2012

Repetitive querying of large random heterogeneous datasets in RDBMS using materialized views

Ammar Albalkhi
University of Windsor

REPETITIVE QUERYING OF LARGE RANDOM HETEROGENEOUS DATASETS IN RDBMS USING MATERIALIZED VIEWS

By

Ammar Albalkhi

A Thesis
Submitted to the Faculty of Graduate Studies
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

2012

REPETITIVE QUERYING OF LARGE RANDOM HETEROGENEOUS DATASETS IN RDBMS USING MATERIALIZED VIEWS

by

Ammar Albalkhi

APPROVED BY:

____

Dr. A. Azab

Department of Engineering

____

Dr. J. Morrissey

School of Computer Science

____

Dr. R. Kent, Advisor

School of Computer Science

____

Dr. Y.Tsin, Chair of Defense

School of Computer Science

DECLARATION OF ORIGINALITY

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances in my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

ABSTRACT

A methodology has been developed to increase time efficiency of querying large heterogeneous datasets repetitively by applying materialized views on repetitive complex queries. Additionally, a simple user interface is provided to demonstrate the utility of this research methodology. The programs demonstrate sufficiently that the core design can be used to deploy a complete system which could be used in different domains. The methodology as developed in this research is presented as an experimental proof-of-concept prototype based on an abstract design.

DEDICATION

To my Wife

To my kids

To my teachers

To all who believe that success is the loveliest word ever

It is to you that this

thesis is dedicated

ACKNOWLEDGEMENTS

Souzan, I thank you for believing in me and in my ability to achieve my goals.

Without your encouragement, I most certainly would never have the chance to earn every degree I have been awarded since I started my educational path.

Dr.Kent, I thank you for giving me the chance to achieve my dream, supporting and criticizing this thesis.

Dr.Morrissey, and Dr.Azab, I thank you for your comments, questions and support.

I would also like to thank Dr.Frost for providing me with all the techniques needed to write this thesis.

I would like to thank Dr.Tsin who was my first prof when I was enrolled in the Master's program. Thank you for believing in me.

I thank Prof.Finlay who was the proof-reader of this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Context and purpose of thesis

The focus of this thesis is to provide an answer to the following question: How can one design and implement a relational database system approach, using materialized views, that allows for comparison of the time and space dependencies in implementing and maintaining complex queries applied to heterogeneous large datasets? A consequent aspect of answering this question affirmatively, which we do in this thesis, is to demonstrate, through experiments, that utilizing materialized views to support efficient queries in heterogeneous data sets is well justified.

In general, data interoperability refers to the ability of different information-technology systems and software applications to communicate, to exchange data effectively, and to use the information that has been exchanged [RYENG11]. In the healthcare context, for instance, patient data and other health records are stored, heterogeneously, in databases which reflect differences in, among other things:

- Naming of columns
- Data types (eg. dates, times, postal codes, numeric data types)

Current research, as reviewed and referenced in Chapter II, is considered in order to validate the purpose and motivation for this thesis and to position the work, relative to current research achievements. The research can be broadly categorized into two different groups: record selection and dimensional selection.

The main objective of this thesis is to develop an algorithm which improves time efficiency when retrieving search results from repetitive complex queries that are issued on heterogeneous random datasets [MISTR00], and which are stored in heterogeneous databases. The second objective of this research is to obtain a methodology which represents the format of heterogeneous data, and to reform this format in a relational database by applying a renaming method. In general, renaming depends on an alias clause when deploying MV's.

The storing and retrieving of patient records in different health organizations in different formats provided the primary and immediate motivation behind this research since each health organization uses a different type of presentation for data. Notwithstanding this important application domain, that similar problems exist in other domains as well and this underscores the need to find effective solutions.

A good example of a differing format is the address of the patient. Some health organizations use the zip code as part of the address column itself, whereas others use a separate column for it, calling it "area code".

Electronic Patient Records (EPR) [GLASR05] have the potential to bring huge benefits to patients, practitioners, administrators and researchers. Storing and sharing health information electronically can speed up clinical communication, reduce the number of errors, and assist doctors in diagnosis and treatment. Administrators and researchers can access aggregate data for population based decision making and study. Patients can have more control of their own healthcare. Electronic data also has a vast potential to improve the quality of healthcare audit and research. However, increasing access to data through EPR also brings new risks to the privacy and security of health records.

The problem of aggregating data started when analysts discovered that patient records are stored in heterogeneous databases, established and maintained by multiple data provider agencies, and they do not follow any standard type of database which makes sense in real life. Now, after storing the records of patients, if anyone wants to share these records or collect them into one database it represents a tremendous set of problems.

The suggested use of XML is to join these records and then use the XML Schema to control the format of these records. Joining all the records using XML, produces huge datasets of data stored in very large tables which have a tremendous number of columns and records [HU91].

Many systems will be faced by problems in organizing, storing and retrieving medical records. A major issue is that different record parts may reside in different information systems, nationally or internationally. A challenge, therefore, is how to provide a seamless integration of possibly globally- distributed information. Furthermore, modern information systems as envisioned must be generic and extendable in the sense that the kinds of information they must be able to manage is not fully known at design-time. The systems will evolve over time, and they must be designed for this integration process [LI10b].

1.2 Justification of thesis

A continuing problem in many organizations is the challenge of data mining within heterogeneous datasets. One such scenario involves health care information gathered by multiple autonomous agencies, where a need exists to share certain kinds of information, but policies restrict or prohibit the consolidation of data within a single standard repository and schema.

Techniques for handling complex, often repeated queries to support data mining, using methodologies within standard query languages and database platforms, do not extend easily to dealing with issues of heterogeneity and interoperability. In particular, minimization of costs in achieving query results is vital.

In this research, I investigate distinct strategies for database materialization of views from the underlying heterogeneous datasets. These include arbitrary datasets which have few, or no, constraints and which lack standardization.

In practice, there are various applications that store and retrieve data about patients. However, sharing patient data among regions, countries and organizations is a problem since this information is considered to be confidential, which leads to finding efforts to solve the independent-platform problem based on different types of legacy databases using semantic schema to represent data. Also problematic is exchanging the format of the data sets in the domains with semi-structured data, such as is found in the health care domain and one of its important branches, Electronic Patient Records (EPR).

In order to make decisions quickly and correctly in health care environments, the decision support applications need to run very large and complex queries over EPR which may take an unacceptably long time to complete. Techniques such as query optimization and indexing can reduce query response times to some extent, but to achieve acceptable query response times, techniques are needed to pre-compute the frequently asked queries (views) and store (materialize) their results, also known as materialized views, in an EPR database. This poses a tradeoff between time and space and imposes the materialization of the most beneficial views.

Electronic patient records are very diverse, and it is difficult to standardize them to a desirable level of detail (even among hospitals and clinics within the same city). They are themselves quite complex information entities and divide patients accurately into groups with similar health patterns. In evidence-based medicine, the information extracted from the medical literature and the corresponding medical decisions provide key information to leverage the decision made by the professional [HU91]; therefore, the system acts as an assistant for the healthcare professionals and provides recommendations to the healthcare professionals. It is difficult to plan for effective information systems management and also to predict medical diagnosis, treatment costs, and length of stay in a hospital [HU91]. Inherent in this complex system is another serious problem, which is the cost of a centralized database requiring a certain protocol to be followed.

1.2.1 Traditional Databases

In traditional databases there is an important property, which is known as *join (*see Figure1*)*. In fact, this property (join) connects important objects in any database, tables and views.

There are different kinds of join according to Postgres documentation:

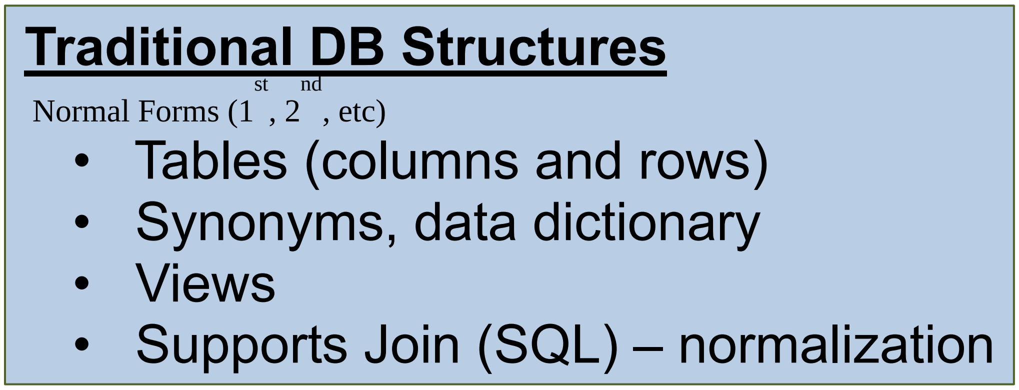
- One to one
- One to many
- Many to many

Figure 2 shows different kinds of join in traditional DB.

According to the Postgres documentation, Meta data, or data dictionary, is a document used to describe data and store information about data. Any traditional database should have a data dictionary to control the behavior of the data stored in it.

Another factor in database efficiency is the presence of constraints. Primary key, foreign key, unique, not null and check are important data constructs in database management. These constraints produce time efficiency and keep the referential integrity of any record in a database.

It was mentioned in the Postgres documentation that normalization is a factor in traditional databases, as a process to govern the integrity of records and data in each table.



Traditional DB Structures

Normal Forms (1st, 2nd, etc)

- Tables (columns and rows)
- Synonyms, data dictionary
- Views
- Supports Join (SQL) – normalization

Figure 1 Traditional DB properties

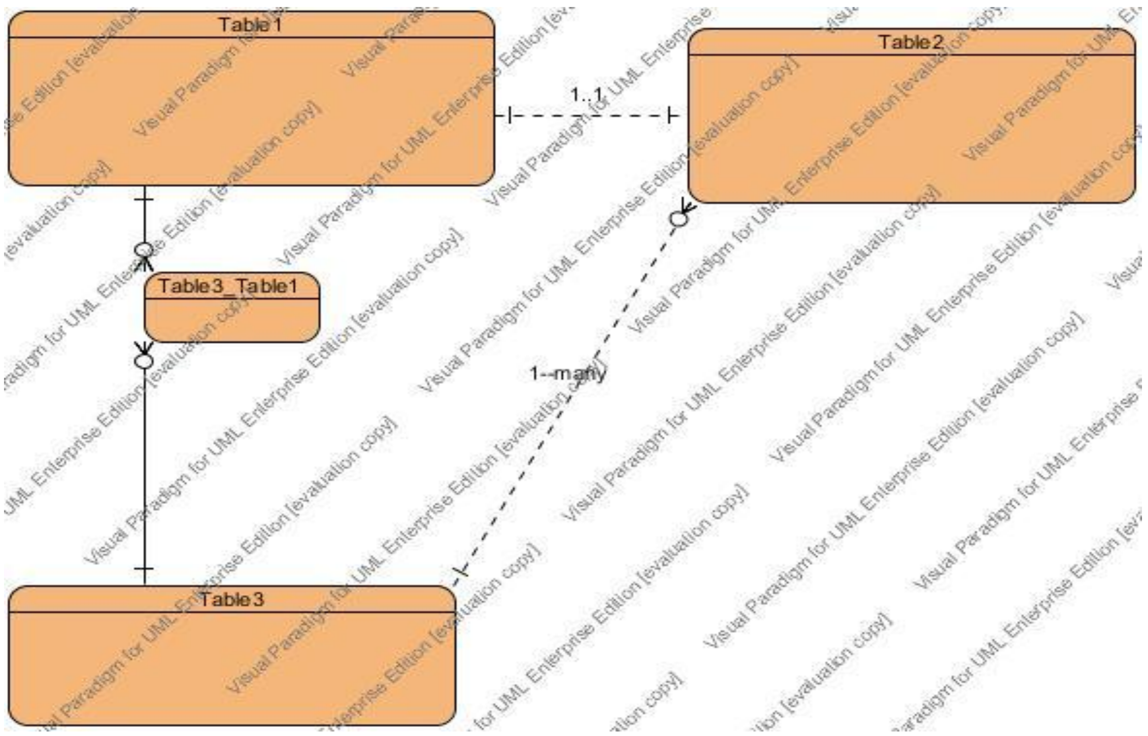


Figure 2 Traditional DB

1.2.1 Non-traditional Databases

In heterogeneous databases there is an absence of *join* between objects (see Figure 3). Objects are connected using the *union* and *union all* clauses. Such databases lack standardization and lack metadata or data dictionary.

