Portland State University

PDXScholar

Dissertations and Theses

Dissertations and Theses

10-21-1997

Materialized View Algorithms

Yubo Fan

Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the [Theory and Algorithms Commons](#)

Let us know how access to this document benefits you.

Recommended Citation

Fan, Yubo, "Materialized View Algorithms" (1997). *Dissertations and Theses*. Paper 5405.

<https://doi.org/10.15760/etd.7278>

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

THESIS APPROVAL

The abstract and thesis of Yubo Fan for the Master of Science in Computer Science were presented on October 21, 1997, and accepted by the thesis committee and the department.

COMMITTEE APPROVALS:

Leonard Shapiro, Chair

Joseph Albert

Marylou Daily
Representative of the Office of
Graduate Studies

DEPARTMENT APPROVAL:

Richard Hamlet, Chair
Department of Computer Science

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by

on

Dec. 23, 1997

ABSTRACT

An abstract of the thesis of Yubo Fan for the Master of Science in Computer Science presented October 21, 1997.

Title: Materialized View Algorithms.

A *data warehouse* is a stand-alone repository of integrated information available for decision support OLAP querying and analysis. Aggregate views can be materialized (stored in disk) to improve query performance in a data warehouse.

Several static and dynamic algorithms for selecting materialized aggregate views (MAV) in a data warehouse are proposed in this thesis. The algorithms are then compared by running a simulation system, which can be configured to compare several algorithms on different type of data warehouses. Simulation results for static algorithms are presented to show that several proposed algorithms perform close to an existing good algorithm (HRU Greedy) and run much faster. Simulation results also show that dynamic algorithms depend on locality and need improvement.

MATERIALIZED VIEW ALGORITHMS

by

YUBO FAN

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

Portland State University

1997

TABLE OF CONTENTS

1	INTRODUCTION.....	1
1.1	TERMS	6
1.2	OVERVIEW OF THE ALGORITHMS.....	7
1.3	OVERVIEW OF THE SIMULATION.....	7
1.4	THESIS OVERVIEW	8
2	RELATED WORK	9
2.1	HRU GREEDY AND ITS PERFORMANCE GUARANTEE	10
2.2	MVPP – MULTIPLE VIEW PROCESSING PLAN	12
2.3	WATCHMAN – AN INTELLIGENT CACHE MANAGER.....	12
3	ALGORITHMS.....	13
3.1	COST MODEL	14
3.2	ASSUMPTION.....	15
3.3	STATIC ALGORITHMS.....	16
3.3.1	SHARED SKELETON FOR STATIC ALGORITHMS	16
3.3.2	VARIOUS BENEFIT DEFINITIONS.....	17
3.3.3	ALGORITHM ANALYSIS	18
3.3.3.1	FS.....	18
3.3.3.2	IFS (Improved FS).....	18
3.3.3.3	FSG (FS Greedy)	18
3.4	DYNAMIC ALGORITHMS.....	19
3.4.1	ADMISSION ALGORITHMS	19
3.4.2	REPLACEMENT ALGORITHMS	21
4	SIMULATION SYSTEM.....	22
4.1	DATA WAREHOUSE BUILDER.....	23
4.1.1	DATA WAREHOUSE DESCRIPTION.....	23
4.1.1.1	Dimension Element.....	23
4.1.1.2	Dimension	24
4.1.2	VIEW CONSTRUCTION	26

4.2	QUERY GENERATOR.....	27
4.3	ALGORITHM EVALUATOR	28
4.4	MAV CONTROLLER.....	29
4.4.1	STATIC MATERIALIZATION.....	29
4.4.2	DYNAMIC MATERIALIZATION	29
5	SIMULATION RESULTS AND ANALYSIS	31
5.1	STATIC ALGORITHMS	31
5.1.1	SYNTHETIC WAREHOUSE	31
5.1.2	APPLICATION WAREHOUSE	34
5.2	DYNAMIC ALGORITHMS.....	38
6	CONCLUSION AND FUTURE WORK	44
	APPENDIX A CLASS DIAGRAMS OF SIMULATION SYSTEM	45
	APPENDIX B OUTPUT SAMPLE	47
B.1	TRACE OUTPUT.....	48
B.2	REPORT OUTPUT.....	49
B.3	EXCEL IMPORTABLE OUTPUT.....	50

LIST OF TABLES

<i>Number</i>	<i>Page</i>
Table 1	Fact table of a data warehouse of a retailer store chain 3
Table 2	Product dimension table..... 3
Table 3	Location dimension table 3
Table 4	Comparison and contrast between related work and MVA..... 9
Table 5	Definition of benefit for static algorithms 17
Table 6	Dynamic Algorithm Experiments Configuration..... 38

LIST OF FIGURES

<i>Number</i>	<i>Page</i>
Figure 1 Data warehouse vs. traditional databases.....	1
Figure 2 HRU Greedy performance guarantee.....	11
Figure 3 Shared skeleton of static algorithms.....	16
Figure 4 Shared skeleton of dynamic admission algorithms	20
Figure 5 Simulation system components and data flow.....	23
Figure 6 Simulation Specification File	25
Figure 7 Vertical dimension vs. horizontal dimension.....	26
Figure 8 Experiment 1, performance	35
Figure 9 Experiment 1, running time.....	35
Figure 10 Experiment 2, performance	36
Figure 11 Experiment 2, running time.....	36
Figure 12 Experiment 3, performance	32
Figure 13 Experiment 3, running time.....	33
Figure 14 Experiment 4, performance	33
Figure 15 Experiment 4, running time.....	34
Figure 16 Experiment 5, performance	39
Figure 17 Experiment 6, performance	40
Figure 18 Experiment 7, performance	40
Figure 19 Experiment 8, performance	41
Figure 20 Trace output sample.....	49
Figure 21 Normal report output sample.....	50
Figure 22 Excel importable output sample.....	51

ACKNOWLEDGMENTS

Many thanks to my advisor, Professor Leonard Shapiro. I owe him a great deal for his help and guidance from the beginning of the research to the end of the thesis.

Thanks to Mary Meredith for suggesting dynamic materialization algorithm that was part of the initiation of this project. Thanks to Dave Maier for helping with the advising of this thesis.

Last but not least, thanks to my fiancé, Shiow-Chen Lin, for her long-lasting understanding and support during the entire project.

