregation of data. Therefore, to maximize the rewritability of a large set of queries when such data relationships exist in a database, you should declare constraints and dimensions.

2.8.3 Types of query rewrite

Queries that have aggregates that require computations over a large number of rows or joins between very large tables can be expensive and thus can take a long time to return the results. Query rewrite transparently rewrites such queries using materialized views that have pre-computed results, so that the queries can be answered almost instantaneously. These materialized views can be broadly categorized into two groups, namely materialized aggregate views and materialized join views. Materialized aggregate views are tables that have pre-computed aggregate values for columns from original tables. Similarly, materialized join views are tables that have pre-computed joins between columns from original tables. Query rewrite transforms an incoming query to fetch the results from materialized view columns. Because these columns contain already pre-computed results, the incoming query can be answered almost instantaneously.



CHAPTER – 3

VIEW MATERIALIZATION

3.1 Introduction

3.2 Brief Description of the Factors

3.3 Performance Evaluation Measurement

3.4 Methodology to Determine View Materialization

3.5 Dynamic Mathematical Model for Selection of Views for Materialization and Removal of Old Materialized Views

3.5.1 Dynamic selection of views for materialization

3.5.2 Dynamic removal of old materialized views

C h a p t e r 3

VIEW MATERIALIZATION

This chapter discusses about the theoretical background of the research work, design and development of methodology, dynamic cost model for view materialization and finally removal of old materialized views.

3.1 Introduction

A materialized view is a pre-calculated result of a query and the materialized view is stored in the database with the data unlike with the virtual views where only the view definition is stored in the database and when a query is issued against the virtual view, the result is actually performed by computing from the base tables. View materialization is profitable for the query performance as the result of the query is already pre-computed. But it should also be in mind that materializing incurs space utilization and maintenance cost. Rather than rematerializing incremental update propagation is desirable to reduce the maintenance cost. It is necessary to determine a query or a virtual view is really profitable for materializing or not in different circumstances like - views with different aggregations, containing only joins and containing set operations or nesting of views and also need to compare whether incremental materialized view maintenance performance is beneficial than rematerializing or not. One particular view or query materialization is not only a solution for improving the query performance as in a certain time period lots of queries are issued to retrieve result from the database and all of the queries or views cannot be materialized due the maintenance cost and space cost constraints. So it is expected to select a set of views from a list of all queries or views requested at a particular time period based on the access frequency of the query or the view, selectivity, database size, query complexity (like number of joins, aggregations and tables involved in the query), query execution cost and also the update frequency and materialized view maintenance cost. It is also necessary to remove the old materialized view that is no longer in use or the materialized view access frequency is too low.

In this thesis, we have developed a methodology that evaluates the incremental materialized view maintenance performance over rematerializing and determines the

various situations in which a view or a query is beneficial for selecting to be materialized considering the incremental materialized view maintenance cost and the performance evaluation criteria based on the issues like - view selectivity, complexity and database size.

We have also developed a dynamic cost model to dynamically select a set of query or virtual views out of all of the queries or views requested by the users at a particular time period considering the access frequency of the queries, weighting factor reflecting the importance of the query, query execution time, query complexity (number of tables, joins and aggregations involved in the query), view selectivity, database size, update frequency, weighting factor reflecting the importance of the table and view maintenance cost. Finally, we have provided another dynamic cost model to remove the old materialized view that is no longer in use or the materialized views access frequency is too low.

In the next section, a brief description of the factors like - view selectivity, complexity, database size, query execution time, view maintenance cost has been provided.

3.2 Brief Description of the Factors

***View selectivity*:** Selectivity is defined as the ratio of the number of records selected by a query to the number of input records. Different view selectivities arise as a result of a view containing the predicates that filter (to different degrees) the input data. For example, a view template contains the following definition:

```
create view <view_name>
as
select <column_i>
from <table>
where φ_i;
```

From the above view template, more than one view can be derived with different selectivities by varying the predicate ϕ_i. Lets say, if there are 1,00,000 records in the table, then each predicate can be obtained with $column <= k_j$ where k_1 = 20,000 to k_5 = 1,00,000 with the selectivities from 0.2 to 1.0 in increments of 0.2.

View complexity: For the structural complexity of the view, here it is assumed to vary the number and the kind of algebraic operators needed to evaluate the query, number of joining and the number of tables involved as a part of the query or the view. For example, a query that contains two joins is more complex than a query that contains one join. From the following two view templates the later template is more complex comparing to the first one as the later selects data from two tables and contains two joins.

create view <*view_name*>
as
select <*column*>
from <*table*>
where ϕ_i;

create view <*view_name*>
as
select $a.col_1$, $b.col_2$
from $table_1$ *a*, $table_2$ *b*
where $a.col_1 <= \phi_j$ and $b.col_2 >= \phi_k$;

Database size: With regard to the database size, the number of records varies uniformly across different databases. For example, the following view template can retrieve all records from its query with the different database as specified in the Table 3.1.

create view <*view_name*>
as
select $a.col_1$, $b.col_2$
from $table_1$ *a*, $table_2$ *b*;

Table 3.1 Database size example

Database Size					
Database	db_1	db_2	db_3	db_4	db_5
Records	100000	200000	300000	400000	500000
Size (GB)	1	2	3	4	5

Maintenance cost: The update propagation time to the materialized view is considered as the maintenance cost of the materialized view.

Cost of query answering: The query response time or the query execution time is the cost of answering a query.

Query access frequency: The number of times a query is requested at a particular time period is the access frequency of that query.

Tables, joins and aggregations: The number of tables involved in a query; the number of joins in the query and the number aggregate operators used in the query.

3.3 Performance Evaluation Measurement

The performance of incremental materialized view maintenance and the determination of view materialization profitable can be evaluated by considering:

i. The cost of incrementally maintaining the materialized view by update propagation;
ii. The cost of answering query over the materialized view and over its virtual equivalent;
iii. The cost of answering query over the materialized view and over its query rewrite;
iv. The cost of incrementally maintaining the materialized view in comparison with the cost of answering a query over its virtual equivalent;
v. The cost of incrementally maintaining the materialized view in comparison with the cost of answering a query rewrite;
vi. The cost of answering a query using query rewrite in comparison with answering a virtual view.

The first consideration (i) compares the view maintenance cost between incremental maintenance and rematerializing. The (ii) and (iii) points compare the query answering costs between materialized views, virtual views and using rewrites. The last three points (iv), (v) and (vi) compare the relative costs incremental maintenance cost, query answering between materialized views, virtual views and rewrites. All of the above evaluation measurements can be applied to the issues like - selectivity of views, complexity and database size.

3.4 Methodology to Determine View Materialization

We have developed a methodology to evaluate the incremental materialized view maintenance performance and to determine the circumstances in which a view is beneficial to be selected for materialization considering the incremental materialized view maintenance cost based on the factors like - view selectivity, complexity and database size in object-relational database management system. The view materialization determination methodology is illustrated in Fig. 3.1.

Given:

V_i = A set of virtual view definitions to be materialized;

I_i = A set of issues affecting the view performance like selectivity, complexity and database size;

M_i = A set of materialized view as the conclusion drawn from simulated output.

Begin

While (all virtual views (V_i) are materialized (M_i)) {

  While (all issues (I_i): selectivities/complexities/database sizes are applied) {

    Apply update events to the base tables;

    Apply rematerializing and incremental maintenance separately;

    Assign:

      IncMAT[*j*] = *Compute the incremental refresh time*;

      ReMAT[*j*] = *Compute the rematerializing refresh time*;

      CoQVIR[*j*] = *Compute the cost of answering query using the virtual view*;

      CoQMAT[*j*] = *Compute the cost of answering query using materialized view*;

      CoQRW[*j*] = *Compute the cost of answering query using query rewrite*;

      RelVvMT[*j*] = *CoQVIR*[*j*] / *IncMAT*[*j*];

      RelRWvMT[*j*] = *CoQRW*[*j*] / *IncMAT*[*j*];

      RelVvRW[*j*] = CoQVIR[*j*] / CoQRW[*j*];

  } ***End While***;

  For *k* = 1 to all selectivities/complexities/database sizes {

    Plot (*a*, *b*) = {(*Elapsed Time*, *IncMAT*[*j*]), (*Elapsed Time*, *ReMAT*[*j*])};

    Plot (*c*, *d*) = {(*Elapsed Time*, *CoQVIR*[*j*]), (*Elapsed Time*, *CoQMAT*[*j*])};

    Plot (*e*, *f*) = {(*Elapsed Time*, *CoQVIR*[*j*]), (*Elapsed Time*, *CoQRW*[*j*])};

    Plot (*g*, *h*) = {(*No. of updates*, *RelVvMT*[*j*])};

    Plot (*m*, *n*) = {(*No. of updates*, *RelRWvMT*[*j*])};

    Plot (*p*, *q*) = {(*No. of rewrites*, *RelVvRW*[*j*])};

  } ***End For***;

} ***End While***;

End;

Fig. 3.1 Methodology to determine view materialization

The goal of the methodology for view materialization is to determine various conditions when a view or a query can be selected for materialization to improve overall query performance. The methodology evaluates the performance of incremental materialized view maintenance and finally from the simulated result conclusion will be drawn for the different situations under which a view can be selected for materialization.

The working principle of the methodology is that it takes a set of virtual view definitions or select queries and the performance affecting factors like - selectivity, complexity or database size as the input. Inference can be drawn from the simulated output that a particular view or all of the views can be selected to be materialized if and only if the incremental maintenance of that view is cost effective. So according to methodology, first all of the virtual view definitions and corresponding defined materialized view definitions are based on the different conditions like - various types of aggregations, joining and nesting of views, set operations etc. Then update events like - insertion of data, deletion of data or modification of data are applied to the base tables. To propagate the changes to the materialized view for the latest changes on the base tables, re-materialization and incremental materialized view maintenance are applied to the materialized views independently. The propagation time and query response time have been considered here as the cost of maintenance and cost of query answering respectively. The materialized view maintenance cost using re-materialization and incremental maintenance are computed.

The cost of answering a virtual view, materialized view and query rewrite are also computed. A measurement of the effectiveness can be done by comparing the different cost and it is defined as relative cost. The relative costs of answering a virtual view vs. incremental maintenance and relative costs of answering a virtual view vs. using rewrite are calculated. Finally, the results are plotted in different simulated output graphs: incremental view maintenance vs. re-materialization graph, query answering using virtual view vs. materialized view graph, query answering using virtual view vs. query rewrite graph, relative costs graph of answering virtual view vs. incremental materialized view maintenance, relative costs graph of query answering using query rewrite vs. incremental materialized view maintenance and relative costs graph of answering virtual view vs. query rewrite. Conclusion is drawn from the simulated output regarding the performance of incremental maintenance and the circumstances under which view materialization is beneficial or not.

3.5 Dynamic Cost Model for Selection of Views for Materialization and Removal of Materialized Views

View materialization improves the query performance but materializing views in a predefined time cannot be a perfect solution. Because those queries or views that have been materialized might not have been used for long time or the access frequencies of these materialized views are very low while the materialized views occupying storage space and also to be updated with the base tables, the maintenance of the materialized views is a necessary in a periodic schedule. At a particular time period, there may be a large number of queries or views is answered that might not be materialized previously to improve the query performance. But it is also not possible that all the queries or views at a particular time period should be materialized for improving query performance as there are lots of issues are involved with the view materialization. The issues are like - access frequency of the queries, execution time, view selectivity, database size, complexity (number of tables, joins and aggregations involved) and maintenance cost based on update frequency and the importance of the table. As our main goal is to improve the query performance, we are considering the storage space is sufficient to provide the necessary space for view materialization.

We have designed a dynamic cost model in the next section that selects a set of queries or views from a pool set of views or queries at a particular time period based on factors like - the access frequencies of the queries, query processing cost and maintenance cost. The dynamic model selects views with the combination of higher query access frequency and higher execution time to materialize at a certain time difference from a set of large number views or queries to improve query performance. The query processing cost includes the cost of execution time of the query, selectivity, complexity (number of tables, joins and aggregations involved), query access frequency and a weighting factor reflecting the importance of the query. Maintenance cost includes the update frequency and the weighting factor reflecting the importance of the base tables. Finally, we have designed another dynamic cost model and proposed algorithm to remove the old materialized views that are no longer in use for a long time or the access frequencies of the materialized views are very low. The most important criteria in the dynamic model is that for both of the dynamic selection of views for materialization and removal of old materialized views, the threshold level is selected dynamically.

3.5.1 Dynamic selection of views for materialization

The costs and factors that we have considered here for the dynamic selection of views for materialization are shown in the following Table 3.2. It shows the access frequencies of the queries, weight of different queries and different cost associated with the queries or views like - execution cost, selectivity, complexity issues, and maintenance cost for the update frequency of the base tables.

The first column in the table represents the SQL statements that are called queries or views. All the queries used in the table are unique SQL statements in a particular time.

The second column is the access frequency count of each query or view at that particular time period. We have assigned a weighting factor to reflect the importance of the query.

The third column presents the query or view execution time and it is called the response time of that query.

The fifth column of the table is the selectivity of the query and it is the ratio of the number of rows retrieved by the query to the number of input rows.

The sixth, seventh and eighth columns focus on the complexity issue of the query by calculating the total number of tables involved in the query, total number of joining occurred and the total number aggregate operators used in the query.

The remaining columns are used to calculate the maintenance cost of the view by summing up the total table maintenance costs for those tables that are involved in the query. For the maintenance cost of the view, corresponding involved table maintenance cost is places in the columns. If a table maintenance cost is not associated with the query then a "0" is placed in that column. For example, if table t_1 and table t_2 are involved in the query Q_1, then the maintenance cost of the table t_1 and t_2 are placed in the columns for the view maintenance cost. Remaining table maintenance costs are set as "0". All the information of the table is found from the database. After filled up the table, a (n, 12 + m) matrix is formed where n is the total number queries and m is the total number tables in the schema. In the subsequent paragraph, the calculation of each of the factors and cost associated with the query or the view has shown.

Table 3.2 Different costs and factors associated with queries or views

Query/ View (Q_n/V_n)	Query Access Frequency (f_n)	Weighting Factor ($w_n = log_k (f_n+1)$)	Query Execution Time (e_n)	Query Selectivity (s_n)	No. of Tables Involved (t_n)	No. of Joining Occurs (j_n)	No. of Aggregations (a_n)	Query Complexities ($c_n = t_n+j_n+a_n$)	Query Processing Cost ($QP_n = f_n w_n e_n s_n c_n$)	Table Maintenance Cost, MT_1	Table Maintenance Cost, MT_2	…	Table Maintenance Cost, MT_m	View Maintenance Cost ($MC_n = MT_1+MT_2+...+MT_m$)	Total Cost ($TC_n = QP_n + MC_n$)
Q_1/V_1	f_1	w_1	e_1	s_1	t_1	j_1	a_1	c_1	QP_1	*cost*	*cost*	…	*cost*	MC_1	TC_1
Q_2/V_2	f_2	w_2	e_2	s_2	t_2	j_2	a_2	c_2	QP_2	0	*cost*	…	*cost*	MC_2	TC_2
Q_3/V_3	f_3	w_3	e_3	s_3	t_3	j_3	a_3	c_3	QP_3	*cost*	0	…	0	MC_3	TC_3
Q_4/V_4	f_4	w_4	e_4	s_4	t_4	j_4	a_4	c_4	QP_4	0	cost	…	0	MC_4	TC_4
Q_5/V_5	f_5	w_5	e_5	s_5	t_5	j_5	a_5	c_5	QP_5	*cost*	0	…	*cost*	MC_5	TC_5
…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…	…	…	…	…	*cost*	…	…
Q_n/V_n	f_n	w_n	e_n	s_n	t_n	j_n	a_n	c_n	QP_n	0	*cost*	…	0	MC_n	TC_n

In the following sections, some important terms have been briefly discussed and we developed some algorithms and equations that are used to compute the cost of query.

Access frequency of the query: The query access frequency (a_n) is the counting of the total number of times of the execution of a particular query occurred in the database or in the application. For example, let's say there are 10 unique queries or views that have been executed at a certain time difference and the execution status of the queries are depicted in a sample representation in the figure 3.2:

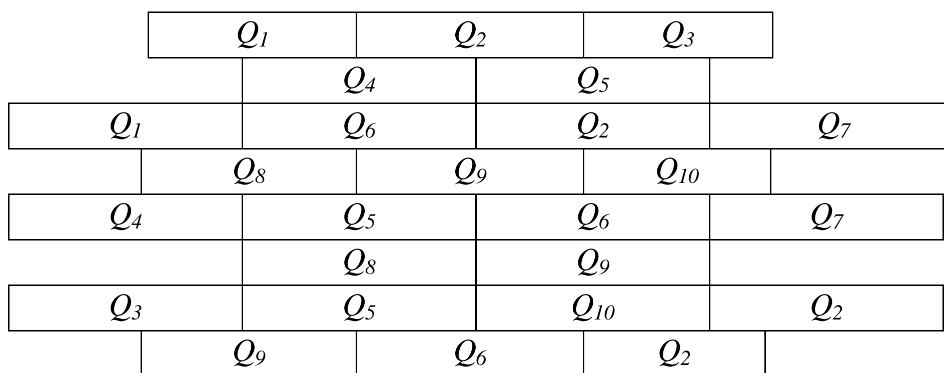


Fig. 3.2 Sample representation of query execution frequencies

From the Fig. 3.2 the access frequency count of different queries can be found in the Table 3.3:

Table 3.3 Access frequency count total for Fig. 3.2

Queries (Q_n)->	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}
Frequency (f_n)->	2	4	2	2	3	3	2	2	3	2

The access frequency of the query or how many times a query has been executed can be found by querying the database history for SQL statements. The following algorithm *FindAccessFrequency* in Fig. 3.3 computes the access frequencies of the queries:

```
Algorithm FindAccessFrequency
{
Given:
        Input: A set of queries or views;
        Output: Query-Access Frequency (n × 2) table matrix;
        count = 0;
Begin
        For i = 1 to total number of queries (n) {
        Assign:
                Qi = getNextQuery ( );
                SearchExistingQuery in the database SQL history;
                If found ( ) {
                        count = count + 1;
                        Add n to the query-access frequency matrix (Qi, fi);
                }
                Else {
                        count = 1;
                        Add count to the query-access frequency matrix (Qi, fi);
                }
        } End For;
End;
}
```

Fig. 3.3 Algorithm for finding access frequencies of the queries

Weighting factor for the query: To emphasis on the query access frequency, a weighting factor (w_n) has been assigned to reflect the importance of the query. If a query has higher access frequency in the system, the higher is the weight. The weight can be calculated as,

$$Weight(w_n) = \log_k(f_n + 1) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

The value of k is either 2 or 10. If the difference between the maximum and the minimum frequency is using high, k is 10; otherwise it is 2.

Query Execution time: The time taken to execute a query is called the execution time (e_n) of that query. The execution time of a query can be found from the database while requesting the query or by enabling the execution time display.

Query Selectivity: As in the section 3.2, selectivity of view or query is defined as,

$$Selectivity\ (s_n) = \frac{No.of\ records\ selected\ by\ the\ query}{Total\ no.of\ input\ records} \quad \ldots\ldots\ldots\ldots (2)$$

Query Complexity: We have assumed that a query is more complex when the query includes more than one table, more than one join or more than one aggregate operator in its *SQL* statements. So the complexity is defined as

Complexity (c_n) = *No. of tables involved + No. of joining occurred + No. of aggregate operator used*

$$= t_n + j_n + a_n \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (3)$$

An algorithm *FindQueryComplexities* is provided in Fig. 3.4 for finding the query complexities of number of tables involved in the query, number of joining occurred and number of aggregate operators used for summarized data.

Algorithm FindQueryComplexities {
Given: Input: A set of queries or views;
Output: Query-Complexity ($n \times 4$) table matrix;
Begin
For i = 1 to total number of queries (n) {
Assign: Q_i = *getNextQuery* ( );
SearchNumberofTablesInvolved (t_i);
Add count total to *QueryTableCount* (Q_i, t_i);
SearchNumberofJoiningOccurred (j_i);
Add count total to *QueryJoiningCount* (Q_i, j_i);
SearchNumberofAggregateOperatorsUsed (a_i);
Add count total to *QueryAggregationCount* (Q_i, a_i);
Total Query Complexity, $c_i = t_i + j_i + a_i$;
} ***End For***;
End; }

Fig. 3.4 Algorithm for finding query complexities namely no. tables, joining and aggregations

Query processing cost: To compute the total query processing cost (QP_n), we have considered the query access frequency, weighting factor, query execution time, selectivity, and complexity of the query. After completion of the (n, 8 + m) cost associated table matrix, by applying the modified data mining algorithm [40] for finding the large total cost associated with the queries, the query processing cost of each query can be calculated as,

Query processing cost of query Q_1, $(QP_1) = f_1 w_1 e_1 s_1 c_1$ . . . . (4a)

Query processing cost of query Q_2, $(QP_2) = f_2 w_2 e_2 s_2 c_2$ . . . . (4b)

Query processing cost of query Q_3, $(QP_3) = f_3 w_3 e_3 s_3 c_3$ . . . . (4c)

. . . .

Query processing cost of query Q_n, $(QP_n) = f_n w_n e_n s_n c_n$ . . . . (4n)

Update frequency of base tables: The update frequency of the base tables (u_m) is the counting of the total number of updates on the base tables at a particular time period. The updates to the base tables can be insertion of new rows, modification of existing rows and deletion of rows from the tables. We can calculate the update frequency of the base tables as,

$$Update\ frequency\ (u_m) = \frac{No.of\ Times\ Insertions + No.of\ Times\ Modification + No.of\ Times\ Deletion}{3} \quad \ldots\ldots (5)$$

$$= \frac{i_m + m_m + d_m}{3}$$

Let's say, in a database there are five base tables and at a particular time period the updates to the tables are insertion, modification and deletion of rows. To illustrate the example, a sample table-update frequency status is shown in Table 3.4.

Table 3.4 Sample table-update frequency status

Tables (t_m)	No. of times updates to the tables			Update Frequency (u_m)
	Insertion (i_m)	Modification (m_m)	Deletion (d_m)	
t_1	10	4	2	$=\frac{10+4+2}{3}=5.33$
t_2	8	3	4	$=\frac{8+3+4}{3}=5$
t_3	0	10	6	$=\frac{0+10+6}{3}=5.33$
t_4	7	0	0	$=\frac{7+0+0}{3}=2.33$
t_5	9	5	0	$=\frac{9+5+0}{3}=4.67$

The following algorithm *FindUpdateFrequency* in Fig. 3.5 is used to compute the total update frequency of a query for the base tables:

Algorithm FindUpdateFrequency

{

Given:

Input: A set of database tables;

Output: Table-Update Frequency ($m \times 2$) matrix;

Begin

For i = 1 to total number of tables (m) {

Assign:

t_i = *getNextTable* ( );

i_j = *SearchTableDataInsertHistory* for table t_i;

m_j = *SearchTableDataModificationHistory* for table t_i;;

d_j = *SearchTableDataDeletionHistory* for table t_i;

$$u_j = \frac{i_j + m_j + d_j}{3};$$

Add u_j to Table-Update Frequency table matrix (t_i, u_i);

} ***End For***;

End;

}

Fig. 3.5 Algorithm for finding update frequencies of the base tables

Weighting factor for the table: To emphasis on the base table update frequency, we have assigned a weighting factor (w_m) considering that the higher the frequency of the update in the tables, the higher is the weight. To each table, the weight can be calculated as,

$$Weight(w_m) = \log_k (u_m + 1) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (6)$$

The value of k is either 2 or 10. If the difference between the minimum and the maximum frequency is using high, k is 10; otherwise it is 2.

View maintenance cost: View maintenance is the process of updating pre-computed views when the base fact table is updated. The maintenance cost for the materialized view is the cost used for refreshing this view whenever a change is made to the base table. We have calculated the maintenance cost using the update frequency of the base table and the weighting factor to reflect the importance of the base table update frequency.

The associated update frequencies, corresponding weighting factors and maintenance cost of the tables can be depicted in the following Table 3.5.

Table 3.5 Table maintenance costs

Table (t_m)	Update Frequency (u_m)	Weighting Factor (w_m)	Table Maintenance Cost (MT_m)
t_1	u_1	w_1	MT_1
t_2	u_2	w_2	MT_2
t_3	u_3	w_3	MT_3
. . .	. . .	. . .	. . .
t_m	u_m	w_m	MT_m

The maintenance cost of a table can be defined as of the multiplication of the update frequency of that table and the weighting factor reflecting the importance of the table by applying the modified data mining algorithm for finding large table maintenance cost,

$$Maintenance\ cost\ of\ a\ table_1,\ MT_1 = u_1 w_1 \quad \ldots\ldots\ldots\ldots\ldots (7a)$$

$$Maintenance\ cost\ of\ a\ table_2, MT_2 = u_2 w_2 \ldots\ldots\ldots\ldots \text{(7b)}$$

$$Maintenance\ cost\ of\ a\ table_3, MT_3 = u_3 w_3 \ldots\ldots\ldots\ldots \text{(7c)}$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$Maintenance\ cost\ of\ a\ table_m, MT_m = u_m w_m \ldots\ldots\ldots\ldots \text{(7m)}$$

The total maintenance cost associated with a query is the summation of the maintenance costs of the tables that are involved with the query and is defined as,

$$Maintenance\ Cost\ of\ Query\ Q_1, MC_1 = MT_1 + MT_2 + MT_3 + \ldots + MT_m \ldots\ldots \text{(8a)}$$

$$Maintenance\ Cost\ of\ Query\ Q_2, MC_2 = MT_1 + MT_2 + MT_3 + \ldots + MT_m \ldots\ldots \text{(8b)}$$

$$Maintenance\ Cost\ of\ Query\ Q_3, MC_3 = MT_1 + MT_2 + MT_3 + \ldots + MT_m \ldots\ldots \text{(8c)}$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$Maintenance\ Cost\ of\ Query\ Q_n, MC_n = MT_1 + MT_2 + MT_3 + \ldots + MT_m \ldots\ldots \text{(8n)}$$

Total query cost: Now, the total cost of the query and the associated maintenance cost can be calculated by summing up the query processing cost and the view maintenance cost.

$$\text{Total}\ Cost\ of\ Query\ Q_1, TC_1 = QP_1 + MC_1 \ldots\ldots\ldots\ldots \text{(9a)}$$

$$\text{Total}\ Cost\ of\ Query\ Q_2, TC_2 = QP_2 + MC_2 \ldots\ldots\ldots\ldots \text{(9b)}$$

$$\text{Total}\ Cost\ of\ Query\ Q_3, TC_3 = QP_3 + MC_3 \ldots\ldots\ldots\ldots \text{(9c)}$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$\text{Total}\ Cost\ of\ Query\ Q_n, TC_n = QP_n + MC_n \ldots\ldots\ldots\ldots \text{(9n)}$$

$$\text{Minimum of query total cost, } Min\,(MC) = \frac{TC_1 + TC_2 + TC_3 + \ldots + TC_n}{n} \ldots\ldots \text{(10)}$$

$$= \frac{\sum_{i=1}^{n} TC_i}{n} \ldots\ldots\ldots\ldots \text{(10a)}$$

View selection algorithm: The following algorithm *DynamicViewMaterializationSelection* in Fig. 3.6 selects the views dynamically for materialization to improve the query performance based on the query processing cost and the view maintenance cost. The algorithm first calculates the query processing cost and the maintenance cost associated with the queries and the total cost for materialized view maintenance. Then it finds the minimum of the total cost. Finally, the algorithm selects the queries with higher total processing cost than the minimum total processing cost, *Min* (*TC*). Here, the *Min* (*TC*) acts as the dynamic threshold level for the total processing cost.

Algorithm DynamicViewMaterializationSelection

{

Given:

Input: A set of queries or views;

Output: Query cost ($n \times 4$) table matrix;

Begin

For $i = 1$ to total number of queries (n) {

Assign:

Q_i = *getNextQuery* ( );

Calculate the query processing cost (QP_i);

Calculate the view maintenance cost (MC_i);

TC_i = *Calculate the total cost* ($QP_i + MC_i$)

} ***End For***;

Find the minimum of the total cost Min (TC);

For $i = 1$ to total number of queries (n) {

If $TC_i > Min$ (TC) ***Then*** {

Select Q_i *for materialization;*

Insertlist (Q_i, QP_i, MC_i, TC_i);

} ***End If;***

} ***End For***;

End;

}

Fig. 3.6 Algorithm for dynamic view selection to materialize

The output of the dynamic view selection algorithm is like in the following Table 3.6.

Table 3.6 Dynamically selected views for materialization

Query	Query Cost	Maintenance Cost	Total Cost
Q_1	QP_1	MC_1	TC_1
Q_3	QP_3	MC_3	TC_2
Q_7	QP_7	MC_7	TC_7
Q_{10}	QP_{10}	MC_{10}	TC_{10}

3.5.2 Dynamic removal of old materialized views

To improve the query performance, the virtual views are materialized and the materialized views need to be periodically updated with the changes to the base tables. Materializing a view not only incurs maintenance cost but also it occupies a large storage space. So it needs to check periodically that the materialized views are useful or not and whether the materialize views are queried frequently. If the materialized views are not queried frequently or the access frequencies to the materialized views are much less, then those materialized views can be removed from the database in order to save the maintenance time and also to free the storage spaces for the new view materialization. Here, we have designed a dynamic model to remove the old materialized views based on the access frequencies. To develop the algorithm for dynamic removal of old materialized views first we need to fill up the following materialized views-access frequencies ($n \times 2$) matrix in Table 3.7.