Figure 4 presents non-traditional DB properties.

A good example of heterogeneous databases is an EPR database where there are no, or few, constraints. Actually, these kinds of databases apply de-normalization, instead of normalization, which leads to the process of homogenization.

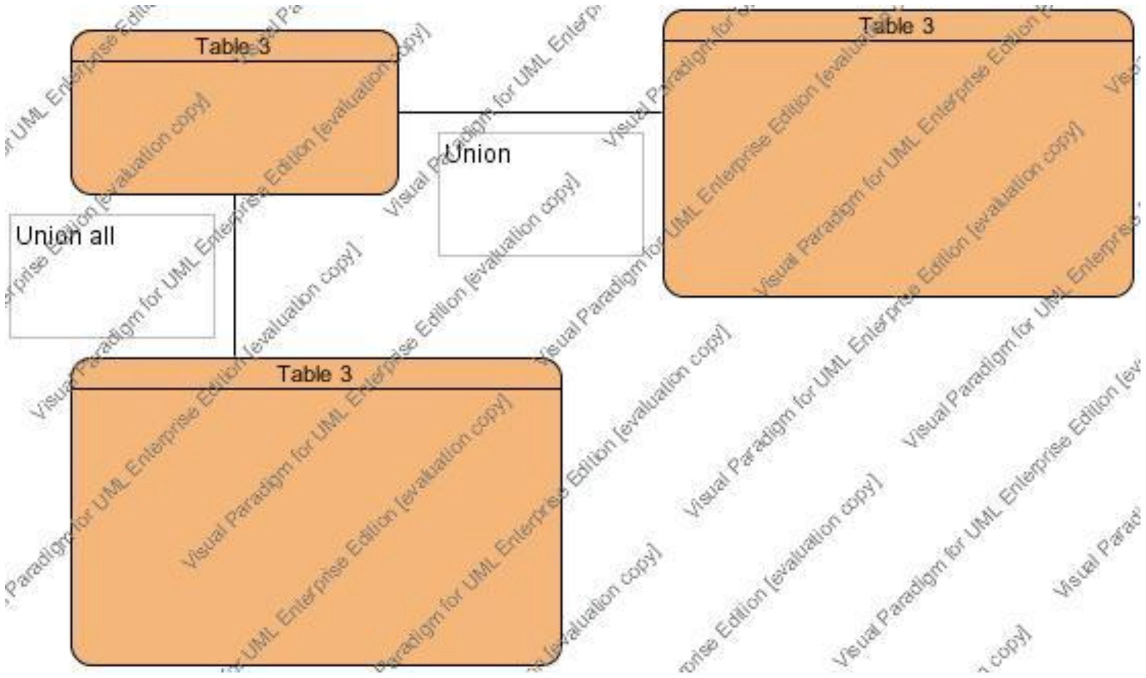


Figure 3 Non-Traditional DB

Non-Traditional DB Structures
Normal Forms (1st, 2nd, etc)
Tables (columns and rows)
No data dictionary
Views
Supports Union (SQL)
– de-normalization

Figure 4 Non-Traditional DB properties

The process for converting data from non-traditional databases to traditional ones is called homogenization. In other words, homogenization is the methodology to make uniform in consistency, especially to render data uniform in consistency by emulsifying the non-standard contents [LI10b].

1.3 Electronic Patient Records (EPR)

In general, the patient record system is a non-traditional database; therefore, a database with simple constraints is required to govern the rules and control this kind of database. The goal is to homogenize a non-traditional database and reshape the contents of it to be stored in simple and, preferably, open-source database tools, such as Postgres as one example. Later on, the standards in the new database [GONG08] can be applied.

The remainder of this thesis is organized as follows. In Chapter II a review of the relevant research literature on approaches using materialized views, is presented. Chapter III presents a detailed discussion of our approach to materialized views and technology and the hypothesis and research methodology used to conduct this research. Chapter IV discusses implementation and the results obtained from our applicative work. Finally, in Chapter V, the thesis is included and is given some additional comments regarding the research and possibilities for further work.

Chapter II

REVIEW OF LITERATURE

Many research efforts dealing with Electronic Patient Records (EPR) problems have been proposed. However, until recently, EPR systems have only used materialized views in minimal and restricted ways to address many real-world combinational applications.

The main reason for using materialized views to solve EPR problems is that materialized views can accomplish tasks, particularly supporting queries, through cooperation while allowing users to retain their privacy.

Current research can be categorized into two different groups: Record selection; and Dimensional selection. These are discussed below.

2.1 Record selection

[GONG08] address the problem of how to eliminate the result of frequent "jitter" phenomenon for materialized view set, which dynamic materialized view selection algorithm generally has. In particular, they focus on how to enhance the overall query response performance of data warehousing which is not very high. The authors refer to implementing data cubes efficiently.

The authors state that they use the DCO-SM (The Dynamic Capacity Optimizer) algorithm and the CBDMVS (clustering-based dynamic materialized view selection) algorithm separately for eliminating the "jitter", and test their dynamic adjustment performance and consumed cost respectively. The CBDMVS algorithm determines the corresponding direct view from the materialized view set. If CBDMVS algorithm finds a direct view, it will immediately return the result.

Both DCO-SM and CBDMVS are dynamic materialized view selection algorithms; a major aspect of algorithm performance is to update materialized view effectively. So, firstly, it needs to compare the query response performance in the data warehouse when the above two algorithms are running. Secondly, it needs to compare the cost which algorithms consume.

The authors state that they use a hardware platform consisting of: CPU P4 3.0G, RAM 1G; the operation system is Windows Server 2003; database platform is SQL Server 2000. Experimental data comes from oilfield exploration and development data. In order to facilitate experiments, the test data warehouse includes four dimension tables and one fact table which is 400MB. The authors claim to eliminate the jitter which dynamic selection algorithm generally has.

Gong and Zhao state the work of a greedy algorithm, BPUS (Benefit per unit space), based on the lattice model. Also, they state and discuss the issue of materialized view selection with the B-tree index. They mention the PBS (Predictive Block Sampling) algorithm which determines the size of materialized view as selection criteria.

Gong and Zhao claim that their algorithm makes a high response performance to one category query and a poor response performance to other types of query. Therefore, the algorithm eliminates the "jitter", which a dynamic selection algorithm generally has, and improves the overall query response performance.

It was mentioned, further, by the authors that when updating certain category materialized view, they just need to calculate the gain of this category instead of the whole materialized view set. Therefore, they have greatly reduced the computational cost for updating materialized view. The authors do not mention any future work.

Segoufin and Vianu [SEGOU05] address rewriting a query in terms of views using a specific language. In particular, Segoufin *et al* address how views determine queries if views provide enough information to uniquely determine the answer to that specific query. The authors refer to complexity of answering queries using materialized views. Also, the authors refer to the foundations of databases.

The authors state that they use conjunctive queries (CQ) and first-order logic to solve the problem. They state that determinacy is decided for special classes of CQs, such as CQs with Boolean or unary answer. Segoufin *et al* state that determinacy says: views provide enough information to uniquely determine the answer to all queries while rewriting queries is equivalent to determinacy and it remains open whether rewriting queries is decidable for regular path views and queries.

They state, further, that concerning FO and CQ, for FO, determinacy is clearly un-decidable. However, FO is complete for FO-to-FO rewritings. For CQ, determinacy is decidable and CQ is complete for CQ-to-CQ rewritings. The authors claim that the problem is only settled for some special fragments of CQs. They show that determinacy is decidable for Boolean or single unary CQ views and arbitrary CQ queries. Furthermore, CQ is complete for rewritings of such queries and views.

The authors discuss determinacy and its connection to rewriting for a variety of view and query languages ranging from FO to CQ in both the unrestricted and finite cases. While the questions were settled for many languages, several interesting problems remain open.

Lijuan *et al* [LIJUA09] address the problem of materialized views as requiring a larger search, greater time consumption and excluding query probability and distribution, and the changes in data sources can't be reflected in data warehouse immediately.

The authors state, in particular, how to improve the CVLC (candidate view lattice construction) and IGA (island genetic algorithm) by using EMVSDIA (Efficient Materialized Views Selection Dynamic Improvement Algorithm).

They refer to implementing data cubes efficiently. Also, they refer to automated selection of materialized views and indexes in Microsoft SQL Server.

Lijuan *et al* [LIJUA09] state that EMVSDIA is a two-step algorithm. In the first part, the input is CV. During the second part, the views with sharply reduced benefits should be replaced by those views which own large query probability. The authors state that the time-cost of EMVDSIA is better than Priority Based Selection PBS . EMVSDIA is fit for running on-line. When dimension is not high, the effect of EMVSDIA is not better than FPUS algorithm, which is based on query frequency in unit space.

Lijuan *et al* [LIJUA09] state that with the candidate view algorithm (CVLV), the multi-dimensional data grid diagram is pruned and those views which have no effects on the view overall cost decrease are excluded. So the algorithm search space is reduced accordingly and the algorithm implementation efficiency and speed are boosted.

They state that EMVSDIA is designed based on CVLV and IGA. The authors claim that EMVSDIA is excellent in both large and medium-sized data warehouse. Also, considering query distribution, query probability and dependence of views, EMVSDIA selects high quality views and increases the responding query ability.

Lijuan *et al* mention that many static selection algorithms have many defects such as time complexity non-running on-line. Although FPUS makes some improvement on some aspects and could run on-line, it did not take into account the dependence relationship among views. EMVSDIA overcomes the above shortcomings. Experimentation demonstrates that EMVSDIA has excellent implementation efficiency and reaches expectant effects.

Yu *et al* [YU03] address the problem of how to select a set of materialized views under certain resource constraints for the purpose of minimizing the total query processing cost. In particular, the search space for possible materialized views may be exponentially large. The authors refer to implementing data cubes efficiently. Also the authors refer to materialized view selection for multidimensional datasets.

Yu *et al* state that there is a new evolutionary algorithm which fits the maintenance-cost view-selection problem well. First, a pool of bit string genomes is generated randomly. The authors discuss that the constraint handling technique and stochastic ranking can deal with constraints effectively. Their algorithm is able to find a near-optimal feasible solution and helps to solve the problem well. The authors discuss the Heuristic Algorithm, the maintenance-cost view selection problem, A multidimensional data warehouse, and the difficulty of maintenance-cost view-selection problem.

The authors claim that with the new constrained evolutionary algorithm, constraints are incorporated into the algorithm through a stochastic ranking procedure. No penalty functions are used. It is mentioned by the authors that the algorithm is based on a novel constraint-handling technique—stochastic ranking. Although stochastic ranking has been used in numerical constrained optimization, its suitability for combinatorial optimization is unclear. This paper demonstrates that a revised stochastic ranking scheme can be applied to constrained combinatorial optimization problems successfully.

Shukla *et al* [SHUKL00] address the problem of how OLAP applications use pre-computation of aggregate data to improve query response time. This problem has been well-studied in recent database literature. In particular, the question is asked concerning how all aggregates are computed from a single cube because many real world applications require aggregates over multiple fact tables. The authors refer to Materialized view selection in a multidimensional database. Also the authors refer to the fundamentals of a database System.

Shukla *et al* [SHUKL00] state that they assume a relational approach to OLAP. This means that each cube of a multi-cube model corresponds to a fact table. However, it is not restricted to the relation model. They propose aggregate selection algorithms based on the new benefit model. They ran four experiments on the database schema by restricting the pre-computation to specific tables and specific derived metrics. They also restrict the measurement of the average query cost to the tables of the database. They state there are actually two drops in the average query cost corresponding to the picking of detailed level aggregates from the two virtual lattices. They assume the average query cost of the set of aggregates picked using the simple cost model can be four times the average query cost of the set of aggregates picked using complex global.

The authors discuss the lattice framework for multidimensional datasets. Then they present the cost model for single cube schemas. They also state an average query cost metric, which makes visualization of the selected and aggregated set for pre-computation easier.