Chapter 1

Introduction

A *data warehouse* is a stand-alone repository of integrated information available for decision support OLAP querying and analysis [IK93, Wid95]. Figure 1 shows the relationship between a data warehouse and a normal database system. In the figure, the group of databases at the top are *operational databases* used by clerks to perform data entries, such as a sale of a bottle of wine. The data warehouse at the bottom is the managerial database used by managers for decision support queries. The operational data bases feed data into the data warehouse on a regular basis. Because operational

Figure 1 Data warehouse vs. traditional databases

databases serve as the original data sources of a data warehouse, we use

operational database and original data source interchangeably in this thesis.

Decision support queries usually involve aggregate results. For example, a regional manager would like to know the monthly revenue for every city in her region. To answer this query, we could go through all the operational databases that contain the data requested. However, the operational databases are not ideal for processing decision support queries for two reasons. First of all, operational databases might be located at multiple sites as a distributed database system. Decision support queries are normally based on a large set of data sources. Therefore, it is expensive to retrieve data from the original sources to answer queries. Secondly, operational databases must accommodate the needs of real-time OLTP queries. Decision support querying on the data sources can compromise the performance of OLTP queries. A data warehouse can solve both problems because it contains integrated information and is separated from the original sources. The disadvantages of data warehouses are substantial redundancy and expensive updates. However, data warehouses are becoming more and more popular since the classic solution (OLTP and Decision Support Queries on one system) is not workable.

Aggregate queries tend to involve *facts* and *dimensions*. Facts are things like revenue, net income, and cost. Typical dimensions are time, location, and product. For example, a query might ask for the total revenue for each store in every month. The fact involved in the query is “total revenue.” The dimensions are “month”(time) and “store”(location). Because facts and dimensions are important elements in a data warehouse, most implementations of data warehouses use a similar scheme in that information is stored in a *fact table* and several *dimension tables*. Table 1 shows an example of a fact table. Table 2 and Table 3 are examples of the corresponding dimension tables. A fact table describes the facts for each record, which also contains the foreign

keys from dimension tables. Dimension tables describe the dimension hierarchies where aggregation can happen. For example, in the time dimension we can have year, quarter, month, week, day, etc.

Daily Sales	Product Key	Location Key	Time Key
\$10000	P001	L101	T010
\$40000	P110	L102	T010
...	...	...	...

Table 1 Fact table of a data warehouse of a retailer store chain

Product ID (Primary Key)	Brand	Color	Size
P001	Sony	White	5'
...	...	...	...

Table 2 Product dimension table

Store ID (Primary Key)	City	State	Region
L101	Portland	OR	Northwest
...	...	...	...

Table 3 Location dimension table

As stated earlier, a data warehouse contains the data from multiple databases. Moreover, the data often covers historic records because decision support queries often involve historic comparison or aggregation. Therefore the fact table of a data warehouse is usually huge. Performance can be very poor when queries are answered by directly accessing the fact table. Because decision support queries are mostly based on aggregates, a common technique

to improve performance is to *materialize* aggregate views. Here, “materialize” means to save the views in a secondary storage. For example, when the regional manager asks for the monthly sales by city, the performance can be greatly improved if we have already saved the results of monthly revenue by store or daily revenue by city. In this thesis, such materialized results are called materialized aggregate views (MAV). Using MAV is a huge double-edged sword in the data warehouse environment [Kim95]. It is very affective at improving performance, but it can take significant space to materialize pre-computed views. Moreover, when the fact table is updated based on the original operational databases, all the MAV’s need to be updated accordingly, which could be very expensive. The storage space for MAV’s is usually limited. Therefore it is impossible to materialize all the aggregate views in a data warehouse. Each data warehouse has limited space assigned for MAV’s. We call that storage space the *MAV buffer*.

Researchers have developed algorithms to select MAV’s and manage MAV buffers at run-time. These algorithms can be divided into two categories, static and dynamic. Static algorithms are used to select MAV’s to fill the MAV buffer before queries are processed. Then the MAV buffer does not change when queries are processed. On the contrary, dynamic algorithms change the contents of the MAV buffer when queries are processed. The MAV buffer is managed at run-time to materialize new views and replace the ones that are already in the buffer. For example, when processing the query of “total revenue for each store in every quarter,” a dynamic algorithm could suggest to materialize the corresponding view or an *ancestor* view, such as “total revenue for each store in every month.” If there is not enough space to materialize the selected view, a replacement algorithm is applied to page out some existing MAV’s in the buffer.

In this thesis we compare several static and dynamic algorithms by using a

data warehouse simulation system. The simulation varies data warehouse and buffer sizes in order to show the effectiveness and speed of algorithms. We also analyze the results to point out the advantages and disadvantages of each algorithm. Therefore our work is called Materialized View Algorithms (MVA).

In the area of MAV strategy, the authors of [ZHKF95] discussed the taxonomy of solution space for data integration, which are fully materialized, hybrid, and fully virtual. All the MAV selection algorithms apply to the hybrid solution (partially materialized) because the other two approaches are straight forward.

In the area of static algorithms, a series of articles from the Stanford database group studied several static algorithms and compared their performance to that of the optimal solution [HRU96, GHRU97, Gup97]. Index selection algorithms and view maintenance have been incorporated into the study. In [YKL96], Yang, Karlapalem, and Li also considered view maintenance when selecting MAV's. View maintenance has been thoroughly studied in [CW91] and [GM95]. In our simulation system, view maintenance is not incorporated. Our static algorithms are mainly based on the skeleton of the HRU Greedy algorithm, which is an application of the general Greedy algorithm [HRU96]. Our goal is to develop algorithms with similar performance to HRU Greedy but much less running time.

In the area of dynamic algorithms, one of the investigations was by Roussopoulos, Chen, and Kelley [RCK95], who proposed an adaptive buffer manager to determine "critical size" and "saturation size" of the buffer for ViewCache and Materialized View Fragment. In a later work, Scheuermann, Shim, and Vingralek proposed query caching [SSV96], which manages cache based on access rate of views by queries. Our dynamic algorithms are similar to the ones proposed in that work. The difference is that we use our data

warehouse simulation system to compare the algorithms by running on various data warehouses. Also, we are not restricted to saving views accessed by queries. We consider a broader range of views to materialize in order to achieve the best query performance.

The rest of the introduction is structured as follows: Section 1.1 contains definitions of the terms used in this thesis. Section 1.2 contains an overview of the algorithms studied in MVA. Section 1.3 presents an overview of the simulation. Section 1.4 is the overview of the thesis.

1.1 Terms

A *query* is a question asked by users based on the relations in the database. For example, retrieve the revenue of all products sold, grouped by year, by store, and by brand.

A *view* is the set of tuples that satisfies a query. For example, the view for the previous query is a set of revenues and their year, store, and brand.