Table 3.7 Materialized views-access frequencies matrix

Materialized Views (MV_n)	Access Frequencies (f_n)
MV_1	f_1
MV_2	f_2
MV_3	f_3
. . .	. . .
MV_n	f_n

The access frequencies of the materialized views can be computed as like the query access frequency finding algorithm *FindAccessFrequency* in Fig. 3.3.

After filling up the table 3.7, the minimum of the access frequencies of the materialized views is calculated using the following equation:

Minimum of the materialized view access frequency, $Min(f) = \frac{f_1 + f_2 + f_3 + ... + f_n}{n}$ (11)

Finally, the materialized views with low access frequencies below the minimum access frequencies of the materialized views, *Min* (f) (*Min* (f) is the dynamic threshold level) are selected for removal from the database. The dynamic old materialized view removal process is depicted through the algorithm *DynamicMaterializedViewRemoval* in Fig. 3.7.

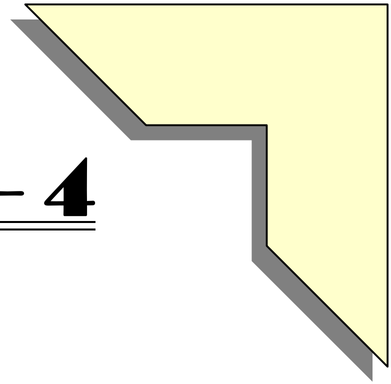
Algorithm DynamicMaterializedViewRemoval
{
Given:
Input: A set of existing materialized views;
Output: Materialized view ($n \times 1$) table matrix;
Begin
For i = 1 to total number of materialized views (n) {
Assign:
MV_i = *getNextMaterializedView* ( );
Calculate the access frequencies of the materialized views (f_i);
} ***End For***;
Find the minimum of materialized view access frequency Min (f);
For i = 1 to total number of materialized views (n) {
If $f_i < Min(f)$ ***Then*** {
Select MV_i *for removal;*
Insertlist (MVi);
Remove the materialized view MV_i *from the database;*
} ***End If;***
} ***End For***;
End;
}

Fig. 3.7 Algorithm for dynamic removal of old materialized views

The output of the dynamic materialized view removal algorithm selects the materialized views to remove the database is like in the following Table 3.8.

Table 3.8 Dynamically selected materialized views to remove

Materialized Views
MV_2
MV_4
MV_8
MV_9



CHAPTER - 4

RESULTS and DISCUSSIONS

4.1 Experimental Background

4.2 Experiments Results on View Materialization Determination Methodology

4.2.1 Varying view selectivity

4.2.2 Varying view structural complexity

4.2.3 Varying database size

4.3 Experiments Results on Dynamic Selection of Views and Removal of Materialized views

4.3.1 Dynamic selection of views

4.3.2 Dynamic removal of materialized views

Chapter 4

RESULTS AND DISCUSSIONS

Following the theoretical design and performance evaluation of view materialization methodology and dynamic model for selecting views dynamically for materialization and dynamic removal of old materialized views in Chapter 3, experimental performance results have been carried out in this chapter.

4.1 Experimental Background

The experiments reported in this chapter have been carried out under the following hardware and software environment:

Hardware: The hardware used in the experiments in a PC with the following specifications:

Processor: Intel(R) Core(TM) 2 Duo, 2.00 GHz;
L2 Cache: 2 MB;
RAM: 3 GB and
Hard disk: 150 GB (where the systems software and 6 GB of paging space reside).

Software: The system software used in the experiments is Microsoft Windows XP Professional Service Pack 3.

Database: The database used in the experiments is Oracle 11*g* Release 1 (11.1.0.6.0) Enterprise Edition.

Database Schema: The popular *sales history* and *order entry* database schemas have been used in the experiments by generating appropriate set of data to meet the research experiment goal. The *sales history* schema contains the tables: *CHANNELS*, *COUNTRIES*, *COSTS*, *CUSTOMERS*, *PRODUCTS*, *PROMOTIONS*, *TIMES* and *SALES*. The *order entry* schema contains the tables: *CUSTOMERS*, *COUNTRIES*, *INVENTORIES*, *WAREHOUSES*, *ORDERS*, *ORDER_ITEMS*, *PRODUCT_INFORMATIONS* and *PRODUCT_DESCRIPTIONS*. The relationship of the *sales history* schema tables is shown in Fig. 4.1. The detail table structures are given in Appendix A.

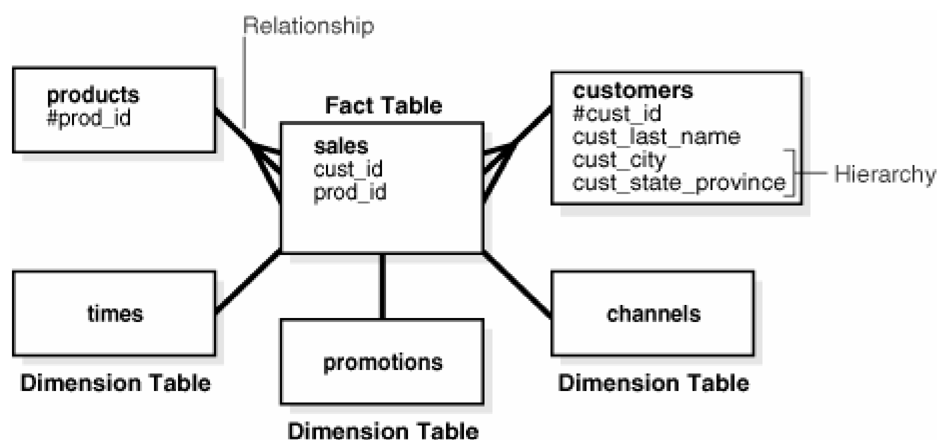


Fig. 4.1 Relationship of the sales history schema tables

The *sales history* schema is a *star schema* representation of the data warehouse environment. The *star schema* representation resembles a star with points radiating from a center. Center of the star consists of one or more fact tables and the points of the star are the dimension tables. The fact tables are the large tables in the data warehouse that store the business measurements. These typically contain facts and foreign keys to the dimension tables. The fact tables represent data, usually numeric and additive, that can be analyzed and examined. Examples of facts tables in the *sales history* schema are *SALES* and *COSTS*. The dimension tables are known as lookup or reference tables. It contains the relatively static data in the data warehouse. Dimension tables store the information normally used to contain in the queries and the information is usually textual and descriptive and can use them as the row headers of the result set. Examples of dimension tables in *sales history* schema are *CUSTOMERS* and *PRODUCTS*. Only one join establishes the relationship between the fact table and any one of the dimension tables. In Fig. 4.1, a relationship between the sales information in the fact table and the dimension tables products and customers enforces the business rules in the database.

Database parameters setting: Normally when a database is installed the default settings of the different database parameters meet the user's operations. But all of the database features are not enabled at the installation time or some parameters need to change to meet the experiment like to use the query rewrite feature for using a materialized view to answer a fully text matched or partially text matched query. The database parameters that have been used for the experiments are listed in Table 4.1.

Table 4.1 List of database parameters and assigned values

Database Parameters	(=) Assigned Values
memory_target	= 314572800
max_memory_target	= 536870912
query_rewrite_enabled	= TRUE
query_rewrite_integrity	= ENFORCED
job_queue_processes	= 1000
optimizer_mode	= FIRST_ROWS
compatible	= 11.1.0.0.0
db_file_multiblock_read_count	= 128
Open_cursors	= 300
processes	= 150

4.2 Experiments Results on View Materialization Determination Methodology

The methodology in the previous chapter evaluates the incremental materialized view maintenance performance and determines the circumstances in which a virtual view or a query can be selected as profitable to materialized for improving the query performance considering the incremental materialized view maintenance cost. The incremental performance and the circumstances being beneficial have been evaluated based on the three issues namely - view selectivity, view structural complexity and database size. Each of the issues then evaluated by considering the view maintenance cost comparison, query answering cost comparison and relative cost of query answering and view maintenance cost. In the subsequent sections, experimental results have shown for different view selectivities, complexities and database size. The following databases in Table 4.2 have been used in the experiments:

Table 4.2 Database used in the experiments

Database Size					
Database	db_1	db_2	db_3	db_4	db_5
Records	730400	1460800	2191200	2921600	3652000
Size (MB)	41.69	83.38	125.12	166.82	208.53

4.2.1 Varying view selectivity

Different view selectivities arise as a result of a view containing predicates that filter the input data. A set of five views and a set of five materialized views with different selectivities are derived by instantiating the templates in Fig. 4.2. Each virtual view and materialized views differ in the predicate ϕ_i in the where clause. Each predicate is obtained by instantiating the template ϕ_i with *cs.cust_id* <= k_i where $k_1 = 400$ to $k_5 = 2000$ in increments of 400, thereby yielding five different views and materialized views with selectivities from 0.2 to 1.0 in increments of 0.2. In the customer table, there are 2000 customers and their *cust_id* is sequentially set from 1 to 2000, so it is possible to control the selectivity of the template. For the experiment of the selectivity issue, the database db_5 in Table 4.2 has been used by varying the *cust_id* in where condition. The detail description of the queries or views is given in Appendix B.

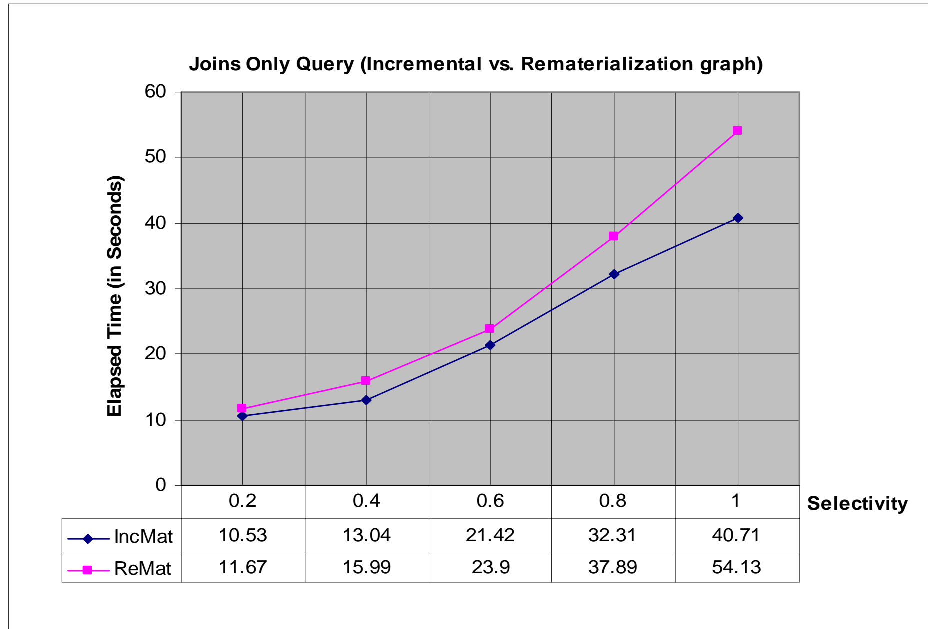
```
CREATE VIEW <join_view_i> AS
SELECT  cn.country_name country, p.prod_name prod, t.calendar_year year,
s.amount_sold sale FROM sales s, times t, customers cs, countries cn, products p
WHERE s.time_id = t.time_id AND s.prod_id = p.prod_id AND s.cust_id = cs.cust_id
AND cs.country_id = cn.country_id AND cs.cust_id <= φ_i;

CREATE VIEW <aggregate_view_i> AS
SELECT ch.channel_desc, t.calendar_month_desc, cn.country_iso_code,
p.prod_name, SUM(s.amount_sold) SALES$, count(s.amount_sold) total FROM sales
s, customers c, times t, channels ch, countries cn, products p WHERE s.time_id=
t.time_id AND s.cust_id=c.cust_id AND s.channel_id= ch.channel_id AND
s.prod_id=p.prod_id AND c.country_id=cn.country_id AND c.cust_id <= φ_i GROUP
BY ch.channel_desc, t.calendar_month_desc, cn.country_iso_code, p.prod_name;
```

Fig. 4.2 Template used to derive view selectivities

The experiments on selectivity test explore the impact of view predicate selectivity on the performance of incremental materialized view maintenance and determine the situations of view materialization profitable. The experiment has been broken down into three parts – view maintenance, query answering and relative costs; each corresponding to the performance measurement factors under scrutiny.

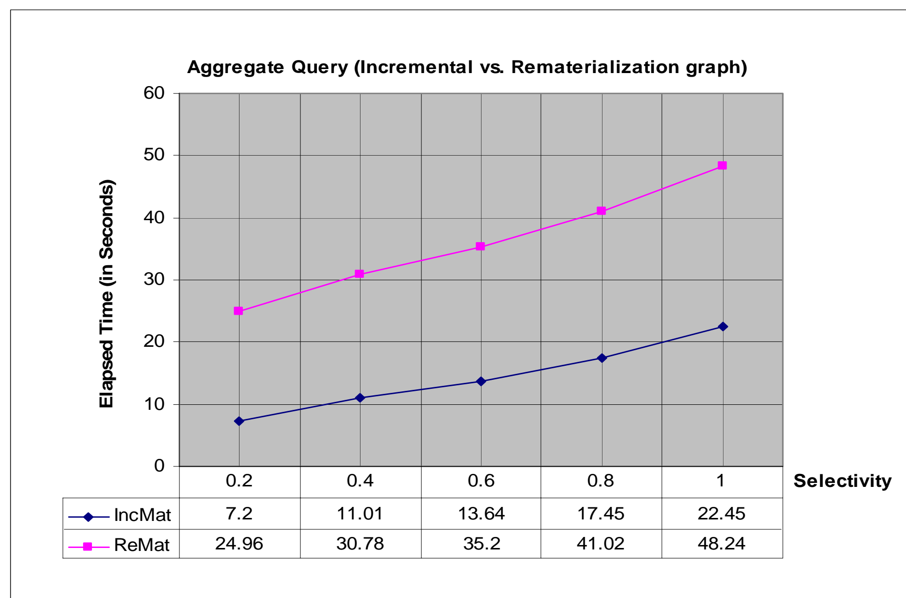
View maintenance: The elapsed times of the incremental view maintenance and rematerializing a view have been measured and the results are plotted in the Fig. 4.3 (a) and Fig. 4.3 (b) for two types of queries: joins only queries and aggregate queries.



	0.2	0.4	0.6	0.8	1
IncMat	10.53	13.04	21.42	32.31	40.71
ReMat	11.67	15.99	23.9	37.89	54.13

Fig. 4.3 (a) View maintenance costs for joins only query

For the joins only query from Fig. 4.3 (a), the incremental maintenance cost and rematerializing cost are almost same at a selectivity of 0.2, but as selectivity increases cost increases for both cases. At selectivity 0.8, the incremental cost is 32.31 seconds whereas for rematerializing the cost is 37.89 seconds.



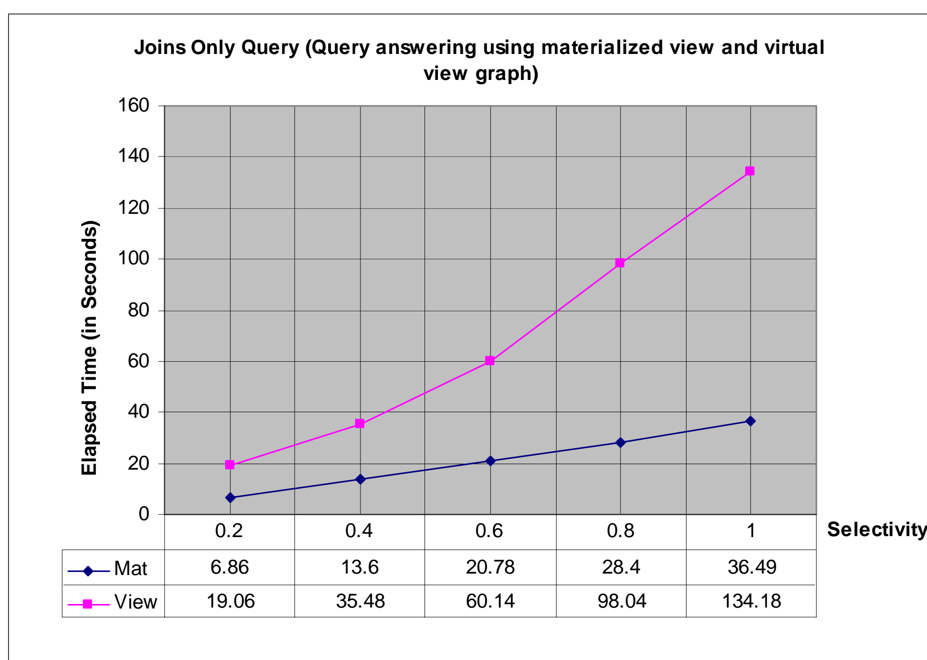
	0.2	0.4	0.6	0.8	1
IncMat	7.2	11.01	13.64	17.45	22.45
ReMat	24.96	30.78	35.2	41.02	48.24

Fig. 4.3 (b) View maintenance costs for aggregate query

For the aggregate query from Fig. 4.3 (b), the incremental maintenance cost and rematerializing cost are 7.2 and 24.96 at a selectivity of 0.2 but as selectivity increases cost increases for both cases. At selectivity 0.6, the incremental cost is 13.64 seconds whereas for rematerializing the cost is 35.2 seconds.

So, the cost of materialized view maintenance increases with an increase in view selectivity in both cases (incremental propagation and rematerializing) but incremental maintenance performs better that rematerializing.

Query answering: The Fig. 4.4 (a), (b), (c) and (d) show the cost of answering a view and using rewrite in comparison with the cost of answering a materialized view.



	0.2	0.4	0.6	0.8	1
Mat	6.86	13.6	20.78	28.4	36.49
View	19.06	35.48	60.14	98.04	134.18

Fig. 4.4 (a) Query answering costs of view for joins only query

For the joins only query, from Fig. 4.4 (a), the query answering cost for materialized view and virtual view are 6.86 and 19.06 at a selectivity of 0.2, but after that the difference is increases with in increase in selectivity. At a selectivity of 1.0, the query answering costs are 36.49 and 134.18 respectively for materialized view and virtual view.

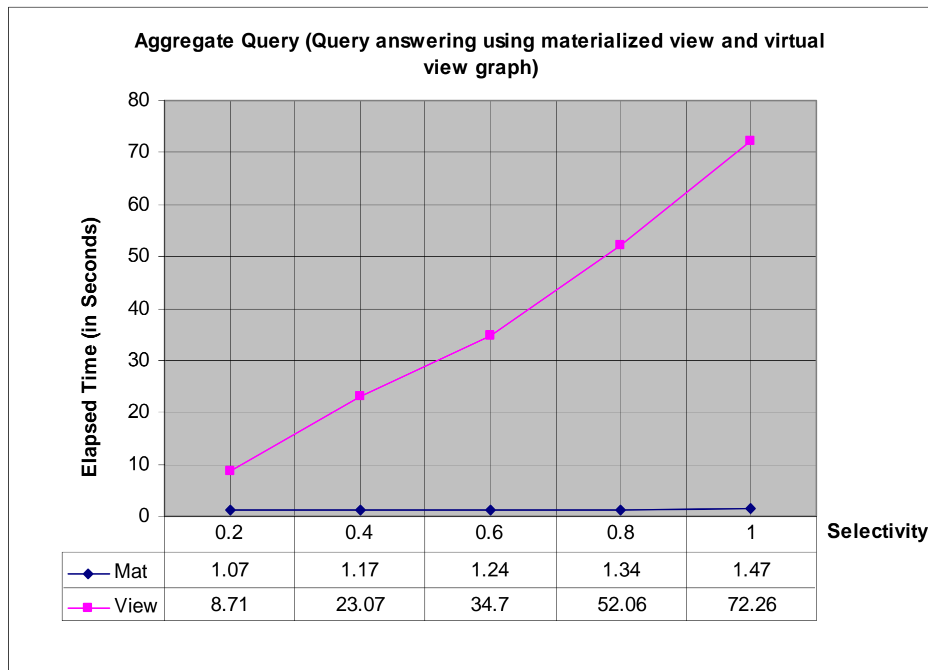


Fig. 4.4 (b) Query answering costs of view for aggregate query

For the aggregate query, from Fig. 4.4 (b), the query answering cost for materialized view and virtual view are 1.07 and 8.71 at a selectivity of 0.2, but after that the difference is increases with in increase in selectivity. At a selectivity of 0.8, the query answering costs are 1.34 and 52.06 respectively for materialized view and virtual view.

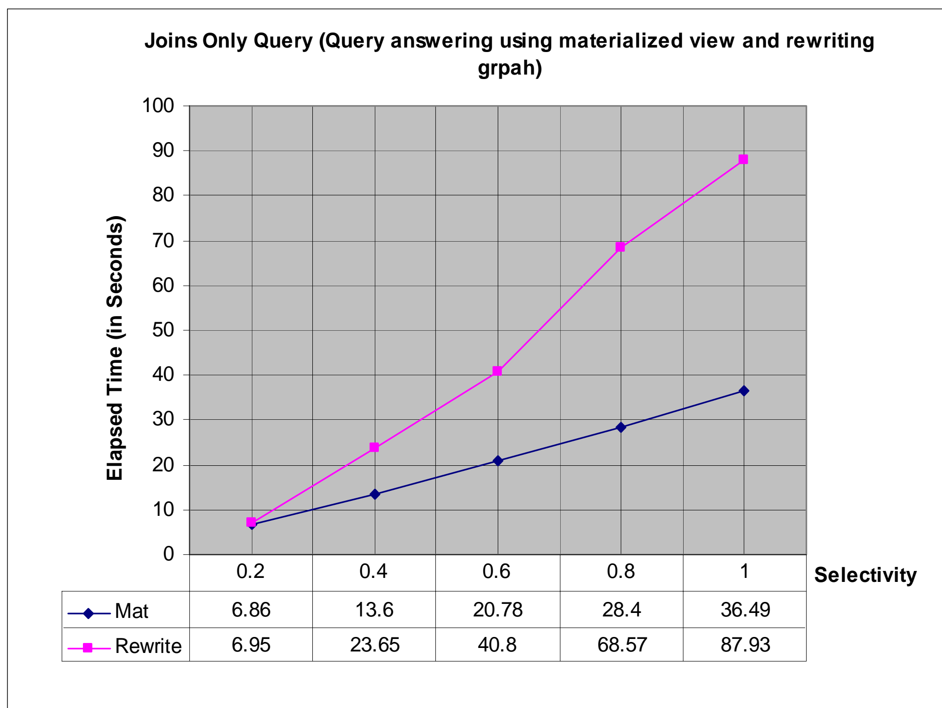


Fig. 4.4 (c) Query answering costs using rewrite for joins only query

For the joins only query, from Fig. 4.4 (c), the query answering cost for materialized view and using rewrite are 6.86 and 6.95 at a selectivity of 0.2, but after that the difference is increases with in increase in selectivity. At a selectivity of 1.0, the query answering costs are 36.49 and 87.93 respectively for materialized view and using rewrite.

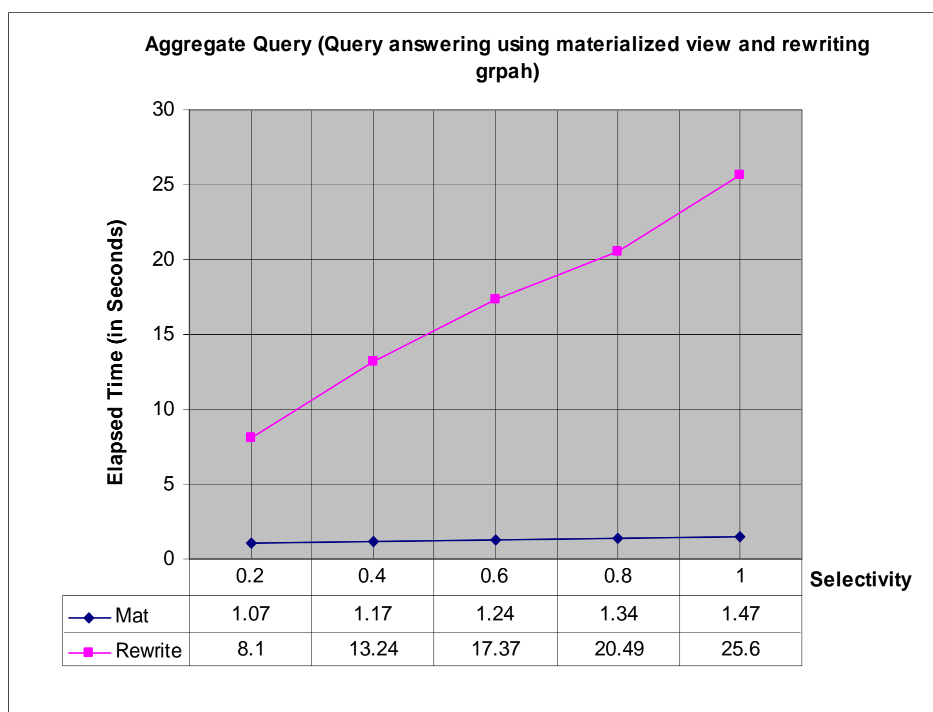


Fig. 4.4 (d) Query answering costs using rewrite for aggregate query

For the aggregate query, from Fig. 4.4 (d), the query answering cost for materialized view and using rewrite are 1.07 and 8.1 at a selectivity of 0.2, but after that the difference is increases with in increase in selectivity. At a selectivity of 0.8, the query answering costs are 1.34 and 20.49 respectively for materialized view and using rewrite.

So, it is seen that the query answering directly from the materialized views outperforms to that of the answering a virtual view and answering query using query rewrite. It is also noted that the query answering using rewrite is better than answering a virtual view. This is because that the query rewrite engine rewrites a query based on full text match or partial match to query from the materialized view rather than the base tables. Rewriting a query takes more time to query than the direct query from the materialized view because it takes time to search the materialized view text to match to its SQL statements and then queries from the materialized views if it is matched.

Relative costs: The ratio of the cost of answering a virtual view to the cost of incremental propagation, ratio of the cost of answering query rewrite to the incremental propagation and the ratio of the cost of answering a virtual view to the cost of answering a query rewrite measure how much it is profitable to select a view to materialize for improving the query performance. Fig. 4.5 (a), (b), (c), (d), (e) and (f) show the relative costs of joins only and aggregate queries.

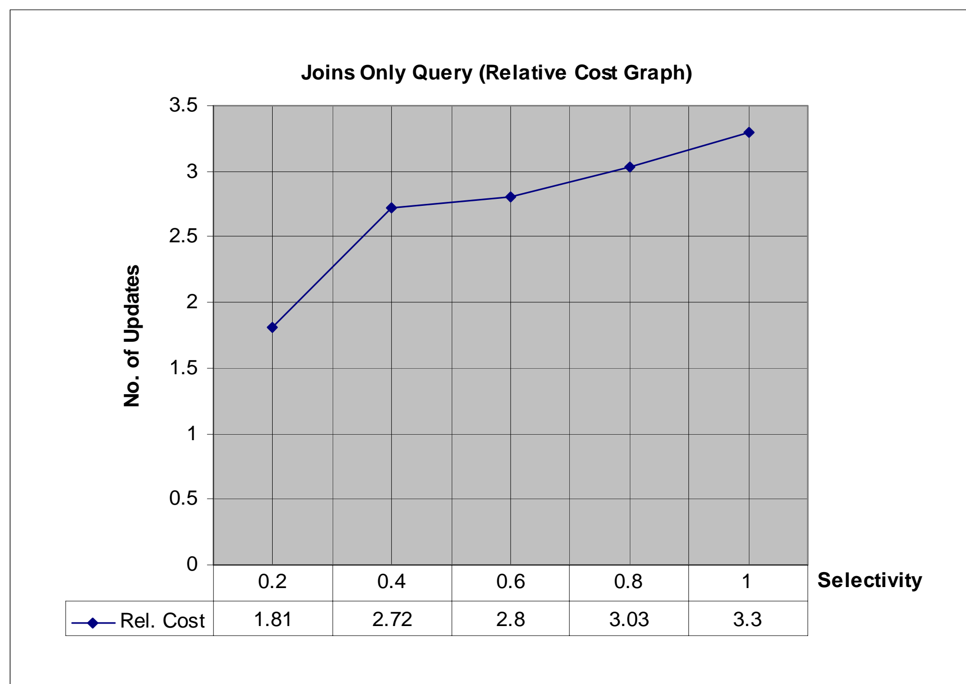


Fig. 4.5 (a) Relative cost of answering a view to the incremental propagation time for joins only query

For the joins only query, from Fig. 4.5 (a), the relative cost of answering a virtual view to the incremental maintenance of the materialized view is 1.81 at a selectivity of 0.2 and it means that with the cost of answering a virtual view, two updates can be propagated to the materialized view. Similarly, at a selectivity of 0.8, the relative cost is 3.30 means three updates can be propagated with the cost of answering a virtual view.

