2003

Data warehouse stream view update with hash filter.

Mohammed Shariful Islam
University of Windsor

In compliance with the
Canadian Privacy Legislation
some supporting forms
may have been removed from
this dissertation.

While these forms may be included
in the document page count,
their removal does not represent
any loss of content from the dissertation.

DATA WAREHOUSE STREAM VIEW UPDATE WITH HASH FILTER

by

Mohammed Shariful Islam

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science
in partial fulfillment of the requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada
2003

Your file Votre référence
ISBN: 0-612-86710-2
Our file Notre référence
ISBN: 0-612-86710-2

Abstract

A data warehouse usually contains large amounts of information representing an integration of base data from one or more external data sources over a long period of time to provide fast-query response time. It stores materialized views which provide aggregation (SUM, MIX, MIN, COUNT and AVG) on some measure attributes of interest for data warehouse users. The process of updating materialized views in response to the modification of the base data is called materialized view maintenance. Some data warehouse application domains, like stock markets, credit cards, automated banking and web log domains depend on data sources updated as continuous streams of data. In particular, electronic stock trading markets such as the NASDAQ, generate large volumes of data, in bursts that are up to 4,200 messages per second.

Updating data warehouse views computed from such stream updated base tables require frequent transportation of either huge updated base tables or the newly arriving streams of data, updated from the data source sites to the data warehouse site. This approach will slow down query response time and lead to a network bottleneck. Data warehouse views can be maintained incrementally with only arriving new stream data tuples by using a semi-join approach to improve on both the network traffic and response time of query. However, this semi-join based view maintenance method still suffers from the drawback of shipping long bytes of update stream of tuples through the network and performing repetitive joining operations even for arriving duplicate tuples.

This thesis proposes a new view maintenance algorithm (StreamVup), which improves on semi-join methods by using hash filters. The new algorithm first, reduce the amount of bytes transported through the network for streams tuples, and secondly reduces the cost of join operations during view update by eliminating the recompution of view updates caused by newly arriving duplicate tuples.

Key Words: Data Warehouse, Materialized View, View Maintenance, Stream Data, Semi-Join, Hash Filter, StreamVup algorithm.

Dedication

To my Mother, my Father and my Wife

Acknowledgements

There are a few people I would like to thank for their help in writing this thesis.

I would like to express my deep appreciation to my supervisor Dr. C.I. Ezeife for her continuous invaluable comments, encouragement, guidance and support during the thesis work. Without her help, this thesis would not have been completed.

I would also thank my internal reader Dr. Y.H. Tsin, his pieces of advice contributed a great deal to the quality of my thesis. I am thankful to my external reader Dr. Robert Schurko for being very accommodating and for finding the time to read my thesis. Special thanks also go to Dr. Robert Kent for chairing my thesis defense.

I would give my sincere gratitude to my beloved wife, for her endless love and everlasting support, which helped me to finish the graduate studies. Thank you and I love you.

I wish to thank my parents and my older brother who always stands behind me, loves me and supports me. Thank you, mom, dad and Bhayia.

Table of Contents

List of Figures

List of Tables

CHAPTER 1
INTRODUCTION

1.1 DATA WAREHOUSE

A data warehouse collects data from a variety of source databases. These source databases hold current data which are periodically populated into the data warehouse fact and dimension tables. Generally, a data warehouse system consists of a ***fact table*** and ***dimension tables***. The fact table is the table that stores the integrated data with some measurable aggregate attributes such as total sales, average sales etc. Attributes of the fact table are foreign keys representing subjects of interest, the integration attributes, the attribute representing the historical (usually time) and the measurable aggregate attributes. Dimension tables store detailed information related to foreign key attributes of the fact table. Thus, each primary key attribute of the dimension table uses as a foreign key attribute of the fact table.

Figure 1 shows the example of two source databases for a simple banking warehouse system is given by [Ez01]. The first source database is used for accepting deposits and withdrawals of money for the savings account customers, while the other source database is used for accepting deposits and withdrawals of money from the checking account.

Savings source database S1

custid	transtype	amount
0518	dep	500.00
0001	wd	200.00
0300	dep	300.00

customer (custid, name, address)
balance (custid, balance)
transaction (transtype, trname)

Checking source database C1

cid	trans	balance
c0001	dep	700.00
c0518	wd	1000.00

customer (cid, name, address)
transaction (trans, trname)

Figure 1: Banking source databases

Figure 1.1 presents the fact and dimension table of the data warehouse banking system, which is integrated from Savings and Checking source databases. Figure 1.2 shows a star schema for two banking source databases. The facts are organized as foreign key attributes (cid, acctype, transtype, time-m) and measurable attribute (amount). A dimension table describes the foreign key table of the fact table.

Fact-table

B-Activity (cid, acctype, transtype, time_m, amount)

Dimension tables

cust (cid, name, address)
acct (acctype, name)
transaction (trtype, trname)
time (time_m, min, hour, day, month, year)

Figure 1.1: Fact and Dimension table of the Banking data

For example, the star is able to answer ***multidimensional*** queries like "*Get the total number of deposits by each customer every day, then every month, and then every year*".

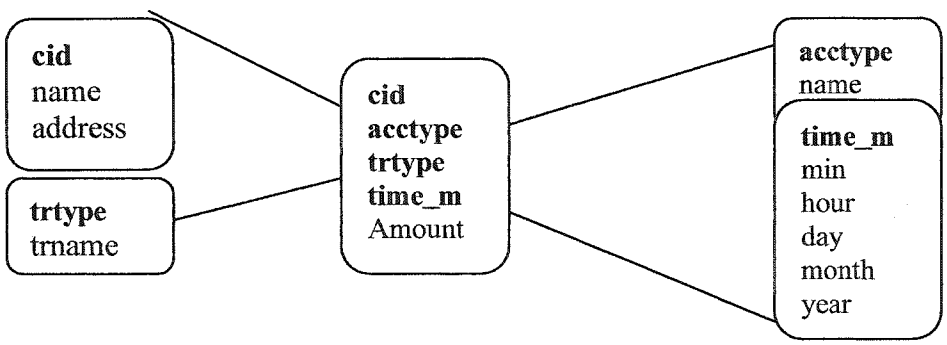


Figure 1.2 Star Schema for a Banking data warehouse system

cid	acctype	trtype	time_m	amount
0518	S1	dep	200307260850	500.00
0001	S1	wd	200307260910	200.00
c0001	C1	dep	200307260915	700.00
0300	S1	dep	200307260917	1000.00
c0518	C1	wd	200307260919	300.00

Figure 1.3: Two source databases periodically updated into the banking data warehouse

Two banking source databases hold current data which are periodically populated into the data warehouse fact and dimension tables. Figure 1.3 shows the historical data for a Banking data warehouse system.

W.H. Inmon in [In96] defines data warehouse as a "*subject oriented, integrated, time-variant, nonvolatile* collection of data to support decision support functions".

A data warehouse is the large database organized around major subjects (entities) of an enterprise, such as customer data, sales data, products data, etc. For example, a company might have a separate order processing system for sales such as retail sales, outlet sales, etc. This order processing system is not able to answer the details of all sales but the data warehouse is able to answer the details of all sales like "*Get the total number of sales by each province every year, then every month, week and hour*". A data warehouse is integrated from different source databases. It is designed for storing the historical database system representing data over a long period of time (up to 10 years). The structure of the data warehouse contains some element of time that able to answer the queries like: *"How much is the total sales record of Canadian Tire Stores in Ontario in the last 10 years?"*

When any new data is loaded into the data warehouse then the old data does not have to be updated. For example, the banking data warehouse may store information for 10 years and the new information will be moved into the data warehouse to refresh it without changing the data that is already in the data warehouse.

The data warehouse also contains ***materialized views*** which are derived from the base table. Therefore, the data warehouse is able to answer user queries by using the materialized views without accessing the remote databases. It is used to speed up the query response time since it is faster to access a materialized view than to recompute the corresponding query from the source databases or data warehouse fact tables. The main reason for developing the materialized views are: (1) identifying the views to be materialized, (2) selecting the materialized views to answer queries with a faster response time, and (3) efficiently updating the materialized views when source databases get updated.

1.1.1 Basic Architecture of a Data Warehouse

Before loading data into the warehouse first go through the process of extraction, transformation and data cleaning. Data extraction implements gateways and standard interfaces which are used for collecting data from multiple operational databases and external sources. The gateways and standard interfaces are Information Builders EDA/SQL, ODBC, Oracle Open Connect, Sybase Enterprise Connect, and Informix Enterprise Gateway, etc.

The data cleaning tools are used for detecting data anomalies and correcting them such as data migration and data scrubbing. When source databases change or update, incremental refreshing can be used.

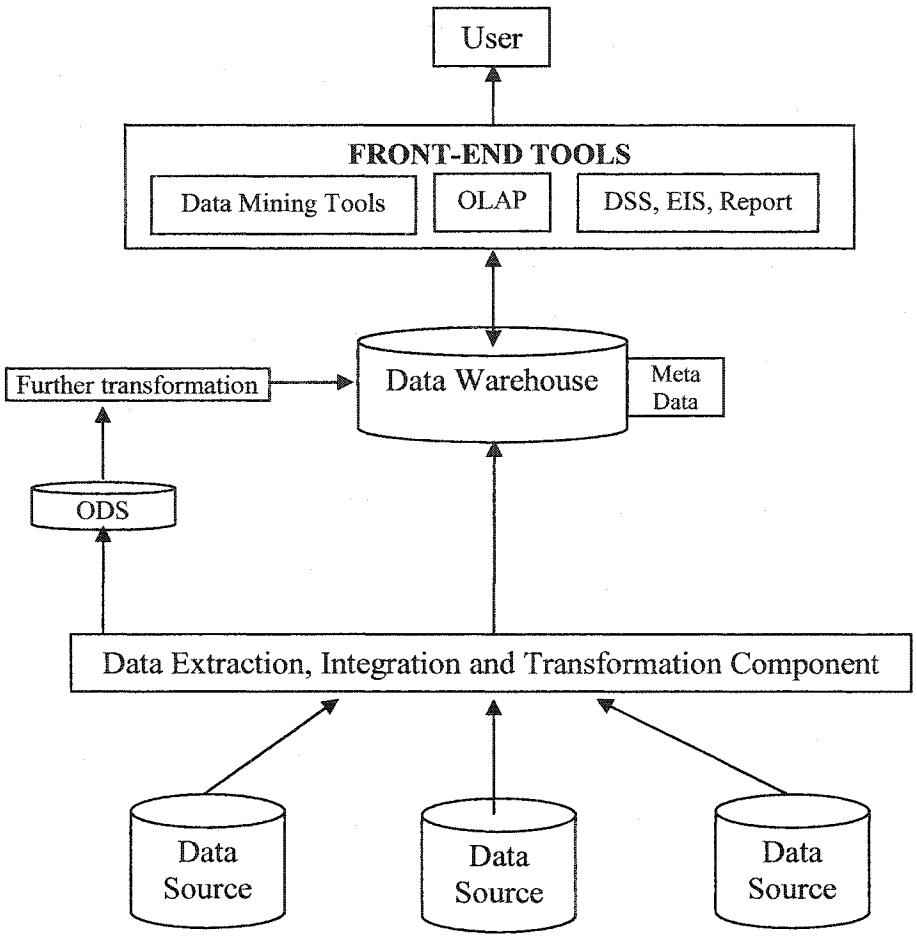


Figure 1.4: Basic architecture of a data warehouse

Figure 1.4 shows basic data warehouse architecture: external data sources, extraction, integration and transformation software, operational data store (ODS), data warehouse (DW), metadata and also have front-end tools such as online analytical processing (OLAP), executive information system (EIS), decision support system (DSS) and data mining tools, etc.

Data sources are the data that are integrated into the data warehouse fact and dimension tables. They can be in different formats, structure, and measurements such as relational databases, object-oriented databases, HTML files, XML files, flat (text) files, stream data and others. S*tream data* is appropriate when the data is changing constantly (often exclusively through insertions or new elements), and it is either unnecessary or impractical to work on a huge portion of the data multiple times. "Each stream has a fixed schema" [BW01]. For example, two streams of data <S999, ABC> and <S999, 345.00> for two base table S (ID, Name) and T (ID, Balance). Data streams occurs in a variety of modern applications such as network monitoring, stock exchange, sensor networks, manufacturing processes and weather or environment monitoring, etc.

Data Extraction, Integration and Transformation is a set of programs that extract data from the data sources and integrate them into the preferred format of the data warehouse. For example, one source database represents gender of customer Male as 'X' and Female as 'Y', and another source database represents 'M' as male and 'F' as female.

Operational Data Store (ODS) is optional in the data warehouse architecture and provide a centralized view of near real-time data from legacy systems. In a data warehouse ODS is used for the staging area before loading the data into the data warehouse. The data in a ODS is current, up-to-date and volatile or dynamic. ODS can also contains the data that is subject-oriented.

Metadata is a directory that describes the data in the data warehouse to users. It is data about data, and provides information such as the number of tables in the data warehouse, the number of rows in each table, etc.

Front–end tools are application system, such as Online Analytical Processing (OLAP), Decision Support System (DSS), Executive Information System (EIS), Data mining tools, etc. are used to answer the query for business decision-making.

1.2 DATA WAREHOUSE DESIGN

1.2.1 Multidimensional Data Model

A data warehouse stores materialized views of data from one or more sources, for the purpose of efficiently implementing to answer OLAP queries. The data in a warehouse is typically modeled multi-dimensionally, and the dimensions are often hierarchical. OLAP uses a multidimensional model to provide a solution, which also supports drill-down and roll-up analysis. ***Drill-down*** starts analysis from the highest level of the summarized data of the warehouse before getting to the most detailed level. For example, "*Get the total number of deposits of each customer every year, then every month, week, hour, minute and second*". ***Roll-up*** starts analysis from the most detailed level data of the warehouse before gradually getting to the highest level of summarized. For example, "*Get the total number of deposits by each customer every minute, then every hour, week, month and year*".