The authors claim that using their algorithm quantifies the improvement in average query cost (average query response time) that can be achieved by the use of the complex cost model.

It was mentioned by the authors that they propose three different algorithms, Simple Local, Simple Global, and Complex Global, which pick aggregates for pre-computation from multi-cube schemas. They state that they show for multi-cube workloads, substantial performance improvements can be realized by using multi-cube algorithms instead of the previously proposed single cube algorithms. In particular, the complex cost model considers join costs, leading to a much better set of aggregates picked for aggregation.

Mistry *et al* [MISTR00] address the problem of how to find an efficient plan for the maintenance of a set of materialized views by exploiting common sub-expressions between different view maintenance expressions. In particular, they discuss how to efficiently select expressions and indices that can be effectively shared by transient materialization; additional expressions and indices for permanent materialization; and the best maintenance plan – incremental or re-computation for each view.

The authors refer to automated selection of materialized views and indexes in SQL databases. Also, the authors refer to efficiently updating materialized views.

The authors state that they extend the volcano query optimization framework, which handle parametric query optimization to generate optimal maintenance plans. They also extend the Query DAG representation. They compute the minimum overall maintenance cost of the given set of permanently materialized views, given a fixed set of additional views to be transiently materialized. They address the problem of determining the respective sets of transient and permanently materialized views that minimize the overall cost.

The authors state that their techniques can generate significant speedup in view maintenance cost, and the increase in cost of optimization is acceptable.

The authors discuss the work of transiently materialized view selection (Multi-Query Optimization). The authors claim that they use a subroutine, the previously mentioned technique for computing time as the best maintenance policy given fixed sets of permanently and temporarily materialized views. The costs of the materialization of transiently materialized views and maintenance of permanently materialized views are taken into account by this step. They propose a greedy heuristic that picks up views in iteratively order of benefit. Benefit is defined as the decrease in the overall materialization cost if this view is transiently or permanently materialized in addition to the views already chosen. Then, depending upon whether transient or permanent materialization of the view produces the greater benefit, the view is categorized as such.

It was mentioned by the authors that they find solutions that exploit commonality between different tasks in view maintenance to minimize the cost of maintenance. Their techniques have been implemented on an existing optimizer, and they have conducted a performance study of their benefits. Future work includes implementing the extensions to handle limited space.

2.2 Dimensional selection

Kabra *et al* [KABRA06] address that there are two significant problems with most implementations of the view replacement model, namely (a) the unnecessary overhead of the access control predicates when they are redundant, and (b) the potential of information leakage through channels such as user-defined functions and operations that cause exceptions and error messages.

In particular, there are several models for fine-grained access control, but the majority of them follow a view replacement strategy. The authors refer to extending relational database systems to automatically enforce privacy policies. Also the authors refer to secured databases.

The authors state that they use the “validity propagation” approach to infer authorization since it can be used for multiple sub-expressions at a low cost. The authors state that authorization is maintained as a group property in the optimizer’s memo structure. They state that authorized views (which replace the relations) in a query plan are all marked as authorized, and any expression generated during optimization is marked as authorized if all its inputs have been marked as authorized. They mention that the validity propagation approach modifies the optimizer to infer authorization of expressions as follows. An expression is represented by a group (or equivalence node) signifying a group of equivalent expressions.

Kabra *et al* [KABRA06] discuss Oracle’s Virtual Private Database (VPD) model, the policy based security management feature of Sybase Adaptive Server Enterprise, and Cell-level access control. The authors claim that redundancy removal can give very significant performance improvements with low optimization overhead and can, in fact, reduce optimization costs greatly. It is feasible to modify an optimizer to generate safe plans. They believe that their approach would reduce the overheads to a reasonable fraction of the optimization costs for the same query (with redundancy removal).

It was mentioned by the authors that their study shows leakage of information through USFs (unsafe function), exceptions and error messages can be efficiently tackled by choosing good, safe plans.

The authors mention that they will extend their prototype to include conditioned authorization, which will reduce optimization time and carry out a more detailed performance study.

Karde and Thakare [KARDE10] address the problem of how query response time plays an important role in timely access to information and it is the basic requirement of successful business application. In particular, Karde and Thakare discuss how to use multiple materialized views to efficiently process a given set of queries.

The authors refer to algorithms for materialized view design in data warehousing environment. The authors state that the space constraint is an important factor while selecting the views to be materialized. The authors discuss the greedy heuristic algorithm, 0-1 integer programming, and genetic algorithm. They claim that the materialized view selection in distributed environment for query processing is also an open issue because it may happen that there are some replicas of materialized view present over the network.

It was mentioned by the authors that materialized views can be subdivided into a number of parts so that any one of the parts can be selected as per the query, which is also an open area for research. Therefore, the node selection, materialized view selection and maintenance of materialized views in a distributed environment and its implication for fast query processing and query optimization can be explored in the future work.

Li *et al* [LI10b] address the problem of how abstract materialized view selection is one of the key techniques for speeding up query answer in a data warehouse environment. In particular, an important question is how to use a novel shuffled frog leaping (SFL) algorithm for materialized view selection.

The authors refer to “Materialized view selection and maintenance using multi-query optimization”. Also the authors refer to “Data warehouse configuration”.

The authors state that the proposed SFL algorithm is an effective method for materialized view selection. The authors state that the proposed SFL algorithm unanimously outperforms the greedy heuristic algorithm (GHA) and the genetic algorithm (GA) in terms of total maintenance costs; they state that the SFL algorithm can find the global optimal solution within a reasonable amount of running time. They mention that the SFL algorithm can avoid being trapped in local optima and improve the quality of solutions by combining the benefits of the mimetic algorithm and the particle swarm optimization algorithm. In addition, the adopted infeasible solution repair algorithm also further improves the quality of solutions.

They discuss the materialized view selection model algorithm, shuffled frog leaping (sfl) algorithm, greedy heuristic algorithm (GHA), and genetic algorithm (GA).

The authors claim that the experimental results show the proposed algorithm provides a good choice for practical data warehouse design and the decision support system.

It was mentioned by the authors that an efficient materialized view selection algorithm is obtained by using the recently introduced shuffled frog leaping (SFL) algorithm. Experimental results on the standard test data sets show that the proposed algorithm is an effective and efficient algorithm for materialized view selection.

Ryeng *et al* [RYENG11] address the problem of semantic caching augments cached data with a semantic description of the data. These semantic descriptions can be used to improve execution time for similar queries by retrieving some data from the cache and issuing a remainder query for the rest. In particular, they discuss how to make an improvement over traditional page caching since caches are no longer limited to only base tables but are extended to contain intermediate results. The authors refer to paving the way for an adaptive database cache. Also the authors refer to adaptive database caching with DB Cache.

Ryeng *et al* [RYENG11] propose a distributed semantic caching method where sites make autonomous caching decisions based on locally available information, thereby reducing the need for centralized control.

The authors state that they generated query workloads consisting of 200 queries from the TPC-H benchmark queries, varying all substitution parameters of the benchmark. The substitution parameters are drawn either from a uniform distribution or from a skewed distribution where 80% of the values are drawn from 20% of the domain.

Ryeng *et al* discuss the execution time and cache hits of repeated executions of our query workload. These measurements are only meaningful on a relative scale, so execution time was measured relative to a baseline execution without caching. During this execution, the caching code was completely disabled. Cache hits were measured relative to the number of queries in the workload. Their results indicate several ways to further improve query processing by semantic caching.

Ryeng *et al* mention semantic caching and predicate-based caching augment cached data with a semantic description of the data. The benefits of semantic caching include low overhead and reduced network traffic cache tables. Semantic caching has also been applied to deductive databases and web querying systems, and all these systems are built for a single query entry point to the system.

The authors claim that they implement the method in the DASCOSA-DB distributed database system prototype and use this implementation to do experiments that show the applicability and efficiency of their approach. They state that execution times for queries with similar sub-queries are significantly reduced and that overhead caused by cache management is marginal.

It was mentioned by Ryeng *et al* that they have developed a new method for semantic caching in a distributed database system with autonomous sites, where caching policies and decisions can vary from site to site and workload statistics are sparse.

They have implemented the semantic cache in the DASCOSA-DB (Database Support for Computational Science Applications). Also, they mention that the cost of re-computing the cached data is also important. The savings made possible by caching the result of a complex query are sometimes higher than the savings from caching the results of less time-consuming queries with a higher hit rate.

Ryeng *et al* [RYENG11] address the problem of semantic caching augments cached data with a semantic description of the data. These semantic descriptions can be used to improve execution time for similar queries by retrieving some data from the cache and issuing a remainder query for the rest.

In particular, they discuss how to improve traditional page caching since caches are no longer limited to only base tables but are extended to contain intermediate results.

The authors refer paving the way for an adaptive database cache. Also, the authors refer to adaptive database caching with DB Cache.

Ryeng *et al* [RYENG11] state that they propose a distributed semantic caching method where sites make autonomous caching decisions based on locally available information, thereby reducing the need for centralized control.

The authors state that they generated query workloads consisting of 200 queries from the TPC-H benchmark queries, varying all substitution parameters of the benchmark. The substitution parameters are drawn either from a uniform distribution or from a skewed distribution where 80% of the values are drawn from 20% of the domain.

They discuss that the execution time and cache hits of repeated executions of their query workload. These measurements are only meaningful on a relative scale, so execution time was measured relative to a baseline execution without caching. During this execution, caching code was completely disabled. Cache hits were measured relative to the number of queries in the workload.

Li *et al* [LI10b] address the problem of how queries that involve aggregate functions, very common in decision support environments, must first compute the aggregate in a sub-query before they can use its value in a comparison.

In particular, they discuss how there is intra-query redundancy between a main query block and a sub-query block, where redundancy means overlap in the “from” clause and possibly in the “where” clause. The authors refer to “an inflationary fixed point in

XQuery “. Also, the authors refer to “moving selections into linear least fix-point queries”.

Li *et al* state that they use new approach to translating practical lass of Xpath queries over (possibly recursive) DTDs to SQL queries with a simple lfp operator found in many commercial RDBMS.

The authors state that in many applications, one would prefer a lightweight tool that provides the capability of answering Xpath queries within the immediate reach of commercial RDBMS instead of using a heavy-duty system. Finally, one cannot use the encoding and indexing approaches to answer xml queries over xml views.

Li *et al* state the work of the middleware-based approach which provides clients with an xml view of the relations representing the XML data. They also mention the first technique to rewrite recursive path queries over recursive DTDs to SQL for schema-based XML storage. The translation consists of two phases. The authors claim that they provide not only the capability of answering important Xpath queries within the immediate reach of most commercial RDBMS, but also the query answering ability for certain xml views.

It was mentioned by Li *et al* that the novelty of the approach consists of a notion of extended Xpath expressions capable of capturing DTD recursion and Xpath recursion in a uniform frame work; and they develop an efficient algorithm for translating an Xpath query over a recursive DTD to an equivalent extended Xpath expression that characterizes all matching paths, without incurring exponential blowup and better still, optimizing the query by filtering unnecessary computation based on the structural properties of the DTD during the translation. They also develop an efficient algorithm for rewriting an extended Xpath expression into an equivalent sequence of SQL queries.

Several extensions are targeted for future work. They recognize that several factors affect the efficiency of the SQL queries produced by their translation algorithms, and they are currently developing a cost model in order to provide better guidance for Xpath query rewriting.

Li *et al* [LI11] address the problem of how the P2P network allows all computers to communicate and share resources as equals and does not depend on a central server for control. In such an environment, tracing how data is copied between peers and how data modifications are performed are not easy because data replications and modifications are performed independently by autonomous peers.

In particular, they discuss how this creates inconsistencies in exchanged information and results in a lack of trustworthiness. The authors refer to “Declarative networking: Language, execution and optimization”. Also, the authors refer to materialized views in general.

Li *et al* state that they consider the fault tolerance issue. Their method can cope with the failure of one peer. If multiple failures occur simultaneously, they may not be able to recover the lineage information for tracing. For example, when two related peers suddenly leave the network at the same time, especially when a peer and its backup peer leave together, it is hard to recover the correct lineage. However, they assume that such events are quite rare in their context.