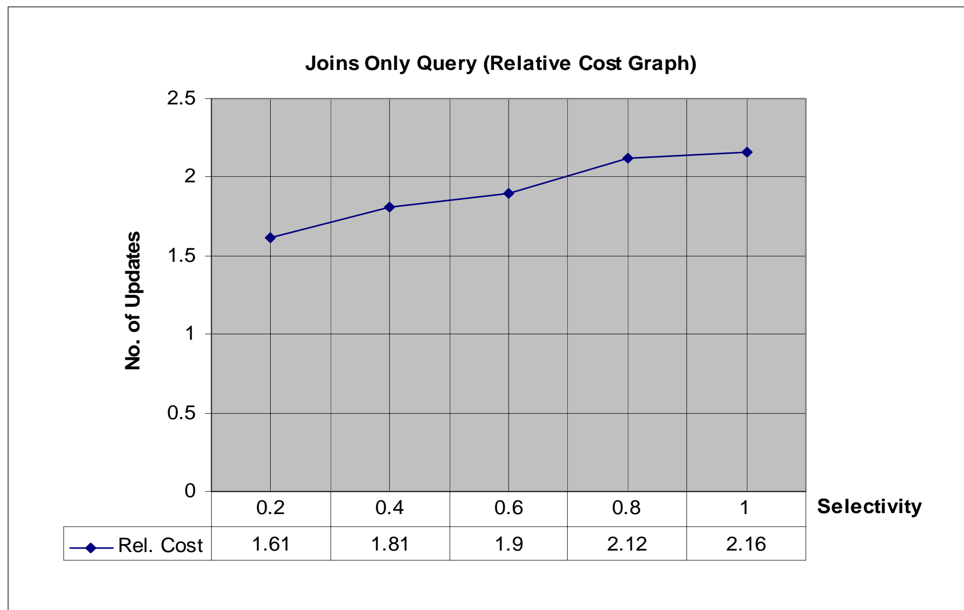


Fig. 4.5 (b) Relative cost of answering a query using rewrite to the incremental propagation time for joins only query

For the joins only query, from Fig. 4.5 (b), the relative cost of answering using rewrite to the incremental maintenance of the materialized view is 1.61 at a selectivity of 0.2 and it means that with the cost of query answering using rewrite, two updates can be propagated to the materialized view. Similarly, at a selectivity of 0.8, the relative cost is 2.12 means two updates can be propagated with the cost of query answering using rewrite.

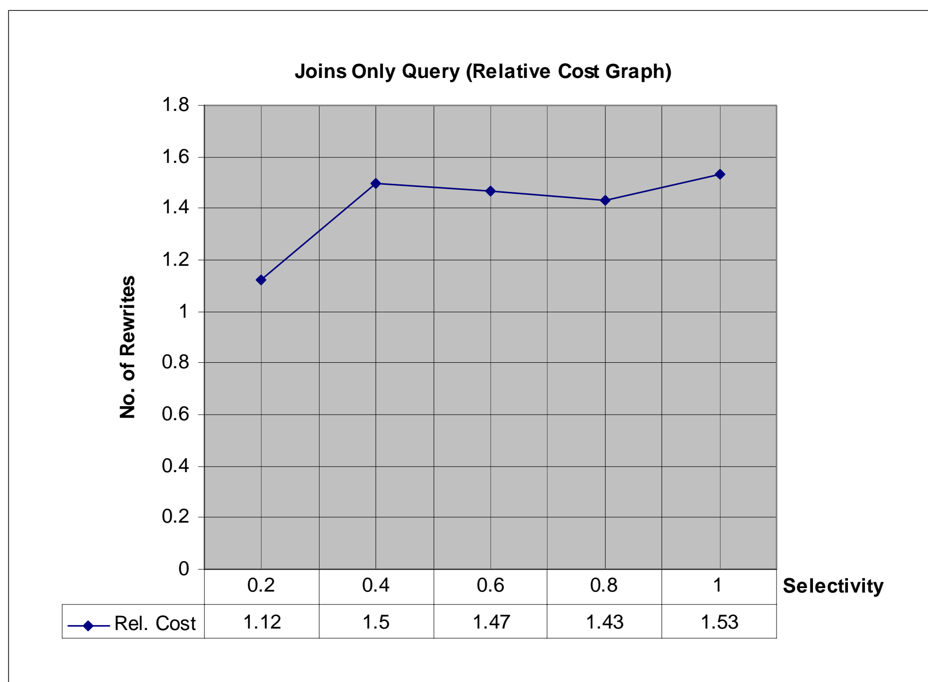


Fig. 4.5 (c) Relative cost of answering a view to that of using rewrite for joins only query

For the joins only query, from Fig. 4.5 (c), the relative cost of answering a virtual view to that of using rewrite is 1.12 at a selectivity of 0.2 and it means that with the cost of answering a virtual view one rewrite can possible. Similarly, at a selectivity of 1.0, two rewrites can possible for the cost of answering a virtual view.

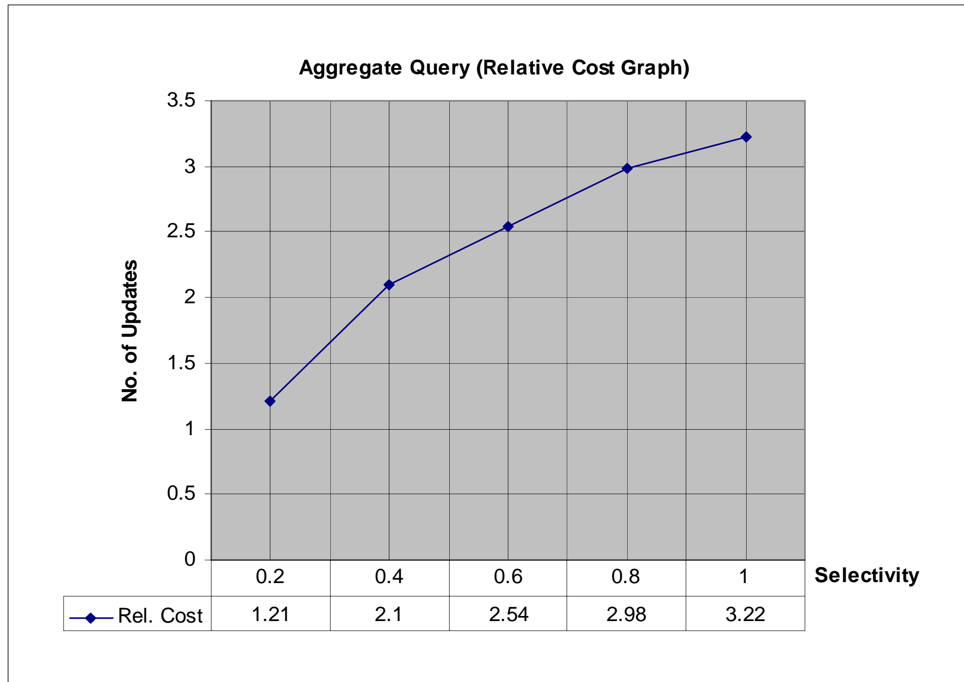


Fig. 4.5 (d) Relative cost of answering a view to the incremental propagation time for aggregate query

For the aggregate query, from Fig. 4.5 (d), the relative cost of answering a virtual view to the incremental maintenance of the materialized view is 1.21 and it means that with the cost of answering a virtual view, one update can be propagated to the materialized view at a selectivity of 0.2. Similarly, at a selectivity of 0.8, the relative cost is 2.98 means three updates can be propagated with the cost of answering a virtual view.

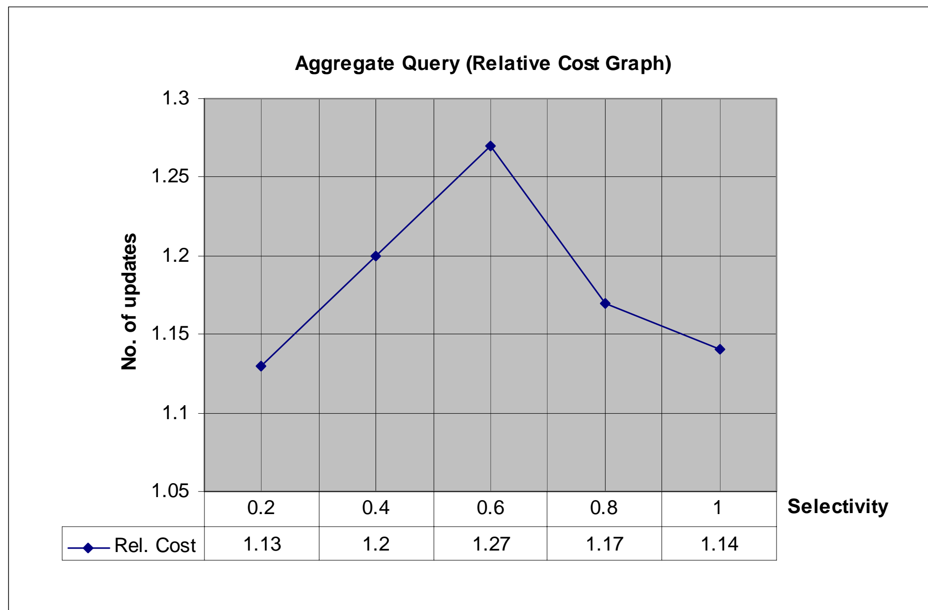


Fig. 4.5 (e) Relative cost of answering a query using rewrite to the incremental propagation time for aggregate query

For the aggregate query, from Fig. 4.5 (e), the relative cost of answering using rewrite to the incremental maintenance of the materialized view is 1.13 at a selectivity of 0.2 and it means that with the cost of query answering using rewrite, one update can be propagated to the materialized view. Similarly, at a selectivity of 0.8, the relative cost is 1.17 means one update can be propagated with the cost of answering using rewrite.

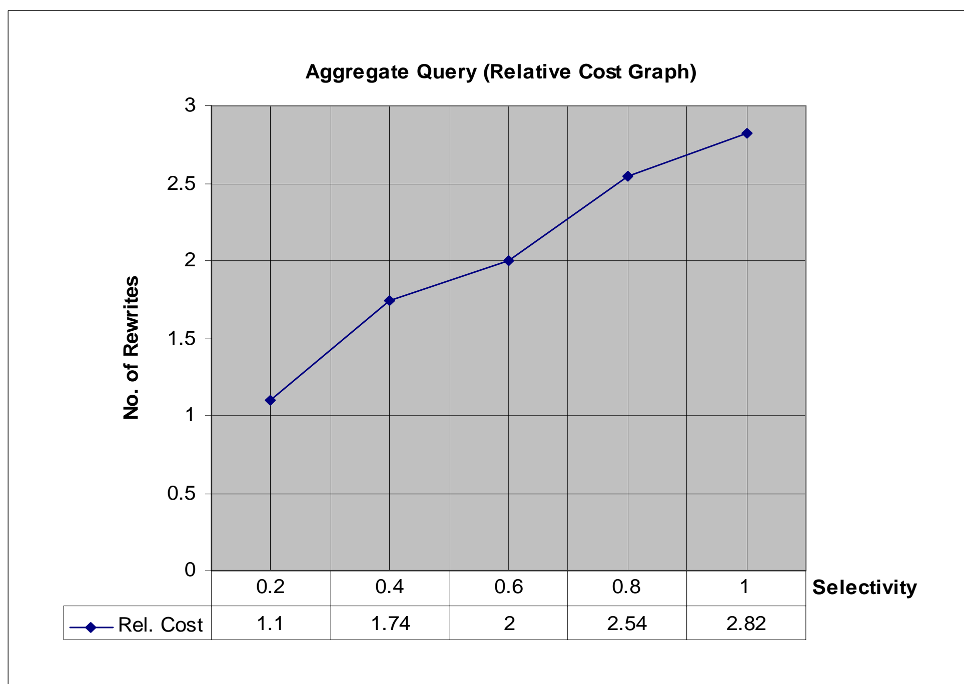


Fig. 4.5 (f) Relative cost of answering a view to that of using rewrite for aggregate query

For the aggregate query, from Fig. 4.5 (f), the relative cost of answering a virtual view to that of using rewrite is 1.1 at a selectivity of 0.2 and it means that with the cost of answering a virtual view one rewrite can possible. Similarly, at a selectivity of 0.8, three rewrites can possible for the cost of answering a virtual view.

So, from the relative costs plots, it is observed that with the increase in selectivity for the cost of answering a virtual view or answering a query using rewrite, more number of updates can be propagated to the materialized views. It is also observed that if a query is materialized then using query rewrite, answering a query time is saved at a considerable amount in comparison with answering a virtual view.

In summary, the incremental materialized view maintenance is profitable over rematerializing a view each time a change is made to the base tables and hence we can infer that with the increase in view selectivity, a view is more cost effective for materialization when the goal is to optimize query execution.

4.2.2 Varying view structural complexity

A view is structurally complex based on the number of tables involves, number of joining conditions and number and kind of algebraic operators used for aggregating different data. The complex views that have been used for the experiments are given in Appendix A. For the experiment of the complexity issue, the database db_5 in Table 4.2 has been used. This experiment seeks to show how the complexity of the views may affect the performance of incremental materialized view maintenance. Again, the experiment has been divided into three parts – view maintenance, query answering and relative costs; each one corresponding to the performance measurement factors under scrutiny.

View maintenance: The elapsed times of the incremental view maintenance and rematerializing a view have been measured and the results are plotted in the Fig. 4.6 (a) and Fig. 4.6 (b) for two types of queries: joins only queries and aggregate queries.

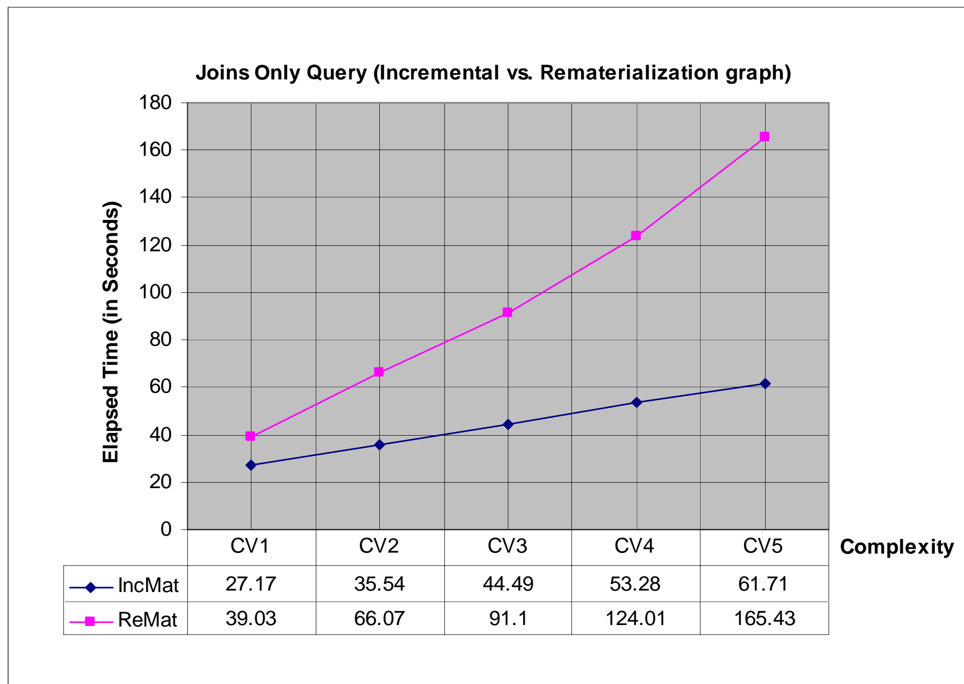


Fig. 4.6 (a) View maintenance costs for joins only query

For the joins only query from Fig. 4.6 (a), the incremental maintenance cost and rematerializing cost are 27.17 and 39.03 respectively at a simple complex view, but as complexity increases cost increases for both cases. At complexity CV_4, the incremental cost is 53.28 seconds whereas for rematerializing the cost is 124.01 seconds.

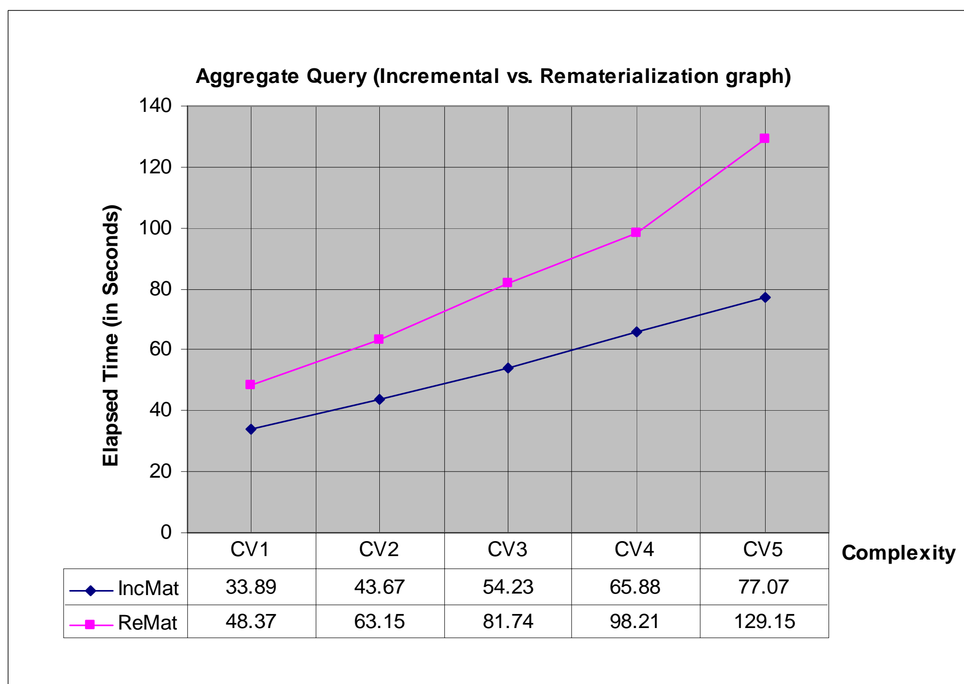
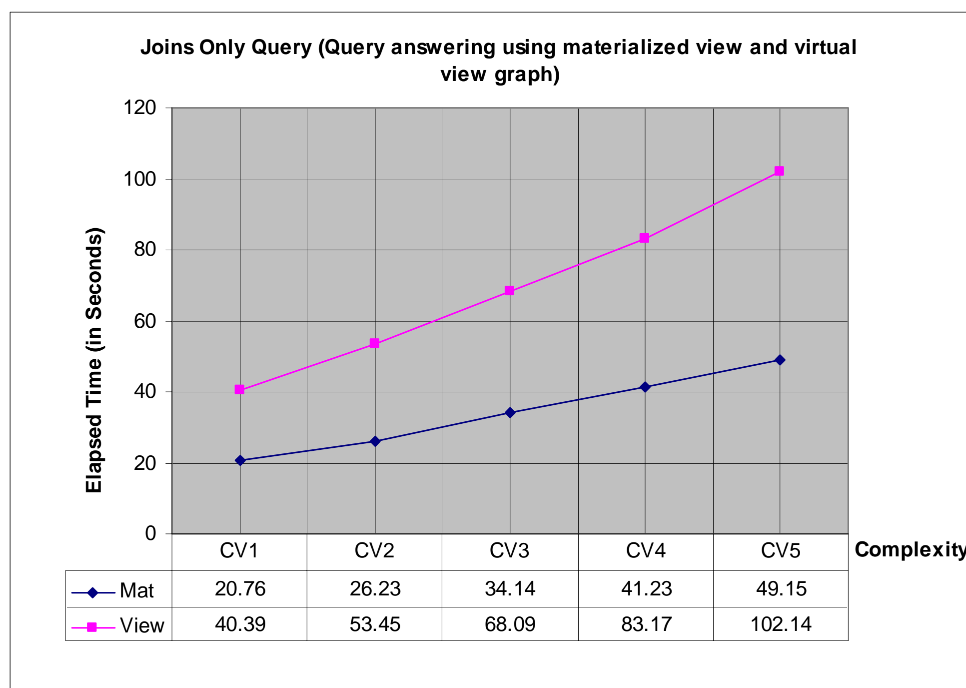


Fig. 4.6 (b) View maintenance costs for aggregate query

For the aggregate query from Fig. 4.6 (b), the incremental maintenance cost and rematerializing cost are 33.89 and 48.37 respectively at complexity CV_1, but as complexity increases cost increases for both cases. At complexity CV_3, the incremental cost is 54.23 seconds whereas for rematerializing the cost is 48.37 seconds.

So, the cost of materialized view maintenance increases with an increase in view complexity in both cases (incremental propagation and rematerializing) but incremental maintenance performs better that rematerializing.

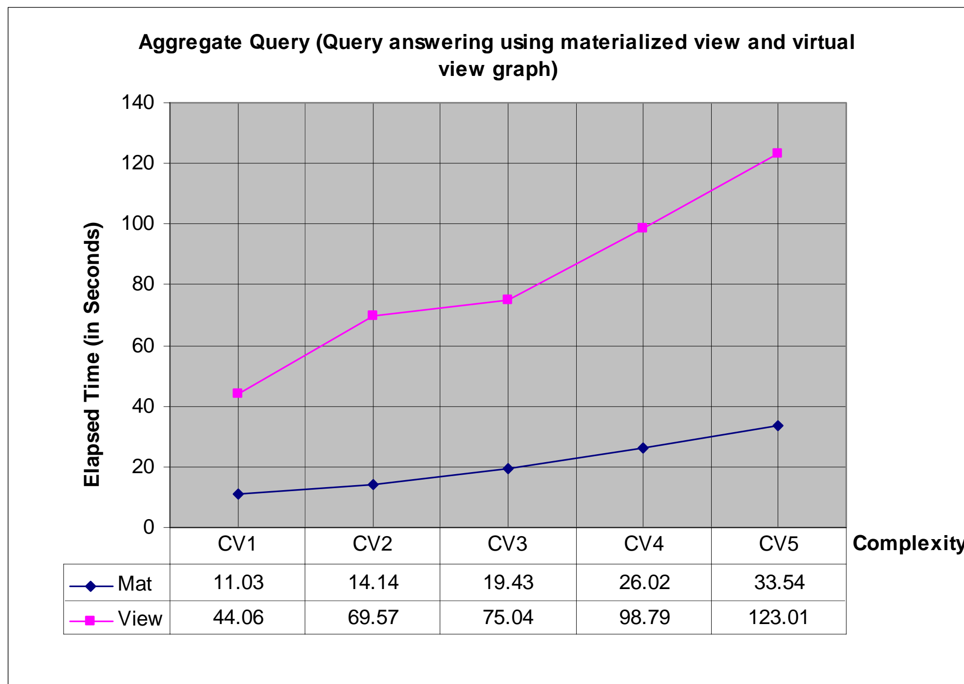
Query answering: The Fig. 4.7 (a), (b), (c) and (d) show the cost of answering a view and using rewrite in comparison with the cost of answering a materialized view.



	CV1	CV2	CV3	CV4	CV5
Mat	20.76	26.23	34.14	41.23	49.15
View	40.39	53.45	68.09	83.17	102.14

Fig. 4.7 (a) Query answering costs of view for joins only query

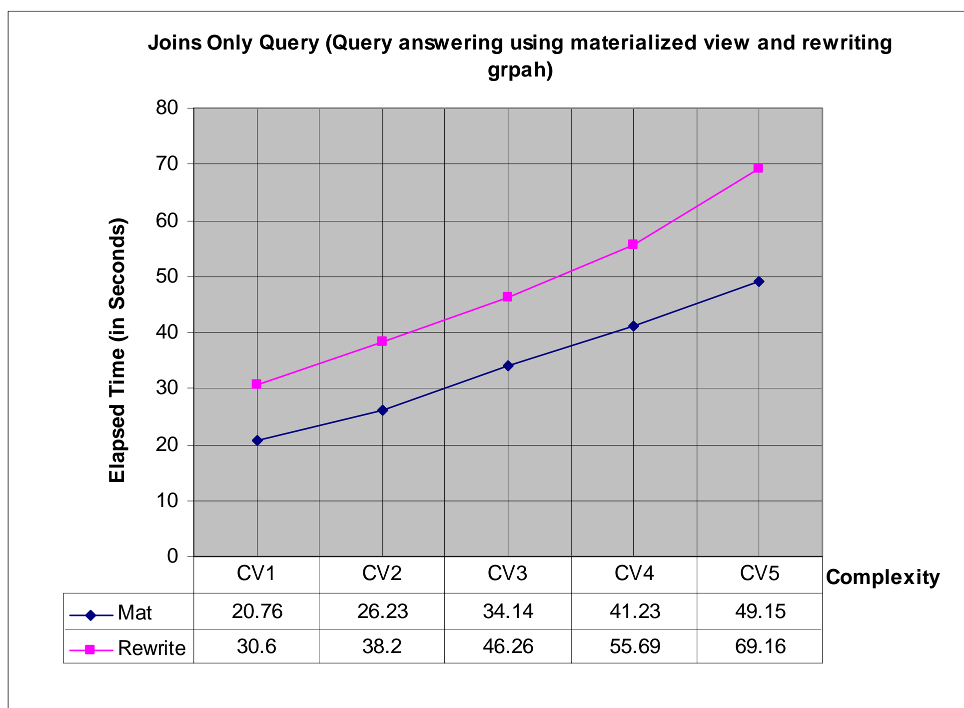
For the joins only query, from Fig. 4.7 (a), the query answering cost for materialized view and virtual view are 20.76 and 40.39 at a complexity of CV_1, but after that the difference is increases with in increase in complexity. At a complexity of CV_5, the query answering costs are 49.15 and 102.14 respectively for materialized view and virtual view.



	CV1	CV2	CV3	CV4	CV5
Mat	11.03	14.14	19.43	26.02	33.54
View	44.06	69.57	75.04	98.79	123.01

Fig. 4.7 (b) Query answering costs of view for aggregate query

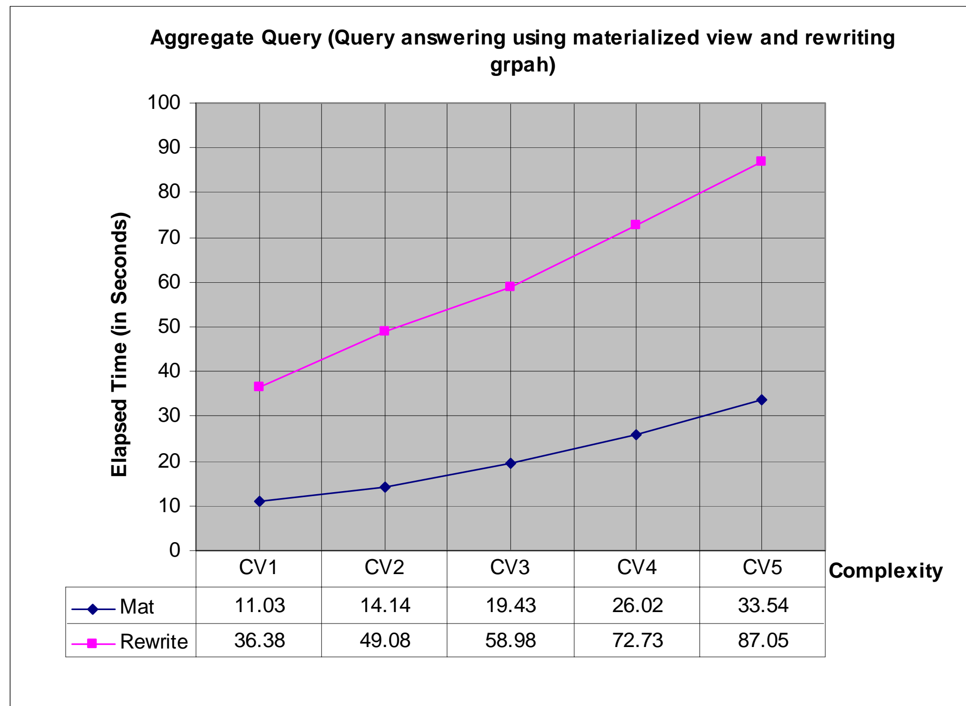
For the aggregate query, from Fig. 4.7 (b), the query answering cost for materialized view and virtual view are 11.03 and 44.06 respectively at a complexity of CV_1 but after that the difference is increases with in increase in complexity. At a complexity of CV_4, the query answering costs are 26.02 and 98.79 respectively for materialized view and virtual view.



	CV1	CV2	CV3	CV4	CV5
Mat	20.76	26.23	34.14	41.23	49.15
Rewrite	30.6	38.2	46.26	55.69	69.16

Fig. 4.7 (c) Query answering costs using rewrite for joins only query

For the joins only query, from Fig. 4.7 (c), the query answering cost for materialized view and using rewrite are 20.76 and 30.6 at a complexity of CV_1, but after that the difference is increases with in increase in complexity. At a complexity of 1.0, the query answering costs are 49.15 and 69.16.93 respectively for materialized view and using rewrite.



	CV1	CV2	CV3	CV4	CV5
Mat	11.03	14.14	19.43	26.02	33.54
Rewrite	36.38	49.08	58.98	72.73	87.05

Fig. 4.7 (d) Query answering costs using rewrite for aggregate query

For the aggregate query, from Fig. 4.7 (d), the query answering cost for materialized view and using rewrite are 11.03 and 36.38 at a complexity of CV_1, but after that the difference is increases with in increase in complexity. At a complexity of CV_4, the query answering costs are 26.02 and 72.73 respectively for materialized view and using rewrite.

So, it is seen that the query answering directly from the materialized views outperforms to that of the answering a virtual view and answering query using query rewrite. It is also noted that the query answering using rewrite is better than answering a virtual view. This is because that the query rewrite engine rewrites a query based on full text match or partial match to query from the materialized view rather than the base tables. Rewriting a query takes more time to query than the direct query from the materialized view because it takes time to search the materialized view text to match to its SQL statements and then queries from the materialized views if it is matched.