A *materialized view* is a view stored on disks.

View A is called an *ancestor* of view B if view B can be generated from view A. For example, suppose view A is daily revenue, and view B is annual revenue. View B can be generated from view A, so view A is an ancestor of view B.

View A is called a *descendant* of view B if view B is an ancestor of View A.

View A is called a *parent* of view B if view A is an ancestor of view B, and

there does not exist view C such that view C is an ancestor of view B and view A is an ancestor of view C. In the example for ancestor, daily revenue is not a parent of annual revenue because there exists monthly revenue and quarterly revenue between them. Quarterly revenue is a parent of annual revenue because there does not exist a view that is an ancestor of annual revenue and also a descendant of quarterly revenue. The definition depends on the structure of dimensions. In this case, there is no dimension level between year and quarter, so the relationship matches the definition.

View A is called a *child* of view B if view B is a parent of view A.

1.2 Overview of the Algorithms

We study both static and dynamic algorithms in this research. For static ones, we adopt the skeleton of the HRU Greedy algorithm [HRU96] and use different benefit formulas in the skeleton. Our goal is to find an algorithm with performance close to that of HRU Greedy and more efficient. For dynamic algorithms, we have developed both admission algorithms and replacement algorithms to manage the MAV buffer. We mainly study whether or not the dynamic algorithms could improve performance by comparing their performance with that of static algorithms. Our dynamic algorithms contain the combination of several admission algorithms and one replacement algorithm, LRU (Least Recently Used).

1.3 Overview of the Simulation

In order to study algorithms, both simulation and benchmark techniques can be used. It takes a great deal of resources, both human and machine, to

implement and run a benchmark. We choose to use the simulation approach because it is easier to develop the simulation for a variety of algorithms and for different types of data warehouses. Also it is more proper to use simulation at the early stage of evaluating algorithms because it is cheaper to filter out the less affective algorithms and narrow down the research directions.

Our simulation system runs through three major steps. At first it builds a data warehouse based on an external description file, which is called the *simulation specification file*. Secondly it builds a query stream, which contains a combination of query patterns, based on the same specification file. Finally, it evaluates the performance and running time of various algorithms during the execution of the query stream.

1.4 Thesis Overview

The remainder of this thesis is organized as follows: Chapter 2 summarizes related work by other authors. Each work is compared and contrasted against MVA. Chapter 3 details the algorithms studied in this thesis. Definition and analysis of each algorithm are presented. Chapter 4 presents the data warehouse simulation system. We provide the object-oriented design model with class diagrams and sequence diagrams in Appendix A. Chapter 5 contains the results produced by the simulation system and our analysis. In Chapter 6 we present our conclusions and suggestions for future work.

Chapter 2

Related Work

In this chapter we present summaries of related work by other authors. After every summary we compare and contrast our work with theirs.

In order to better compare our ideas with that of other authors, we consider several important features of materialized view design. The features include static algorithms, dynamic algorithms, number of algorithms studied, and considerations of update cost and limited space. The following table shows the similarity and difference among all the work:

work	static	dynamic	limited space	update cost	number of algorithms
[HRU96]	Y	N	Y	Y	1
[Gup97]	Y	N	Y	Y	5
[YKL96]	Y	N	N	Y	1
[SSV96]	N	Y	Y	N	1
MVA	Y	Y	Y	N	15

Table 4 Comparison and contrast between related work and MVA

In Table 4, column “static” and “dynamic” indicate whether or not the work studies static or dynamic MAV algorithms. Column “limited space” indicates whether or not the work considers MAV buffer space as a constraint. Column “update cost” shows whether or not the research incorporates the update cost of MAV’s. The last column indicates the number of algorithms studied in the work.

2.1 HRU Greedy and its performance guarantee

The Stanford database group has contributed a great deal in the area of static materialized view design. Harinarayan et al in [HRU96] presented a *lattice* framework and an application of the HRU Greedy algorithm in the selection of MAV's. A lattice is a network of elements, in which for any two elements x and y there exist a least element including both x and y and a greatest element included by both x and y . In a data warehouse, all the views are considered elements in the lattice. For any two views there exist a least common ancestor and a greatest common descendant. The HRU Greedy algorithm is defined in [HRU96]. The authors proved that the HRU Greedy algorithm is guaranteed to perform at least 63% as well as the optimal solution in terms of benefit produced by MAV's.

In a later paper, Gupta incorporated index and view maintenance into the framework [Gup97]. By introducing AND-OR view graphs, the author applied the general Greedy algorithm and its variations on some important cases of the general data warehouse scenario. The author also proved the performance guarantee of the algorithms is within a constant factor ratio of the optimal solution.

Our work is based on the Stanford group's lattice framework. We also use their simplified cost model, in which the cost of a query execution is the number of tuples that have to be scanned during the execution. We agree that HRU Greedy is an effective algorithm that guarantees performance, but we also notice its high complexity. The complexity of Greedy algorithm is $O(kn^2)$, where k is the number of views selected to materialize, and n is the total number of nodes to choose from. We propose several algorithms that take less time than HRU Greedy. The complexities of those algorithms are $O(n)$ or $O(kn)$. We compare their performance with that of HRU Greedy

using out simulation system on a variety of types of data warehouses.

Thanks to Dave Maier [Mai96], we have also noticed that the performance guarantee presented for HRU Greedy is based on performance benefit instead of performance itself. Figure 2 illustrates the difference between the two. In the figure, the benefit for HRU Greedy is $10000 - 1000 = 9000$, while the benefit for optimal is $10000 - 10 = 9990$. The benefit ratio between HRU Greedy and optimal is 91%, which seems to be very promising. However, the actual cost of HRU Greedy is 100 times greater than that of the optimal solution. Thus we can see that the relative benefit can be arbitrarily high while the performance, relative to optimal, is arbitrarily poor. Our simulation results use cost (as measured by response time) as the quantity to minimize in order to avoid the misleading evaluation given by the benefit.

Figure 2 HRU Greedy performance guarantee

Dynamic selection of MAV's is another area that is not studied in [HRU96] and [Gup97]. By using our simulation system, we are able to

compare our dynamic algorithms with static algorithms.

2.2 MVPP -- multiple view processing plan

From another approach, the authors of [YKL96] provided a detailed algorithm that selects materialized views based on query execution plans. The framework presented in the paper is based on the specification of Multiple View Processing Plan (MVPP) in a distributed data warehouse environment. The cost model used in the paper is abstract although it includes the query performance as well as the view maintenance. In order to accomplish the algorithm in [YKL96], the database system must have substantial knowledge of the queries.