Li *et al* state that their framework incurs a maintenance cost; it would be more efficient than the centralized approach in which all the histories are maintained in one or more servers. In such a case, the processing cost for query processing and maintenance would become the bottleneck for the whole system.

Li *et al* discuss P2P databases, data provenance, and data space management, declarative networking, and materialized views. The authors claim that in their approach they use a rule for the parameter k, which determines the policy of materialized view maintenance. The parameter k is initially fixed to some value (e.g., k = 2), when P2P record exchange is started. An alternative strategy would be to treat k as a variable, allowing different k values to be selected for different peers. This option is interesting, especially when some peers have large storage and high processing power. However, to simplify the algorithms, they do not consider this option and leave the problem for the future.

It was mentioned by Li *et al* that for efficient query processing, data replication and caching are popular techniques. Taking into account the practical requirements of tracing, they add features to their traceable P2P record exchange framework. Although the storage and maintenance cost will increase, the query processing cost can be reduced and failure of peer to peer communication can be overcome. They consider the trade-off between query processing cost and maintenance cost when evaluating the total cost reduction.

Future research can be summarized as follows. Enhancement of query expression power: the authors will enhance the strategies to handle more complex tracing queries (e.g., tracing queries that involve aggregation requirements). The effectiveness and limitations of the declarative language-based approach will become clearer.

Efficient coupling with DBMSs: For implementing their framework, Li *et al* assume that a local record management system in each peer is implemented using a conventional RDBMS. They would like to use more powerful and robust DBMS functionalities that can come from the tight coupling of the record management system and the underlying RDBMS.

They are developing a prototype system of their P2P record exchange framework, and they have also designed a P2P network simulator that can be used for simulating their prototype system as a virtualP2P network. These developments will provide a positive feedback and help to improve their fundamental framework.

Calvanese *et al* [CALVA11] address the problem of answering a query based only on the pre-computed answers to a set of views. In particular, they discuss how this problem is largely unexplored in the context of description logic ontologies. Different from traditional databases, description logics may express several forms of incomplete information, and this poses challenging problems in characterizing the semantics of views.

Calvanese *et al* refer to "information integration using logical views". Also, the authors refer to complexity of answering queries using materialized views. The authors state another interesting issue is to investigate the impact of both extending and restricting the language used to express the query and the views on the complexity of view-based query answering, with the goal of singling out more cases where the problem is tractable.

Calvanese *et al* state that their work shows, for all DLs, the complexity of view-based query answering under the model-centered semantics is the same as the complexity of computing certain answers without UNA (unique name assumption). The authors state that query rewriting has been studied for the case of conjunctive queries (with or without arithmetic comparisons), disjunctive views, queries with aggregates, recursive queries and non-recursive views, queries with negated goals, and in the presence of integrity constraints of limitations in accessing the views, and rewriting techniques for query optimization.

Calvanese *et al* claim that their research constitutes a first systematic study of the semantics and the complexity of view-based query answering in DLs. The framework they have introduced distinguishes between different semantics for the problem, corresponding to different variants of the notion of solution.

They have related view-based query answering to privacy-aware information access, and we have presented several algorithms and complexity results for various DLs, both in the model-centered and in the TBox-centered semantics.

It was mentioned by Calvanese *et al* that they exploit the relationship between description logics and disjunctive databases in the study of view-based query answering. In particular it is possible to reduce the usual reasoning tasks in Description Logics (e.g., instance checking and query answering) to reasoning in disjunctive data-log programs.

The computational results presented in this paper are compatible with those that would be obtained by encoding view-based query answering in disjunctive data-log. Hence, it would be interesting to explore whether the correspondence between description logics and disjunctive databases can be extended to view-based query answering.

Bahloul [BAHLO09] addresses the problem of not investigating automatically and generating materialized views from access control rules defined over base relations and to control the rules needed to improve materialized views efficiency.

In particular, Bahloul discusses how to automatically ensure confidentiality of materialized views based on basic access control rules, and how to identify formal tools to tackle this problem and resort to an adaptation of query rewriting techniques.

Bahloul refers to view security as the basis for data warehouse security. Also, Bahloul refers to administering permissions for distributed data. Also they mention a scalable algorithm for answering queries using views.

Bahloul considers fine-grained authorization policies that are defined and enforced in the database through authorization views. Authorization views specify the accessible data by projecting out of specific columns in addition to selecting rows. This framework allows fine-grained authorization at the cell level.

Bahloul proposes a more flexible model by inferring the set of authorizations views to control access to the materialized view; she allows users to access even a part of the materialized view.

Bahloul states that she builds on the Bucket algorithm for query rewriting algorithms in data integration and then when applying the original bucket algorithm, the authorization view is considered as irrelevant. The idea behind using the bucket algorithm is that it exploits the predicates in the query to significantly prune the number of candidate conjunctive rewritings that need to be considered.

Bahloul states the problem of how to automatically coordinate the access rights of the warehouse with those of sources. The author proposes a theory that allows automated inference of such permissions for the warehouse by a natural extension of the standard SQL grant/revoke model from systems with redundant and derived data.

Bahloul claims that she discusses ingredients for an automated method to derive access control rules for materialized views by selecting the appropriate access control rules that are attached to underlying base tables. She adapts the bucket query rewriting algorithm.

2.3 Literature summary

After reviewing these research papers it is clear that MV's were used primarily in traditional databases, in particular, warehousing databases. As such, there is considerable scope for studying MV's used with heterogeneous data sets.

Regardless of the varieties of algorithms used in the previous research papers, there were two different techniques. The first one was built upon the algorithm which depends on the fields mentioned in the where clause to be indexed, while the second approach was built on the algorithm which depends on eliminating certain columns or records (horizontal and vertical selection).

CHAPTER III

MATERIALIZED VIEWS DESIGN AND METHODOLOGY

3.1 Materialized views

Materialized views are database objects that store query results. In fact, they are actual tables that store queries results. It was mentioned in all research papers that complex queries take a long time to get back the desired results. In fact, many techniques were used to speed up those kinds of queries where time plays a critical factor in the data retrieval process. One of the most effective techniques is using Indices on columns which are used frequently in the select statement. The problem found with the EPR data, was how to get enough information about the columns of tables which were obtained from many public service institutions, which led us to have a lack of standards [LI11].

By storing pre-queried information it was not needed to run the real query when data is obtained. This is typically called "caching" outside of the database environment. In other words, a view is taken and turned it into a real table that holds real data, rather than employing a gateway to a select query. It was hypothesized that materialized views should speed up the queries regardless of the ability of adding indexes on the base table. In the next chapter this theory will be proved to be true.

Materialized views are schema objects that can be used to summarize, pre-compute, replicate, and distribute data. A materialized view provides the results of a query in a separate schema object. The existence of a materialized view is transparent to SQL and will improve the performance of SQL execution [LI10b].

Materialized views (MV) are different from standard Views in that MVs are actual physical objects that are built from the data in other tables; or, as in our case, they are built from the same tables, but with different query results. When using physical objects, in other words materialized views, they could be indexed, partitioned and clustered to improve performance.

Materialized views, or MVs, also reduce network loads in respect of multi-master replication; also, replicating data with materialized views increases data availability and greatly enhances the performance and reliability of the replicated database. When using MVs, the data based on both column and row level clustering could be replicated; in other words, data can be replicated that is needed thereby cutting down and reducing network traffic.

3.2 Postgres MV's

Materialized views do not require a dedicated network connection. So the refresh process can be applied by scheduling a job. Then refresh materialized views manually on-demand or by using database/tables trigger.

The refresh process is sometimes called night batch job which means that the interface, which is used, is responsible for the update process. The night batch job will assure the database consistency which means updating the database before issuing any repetitive complex query. Sometimes it is used for validating the data which is received at any time.

By using the interface, as the engine for the night batch job, there is no need to write specific database triggers for the update procedure. The interface which could be any high level object oriented programming language is responsible for the incremental update of the all records, which let the night batch job an independent database platform process.

There are four types of MV’s in Postgres:

1. Snapshot
2. Very Lazy
3. lazy
4. Eager

The first type, Snapshot, creates MV’s as stored tables selecting everything from a view or a table. Sometimes tables can be stored by selecting the where clause, the group by clause, the order by clause, and aggregating functions. This type could be refreshed at interval time using certain types of trigger.

Snapshot MV’s are easy to setup and to be created but expensive when full refreshment is applied [LIJUA09]. Snapshot MV’s are the only ones that can be used at the current time according to given input. Later, all kinds could be applied.

The second type is called Very Lazy MV’s, which are, in a way, very much like the first ones, but with one main difference which occurs when synchronized rows are updated. This leads to the need for tracking rows which have been updated.

The third type is the lazy one; it starts out as the snapshot type. The refreshment phase takes place at the end of each transaction.

The last type is the eager MV’s. This type of materialized views is like the lazy MV’s but each statement is updated. Eager MV’s use triggers after DML (insert, update, and delete).

It was mentioned in some research papers [LI11,LIJUA09] that materialized views performance reduces bottleneck O(f(n)) query to O(1).

In general, reducing the system resource requirements by pre-computing and storing results of complex queries allows for the automatic rewriting of complex queries, and they are transparent to the database front-end. They have maintenance requirements. They can apply complex relationships and they can be refreshed on demand or on a schedule.

In this research, different types of materialized views were applied. Our milestone was the snapshot technique according to the nature of the current database. We claim that regardless of the type of materialized view, they can all be applied on any heterogeneous database. They have the following qualities:

- It is useful for summarizing, pre-computing, replicating and distributing data.
- It provides faster access for expensive and complex joins.
- It is transparent to back-end users.
- It has grouping compatibility.
- It has aggregate compatibility.

It was found that there are two methodologies for creating materialized views:

- Current state: using "CREATE TABLE AS"
- Optimal: "CREATE MATERIALIZED VIEW" grammar

Also, to update materialized views, one of the following had to be followed:

- Current state: Periodically create new snapshots, or maintain using triggers
- Optimal: Built-in refresh via multiple strategies

The current state method was chosen so this technique could be applied in any future database, which leads us to use the snapshot type.

3.3 Hypothesis

This research has a main assumption which is, by applying MV's on random heterogeneous datasets to obtain results from repetitive complex queries, time efficiency is increased. The hypothesis of this research has two sub-assumptions. The first one is, if MV's are employed with existing datasets, time efficiency is increased. The second assumption is, if MV's are applied on new inserted datasets, time efficiency is still increased. The first sub-assumption is called "static deployment". The second sub-assumption is called "dynamic deployment", sometimes is referred to as "incremental update".

To achieve acceptable query response times, different techniques were needed that pre-compute the frequently, complex, and asked queries (views) and store (materialize) the results of the database queries in what is called materialized views. This poses a trade off between time and space and imposes materialization of the most beneficial views in different kinds of domains [RYENG11]. Time efficiency can be achieved through applying MV's on the complex queries of a heterogeneous database aided by applying different types of MV's.

This leads to our Statement of Hypothesis, namely:

Statement of Hypothesis:

By applying Materialized Views to heterogeneous data sets, one obtains degrees of homogeneity that support straightforward and effective management of space *resources, while improving significantly the time complexity of achieving repetitive queries.*

In addition, periodic and incremental updates from various data providers can be efficiently managed so as to integrate new data into MVs without substantial negative impact on query time or space complexity.

In the next chapter, this hypothesis will be demonstrated with experiments, results, SQL code samples, and mathematical formulas. A demonstration of this hypothesis can be accomplished through achieving the following steps:

- Developing a prototype database system such is outlined in Figure 18.
- Validating the prototype system against a known system used in healthcare projects.

The objectives of this thesis research include, therefore:

- Automation of querying functionality on non-traditional databases.
- Designing and building an interface that supports the continuous querying of different datasets (see Figure 17).
- Implementing a system interface that completes query processes of all data at all times.
- Developing a schema that provides efficiency improvements.
- Using a semantic layer for mapping the MV's that have been used .