Relative costs: The ratio of the cost of answering a virtual view to the cost of incremental propagation, ratio of the cost of answering query rewrite to the incremental propagation and the ratio of the cost of answering a virtual view to the cost of answering a query rewrite measure how much it is profitable to select a view to materialize for improving the query performance. Fig. 4.8 (a), (b), (c), (d), (e) and (f) show the relative costs of joins only and aggregate queries.

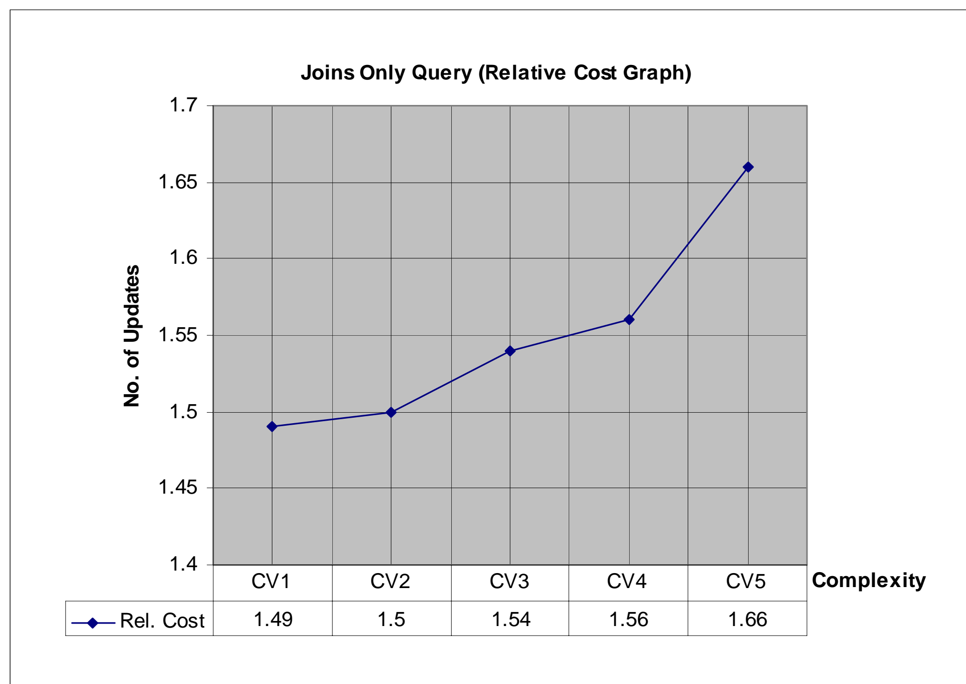


Fig. 4.8 (a) Relative cost of answering a view to the incremental propagation time for joins only query

For the joins only query, from Fig. 4.8 (a), the relative cost of answering a virtual view to the incremental maintenance of the materialized view is 1.49 and it means that with the cost of answering a virtual view, one update can be propagated to the materialized view at a complexity of CV_1. Similarly, at a complexity of CV_4, the relative cost is 1.56 means two updates can be propagated with the cost of answering a virtual view.

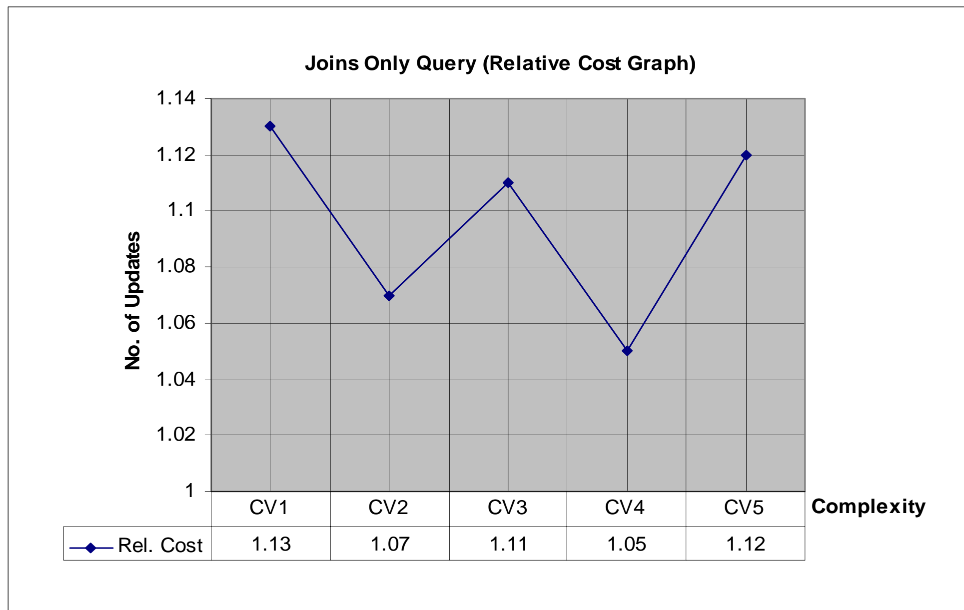


Fig. 4.8 (b) Relative cost of answering a query using rewrite to the incremental propagation time for joins only query

For the joins only query, from Fig. 4.8 (b), the relative cost of answering using rewrite to the incremental maintenance of the materialized view is 1.13 at a complexity of CV_1 and it means that with the cost of answering using rewrite, one update can be propagated to the materialized view. Similarly, at a complexity of CV_4, the relative cost is 1.05 means one update can be propagated with the cost of answering using rewrite.

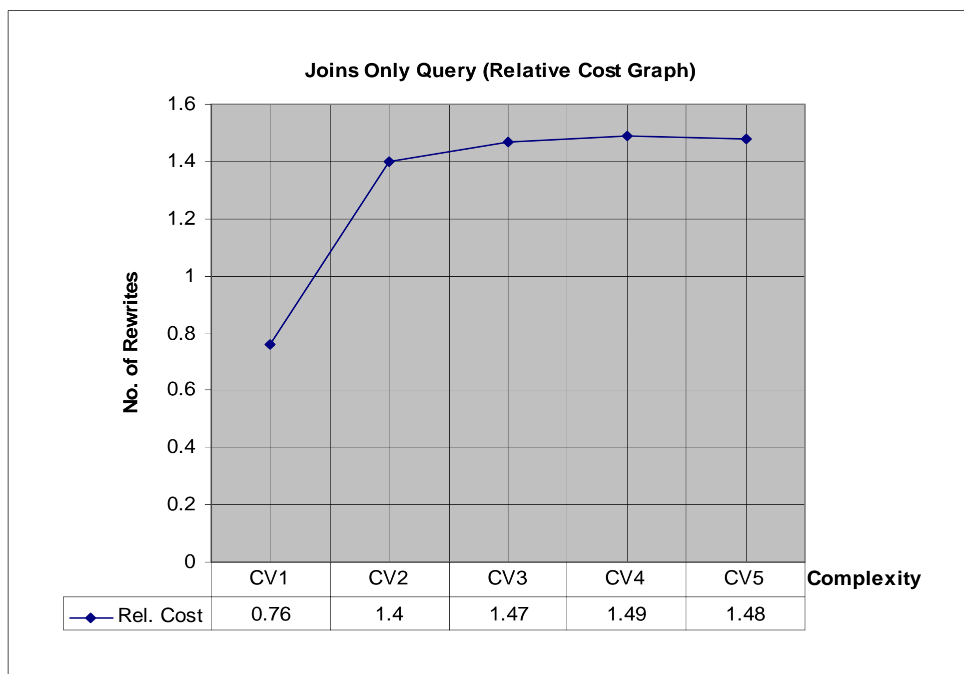


Fig. 4.8 (c) Relative cost of answering a view to that of using rewrite for joins only query

For the joins only query, from Fig. 4.8 (c), the relative cost of answering a virtual view to that of using rewrite is 0.76 at a complexity of CV_1 and it means that with the cost of query answering, one rewrite can possible. Similarly, at a complexity of CV_4, one rewrite can possible for the cost of answering a virtual view.

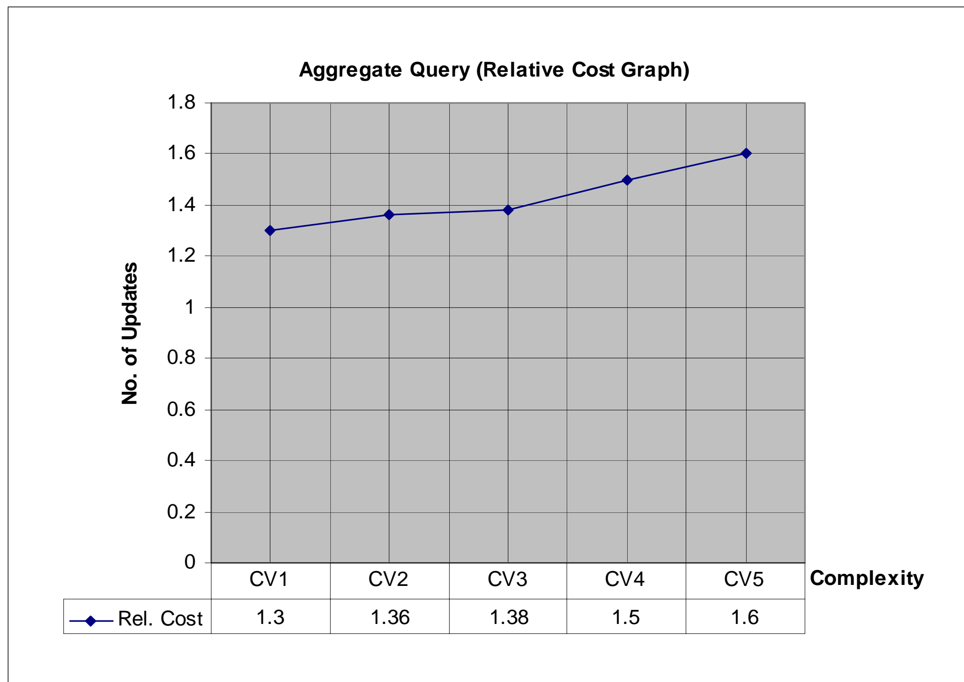


Fig. 4.8 (d) Relative cost of answering a view to the incremental propagation time for aggregate query

For the aggregate query, from Fig. 4.8 (d), the relative cost of answering a virtual view to the incremental maintenance of the materialized view is 1.3 and it means that with the cost of answering a virtual view, one update can be propagated to the materialized view at a complexity of CV_1. Similarly, at a complexity of CV_4, the relative cost is 1.5 means two updates can be propagated with the cost of answering a virtual view.

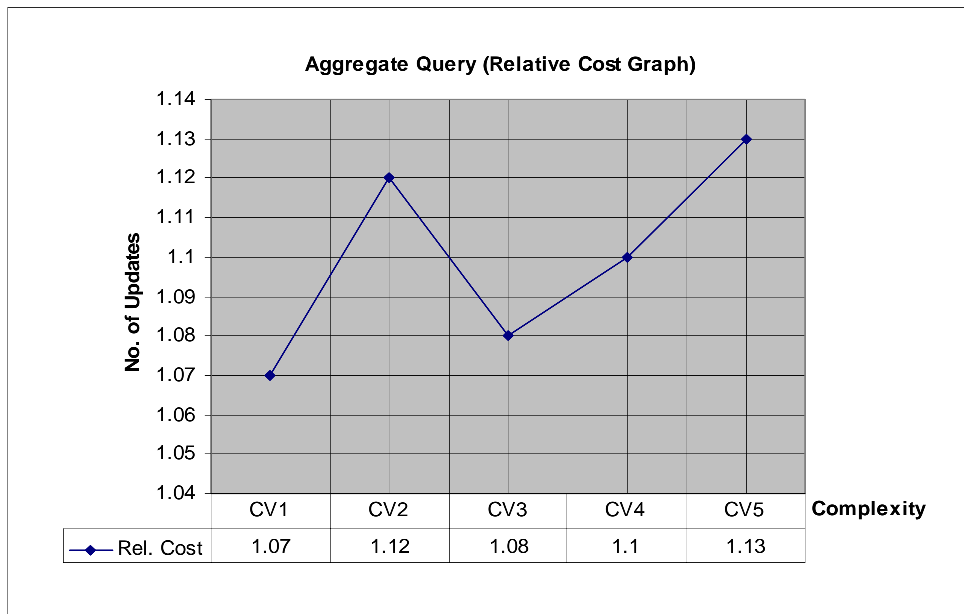


Fig. 4.8 (e) Relative cost of answering a query using rewrite to the incremental propagation time for aggregate query

For the aggregate query, from Fig. 4.8 (e), the relative cost of answering using rewrite to the incremental maintenance of the materialized view is 1.07 at a complexity of CV_1 and it means that with the cost of answering using rewrite, one update can be propagated to the materialized view. Similarly, at a complexity of CV_4, the relative cost is 1.1 means one update can be propagated with the cost of answering using rewrite.

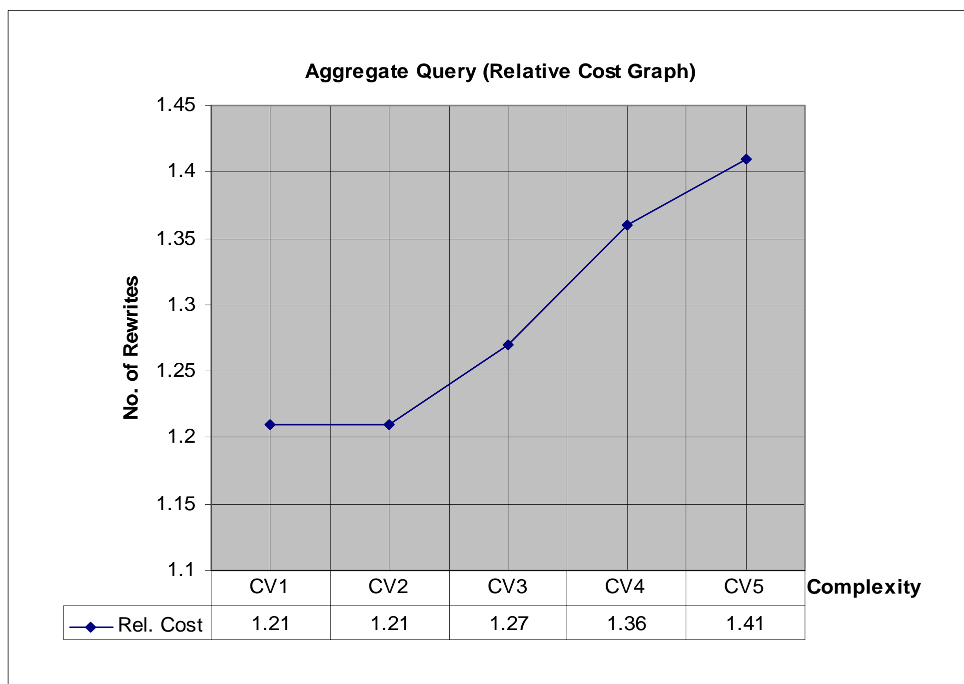


Fig. 4.8 (f) Relative cost of answering a view to that of using rewrite for aggregate query

For the aggregate query, from Fig. 4.8 (f), the relative cost of answering a virtual view to that of using rewrite is 1.21 at a selectivity of 0.2 and it means that with the cost of answering a virtual view one rewrite can possible. Similarly, at a complexity of CV_4, one rewrite can possible for the cost of answering a virtual view.

So, from the relative costs plots, it is observed that with the increase in view complexity for the cost of answering a virtual view or answering a query using rewrite, more number of updates can be propagated to the materialized views. It is also observed that if a query is materialized then using query rewrite, answering a query time is saved at a considerable amount in comparison with answering a virtual view.

In summary, the incremental materialized view maintenance is profitable over rematerializing a view each time a change is made to the base tables and hence we can infer that with the increase in view complexity, a view is more cost effective for materialization when the goal is to optimize query execution.

4.2.3 Varying database size

When the database size varies from small size to large databases, requesting a query causes more time to execute. The views that have been used for the database size experiments are given in Appendix A. For the experiment of the database size issue, the databases db_1 to db_5 in Table 4.2 have been used. This experiment explores the impact of database size on the performance of incremental materialized view maintenance. Once again, the experiment has been divided into three parts – view maintenance, query answering and relative costs; each one corresponding to the performance measurement factors under study.

View maintenance: The elapsed time of the incremental view maintenance and rematerializing a view has been measured and the results are plotted in the Fig. 4.9 (a) and Fig. 4.9 (b) for two types of queries: joins only queries and aggregate queries.

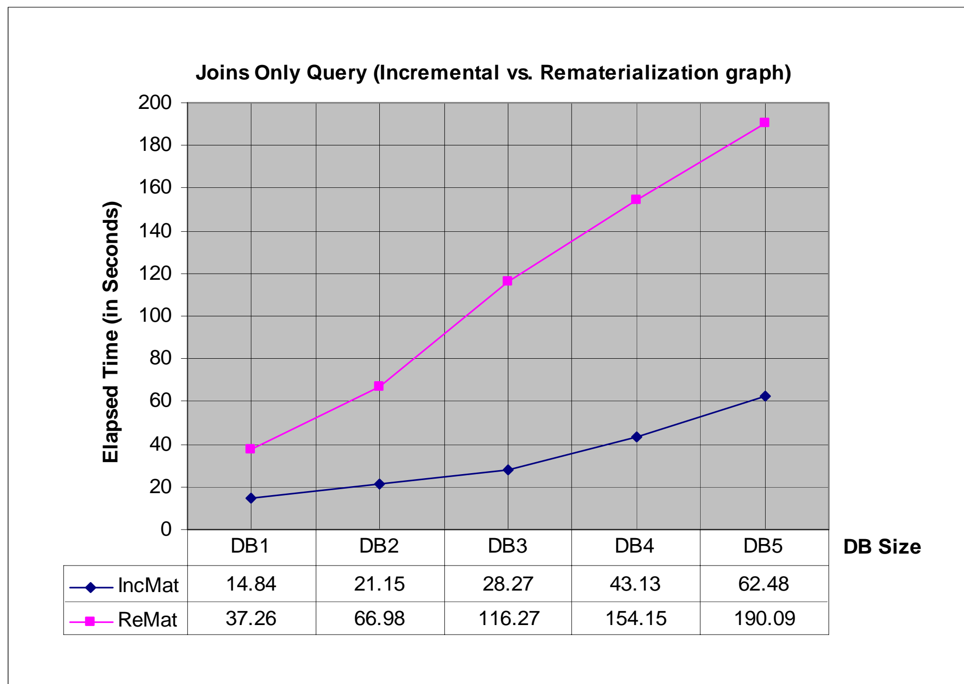


Fig. 4.9 (a) View maintenance costs for joins only query

For the joins only query from Fig. 4.9 (a), the incremental maintenance cost and rematerializing cost are 14.84 and 37.26 respectively at a database size of DB_1, but as database size increases cost increases for both cases. At a database size of DB_4, the incremental cost is 43.13 seconds whereas for rematerializing the cost is 154.15 seconds.

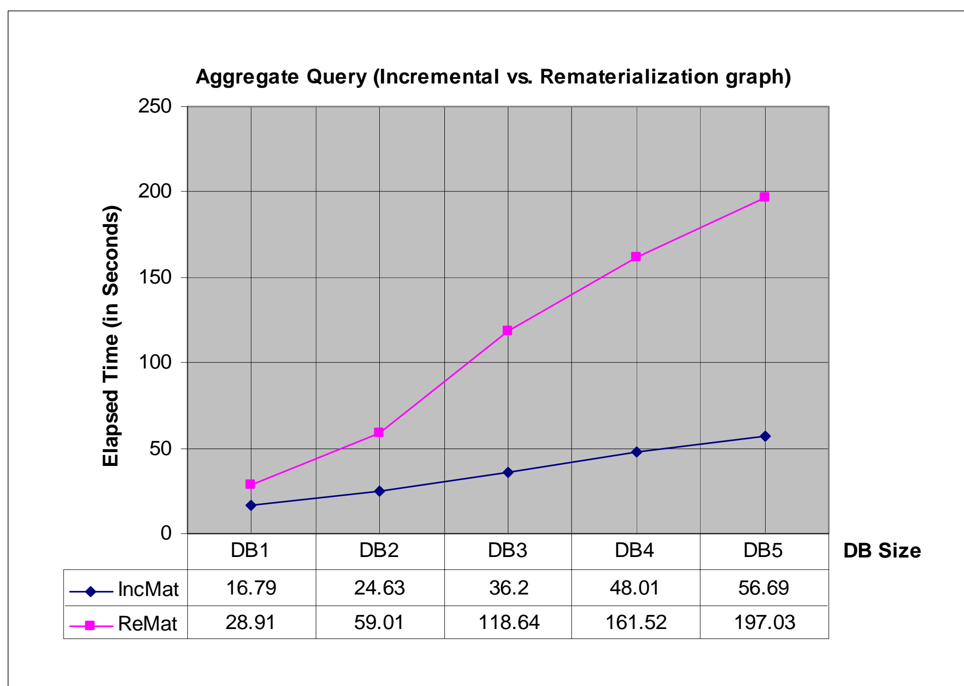
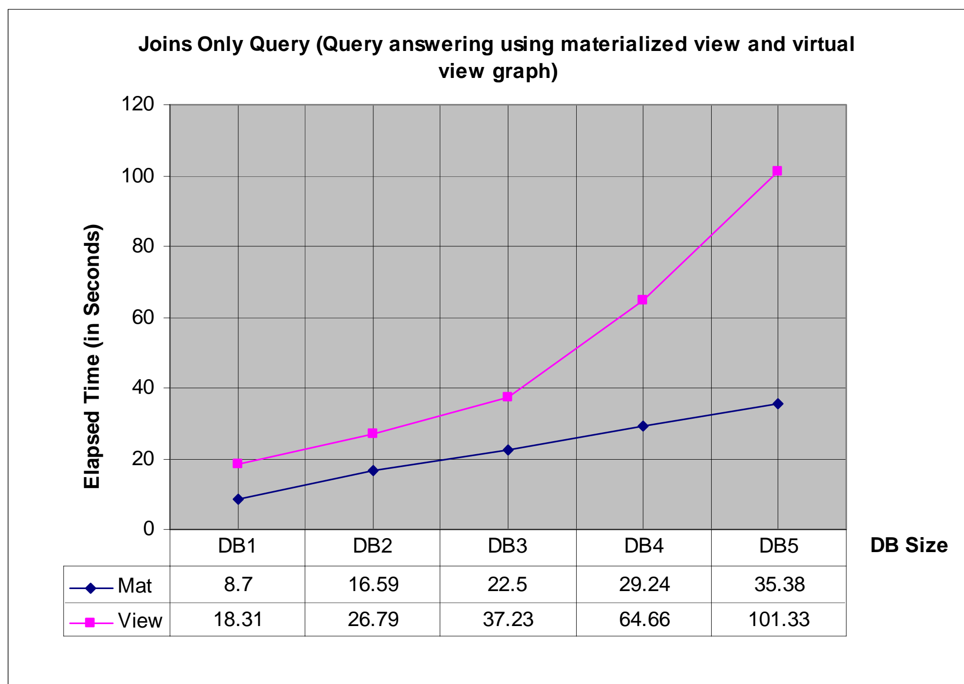


Fig. 4.9 (b) View maintenance costs for aggregate query

For the aggregate query from Fig. 4.9 (b), the incremental maintenance cost and rematerializing cost are 16.79 and 28.91 at a database size of DB_1, but as database size increases cost increases for both cases. At a database size of DB_3, the incremental cost is 36.2 seconds whereas for rematerializing the cost is 118.64.

So, the cost of materialized view maintenance increases with an increase in database size in both cases (incremental propagation and rematerializing) but incremental maintenance performs better that rematerializing.

Query answering: The Fig. 4.10 (a), (b), (c) and (d) show the cost of answering a view and using rewrite in comparison with the cost of answering a materialized view.



	DB1	DB2	DB3	DB4	DB5
Mat	8.7	16.59	22.5	29.24	35.38
View	18.31	26.79	37.23	64.66	101.33

Fig. 4.10 (a) Query answering costs of view for joins only query

For the joins only query, from Fig. 4.10 (a), the query answering cost for materialized view and virtual view are 8.7 and 18.31 at a database size of DB_1, but after that the difference is increases with in increase in database size. At a database size of DB_5, the query answering costs are 35.38 and 101.33 respectively for materialized view and virtual view.

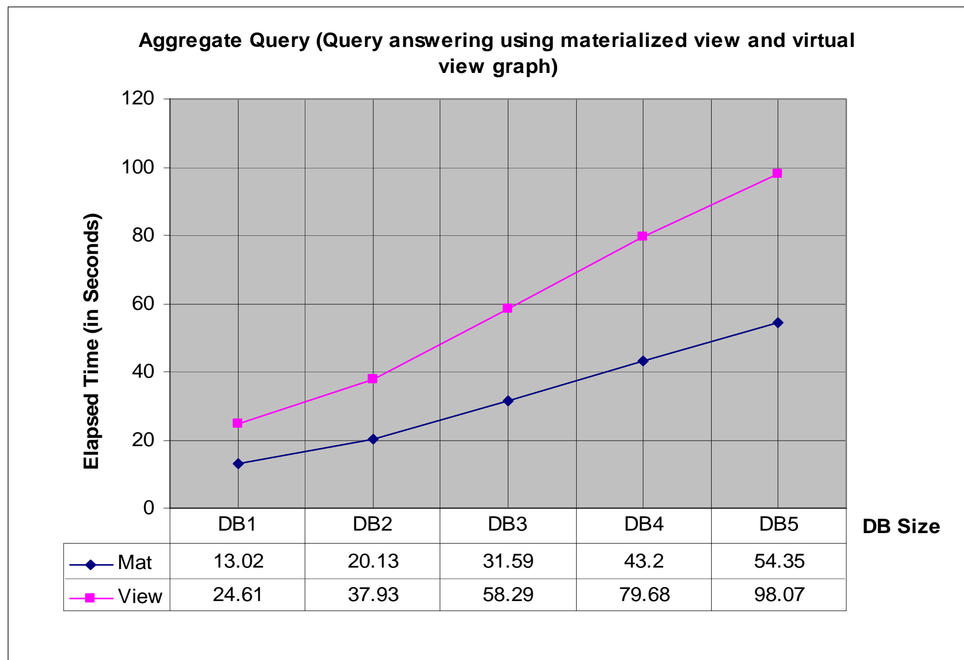


Fig. 4.10 (b) Query answering costs of view for aggregate query

For the aggregate query, from Fig. 4.10 (b), the query answering cost for materialized view and virtual view are 13.02 and 24.61 at a database size of DB_1, but after that the difference is increases with in increase in database size. At a database size of DB_4, the query answering costs are 43.2 and 79.68 respectively for materialized view and virtual view.

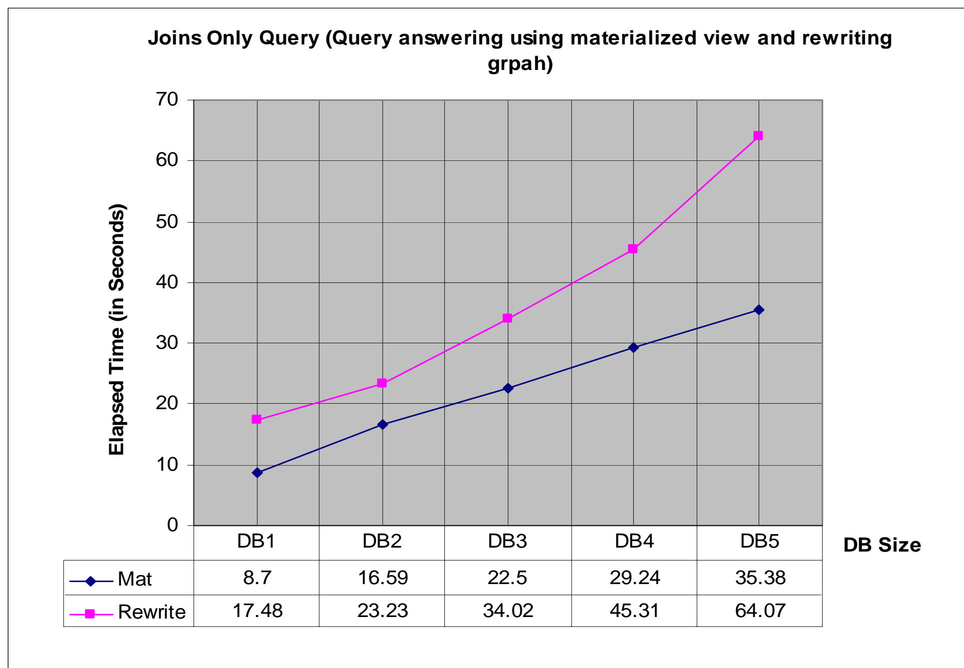
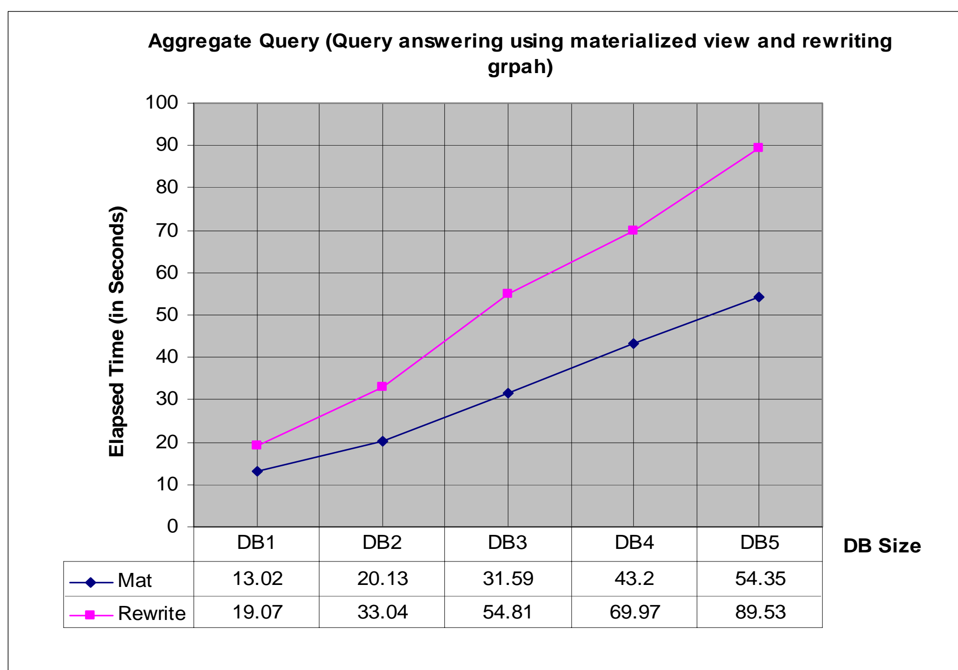


Fig. 4.10 (c) Query answering costs using rewrite for joins only query

For the joins only query, from Fig. 4.10 (c), the query answering cost for materialized view and using rewrite are 8.7 and 17.48 at a database size of DB_1, but after that the difference is increases with in increase in database size. At a database size of DB_5, the query answering costs are 35.38 and 64.07 respectively for materialized view and using rewrite.