The paper demonstrated its algorithm by using a sample data warehouse without comparing with other algorithms. Our work does compare several algorithms that could be used to select materialized views. The algorithm proposed in [YKL96] does not consider space as a constraint. Instead, it compares cost with benefit to justify whether or not a view should be materialized. Our MVA uses space as both constraint and variable to investigate its effect on the performance of several algorithms.

Another major difference between MVPP and MVA is that we also consider dynamic algorithms for the selection of materialized views.

2.3 WATCHMAN – An intelligent cache manager

In the area of dynamic selection of materialized views, the authors of [SSV96] presented an intelligent cache manager for view sets retrieved by queries. Two novel and complimentary algorithms are used by the manager to address cache replacement and cache admission. The two algorithms, LNC-A (Least

Normalized Cost Admission) and LNC-R (Least Normalized Cost Replacement), were designed to minimize the execution time of the queries that miss the cache. For a retrieved set, the algorithms are based on the average rate of reference by a query, the size of the set and the cost of execution of the query. The emphasis of [SSV96] is on the difference between a data warehouse cache manager and a normal database buffer manager.

In our dynamic algorithms, we use reference time, view size, and cost of the query as basis for cache replacement and admission. Our work differs from WATCHMAN in the following ways:

- When considering cache admission, WATCHMAN only looks at the set retrieved by a query while MVA considers other sets.
- Our benefit formula for a candidate cache also considers the reference to its descendant views. Thus the goal of minimizing the overall execution time is better achieved.

Chapter 3

Algorithms

In this chapter, we present the algorithms implemented in our simulation system. First of all, we introduce the cost model. Secondly we list all the static algorithms. Finally we present the dynamic view management algorithms. The results of using our simulation system to study these algorithms will be compared and analyzed in Chapter 5.

3.1 Cost Model

The cost model that we used to define the algorithms is similar to that of [HRU96]. The cost of a query is the number of tuples that have to be scanned in order to answer it. This model assumes every tuple in the fact table and every view has the same length.

As defined in the introduction, each query corresponds to a view. Let V_i be the corresponding view of query Q_i . Q_i 's cost of execution (C_i) will be minimum if V_i is materialized, i.e. $C_i = \text{Card}(V_i)$. On the other hand, when nothing except the fact table is materialized, every query has to be answered by scanning the fact table, therefore $C_i = \text{Card}(V_{\text{fact}})$. When some of the views are materialized, the cost of Q_i is the cardinality of the smallest materialized ancestor view of V_i . In particular, $C_i = \text{Card}(V_{\text{smallest}})$, where V_{smallest} is the smallest materialized ancestor of V_i .

A better cost model can be defined based on the work in [SDFR96]. Shukla et al have studied storage estimation for multidimensional aggregates.

Although their work is not restricted to the data warehouse environment, the authors used techniques such as sampling, mathematical approximation, and probabilistic counting, which are still applicable to our research. The authors used uniformity as an assumption to estimate sparsity on the basis of sampling, which is very useful in a data warehouse system.

We chose the simple [HRU96] cost model for these reasons

- It is easy to use.
- We wanted to spend our limited time focusing on the algorithms for view admission and replacement, where we feel we have something new to contribute
- Many researchers have studied cost models for database systems
- Our algorithms can be extended easily to use new cost models.

3.2 Assumption

All our algorithms assume that there is a space constraint for the materialized view storage. The available space is not enough to materialize all possible views. Static algorithms finish the selection of MAV's when there is no space left for materialization. Dynamic algorithms manage the limited space by applying an admission algorithm and a replacement algorithm.

Our algorithms also have knowledge of all statistics related to all the views, such as cardinality and dimension levels. In real database systems, this kind of knowledge could be obtained by going through the catalog or sampling [SDNR96]. In our simulation system, however, the knowledge is directly obtained from the view objects (Chapter 4).

3.3 Static algorithms

In this section, we present the static algorithms studied in our research. Because all these static algorithms share the same skeleton, we first introduce the shared structure, then cover the differences among the algorithms.

3.3.1 Shared skeleton for static algorithms

All the static algorithms have the same skeleton as HRU Greedy [HRU96]. This skeleton algorithm is illustrated in Figure 3. In the figure, given a set of all possible views W , a set of materialized views MV , and the space constraint for MAVs S ,

$V_{\text{candidate}}$ denotes the views which can be added to the MAV buffer this time, namely, the set of all views V_i satisfying

$$V_i \in W, V_i \notin MV, \text{Card}(V_i) \leq S - \text{Space}(MV), \text{ and } \text{Benefit}(V_i) > 0;$$

V_{best} denotes the view in $V_{\text{candidate}}$ with the biggest benefit.

The definition of “benefit” is given in the next section.

<p>Given: W, the set of all possible views S, the space constraint</p> <p>Output: MV, a set of MAVs.</p> <p>BEGIN</p> <p>$MV = \emptyset;$ <i>// Initially no views are materialized</i></p> <p>while ($\text{Space}(MV) < S$) <i>// While there is space left for MAVs</i></p> <p> if $V_{\text{candidate}} = \emptyset$, end; <i>// No more views can be materialized</i></p> <p> find V_{best} with the maximum benefit in $V_{\text{candidate}};$</p> <p> $MV = MV \cup \{V_{\text{best}}\};$</p> <p> end while;</p> <p>END.</p>
--

Figure 3 Shared skeleton of static algorithms

3.3.2 Various benefit definitions

The static algorithms we discuss share the skeleton code described in Figure 3. They differ in how they define benefit. Table 5 shows the definition of benefit for each algorithm.

Algorithm	Benefit
HRU Greedy	$\sum (C_i - C_{i-after})$
FS	$Card(V_{smallest_parent}) - Card(V_{can})$
IFS	$(Card(V_{smallest_parent}) - Card(V_{can})) * (1 + numDescendants(V_{can}))$
FSG	$(C_{can} - Card(V_{can})) * (1 + numDescendants(V_{can}))$
HRU Greedy Space	$\frac{Benefit_{Greedy}(V_{can})}{Card(V_{can})}$
FS Space	$\frac{Benefit_{FS}(V_{can})}{Card(V_{can})}$
IFS Space	$\frac{Benefit_{IFS}(V_{can})}{Card(V_{can})}$
FSG Space	$\frac{Benefit_{FSG}(V_{can})}{Card(V_{can})}$

Table 5 Definition of benefit for static algorithms

In Table 5, we use the following terms:

V_{can} candidate view to be materialized

$V_{smallest_parent}$ the smallest parent of V_{can}

The following three costs assume that the views in MV have been

materialized.