3.4 Methodology

The research methodology involves the following steps:

1. Create the prototype system to test the prototype data

2. Use the Postgres database to obtain heterogeneous data sets and to store them in mega-tables, which contain approximately 1 million records and 255 columns
    - Use SQL and PL/SQL to create and deploy MV's
    - Use scripts to create data in the prototype system
3. Apply MV's on mega-tables
4. Create java interface to deploy the above three steps
    - Use Java abstract classes
    - Use Java beans
5. Map complex queries to the MV's using the Java interface
6. Record the time spent after applying MV's on the large datasets

Snapshot MV's are used according to the type of data available, considering the lack of constraints and the lack of standardization:

- No trigger is used
- Snapshot MV's are the basic form of using MV's
- Maintenance is done manually

Also, various technologies are employed, such as Postgres MV, PL/SQL, Postgres PL, and Java, which is the programming language used to construct the user interface (UI) employed herein. This UI is designed and constructed based on:

- Abstract classes which is used with any database
    - Java class to connect to the database
    - Java class to register the required drivers
    - Java class to implement the GUI

- Java class to show mapping
- Java class to drive the whole process

The above methodology and technologies are explicated in the following section and implemented to achieve results, which will prove the research hypothesis.

CHAPTER IV

IMPLEMENTATION AND RESULTS

4.1 Interface

The nature of the datasets within this research scope is random datasets with no constraints.

When building a new application that has an interface to manipulate the legacy data, there are several aspects to be considered. The key aspects include:

- Obtaining the data.
- Extracting the data from old resources.
- Mapping the data from its current form to a new one in order to use it with the new interface.

Data was obtained from different resources, such as MS Word, Excel, Oracle, and other database types. The problem was in considering the autonomous nature of many public service institutions and how they gather and deal with data in different ways [YU03]. Postgres SQL was selected because a database is needed which can combine with XML, an extensible type system, a powerful procedural language, has transactions uses MVCC to manage concurrency instead of locking, and has common table expressions including recursive, and analytic functions along with several hundred new features. Postgres SQL is very strong on data integrity, stricter at complying with SQL specifications, and the fact that it is an open-source database permitted access to all features and capabilities relevant for this research study.

The Postgres system consists of three top level components:

1. Backend
2. Postmaster
3. Frontend

The backend is the database itself where all SQL statements are handled. The postmaster is the supervisor thread which establishes the connections with the backend. The frontend is the interface which interacts with the user requests.

Postgres is a high performance, full-featured professional DBMS that is free of cost and available under open source licensing. The Postgres system has limitations presented in Figure 5.

Limitation type	Postgres Limitation
Database size	Unlimited
Table size	64 Terabytes
Row size	Limited by table size
Column size	1 Gigabyte
Rows in table	Unlimited
Columns per row	1600

Figure 5 Postgres DB limitation

Based on the results of various health surveys and other data sets obtained in our laboratory, through collaborative and contract research projects, it was obvious that it is needed to employ tables which have a huge number of columns. Mega tables were created to obtain the information from heterogeneous databases with respect to any data loss which could occur when importing data using indexes and constraints [SHUKL00].

These mega tables may contain upwards of 1 million records and 255 columns. It was found that executing any query on these tables would slow down the retrieving process time and a way should be found to improve that process [YU03, RYENG11].

The solution is to use materialized views to increase the efficiency time of any repetitive complex query request.

Disk storage technologies	**Memory based technologies**
Complexity – Seek times limited by electromechanical devices	Complexity: Seek times result of direct access, microprocessors
Performance: millisecond	Performance: micro- to nano-second
Capacity: Terabytes	Capacity: Gigabytes
Cost: ~ $100/100GB	Cost: ~ $100/GB

Figure 6 Disk/ Cache comparison

Figure 6 shows the comparison between using the MV's technology and the select gateway technology.

System performance measurements were obtained to compare the results of query sets against different views. First-time queries, repeated queries, and the effect of view updates on queries were studied. Experimental findings are related to theoretical results of complexity analysis of critical algorithms and system components.

4.2 Algorithms

The following algorithm was applied to existing data and records.

- Algorithm 1:
  - Purpose: Using MV's general case

- Assumptions: Time spent when using MV's is less than the time spent without applying them
- Input:
    - Tables with 255 columns and more than 900,000 records
    - Set of repetitive complex queries
- Output:
    - Set of materialized views

Pseudo-code

This algorithm is used for existing data (static deployment)

```
• Accept a complex query to get the results
• If this query is issued for the first time then
    ▪ Read records from table/s
    ▪ Apply union/all , sorting, grouping and aggregation functions
    ▪ Map an  MV to this query
• Else (repetitive usage)
    Read records from MV
```

<u>Complexity</u>:

In the beginning, we had to discuss the complexity of the queries without employing our algorithm. When we investigated complex queries [CALVA11], we took into consideration the worst case scenario which might have included union/all, aggregation, grouping, and sorting [KARDE10]. We require the following formula:

Formula 1:

$$Ttot=Tr+Tu+Tc+Tg+To$$

Where:

Ttot is the total time needed to retrieve data from a complex query.

Tr is the time needed to read all records from all tables mentioned in the query.

Tu is the time needed to apply union/all on tables.

Tc is the time needed to apply aggregation functions on the records.

Tg is the time needed to apply grouping by operator when aggregation functions are used.

T0 is the time needed to apply order by operator (sorting)

Tr could be represented as Tr= $\sum_{i=1}^{n} tk$,where tk represents the time needed to read all records from one table k.

Tk= $\sum_{i=0}^{n} ti$,where ti represents the time needed to read one record in one table.

Tr= $\sum_{k=1}^{z} \sum_{i=0}^{n} ti$, where z is the total number of tables and n is the total number of record residing in each table.

Let us assume that Tcalc=Tu+Tc+Tg+To.

This means that Tcalc represents the time needed for all kinds of calculations including joining records from different tables, applying aggregation functions, grouping records, and finally sorting the result.

The previous formula would be written as formula2:

$$Ttot= \sum_{k=1}^{z} \sum_{i=0}^{n} ti+Tu+Tc+Tg+To$$

$$Ttot=Tr+Tcalc$$

Without using MV's, time needed is O (N) + O(C) where C is the time needed for any calculation on the N records. The worst case for Algorithm1 is O (N) +O (1) →O (N).

All operations for joining, calculating aggregation functions, grouping, and sorting would be eliminated because the result of any complex query would be saved into an independent object in the database. This algorithm is used to prove the first hypothesis, which is referred to as static deployment.

It is important to mention that there is a price to be paid for first time usage of Algorithm1 since there is time for creating MV's. However, this algorithm is useful for repetitive complex queries. This Algorithm can be called "naming queries". It fires the query and stores it in a meaningful name.

Figure 7 shows a complex query sample:

Example Query #1:

```
SELECT count(*)
      AS count,
sum(bigone.c0)
      AS sum,
avg(bigone.c0)
      AS avg, bigone.c1
, bigone.c0
      FROM bigone
      WHERE
bigone.c0 > 800000
     GROUP BY
bigone.c0, bigone.c1
ORDER BY count
```

Figure 7 Complex query example1

In Figure 7, bigone is a mega table where c0 is the first column, which its data type is a number, and c1 is a text column.

The first algorithm would take any complex query and store the result of it as an MV. The complex query is mapped to a specified MV using a table called "mapit" which has five columns: Id, name, Text, mvid, and time. Following is the description of the "mapit" table:

- Id is the primary key of the table which will play the role of a controlling constraint for this table.
- Name is an indexed field which contains the keywords of the complex query or it could be the description of the complex query.
- Text is the column which has the text of the SQL statement issued by the complex query.
- Mvid is the id of the MV connected to this complex query.
- Time is an indexed column which would record the time of creating this record in the table. This column has a default value which is the now () function.

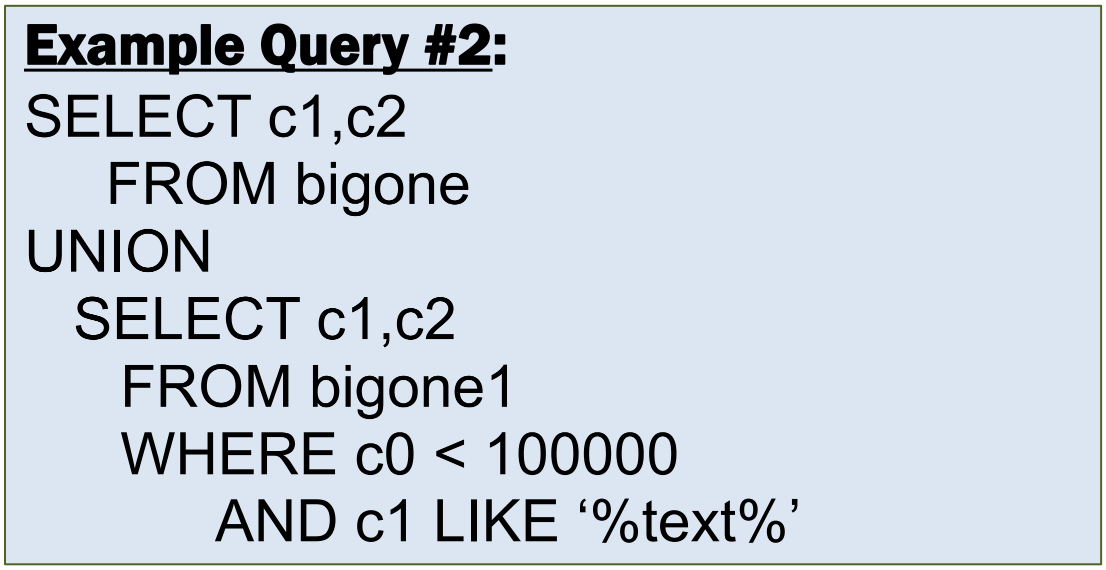
As a result every time the user fires a certain complex query, the "mapit" table is the semantic layer mapping the query to the MV. Figure 8 presents table "mapit" which has the Id, name, date of creation and the text of the query.

Id	***Name***	***Text***	***Mvid***	***Time***
1	Ag fun on bigone group by c0,c1	SELECT count(*) AS count, sum(bigone.c0) AS sum, avg(bigone.c0) AS avg, bigone.c1 , bigone.c0 FROM bigone WHERE bigone.c0 >800000 GROUP BY bigone.c0, bigone.c1 ORDER BY count	Mv1	12/10/2011 12:30:30

Figure 8 Table mapit which maps MV to a complex query

The “mapit” table converts the query into another query which is “select * from mv1”.

Another example of a complex query is presented in Figure9.



Example Query #2:

```
SELECT c1,c2
     FROM bigone
UNION
   SELECT c1,c2
      FROM bigone1
      WHERE c0 < 100000
            AND c1 LIKE ‘%text%’
```

Figure 9 Complex query example 2

After deploying Algorithm1, the “mapit” table would be presented in Figure 10.

Id	*Name*	*Text*	*Mvid*	*Time*
1	Ag fun on bigone group by c0,c1	SELECT count(*) AS count, sum(bigone.c0) AS sum, avg(bigone.c0) AS avg, bigone.c1 , bigone.c0 FROM bigone WHERE bigone.c0 >800000 GROUP BY bigone.c0, bigone.c1 ORDER BY count	Mv1	12/10/2011 12:30:30
2	Bigone& bigone1c1,c2	SELECT c1,c2 FROM bigone UNION SELECT c1,c2 FROM bigone1 WHERE c0 < 100000 AND c1 LIKE ‘%text%’	Mv2	13/10/2011 09:30:30

Figure 10 Table mapit updated

The last sample of a complex query is the worst case scenario, where this complex query produces an idol complex query. Such complex query is illustrated in Figure 11.

Example Query #3:

```
SELECT c1, c2, c7, c123, count(*)
     FROM bigone
UNION
SELECT c1, c2, c7, c123, count(c0)
     FROM bigone1
        WHERE c0<100000
            AND c7 LIKE ’%c1Text%’
            AND NOT IN
(‘c123Text’,’c3text’,’c4text’)
GROUP BY c1, c2, c7, c123
ORDER BY c123
```

Figure 11 Complex query example3

Figure 11 presents an idol complex query since this query has union, grouping, and sorting.

Table “mapit” would have a new record as a result of deploying the previous complex query. The new record is shown in Figure 12.