The multidimensional data model can be seen as a cube of data. "A data cube model is introduced that can represent warehouse data in many dimensions" [GBLP96]. A ***data cube*** is a database where a set of critical measure aggregate values, like total sales, is stored. "An n-dimensional data cube in relational OLAP is a table with 2^n sub-views of the data cube" [Ez01]. An example of a multidimensional question is "(i) *how much revenue did the new product generate by month* (ii) *in the northeastern division,* (iii) *broken down by user demographic,* (iv) *by sales office,* (v) *relative to the previous version of the product,* (vi) *compared with the plan?*" - a six dimensional question.

1.2.2 Maintenance of Materialized Views

There are two important problems related to view maintenance: 1) how to maintain materialized views when the base tables get updated, and 2) how to reduce the time for which the view is inaccessible during maintenance [CGL96]. Materialized views occur in a variety of modern applications such as the data warehouse, the mobile system, query optimization, etc.

There are three maintenance policies for maintaining views after the base tables have been updated. Those techniques depend on when the view is refreshed [CGL96]. The step that brings a view table up to date is called *refresh*. A view can be maintained in an immediate, a deferred, or a snapshot manner. There are three different view maintenance policies that have been proposed:

Immediate View Maintenance [BLT86, CW91, GL95, CGL96, CKL+97]: Refreshes a view, immediately updates only part of the transaction that is updated to the base table. Immediate maintenance allows fast querying, but comes at the cost of delaying update transaction that usually can not be tolerated in many applications such as the stock market, weather or environment monitoring, the banking system, etc.

Deferred View Maintenance [KLM+97]: Refreshes when the view is queried. It is separate from the update transactions and allows a view to become inconsistent in the short period of time. "Deferred maintenance leads to comparatively slower querying than immediate maintenance, but it allows faster updates" [KLM+97].

Snapshot View Maintenance [CKL+97]: The view is maintained periodically by asynchronous process. "Snapshot maintenance allows fast querying and updates, but queries can read data that is not up-to-date with base tables" [CKL+97]

Some papers [CGL96, KLM+97] classify deferred view maintenance and snapshot view maintenance together as deferred view maintenance.

1.3 DEFINITIONS

The following are some definitions relevant to this thesis:

Definition 1.3.1

A tuple is a row of information which carries values of attributes for the table. For example, in Table 1.1, the student relation has one row of information as {<**S#:**57>, <**NAME:** BROWN>, <**LCODE:** NY2092>}

Definition 1.3.2

A *semi-join* sends the joining attribute values of one relation to the site of the other and reduces the other relation by eliminating tuples which are not joinable. To compute the joining of R and S, where R and S are at different sites, shipping all tuples of R is expensive. In that case we can ship only those tuples of R that join with S. Semi-join can be defined as $R \propto S = \prod_R (R \bowtie S)$

$R \propto S$, the semi-join of R by S consists of the following steps:

1) The joining column of S is sent to the site of R

2) The tuples of R are scanned and the non-matching value of the column of S is deleted (thus, R is reduced).

S#	SNAME	LCODE
25	CLAY	NJ5101
32	THAISZ	NJ5102
38	GOOD	FL6321
17	BAID	NY2091
57	BROWN	NY2092

Table 1.1: STUDENT relation

S#	C#	Grade
32	8	89
32	7	91
32	6	62
38	6	98

Table 1.2: ENROLL relation

For example, Table 1.1 and Table 1.2 shows the information for STUDENT and ENROLL. Semi-join of ENROLL (S#,C#, Grade) to STUDENT(S#,SNAME, LCODE) on S# is: (written: ENROLL (S#) ∝ STUDENT). The SQL command for semi-join is SELECT * FROM **ENROLL R** LEFT SEMI-JOIN **STUDENT S** is simply "*SELECT DISTINCT R.* FROM R JOIN S.*" The following steps show how semi-join works for ENROLL S# ∝ STUDENT

1. Project ENROLL onto the S# attribute:

S#
32
38

2. Join the two relations on S#

S#
32
38

⋈

S#	SNAME	LCODE
25	CLAY	NJ5101
32	THAISZ	NJ5102
38	GOOD	FL6321
17	BAID	NY2091
57	BROWN	NY2092

Resulting in:

S#	SNAME	LCODE
32	THAISZ	NJ5102
38	GOOD	FL6321

Definition 1.3.3

Hash table stores the key in an array using the hash function. Hash function divides the data key by the length of array and remainder use as an index into the table. For example, in Figure 1.5 shows the range of indexes for Hash Table is 0 to 4 with 5 elements. Each element is addressing to the linked list of the numeric data.

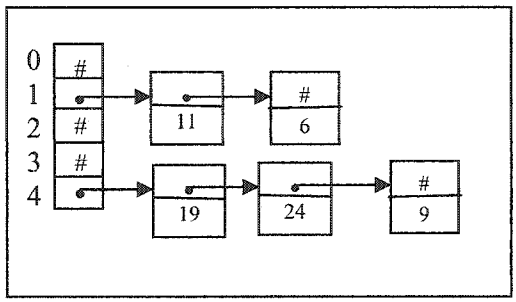
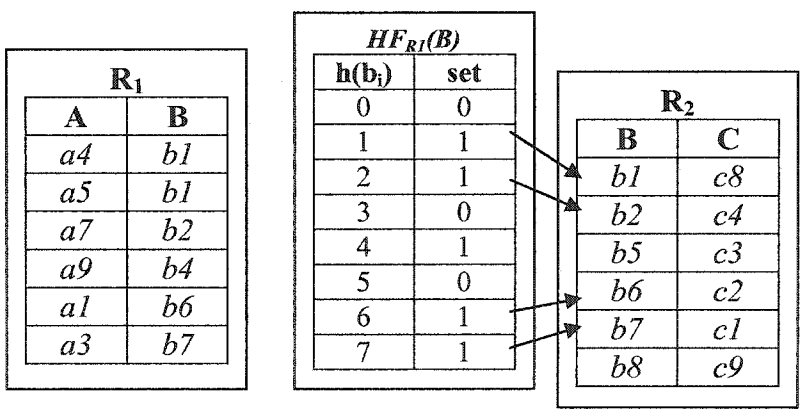


Figure 1.5: Hash table

For example, to insert 11, we divide 11 by 5 giving us two with a remainder of 1. Thus, 11 goes on the list starting at Hash table [1]. To find a number, we hash the number and chain down the correct list to see if it is on the table. To delete a number, we find the number and remove the node from the linked list. Entries in the hash table are dynamically allocated and entered on a linked list associated with each hash table entry.

Definition 1.3.4

"A hash filter is an array of bits which store only distinct values of attributes in an array before joining relations on their common attributes" [HCY94]. The following example



(a) (b) (c)

Figure 1.6: Use the hash filter $HF_{R1}(B)$ is build by R_1

$HF_{R1}(B) \rightarrow R_2$	
B	**C**
b1	*c2*
b2	*c1*
b6	*c4*
b7	*c2*

Figure 1.7: R_2 after the application of $HF_{R1}(B) \rightarrow R_2$

Figure 1.6 and 17 are given by [HCY94]. In figure 1.6 (b), relation R_1 store the attribute values of B in an array of bits 1 for true and 0 for false which is denoted by $HF_{R1}(B)$. Figure 1.6 (b) shows that the hash function stores same array of a bit, which is 1 for both tuples {<A:a1>, <B:b1>} and {<A:a2>, <B:b1>}in relation R_1 as shows of figure 1.6(a). The hash function in the hash filter selects the bits of $R_1(B)$, which are set to 1 (b1, b2, b4, b6, b7, b9) before joining with R_2. The $HF_{R1}(B)$ is next applied to R_2 which is denoted as $HF_{R1}(B) \rightarrow R_2$ such that only b1, b2, b6 and b7 will join with tuples of R_2 to yield the table in figure 1.7. The effect of the hash filter operation is reducing the overall cost of the cutting (e.g; both communication and memory cost) of $R_1 \bowtie R_2$.

1.4 THE MOTIVATION FOR THESIS

The most vital choice in designing a data warehouse materialized views to be maintained for implementing decision support or online analytical processing queries efficiently. It is very hard to select a set of derived views for reducing the total query response time and maintaining the selected views in a limited storage space. A new approach selects the sub-views from a set of 2^n data cube in order to reduce response time for the business decision support or OLAP query. Several algorithms on view maintenance have been described in this thesis report. Among them *Counting & Dred algorithm* [GMS93] and *ECA* algorithms [ZGH+95] are more important for maintaining a large class of materialized views. These algorithms use the view definition to compute the changes of the view by using the changes made to the base relations and the old materialized view. The most recent work on materialized views is rewriting the online analytical processing

queries, using materialized views in the data warehouse [PKL01]. Data Stream and online aggregation are recent techniques for identifying redundant views.

Materialized views are very effective for speeding up queries, when the number of new data entering into the data warehouse and the number of materialized views are quickly increasing. In [MRS+01], Sudarshan *et al.* improved greedy algorithm with Query DAG structure that can efficiently select materialized views to speed up queries, then updates can reduce maintenance cost.

OLAP queries increase the query execution cost, reducing performance and the productivity of business decision making. In [PKL01], Chang-Sup Park *et al.* proposed the algorithm to rewrite the OLAP query for using different types of materialized views that exist in the data warehouse. Recently, research work is increasing in designing algorithm for analyzing streaming data. In [DLY02], update the stream view efficiently from original sources to the data warehouse materialized views.

In this thesis, we propose a new algorithm, StreamVup that supports the data warehouse stream view update with a hash filter. In [DLY02], semi-join is used with an updated base table and other non-updated base table before producing delta view. The semi-join process will start when any update tuple arrives. We are using the hash filter to avoid using the semi-join for the new tuples. The hash filter keeps the information in an array and when any new tuple arrives it will first check the array table. If the existing array table has the information for the new tuples, then it will join with the existing rebuilt base table rather than rebuild it. This way we can save some joining costs when frequently updating the base table.

1.5 THESIS PROBLEM AND CONTRIBUTIONS

This thesis presents a new algorithm for efficiently update the stream data from the sources data to the data warehouse materialized views. In [DLY02], the semi-join method is used for updating the stream data to the data warehouse materialized views. The problem with this method is that semi-join works whenever any new update occurs in the base table, but it cannot reuse the same information if the new update tuple matches with the previous update tuple. Our proposed algorithm uses the hash filter instead of the semi-join to manage the resources efficiently such as CPU response time (the number of pages that can be processed per unit time), network bandwidth (the number of tuples that can be delivered by the network per unit time) and I/O ports (the number of tuples that can be scanned per unit time).

The hash function of hash filter keeps the result in an array of bits, so that when new update tuples arrive, the hash function will first check the existing array table before starting the joining to rebuild the new base table.

1.6 OUTLINE OF THE THESIS

The rest of this thesis is organized as follows: Chapter 2 reviews existing related work to the thesis. Chapter 3 presents detailed description of the new algorithms (StreamVup) for data warehouse stream view update with Hash Filter. Chapter 4 gives a performance analysis and Chapter 5 discusses conclusions and future research.

CHAPTER 2
PREVIOUS / RELATED WORK

In this chapter we review related previous work on data warehouse materialized view maintenance. Section 2.1 presents early research work on materialized views. Section 2.2 discusses later research works of Counting, DRed and ECA algorithms on materialized views. Section 2.3 presents early research on data cube lattice and maintenance of views with aggregation. Section 2.4 reviews recent work on selecting and materializing views horizontally while partitioning and propagation of updates from the original data sources to the data warehouse materialized views.

2.1 EARLY RESEARCH

Both of the *counting* and *DRed* algorithms use the view definition for incremental maintenance a large class of data warehouse materialized views to produce rules that compute the changes to the materialized views using the changes made to base table and the old views [GMS93]. They can be applied to SQL views that may or may not have duplicates, and that may be defined using union, aggregation, linear recursion and negation. A recursive view means that the view is defined using other views of the data warehouse rather than the data sources. "It proposes the counting algorithms for nonrecursive views, and the *DRed* algorithms for recursive views, since each one is better than the other on the specified domain" [GMS93].

2.1.1 Counting Algorithm

In [GMS93], the counting algorithm keeps track of the number of derivations (link from source node to destination) for a tuple *t* in the materialized view and only the number of the derivations (count) is interesting. The tuple is kept if it has at least one derivation, otherwise the tuple is deleted.

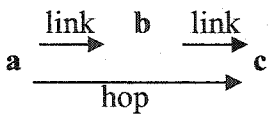


Figure 2.1: Link and Hop

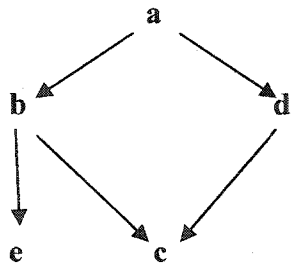


Figure 2.2: example of Link and Hop

For example, *Link (a,b)* is true if there is a link from source *a* to destination *b*, and *hop (c,d)* is true if *c* is connected to *d* via an intermediate node. Figure 2.1 shows *link* and *hop* and figure 2.2 shows *hop(a,e)* has a unique derivation {*link(a,b)* + *link(b,e)*}(here '+' is used for joining the links), whereas *hop(a,c)* has two derivations {*link(a,b)* + *link(b,c)*} and {*link(a,d)* + *link(d,c)*}. If we define count as the number of derivations then *hop (a,e)* has a count of 1 and *hop (a,c)* has a count of 2.