C_{can}	cost of V_{can}
C_i	current cost of Q_i
$C_{i-after}$	cost of Q_i after the materialization of the V_{can}

3.3.3 Algorithm Analysis

3.3.3.1 FS

The FS algorithm measures a candidate view's benefit by the cardinality gap between its smallest parent and itself. The rationale behind the algorithm is that a view is more valuable when there is a big gap between its ancestors and itself. The materialization of the view can eliminate this gap thus benefiting the view and its descendants.

The disadvantage of the FS algorithm is that it does not consider the number of descendants that can benefit from its materialization. Neither does it consider the current materialization status of other views.

The complexity of FS is $O(n)$, where n is the total number of views.

3.3.3.2 IFS (*Improved FS*)

The IFS algorithm is the same as FS except that the benefit formula used by IFS take into consideration the benefits for the descendants. Because of this, we expect IFS to perform better than FS.

The disadvantage of the IFS algorithm, as in FS, is that the benefit does not change based on the materialization of other views.

The complexity of IFS is also $O(n)$.

3.3.3.3 FSG (*FS Greedy*)

The FSG algorithm calculates the benefit by multiplying the difference of its current cost and cardinality with the number of possible beneficiaries.

This algorithm is much closer to the HRU Greedy algorithm than FS and IFS because it considers the change in the current cost of a view. However, it

does not consider the materialization status of a view's descendants, which results in a benefit bigger than reality.

The complexity of FSG is $O(kn)$, where k is the number of views selected to be materialized.

3.4 Dynamic Algorithms

Dynamic algorithms differ from static algorithms in that the contents of the MAV buffer can change dynamically as queries are issued to the database. The goal of a dynamic algorithm is to adjust the MAV buffer continually, in response to query activity, so that performance is maximized. In this way it is similar to virtual memory. Virtual memory has only a replacement policy, as do MAV dynamic algorithms. Virtual memory has no choice of what to materialize – it must materialize the latest page fault. However, MAV dynamic algorithms can materialize anything. Their goal is to choose views to materialize in order to maximize performance. Thus MAV dynamic algorithms consist of two parts, admission and replacement algorithms. An admission algorithm is used to select one or a set of candidate views to materialize. A replacement algorithm helps the system decide which materialized views to remove when there is not enough storage space for new materialized views. We present them separately in the following sections.

3.4.1 Admission algorithms

Admission algorithms are similar to static algorithms because they also share the same code skeleton, which is described in Figure 4. The term T in Figure 4 refers to a threshold, which defines the customer requirement for the response time. Given a query Q_i ,

V_i denotes its corresponding view,

$V_{\text{smallest_mv}}$ denotes the smallest materialized ancestor of V_i .

$V_{\text{candidate}}$ denotes the set of views V_j satisfying

$V_j = V_i$, or V_j is an ancestor of V_i and a descendant of $V_{\text{smallest_mv}}$

Given: Q , an ordered set of queries; T , threshold

BEGIN

for each query Q_i in Q

if $\text{Cost}(Q_i) > T$ **then** // *Is query execution too expensive*

 find the best view (V_{best}) with the biggest benefit in $V_{\text{candidate}}$

 materialize V_{best} *

end if

end for

END.

* If there is not enough space to materialize V_{best} , a replacement algorithm is used to allocate necessary space.

Figure 4 Shared skeleton of dynamic admission algorithms

While determining V_{best} , the admission algorithms use the same benefit formulas as the static algorithms, which are described in Table 5.

Groups of queries usually come in patterns because users ask for follow-up queries after seeing the results of previous ones. Two common query patterns are *drill-up* and *drill-down*. For example, after a user gets the results of yearly revenue by region, she might look into monthly revenue by region, then monthly revenue by store. This is called a drill-down pattern, which can involve several queries in similar sequence. Drill-up patterns are the opposite, and are less common than the drill-down ones. Note that drill-up queries move down in the lattice.

In Figure 4, the threshold is used to identify an expensive execution of a

query determined by customer requirement of response time, such as five minutes. Normally an expensive execution is caused by a big gap between the query's corresponding view and its materialized ancestor views. The purpose of looking for the best view along the paths is to reduce the gap effectively for the queries asked in the future. We consider the ones along the path and hope the queries asked in a pattern can benefit from the recent materialization.

3.4.2 Replacement algorithms

In this research, we only study one replace algorithm, LRU. Every MAV in the buffer is stamped by the last reference time. When there is not enough space to materialize new views, the buffer manager clears the views that have not been referenced after a certain time point. It keeps moving the time point closer to the current time to throw out more materialized views until enough space is made.

Chapter 4

Simulation System

The simulation system is designed with the Unified Modeling Language (UML) and implemented in C++ in the Borland C++ 4.52 environment.

The components of the system are a data warehouse builder, a query generator, an algorithm evaluator, and a MAV controller. As shown in Figure 5, the data warehouse builder builds a data warehouse. Then the query generator constructs a query stream based on the data warehouse. The algorithm evaluator simulates the algorithms on the data warehouse and the query stream. During the simulation, the MAV controller performs static materialization and dynamic materialization based on the algorithm combinations. After all the algorithm combinations have been simulated, the algorithm evaluator generates the result of the simulation.

The rest of this chapter details the design of the Simulation system at the component level. For design details at the class level, see class diagrams, sequence diagrams, and class documentation in Appendix A.

Figure 5 Simulation system components and data flow

4.1 Data Warehouse Builder

The data warehouse builder component of the Simulation system builds the data warehouse based on the simulation specification file (Figure 6). It first builds the dimension hierarchy, then constructs all the possible views based on the dimensions.

4.1.1 Data Warehouse Description

4.1.1.1 Dimension Element

In each dimension, a dimension element represents one dimension level. For example, *year* is a dimension element in the *time* dimension (Figure 6, line 18). In the specification file, we can define the cardinality of each dimension

element for a particular data warehouse. For example, by defining the cardinality of year to be 8, we can produce a data warehouse that contains eight years of data. For simplicity, the system assumes no sparsity. Therefore, the system is able to determine the cardinality and hierarchy relationship of the fact table and all the views once the dimension elements are specified in every dimension.

4.1.1.2 Dimension

In our simulation system, we define two types of dimensions, *vertical dimensions* and *horizontal dimensions*.

A typical example of a *vertical dimension* is “location.” As shown in Figure 7, from the top to the bottom, every level of the dimension contains only one element. Each element is one level higher than the next one, which means the aggregate results of the lower level can be generated if the results of the higher level are already known. In Figure 7, for example, when the aggregate results of all the cities are known, we can produce the aggregate results of the states and the countries.