	DB1	DB2	DB3	DB4	DB5
Mat	13.02	20.13	31.59	43.2	54.35
Rewrite	19.07	33.04	54.81	69.97	89.53

Fig. 4.10 (d) Query answering costs using rewrite for aggregate query

For the aggregate query, from Fig. 4.10 (d), the query answering cost for materialized view and using rewrite are 13.02 and 19.07 at a database size of DB_1, but after that the difference is increases with in increase in database size. At a database size of DB_4, the query answering costs are 43.2 and 69.97 respectively for materialized view and using rewrite.

So, it is seen that the query answering directly from the materialized views outperforms to that of the answering a virtual view and answering query using query rewrite. It is also noted that the query answering using rewrite is better than answering a virtual view. This is because that the query rewrite engine rewrites a query based on full text match or partial match to query from the materialized view rather than the base tables. Rewriting a query takes more time to query than the direct query from the materialized view because it takes time to search the materialized view text to match to its SQL statements and then queries from the materialized views if it is matched.

Relative costs: The ratio of the cost of answering a virtual view to the cost of incremental propagation, ratio of the cost of answering query rewrite to the incremental propagation and the ratio of the cost of answering a virtual view to the cost of answering a query rewrite measure how much it is profitable to select a view to materialize for improving the query performance. Fig. 4.11 (a), (b), (c), (d), (e) and (f) show the relative costs of joins only and aggregate queries.

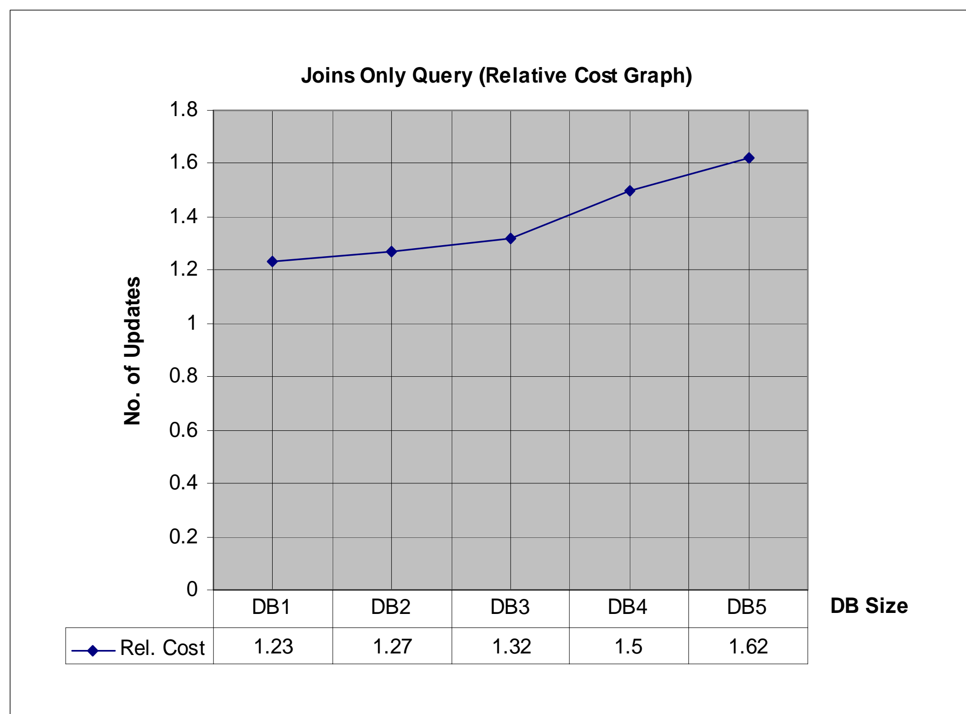
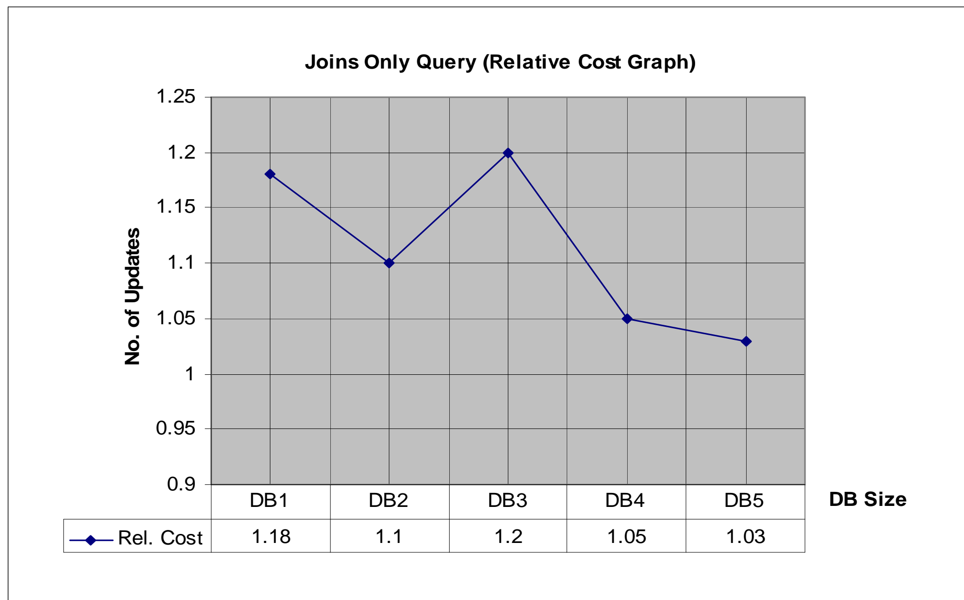


Fig. 4.11 (a) Relative cost of answering a view to the incremental propagation time for joins only query

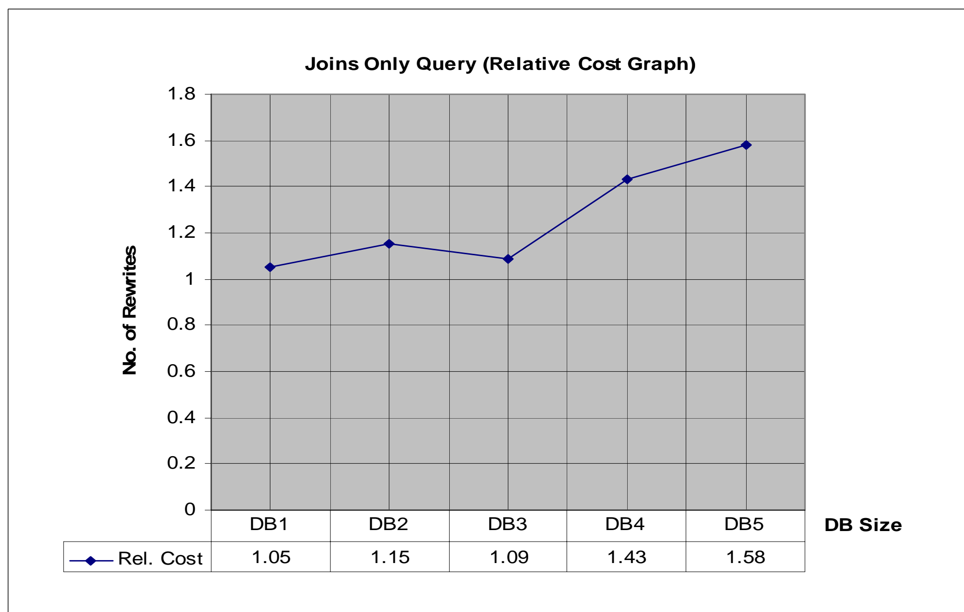
For the joins only query, from Fig. 4.11 (a), the relative cost of answering a virtual view to the incremental maintenance of the materialized view is 1.23 and it means that with the cost of answering a virtual view, one update can be propagated to the materialized view at a database size of DB_1. Similarly, at a database size of DB_5, the relative cost is 1.62 means two updates can be propagated with the cost of answering a virtual view.



	DB1	DB2	DB3	DB4	DB5
Rel. Cost	1.18	1.1	1.2	1.05	1.03

Fig. 4.11 (b) Relative cost of answering a query using rewrite to the incremental propagation time for joins only query

For the joins only query, from Fig. 4.11 (b), the relative cost of answering using rewrite to the incremental maintenance of the materialized view is 1.18 at a database size of DB_1 and with the cost of query answering using rewrite, one update can be propagated to the materialized view. Similarly, at a database size of DB_4, the relative cost is 1.05 means one update can be propagated with the cost of query answering using rewrite.



	DB1	DB2	DB3	DB4	DB5
Rel. Cost	1.05	1.15	1.09	1.43	1.58

Fig. 4.11 (c) Relative cost of answering a view to that of using rewrite for joins only query

For the joins only query, from Fig. 4.11 (c), the relative cost of answering a virtual view to that of using rewrite is 1.05 at a database size of DB_1 and it means that with the cost of answering a virtual view one rewrite can possible. Similarly, at a database size of DB_5, two rewrites can possible for the cost of answering a virtual view.

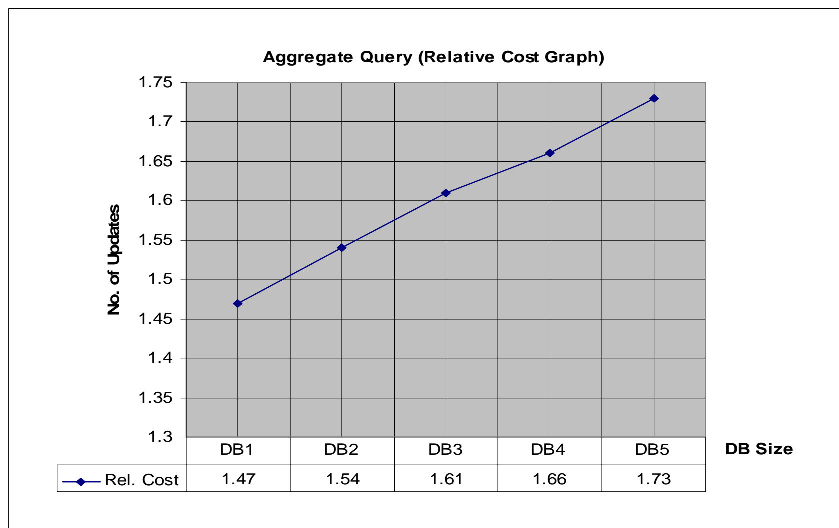


Fig. 4.11 (d) Relative cost of answering a view to the incremental propagation time for aggregate query

For the aggregate query, from Fig. 4.11 (d), the relative cost of answering a virtual view to the incremental maintenance of the materialized view is 1.47 and it means that with the cost of answering a virtual view, one update can be propagated to the materialized view at a database size of DB_1. Similarly, at a database size of DB_4, the relative cost is 1.66 means two updates can be propagated with the cost of answering a virtual view.

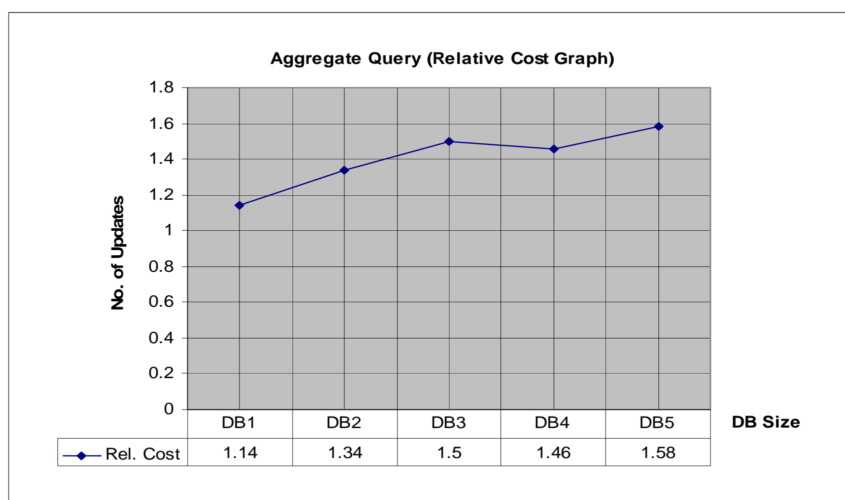


Fig. 4.11 (e) Relative cost of answering a query using rewrite to the incremental propagation time for aggregate query

For the aggregate query, from Fig. 4.11 (e), the relative cost of answering using rewrite to the incremental maintenance of the materialized view is 1.14 at a database size of DB_1 and it means that with the cost of query answering using rewrite, one update can be propagated to the materialized view. Similarly, at a database size of DB_5, the relative cost is 1.58 means two update can be propagated with the cost of answering using rewrite.

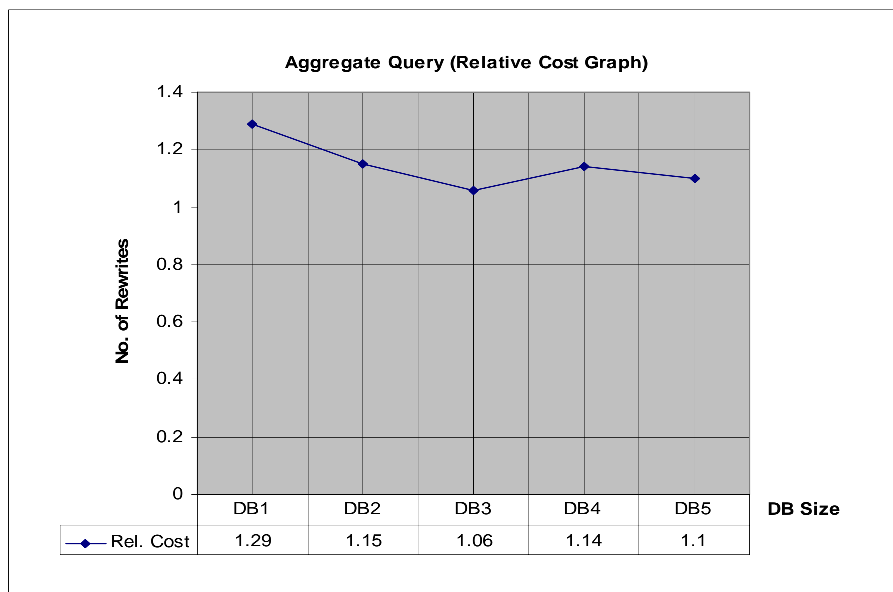


Fig. 4.11 (f) Relative cost of answering a view to that of using rewrite for aggregate query

For the aggregate query, from Fig. 4.11 (f), the relative cost of answering a virtual view to that of using rewrite is 1.29 at a database size of DB_1 and it means that with the cost of answering a virtual view, one rewrite can possible. With the increase in selectivity, less rewrite can possible with the cost of answering a virtual view and at a database size of DB_4, one rewrite can possible for the cost of answering a virtual view.

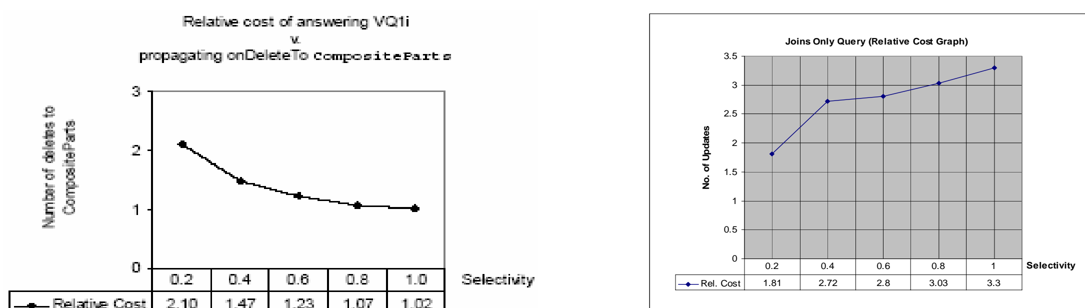
So, from the relative costs plots, it is observed that with the increase in database size for the cost of answering a virtual view or answering a query using rewrite, more number of updates can be propagated to the materialized views. It is also observed that if a query is materialized then using query rewrite, answering a query time is saved at a considerable amount in comparison with answering a virtual view.

In summary, the incremental materialized view maintenance is profitable over rematerializing a view each time a change is made to the base tables and hence we can infer that with the increase in database size, a view is more cost effective for materialization when the goal is to optimize query execution.

From the experimental results performed for view selectivity, complexity and database size, it is found that in all cases, incremental maintenance is always better than rematerializing and answering a query is cost effective in comparison with answering a virtual view and if a view satisfies the above measurements, that view can be selected for materialization for faster execution of that view or query.

4.2.4 Comparison with related work

Akhtar Ali *et al.* [13] evaluated the incremental materialized view maintenance performance on object databases to determine the view materialization circumstances by considering the relative cost comparison of answering a virtual view to the incremental propagation time. Here, additionally, the relative costs of answering query rewrite to the incremental propagation and the answering a virtual view to that of using rewrite have been considered to evaluate the incremental materialized view maintenance performance in ORDBMS and to determine the various situations of view materialization profitable. In Fig. 4.12, a relative cost comparison of answering a virtual view to the incremental propagation has shown on object database and on ORDBMS for selectivity issue. Here, it is found that for the view with selectivity of 0.2, two updates can be propagated incrementally for the cost of executing a query against the corresponding virtual view in both cases. With an increase in the selectivity that ratio reduces to one for object database while the ratio increases to three for ORDBMS, reflecting that for views with high selectivity, if updates are all frequent, incremental maintenance may not be beneficial for object databases but it can be beneficial for ORDBMS. It is noted that the experimental results may infer other conclusion depending on the query types selection and also the dependent data whether it contains only the object data or a combination of all kinds of data.



(a) Relative cost in object database **(b)** Relative cost in ODRBMS

Fig. 4.12 Relative cost of answering a view to the incremental propagation

4.3 Experiments Results on Dynamic Selection of Views and Removal of Materialized views

For the dynamic selection of views for materialization and dynamic removal of old materialized views, we need to find the a set of individual queries and a set of materialized views with the execution time, selectivity, complexity calculation, table maintenance cost, query processing cost, view maintenance cost etc. The subsequent sections illustrate the experimental results for the dynamic view selection for materialization and dynamic selection of existing materialized views to remove from the database.

4.3.1 Dynamic selection of views

Experiment no. 01:

First, we have calculated the maintenance cost associated with the tables using the update frequencies of the tables and the weighting factors reflecting the importance of the tables. The maintenance cost calculation of the tables is shown in the following Table 4.3.

Table 4.3 Maintenance cost calculation of the tables

SL	t_m	i_m	m_m	d_m	u_m	w_m	MT_m
1	*channels*	5	0	0	1.67	1.42	2.37
2	*countries*	0	0	0	0.00	0.00	0.000
3	*costs*	10	10	5	8.33	3.22	26.82
4	*customers*	50	0	10	20.00	4.39	87.80
5	*products*	10	0	0	3.33	2.11	7.03
6	*promotions*	5	2	0	2.33	1.74	4.05
7	*times*	1	0	0	0.33	0.41	0.14
8	*sales*	200	0	0	66.67	6.08	405.35

In Table 4.3, the weighting factor of the table is calculated using $Weight(w_m) = \log_2(u_m + 1)$.

Now, the different cost associated with a query is calculated using the formula and algorithm defined in Chapter 3 and a table of (n, 12 + m) is filled up in the Table 4.4. The detail query definitions are provided in Appendix C.

Table 4.4 Query associated cost calculation

Q_n	f_n	w_n	e_n	s_n	t_n	j_n	a_n	C_n	QP_n	MT_1	MT_2	MT_3	MT_4	MT_5	MT_6	MT_7	MT_8	MC_n	TC_n
Q_1	4	2.32	10.79	1.0000	1	0	5	6	600.79	0	0	0	0	0	0	0	405.35	405.35	1006.14
Q_2	12	3.70	77.4	1.0000	3	2	0	5	17182.80	0	0	0	87.80	0	0	0.14	405.35	493.29	17676.09
Q_3	14	3.91	11.92	0.0030	4	4	0	8	15.66	0	0	0	87.80	0	0	0	405.35	493.15	508.81
Q_4	13	3.81	32.31	0.4000	6	5	6	17	10882.14	0	0	0	0	0	0	0.14	405.35	405.49	11287.63
Q_5	11	3.58	9.6	1.0000	5	4	3	12	4536.58	0	0.00	0	87.80	7.03	0	0.14	405.35	500.32	5036.90
Q_6	6	2.81	0.31	0.0005	2	2	2	6	0.16	0	0	0	0	7.03	0	0	405.35	412.38	412.54
Q_7	5	2.58	9.01	1.0000	3	2	5	10	1162.29	0	0	0	0	7.03	0	0.14	405.35	412.52	1574.81
Q_8	9	3.32	7.71	1.0000	3	2	1	6	1382.25	0	0	0	0	7.03	0	0.14	405.35	412.52	1794.77
Q_9	12	3.70	15.03	1.0000	3	2	1	6	4003.99	0	0	0	0	7.03	0	0.14	405.35	412.52	4416.51
Q_{10}	12	3.70	10.2	1.0000	4	3	2	9	4075.92	0	0	0	87.80	7.03	0	0.14	405.35	500.32	4576.24
Q_{11}	18	4.25	69.34	1.0000	3	2	0	5	26522.55	0	0	0	0	7.03	0	0.14	405.35	412.52	26935.07
Q_{12}	10	3.46	11.04	1.0000	4	3	1	8	3055.87	0	0	0	87.80	7.03	0	0.14	405.35	500.32	3556.19
Q_{13}	11	3.58	0.65	0.0080	2	3	1	6	1.32	0	0	0	0	0	0	0.14	405.35	405.49	406.81
Q_{14}	13	3.81	8.67	1.0000	4	3	2	9	3864.83	0	0	0	87.80	7.03	0	0.14	405.35	500.32	4365.15
Q_{15}	20	4.39	9.62	1.0000	4	3	1	8	6757.09	0	0	0	87.80	7.03	0	0.14	405.35	500.32	7257.41
Q_{16}	1	1.00	44.67	0.4440	4	4	0	8	158.67	0	0.00	0	87.80	0	0	0.14	405.35	493.29	651.96
Q_{17}	11	3.58	40.7	1.0000	4	3	1	8	12822.13	0	0	0	87.80	7.03	0	0.14	405.35	500.32	13322.45
Q_{18}	4	2.32	11.2	1.0000	3	2	1	6	623.62	0	0	0	87.80	7.03	0	0	405.35	500.18	1123.80
Q_{19}	11	3.58	30.09	1.0000	4	2	1	7	8294.61	0	0	0	87.80	7.03	0	0.14	405.35	500.32	8794.93

$w_n = log_2 (f_n + 1)$; $c_n = t_n + j_n + a_n$; $QP_n = f_n w_n e_n s_n c_n$; $MC_n = MT_1 + MT_2 + \ldots + MT_8$; $TC_n = QP_n + MC_n$;

Q_n	f_n	w_n	e_n	s_n	t_n	j_n	a_n	C_n	QP_n	MT_1	MT_2	MT_3	MT_4	MT_5	MT_6	MT_7	MT_8	MC_n	TC_n
Q_{20}	6	2.81	8.56	1.000	4	3	1	8	1154.57	2.37	0	0	87.80	0	0	0.14	405.35	495.66	1650.23
Q_{21}	6	2.81	7.78	1.000	5	4	1	10	1311.71	2.37	0.00	0	87.80	0	0	0.14	405.35	495.66	1807.37
Q_{22}	11	3.58	8.73	1.000	5	4	1	10	3437.87	2.37	0.00	0	87.80	0	0	0.14	405.35	495.66	3933.53
Q_{23}	20	4.39	8.53	1.000	5	4	1	10	7489.34	2.37	0.00	0	87.80	0	0	0.14	405.35	495.66	7985.00
Q_{24}	4	2.32	8.98	1.000	5	4	1	10	833.34	2.37	0.00	0	87.80	0	0	0.14	405.35	495.66	1329.00
Q_{25}	1	1.00	8.84	1.000	5	4	1	10	88.40	2.37	0.00	0	87.80	0	0	0.14	405.35	495.66	584.06
Q_{26}	9	3.32	10.39	1.000	5	4	1	10	3104.53	2.37	0.00	0	87.80	0	0	0.14	405.35	495.66	3600.19
Q_{27}	17	4.17	7.54	1.000	5	4	1	10	5345.11	2.37	0.00	0	87.80	0	0	0.14	405.35	495.66	5840.77
Q_{28}	7	3.00	36.18	1.000	5	4	1	10	7597.80	2.37	0.00	0	87.80	0	0	0.14	405.35	495.66	8093.46
Q_{29}	8	3.17	0.21	0.128	4	5	1	10	6.82	0	0.00	0	87.80	0	0	0.14	405.35	493.29	500.11
Q_{30}	4	2.32	125.59	1.000	5	3	1	9	10489.28	2.37	0.00	0	87.80	0	0	0.14	405.35	495.66	10984.94
Q_{31}	13	3.81	111.06	1.000	5	3	1	9	49507.22	2.37	0.00	0	87.80	0	0	0.14	405.35	495.66	50002.88
Q_{32}	7	3.00	8.75	1.000	5	4	1	10	1837.50	2.37	0.00	0	87.80	0	0	0.14	405.35	495.66	2333.16
Q_{33}	19	4.32	118.46	1.000	5	3	1	9	87508.77	2.37	0.00	0	87.80	0	0	0.14	405.35	495.66	88004.43
Q_{34}	5	2.58	41.51	1.000	5	4	1	10	5354.79	2.37	0.00	0	87.80	0	0	0.14	405.35	495.66	5850.75
Q_{35}	14	3.91	47.17	1.000	5	4	1	10	25820.86	2.37	0.00	0	87.80	0	0	0.14	405.35	495.66	26316.52
Q_{36}	11	3.58	22.28	1.000	6	5	1	12	10528.64	2.37	0.00	0	87.80	7.03	0	0.14	405.35	502.69	11031.33
Q_{37}	10	3.46	2.59	0.209	5	7	1	13	243.48	0	0.00	0	87.80	7.03	0	0.14	405.35	500.32	743.80
Q_{38}	13	3.81	10.37	1.000	5	7	2	14	7190.77	0	0.00	0	87.80	7.03	0	0.14	405.35	500.32	7691.09

$w_n = log_2 (f_n + 1)$; $c_n = t_n + j_n + a_n$; $QP_n = f_n w_n e_n s_n c_n$; $MC_n = MT_1 + MT_2 + \ldots + MT_8$; $TC_n = QP_n + MC_n$;

The minimum of the total query cost, *Min* (*TC*) = 9289.13. So according to the *DynamicViewMaterializationSelection* in Chapter 3, the following queries are selected for materialization to improve the query performance.

Table 4.5 Candidates query for the view materialization

Query	Query Cost	Maintenance Cost	Total Cost
Q_2	17182.80	493.29	17676.09
Q_4	10882.14	405.49	11287.63
Q_{11}	26522.55	412.52	26935.07
Q_{17}	12822.13	500.32	13322.45
Q_{30}	10489.28	495.66	10984.94
Q_{31}	49507.22	495.66	50002.88
Q_{33}	87508.77	495.66	88004.43
Q_{35}	25820.86	495.66	26316.52
Q_{36}	10528.64	502.69	11031.33

From the experimental results, it reveals that the dynamic selection algorithm selects the queries with not only high access frequencies but it also takes into consideration for higher execution cost of the query. The Fig. 4.13 (a) and 4.13 (b) illustrate the dynamically selected queries access frequency-complexity-execution cost graph.