Suppose we delete the *link(a,b)* from figure 2.2, then we have to reevaluate the hop. Then the counting algorithm assume that one derivation of each the tuples hop(a,c) and hop(a,e) is deleted. The result is as follows:
hop (a,e) has no derivation and
hop(a,c) has one derivation {*link(a,d)* + *link(d,c)*}

So, *hop(a,e)* is deleted because *hop(a,e)* has no derivation but the *hop(a,c)* is kept because *hop(a,c)*, has one remaining derivation.

2.1.2 Deletion and Rederivation (DRed) Algorithm

The DRed algorithm [GMS93] applies to the recursive views that use negation and aggregation and have semantics with SQL views. The DRed algorithm computes the changes to the view if any change occurred at the base tables or relations.

For example, the *link(a,b)* is removed from figure 2.2. The relation hop is modified by using DRed, the first step is to compute an overestimate. The overestimate is all the tuples that its hop include the *link(a,b)*. Then *{(a,e),(a,c)}* is included into the set of overestimates. When other paths are traversed, it soon discovers that there is a path *(a,d)* and *(d,c)* that from the *hop(a,c)*. The second step of DRed removes the *hop(a,c)* from the set of overestimate. Finally, the overestimate set contains only *(a,e)* then the *hop(a,e)* is removed.

2.1.3 Eager Compensating Algorithm (ECA)

The ECA algorithm is used correctly only when the data warehouse keeps up-to-date copies of all relations involved in the views. This has some disadvantages: "1) The warehouse needs to store copies of all base relations used in its views and 2) Copies of relations at the warehouse need to be updated whenever an update occurs at the source" [ZGH+95].

So, it is a challenge to avoid the overhead of storing copies of base relations, since the sources can be simple systems that do not understand the materialized views in a warehousing environment [ZGH+95]. In [BLT86], describes an algorithm, which applies incremental changes to a view each time changes at the source are made to relevant base relations at the warehouse.

In summary, when an update U_i occurs in the source database S, the sequence of operations that must take place in order to update the warehouse view is:

1) Source sends update U_i to warehouse U^s_i
2) Warehouse queries the source database Q_i^{wh}.
3) Source database sends answer A_i of query Q_i to warehouse.
4) Warehouse updates the view by adding answer to the current view U_i^{wh}.

Thus, for the correct update of a warehouse, the sequence of action is: U_i^s, Q_i^{wh}, A_i, U_i^{wh}. However, if a later update U_i^s is recorded at the source before A_i is computed, giving a sequence like U_i^s, Q_i^{wh}, U_j^s, Q_j^{wh}, A_i, U_i^{wh}, A_j, U_j^{wh}, a warehouse update anomaly arises because an update that should not be used in the computation of the first answer has been used.

Figure 2.3 shows, how the ECA algorithm hands the view at the warehouse when the three insertions to the relations R, S and T. The warehouse view is defined by the relational algebra expression, $W = \Pi_W(R \bowtie S \bowtie T)$.

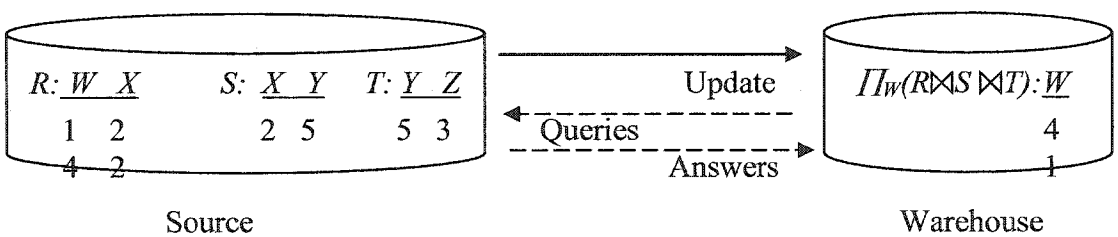


Figure 2.3: Processing of updates in a single source and a single view

The following example shows how ECA algorithm updates the view at the warehouse. Assume initially the view W is empty. This means the warehouse is initially empty. These three updates occur at the source before any queries are answered. Source evaluates Q and returns answer A and warehouse receives A and adds it to the view.

1. Source executes and sends U_1 = insert(R, [4, 2])

2. Warehouse receives U_1 and sends $Q_1 = \Pi_W$ ([4 , 2] $\bowtie S \bowtie T$).

3. Source executes and sends U_2 = insert (T, [5, 3]) for Q_1

4. Warehouse receives U_2 and sends $Q_2 = \Pi_W(R \bowtie S \bowtie [5, 3]) \ominus \Pi_W([4 , 2] \bowtie S \bowtie [5 , 3])$

5. Source executes and sends U_3=insert(S, [2, 5]) for Q_1

6. Warehouse receives U_3 and sends $Q_3 = \Pi_W(R \bowtie [2, 5] \bowtie T) \ominus \Pi_W([4, 2] \bowtie [2, 5] \bowtie T) \ominus \Pi_W(R \bowtie [2,5] \bowtie [5,3]) \ominus \Pi_W([4 , 2] \bowtie [2,5] \bowtie [5, 3])$ for Q_2

7. Source receives and evaluates Q_1, returns A_1 = [4]

8. Warehouse receives A_1= [4] and adds [4] to the view, $W = \varnothing$ + [4]

9. Source receives and evaluates Q_2, return A_2 = [1]

10. Warehouse receives A_2 = [1] and adds [1] to the view, W= [4] + [1]

11. Source receives and evaluates Q_3, return $A_3 = \varnothing$

12. Warehouse receives $A_3 = \varnothing$ and adds $\varnothing$ to the view, W = [4] + [1] + $\varnothing$

The updated view W at the warehouse is ([4], [1]). The result is correct.

In summary, the new algorithm (ECA) is good for correctly maintaining materialized views with respect to consistency in a warehousing environment. But it works with a restricted warehousing environment with only one source and one simple defined view.

2.2 LATER RESEARCH

2.2.1 Data Cube

Data warehouse star schema able to answer the multidimensional OLAP queries. The multidimensional data model can be seen as a data cube. "An n-dimensional data cube in relational OLAP is a table with 2^n sub-views of the data cube" [Ez01]. For example, a Banking data warehouse has one fact table B-Activity (cid, acctype, transtype, time_m, amount) and four dimension tables are Cust (cid, name, address), Acct (acctype, name), transaction (trtype, trname) and Time (time_m, min, hour, day, month, year). Figure 2.4 shows the data cube for the B-Activity table as a lattice structure. The dimension attributes of the data cube are cid (I), acctype (A), transtype (P), time_m (T) and amount, and the measurable attribute is SUM(amount). The measures computed are assumed to be the same, each point is annotated simply by the group-by attributes. The point (cid, acctype) represents the cube view corresponding to the query below.

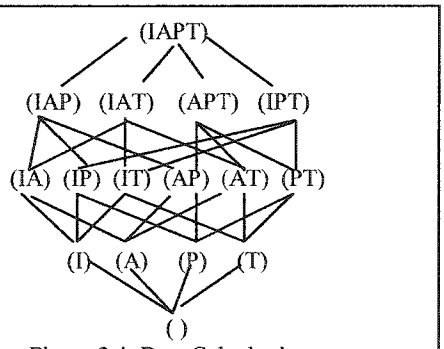


Figure 2.4: Data Cube lattice

(S1): SELECT cid, acctype, SUM (amount)
FROM B-Activity
GROUP BY cid, acctype

Each edges in a lattice can answered the below node without accessing the base table, like the edge v_1 = (*cid, acctype, time_m*) and the edge v_2= (*cid, acctype*). So, v_2 can answer the query from v_1, defining the following query equivalent to the query above.

(S1'): SELECT cid, acctype, SUM(amount)
FROM v_1
GROUP BY cid, acctype

S = {data cube top view}
for i =1 to K do begin
select that view v not in S such that B (v, S) is maximized;
$S = S \cup \{v\}$
end
resulting S is the greedy selecting;

Figure 2.5: The Greedy algorithm

Figure 2.5 shows the greedy algorithm which is defined in [HRU96]. The algorithm is selecting a set of k views to materialize for improving the space cost (number of rows in the view) for evaluating views. Suppose $C(v)$ is the cost of view v and k is the limit on the number of views. The algorithm will first select the top view before selecting some set S of views. "The total benefit B (v, S) is the sum over all views w of the benefit of using v to evaluating w, providing that benefit is position" [HRU96].

For example, consider the lattice of figure 2.6. Eight views, named a, b, c, d, e, f, g and h have space costs also shown on the figure. The top view a, with cost 100 must be chosen. If we want to choose two more views, then the greedy algorithm on this lattice must make two successive choices of the view to materialize.

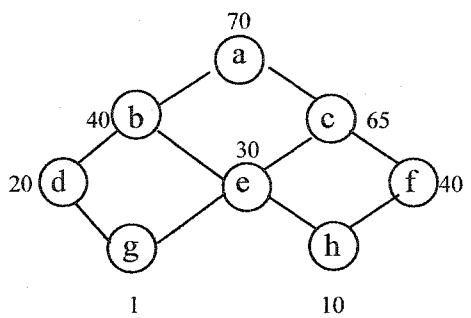


Figure 2.6: lattice with space costs

View	First Choice	Second Choice
b	**30 x 5 = 150**	
c	5x5 = 25	5x2=10
d	50 x 2 = 100	**20x2=40**
e	40 x 3 = 120	10x3=30
f	30 x 2 = 60	30+0=30
g	69 x 1 = 69	39x1=39
h	60 x 1 = 60	30x1=30

Table 2.1: Benefit of possible choices at each round

The column headed "First Choice" in table 2.1 gives us the benefit of each of the views besides *a*. Each view evaluated using *a*, therefore will have a cost of 100. For example, if view b materializes first, then it reduces by 30 its cost and each of the views *d*, *e*, *g* and *h* below it. The benefit of *b* is 5 times 5 or 25 in the first column of table 2.1. Another example, if the view *e* is picked, then it and the views below it, *g* and *h*, each have the cost reduced by 40, from 70 to 30 and the benefit of *e* is 120. So, only view *b* is selected to materialize in the first column. To recalculated the benefits for the second column, view (b, c, d, e, f, g, h) will be created from b, at cost of 50, if b is in view (b, c, d, e, f, g, h) or from a cost of 100, if not. For example, the benefit of c is 10 (from 70 to 65), 5 for itself and *f*. Choosing *f* yields a benefit of 30 for itself, from 70 to 40 and *h* yields a benefit of 0, from 40 to 40. So, only view *d* is selected to materialize in the second column. The table 2.2 shows the maximum benefits of the materialized views are selected by the greedy algorithm. If only the view was materialized then the total cost of evaluating all the view is 800. So, the cost would be reduced by 480, from 800 to 320, that cost is actually optimal after view *b* and *f* are selected by the greedy algorithm.

	Materialized View	Maximum benefit
First Choice	*b*	250
Second Choice	*f*	70
Total Benefit		320

Table 2.2: Materialized views are selected by the greedy algorithm

2.2.2 Maintenance of Views with Aggregation

Materialized views are involving aggregation to speed up the query and to summarize data in the data warehouse. As changes are made to the base data, materialized views become out of date. Materialized views can be maintained, either start recomputed from the beginning if the size of the view changes is larger or incrementally maintained by propagating changes to the base data onto the view.

Many algorithms have been presented to maintain the materialized view. Griffin and Libkin [GL95] provide an algorithm to maintain the materialized view by propagating changes (deletions and insertions) from base relations to a materialized view through each of the aggregations. But this algorithm does not support aggregation with group-by attributes. Dallan Quass [Qu96] extends the framework of [GL95] for maintaining materialized views with aggregation where aggregation support the group-by attributes. "It presents simple maintenance expressions for propagations, insertions and deletions, (except deletion with MIN and MAX) through aggregation operations" [Qu96]. The expressions are explained systematically in [MQM97] with propagate and refresh functions. "The aggregate functions are divided into three classes: distributive, algebraic and holistic" [GBL+96].

Distributive Aggregate functions COUNT, SUM, MIN and MAX are supported for computing by partitioning their input into disjoin sets [MQM97]. For example, COUNT can be computed by summing partial counts. If the DISTINCT keyword is used as in COUNT (DISTINCT *E*) (count the distinct values of *E*) then these functions are no longer distributive.

Algebraic Aggregate functions SUM and COUNT are used for computing AVERAGE function and it is algebraic, since it can be written as SUM/COUNT. If the view contains the AVG aggregate function, the data warehouse materialized views will contain instead the SUM and COUNT functions [MQM97].

Holistic Aggregate functions SUM, COUNT, MIN, MAX, AVG cannot be used for dividing into parts. Median, Mode and Rank are example of a holistic aggregate function [MQM97].

When the data warehouse is being updated, it is made unavailable to the user for querying. When changes occur in the source tables, most warehouses do not apply the changes immediately. It is usually maintained at regular intervals, like once a day, once a week, in a single batch window. For example, the changes from the sources are received during the day, and the views are refreshed during the night. Deferring the changes can make the maintenance more efficient, and the views can be frozen for analysis and other functions. Usually a data warehouse contains large amounts of data for a long period of time. Therefore, efficiently maintaining the summary tables is a critical issue. Mumick *et al.* in [MQM97] presents algorithms which can maintain a large set of summary tables defined over the same base tables efficiently. For simplicity of presentation, that paper assumes the fact table has been updated, and the maintenance has worked in response to the changes only to the fact table, not source tables. It gives a definition of self-maintainable aggregation: A set of aggregate functions can be self-maintainable if it is computed from old values of the aggregate functions and from the changes to the base table [MQM97]. The aggregation functions COUNT, SUM, AVG, MIN and MAX are self-maintainable with respect to insertion. For example, the new value of function MAX can be computed by comparing the old value of MAX aggregation function with the change to the base table. If the change is larger than the old value, then the new value of MAX should be the changed. Since AVG can be computed as SUM/COUNT, it is also self-maintainable. COUNT is self-maintainable, but COUNT (DISTINCT E) is not. Not all aggregate functions are self-maintainable with respect to -deletion. For example, MIN and MAX are not self-maintainable with respect to deletion.