A *horizontal dimension* is like a flat diamond as shown in Figure 7. A horizontal dimension has three levels. The top level is the finest grain level, such as product id. The middle level contains several elements, such as brand, color, and size. The bottom level is the sum over all dimension elements, i.e. sum of sales of all products.

Although there are other dimensions, such as a combination of vertical and horizontal dimensions, we consider the two types of dimensions typical and representative.

```

1  # Application Warehouse 001
2  #
3  # must use detail builder
4  #
5  # grammar:
6  # '#' as comment
7  # [...] as segment title
8  # key = value as segment content
9
10 [dimension]
11 name = time           # time dimension
12 type = vertical       # is vertical
13 elem = 2880, day      # element and its cardinality
14 elem = 384, week      #
15 elem = 96, month      #
16 elem = 32, quarter    #
17 elem = 8, year        #
18 elem = 1, all         #
19
20
21 [dimension]
22 name = product        # product dimension
23 type = horizontal     # is horizontal
24 elem = 200, prod_id
25 elem = 40, size
26 elem = 25, brand
27 elem = 10, color
28 elem = 1, all
29
30 [dimension]
31 name = location       # location dimension
32 type = vertical       # is vertical
33 elem = 600, store
34 elem = 200, city
35 elem = 50, state
36 elem = 6, region
37 elem = 1, all
38
39 [mav]
40 admissionThreshold = 5
41
42 [queryStream]
43 seed = 100            # random number generator seed
44
45 uniform = false       # does query stream contain all the queries ?
46 numQueryPattern = 100 # number of query patterns
47
48 drillUp = 90          # percentage of drill up pattern
49 drillDown = 10        # percentage of drill down pattern
50
51 [algorithm]
52 NumAlgor = 2          # number of algorithms
53
54 static = GREEDY_SPACE
55 admission = NONE      # no dynamic materialization
56 replacement = LRU
57
58 static = FS_SPACE
59 admission = IFS
60 replacement = LRU

```

Figure 6 Simulation Specification File

Figure 7 Vertical dimension vs. horizontal dimension

4.1.2 View Construction

After building all the dimensions with their elements, the data warehouse builder then continues to construct the cardinality of the views and the relationships among them. Because we do not consider selection in our queries, every view can be defined by the element level of each dimension. For example, suppose a warehouse has three dimensions, time, location, and product. A view can be defined by a set of dimension elements, such as month, state, and brand. The definition of the view determines its hierarchy position in the data warehouse and its cardinality. Following this convention, the data warehouse builder is able to construct the properties of all aggregate views and mark them as not materialized except the top one, the fact table.

A tree-type data structure is designed for the aggregate views in the data

warehouse so that every view is able to traverse through its parents, children, ancestors, and descendants. Later when some of the views are materialized, each view also knows which ancestor is the smallest materialized one so that it is easy to calculate the cost of a certain query.

4.2 Query Generator

The query generator constructs the query stream to be tested on selected algorithms. A *query stream* is an ordered sequence of queries issued to the database. In our system, a query stream consists of a series of query patterns. A *query pattern* consists of a group of queries that are issued sequentially to follow up previous queries. Our simulation system is able to generate three patterns, drill-up, drill-down, and jump. A drill-down pattern moves up in the lattice model, which looks for more and more detailed results. For example, a user might at first ask for total revenue by state, secondly by city, and thirdly by store. In this way, the user can finally locate her interested information by querying at finer grain. By using the previous pattern, a regional manager is able to find out stores with bad performance that has caused a low total revenue in a particular state. A drill-up query pattern goes the opposite direction of a drill-down pattern. It moves down in the lattice model, which contains a series of queries that ask for higher level of aggregation. A jump query pattern randomly jumps to a different area in the lattice. In reality, it could be caused by another user or the same user switching to another problem.

Queries are generated randomly to build all the query patterns. By defining the probability of each pattern type in the stream in the simulation specification file (Figure 6), we are able to test how the distribution of the patterns could effect the performance of the algorithms.

4.3 Algorithm Evaluator

A single execution of the Simulation system tests selected algorithms for one query stream. The simulation specification file defines the algorithm combinations to be tested. Each algorithm combination contains a static algorithm, a dynamic admission algorithm, and a dynamic replacement algorithm. For example, line 52-60 in Figure 6 defines two combinations of algorithms. The first combination contains GREEDY_SPACE as the static algorithm and no dynamic algorithms. The second combination contains FS_SPACE as the static algorithm, IFS as the dynamic admission algorithm, and LRU as the dynamic replacement algorithm. After the data warehouse builder and the query generator pass the data warehouse and the query stream to the algorithm evaluator, the evaluator simulates the query execution with the algorithms to be tested. Before the evaluator simulates the execution, it lets a MAV controller perform a static materialization based on the given static algorithm, which could be defined as NONE, meaning no static materialization. If the static algorithm is not NONE, the algorithm is applied to select views until there is not enough space to materialize any more.

After simulating the static algorithm, the evaluator processes each query in the query stream. For every query, it asks a MAV controller for the current cost of the query and lets the controller decide if any dynamic materialization is necessary based on the particular query process. The evaluator keeps track of the cost of all the queries for each algorithm combination. After one combination is finished, the evaluator resets the MAV buffer and cost attribute for all the views. After these data structures are reset, simulation begins for the next algorithm combination.

The simulation is complete when all algorithm combinations have been simulated. Upon completion of the simulation, the evaluator generates three

types of reports (see examples in Appendix B). The trace report has the most detailed information, which records the steps of materialization and all the cost information. The normal report summarizes the results in small sections for all the tested algorithm combinations. The excel report generates a summary in a format that can be easily processed by Microsoft Excel so that various comparison graphs can be produced.

4.4 MAV Controller

The MAV controller is in charge of all the materialization. It can perform static materialization in order to fill the MAV buffer, or dynamically materialize aggregate views by applying both admission and replacement algorithms.

4.4.1 Static Materialization

As mentioned in the view construction section, each view is aware of its smallest materialized ancestor. When a view is materialized, the system may need to go find each of its descendants in order to update the view's statistics. Thus all the cost information is guaranteed to be accurate.

Static algorithms use different benefit formulas. Some do not require recalculation of the benefit during materialization. For example, the benefit for the FS algorithm stays the same no matter how many views are materialized. The system stores the benefit for each view. For algorithms like Greedy or FSG, the system also updates the benefit every time after a new view is materialized.