Id	***Name***	***Text***	***Mvid***	***Time***
3	Bigone&bigone1 All	SELECT c1, c2, c7, c123, count(*) FROM bigone UNION SELECT c1, c2, c7, c123, count(c0) FROM bigone1 WHERE c0<100000 AND c7 LIKE '%c1Text%' AND NOT IN ('c123Text','c3text','c4text') GROUP BY c1, c2, c7, c123 ORDER BY c123	Mv3	15/10/2011 12:20:44

Figure 12 Mapit updated 3

Space complexity:

Let S tot=S o+ S m where Stot is the total size of the database after applying MV's

S o is the original size of the database before applying any MV

S m is the size of all MV's which are used for all repetitive complex queries.

Without using MV's, space needed is O (N) where N is the space needed for the original database before adding MV's

After deploying MV's to present all repetitive complex queries the space needed is O (N) + O(M) where M is the space needed for all MV's when using algorithm1.

It is clear that O (M) is a fixed size in this phase where algorithm1 is used for existing records (static view) and no increment update is issued.

- <u>Algorithm 2</u>:
  - Purpose: Increase the time efficiency when accepting new records
  - Assumptions: Time spent when using the hybrid Algorithm would be more efficient than applying regular readings
  - Input:

        - New Records
        - Set of complex queries
    - Output:
        - Set of temporary materialized views

<u>Pseudo-code</u>:

This algorithm is used for new data (dynamic deployment)

- Read only new records adding to any query, by using a where clause depending on the timestamp column in the mega tables
- Create temporary MV for each query
- Map each temp MV to its original MV
- Drop all temp MV's
- Read records from MV

<u>Complexity</u>:

Let m be the number of records per cycle (week, day, hour, minute, second)

Let n be the number of cycles.

Assume that a fixed number of records would be inserted in the mega tables for each cycle.

Time needed to read the original tables after inserting the new records is calculated with this formula:

Ttot= $\sum_{i=1}^{n} mi$=m$\sum_{i=1}^{n} i$ m=m.n(n+1)/2=m.n^2→O(N^2)

When Algorithm2 is implied, the formula would be:

Ttot= $\sum_{i=1}^{n} m$ =m$\sum_{i=1}^{n} 1$ =m.n→O(N)

All records which include both the new and the old records are not needed to be read. A timestamp field is added in each mega table. This field is indexed to increase query performance efficiency. When inserting new records into the mega tables, the second algorithm is implied. For any new query, extra **where** clause, is added, depending on the date field.

The first Algorithm is applied, which creates MV on the new records only by creating temporary MV's which are called TMV's. Then the records are inserted from the temporary TMV's into the old MV's. Then the temporary MV's are dropped to save space.

Let us assume the following query Q1:

Select col1,col2, col3 from table1 union all select col2,col22,col30 from table2

Order by col1

Now, to create an MV, the following command is issued:

Create table mv1 as select col1,col2, col3 from table1 union all select col2,col22,col30 from table2 Order by col1;

The new status of table mapit is represented in the following table:

Id	***Name***	***Text***	***Mvid***	***Time***
3	Any descriptive name	Select col1,col2,col3 from table1 union all select col2,col22,col30 from table2 Order by col1	Mv1	15/03/2012 12:20:44

To obtain new records in a new cycle the same query is issued with new where clause so that Q1new will be:

Select col1,col2, col3 from table1 *where timestamp* > *15/03/2012 12:20:44* union all select col2,col22,col30 from table2 *where timestamp* > *15/03/2012 12:20:44*

Order by col1

Then, the temp MV is created and it is called TMV1.

Create table tmv1 as Select col1,col2, col3 from table1 *where timestamp* > *15/03/2012 12:20:44* union all select col2,col22,col30 from table2 *where timestamp* > *15/03/2012 12:20:44 o*rder by col1;

The last step is to insert the new records results into the original MV's and drop the temporary ones.

Insert into mv1 select * from tmv1;

Drop table tmv1.

At the end of this process Mv1 has the new data sets.

Let μ= the ratio between the total number of old records and the total number of new records in the entire existing tables.

$\mu = Rn/Ro \rightarrow 0 =< \mu <<< 1$

To demonstrate the process after applying the hybrid algorithm, consider the following scenario to explain the changes in efficiency. 1000 records every cycle are accepted. The cycle is one week and in the beginning 10000 records are stored in the database. However, the result would be presented in Figure 13 which shows the values of μ through 17 weeks:

Week	R0	Rn	µ
1	10000	1000	0.1000
2	11000	1000	0.0909
3	12000	1000	0.0833
4	13000	1000	0.0769
5	14000	1000	0.0714
6	15000	1000	0.0667
7	16000	1000	0.0625
8	17000	1000	0.0588
9	18000	1000	0.0556
10	19000	1000	0.0526
11	20000	1000	0.0500
12	21000	1000	0.0476
13	22000	1000	0.0455
14	23000	1000	0.0435
15	24000	1000	0.0417
16	25000	1000	0.0400
17	26000	1000	0.0385

Figure 13 (µ) Total number of old records / total number of new records

The same values, which are presented in Figure 13, are now represented in Figure 14 which shows the values in a chart type. The Y axis represents number of new records which are accepted in a cycle. The X axis presents number of cycles (here number of weeks).

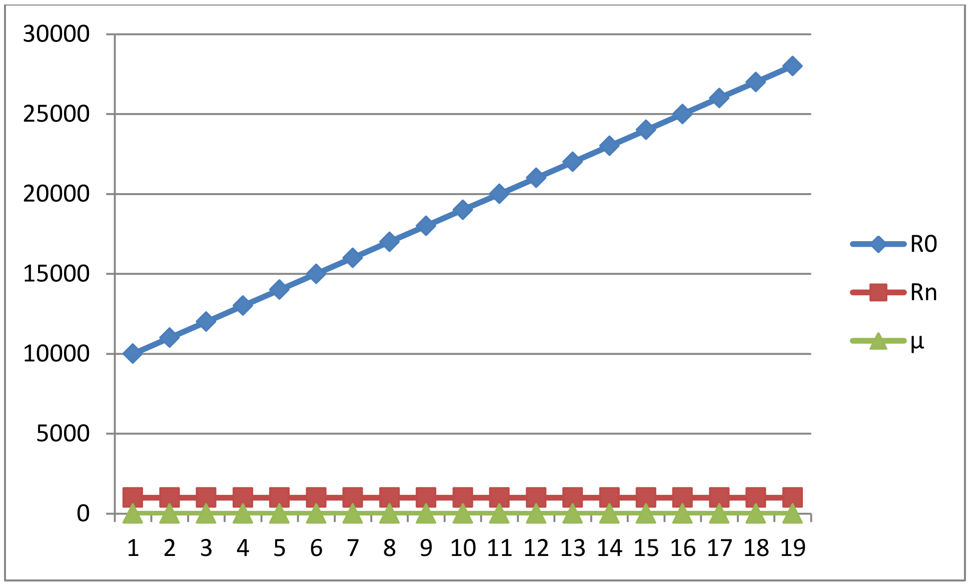


Figure 14 New records Rn, old records Ro, and the ratio between Rn/Ro (µ)

The value of µ is shown more clearly in the Figure 15 where µ is the ratio of Ro/Rn decreased every next cycle (week).

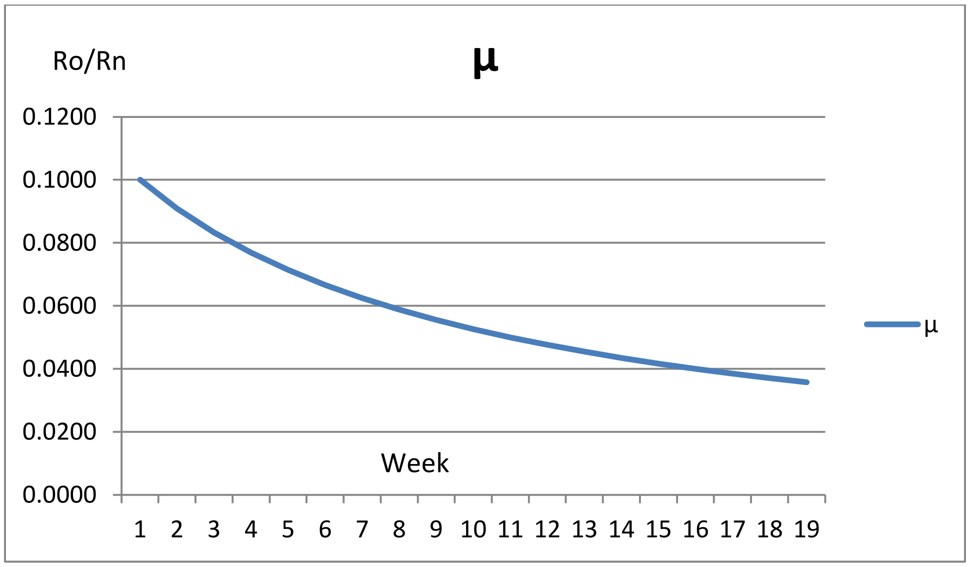


Figure 15 µ values according to cycles(weeks)

It was found that the more cycles the system has, the more µ decreases in value.

Figure 16 shows the importance of Algorithm2 by showing time saved for reading new records only.

		Time for DML Using Oracle or Postgres	Time for DML Using Algorithm2	Saving
# of new records	# of old records			
100K	1000 K	T(1100)	T(100)	1000K
100K	1200K	T(1300)	T(100)	1200K
…	…			
…..	…..			
100K	60000K	T(60100)	T(100)	60000K

Figure 16 Importance of μ

Two important fields were added to each table in the new database. The first one is a primary key constraint. It is named "Id". The second field is a timestamp. It is called "timestamp". The main purpose behind adding these two fields was to obtain two flags for each table. Actually, the "Id" field is a flag for each record in each table and for later usage.

The second field is produced to help us in applying our algorithms when it comes to the stage of accepting new records in each table.

In simple words, these two fields provide the solution for homogenizing the database and to play the role of a physical layer to transfer the data which is collected from non-traditional databases into a traditional one.

Space complexity:

Let S tot=S o+ S Tm where Stot is the total size of the database after applying MV's

S o is the original size of the database before applying any MV

S Tm is the size of all temporary MV's which are used for new records.

Without using MV's, space needed is O (N) where N is the space needed for the database after adding MV's in phase1.

After deploying incremental update to present all new records the space needed is O (N) + O(TM) where TM is the space needed for adding only new records to the original MV's.

O (TM) is a variant size in this phase where algorithm2 is used for new records (dynamic view) and an increment update is issued. TM based on μ which is the ratio between the number of new records and the number of the existing ones.

TM depends on the complexity of the repetitive query which means the space needed for copying only new records into the original MV's

4.3 Interface implementation

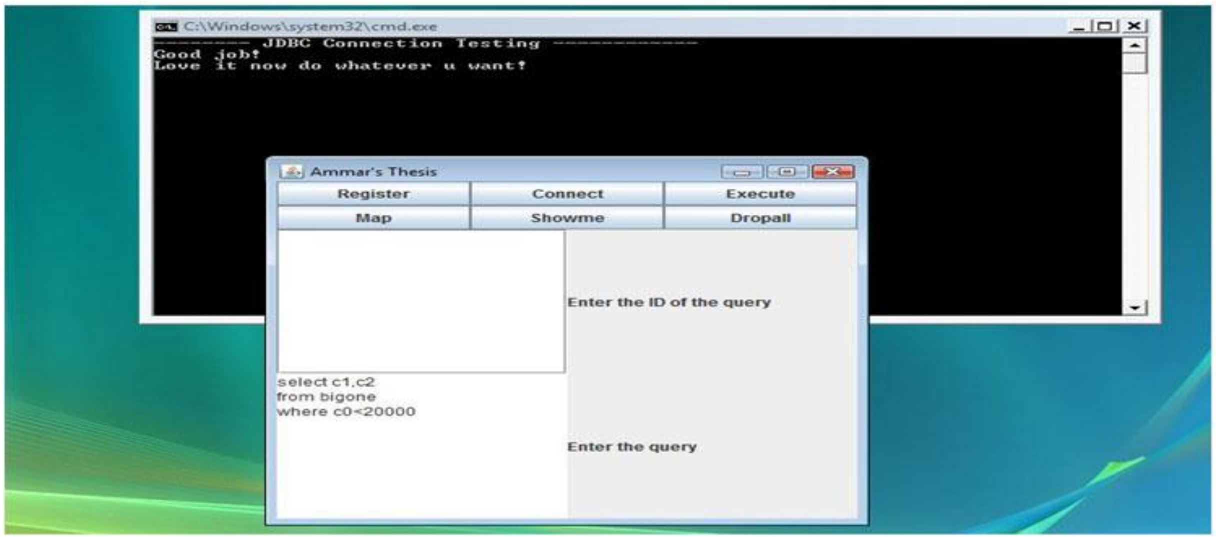
The selection of a suitable environment to implement the job of the MV's was not an easy process. A simple, object-oriented, multithreaded, distributed, and portable language was required. Java was the solution according to the nature of the data sets. Java is an independent platform language which helped the research to be deployed in different environments and domains.