After deletion a tuple which contains the minimum or maximum value, the new minimum or maximum value must be recomputed if there are changes to the base data. Adding a COUNT (*) aggregation function to view can help to handle deletions in some situations. If the value of COUNT (*) reaches zero, there is no other tuple in the group that can be deleted. The use of COUNT (*) will be shown on an example later.

The view maintenance can be processed into two steps: *propagate* and *refresh*. For the propagate function, this paper presents a new method which creates summary-delta tables. It contains the net effect of the changes on a summary table, for the incremental maintenance of aggregate views. It defines three virtual views for a summary table that needs to be maintained: *prepare-changes*, *prepare-insertions* and *prepare deletions*. The prepare-insertions and prepare-deletions views are derived from the changes in the base relations caused by individual insertions and deletions to the aggregate functions. They contain every group-by attribute of the summary table and aggregate-source attribute that each of the aggregate function computes in the summary table. For example, if the summary table contains group-by attributes such as customer id (cid), Account type (Acctype), and aggregation function SUM (Amount), the prepare-insertions and prepare-deletions views should contain cid, acctype, and aggregate-source attributes Amount for insertion, or - (Amount) for deletion. The prepare-changes view is computed as the union of prepare-insertions view and prepare-deletions view. The table 2.3 shows how to derive the aggregation-source attributes is given by [MQM97].

	prepare-insertions	*prepare-deletions*
COUNT(*)	1	-1
COUNT(expr)	case when expr is Null then 0 else 1	case when expr is null then 0 else -1
SUM(expr)	expr	- expr
MIN(expr)	expr	- expr
MAX(expr)	expr	- expr

Table 2.3: Deriving Aggregation-source Attributes

The summary-delta table is computed from prepare-changes virtual view with the same schema as the summary table except the attributes that result from the aggregate functions in the summary-delta table present changes to the corresponding aggregation functions in the summary table. In the refresh step the changes in the summary-delta table are applied to the summary table. Since each tuple δt in the summary table cause a change to a corresponding tuple t in the summary table, for each δt, t is looked up. If t is not found it means no tuples in the summary table have same values of group-by attributes as δt into the summary table. If t is found, the algorithm checks if the sum of COUNT (*) from δt plus COUNT (*) from t is zero. If it is, then t is deleted. Otherwise, it checks each of the MIN and MAX aggregate function to see if the new MIN and MAX values should be recomputed from the base data for t's group. The recomputation is performed when the value δt.MIN (e) (δt.MAX(e)) is less than or equal to t.MIN(e) (greater than or equal to t.MAX(e)) was deleted. This algorithm only considers COUNT, SUM, MIN and MAX, not including AVG.

2.3 RECENT RESEARCH

2.3.1 Selecting and Materializing Horizontally Partitioned Warehouse Views

Materialized views are selected from the data cube for speeding up query response time for data warehouse view maintenance cost. In [Ez01], C.I. Ezeife presents the horizontally fragments method for recomputing the size of the stored partitioned view in the data warehouse materialized views.

Data warehouse aggregate views can be stored as 2^n subviews of a data cube with n attributes answer the data warehouse multidimensional OLAP queries for business decision making. The cost of the data warehouse materialized view maintenance is increasing when 2^n views are stored because these views are huge. In [HRU96], V. Harinarayan *et al.* proposed a greedy algorithm for selecting the best views to materialized in order to answer the queries with minimum response time. In [Gu97], H. Gupta extended the greedy algorithm to select both view and index. In [Ez97a], C.I. Ezeife defines a uniform schema based on a comprehensive cost model for selecting both view and index. In [Ez97b], C.I. Ezeife extends this uniform schema to handle dimension hierarchies. In [OV91], M.T. Ozsu *et al.* presented horizontal fragmentation schemas for relational databases based on simple predicates and with no access frequencies taken into consideration. In [EB98], C.I. Ezeife *et al.* proposed a partition selection schema for partitioning any selected view by using the re-computed size of the partitioned view and the fragment-advisor algorithm finds the best view for answering the user queries. The partitioned view is based on the actual fragments of the view scanned by queries. The greedy algorithm is applied to further selection of the views. In [Ez01], C.I. Ezeife extends the greedy algorithm and proposes selection-partitioning algorithm for recomputing the size of the partition views. The selection partition algorithm selects n

best views to materialize. Algorithm first selected the top level view from the data cube lattice. After that, it is recomputed the size of the horizontal fragmented view. The view partition algorithm finds the simple predicates from each user query by using the partition attributes. Simple predicates are defined from the partition attributes value with the help of logical operator from the set {=, <, >, ≠, ≤, ≥}and the value is from the domain of partition attributes. "The importance of each predicate is obtained by adding up the product of the application frequency and the cardinality of this predicate on view for every application that access the predicate" [Ez01].

The formula for obtaining the importance of a predicate is given as:

$$\sum_{Q_i \mid \text{access}(Q_i, P_{ik}) = 1} (\text{access frequency of } (Q_i) * |P_{ik}|),$$

The number of rows or tuples in the partition of the materialized view is defined by predicate P_{ik} , while access $(Q_i, P_{ik}) = 1$ means that the number of time query Q_i accesses the predicate P_{ik}. The view partition schema also produces the horizontal fragments of the materialized view by minterm predicates. "A minterm predicate is a conjunction of simple predicates in either their natural or negated forms" [OV91].

The fragment advisor algorithm finds the best answers to a query from a set of fragments of the view. Each fragment comes with the conjunctive minterm predicate defines which the minterm predicate is true. Algorithm first set all attributes that is needed by the query's partition attributes (PA), analysis attributes (AA) and measure attributes (MA) sets then finding the best view with fewer rows can answer the user query. So this way fragments advisor help to find out the lowest total number of rows for answering the user query.

2.3.2 Propagation of updates from original data sources to materialized views

In recent years some research has been done to synchronize the views when the data sources are updated. The view maintenance on updating is much more difficult when data sources and views are at the different locations, a view synchronization problem occurs. That is, views need to be updated according to the updates to the base table. So far there are two ways to solve this problem: using self-maintainable views and using updategram plus boosters.

In [DLY02], Xin Dong *et al.* trying to solve this problem as follows: update the data warehouse materialized view f-rom the two base tables. One base table is located at the updategram side and another base table is located at the booster side. Only the updategram side base table collects updated into the updategram, and send it to the booster side and data warehouse materialized view side at the same time. The base table at the booster side semi-joins with new updated tuples of the updategram, and then sends it to the data warehouse materialized view to produce delta view. After that delta view merge with the data warehouse original materialized view.

Updategram has two hash pools, one is insert pool and another is delete pool. Insert pool collects only insert items of the updated tuples and delete pool collects only delete items but before putting the data into the delete pool, it checks the insert pool for checking whether the same tuple appears in the insert pool or not. If so, then the tuple will removed from the insert pool and the deletion item is discarded. This way, it is removed the overlaps as soon as possible when the new tuples are updated into the updategram. Figure 2.7 shows the updategrm data structure after updating insert tuple: <a1,b1>, delete tuples <a2,b2> and <a3,b3>. The delete tuple <a2,b2> is discarded because the same tuple appears in the i-pool.

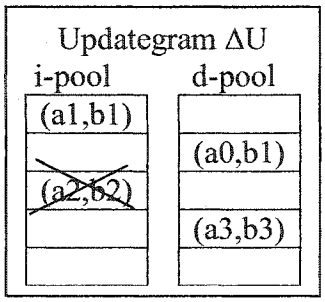


Figure 2.7: Updategram data structure

The booster side semi-joins S with the updategram tuples <b1,c2> and <b3,c3> to produce two boosters <b1,c2> and <b3,c3> is shown figure 2.8.

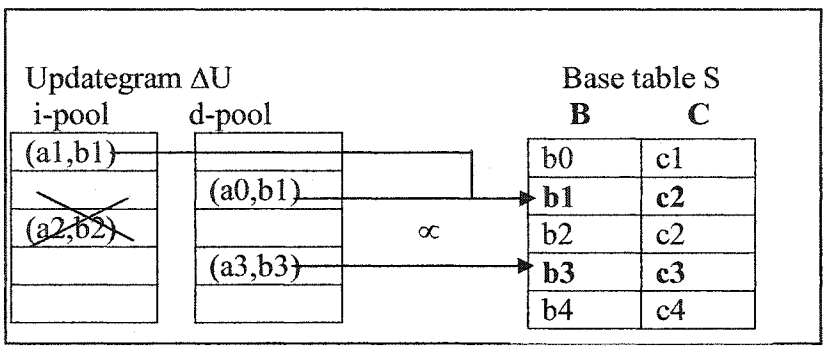


Figure 2.8: Booster side semi-joins with S

At the view side booster item first join with the delete pool of the updategram then it can check the insert pool of the delta view for removing the same tuple before putting it into the delete pool of delta view. The result of the delta view pools are updated same way as for the updategram. For example, Figure 2.9 shows tuple <a3,b3,c3> is produced after updategram tuple <a3,b3> of the d-pool joins with <b3,c3>. Before putting it into the d-pool of the deltaview, we first check whether the same tuple appears in the insert pool. This tuple <a3,b3,c3> is discarded because the same tuple appears in the i-pool of the d-pool, then delivers it to the view side. The view side delta view join with the updategram can be defined as, ΔView = updategram ⋈ booster and the delta view merge into the original data warehouse materialized view can be defined as, View′ = View ⊕ ΔView.

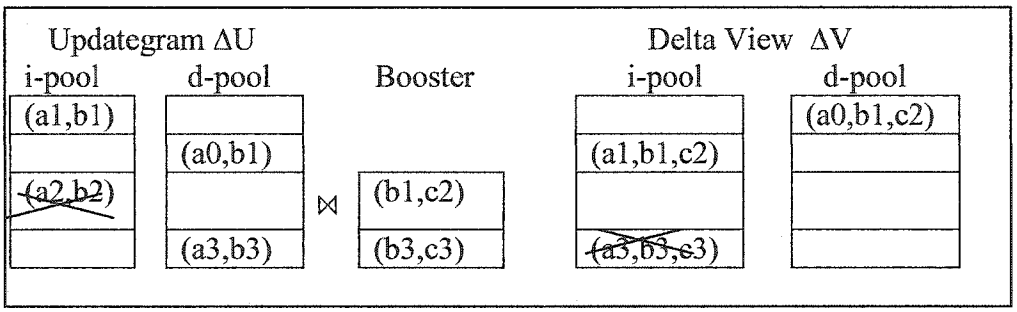


Figure 2.9: Joins the updategram and the booster to produce ΔV

In summary, the proposed algorithm [DLY02] is good for managing the propagation of updates from the original data sources to materialized views. But this algorithm performance will decrease when the update in the base table is too much.

CHAPTER 3
DATA WAREHOUSE STREAM VIEW UPDATE

3.1 INTRODUCTION

A data warehouse stores large amount of data to supports on-line analytical processing (OLAP). "Decision support and OLAP application make heavy use of complex grouping / aggregation queries" [The01]. There are two types of materialized views can be seen into the data warehouse: *simple views* and *auxiliary views*. Simple views are used for optimally answering the OLAP queries and auxiliary views are used for reducing the data warehouse view maintenance cost [The01]. D. Quass *et al.* in [QGM+96] presented an algorithm to make a view self-maintainable by using several auxiliary views stored in the same location. A view maintenance in a data warehouse requires access to data that is not available in the view itself because, while the materialized views are available for view maintenance, access to the remote database may be restricted or expensive. Self-maintainable views are useful to maintain a data warehouse views by using several auxiliary views stored in the same location [DLY02]. It allows views to be maintained using only the materialized view without accessing the base relations.

When data sources and views are at different locations, then a view synchronization problem occurs; that is, views need to be updated according to the updates in the base table. Stream update propagation using *updategram*, which contains updated tuples, (both insert and delete tuples), and *boosters*, contain tuples which will join with updategram to generate delta view (updategram $\bowtie$ booster is called delta view) to produce delta view update at the view site [DLY02]. The proposed algorithm uses *updategramfilter*, which generates distinct tuples of the updategram to join with the booster site base table. The updategramfilter avoids transmitting the whole updategram data in order to reduce data traffic. For example, if the view V joins with both two base tables R and S, all of which are located on different locations, if one of the tables R is updated, which is denoted as ΔR. ΔR will send into the updategramsfilter, which rebuild new S after joining with ΔR

and existing S at the booster side. The updategram joins with the booster to generate delta view, which is denoted as ΔV. This proposed technique saves a great deal of transportation and computation. Maintaining a set of auxiliary views means more storage space and more cascading maintaining computation at the view side. This algorithm divides the workload among the booster site and the view site and divides the resource usage among the network bandwidth (the number of tuples that can be delivered by network per unit time), I/O (the number of tuples that can be scanned per unit time) and CPU (the number of pages that can be processed per unit time).

In many applications, such as stock market databases and sensor network databases, source data is updated much more frequently than users query the views. These queries require up-to-date data. We propose the ***Data Warehouse Stream View Update with Hash filter (StreamVup)*** algorithm based on the Hash filter (discussed in section 1.3) and Hash table (discussed in section 3.2) techniques which is capable of efficiently managing the propagation of updates from original sources to materialize views by using Hash filter (HF) technique at the booster side. Section 3.2 defines several terms in order to present the algorithm formally. Section 3.3 describes the proposed algorithm with example. The proposed algorithm updates the data warehouse materialized view using *Propagate* and *refresh* [MQM97]. *Propagate* has three sub stages which are *updategram*, *updategram-filter and booster*.