4.4.2 Dynamic Materialization

The MAV controller compares the cost of each query to a defined threshold. If the cost is greater than the threshold, the given admission algorithm is used

to find a view to materialize, which is similar to a single step in the static materialization. The difference is that the searching range is narrower for admission algorithm.

After a view is selected, if there is not enough space, the replacement algorithm is used to page out materialized views to make enough space. To simulate LRU, the system records the reference time for each query. When space is needed, the system selects the ones that are the least recently referenced and dematerializes them. The reverse operation of materialization also requires the system to update the information kept by each view about its smallest materialized ancestor.

Chapter 5

Simulation Results and Analysis

In this chapter, we present results for static algorithms and dynamic algorithms separately. We compare static algorithms in terms of average cost of query execution and algorithm running time. In section 5.2, on dynamic algorithms, we compare the performance of several dynamic algorithms with that of one static-only algorithm.

5.1 Static Algorithms

In this section, we compare four static algorithms, namely, HRU Greedy-space, FS-space, IFS-space, and FSG-space. We do not show the results for HRU Greedy, FS, IFS, and FSG because they perform relatively poorly compared with their “space” siblings. The algorithms are simulated on four data warehouses. The first two data warehouses are *synthetic warehouses*, while the other two are *application warehouses*.

5.1.1 Synthetic Warehouse

Synthetic Warehouses contain only vertical dimensions. Every vertical dimension has two levels. The cardinality ratio is 2 to 1 between the two dimension levels. We start with these simple structure because it’s easier to understand the structure of the views and the behavior of the algorithm. We can create different synthetic warehouses by varying the number of dimensions. Experiment 1 (Figure 8 and Figure 9) is running the four static algorithms on a synthetic warehouse with 7 dimensions. Experiment 2 (Figure

10 and Figure 11) is running on a synthetic warehouse with 10 dimensions.

All the graphs have the same axes in this chapter. The vertical axis is the ratio between the average cost of all the queries in the query stream and the minimum average cost. By minimum average cost we mean the average cost of the query execution when everything is materialized. The horizontal axis is the space in terms of the percentage of total cardinality of all the views. The total cardinality is the same as the space it takes to materialize everything.

In the results, we can see that although HRU Greedy-space is overall the best in performance, other algorithms are running close. Moreover, in terms of running time, other algorithms, especially FS-space and IFS-space, run much faster than HRU Greedy-space.

Figure 8 Experiment 1, performance

Figure 9 Experiment 1, running time

Figure 10 Experiment 2, performance

Figure 11 Experiment 2, running time

5.1.2 Application Warehouse

Application data warehouses are based on a retail company, which has similar properties to the one in the sample simulation specification file (Figure 6).

In Experiment 3 (Figure 12 and Figure 13), we compare four algorithms in an application warehouse with three dimensions, which are time, location, and product. In Experiment 4 (Figure 14 and Figure 15), the application warehouse has four dimensions. This warehouse has an additional dimension compared to the previous data warehouse.

Figure 12 Experiment 3, performance

Figure 13 Experiment 3, running time

Figure 14 Experiment 4, performance

Figure 15 Experiment 4, running time

The results are similar to the ones for synthetic warehouses. Alternate algorithms are performing close to HRU Greedy-space but running much faster.

We notice another interesting phenomenon that the running time for HRU Greedy-Space does not always increase when there is more space for the MAV buffer. For example, in Figure 15, it takes more time to fill a 20% buffer than to fill a 25% buffer. We have discussed earlier that the running time of Greedy/Greedy-space is $O(kn^2)$, where n is the number of all possible views, and k is the number of views selected to materialize. Normally k is bigger as the MAV buffer is larger. However, sometimes a larger buffer can contain fewer views than a smaller buffer. Therefore, it takes longer to fill the smaller buffer.

5.2 Dynamic Algorithms

In this section, we compare three dynamic algorithms with a static-only algorithm. The data warehouse used in the experiments is an application warehouse with three dimensions. Before executing the queries, all the MAV buffers are filled with MAVs by applying HRU Greedy-space. Among the dynamic algorithms used in the experiments, the replacement algorithms are LRU, the admission algorithms use the same skeleton with different benefit formulas (section 3.4.1). The benefit formulas we use in the experiments are HRU Greedy, FS, and IFS. In the graphs in this section, we use the name of the benefit formula to identify the algorithms. For the static-only algorithm, we use “NONE” as the legend.

While using the same data warehouse (an application warehouse with three dimensions), we vary the experiments by the query streams. One of the variables is query pattern distribution. The other one is query pattern *locality*. Locality is a property of a query stream that indicates how close the query patterns are. A query stream with locality means that series of query patterns might start with the last query of the previous pattern. The four experiments are set up with the properties shown in Table 6:

	Query pattern distribution	Query pattern locality
Experiment 5	10% drill down, 90% drill up	low
Experiment 6	50% drill down, 50% drill up	low
Experiment 7	10% drill down, 90% drill up	high
Experiment 8	50% drill down, 50% drill down	high

Table 6 Dynamic Algorithm Experiments Configuration

The first variable, query pattern distribution, does not seem to have an

obvious influence on the results except that the query streams are totally different.

The other variable, query pattern locality, does effect the comparison a great deal. In Figure 16 and Figure 17, the static-only algorithm performs almost as well as (sometimes even better than) the dynamic algorithms. However, in Figure 18 and Figure 19, most of the dynamic algorithms perform substantially better than the static-only algorithm. This demonstrates that dynamic algorithms are suitable for query streams with high locality.

While examining the trace file of the simulation, we find a weakness of the dynamic algorithms. The algorithms sometimes choose a very big view to materialize, which in turn requires it to throw away a lot of useful MAV's in order to make space.

Figure 16 Experiment 5, performance

Figure 17 Experiment 6, performance

Figure 18 Experiment 7, performance

Figure 19 Experiment 8, performance

Chapter 6

Conclusion and Future Work

In this thesis, we proposed several static and dynamic algorithms for MAV selection in a data warehouse.

We have simulated the proposed algorithms in several different data warehouses. As expected for static algorithms, we found that our proposed algorithms performed close to the HRU Greedy-space algorithm and ran much faster.

The dynamic algorithms did not perform as well as we expected. One reason is that locality plays a very important role in the performance issue. Without a substantial locality, dynamic algorithms can hardly perform any better than static-only algorithms. The second reason is that the proposed combination of the admission algorithm and the replacement algorithm might not be suitable for the overall performance.

Dynamic algorithms can be further studied in the future. It is possible to apply some heuristics in the admission algorithm and the replacement algorithm to avoid throwing out useful views. Similar ideas were presented in the Greedy Interchange algorithm of [Gup97] and the LNC-RA algorithm of [SSV96].