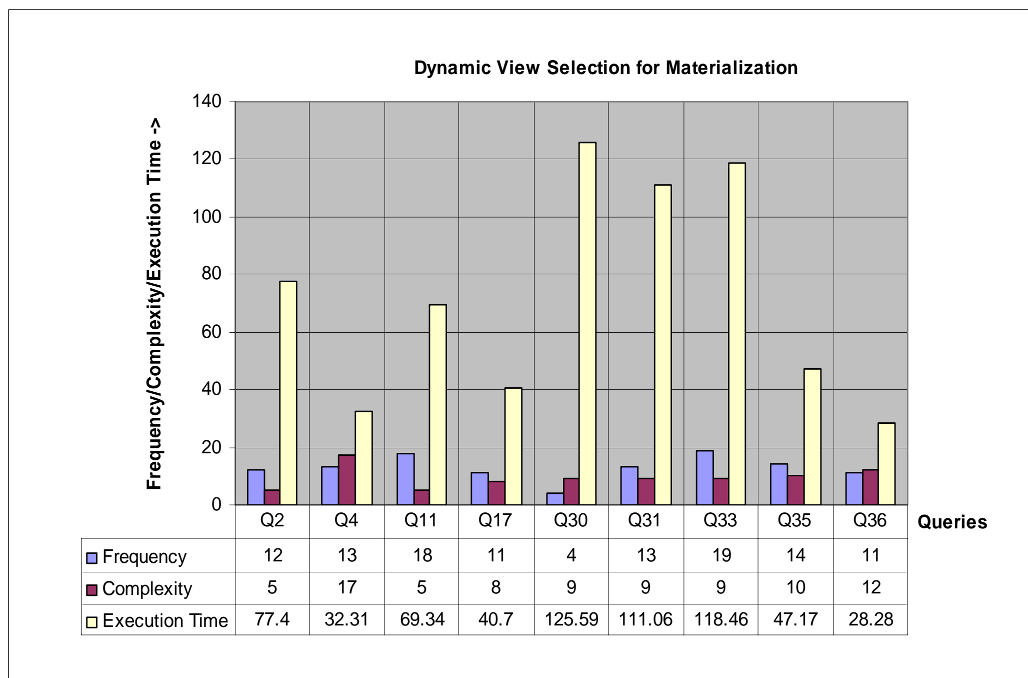


Fig. 4.13 (a) Dynamically selected queries for materialization (Column Chart)

From Fig. 4.13 (a), it is found that the dynamic model of the selection of views for materialization dynamically selects queries not only those have higher access frequencies but selects queries having execution cost. This means that if a query or a virtual view can execute in a considerable amount of time but the query is executed a number of times, the model does not select that query for materialization. The model also recommends for view materialization of those queries that have higher execution cost but access frequencies are a little bit low than the higher access frequencies at a particular time period. In figure, the query Q_{33} has the higher access frequency 19 among all other selected queries for materialization and also this query has an execution time of 118.46 less than the higher execution time of other queries that is 125.59. So the selection of this query for materialization is profitable. In a second case, the query Q_{30} has a higher execution cost but the access frequency is only 4 among all other queries and it is selected for materialization because the query is also a complex query having a complexity count of 9 which is more than the minimum of the complexity count of the queries. The same information of the selected queries is depicted in Fig. 4.13 (b) with a line chart.

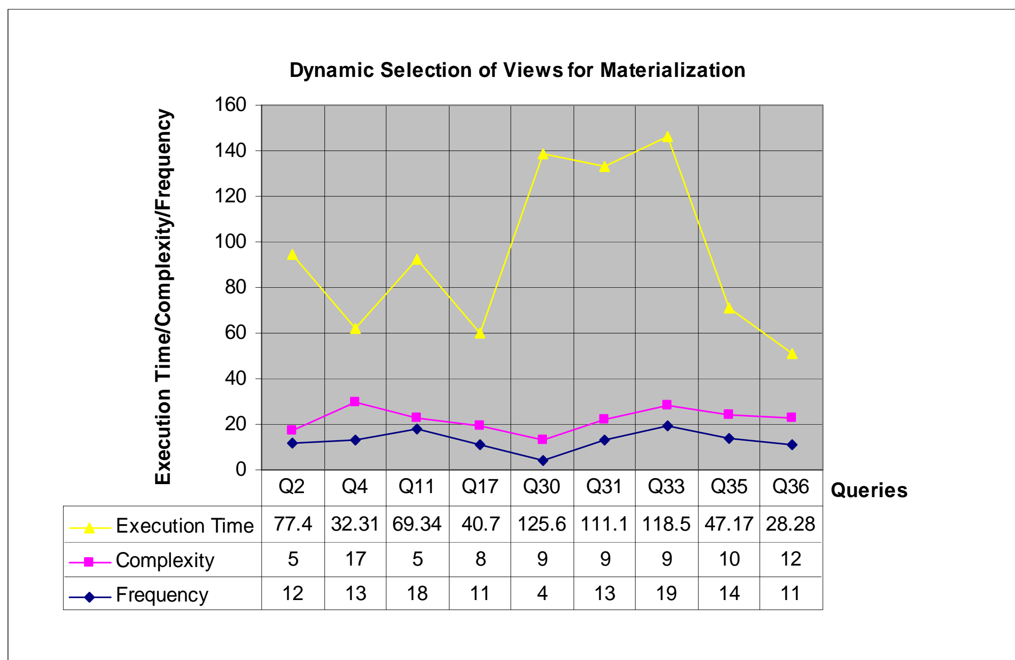


Fig. 4.13 (b) Dynamically selected queries for materialization (Line Chart)

After the selected queries have been materialized, the query answering cost comparison between the virtual view and materialized view is shown in the following Fig. 4.14 where it is clearly identified that materializing a query increase query response time.

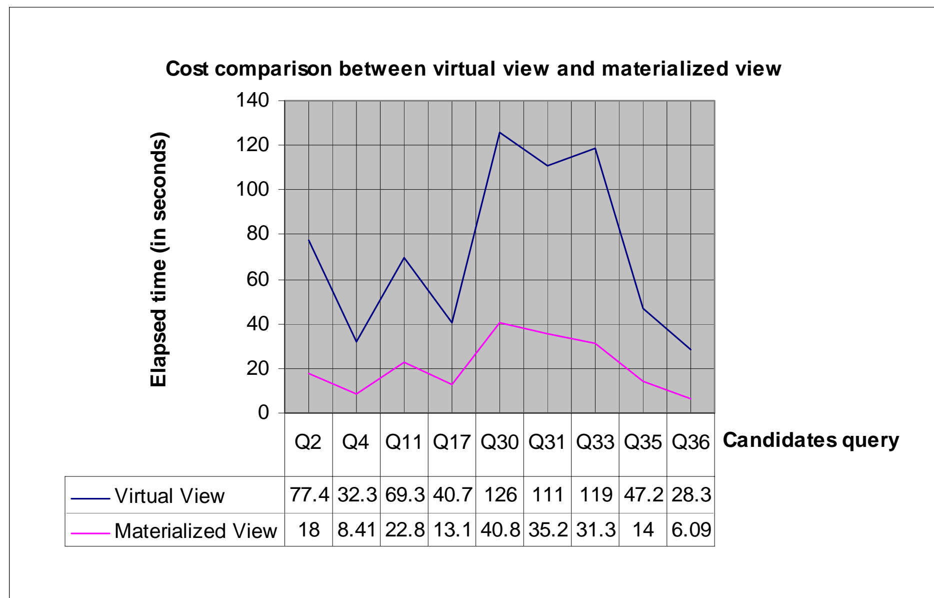


Fig. 4.14 Query answering cost comparison for experiment 01

Experiment no. 02:

Again, first, we have calculated the maintenance cost associated with the tables using the update frequencies of the tables and the weighting factors reflecting the importance of the tables. The maintenance cost calculation of the tables is shown in the following Table 4.6.

Table 4.6 Maintenance cost calculation of the tables

SL	t_m	i_m	m_m	d_m	u_m	w_m	MT_m
1	*countries*	0	0	0	0.00	0.00	0.000
2	*customers*	50	0	10	20.00	4.39	87.80
3	*inventories*	200	0	0	66.67	6.08	405.35
4	*orders*	300	20	50	123.33	6.96	858.38
5	*order_items*	300	20	50	123.33	6.96	858.38
6	*product_description*	10	0	0	3.33	2.11	7.03
7	*product_information*	10	0	0	3.33	2.11	7.03
8	*warehouses*	50	0	0	16.67	4.14	69.01

In Table 4.6, the weighting factor of the table is calculated using $Weight(w_m) = \log_2(u_m + 1)$.

Now, the different cost associated with a query is calculated using the formula and algorithm defined in Chapter 3 and a table of (n, 12 + m) is filled up in the Table 4.7. The detail query definitions are provided in Appendix C.

Table 4.7 Query associated cost calculation (experiment-02)

Q_n	f_n	w_n	e_n	s_n	t_n	j_n	a_n	C_n	QP_n	MT_1	MT_2	MT_3	MT_4	MT_5	MT_6	MT_7	MT_8	MC_n	TC_n
Q_1	13	3.81	31.03	1	2	1	1	4	6147.66	0	87.8	0	0	0	0	0	69.01	87.80	6234.80
Q_2	11	3.58	32.47	1	4	4	0	8	10229.35	0	0	405.34	0	0	7.03	7.03	0	488.41	10717.76
Q_3	5	2.58	438.23	1	2	2	0	4	22612.67	0	87.8	0	0	0	0	0	0	87.80	22700.47
Q_4	9	4.39	281.34	1	3	2	0	5	55578.72	0	87.8	0	858.38	858.38	0	0	0	1804.56	57383.28
Q_5	15	4.00	109.10	1	2	1	0	3	19638.00	0	0	405.34	0	0	0	0	69.01	474.35	20112.35
Q_6	10	3.46	156.86	1	2	1	0	3	16282.07	0	0	0	858.38	858.38	0	0	0	1716.76	17998.83
Q_7	20	4.39	188.11	1	3	2	0	5	82580.29	0	0	405.34	0	0	0	7.03	69.01	481.38	83061.67
Q_8	12	3.70	187.56	1	2	2	0	4	33310.66	0	0	0	0	0	7.03	7.03	0	14.06	33324.72
Q_9	7	3.00	220.21	1	3	2	2	7	32370.87	0	87.8	0	858.38	858.38	0	0	0	1804.56	34175.43
Q_{10}	9	3.32	209.13	1	3	2	1	6	37492.83	0	87.8	0	858.38	858.38	0	0	0	1804.56	39297.39

$w_n = log_2 (f_n + 1)$; $c_n = t_n + j_n + a_n$; $QP_n = f_n w_n e_n s_n c_n$; $MC_n = MT_1 + MT_2 + \ldots + MT_8$; $TC_n = QP_n + MC_n$;

The minimum of the total query cost, *Min* (*TC*) = 29500.67. So according to the *DynamicViewMaterializationSelection* in Chapter 3, the following queries are selected for materialization to improve the query performance.

Table 4.8 Candidates query for the view materialization

Query	Query Cost	Maintenance Cost	Total Cost
Q_4	55578.72	1804.56	57383.28
Q_7	82580.29	481.38	83061.67
Q_8	33310.66	14.06	33324.72
Q_9	32370.87	1804.56	34175.43
Q_{10}	37492.83	1804.56	39297.39

From the experimental results, it reveals that the dynamic selection algorithm selects the queries with not only high access frequencies but it also takes into consideration for higher execution cost of the query. The Fig. 4.15 illustrates the dynamically selected queries access frequency-complexity-execution cost graph.

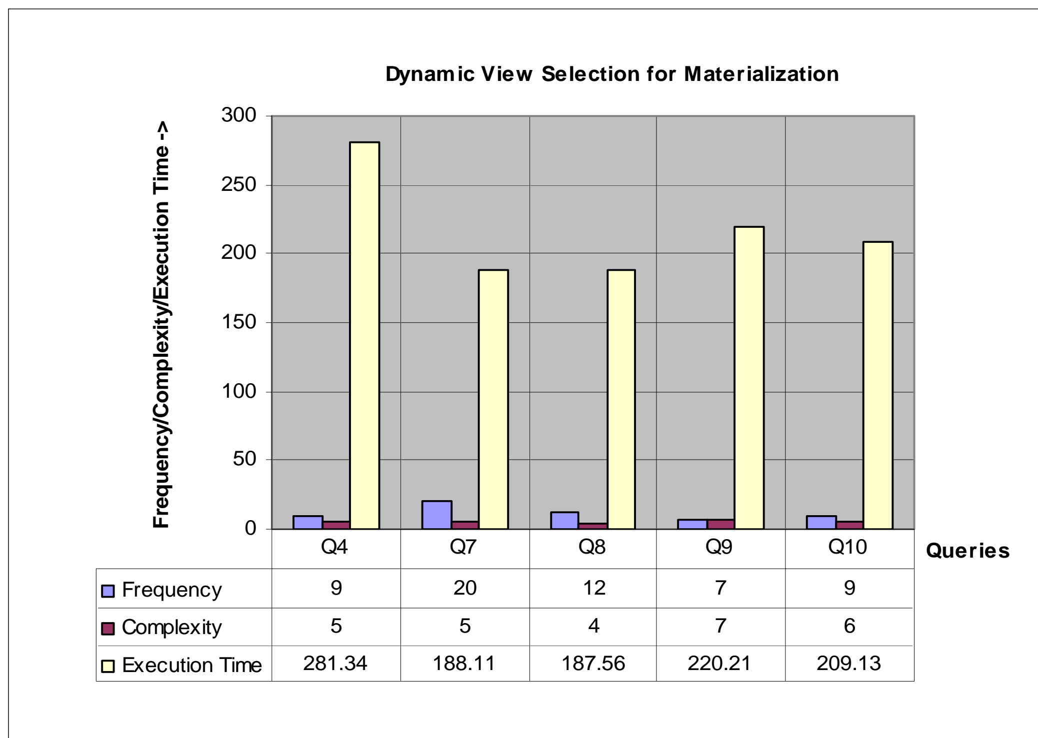
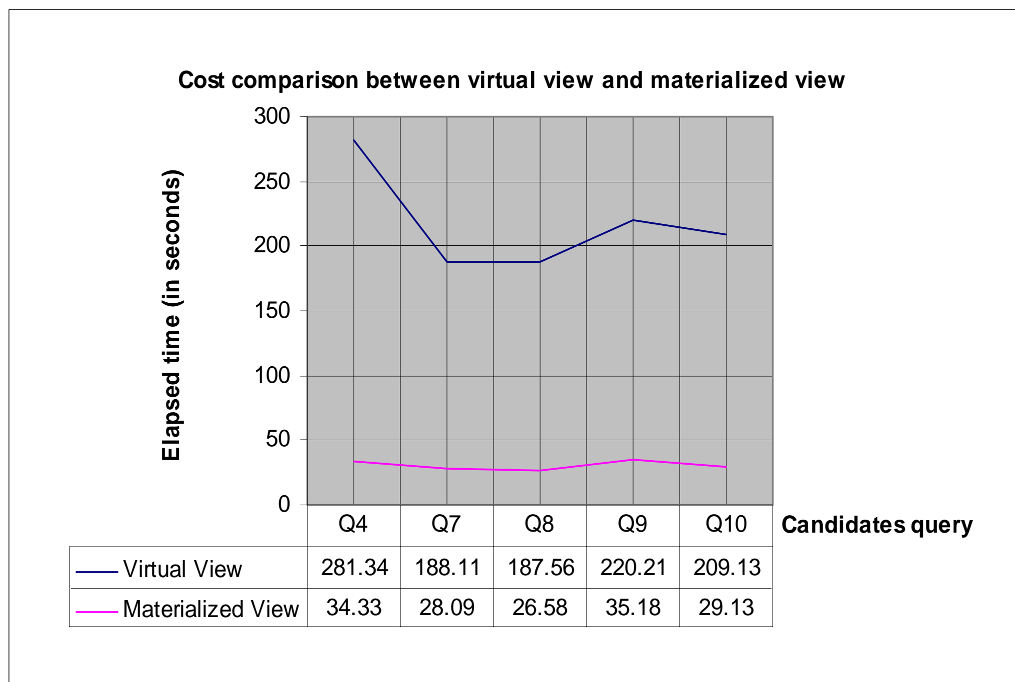


Fig. 4.15 Dynamically selected queries for materialization

From Fig. 4.15, it is found that the dynamic model of the selection of views for materialization dynamically selects queries not only those have higher access frequencies

but selects queries having execution cost. This means that if a query or a virtual view can execute in a considerable amount of time but the query is executed a number of times, the model does not select that query for materialization. The model also recommends for view materialization of those queries that have higher execution cost but access frequencies are a little bit low than the higher access frequencies at a particular time period. In figure, the query Q_7 has the higher access frequency 20 among all other selected queries for materialization and also this query has an execution time of 188.11 less than the higher execution time of other queries that is 281.34. So the selection of this query for materialization is profitable. In a second case, the query Q_4 has a higher execution cost and the access frequency is 9 among all other queries and it is selected for materialization because the query is also a complex query having a complexity count of 5 which is more than the minimum of the complexity count of the queries.

After the selected queries have been materialized, the query answering cost comparison between the virtual view and materialized view is shown in the following Fig. 4.16 where it is clearly identified that materializing a query increase query response time.



	Q4	Q7	Q8	Q9	Q10
Virtual View	281.34	188.11	187.56	220.21	209.13
Materialized View	34.33	28.09	26.58	35.18	29.13

Fig. 4.16 Query answering cost comparison for experiment 02

4.3.2 Dynamic removal of materialized views

In order to dynamically select the materialized views with low access frequencies for the removal from the database to free the storage space for the new view materialization and also to decrease the overall view maintenance cost, first we have computed the access frequencies of the existing materialized views. The existing materialized views with the access frequencies of these are shown in the Table 4.9.

Table 4.9 Access frequencies of existing materialized views

SL No.	Materialized views (MV_n)	Access frequencies (f_n)
1.	MV_2	6
2.	MV_4	4
3.	MV_{11}	19
4.	MV_{17}	10
5.	MV_{30}	3
6.	MV_{31}	11
7.	MV_{33}	5
8.	MV_{35}	11
9.	MV_{36}	18

Now the minimum of the access frequencies is calculated by dividing the total number of materialized views to the summation of the access frequencies and the $Min\ (f) = 9.67$. So according the *DynamicMaterializedViewRemoval* algorithm in the Chapter 3, the following materialized views are selected for removal dynamically.

Table 4.10 Dynamically selected materialized views for removal

SL No.	Materialized views (MV_n)
1.	MV_2
2.	MV_4
3.	MV_{30}
4.	MV_{33}

4.3.3 Comparison with related work

The factors that have been considered here to compare our work with work done by Ashadevi and Dr. Balashubramanian [35] are listed out in the following Table 4.11.

Table 4.11 Comparison of factors for selection of views with [35]

SL	Factors	Existing Method [35]	Proposed Method
1.	Query access frequency	Yes	Yes
2.	Query weight	No	Yes
3.	Query execution time	No	Yes
4.	View selectivity	No	Yes
5.	View complexity	No	Yes
6.	Base table update frequency	Yes	Yes
7.	Table weight	No	Yes
8.	Priority of table	Yes	No
9.	Threshold level	Arbitrary	Dynamic

The following points can be noted based on factors in the above table:

i. A weighting factor based on the query access frequency has been assigned to each of the query to reflect the importance of the query which has not been considered in [35];

ii. Query execution time has been considered here in the sense that if any query can be executed in reasonable amount of time then that query should not be selected for materialization;

iii. If a query retrieves more rows in comparison with the input rows and access frequency is high can be selected for materialization and the query executes quickly;

iv. Computing large number of joining at run time of a query faces longer execution time. In order to avoid, large joining or aggregation operations at run time, complexity based on joining, aggregations and tables involved has been considered for the view selection;

v. The base tables may be updated frequently, so a weighting factor has been assigned to each of the base table to reflect the importance of the tables based on

the update frequency of the base tables rather than using a predefined priority value which is used in [35]. A predefined priority value for the base tables might not properly reflect the table's importance;

vi. In [35], the threshold level for the selection of views or any other stage has been selected arbitrary. An arbitrary selection of threshold level is difficult to choose as it may results unnecessary selection of views to materialize. In this work, the threshold level has been calculated dynamically based on the total cost associated with the queries;

vii. In [35], the arbitrary threshold level has been selected two times – first at the time of addition of the high frequency queries to the vector of selected queries and second time at the selection of views for further process. Here, the dynamic threshold level need to be defined only at the end of all calculation to the views to materialize;

viii. At the beginning, queries with high access frequencies have been selected to process further in [35] but this may lead to opt out the selection of queries with higher execution time, more complex or retrieving most of the input records. This research guarantees the view selection with not only the high access frequencies but also with higher execution time, complex or higher selectivities i.e., an appropriate set of views is selected for materialization.

In case of removing the old existing materialized views and to free storage space for future view materialization, access frequencies and storage space have been calculated and then materialized views are selected for removal based on arbitrary threshold level in [35]. In this research, we have considered only the access frequencies of the materialized views for removal based on the dynamic threshold level computing from the access frequencies of the materialized views. The consideration of only materialized view access frequencies in the sense that materialized views occupying large storage spaces might have higher access frequencies and removing it would result unnecessary higher execution time. So, the materialized views with only low access frequencies are selected to remove from the database to free storage space for future view materialization.

Here, experiments have shown based on two points of arbitrary threshold level selection and selecting first the queries with higher access frequencies for further process. Let, the

threshold level for the initial selection of queries or views with higher access frequencies is 12. So, the queries with higher frequencies than the threshold level 12 are selected which are listed in Table 4.9.

Table 4.12 Initial selection of views for further process

View	Access Frequency	Execution Time	Complexity	Total Cost
Q_3	14	11.92	8	508.81
Q_4	13	32.31	17	11287.63
Q_{11}	18	69.34	5	26935.07
Q_{14}	13	8.67	9	4365.15
Q_{15}	20	9.62	8	7257.41
Q_{23}	20	4.39	10	7985.00
Q_{27}	17	4.17	10	5840.77
Q_{31}	13	3.81	9	50002.88
Q_{33}	19	4.32	9	88004.43
Q_{35}	14	3.91	10	26316.52
Q_{38}	13	3.81	14	7691.09

From Table 4.9, it is seen that the initial selection of views higher access frequencies has opt out the views Q_2, Q_{17}, Q_{30} and Q_{36} for further process which views have selected for materialization using our method in the final stage, because these views may have low access frequencies than the threshold level 12 but these have higher execution cost and also complex. On the other hand, in Table 4.9, there are some views that may have higher access frequencies but can execute in a reasonable amount of time.

Now, let the threshold level for the final selection based on the total cost associated with the queries is 30000. So, according to our view selection algorithm, views Q_{31} and Q_{33} are selected for materialization. If we determine the dynamic threshold level by dividing the total number of views to the summation of the total costs of the views, the threshold level is selected as 21472.25. By using this dynamic threshold, we get the following views for materialization in Table 4.10.

Table 4.13 Selected views for materialization

View	Access Frequency	Execution Time	Complexity	Total Cost
Q_{11}	18	69.34	5	26935.07
Q_{31}	13	3.81	9	50002.88
Q_{33}	19	4.32	9	88004.43
Q_{35}	14	3.91	10	26316.52

The following Fig. 4.17 shows the selected views status for materialization to materialize.

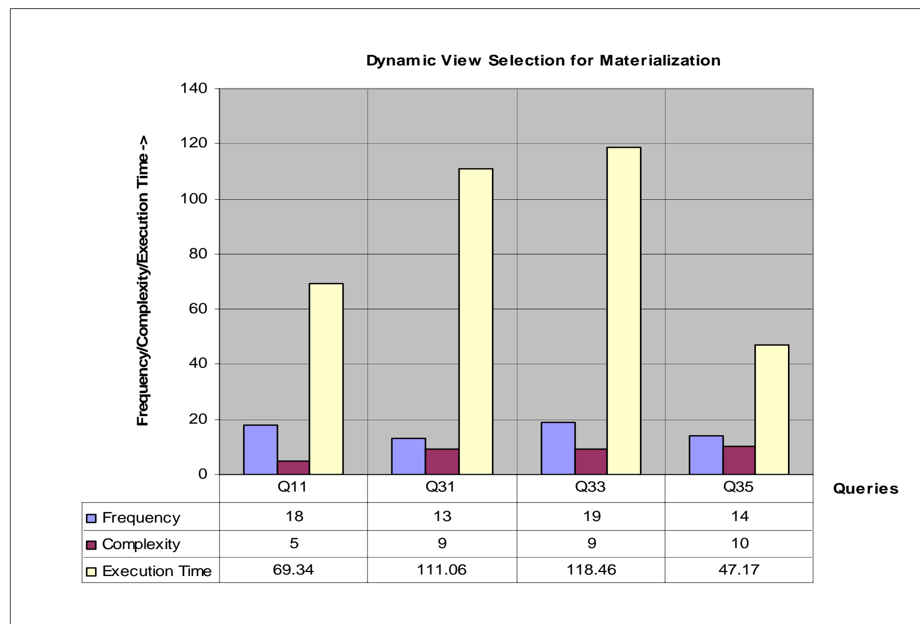
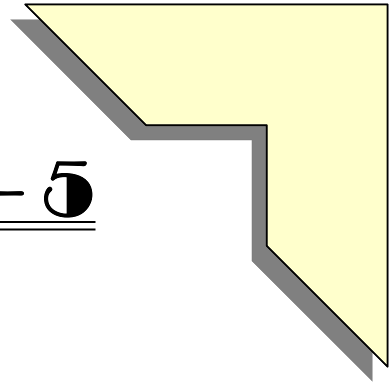


Fig. 4.17 Dynamically selected queries for materialization

So, in comparison with the previously selected views in Section 4.3.1 and from Fig. 4.17, we found that if the views with high access frequencies are selected for further processing based on arbitrary threshold level, there is chance of opt out of the views that could be materialized and be profitable to improve the view performance. But as at first, the views with less access frequencies than the threshold level are not selected (for example, views like Q_2 or Q_{30} where Q_2 has an access frequency of 12 and normal execution requires 77.4 seconds and Q_{30} has an access frequency of 4 but normal execution requires 125.59 seconds). After that, arbitrary threshold level selection for view selection selects only two views Q_{31} and Q_{33} while all other views that could be beneficial for materialization are not selected. Finally, based on the dynamic threshold level in the seconds case (Fig. 4.17), four views are selected for materialization. In comparison with the Fig. 4.13 and Fig. 4.17, it is found that dynamic threshold level is much better than arbitrary threshold level for selecting appropriate set of views to materialize for improving query performance.



CHAPTER - 5

CONCLUSION and FUTURE RESEARCH

5.1 Conclusion

5.2 Recommendation for Future Work

C h a p t e r 5

CONCLUSION AND FUTURE RESEARCH

This chapter summarizes the conclusion drawn from the research performed for this thesis and finally recommends for future research works.

5.1 Conclusion

From a user point of view, a query needs to execute very quickly. And for the faster query response, the results of that query should have to be stored in the database prior to the execution of the query. Materialized views provide this benefit. But materializing needs to reflect the changes that are made in its base relations with very cost effectively. Also users request usually lots of queries at a time to execute faster, but all of those queries cannot be materialized as it incurs maintenance cost. All of the views that have been materialized before may not used for long time and hence to reduce the maintenance cost and to free the storage space for new view materialization old materialized views need to be removed periodically. In this research work, we have developed a methodology to evaluate the incremental materialized view maintenance performance and to determine the circumstances in which a cost effective view can be selected for materialization. We have also designed the dynamic cost model for dynamic selection of views to materialize and dynamically remove the old materialized views. The general findings of the thesis can be pointed out as follows:

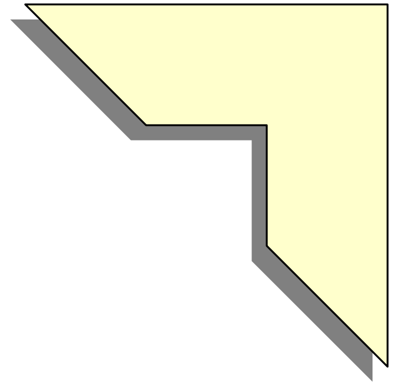
- The methodology evaluates the incremental materialized view maintenance performance by considering the incremental propagation time in comparison with the rematerializing, answering query using materialized view, virtual view and using query rewrite.
- The experiments carried out for the developed methodology infers that:
    - Incremental materialized view maintenance is better than rematerializing a view;
    - Answering a query using materialized views and using query rewrite is beneficial than answering a virtual view.

- The methodology determines the situations of the view materialization by considering the relative costs of answering query in comparison with the materialized view maintenance where materializing a view benefits over the virtual view.
- For the dynamic selection of views for materialization, we have considered factors like - query access frequency, execution time, query selectivity and complexity to calculate the query processing cost and we have considered the table update frequency for the calculation of view maintenance cost as the base table update cost is actually the view maintenance cost to be updated with the base table's changes.
- A weighting factor is calculated based on the query access frequencies and the tables update frequencies to reflect the importance of the queries and the tables.
- The total cost of the query is calculated by adding the query processing cost and the view maintenance cost. Then the queries with higher total cost than that of a minimum total cost are selected for materialization. The most important thing is that the minimum total cost which is the threshold level for view selection is selected dynamically and no previous works have been found on dynamic threshold level for the dynamic selection of views.
- The dynamic selection of views selects queries not only those have higher access frequencies but also queries with higher execution costs are selected for materialization. That means, this model does not select a query for materialization if the access frequency of this query is very high but execution cost is very low. Because as the query can be executed from the base tables directly within a considerable amount of time, the model will not recommend it to materialize where extra maintenance cost will incur and storage space will be occupied for that. Conversely, a query with not much less in access frequency but the execution cost is too high then the model may determine that query for materialization considering other factors like selectivity and complexity.
- Finally, the old materialized views that have low access frequencies are selected for removal from the database to free the storage space for future view materialization to remove the maintenance cost associated with those views. Again, in here, the threshold level is determined dynamically from all of the access frequencies.