3.2 THE PROBLEM DOMAIN

Assume share holder no. 45459 decides to move some savings into company A's stocks. He calls his broker and places an order to buy 750 of A's shares at a maximum price of $11.89. This is called *a bidding price*. Share holder no. 45459's broker uses the stock market's trading system to broadcast share holder no. 45459's bid. When it hits the market, the best offer for A's shares, called its *asking price*, is $11.89. Share holder no. 38387 wants to sell some of his shares of A. He instructs his broker to sell 1000 of his A's shares at market value. When his offer meets share holder no. 45459's bid, a trade is done. Each 'bid', 'ask', 'sell' or 'buy' is called a *tick*. There are up to 100 million ticks

(six-and-a-half hours a day) that can be achieved in some electronic trading markets such as NASDAQ.

To facilitate our discussion, let us introduce an imaginary example. The following is an example of the data warehouse for the trading activities of the stock market. We will use this example throughout this chapter to show how our proposed **StreamVup** works.

A sample RTicks table data is given in table 3.1 for only 10 tuples although this table holds millions of rows typically. This RTick holds continuous streams of data in a memory. At the end of the day RTick will erase all shares information records. Ask and bid will be erased from the memory. The RTick schema is (*ID, Timestamp, SID, MMID, PRICE, Volume, Type*) where ***ID*** is security identifier, ***TIMESTAMP*** corresponds to the system clock when tick enters the trading system from a single stream. There are up to 100 million (100 x 1,000,000) ticks per six-and-a-half-hours (about 4,200 ticks per second). The timestamp transaction which took place is recorded as year/month/day/ minute/second [Ez01]. Since in a day there are 1440 minutes (24 x 60), these four digits represent both minutes and hours of a day. The last two digits of time are used to represent seconds (1 x 60). For example, first record of the timestamp in table 3.1 is 20030503093245. The first four digits indicates the year, which is 2003; the next four digits represent the month and day, which is May 3 (0503); the next four digits represent minute and hour, which is the 9^{th} hour and the 32^{nd} minute (0932); and the last two digits represent the seconds of a minute, which is 45 sec. ***SID*** is a stock ID, ***MMID*** is the Market Maker behind this tick and price. Other attributes are ***VOLUME*** and ***TYPE*** which is either 'bid', 'ask', 'sell' or 'buy'. The *RTick* is shown in table 3.1. Trading activities source database has two base tables *R* and *S*. The base table *R* holds the current share information records, which is updated when any share is sold or bought in the RTicks. The schema for *R* is (*Timestamp, MMID, SID, Price, Volume, Type*), which is shown in table 3.2 for only 5 MMIDs as 34349, 38459, 41258, 43458 and 45459 although this table holds millions of MMIDs. Data warehouse TicksFT stores all the historical data from table R, when any new data is updated in the R. At the same time it will send them into the TicksFT, which is shown in Table 3.4. No transactions by the same person occur at exact same second, but transactions by different person occurring

ID	TIMESTAMP	MMID	SID	PRICE	VOLUME	TYPE
1000001	20030503093245	38387	AC	20.15	600	Bid
1000002	20030503093245	41258	BG	50.18	100	Bid
1000003	20030503093245	43458	AC	20.15	600	Ask
1000004	20030503093245	45457	BG	50.18	100	Ask
1000005	20030503093245	45459	BG	50.17	500	Bid
1000006	20030503093247	47478	BG	50.17	500	Ask
1000007	20030503093247	38387	AC	21.10	600	Ask
1000008	20030503093248	41258	BG	51.07	100	Ask
1000009	20030503093248	43458	AC	21.10	600	Bid
1000010	20030503093249	45457	BG	51.07	100	Bid

Table 3.1: An Instance of the most recent trading activities for RTicks

within the same minutes (eg. BUY and SELL) are shown in table 3.2. The base table *S* holds the details of share information records. The schema for S is (*SID, StockName, PhaseValue*) which is shown in Table 3.3 for only 7 shares information, although this table holds thousands of shares information. The R and S are the base tables of the *trading activities source database* and are shown in table 3.2 and table 3.3 respectively. Table 3.2 shows the records of trading activities at time 200307300950 and table 3.4 shows the records of trading activities at time 200307300953.

MMID	SID	PRICE	VOLUME	TYPE	TIMESTAMP
38387	AC	20.15	600	BUY	20030730095015
41258	BG	50.18	100	BUY	20030730095016
43458	AC	20.15	600	SELL	20030730095018
45457	BG	50.18	100	SELL	20030730095021
45459	-	-	-	-	-
47478	-	-	-	-	-

Table 3.2: An Instance for R in time 2003,07,30,0950

SID	STOCKNAME	Phase Value
AC	ACME	10
BA	British Airways	25
BG	BG Group	15
CP	Compuware	5
MI	MIG	30
NR	Northern Rock	10
SP	Scottish Power	50

Table 3.3: An Instance for S

MMID	SID	PRICE	VOLUME	TYPE	TIMESTAMP
38387	**AC**	**21.10**	**600**	**SELL**	**20030730095315**
41258	**BG**	**51.07**	**100**	**SELL**	**20030730095319**
43458	**AC**	**21.10**	**600**	**BUY**	**20030730095320**
45457	**BG**	**51.07**	**100**	**BUY**	**20030730095317**
45459	BG	50.17	500	BUY	20030730095323
47478	BG	50.17	500	SELL	20030730095325

Table 3.4: An Instance for R in time 2003,07,30,0953

We assume that our simple *stock* warehouse system is used for only historical data from base table R of the *trading activities database* since there is no integration with other source databases. Figure 3.1 and 3.2 shows the data warehouse fact, dimension tables and star schema. Table 3.5 – Table 3.8 shows an instance of TicksFT for data warehouse, of stock dimension table, of market maker dimension table and of time dimension table

TicksFT (Timestamp, SID, MMID, Price, Volume, Type)

Stock (SID, StockName, PhaseValue)

MM (MMID, Name, address, phone)

TickTime (Timestamp, day, month, year)

Figure 3.1: Schema of the fact and dimension table for data warehouse

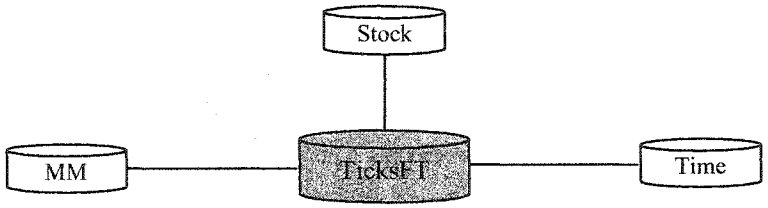


Figure 3.2: Data warehouse star schema

TIMESTAMP	SID	MMID	PRICE	VOLUME	TYPE
20030730095015	AC	38387	20.15	600	BUY
20030730095016	BG	41258	50.18	100	BUY
20030730095018	AC	43458	20.15	600	SELL
20030730095021	BG	45457	50.18	100	SELL
20030730095315	**AC**	**38387**	**21.10**	**600**	**SELL**
20030730095317	**BG**	**45457**	**51.07**	**100**	**BUY**
20030730095319	**BG**	**41258**	**51.07**	**100**	**SELL**
20030730095320	**AC**	**43458**	**21.10**	**600**	**BUY**
20030730095323	**BG**	**45459**	**50.17**	**500**	**BUY**
20030730095325	**BG**	**47478**	**50.17**	**500**	**SELL**

Table 3.5: An Instance of TicksFT for Data Warehouse

SID	STOCKNAME	PhaseValue
AC	ACME	10
BA	British Airways	25
BG	BG Group	15
CP	Compuware	50
MI	MIG	30
NR	Northern Rock	10
SP	Scottish Power	50

Table 3.6: An Instance of Stock Dimension Table

MMID	Name	Address	Phone
38387	Smith	3233 Peter St	253-3012
41258	Bob	1487 Josephine Ave	231-5897
43458	Alice	1875 Partington Ave	562-8954
45457	Joe	2356 Pelletier St	532-6532
45459	John	1121Bridge Ave	326-5687
47478	Mike	686 Church Street	589-4512

Table 3.7: An Instance of Market Maker Dimension Table

Timestamp	Second	Minute	Day	Month	Year
20030730095015	15	0950	30	07	2003
20030730095016	16	0950	30	07	2003
20030730095018	18	0950	30	07	2003
20030730095021	21	0950	30	07	2003
20030730095315	**15**	**0950**	**30**	**07**	**2003**
20030730095317	**17**	**0950**	**30**	**07**	**2003**
20030730095319	**19**	**0953**	**30**	**07**	**2003**
20030730095320	**20**	**0953**	**30**	**07**	**2003**
20030730095323	**23**	**0953**	**30**	**07**	**2003**
20030730095325	**25**	**0953**	**30**	**07**	**2003**

Table 3.8: An Instance of Time Dimension Table

We also assume that two data warehouse stored views as follows: the first view Ticks is denoted as V_1 and the second view Ticks HISTORICAL is denoted as V_2. The V_1 records "*Get the number of SID, stock name, phase value and the balance of the Volume for each MMID by SID*". The V_2 view records "*Get the maximum, minimum price and total volume of the share sold, by each MMID and SID in every month*". V_1 and V_2 have the following heading formats:

V_1 (MMID, SID, StockName, PhaseValue, BalVolune)

V_2 (MMID, SID, MaxPrice, MinPrice, PhaseValue, MonthlyTotal)

For simplicity of discussion, we are using view V_1 in the data warehouse, which is $V_1 = R \bowtie S$. We assume R receives continuous data records while S does not change meaning, and the size of R keeps growing but the size of S remains the same. Whenever a new tuple arrives in R, V_1 needs to be updated. We can update the V_1 simply by moving each tuple t_c (t_c is the new tuple) of R that arrives into the data warehouse, so that $t_c \bowtie V_1$ updates V_1. The view V_1 from table R and S, which is shown in the table 3.9:

MMID	SID	StockName	PhaseValue	BalVolume
43458	**AC**	**ACME**	**10**	**600**
38387	AC	ACME	10	600
41258	BG	BG Group	15	100
43458	**AC**	**ACME**	**10**	**-600**
45457	BG	BG Group	15	-100

Table 3.9(a): View $V' = t_c \bowtie V_1$

MMID	SID	StockName	PhaseValue	BalVolume
38387	AC	ACME	10	600
41258	BG	BG Group	15	100
45457	BG	BG Group	15	-100

Table 3.9(b): View V'

If the new tuple of R arrives, we need to update exit V_1 record <MMID: 43458, SID: AC, StockName: ACME, PhaseValue:10, BalValue: 600> with this new tuple t_c. We can simply move this tuple to V_1 and $V' = t_c \bowtie V_1$ is shown in Table 3.9(a) (here t_c is new tuple at time 200307300950 which is shown in table 3.2). Table 3.9(b) shows the new view after updated new tuples.

The above traditional view update technique cannot update materialized views when frequently updating at the base table *R* (e.g. 4,200 records per second). That is why, in [DLY02] proposes the streamed update algorithm from original sources to materialized

view using semi-join. When an update occurs in the base table *R*, the sequence of the operations that must take place in order to update warehouse view is:

1) The update base table sends update to the other non-updated base table *S* for semi-join Figure 3.9(c) shows the result of the semi-join with new updated tuples which is shown in Table 3.2.

SID	STOCKNAME	PhaseValue
AC	ACME	10
BG	BG Group	15

Table 3.9(c): R $\propto$ S

2) Semi-join result will join with new updated tuples which is shown in Table 3.2 before updating the warehouse materialized view.

The advantage of semi-join is reducing the joining cost rather than joining the whole base table of *S*. This approach is not efficient to update the materialized views because it needs semi-join when updated tuple arrives. So, we need alternative approach to update the view efficiently into the data warehouse. Our proposed algorithm solves the materialized views update problem using hash filter and hash table (discussed in section 1.3).

3.3 THE PROPOSED DATA WAREHOUSE STREAM VIEW UPDATE ALGORITHM BASED ON HASH FILTER AND HASH TABLE

We propose a new algorithm for the data warehouse stream view update based on hash filter and hash table (Figure 3.4). Algorithm efficiently manages the propagation of updates from original sources to materialized views. We use the strategy described in [CGL96], [MQM97], [EX00], [DLY02] which splits the maintenance work into *Propagate* and *Refresh* functions in our algorithm. Our algorithm has two main functions: propagate and refresh, which are based on hash pool. The Propagate function is used to compute the net effect of changes on a view based on the changes that occurred on the base table. There are three sub-functions *updategram, updategramfilter,* and *booster* that help to create the delta view.

The updategram side packs updated table R, which is denoted as ΔR. ΔR tuples are sent into the updategram, and updategramfilter simultaneously. The booster side rebuilds S when R is updated, which is denoted as ΔR. A Hash filter is applied on ΔR to get the distinct value of attribute SID in ΔR. S is rebuilt after the application of $HF_{\Delta R}(SID) \rightarrow S$, to produce the booster and then send it into the view side. The view side joins the updategram and the booster to produce its update, delta view (ΔV). Finally, the refresh function applies the changes represented in the ΔV to the V_1.

Figure 3.3 shows the data warehouse stream view update algorithm work flow. When the job at the source database site is completed, the view information will send into the data warehouse site for updating data warehouse materialized view.