BIBLIOGRAPHY

- [CW91] S. Ceri, and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases*, Barcelona, Italy, September, 1991.
- [GM95] A. Gupta, and I. S. Mumick. Maintenance of materialized views: problems, techniques, and applications. *Data Engineering*, 18(2):3-18, June, 1995.
- [Gup97] H. Gupta. Selection of views to materialize in a data warehouse. In *Proceedings of the International Conference on Database Theory*, Delphi, Greece, Jan 1997
- [HGW95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The Stanford data warehousing project. *Data Engineering*, 18(2):41-48, June, 1995.
- [HRU96] V. Harinarayan, A. Rajarman, and J. Ullman. Implementing data cubes efficiently. In *Proceedings of the ACM SIGMOD International Conference of Management of Data*, Montreal, Canada, June 1996.
- [IK93] W.H. Inmon and C. Kelley. *Rdb/VMS: Developing the Data Warehouse*. QED Publishing Group, Boston, Massachusetts, 1993.
- [Kim95] R. Kimball. The aggregate navigator. *DBMS*, November 1995.
- [Mai96] D. Maier. Personal communication.
- [RCK95] N. Roussopoulos, C. M. Chen, and S. Kelley. The ADMS project: views “R” us. *Data Engineering*, 18(2):19-28, June, 1995.

- [SDNR96] A. Shukla, P. M. Deshpande, J. F. Naughton, and K. Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. *22nd International Conference on Very Large Databases*, Mumbai (Bombay), 1996.
- [SSV96] P. Scheuermann, J. Shim, and R. Vingralek. WATCHMAN: A data warehouse intelligent cache manager. In *Proceedings of the 22nd VLDB Conference*, Mumbai(Bombay), India, 1996.
- [Wid95] J. Widom. Research problems in data warehousing. In *Proceedings of the Fourth International Conference on Information and Knowledge Management*, pages 25-30, Baltimore, Maryland, November 1995.
- [YKL96] J. Yang, K. Karlapalem, and Q. Li. A framework for designing materialized views in data warehousing environment. *Technical Report HKUST-CS96-35*, October, 1996.
- [ZHKF95] G. Zhou, R. Hull, R. King, and J. Franchitti. Data integration and warehousing using H2O. *Data Engineering*, 18(2):29-40, June, 1995.

Appendix A

Class Diagrams of the Simulation System

In this appendix we present the class diagrams of the simulation system with the UML notation.

Figure 20 is the diagram for the data warehouse package, which contains data warehouse related classes, such as WareHouse, View, and Dimension.

Figure 21 is the class diagram for the algorithm evaluation package, which contains MavAlgorEval, MavControl, and Builder. MavAlgorEval is the class corresponding to the component Algorithm Evaluator. MavControl corresponds to MAV Controller and Builder serves as both Data Warehouse Builder and Query Generator.

Some classes, such as QueryPattern, are not shown in the class diagrams for simplicity. Only major classes are presented in the class diagrams.

Figure 20 Data warehouse package

Appendix B

Sample Output of the Simulation System

In this appendix, we present the three types of output of our simulation system.

B.1 Trace Output

The trace is the most detailed output, which records every materialization, dematerialization and query execution. A sample is shown in Figure 22. In the example, a query (0202) is executed and the cost indicates a dynamic materialization is necessary. Then a view (0002) is selected to be materialized. Since there is not enough space to materialize the new view, several materialized views (0003 and 0011) are thrown away. Finally the new view is materialized and the query execution continues. The next query (0002) is a drill-up of the previous one and can benefit from the last materialization.

```

Query id: 0200, card: 1.728e+10, name: day, brand, store, ,
Ref. view id: 0000, card: 1.3824e+11, name: day, prod_id, store, ,
Query id: 0202, card: 1.728e+09, name: day, brand, store, ,
Ref. view id: 0000, card: 1.3824e+11, name: day, prod_id, store, ,
To materialize id : 0002, card: 1.3824e+10, benefit: 1.86624e+13, name: day,
prod_id, store, ,
MavBuffer makeSpace, space needed: 1.36042e+10
0003,
0011,
space made: 1.3824e+10
Query id: 0002, card: 1.3824e+10, name: day, prod_id, store, ,
Ref. view id: 0002, card: 1.3824e+10, name: day, prod_id, store, ,

```

Figure 22 Trace output sample

B.2 Report Output

The report output is a brief overview of the running results. It records the average cost, running time, and space usage for each algorithm after its simulation. The sample output (Figure 23) shows the result for several static algorithms when the space limit is 10% of the total cardinality.

Space limit	10%	6.11811e+10			
<hr/> <hr/>					
StaticAlgor	Admission	AvgCost	SpcUsage	Time	
SIMPLE	NONE	3.4206e+10	97%	5097	
GREEDY	NONE	4.59749e+10	99%	9013	
GREEDY_SPACE	NONE	3.4206e+10	97%	27553	
FS	NONE	7.17574e+10	99%	273	
FS_SPACE	NONE	5.69401e+10	99%	859	
IFS	NONE	4.8116e+10	99%	214	
IFS_SPACE	NONE	5.78937e+10	99%	371	
FSG	NONE	4.74875e+10	99%	644	
FSG_SPACE	NONE	3.31631e+10	99%	15341	

Figure 23 Normal report output sample

B.3 Excel Importable Output

The excel importable output is used by Microsoft Excel to generate comparison graphs. It contains three tables, average cost, running time, and space usage, which use the 'TAB' character as the delimiter. The tables contain algorithms in the row and space limits in the column. A table of average cost is shown in Figure 24. In the table, the numbers in the first row are the space percentage of the total cardinality, and the first two columns are the static algorithms and dynamic admission algorithms, respectively.

	Ratio(avgCost/avgCard)	5	10	15	20	25
GREEDY_SPACE	NONE	245.541	216.511	204.469	182.916	179.749
GREEDY_SPACE	GREEDY	171.81	155.897	152.985	155.851	153.665
GREEDY_SPACE	GREEDY_SPACE	184.785	174.644	170.728	167.348	157.785
GREEDY_SPACE	FS	174.552	156.277	148.891	153.844	144.41
GREEDY_SPACE	FS_SPACE	186.692	166.832	160.631	156.108	145.814
GREEDY_SPACE	IFS	179.348	165.669	159.994	154.231	148.419
GREEDY_SPACE	IFS_SPACE	173.563	167.46	152.912	154.322	150.266

Figure 24 Excel importable output sample