Another aspect was the connectivity choice. An application programming interface that allows any programmer to access a database management system from a Java code was required [MISTR00, SEGOU05]. JDBC is a solution that allows multiple implementations to exist and be used by the same application. This interface provides a mechanism for dynamically loading the correct Java packages and drivers and registering them with the JDBC driver manager. JDBC supports creating and executing all DML statements like update, insert, and delete with respect to DDL statements, such as creating objects in the database.

Our Java interface (see Figure 17), which shows our implementation, has been designed and constructed based on:

- Abstract classes which could be used in any database
  - Java class to connect our classes to our database
  - Java class to register the required drivers for the database
  - Java class to implement the GUI
  - Java class to show mapping
  - Java class to drive the whole process

All the required abstract classes are included in Appendix B. Figure 17 illustrates the java interface.



```
-------- JDBC Connection Testing ------------
Good job!
Love it now do whatever u want!
```

```
select c1,c2
from bigone
where c0<20000
```

Figure 17 Java interface used in this thesis

4.4 Database implementation

Our research requires a suitable open source database which has suitable features and powerful controlling keys. Postgres was an excellent candidate for the job process. Postgres is a strong RDBMS where all the advantages of powerful giant databases, such as Oracle and Msql server, are invoked.

The process of homogenizing records from non-traditional databases into Postgres is simple and efficient. Postgres supports JDBC for our interface implementation. Such environment is shown in Figure 18.

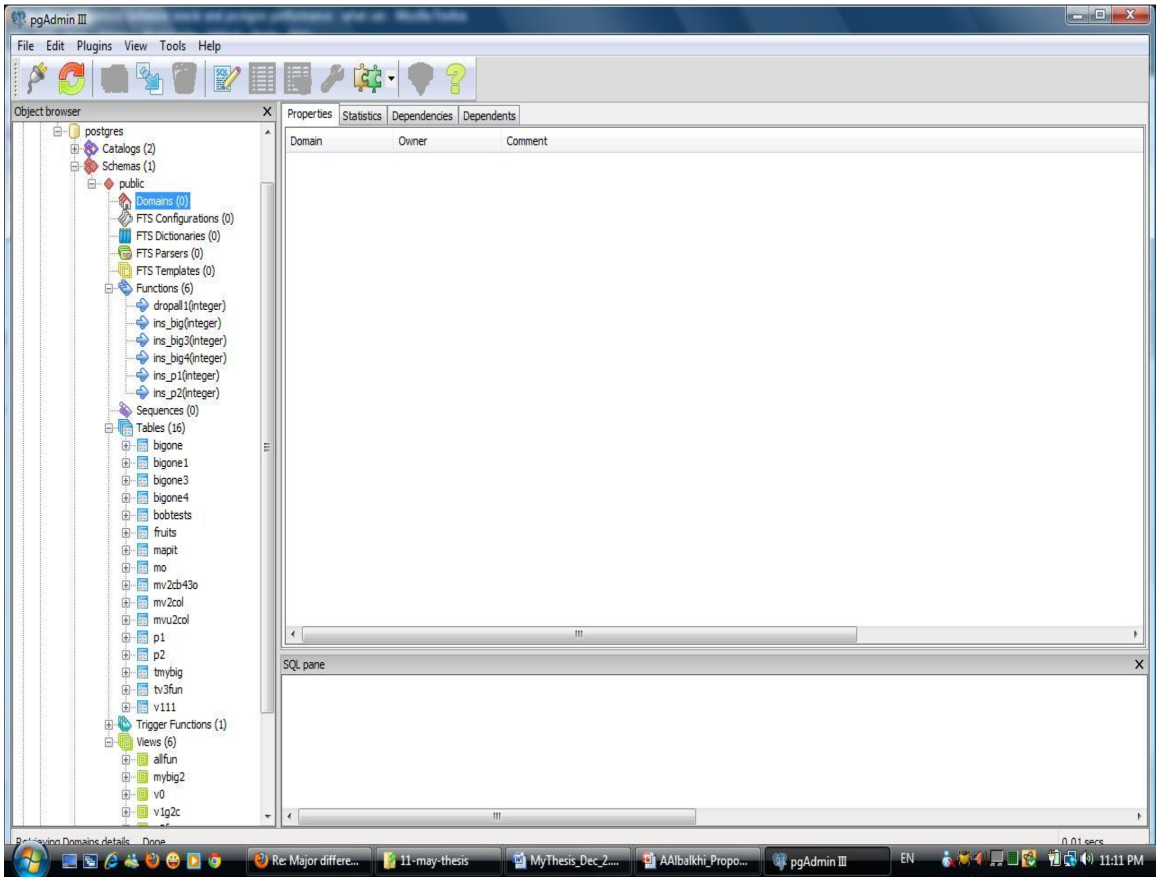


Figure 18 Postgres environment

4.5 Results

Various experiments were performed to study and confirm the feasibility of our approach:

- Select random complex queries on random data
- Record the time spent with and without using MV's

Primary results show that using MV on EPR gives a response time faster than using regular view (see Figure 19 and Figure 20). Our results also showed that the space of the database increased when new MV's are created (see Figure 21).

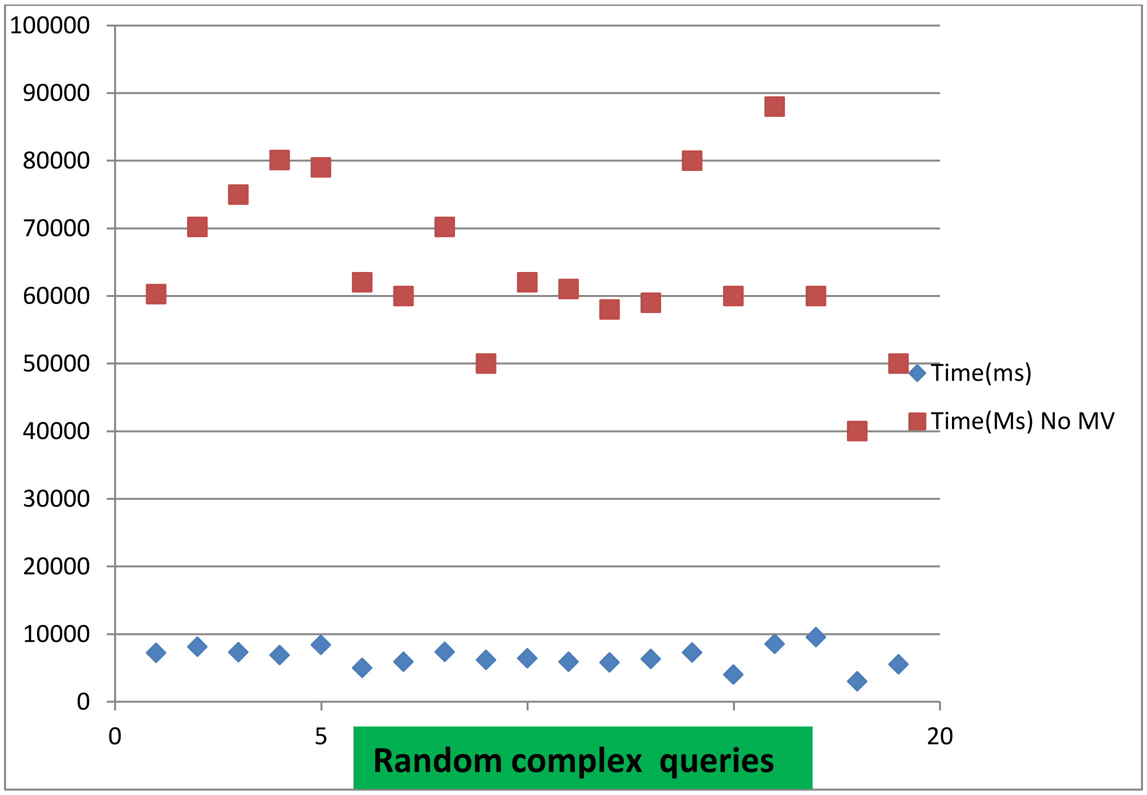


Figure 19 Time comparison with and without using MV using Windows(OS)

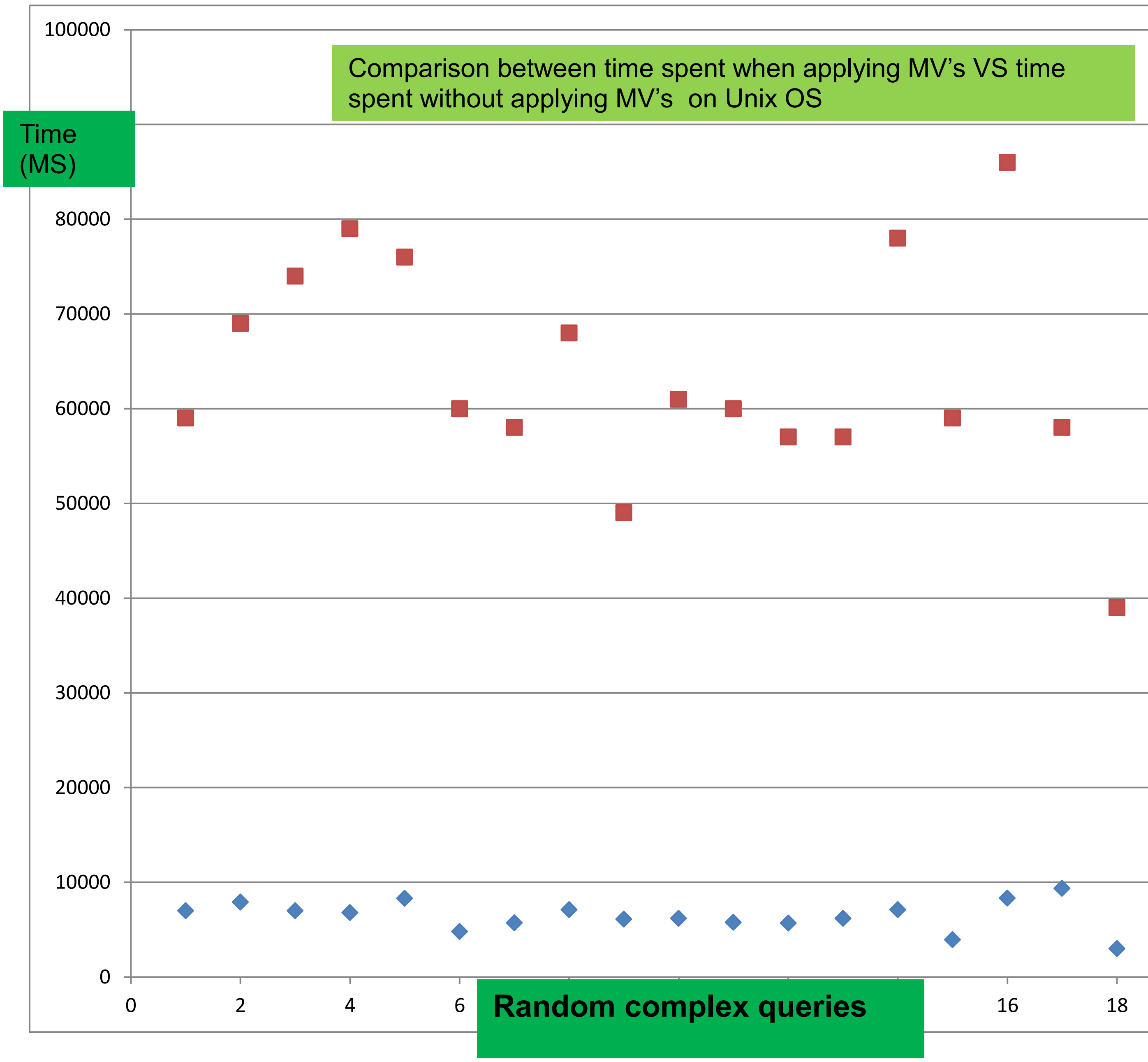


Figure 20 Time comparison with and witout using MV using UNIX (OS)

The same complex query time spent when applying MV's is at least four times less than the time spent in retrieving data from the same complex query without using MV's. Different operating systems were used to obtain a sense of background effect of O/S on the database operations; since modern O/S do not differ substantially, it was not expected to have any substantial effects. Windows and Unix were selected for their popularity of usage. The results show that there is a slight difference between the times spent with these two operating systems. Our experiments show that when using Unix, somewhat better results are collected than when Windows OS is used.