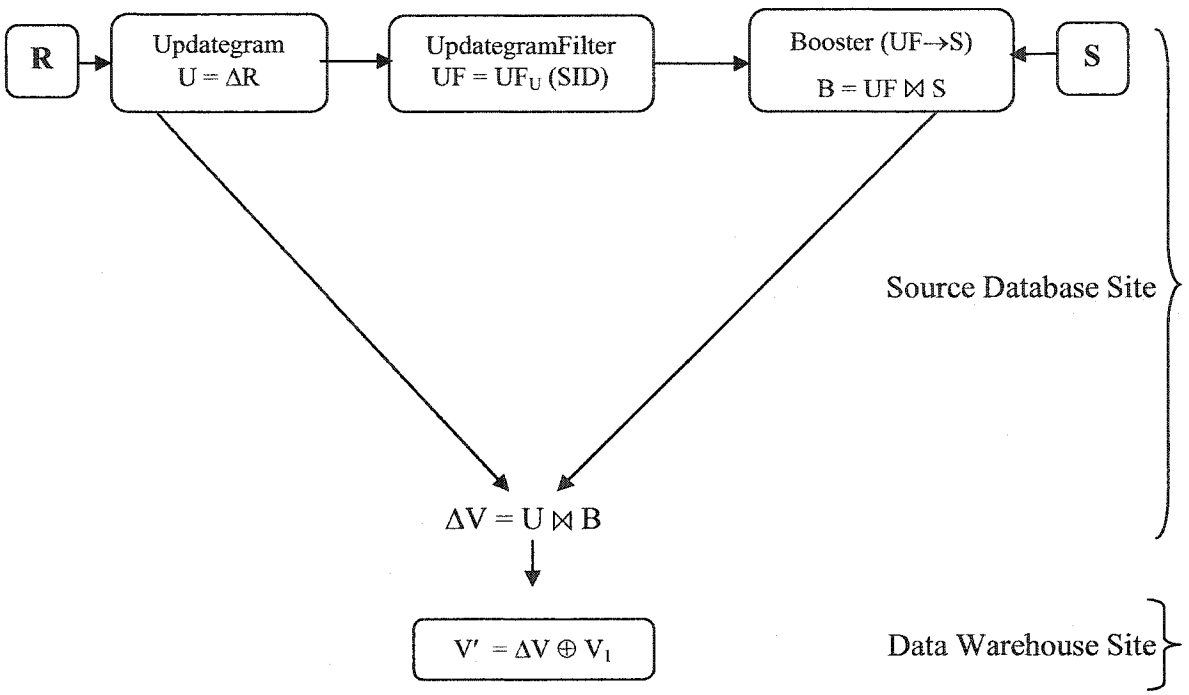


Figure 3.3: Problem definition of the data warehouse stream view update algorithm

Algorithm 3.1 (Stream data update warehouse view algorithm using Hash filter and hash table: Maintaining hash filter and hash table for stream update propagation)

Algorithm Stream-Materialized-View()

Input: set of update tuple, ΔR from relation R {MMID, SID, Volume}
Set of tuple from relation S, set of change updategramPool,
set of change S // Updategramfilter function rebuilt S joining with ΔR
set of change deltaviewPool
Output: a set of updated View- V′

Begin
// Propagate functions: create the delta view
Generate-Updategram = **Updategram(ΔRtuple, type, updategramPool)**
Updategramfilter = **Updategramfilter(updategramPool, S)**
Generate-Booster = **Booster(updategramPool, S, deltaviewPool)**
// Refresh function: refresh the delta view to the original view
Up-to-Date-View = **Refresh-View(deltaviewpool, View- V′)**

End

Figure 3.4: Algorithm for stream data update warehouse view

3.3.1 Propagate functions

The three sub functions of the propagate functions are updategram, updategramfilter and booster. The updategram function is to maintain two hash pools, the insert-pool and the delete-pool for removing the overlap of the updategram. The updategramfilter function uses hash filter on the updategram to get distinct data from the updategram. The booster function maintains delta-view by using two hash pools of the insert-pool and the delete-pool. The updategram function is described in section 3.3.2.1, the updategramfilter function is presented in section 3.3.2.2 and the booster function is explained in section 3.3.2.3.

3.3.2.1 Updategram function

We use two hash pools, the updategram insert pool and the updategram delete pool. Each pool is maintained as a hash table on the joining columns. When an insertion updategram item comes, it is put into the updategram insert pool. When a deletion item comes, before putting it into the delete pool, we first check whether the same tuple appears in the insert pool. If so, the tuple is removed from the pool and the deletion item is discarded. This way it is ensured that overlaps of the updategram are removed as early as possible. The complete algorithm in Figure 3.7 uses three other functions, the hashInsert, the hashDelete, and the hashLookup, to insert an item, delete an item, or judge whether a tuple is in the hash table. These functions are used in the i-pool and the d-pool of the updategram and the delta view hash table.

Figure 3.5 shows how the updategram algorithm puts the new records of table 3.2 into the updategram hash table. If the update tuple information type is SELL, then it is sent into the delete pool (d-pool). If the update tuple information type is BUY then it is sent into the insert pool (i-pool).

Figure 3.5 shows the new updated base table R at time 200307300950, tuples (38387, AC, 600, BUY) which are put into the i-pool using the hashInsert function. The second tuple (41258, BG, 100, BUY) is also sent into the i-pool. Before we put the third tuple (43458, AC, 600, SELL) into the d-pool algorithm, we first call the hashlookup function to check

whether this tuple is in the insert pool. If so, then the tuple is removed from the pool, otherwise it is sent into the d-pool. In this case this tuple does not have a match in the i-pool. Therefore the algorithm will call the hashInsert function to insert the tuple into the i-pool. The same is done for the fourth tuple (45457, BG, 100, SELL).

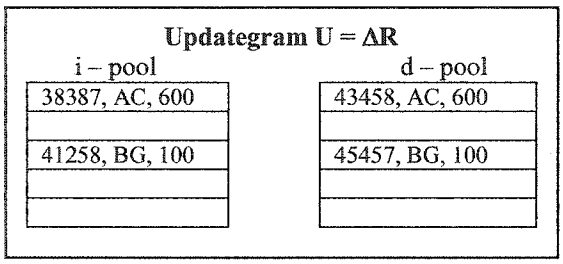


Figure 3.5:Updategram data structure after updating stream data at time 200307300950

Figure 3.6 shows the new updated base table R at time 200307300953, tuples (38387, AC, 600, SELL). We first call the hashlookup function to check whether this tuple is in the insert pool before we put it into the d-pool. If so, the tuple is removed from the pool; otherwise it will be sent into the d-pool. If this tuple matches in the i-pool then the algorithm will call the hashDelete function to delete the tuple from the i-pool, and also for the tuple <41258, BG, 100, SELL>. The other two new tuples <43458, AC, 600, BUY> and <45457, BG, 100, BUY> are put into the i-pool because these tuples do not have a match in the i-pool. Then the algorithm uses hashInsert function to insert the tuple into the i-pool.

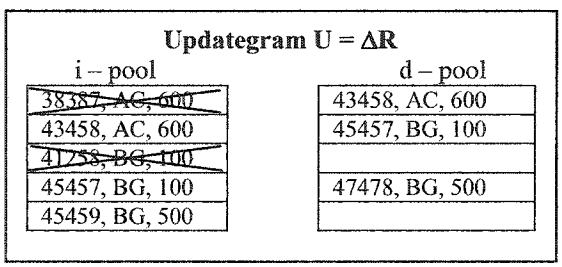


Figure 3.6: Updategram data structure after updating stream data at time 200307300953

Algorithm 3.2 (Updategram - Removing Overlaps on Updategrams)

Algorithm Updategram(ΔRtuple, type, updategramPool)
Input: set of update tuple from relation R and type
Output: updategramPool

Begin

```
//tuple  insert into the i-pool of the upategramPool
        if (type = = insert)
                hashInsert(tuple, insert, updategramPool)
// deleted tuple insert into the d-pool of the updategramPool but it will first
check the i-
        pool for removing the matching tuple from i-pool.
        else if (type = =delete)
        if (hashLookup (tuple,insert, UpdategramPool))
                hashDelete(tuple, insert, updategramPool)

        else hashInsert(tuple, delete, updategramPool)
```

End

Figure 3.7: Updategram Algorithm

```
// hashlooup function is used when deleted tuple are insert into the i-pool
// T is tuple, it could be to delete, insert or search and function answer is returned
by K

HashLookup(T, k)
    i = 0
    repeat j = h(k, i)
          if (T[j] == k and deleted_bit(T[j]) == false)
     return j
     i = i + 1
    until (i == m or T[]j == NIL)
    return -1
//hashinsert function is used to insert new tuple into the i-pool of
updategramPool

HashInsert(T, k)
    i = 0
repeat j = h(k, i)

if (T[j] == NIL or deleted_bit(T[j]) == true)
          T[j] = k
    return
    i = i + 1
    until (i == m)
   error("hash table overflow")
//hashdelete function is used to delete tuple from i-pool when it is matched
and if it is false then to insert the tuple into the d-pool of the
updategramPool.
HashDelete(T, k)
    i = 0
    repeat j = h(k, i)
          if (T[j] == k and deleted_bit(T[j]) == false)
          deleted_bit(T[j]) == true
    return k
    i = i + 1
    until (i == m or T[j] == NIL)
    error("k not found")
```

Figure 3.8: HashInsert, HashDelete and HashLookup functions for Algorithm Updategram and Booster

3.3.2.2 Updategramfilter function

The updategramfilter uses the hash filter (HF) on attribute SID to update the tuple of R denoted as ΔR, then applies it to S to reduce the joining cost of ΔR join S. Before joining ΔR and S to their common attribute SID, probing the tuple of S against $HF_{\Delta R}$(SID) (HF applies on attribute SID of the ΔR) and removing non-matching tuples to

UHash filter UHF = HF_U (SID)

h (SID)	Set
AC	1
BG	1

Figure 3.9: Hash filter built by $HF_{\Delta R}$ (SID) previous ΔR tuple

Booster ($HF_{\Delta R}$ (SID) → S)

B = UHF ⋈ S

SID	STOCK_NAME	PhaseValue
AC	ACME	10.00
BG	BG Group	15.00

Figure 3.10: Rebuilt S with previous ΔR tuple after the applying $HF_{\Delta R}$ (SID) → S

reduce the number of tuples of S to participate in the joining. The joining cost is thus reduced. The updategramfilter algorithm is described in figure 3.11. Figure 3.10 shows the hash filter (HF) built by the relation ΔR to its attributes SID, denoted by $HF_{\Delta R}$ *(SID)*, as an array of bits. Let ΔR (SID) be the set of distinct values of the attributes SID in ΔR, and h be the corresponding hash function employed. Figure 3.9 found two arrays of bits which set to 1 for h (SID) after applying $HF_{\Delta R}$ *(SID)*. The advantage of using the hash filter is that it can store the previous information in array. Whenever any new tuple arrives it will check the existing array table first, rather than doing the whole process again like semi-join. If all the tuples ΔR in updategram match with the existing hash filter array table, then we can use the exiting S rather than rebuild the S again. This way we can reduce the joining cost. Figure 3.12 shows that ΔR in Table 3.2 and Table 3.4 are using the existing S rather than rebuilding it again.

Algorithm 3.3 (UpdategramFilter – use Hash filter (HF) built by Relation ΔR on its attribute SID)

Algorithm UpdategramFilter(updategramPool, S)
Input: set of all update tuples from relation R
Output: S
Initialized: Array $\Delta R(SID[i])$, i = 0
Let J_{att} be the set of all joining attributes in R
if ($J_{att} \neq \varnothing$)
Begin
    Scan ΔR, and $\forall \in J_{att}$ built $HF_{\Delta R}(SID)$ // $HF_{\Delta R}(SID)$ is an array of bits
    if SID[i] $\in$ $\Delta R(SID)$ // $\Delta R(SID)$ the set of distinct values of attributes SID in
                ΔR, and h be the corresponding hash function
    h (SID[i]) = 1
    Scan $HF_{\Delta R}(SID)$ to S where S contains a relations joining with ΔR on SID
    if (ΔR receives all HF for its joining attributes) then
        ΔR applies HF to filter out non-matching tuples and
        builds the hash table for S
End

Figure 3.11: Algorithm for UpdategramFilter

3.3.2.3 Booster function

Booster is first joined with the corresponding updategram items in the delete pool and then joined with the insert pool. The result is put into the *delta view insert pool* and the *delta view delete pool* in the same way as was done before for the updategram. The whole algorithm is shown in figure 3.13. Figure 3.12 shows the delta view joining the updategram and the booster. The booster first joins with the tuple (43458, AC, 600) in d-pool of the updategram. The joining resulting tuple is (43458, AC, ACME, 600, 10) which is inserted into the i-pool of the delta view to remove the same tuple from the i-pool, otherwise it is sent into the d-pool. In this case the tuple (43458, AC, ACME, 600, 10) is removed from the i-pool of the delta view because it matches the tuple in the i-pool and the rest of the tuples are removed in the same way.