5.2 Recommendation for Future Work

The future expansion of this research may explore the following issues:

- Table and materialized view partitioning have not been considered here; partitioning could further improve the overall query performance.
- It was assumed that there is sufficient storage space available for the dynamically selected view materialization. How much storage space is required for the new view materialization and how much storage space is available in the disk can be calculated and based on the available storage space again a minimal subset of views those are higher profitable for materialization can be selected from the already selected subset of views for materialization.
- System's present workload has not been considered. Based on the system workload the higher profitable views for materialization can be selected as a minimal subset from the previous dynamically selected subset of views.
- Indexing not only makes table access faster but also makes faster materialized view access for query partial rewrite; the dynamic selection of materialized view columns and table columns for indexing will make query to execute faster.



REFERENCES

REFERENCES

[1] Codd, E. F., “A relational model of data for large data banks,” Communications of the ACM, vol. 13, no. 6, pp. 377-387, 1970.

[2] *Data Integration Glossary*, U.S. Department of Transportation, August 2001.

[3] Gupta, A., Mumick, I. S., *Materialized views: problems, techniques and applications*, MIT Press, USA, pp. 589, 1999.

[4] Chaudhuri, S., and Dayal, U., “An Overview of Data Warehousing and OLAP Technology,” SIGMOD Record, vol. 26, no. 1, pp. 65-74, 1997.

[5] Chen, S., and Rundensteiner, E. A., “GPIVOT: Efficient Incremental Maintenance of Complex ROLAP Views,” 21st International Conference on Data Engineering (ICDE’05), pp. 552-563, 2005.

[6] Rashid, A. N. M. B., and Islam, M. S., “Role of Materialized View Maintenance with PIVOT and UNPIVOT Operators,” IEEE International Advance Computing Conference (IACC’09), Patiala, India, pp. 951-955, March 6-7, 2009.

[7] Valluri, S. R., Vadapalli, S., and Karlapalem, K., “View Relevance Driven Materialized View Selection in Data Warehousing Environment,” Proceedings of the 13th Australian Database Conference (ADC2002), Melbourne, Australia, vol. 5, pp. 187-196, 2002.

[8] Gupta, A., Mumick, I., and Subrahmanian, V., “Maintaining views incrementally,” Proceedings of SIGMOD, pp. 157–166, 1993.

[9] Griffin, T., and Libkin, L., “Incremental Maintenance of Views with Duplicates,” Proceedings of SIGMOD, pp. 328-339, 1995.

[10] Agrawal, R., Abbadi, A. E., Singh, A., and Yurek, T., “Efficient View Maintenance at Data Warehouses,” Proceedings of SIGMOD, Arizona, USA, pp. 417-427, 1997.

[11] Gluche, D., Grust, T., Mainberger, C., and Scholl, M., “Incremental updates for materialized OQL views,” Proceedings of DOOD, pp. 52–66, 1997.

[12] Zhuge, Y., Molina, H. G., Hammer, J., and Widom, J., “View maintenance in a warehousing environment,” Proceedings of SIGMOD, pp. 316-327, May 1995.

[13] Ali, M. A., Paton, N. W., and Farnandes, A. A. A., "MOVIEW: An incremental maintenance system for materialized object views," Data & Knowledge Engineering, vol. 47, no. 2, pp. 131-166, November 2003.

[14] Lee, K. Y., and Son, J. H., "Reducing cost of accessing relations in incremental view maintenance," Decision Support Systems, vol. 43, no. 2, pp. 512-526, March 2007.

[15] Chen, J., Long, T., and Deng, K., "The consistency of materialized view maintenance and drill-down in a warehousing environment," Proceedings of the 2008 the 9th International Conference for Young Computer Scientist (ICYCS), pp. 1169-1174, 2008.

[16] Lee, K. Y., and Kim, M. H., "Optimizing the incremental maintenance of multiple join views," Proceedings of the 8th ACM International Workshop on Data Warehousing and OLAP, Bremen, Germany, pp. 107-113, 2005.

[17] Surendrababu, B., Reshmy, K. R., and Srivasta, S. K., "Automatic incremental view maintenance in SchemaSQL," Information Technology Journal, vol. 5, no. 2, pp. 314-321, 2006.

[18] Hanson, E. N., "A performance analysis of view materialization strategies," Proceedings of SIGMOD, pp. 440–453, 1987.

[19] Blakeley, J. A., and Martin, N. L., "Join index, materialized view, and hybrid-hash join: a performance analysis," Proceedings of ICDE, pp. 256–263, 1990.

[20] Hull, R., and Zhou, G., "Towards the study of performance trade-offs between materialized and virtual integrated views," Proceedings of VIEWS, pp. 91–102, 1996.

[21] Ashadevi, B., and Balasubramanian R., "Cost effective approach for materialized views selection in data warehousing environment," International Journal of Computer Science and Network Security (IJCSNS), vol. 8, no. 10, pp. 236-242, October 2008.

[22] Harianarayan, V., Rajaraman, A., and Ullman, J., "Implementing data cubes efficiently," Proceedings of ACM SIGMOD 1996 International Conference on Management of Data, Montreal, Canada, pp. 205-216, 1996.

[23] Yang, J., Karlapalem, K., and Li, Q., "A framework for designing materialized views in data warehousing environment," Proceedings of 17th IEEE International Conference on Distributed Comuting Systems, Maryland, USA, pp. 458, May 1997.

[24] Gupta, H., and Mumick, I. S., "Selection of views to materialized in a data warehouse," IEEE Transactions on Knowledge and Data Engineering, vol. 17, no. 1, pp. 24-43, 2005.

[25] Shukla, A., Deshpande, P., and Naughton, J. F., "Materialized view selection for multidimensional datasets," Proceedings of 24th International Conference on Very Large Data Bases, pp. 488-499, 1998.

[26] Gupta, H., and Mumic, I. S., "Selection of views to materialize under a maintenance cost constraint," Proceedings of 7th International Conference on Database Theory (ICDT'99), Jerusalem, Israel, pp. 453-470, 1999.

[27] Zhang, C., and Yang, J., "Genetic algorithm for materialized view selection in data warehouse environments," Proceedings of the International Conference on Data Warehousing and Knowledge Discovery, LNCS, vol. 1676, pp. 116-125, 1999.

[28] Agrawal, S., Chaudhuri, S., and Narasayya, V., "Automated selection of materialized views and indexes in SQL databases," Proceedings of the 26th International Conference on Very Large Data Bases, pp. 496-505, 2000.

[29] Zhang, C., Yao, X., and Yang, J., "An evolutionary approach to materialized view selection in a data warehouse environment," IEEE Transactions on Systems, Man and Cybernetics, vol. 31, no. 3, pp. 282-293, 2001.

[30] Lee, M., and Hammer, J., "Speeding up materialized view selection in data warehouses using a randomized algorithm," International Journal of Cooperative Information Systems, vol. 10, no. 3, pp. 327-353, 2001.

[31] Kalnis, P., Mamoulis, N., and Papdias, D., "View selection using randomized search," Data and Knowledge Engineering, vol. 42, no. 1, pp. 89-111, 2002.

[32] Yu, J. X., Yao, X., Choi, C., and Gou, G., "Materialized view selection as constrained evolutionary optimization," IEEE Transactions on Systems, Man and Cybernetics, part. C, vol. 33, no. 4, pp. 458-467, 2003.

[33] Wang, Z., and Zhang, D., "Optimal genetic view selection algorithm under space constraint," International Journal of Information Technology, vol. 11, no. 5, pp. 44-51, 2005.

[34] Aouiche, K., Jouve, P., and Darmont, J., "Clustering-based materialized view selection in data warehouses," Advances in Databases Information Systems (ADBIS'06), LNCS, vol. 4152, pp. 81-95, 2006.

[35] Ashadevi, B., and Balasubramanian R., "Optimized cost effective approach for selection of materialized views in data warehousing," Journal of Computer Science and Technology, vol. 9, no. 1, pp. 21-26, April 2009.

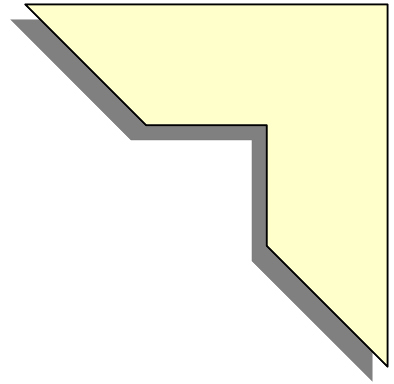
[36] Silberschatz, A., Korth, H. F., and Sudarshan, S., *Database system concepts*. Fourth Edition, McGraw-Hill Companies Inc., New York, USA, chap. 14, pp. 529-562, 2002.

[37] Lane, P., *Oracle® database data warehousing guide 11g release 1 (11.1)*, Part. B28313-02, September, 2007.

[38] Urbano, R., *Oracle® database advanced replication 11g release 1 (11.1)*, Part. B28326-01, July, 2007.

[39] Cunningham, C., Legaria, C. A. G., and Graefe, G., "PIVOT and UNPIVOT: optimization and execution strategies in an RDMS," Proceedings of the 30th VLDB Conference, Toronto, Canada, pp. 998-1009, 2004.

[40] Shafey, M. A. L., *Performance Evaluation of Database Design Approaches for Object Relational Data Management*, M. Sc. Engg. Thesis, Institute of Information and Communication Technology, Bangladesh University of Engineering and Technology, 2008.



APPENDICES

APPENDIX A

DATABASE SCHEMA TABLES WITH COLUMN DEFINITIONS

Table A.1 SALES HISTORY SCHEMA TABLES

a. CHANNELS (small dimension table)

Column Name	Pk	Null?	Data Type	Comments
channel_id	1	N	number	primary key column
channel_desc		N	varchar2(20)	e.g., telesales, internet, catalog
channel_class		N	varchar2(20)	e.g., direct, indirect
channel_class_id		N	number	
channel_total		N	varchar2(13)	
channel_total_id		N	number	

b. COUNTRIES (dimension table)

Column Name	Pk	Null?	Data Type	Comments
country_id	1	N	number	primary key
country_iso_code		N	char(2)	
country_name		N	varchar2(40)	country name
country_subregion		N	varchar2(30)	e.g. Western Europe, to allow hierarchies
country_subregion_id		N	number	
country_region		N	varchar2(20)	e.g. Europe, Asia
country_region_id		N	number	
country_total		N	varchar2(11)	
country_total_id		N	number	
country_name_hist		Y	varchar2(40)	

c. COSTOMERS (dimension table)

Column Name	Pk	Null?	Data Type	Comments
cust_id	1	N	number	primary key
cust_first_name		N	varchar2(20)	first name of the customer
cust_last_name		N	varchar2(40)	last name of the customer

cust_gender		N	char(1)	gender; low cardinality attribute
cust_year_of_birth		N	number(4)	Customer year of birth
cust_marital_status		Y	varchar2(20)	customer marital status
cust_street_address		N	varchar2(40)	Customer street address
cust_postal_code		N	varchar2(10)	postal code of the customer
cust_city		N	varchar2(30)	city where the customer lives
cust_city_id		N	number	
cust_state_province		N	varchar2(40)	customer geography: state or province
cust_state_province_id		N	number	
country_id		N	number	foreign key to the countries table (snowflake)
cust_main_phone_number		N	varchar2(25)	customer main phone number
cust_income_level		Y	varchar2(30)	customer income level
cust_credit_limit		Y	number	customer credit limit
cust_email		Y	varchar2(30)	customer email id
cust_total		N	varchar2(14)	
cust_total_id		N	number	
cust_src_id		Y	number	
cust_eff_from		Y	date	
cust_eff_to		Y	date	
cust_valid		Y	varchar2(1)	

d. PRODUCTS (dimension table)

Column Name	Pk	Null?	Data Type	Comments
prod_id	1	N	number(6)	primary key
prod_name		N	varchar2(50)	product name
prod_desc		N	varchar2(4000)	product description
prod_subcategory		N	varchar2(50)	product subcategory
prod_subcategory_id		N	number	

prod_subcategory_desc		N	varchar2(2000)	product subcategory description
prod_category		N	varchar2(50)	product category
prod_category_id		N	number	
prod_category_desc		N	varchar2(2000)	product category description
prod_weight_class		N	number(3)	product weight class
prod_unit_of_measure		Y	varchar2(20)	product unit of measure
prod_pack_size		N	varchar2(30)	product package size
supplier_id		N	number(6)	this column
prod_status		N	varchar2(20)	product status
prod_list_price		N	number(8,2)	product list price
prod_min_price		N	number(8,2)	product minimum price
prod_total		N	varchar2(13)	
prod_total_id		N	number	
prod_src_id		Y	number	
prod_eff_from		Y	date	
prod_eff_to		Y	date	
prod_valid		Y	varchar2(1)	

e. PROMOTIONS (dimension table)

Column Name	Pk	Null?	Data Type	Comments
promo_id	1	N	number(6)	primary key column
promo_name		N	varchar2(30)	promotion description
promo_subcategory		N	varchar2(30)	investigate promotion hierarchies
promo_subcategory_id		N	number	
promo_category		N	varchar2(30)	promotion category
promo_category_id		N	number	
promo_cost		N	number(10,2)	promotion cost, to do promotion effect calculations
promo_begin_date		N	date	promotion begin day
promo_end_date		N	date	promotion end day
promo_total		N	number(3)	product weight class
promo_total_id		Y	varchar2(20)	product unit of measure

Table A.6 SALES (fact table)

Column Name	Pk	Null?	Data Type	Comments
prod_id		N	number	FK to the products dimension table
cust_id		N	number	FK to the customers dimension table
time_id		N	date	FK to the times dimension table
channel_id		N	number	FK to the channels dimension table
promo_id		N	number	promotion identifier, without FK constraint (intentionally) to show outer join optimization
quantity_sold		N	number(10,2)	product quantity sold with the transaction
amount_sold		N	number(10,2)	invoiced amount to the customer

f. TIMES (dimension table)

Column Name	Pk	Null?	Data Type	Comments
time_id	1	N	date	primary key; day date, finest granularity, CORRECT ORDER
day_name		N	varchar2(9)	Monday to Sunday, repeating
day_number_in_week		N	number(1)	1 to 7, repeating
day_number_in_month		N	number(2)	1 to 31, repeating
calendar_week_number		N	number(2)	1 to 53, repeating
fiscal_week_number		N	number(2)	1 to 53, repeating
week_ending_day		N	date	date of last day in week, CORRECT ORDER
week_ending_day_id		N	number	
calendar_month_number		N	number(2)	1 to 12, repeating
fiscal_month_number		N	number(2)	1 to 12, repeating
calendar_month_desc		N	varchar2(8)	e.g. 1998-01, CORRECT ORDER
calendar_month_id		N	number	
fiscal_month_desc		N	varchar2(8)	e.g. 1998-01, CORRECT ORDER
fiscal_month_id		N	number	
days_in_cal_month		N	number	e.g. 28,31, repeating

days_in_fis_month		N	number	e.g. 25,32, repeating
end_of_cal_month		N	date	last day of calendar month
end_of_fis_month		N	date	last day of fiscal month
calendar_month_name		N	varchar2(9)	January to December, repeating
fiscal_month_name		N	varchar2(9)	January to December, repeating
calendar_quarter_desc		N	char(7)	e.g. 1998-Q1, CORRECT ORDER
calendar_quarter_id		N	number	
fiscal_quarter_desc		N	char(7)	e.g. 1999-Q3, CORRECT ORDER
fiscal_quarter_id		N	number	
days_in_cal_quarter		N	number	e.g. 88,90, repeating
days_in_fis_quarter		N	number	e.g. 88,90, repeating
end_of_cal_quarter		N	date	last day of calendar quarter
end_of_fis_quarter		N	date	last day of fiscal quarter
calendar_quarter_number		N	number(1)	1 to 4, repeating
fiscal_quarter_number		N	number(1)	1 to 4, repeating
calendar_year		N	number(4)	e.g. 1999, CORRECT ORDER
calendar_year_id		N	number	
fiscal_year		N	number(4)	e.g. 1999, CORRECT ORDER
fiscal_year_id		N	number	
days_in_cal_year		N	number	365,366 repeating
days_in_fis_year		N	number	e.g. 355,364, repeating
end_of_cal_year		N	date	last day of cal year
end_of_fis_year		N	date	last day of fiscal year

Table A.2 ORDER ENTRY SCHEMA TABLES

a. COSTOMERS

Column Name	Pk	Null?	Data Type	Comments
customer_id	1	N	number(6)	primary key
cust_first_name		N	varchar2(20)	first name of the customer
cust_last_name		N	varchar2(20)	last name of the customer
Gender		Y	varchar2(1)	gender
date_of_birth		Y	Date	customer date of birth

cust_marital_status		Y	varchar2(20)	customer marital status; low cardinality attribute
cust_address		Y	CUST_ADDRESS _TYP	customer address
phone_numbers		Y	PHONE_LIST _TYP	customer phone numbers
income_level		Y	varchar2(20)	customer income level
credit_limit		Y	number(9,2)	customer credit limit
cust_email		Y	varchar2(30)	customer email id
nls_language		Y	varchar2(3)	
nls_territory		Y	varchar2(30)	
account_mgr_id		Y	number(6)	
cust_geo_location		Y	SDO_GEOMETRY	

b. COUNTRIES

Column Name	Pk	Null?	Data Type	Comments
country_id	1	N	char(2)	primary key
country_name		Y	varchar2(40)	name of the country
region_id		Y	number	

c. INVENTORIES

Column Name	Pk	Null?	Data Type	Comments
product_id	1	N	number(6)	primary key
warehouse_id	2	N	number(3)	primary key
quantity_on_hand		N	number(8)	

d. WAREHOUSES

Column Name	Pk	Null?	Data Type	Comments
warehouse_id	1	N	number(3)	primary key
warehouse_spec		Y	XMLTYPE	
warehouse_name		Y	varchar2(35)	
location_id		Y	number(4)	
wh_geo_location		Y	SDO_GEOMETRY	

e. ORDER_ITEMS

Column Name	Pk	Null?	Data Type	Comments
order_id	1	N	number(12)	primary key
line_item_id	2	N	number(3)	
product_id		N	number(6)	
unit_price		Y	number(8,2)	
Quantity		Y	number(8)	

f. ORDERS

Column Name	Pk	Null?	Data Type	Comments
order_id	1	N	number(12)	primary key
order_date		N	timestamp(6) with local time zone	
order_mode		Y	varchar2(8)	
customer_id		N	number(6)	
order_status		Y	number(2)	
order_total		Y	number(8,2)	
sales_rep_id		Y	number(6)	
promotion_id		Y	number(6)	

g. PRODUCT_INFORMATION

Column Name	Pk	Null?	Data Type	Comments
product_id	1	N	number(6)	primary key
product_name		Y	varchar2(50)	
product_description		Y	varchar2(2000)	
category_id		Y	number(2)	
weight_class		Y	number(1)	
warranty_period		Y	interval year(2) to month	
supplier_id		Y	number(6)	

product_status		Y	varchar2(20)	
list_price		Y	number(8,2)	
min_price		Y	number(8,2)	
catalog_url		Y	varchar2(50)	

h. PRODUCT_DESCRIPTION

Column Name	Pk	Null?	Data Type	Comments
product_id	1	N	number(6)	primary key
language_id	2	Y	varchar2(3)	primary key
translated_name		N	nvarchar2(50)	
translated_description		N	nvarchar2(2000)	

A P P E N D I X B

QUERIES FOR INCREMENTAL MAINTENANCE PERFORMANCE EVALUATION

Table B.1 List of the queries used for selectivity experiments

Selectivity	Query Statements
0.2	***Joins Only Query:*** *Query:* **Find the daily total sales and amount of each product and for each customer in the store whose customer id is less than or equal to 400.** ***SQL:*** *SELECT cn.country_name country, p.prod_name prod, t.calendar_year year, s.amount_sold sale FROM sales s, times t, customers cs, countries cn, products p WHERE s.time_id = t.time_id AND s.prod_id = p.prod_id AND s.cust_id = cs.cust_id AND cs.country_id = cn.country_id AND cs.cust_id <= 400;* ***Aggregate Query:*** *Query:* **Find the total sales amount and number of sales amount with respect to the channel used and for each product and calendar month of the customers whose customer id is less than or equal to 400.** ***SQL:*** *SELECT ch.channel_desc, t.calendar_month_desc, cn.country_iso_code, p.prod_name, SUM(s.amount_sold) SALES$, count(s.amount_sold) total FROM sales s, customers c, times t, channels ch, countries cn, products p WHERE s.time_id=t.time_id AND s.cust_id=c.cust_id AND s.channel_id= ch.channel_id AND s.prod_id=p.prod_id AND c.country_id=cn.country_id AND c.cust_id <= 400 GROUP BY ch.channel_desc, t.calendar_month_desc, cn.country_iso_code, p.prod_name;*
0.4	***Joins Only Query:*** *Query:* **Find the daily total sales and amount of each product and for each customer in the store whose customer id is less than or equal to 800.**

	SQL: *SELECT cn.country_name country, p.prod_name prod, t.calendar_year year, s.amount_sold sale FROM sales s, times t, customers cs, countries cn, products p WHERE s.time_id = t.time_id AND s.prod_id = p.prod_id AND s.cust_id = cs.cust_id AND cs.country_id = cn.country_id AND cs.cust_id <= 800;* ***<u>Aggregate Query:</u>*** ***Query:*** **Find the total sales amount and number of sales amount with respect to the channel used and for each product and calendar month of the customers whose customer id is less than or equal to 800.** ***SQL:*** *SELECT ch.channel_desc, t.calendar_month_desc, cn.country_iso_code, p.prod_name, SUM(s.amount_sold) SALES$, count(s.amount_sold) total FROM sales s, customers c, times t, channels ch, countries cn, products p WHERE s.time_id=t.time_id AND s.cust_id=c.cust_id AND s.channel_id= ch.channel_id AND s.prod_id=p.prod_id AND c.country_id=cn.country_id AND c.cust_id <= 800 GROUP BY ch.channel_desc, t.calendar_month_desc, cn.country_iso_code, p.prod_name;*
0.6	***<u>Joins Only Query:</u>*** ***Query:*** **Find the daily total sales and amount of each product and for each customer in the store whose customer id is less than or equal to 1200.** ***SQL:*** *SELECT cn.country_name country, p.prod_name prod, t.calendar_year year, s.amount_sold sale FROM sales s, times t, customers cs, countries cn, products p WHERE s.time_id = t.time_id AND s.prod_id = p.prod_id AND s.cust_id = cs.cust_id AND cs.country_id = cn.country_id AND cs.cust_id <= 1200;* ***<u>Aggregate Query:</u>*** ***Query:*** **Find the total sales amount and number of sales amount with respect to the channel used and for each product and calendar month of the customers whose customer id is less than or equal to 1200.**

	SQL: *SELECT ch.channel_desc, t.calendar_month_desc, cn.country_iso_code, p.prod_name, SUM(s.amount_sold) SALES$, count(s.amount_sold) total FROM sales s, customers c, times t, channels ch, countries cn, products p WHERE s.time_id=t.time_id AND s.cust_id=c.cust_id AND s.channel_id= ch.channel_id AND s.prod_id=p.prod_id AND c.country_id=cn.country_id AND c.cust_id <= 1200 GROUP BY ch.channel_desc, t.calendar_month_desc, cn.country_iso_code, p.prod_name;*
0.8	***<u>Joins Only Query:</u>*** ***Query:*** **Find the daily total sales and amount of each product and for each customer in the store whose customer id is less than or equal to 1600.** ***SQL:*** *SELECT cn.country_name country, p.prod_name prod, t.calendar_year year, s.amount_sold sale FROM sales s, times t, customers cs, countries cn, products p WHERE s.time_id = t.time_id AND s.prod_id = p.prod_id AND s.cust_id = cs.cust_id AND cs.country_id = cn.country_id AND cs.cust_id <= 1600;* ***<u>Aggregate Query:</u>*** ***Query:*** **Find the total sales amount and number of sales amount with respect to the channel used and for each product and calendar month of the customers whose customer id is less than or equal to 1600.** ***SQL:*** *SELECT ch.channel_desc, t.calendar_month_desc, cn.country_iso_code, p.prod_name, SUM(s.amount_sold) SALES$, count(s.amount_sold) total FROM sales s, customers c, times t, channels ch, countries cn, products p WHERE s.time_id=t.time_id AND s.cust_id=c.cust_id AND s.channel_id= ch.channel_id AND s.prod_id=p.prod_id AND c.country_id=cn.country_id AND c.cust_id <= 1600 GROUP BY ch.channel_desc, t.calendar_month_desc, cn.country_iso_code, p.prod_name;*
1.0	***<u>Joins Only Query:</u>*** ***Query:*** **Find the daily total sales and amount of each product and**

	for each customer in the store whose customer id is less than or equal to 2000. ***SQL:*** *SELECT cn.country_name country, p.prod_name prod, t.calendar_year year, s.amount_sold sale FROM sales s, times t, customers cs, countries cn, products p WHERE s.time_id = t.time_id AND s.prod_id = p.prod_id AND s.cust_id = cs.cust_id AND cs.country_id = cn.country_id AND cs.cust_id <= 2000;* ***<u>Aggregate Query:</u>*** *Query:* **Find the total sales amount and number of sales amount with respect to the channel used and for each product and calendar month of the customers whose customer id is less than or equal to 2000.** ***SQL:*** *SELECT ch.channel_desc, t.calendar_month_desc, cn.country_iso_code, p.prod_name, SUM(s.amount_sold) SALES$, count(s.amount_sold) total FROM sales s, customers c, times t, channels ch, countries cn, products p WHERE s.time_id=t.time_id AND s.cust_id=c.cust_id AND s.channel_id= ch.channel_id AND s.prod_id=p.prod_id AND c.country_id=cn.country_id AND c.cust_id <= 1600 GROUP BY ch.channel_desc, t.calendar_month_desc, cn.country_iso_code, p.prod_name;*

Table B.2 List of the queries used for complexity experiments

Complexity	Query Statements
CV_1	***<u>Joins Only Query:</u>*** *Query:* **Find the daily total sales amount.** ***SQL:*** *SELECT t.calendar_year year, s.amount_sold sale FROM sales s, times t WHERE s.time_id = t.time_id;* ***<u>Aggregate Query:</u>*** *Query:* **Find the total sales amount and number of sales in each calendar month.** ***SQL:*** *SELECT t.calendar_month_desc, SUM(s.amount_sold) SALES$, count (s.amount_sold) total FROM sales s, times t WHERE s.time_id=t.time_id GROUP BY t.calendar_month_desc;*

CV_2	***Joins Only Query:*** ***Query:*** **Find the daily total sales amount for each customer.** ***SQL:*** *SELECT t.calendar_year year, s.amount_sold sale FROM sales s, times t, customers cs WHERE s.time_id = t.time_id AND s.cust_id = cs.cust_id;* ***Aggregate Query:*** ***Query:*** **Find the total sales amount and number of sales for every customer in each calendar month.** ***SQL:*** *SELECT t.calendar_month_desc, c.cust_id, SUM(s.amount_sold) SALES$, count (s.amount_sold) total FROM sales s, times t, customers c WHERE s.time_id=t.time_id AND s.cust_id=c.cust_id GROUP BY t.calendar_month_desc, c.cust_id;*
CV_3	***Joins Only Query:*** ***Query:*** **Find the daily total sales amount for each customer with their country name.** ***SQL:*** *SELECT cn.country_name country, t.calendar_year year, s.amount_sold sale FROM sales s, times t, customers cs, countries cn WHERE s.time_id = t.time_id AND s.cust_id = cs.cust_id AND cs.country_id = cn.country_id;* ***Aggregate Query:*** ***Query:*** **Find the total sales amount and number of sales for every customer and for every country international standard code in each calendar month.** ***SQL:*** *SELECT t.calendar_month_desc, c.cust_id, cn.country_iso_code, SUM(s.amount_sold) SALES$, count (s.amount_sold) total FROM sales s, times t, customers c, countries cn WHERE s.time_id=t.time_id AND s.cust_id=c.cust_id AND c.country_id=cn.country_id GROUP BY t.calendar_month_desc, c.cust_id, cn.country_iso_code;*
CV_4	***Joins Only Query:*** ***Query:*** **Find the daily total sales amount of each product and for each customer with their country name.**

	SQL: *SELECT cn.country_name country, p.prod_name prod, t.calendar_year year, s.amount_sold sale FROM sales s, times t, customers cs, countries cn, products p WHERE s.time_id = t.time_id AND s.prod_id = p.prod_id AND s.cust_id = cs.cust_id AND cs.country_id = cn.country_id;* ***Aggregate Query:*** ***Query:*** **Find the total sales amount and number of sales for every customer and for every country international standard code in each calendar month and in each channel.** ***SQL:*** *SELECT t.calendar_month_desc, ch.channel_desc, c.cust_id, cn.country_iso_code, SUM(s.amount_sold) SALES$, count (s.amount_sold) total FROM sales s, times t, customers c, countries cn, channels ch WHERE s.time_id=t.time_id AND s.cust_id=c.cust_id AND c.country_id=cn.country_id AND s.channel_id=ch.channel_id GROUP BY t.calendar_month_desc, ch.channel_desc, c.cust_id, cn.country_iso_code;*
CV_5	***Joins Only Query:*** ***Query:*** **Find the daily total sales amount of each product and for each customer with their country name and channel of sales.** ***SQL:*** *SELECT cn.country_name country, p.prod_name prod, t.calendar_year year, ch.channel_desc, s.amount_sold sale FROM sales s, times t, customers cs, countries cn, products p, channels ch WHERE s.time_id = t.time_id AND s.prod_id = p.prod_id AND s.cust_id = cs.cust_id AND cs.country_id = cn.country_id AND s.channel_id = ch.channel_id;* ***Aggregate Query:*** ***Query:*** **Find the total sales amount and number of sales for every customer and for every country international standard code in each calendar month and in each channel and in product.** ***SQL:*** *SELECT t.calendar_month_desc, ch.channel_desc, c.cust_id, cn.country_iso_code, p.prod_name, SUM(s.amount_sold) SALES$, count (s.amount_sold) total FROM sales s, times t, customers c,*

	countries cn, channels ch, products p WHERE s.time_id=t.time_id AND s.cust_id=c.cust_id AND c.country_id=cn.country_id AND s.channel_id=ch.channel_id AND s.prod_id=p.prod_id GROUP BY t.calendar_month_desc, ch.channel_desc, c.cust_id, cn.country_iso_code, p.prod_name;