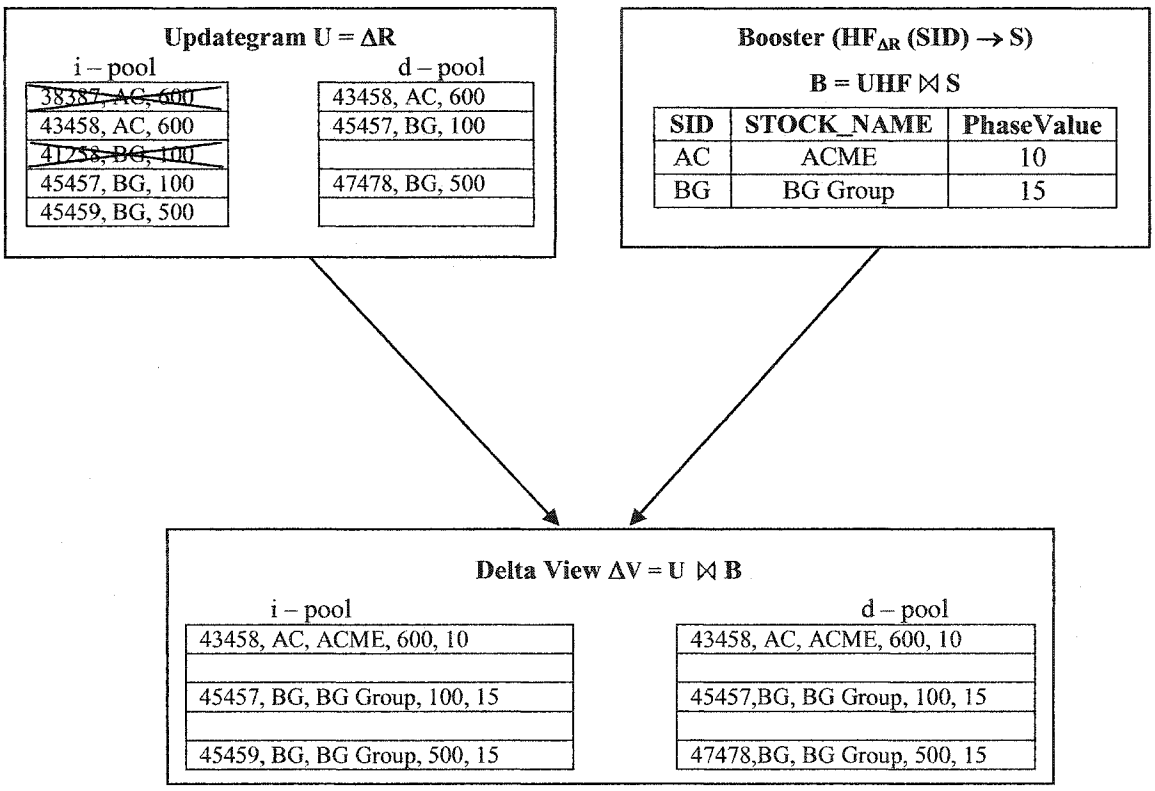


Figure 3.12: Joining Updategrams and Boosters in Delta View

Algorithm 3.4 (Booster Generation – delta view, joining with Updategram and Booster)

Algorithm Booster(updategramPool, S, deltaviewPool,)
Input: S, updategramPool
Output: deltaviewPool
Begin
    if (ugram=hashGet(booster.getJoinKey(), delete, ugramPool))
Begin
        Δview = ugram ⋈ booster
        select_on_Δview
        poolEntry(Δview, delete, deltaViewPool)
        if (is the last item of current booster)
            for (all item u in Updategram Delete Pool)
            hashDelete(ugram, delete, ugramPool)
End
    if (ugram=hashGet(booster.getJoinKey(), insert, ugramPool))
Begin
        Δview = ugram ⋈ booster
        select_on_Δview
        poolEntry(Δview, insert, deltaViewPool)
        if (is the last item of current booster)
            for (all item u in Updategram Insert Pool)
            hashDelete(ugram, insert, ugramPool)
End

Figure 3.13: Booster Generation Algorithm

3.3.3 Refresh function

The original view is updated when the delta view is almost full. When merging the delta view back to the original view then the operation on the view side can be defined as: View ′ = ΔView ⊕ View, shown in table 3.12

ΔView

MMID	SID	StockName	PhaseValue	BalVolume
43458	AC	ACME	10	600
45457	BG	BG Group	15	100
45459	BG	BG Group	15	500
43458	AC	ACME	10	-600
45457	BG	BG Group	15	-100
47478	BG	BG Group	15	-500

Table 3.10: An Instance of ΔView Table

View V

MMID	SID	StockName	PhaseValue	BalVolume
43458	AC	ACME	10	600

Table 3.11: An Instance of original view Table

View ′ = ΔView ⊕ V

MMID	SID	StockName	PhaseValue	BalVolume
43458	AC	ACME	10	600
45459	BG	BG Group	15	500
47478	BG	BG Group	15	-500

Table 3.12: An Instance of merging original View and Delta View Table

Algorithm 3.5 (Refresh – update the view)

Algorithm Refresh-View(deltaview, View- V′)
Input: deltaview
Output: View-V′
Begin
CREATE View View-V′
FROM deltaviewPool
UNION
VIEW-V;
End

Figure 3.14: Merging Original View and Delta View Algorithm

CHAPTER 4
EXPERIMENTAL EVALUATION AND PERFORMANCE ANALYSIS

In this chapter, we compare the performance of DLY02 algorithm with StreamVup algorithm. The StreamVup algorithm is implemented as described in chapter 3. All the experiments are performed on a 1.2 GHz PC machine with 256 megabytes main memory. The operating system is Windows XP professional. All programs are written in Java. The program shows the performance comparison report of both the existing stream data update in [DLY02] with the semi-join and proposed Stream data update with the hash filter.

4.1 Dataset

The parameters shown below are used to generate the datasets.

[T]: Number of transactions

[B]: Number of buy tuples

[S]: Number of sell tuples

[R]: Number of remove tuples

[D]: Number of duplicate share name

[N]: Number of update tuples for the data warehouse materialized views.

For example, T1000.B600.S400.R200.D100.N800 means that $|T| = 1K$, $|B| = 600$, $|S| = 400$, $|R| = 200$, $|D| = 200$ and $|N| = 800$. It represents a group of new update tuples is 1K, the BUY shares transaction is 600, the SELL share transaction is 400, the removed tuples (100 same shares is bought and sold) is 200, the duplicate share name is 100 and the total number of update tuples for the data warehouse materialized views is 800. The datasets with different parameters test different aspects of the algorithms. Basically, if the number of these six parameters becomes larger, the execution time becomes longer.

4.2 Experiment 1: Execution time at different data size with less duplicate tuple

This experiment use fixed sized duplicate tuples and different data size of of the new update tuples to compare the performance of DLY02 and StreamVup. The six datasets are T1K.B600.S400.R200.D200.N800, T2.5K.B600.S400.R200.D500.N800, T5K.B600. S400.R200.D1K.N800,T7K.B600.S400.R200.D1400.N800,T10K.B600.S400.R200. D2K. N800, and T15K.B600.S400.R200.D3K.N800.

Algorithm	Runtime (in milliseconds) at different data size with 20% duplicate tuples					
	1K	2.5K	5K	7K	10K	15K
DLY02	2750	9641	34844	70860	160734	382500
StreamVup	2610	9120	33194	68584	131531	341407

Figure 4.1: Execution Time (milliseconds) with different data size

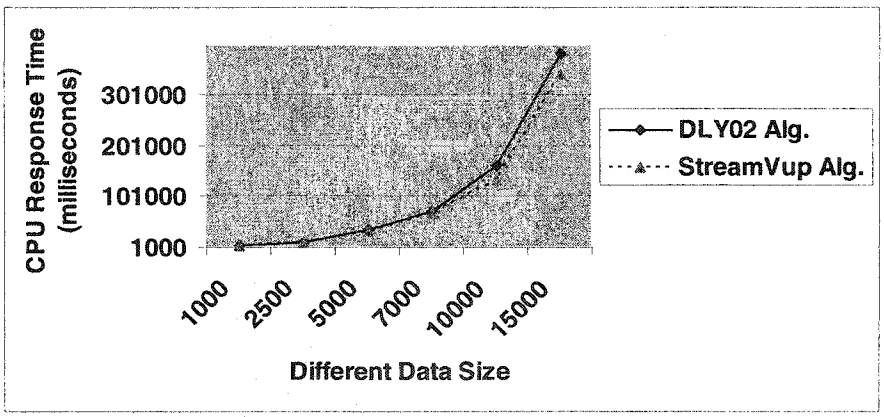


Figure 4.2: Execution Time (milliseconds) with different data size

We test algorithms with the maximum support from 5% to 10% against the data size from 1K to 15K. From figure 4.1 and 4.2, we can find that the execution time of StremVup algorithm less than DLY02 algorithm but the time difference is very small. StreamVup algorithm increase the performance of the execution time when number of update tuples is higher. Next experiment we will check performance with increasing duplicate tuples.

4.3 Experiment 2: Execution time at different data size with increase the % of duplicate tuple in the StreamVup algorithm

In this experiment we use different data size with increasing the number of duplicate tuples in the transaction. The datasets is described as T(1-15K).B(.6-9K).S(.4-6K).R(.2-3K).D(.2-3K).N(.8-12K). In this experiment, we only compare the efficiency with increase both the data size and the duplicate tuples.

% of duplicate tuples	Runtime (in milliseconds) at different data size					
	1K	2.5K	5K	7K	10K	15K
20%	2610	9120	33194	68584	131531	341407
40%	2562	9047	32937	66062	136172	336016
60%	2469	8656	31672	63578	133234	329125
80%	2437	8594	31504	63469	131312	315422
100%	2417	8503	30732	59632	128672	309748

Figure 4.3 Execution Time (milliseconds) with different data size

Figure 4.3, we can find that performance gain increases with increasing duplicate tuples. It may happen that the gain will increase more with increasing data size at higher duplicate levels. In our next experiment we will check the performance gain at different data size with the 60% of duplicate tuples.

4.4 Experiment 3: Execution time at different data size with the 60% of duplicate tuple

In this experiment 3 we check the efficiency of DLY02 and StreamVup algorithms compare the data size with the fixed size (60%) of duplicate tuples in the transaction. The five datasets are T1K.B600.S400.R200.D200.N800, T2.5K.B600.S400.R200.D500.N800, T5K.B600.S400.R200.D1K.N800,T7K.B600.S400.R200.D1400.N800, T10K.B600.S400 R200.D2K.N800, and T15K.B600.S400.R200.D3K.N800.

Algorithm	Runtime (in milliseconds) at different data size with 60% duplicate tuples					
	1K	2.5K	5K	7K	10K	15K
DLY02	2750	9641	34844	70860	160734	382500
StreamVup	2469	8656	31672	63578	128738	329125

Figure 4.4 Execution Time (milliseconds) with different data size

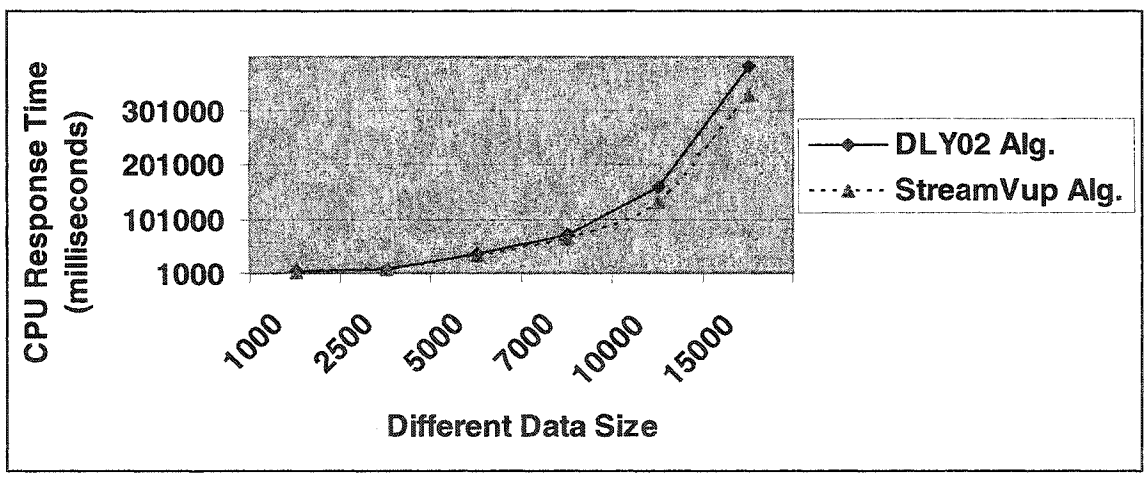


Figure 4.5 Execution Time (milliseconds) with different data size

Based on our testing data size which is shown in figure 4.4 and 4.5, we found that StreamVup algorithm takes less time than DLY02 algorithm when we increased the duplicate tuples.

4.5 Correctness of Algorithm Implementations

Based on our sample test result we can defined the formula for both DLY02 and StreamVup algorithm as a α_t (n) $\propto$ β_t (n), here α_t , β_t represents the CPU response time and n represent the data size. That means, CPU response time increases propositionally for both algorithms which represents α_t for DLY02 and β_t for StreamVup algorithm. We can rewrite the formula as a α_t (n) = k β_t (n), here k used as a constant. StreamVup algorithm performance depends on the value of k. If the value of k >1 then the performance of StreamVup algorithm is better than DLY02 algorithm otherwise it is poor. The following figure 4.6 shows the value of k is greater than 1 at different sample data size that means proposed algorithm CPU response time is better than DLY02 algorithm. Having these results, we can say the experiments conducted in previous 3 sections are based on correctness of algorithm implementations.

n	α_t (n)	β_t (n)	$k = \alpha_t$ (n) / β_t (n)
1K	2750	2469	1.12
2.5K	9641	8656	1.11
5K	34844	31672	1.10
7K	70860	63578	1.11
10K	160734	128738	1.24
15K	382500	329125	1.16

Figure 4.6 Sample Datasets showing the value of k

CHAPTER 5
CONCLUSIONS AND FUTURE RESEARCH

5.1 CONCLUSIONS

This thesis proposes a new algorithm to support the stream materialized view update mechanism which makes full use of available resources to improve performance, and at the same time, have the flexibility of adapting to limited resources to avoid a dramatic performance break. The proposed stream materialized view updated algorithm is based on the hash filter and the hash table to support the huge traffic of stream data efficiently.

Our proposed algorithm uses the hash table for both the updategram and the booster. The hash filter is used only in the updategramfilter to perform better instead of using the base table. Several updategrams may operate on the same tuple. Removing this tuple reduces the computation of joining the updategram and the booster. We are using the hash filter rather than using the semi-join, when the updategram site packs the updated tuples into the updategram and sends it to the booster side and the view side simultaneously. The hash function of the hash filter is storing the result in an array of bit that can later be reused if needed. In semi-join, we can not do that, as it will rebuild the base table at the booster side when the updated tuple arrives.

We are using the hash filter to reduce not only response time but also computation resources and memory requirement. That is why the updategram, the updategramfilter and the booster are small enough to stay in the memory. The join of the updategram and the booster also can stay in the memory.

5.2 FUTURE RESEARCH

If many tables are updated simultaneously, we need to synchronize the update items from all the base tables. A key point here is how to add timestamps that can help the view side tell the exact order of those updategram from all updated sides. We need an exact order for the timestamp, otherwise, we might update the second record before the first record and then we will not get the correct information. It is very hard to maintain a timestamp if all sides are updated at the same time.

REFERENCE

[AFP00]M. Akhtar Ali, Alvaro A. A. Fernandes, Norman W. Paton, ***Incremental maintenance of materialized, Incremental maintenance of materialized OQL views***, In Proceedings of the third ACM international workshop on Data warehousing and OLAP, 2000, P: 41- 48

[AH00] R. Avnur and J. Hellerstein. ***Eddies: Continuously Adaptive Query Processing.*** In Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data, Dallas, TX, 2000, Page: 261-272.

[BBD+02]Brian Babcock , Shivnath Babu , Mayur Datar , Rajeev Motwani , Jennifer Widom, ***Models and issues in data stream systems*** , In Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems June 2002, Pages: 1 – 16

[BLT86] Blakeley, J.A., Larson, P.A. and Tompa, F.W. ***Efficiently updating materialized views.*** In ACM SIGMOD Intl. Conf. on Management of Data, 1986

[BS97] A. Berson and Smith, ***Data Warehousing, data Mining & OLAP*** , McGrew Hill,1997.

[BSW01] S. Babu, L. Subramanian, and J. Widom. ***A data stream management system for network traffic management.*** In Proc. of the Workshop on Network-Related Data Management, Santa Barbara, California, May 2001.

[BW01] Shivath Babu and Jennifer Widom. ***Continuous Queries over Data Streams.*** In Proc. of the SIGMOD Record, Vol. 30, No. 3, September 2001, Page:109-120

[CGL96] L.S. Colby, T. Griffin, and L. Libkin, ***Algorithms for Deferred View Maintenance***, In ACM SIGMOD, 1996.

[CKL+97] L.S. Colby, A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross, ***Supporting Multiple View Maintenance Policies***, ACM SIGMOD Conference, May 1997, pp. 405 - 416.

[CW91] S. Ceri and J. Widom. ***Deriving production rules for incremental view maintenance.*** In Proceedings of the Seventeenth International Conference on Very Large Data Bases, 1991, page 577-589

[CCC+02] Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., And Zdonik, S. ***Monitoring Streams - A New Class of Data Management Applications.*** In VLDB 2002.

[CF02] S. Chandrasekaran and M. J. Franklin, ***Streaming Queries over Streaming Data***, In Processing of the 28th VLDB Conference, Hong Kong, China, 2002

[CFZ01] Cherniack M., Franklin, M., And Zdonik, S. ***Expressing User Profiles for Data Recharging***. In IEEE Personal Communications, August, 2001, pp. 6-13.

[DGG+02]Alin Dobra , Minos Garofalakis , Johannes Gehrke and Rajeev Rastogi, ***Processing complex aggregate queries over data streams*** In Proc. of ACM SIGMOD Intl. Conf. of Management of Data, 2002, Pages: 61 – 72

[DLY02] Xin Dong, Lin Liaon and Zizhen Yao ***Streamed update propagation in a peer data management system***, Department of Computer Science and Engineering, University of Washington.

[DGI+02] Datar, M., Gionis, A., Indyk, P. And Motwani, R. ***Maintaining Stream Statistics over Sliding Windows.*** In ACM-SIAM SODA 2002.

[EB98] C. I. Ezeife and S.R. Baksh, ***A Partition-Selection Schema for Warehouse Aggregate Views***, http:// www.cs.uwindsor.ca/users/cezeife.

[EX00] Ezeife, C.I. and Xu, Mei, ***Maintaning Horizontally Partitioned Warehouse Views***, In proceedings of the second international conference on Data Warehousing and Knowledge Discovery, DaWak'00, Greenwich, U.K, DEXA conference, published in the Lecture Notes in Computer Science (LNCS) by Springer Verlag, Sept. 4-6, 2000.

[Ez97a] C. I., Ezeife, ***A uniform Approach for Selecting Views and Indexes in a Data Warehouse***, In Proceedings of the 1997 International Database Engineering and Applications Symposium, 1997.

[Ez97b] C.I. Ezeife, ***Accommodating Dimension Hierarchies in a Data Warehouse View/Index Selection Schema***, In Proc. of the 6th Int. Conf. on Information Systems Development – Methods and Tools, Theory and Practice, 1997, Page195- 211.

[Ez99a] C. I., Ezeife, 60-539: ***Emerging Non-traditional Database Systems (Data warehousing and mining)***, Lecture notes, School of CS, Uni. Of Windsor, Canada, 1999.

[Ez99b] C.I. Ezeife, ***Optimizing – Selecting Schema for Warehouse Aggregate Views***, Transaction in International Systems: Systems Analysis and Development, Theory and Practices, University of Economics in Wroclaw, Poland, Vol. 1999. No 1, 1999.

[Ez01] C.I. Ezeife, ***Selecting and Materialized Horizontally Partitioned Warehouse Views***, In Data & Knowledge Engineering 36, 2001. Page 185 – 210.

[EV01] Mauricio Minuto Espil, Alejandro A. Vaisman, ***Efficient intensional redefinintion of aggregation hierarchies in multidimensional databases***, In Proceedings of the fourth ACM international workshop on Data warehousing and OLAP, 2001 Pages: 1 – 8

[EWD+02] Maged EL-Sayed, Ling Wang, Luping Ding, Elke A. Rundensteiner, ***An algebraic approach for incremental maintenance of materialized XQuery views***,In Proceedings of the fourth international workshop on Web information and data management, 02 P: 88-91

[FG01] S. Flesca, S. Greco; ***Rewriting queries using views*** , In Proce. of Knowledge and Data Engineering, IEEE Transactions on , Volume: 13 Issue: 6 , Nov/Dec 01 P.980–995

[GBL+96] J. Gray, A. Bosworth, A. Layman and H. Pirahesh. ***Data Cube: A Relational Aggregation operator Generalizing Group-By, Cross-Tab and Sub-Totals***, In Proc of the 12th Conf. on Data Engineering, P 152 – 159, 1996

[GJ01]César Galindo-Legaria, Milind Joshi, ***Orthogonal optimization of subqueries and aggregation***, In Proceedings of ACM SIGMOD international conference on Management of data, 2001, Pages: 571 – 581

[GL95] T. Griffin and L. Libkin, ***Incremental Maintenance of Views with Duplicates***, In SIGMOD, 1995.

[GM95] Gupta, A., and Mumick, I. S. ***Maintenance of materialized views: Problems, techniques and applications***. IEEE Data Engineering Bulletin 18, 2 June1995.

[GM97] H. Gupta and I. Mumick. ***Selection of Views to Materialize Under a Maintenance-Time Constraint***, Technical Note, 1997.

[GMS93] A. Gupta, I. Mumick, and V. Subrahmanian. ***Maintaining views incrementally***.In proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, page 157 - 166, Washington, D.C., May 1993.

[GS97] H. Gupta and D. Srivastava, ***Selecting and Maintaining Materialized Views for Message Management***, Technical Note 1997.

[GT00]Stéphane Grumbach, Leonardo Tininini, ***On the content of materialized aggregate views,*** In Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, 2000, Pages: 47 – 57

[Gu97] H.Gupta, ***Selection of views to materialize in a Data Warehouse***, In Proc. Intl. Conf. on database theroy, Jan 1997. Http://hgupta@cs.stanford.edu.

[GKS01] Johannes Gehrke, Flip Korn, Divesh Srivastava, ***On computing correlated aggregates over continual data streams***, In Proceedings of ACM SIGMOD international conference on Management of data, 2001, Pages: 13 -- 24

[GKS01] J. Gehrke, F. Korn, and D. Srivastava. ***On Computing Correlated Aggregates over Continual Data Streams.*** In Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data, Santa Barbara, CA, 2001, pp. 13-24.

[HCY94] Hui-I Hsiao, Ming-Syan Chen and Philip S. Yu, ***On parallel execution of multiple pipelined hash joins***, In ACM SIGMOD Conference, 1994 p.p (185-196)

[HPD01]Jiawei Han, Jian Pei, Guozhu Dong, Ke Wang, ***Efficient computation of Iceberg cubes with complex measures***, In Proceedings of ACM SIGMOD international conference on Management of data, 2001, Pages: 1 – 12

[HRU96] Venky Harinarayan, Anand Rajaraman, and Jeffrey Ullman. ***Implementing Data Cubes Efficiently***. In ACM SIGMOD International Conference on Management of Data,June,96

[Hu96] N. Huyn, ***Efficient Self-maintenance of Materialized views*** Technical Note, 1996, huyn@cs.stanford.edu.

[In96] W.H. Inmon. ***Building the Data Warehouse***. John Wiley Symand Sons, Inc. 2nd edition,96

[KLM+97] A. Kawaguchi, D. Lieuwen, I. S. Mumick and K. A. Ross, ***Implementing Incremental View Maintenance in Nested Data Models***, International Workshops on Database Programming Languages, August 1997.

[MK00] Mukesh Mohania and Y. ambayashi, ***Making aggregate views self-maintainable***, Journal of Data and Knowledge Engineering, Volume 32, Number 1, p87-109,2000

[MKK97] M. Mohania, S. Konomi and Y. Kambayashi, ***Incremental Maintenance of Materialized Views***, In Proceedings of the 8th International Conference on Database and Expert systems Applications, p 551-560, 1997.

[MQM97] I. S. Mumick, Dallan Quass and B. S. Mumick, ***Maintenance of Data Cubes and Summary Tables in a Warehouse***, In Proc. ACM SIGMOD Conf. 1997

[MRS+01] Hoshi Mistry, Prasan Roy, S. Sudarshan, Krithi Ramamritham, ***Materialized View Selection and Maintenance using Multi-Query Optimization***, In Procs. of the ACM SIGMOD Conf. on Management of Data, May 2001. P.13 -20

[MS01] G. Moro, C. Sartori; ***Incremental maintenance of multi-source views*** In Proc.of Database Conference 12th Australasian , 2001 P.13 -20

[OV91] M.T. Ozsu and P. Valduriez. ***Principles of Distribuited Database Systems.*** Prentice Hall, 1991.

[PKL01]Chang-Sup Park; Myoung Ho Kim; Yoon-Joon Lee, ***Rewriting OLAP queries using materialized views and dimension hierarchies in data warehouses***, In the Proc. of Data Engineering, 17th International Conference, 2001 P.515 -523

[QGM+96] D. Quass, A. Gupta, I. S. Mumick, and J. Widom, ***Making Views Self-Maintenance for Data Warehousing***, In Proc. VLDB'96 p.63074

[Qu96] D. Quass, ***Maintenance Expression for views with Aggregration***, http://www-db.standford.edu/pub/papers/views-aggr.ps

[QW91] X. Qian and G. Wiederhold. ***Incremental recomputation of active relational expressions***. IEEE Transactions on Knowledge and Data Engineering, 3(3):337-341, September 1991.

[RSB00] Roy, P., S. Sudarshan, S. and Bhobhe, S. ***Efficient and extensible algorithms for multi-query optimization***. In ACM SIGMOD Intl. Conf. on Management of Data, 2000.

[SBL01] L. Schlesinger, A. Bauer, W. Lehner, G. Ediberidze, M. Gutzmann, ***Efficiently synchronizing multidimensional schema data,*** In Proceedings of the fourth ACM international workshop on Data warehousing and OLAP, 2001, Pages: 69 – 76

[SP89a] A. Segev and J. Park. ***Updating distributed materialized views***. IEEE Transactions on Knowledge and Data Engineering, 1(2):173- 184, June 1989.

[SU97] S. Chaudhuri, U. Dayal, ***An Overview of Data warehousing and OLAP Technology***, In ACM SIGMOD record 26(1), March 1997.

[The01] Dimitri Theodoratos, ***Detecting redundant materialized views in data warehouse evolution***, In Elsevier Science Ltd. Oxford, UK, UK, 2001, Pages: 363 – 381

[ZGH+95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and Widom, ***View Maintenance in a Warehousing Envireonment***, In SIGMOD, May 1995, page: 316-327.

[ZY99]C. Zhang, J. Yang, ***Materialized view evolution support in data warehouse environment.*** In Proceedings of the Sixth International Conference on Database Systems for Advanced Applications, April19-21, Hsinchu, Taiwan. IEEE Computer Society, 1999, P247-254

VITA AUCTORIS

Mohammed Shariful Islam was born in 1975 in Dhaka, Bangladesh. He completed his Bachelor of Science honors in Computer Science at the Independent University, Dhaka, Bangladesh in 1997. He also worked as an analyst programmer at the CIPROCO Computers Ltd, Dhaka, Bangladesh from August 1997 to August 1999. He has been working as a database and web developer at the Amherstburg Chamber of Commerce, Amherstburg, Ontario from July 2003. He is currently a candidate for the Master's degree in Computer Science at the University of Windsor and hopes to graduate in Fall 2003.