Table B.3 List of the queries used for database size experiments

Database Size	**Query Statements**
DSizeView	***Joins Only Query:*** ***Query:*** **Find the daily total sales amount of each product and for each customer with their country name.** ***SQL:*** *SELECT cn.country_name country, p.prod_name prod, t.calendar_year year, ch.channel_desc, s.amount_sold sale FROM sales s, times t, customers cs, countries cn, products p, channels ch WHERE s.time_id = t.time_id AND s.prod_id = p.prod_id AND s.cust_id = cs.cust_id AND cs.country_id = cn.country_id AND s.channel_id = ch.channel_id;* ***Aggregate Query:*** ***Query:*** **Find the total sales amount and number of sales amount with respect to the channel used and for each product and calendar month of each customer.** ***SQL:*** *SELECT ch.channel_desc, t.calendar_month_desc, cn.country_iso_code, p.prod_name, SUM(s.amount_sold) SALES$, count(s.amount_sold) total FROM sales s, customers c, times t, channels ch, countries cn, products p WHERE s.time_id=t.time_id AND s.cust_id=c.cust_id AND s.channel_id= ch.channel_id AND s.prod_id=p.prod_id AND c.country_id=cn.country_id GROUP BY ch.channel_desc, t.calendar_month_desc, cn.country_iso_code, p.prod_name;*

A P P E N D I X   C

QUERIES FOR DYNAMIC SELECTION OF VIEWS

Table C.1 List of the queries used for dynamic selection of views in experiment 01

Query No.	Query Statements
Q_1	***Query:*** **Find the daily total sales and amount of each product in the store.** ***SQL:*** *SELECT s.prod_id, s.time_id, COUNT(*) AS count_grp, SUM(s.amount_sold) AS sum_dollar_sales, COUNT(s.amount_sold) AS count_dollar_sales, SUM(s.quantity_sold) AS sum_quantity_sales, COUNT(s.quantity_sold) AS count_quantity_sales FROM sales s GROUP BY s.prod_id, s.time_id;*
Q_2	***Query:*** **Find the daily sales quantity and amount of the customer including those customers that don't buy any item and the days in which there is no sale.** ***SQL:*** *SELECT s.rowid "sales_rid", t.rowid "times_rid", c.rowid "customers_rid", c.cust_id, c.cust_last_name, s.amount_sold, s.quantity_sold, s.time_id FROM sales s, times t, customers c WHERE s.cust_id = c.cust_id(+) AND s.time_id = t.time_id(+);*
Q_3	***Query:*** **Find the sales information of the customers "Smith" and "Brown" and marking differently their purchase.** ***SQL:*** *(SELECT c.rowid crid, s.rowid srid, c.cust_id, s.amount_sold, 1 marker FROM sales s, customers c WHERE s.cust_id = c.cust_id AND c.cust_last_name = 'Smith') UNION ALL (SELECT c.rowid crid, s.rowid srid, c.cust_id, s.amount_sold, 2 marker FROM sales s, customers c WHERE s.cust_id = c.cust_id AND c.cust_last_name = 'Brown');*
Q_4	***Query:*** **Find the yearly, quarterly and daily total sales amount and number of sales.** ***SQL:*** *(SELECT 'Year' umarker, NULL, NULL, t.fiscal_year, SUM(s.amount_sold) amt, COUNT(s.amount_sold), COUNT(*) FROM sales s, times t WHERE s.time_id = t.time_id GROUP BY t.fiscal_year) UNION ALL (SELECT 'Quarter' umarker, NULL, NULL,*

	t.fiscal_quarter_number, SUM(s.amount_sold) amt, COUNT (s.amount_sold), COUNT(*) FROM sales s, times t WHERE s.time_id = t.time_id and t.fiscal_year = 2001 GROUP BY t.fiscal_quarter_number) UNION ALL (SELECT 'Daily' umarker, s.rowid rid, t.rowid rid2, t.day_number_in_week, s.amount_sold amt, 1,1 FROM sales s, times t WHERE s.time_id = t.time_id AND t.time_id between '01-Jan-01' AND '01-Dec-31');
Q_5	***Query:* Find the country, product and time wise total sales and amount of each product in the store.** ***SQL:*** *SELECT country_name country, prod_name prod, calendar_year year, SUM(amount_sold) sale, COUNT(amount_sold) cnt, COUNT(*) cntstr FROM sales, times, customers, countries, products WHERE sales.time_id = times.time_id AND sales.prod_id = products.prod_id AND sales.cust_id = customers.cust_id AND customers.country_id = countries.country_id GROUP BY country_name, prod_name, calendar_year;*
Q_6	***Query:* Find the total sales amount and number of sales of each product and each customer for a particular time period.** ***SQL:*** *SELECT s.time_id, s.cust_id, s.prod_id, p.prod_weight_class, SUM(amount_sold) AS sum_amount_sold, SUM(quantity_sold) AS sum_quantity_sold FROM sales s, products p WHERE s.prod_id = p.prod_id AND s.time_id = TRUNC(SYSDATE-3000) GROUP BY s.time_id, s.cust_id, s.prod_id, p.prod_weight_class;*
Q_7	***Query:* Find the monthly product wise total sales amount and number of sales quantity.** ***SQL:*** *SELECT s.time_id, s.prod_id, SUM(s.quantity_sold), SUM (s.amount_sold), p.prod_name, t.calendar_month_name, COUNT(*), COUNT(s.quantity_sold), COUNT(s.amount_sold) FROM sales s, products p, times t WHERE s.time_id = t.time_id AND s.prod_id = p.prod_id GROUP BY t.calendar_month_name, s.prod_id, p.prod_name, s.time_id;*

Q_8	***Query:*** **Find the weekend product sub category wise total sales amount and quantity.** ***SQL:*** *SELECT p.prod_subcategory, t.week_ending_day, SUM (s.amount_sold) AS sum_amount_sold FROM sales s, products p, times t WHERE s.time_id=t.time_id AND s.prod_id=p.prod_id GROUP BY p.prod_subcategory, t.week_ending_day;*
Q_9	***Query:*** **Find the weekend product and customer wise total sales amount.** ***SQL:*** *SELECT p.prod_id, t.week_ending_day, s.cust_id, SUM (s.amount_sold) AS sum_amount_sold FROM sales s, products p, times t WHERE s.time_id=t.time_id AND s.prod_id=p.prod_id GROUP BY p.prod_id, t.week_ending_day, s.cust_id;*
Q_{10}	***Query:*** **Find the monthly city and product wise total sales amount and quantity.** ***SQL:*** *SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city, SUM(s.amount_sold) AS sum_amount_sold, COUNT(s.amount_sold) AS count_amount_sold FROM sales s, products p, times t, customers c WHERE s.time_id=t.time_id AND s.prod_id=p.prod_id AND s.cust_id =c.cust_id GROUP BY p.prod_subcategory, t.calendar_month_desc, c.cust_city;*
Q_{11}	***Query:*** **Find the daily sales history of the products including those products that have not been sold.** ***SQL:*** *SELECT p.prod_id, p.prod_name, t.time_id, t.week_ending_day, s.channel_id, s.promo_id, s.cust_id, s.amount_sold FROM sales s, products p, times t WHERE s.time_id=t.time_id AND s.prod_id =p.prod_id(+);*
Q_{12}	***Query:*** **Find the weekend product and city wise sales total amount.** ***SQL:*** *SELECT p.prod_name, t.week_ending_day, c.cust_city, SUM (s.amount_sold) FROM sales s, products p, times t, customers c WHERE s.time_id=t.time_id AND s.prod_id = p.prod_id AND s.cust_id = c.cust_id GROUP BY p.prod_name, t.week_ending_day, c.cust_city;*

Q_{13}	***Query:*** **Find the weekend total sales amount for the month of August 1999.** ***SQL:*** *SELECT t.week_ending_day, SUM(s.amount_sold) FROM sales s, times t WHERE s.time_id = t.time_id AND t.week_ending_day BETWEEN TO_DATE ('01-AUG-1999', 'DD-MON-YYYY') AND TO_DATE('10-AUG-1999', 'DD-MON-YYYY') GROUP BY week_ending_day;*
Q_{14}	***Query:*** **Find the total sales amount and quantity based on product subcategory and city for each month.** ***SQL:*** *SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city, SUM(s.amount_sold) AS sum_amount_sold, COUNT(s.amount_sold) AS count_amount_sold FROM sales s, products p, times t, customers c WHERE s.time_id=t.time_id AND s.prod_id=p.prod_id AND s.cust_id =c.cust_id GROUP BY p.prod_subcategory, t.calendar_month_desc, c.cust_city;*
Q_{15}	***Query:*** **Find the total sales amount and quantity based on product subcategory and city for each month.** ***SQL:*** *SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city, AVG(s.amount_sold) FROM sales s, products p, times t, customers c WHERE s.time_id=t.time_id AND s.prod_id=p.prod_id AND s.cust_id =c.cust_id GROUP BY p.prod_subcategory, t.calendar_month_desc, c.cust_city;*
Q_{16}	***Query:*** **Find the sales history of the country USA, Argentina, Japan, India, France, Spain and Ireland.** ***SQL:*** *SELECT t.calendar_year, t.calendar_month_number, t.day_number_in_month, c1.country_name, s.prod_id, s.quantity_sold, s.amount_sold FROM times t, countries c1, sales s, customers c2 WHERE s.time_id = t.time_id and s.cust_id = c2.cust_id and c2.country_id = c1.country_id and c1.country_name IN ('United States of America', 'Argentina', 'Japan', 'India', 'France', 'Spain', 'Ireland');*
Q_{17}	***Query:*** **Find the total sales amount by grouping first product subcategory and month and then customer city and product sub**

	category wise. ***SQL:*** *SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city, SUM(s.amount_sold) AS sum_amount_sold FROM sales s, customers c, products p, times t WHERE s.time_id=t.time_id AND s.prod_id = p.prod_id AND s.cust_id = c.cust_id GROUP BY GROUPING SETS ((p.prod_subcategory, t.calendar_month_desc), (c.cust_city, p.prod_subcategory));*
Q_{18}	***Query:*** **Find the total sales amount by grouping product, subcategory, state, city wise.** ***SQL:*** *SELECT p.prod_category, p.prod_subcategory, c.cust_state_province, c.cust_city, GROUPING_ID(p.prod_category, p.prod_subcategory, c.cust_state_province,c.cust_city) AS gid, SUM(s.amount_sold) AS sum_amount_sold FROM sales s, products p, customers c WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id GROUP BY GROUPING SETS ((p.prod_category, p.prod_subcategory, c.cust_city), (p.prod_category, p.prod_subcategory, c.cust_state_province, c.cust_city), (p.prod_category, p.prod_subcategory));*
Q_{19}	***Query:*** **Find the product, month wise, monthly and product sub category wise total sales amount and then combine all the results together.** ***SQL:*** *SELECT null, p.prod_subcategory, null, t.calendar_month_desc, SUM(s.amount_sold) AS sum_amount_sold FROM sales s, products p, customers c, times t WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id GROUP BY p.prod_subcategory, t.calendar_month_desc UNION ALL SELECT null, null, null, t.calendar_month_desc, SUM(s.amount_sold) AS sum_amount_sold FROM sales s, products p, customers c, times t WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id GROUP BY t.calendar_month_desc UNION ALL SELECT p.prod_category, p.prod_subcategory, c.cust_state_province, null, SUM(s.amount_sold) AS sum_amount_sold FROM sales s, products p, customers c, times t WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id GROUP BY p.prod_category, p.prod_subcategory,*

	c.cust_state_province UNION ALL SELECT p.prod_category, p.prod_subcategory, null, null, SUM(s.amount_sold) AS sum_amount_sold FROM sales s, products p, customers c, times t WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id GROUP BY p.prod_category, p.prod_subcategory;
Q_{20}	***Query:*** **Find the total sales amount for each channel, customer city and for each quarter.** ***SQL:*** *SELECT ch.channel_class, c.cust_city, t.calendar_quarter_desc, SUM(s.amount_sold) sales_amount FROM sales s, times t, customers c, channels ch WHERE s.time_id = t.time_id AND s.cust_id = c.cust_id AND s.channel_id = ch.channel_id GROUP BY ch.channel_class, c.cust_city, t.calendar_quarter_desc;*
Q_{21}	***Query:*** **Find the channel and country standard code wise total sales amount and compute the sales amount for each channel and sales for all channels.** ***SQL:*** *SELECT channels.channel_desc, countries.country_iso_code, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$ FROM sales, customers, times, channels, countries WHERE sales.time_id =times.time_id AND sales.cust_id=customers.cust_id AND sales.channel_id= channels.channel_id AND customers.country_id =countries.country_id GROUP BY CUBE(channels.channel_desc, countries.country_iso_code);*
Q_{22}	***Query:*** **Find the channel and country standard code wise total sales amount and compute the sales amount for each channel.** ***SQL:*** *SELECT channels.channel_desc, calendar_month_desc, countries.country_iso_code, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$ FROM sales, customers, times, channels, countries WHERE sales.time_id=times.time_id AND sales.cust_id =customers.cust_id AND customers.country_id = countries.country_id AND sales.channel_id = channels.channel_id GROUP BY ROLLUP (channels.channel_desc, calendar_month_desc, countries.country_iso_code);*

Q_{23}	***Query:*** **Find the channel wise and monthly and country standard code wise total sales amount and compute the sales amount for each channel.** ***SQL:*** *SELECT channel_desc, calendar_month_desc, countries.country_iso_code, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$ FROM sales, customers, times, channels, countries WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND customers.country_id = countries.country_id AND sales.channel_id= channels.channel_id GROUP BY channel_desc, ROLLUP(calendar_month_desc, countries.country_iso_code);*
Q_{24}	***Query:*** **Find the channel, month and country standard code wise total sales amount and compute the sales amount for each channel and the total for all channels.** ***SQL:*** *SELECT channel_desc, calendar_month_desc, countries.country_iso_code, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$ FROM sales, customers, times, channels, countries WHERE sales.time_id=times.time_id AND sales.cust_id =customers.cust_id AND sales.channel_id= channels.channel_id AND customers.country_id = countries.country_id GROUP BY CUBE (channel_desc, calendar_month_desc, countries.country_iso_code);*
Q_{25}	***Query:*** **Find the channel, month and country standard code wise total sales amount and compute the sales amount for each channel and the total for all channels.** ***SQL:*** *SELECT channel_desc, calendar_month_desc, countries.country_iso_code, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$ FROM sales, customers, times, channels, countries WHERE sales.time_id = times.time_id AND sales.cust_id = customers.cust_id AND customers.country_id=countries.country_id AND sales.channel_id = channels.channel_id GROUP BY channel_desc, CUBE(calendar_month_desc, countries.country_iso_code);*

Q_{26}	***Query:*** **Find the total sales amount of each of the channel, month and country standard code group category.** ***SQL:*** *SELECT channel_desc, calendar_month_desc, country_iso_code, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$, GROUPING(channel_desc) AS Ch, GROUPING(calendar_month_desc) AS Mo, GROUPING(country_iso_code) AS Co FROM sales, customers, times, channels, countries WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND customers.country_id = countries.country_id AND sales.channel_id= channels.channel_id GROUP BY ROLLUP(channel_desc, calendar_month_desc, countries.country_iso_code);*
Q_{27}	***Query:*** **Find the total sales amount of each channel and country standard code wise details and computing channel 1 for multi channel and country standard code 1 for multi country.** ***SQL:*** *SELECT DECODE(GROUPING(channel_desc), 1, 'Multi-channel sum', channel_desc) AS Channel, DECODE (GROUPING (country_iso_code), 1, 'Multi-country sum', country_iso_code) AS Country, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$ FROM sales, customers, times, channels, countries WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND customers.country_id = countries.country_id AND sales.channel_id= channels.channel_id GROUP BY CUBE(channel_desc, country_iso_code);*
Q_{28}	***Query:*** **Find the total sales amount of each channel, month and country standard code of those groups for which the channel, country standard code or the monthly grouping is 1.** ***SQL:*** *SELECT channel_desc, calendar_month_desc, country_iso_code, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$, GROUPING(channel_desc) CH, GROUPING (calendar_month_desc) MO, GROUPING(country_iso_code) CO FROM sales, customers, times, channels, countries WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND customers.country_id = countries.country_id AND sales.channel_id= channels.channel_id*

	GROUP BY CUBE(channel_desc, calendar_month_desc, country_iso_code) HAVING (GROUPING(channel_desc)=1 AND GROUPING(calendar_month_desc)= 1 AND GROUPING (country_iso_code)=1) OR (GROUPING(channel_desc)=1 AND GROUPING (calendar_month_desc)= 1) OR (GROUPING (country_iso_code)=1 AND GROUPING(calendar_month_desc)= 1);
Q_{29}	***Query:*** **Find the summation of the sales of the group set country standard code and customer state province.** ***SQL:*** *SELECT country_iso_code, SUBSTR(cust_state_province,1,12), SUM(amount_sold), GROUPING_ID(country_iso_code, cust_state_province) GROUPING_ID, GROUP_ID() FROM sales, customers, times, countries WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND customers.country_id= countries.country_id AND times.time_id= '30-OCT-00' AND country_iso_code IN ('FR', 'ES') GROUP BY GROUPING SETS (country_iso_code, ROLLUP(country_iso_code, cust_state_province));*
Q_{30}	***Query:*** **Find the all sales amount of the groups (channel, month, country standard code), (channel, country standard code) and (month, country standard code).** ***SQL:*** *SELECT channel_desc, calendar_month_desc, country_iso_code, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$ FROM sales, customers, times, channels, countries WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND sales.channel_id= channels.channel_id GROUP BY GROUPING SETS ((channel_desc, calendar_month_desc, country_iso_code), (channel_desc, country_iso_code), (calendar_month_desc, country_iso_code));*
Q_{31}	***Query:*** **Find the all sales amount of the channels, months and country standard code wise customer having channels, months and country standard code groupings is equal to 0, 2 or 4.** ***SQL:*** *SELECT channel_desc, calendar_month_desc, country_iso_code, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$, GROUPING_ID (channel_desc, calendar_month_desc,*

	country_iso_code) gid FROM sales, customers, times, channels, countries WHERE sales.time_id=times.time_id AND sales.cust_id =customers.cust_id AND sales.channel_id= channels.channel_id GROUP BY CUBE(channel_desc, calendar_month_desc, country_iso_code) HAVING GROUPING_ID(channel_desc, calendar_month_desc, country_iso_code)=0 OR GROUPING_ID (channel_desc, calendar_month_desc, country_iso_code)=2 OR GROUPING_ID(channel_desc, calendar_month_desc, country_iso_code)=4;
Q_{32}	***Query:*** **Find the channels, months and country standard code wise total sales and find channels, months and country standard code wise total amount.** ***SQL:*** *SELECT channel_desc, calendar_month_desc, country_iso_code, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$ FROM sales, customers, times, channels, countries WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND customers.country_id = countries.country_id AND sales.channel_id= channels.channel_id GROUP BY ROLLUP(channel_desc, calendar_month_desc, country_iso_code);*
Q_{33}	***Query:*** **Find the channels, months and country standard code wise total sales and find channels total amount.** ***SQL:*** *SELECT channel_desc, calendar_month_desc, country_iso_code, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$ FROM sales, customers, times, channels, countries WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND sales.channel_id= channels.channel_id GROUP BY ROLLUP(channel_desc, (calendar_month_desc, country_iso_code));*
Q_{34}	***Query:*** **Find the total sales amount of each channel and of the grouping sets (year, month) and (country standard code and province).** ***SQL:*** *SELECT channel_desc, calendar_year, calendar_quarter_desc, country_iso_code, cust_state_province, TO_CHAR(SUM (amount_sold), '9,999,999,999') SALES$ FROM sales, customers, times, channels,*

	countries WHERE sales.time_id = times.time_id AND sales.cust_id = customers.cust_id AND sales.channel_id = channels.channel_id AND countries.country_id = customers.country_id GROUP BY channel_desc, GROUPING SETS (ROLLUP (calendar_year, calendar_quarter_desc), ROLLUP(country_iso_code, cust_state_province));
Q_{35}	***Query:*** **Find the total sales amount of the grouping sets (country standard code, province) and (year, quarter).** ***SQL:*** *SELECT country_iso_code, cust_state_province, calendar_year, calendar_quarter_desc, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$ FROM sales, customers, times, channels, countries WHERE sales.time_id=times.time_id AND sales.cust_id =customers.cust_id AND countries.country_id =customers.country_id AND sales.channel_id= channels.channel_id GROUP BY GROUPING SETS (country_iso_code, cust_state_province), GROUPING SETS (calendar_year, calendar_quarter_desc);*
Q_{36}	***Query:*** **Find the total sales amount of each the (year, quarter, month), (region, sub-region, standard code, province, city) and (product category, subcategory and product name) sub totals.** ***SQL:*** *SELECT calendar_year, calendar_quarter_desc, calendar_month_desc, country_region, country_subregion, countries.country_iso_code, cust_state_province, cust_city, prod_category_desc, prod_subcategory_desc, prod_name, TO_CHAR (SUM (amount_sold), '9,999,999,999') SALES$ FROM sales, customers, times, channels, countries, products WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND sales.channel_id= channels.channel_id AND sales.prod_id= products.prod_id AND customers.country_id=countries.country_id GROUP BY ROLLUP(calendar_year, calendar_quarter_desc, calendar_month_desc), ROLLUP(country_region, country_subregion, countries.country_iso_code, cust_state_province, cust_city), ROLLUP(prod_category_desc, prod_subcategory_desc, prod_name);*
Q_{37}	***Query:*** **Find the total sales amount of the year, quarter and month wise subtotals.**

	SQL: *SELECT calendar_year, calendar_quarter_number, calendar_month_number, SUM(amount_sold) FROM sales, times, products, customers, countries WHERE sales.time_id=times.time_id AND sales.prod_id=products.prod_id AND customers.country_id = countries.country_id AND sales.cust_id=customers.cust_id AND calendar_year=1999 GROUP BY ROLLUP(calendar_year, calendar_quarter_number, calendar_month_number);*
Q_{38}	***Query:*** **Find the total sales amount and quantity of each product name, country name, and channel and of each quarter.** ***SQL:*** *SELECT prod_name product, country_name country, channel_id channel, SUBSTR(calendar_quarter_desc, 6,2) quarter, SUM(amount_sold) amount_sold, SUM(quantity_sold) quantity_sold FROM sales, times, customers, countries, products WHERE sales.time_id = times.time_id AND sales.prod_id = products.prod_id AND sales.cust_id = customers.cust_id AND customers.country_id = countries.country_id GROUP BY prod_name, country_name, channel_id, SUBSTR(calendar_quarter_desc, 6, 2);*

Table C.2 List of the queries used for dynamic selection of views in experiment 02

Query No.	Query Statements
Q_1	***Query:*** **Find the total number of customers in country, state wise for each account manager.** ***SQL:*** *SELECT c.account_mgr_id acct_mgr, cr.region_id region, c.cust_address.country_id country, c.cust_address.state_province province, COUNT (*) num_customers FROM customers c, countries cr WHERE c.cust_address.country_id = cr.country_id GROUP BY ROLLUP (c.account_mgr_id, cr.region_id, c.cust_address.country_id, c.cust_address.state_province);*
Q_2	***Query:*** **Find the product quantity in store.** ***SQL:*** *SELECT p.product_id, p.product_name, i.quantity_on_hand FROM inventories i, warehouses w, products p WHERE p.product_id = i.product_id AND i.warehouse_id = w.warehouse_id;*

Q_3	***Query:*** **Find the all customer information.** ***SQL:*** *SELECT c.customer_id, c.cust_first_name, c.cust_last_name, c.cust_address.street_address street_address, c.cust_address. postal_code postal_code, c.cust_address.city city, c.cust_address. state_province state_province, co.country_id, co.country_name, co.region_id, c.nls_language, c.nls_territory, c.credit_limit, c.cust_email, SUBSTR (get_phone_number_f (1, phone_numbers), 1, 25) primary_phone_number, SUBSTR (get_phone_number_f (2, phone_numbers), 1, 25) phone_number_2, SUBSTR (get_phone_number_f (3, phone_numbers), 1, 25) phone_number_3, SUBSTR (get_phone_number_f (4, phone_numbers), 1, 25) phone_number_4, SUBSTR (get_phone_number_f (5, phone_numbers), 1, 25) phone_number_5, c.account_mgr_id, c.cust_geo_location. sdo_gtype location_gtype, c.cust_geo_location.sdo_srid location_srid, c.cust_geo_location.sdo_point.x location_x, c.cust_geo_location. sdo_point.y location_y, c.cust_geo_location.sdo_point.z location_z FROM countries co, customers c WHERE c.cust_address.country_id = co.country_id(+);*
Q_4	***Query:*** **Find all the customer information with their orders.** ***SQL:*** *SELECT c.customer_id, c.cust_first_name, c.cust_last_name, c.cust_address, c.phone_numbers, c.nls_language, c.nls_territory, c.credit_limit, c.cust_email, CAST(MULTISET(SELECT o.order_id, o.order_mode, make_ref (oc_customers, o.customer_id), o.order_status, o.order_total, o.sales_rep_id, CAST(MULTISET(SELECT l.order_id, l.line_item_id, l.unit_price, l.quantity, make_ref (oc_product_information, l.product_id) FROM order_items l WHERE o.order_id = l.order_id) AS order_item_list_typ) FROM orders o WHERE c.customer_id = o.customer_id) AS order_list_typ) order_type, c.account_mgr_id FROM customers c;*
Q_5	***Query:*** **Find the product quantity in store for each warehouse.** ***SQL:*** *SELECT i.product_id, warehouse_typ (w.warehouse_id, w.warehouse_name, w.location_id) ware_typ, i.quantity_on_hand FROM inventories i, warehouses w WHERE i.warehouse_id = w.warehouse_id;*

Q_6	***Query:*** **Find the orders with the customer reference type.** ***SQL:*** *SELECT o.order_id, o.order_mode, make_ref (oc_customers, o.customer_id) cust_ref, o.order_status, o.order_total, o.sales_rep_id, CAST (MULTISET (SELECT l.order_id, l.line_item_id, l.unit_price, l.quantity, make_ref (oc_product_information, l.product_id) FROM order_items l WHERE o.order_id = l.order_id) AS order_item_list_typ) order_type FROM orders o;*
Q_7	***Query:*** **Find all the product information.** ***SQL:*** *SELECT p.product_id, p.product_name, p.product_description, p.category_id, p.weight_class, p.warranty_period, p.supplier_id, p.product_status, p.list_price, p.min_price, p.catalog_url, CAST (MULTISET (SELECT i.product_id, i.warehouse, i.quantity_on_hand FROM oc_inventories i WHERE p.product_id = i.product_id) AS inventory_list_typ) inv_typ FROM product_information p;*
Q_8	***Query:*** **Find the product information for each user logged separately.** ***SQL:*** *SELECT i.product_id, d.language_id, CASE WHEN d.language_id IS NOT NULL THEN d.translated_name ELSE TRANSLATE (i.product_name USING NCHAR_CS) END AS product_name, i.category_id, CASE WHEN d.language_id IS NOT NULL THEN d.translated_description ELSE TRANSLATE (i.product_description USING NCHAR_CS) END AS product_description, i.weight_class, i.warranty_period, i.supplier_id, i.product_status, i.list_price, i.min_price, i.catalog_url FROM product_information i, product_descriptions d WHERE d.product_id(+) = i.product_id AND d.language_id(+) = SYS_CONTEXT ('USERENV', 'LANG');*
Q_9	***Query:*** **Find the total order amount and quantity of each customer.** ***SQL:*** *select c.customer_id, sum(a.order_total), count(quantity) from(SELECT o.order_id, o.order_mode, o.order_status, o.order_total, o.sales_rep_id, l.order_id, l.line_item_id, l.unit_price, l.quantity, o.customer_id from order_items l, orders o WHERE o.order_id = l.order_id) a, customers c where a.customer_id=c.customer_id group by c.customer_id;*
Q_{10}	***Query:*** **Find the total order total amount of each customer.** ***SQL:*** *select c.customer_id, sum(a.order_total) from(SELECT o.order_id, o.order_mode, o.order_status, o.order_total, o.sales_rep_id, l.order_id, l.line_item_id, l.unit_price, l.quantity, o.customer_id from order_items l, orders o WHERE o.order_id = l.order_id) a, customers c where a.customer_id=c.customer_id group by c.customer_id;*