Many factors play important roles in the experiments which were applied in this research. It was found that timing results depend on the complexity of the query which is used in our algorithms. Therefore, it was necessary to specify the meaning of complexity of a query. The measurement of a query complexity is not measured by the number of lines of that query but by using union, group by, *where clause*, or sorting.

Results show that time efficiency is increased by applying MV's on heterogeneous datasets. Results are also able to show that in both assumptions, static and dynamic deployment, time efficiency is still gained and increased.

Clearly, the size of the database increases when using MV's. The size increment of the database relies on the complexity of the MV, which includes the results of the complex query. The size of the original data is not changed unless new records are inserted into the underlying tables.

MVs (#)	**size(Mb)**
mv1	2,059
mv2	2,059
mv3	2,069
mv4	2,072
mv5	2,077
mv6	3,823
mv7	3,823
mv8	3,840
mv9	5,082
mv10	5,212

Figure 21 Size of database when using MV’s

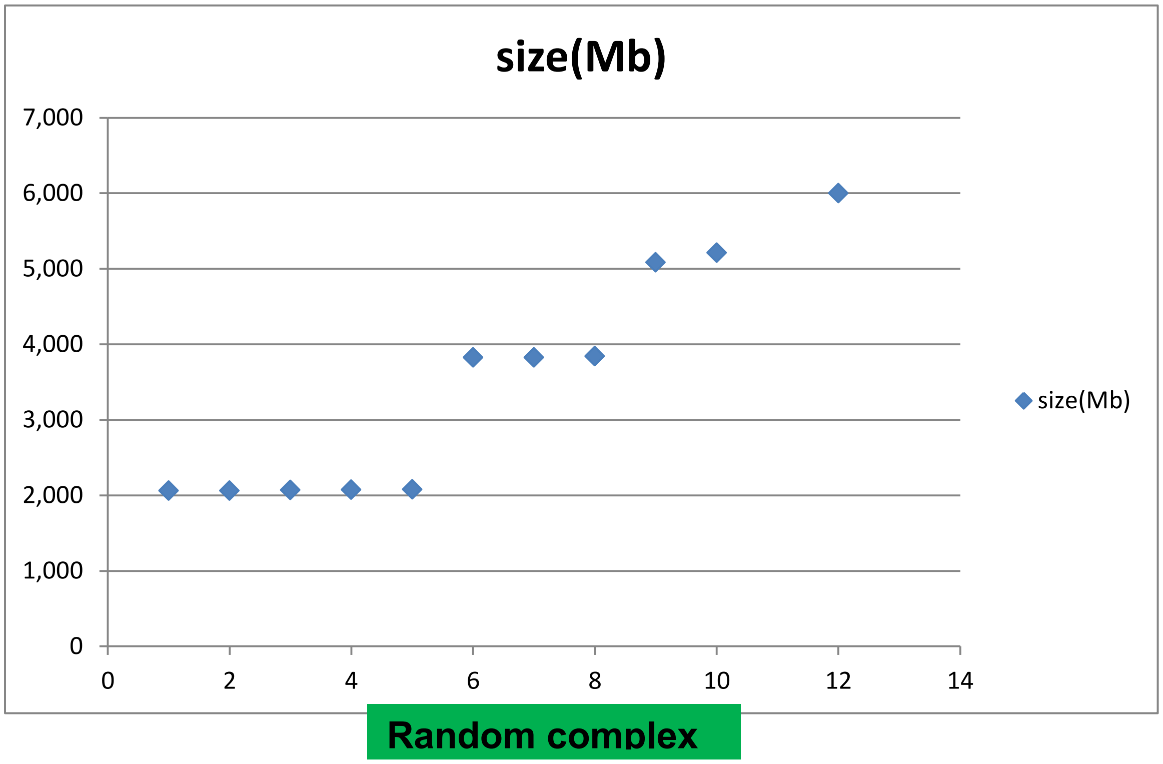


Figure 22 Chart of size of the database when using MV’s

Figure 22 shows that the increment size of the database depends on the complexity of the MV’s. The X axis presents number of sets of complex queries which were used while the Y axis presents the size of the database in megabyte. Results show that the more complex query is used, the more size is increased. It was obvious that the more MV’s are applied the more database size increases, since each one of these MV’s creates an independent object that contains different datasets with different format and store these data sets in the database.

It is important to refer to the fact that the increment size of the database could be resolved by using more hard drives since the storing devices are cheap compared to the cost of using RAM, although it was clear that memory prices have also declined significantly, recently, and memory capacities have increased substantially to support caching, in particular. It was found that with time passing, archiving should be used continuously since this technique would help solve the problem of size increment of the database.

In our interface an option was provided for decision makers to select an increase in time efficiency or space efficiency. The selection would, in fact, either apply our algorithms, which means increasing the time efficiency by applying MV's on each complex query, or not apply our algorithms, which means dropping all the MV's in the database at anytime.

4.6 Contribution

For each repetitive complex query, MV's can be applied to store the results of these queries as independent objects in the database. By placing MV's the time efficiency is gained. However, we sacrifice the space and size of the database each time the MV's are applied. The size would rely on a main factor which is the complexity of the query itself. The complexity of each repetitive query can be defined by containing grouping, union, sorting, and complex criteria in the WHERE clause of the query.

The main condition for applying MV's on complex queries is that those complex queries should be repetitive queries. If there is a one-time complex query, there is no need to apply MV's on it since the time spent on MV's would be more than the time needed for obtaining the results directly behind firing the query itself.

One of the impediments was how to guarantee database concurrency. In context of the classic Readers-Writers problem, readers do not wait for writers and writers do not wait for readers. Postgres does not support system level triggers; however, this impediment was solved by creating a scheduler in our abstract interface regardless of the programming language that was used.

The scheduler calls the functions which are responsible for creating and mapping MV’s inside the database. It also makes sure that the temporary MV’s are dropped after obtaining the new materialized records and inserting all of them into the appropriate materialized view.

4.7 Phase 1

Each repetitive complex query should be related to an MV. The mapping process requires a real physical table to reside inside the database. This table contains all necessary metadata to describe the mapping procedure. Having a real physical table for the mapping procedure would assist the researcher’s capability of retrieving all data needed for managing the MV’s used in the process.

After mapping is done correctly, the interface would always refer to the suitable MV associated with the repetitive complex query.

As shown previously, it was proved by experiments and equations that applying MV’s improves time efficiency when retrieving results from complex queries. The first phase would rely on our first algorithm, which was explained in detail in section 4.1.

4.8 Phase 2

When accepting new records SQL function is used which would add to each complex query an additional WHERE clause to grab new records only. The added WHERE clause would rely on the date field we added to each mega table since this column would be the main factor in the filtering process to allow reading only new records.

Collections of functions are provided to create temporary MV’s for the new inserted records and later, those temporary MV’s could be dropped to gain space and to guarantee database consistency.

All our functions could be called from the interface created using Java abstract classes; however, they could be implemented in other OOP languages, such as C++. The interface which is used would be responsible for scheduling the time for firing the triggers needed for our functions according to the absence of system level triggers in Postgres.

The timer, also known as “the scheduler“, will assure the database management system that is applying filtering and creating temporary MV’s would always perform updates before decision makers would be involve in any complex query. Another solution is to use manual maintenance as Postgres snapshot MV’s do the job. Either methodology would assure the database consistency.

A sample of SQL code is added in Appendix C. Also, abstract Java interface classes are in Appendix B.

This research on using MVs is innovative because it extends understanding of issues related to retrieving data from arbitrary heterogeneous and non-traditional datasets. This research could be applied on any heterogeneous datasets, such as surveys, healthcare, and questionnaire databases.

We claim that our methodology is important and efficient because it provides an optimal method for incremental update, which is sometimes mentioned as “updating MV’s”. Oracle uses an incremental update technique to refresh MV’s. A Complete refresh rebuilds the entire MVIEW. This is done by applying “refresh fast on commit” statement. The update process needs deleting all records which are in the MV then inserting the new records after all underlying tables are updated.

The incremental update, which Oracle uses, applies the following procedures:

- Delete all records which are inside the MV
- Commit the previous procedure
- Obtain all records (old and new) from the base tables
- Insert into the MV all records which were obtained in the previous step

Algorithm2 does not delete the records in any MV, which means saving the time needed for the deletion process, which Oracle is using. Algorithm2 inserts only new records by adding a where clause to each repetitive complex query, which means saving the time needed to insert all records.

Actually, Algorithm2 applies the following steps:

- Obtain new records from base tables
- Insert new records in MV

The Postgres database uses the same technique Oracle uses, but by applying triggers, functions and procedures.

Algorithm2 shows the importance of µ, which is the ratio of new records to old records. When any DML is applied to any MV in the database, time is saved by the factor of 1/ µ. Since µ is getting smaller after each cycle, then time saving is getting bigger, which means time efficiency is increased after each cycle. Algorithm2 does not DML all records (new +old), it manipulates only new records.

CHAPTER V

CONCLUSION

This thesis is based on the question, how can one design and implement a relational database system approach, using materialized views, that allows for comparison of the time and space dependencies in implementing and maintaining complex queries applied to heterogeneous large datasets?

To achieve this goal, an open source RDBMS (Postgres) was used, which was suitable for applying materialized views (MV's) on the heterogeneous datasets.

The usage of MV's leads to applying our two algorithms: the static algorithm referred to as Algorithm 1, and the hybrid algorithm referred to as Algorithm 2. These two algorithms are the primary achievements of this thesis.

5.1 Thesis achievement

The first objective, successfully achieved, was increasing time efficiency when applying MV's on each repetitive complex query in the large heterogeneous data sets by using algorithm 1.This algorithm was used to furnish the mapping road between the MV's and the repetitive complex queries as explained in Section 4.2.

The second objective, also successfully achieved, increased time efficiency when accepting new records into the database by using algorithm 2 as explained in Section 4.2.

Finally, an implementation of the two algorithms was completed with two systems. The first one is a prototype system that is presented in the worst case scenario. The second one is presented via a real life healthcare system. The interface is built on Java abstract classes to connect and register the required drivers for any database and to manipulate the datasets which reside in a heterogeneous form and then to homogenize these datasets into RDBMS.

The storing and retrieving of patient records in different health organizations in different formats was the starting point foundation of this research since each health organization uses a different type of presentation for data.

The inventive aspect of this thesis is noted that by applying the methodology developed through this research, health organization systems will be improved. Specifically, they will save money as well as time. Consequently, health personnel will help more patients in a more timely manner because they will be able to make more effective medical decisions.

The results of this research shows that time efficiency is increased 4-10 times depending on the complexity of the repetitive query as shown in Figure 20.

5.2 Comments

This research presented challenges which include column naming and data type shown with different format representations in large random heterogeneous datasets. A technique was developed that could allow supervision of a main flag in each of the datasets, which is the ultimate way to homogenize the datasets in a certain domain.

Another challenge is satisfied by adding a timestamp flag to the datasets; by so doing, the mechanism for the filtering process was successfully achieved. Applying MV’s on heterogeneous datasets in RDBMS without creative methodologies would cause time proficiency waste. This thesis provides a full implementation of the Java interface which is simple and flexible. The interface provides two options for time and space efficiency. Also the developed hardcode could be used in different RDBMS’s.

The contribution of this research is based on the modification of existing heterogeneous data sets by applying MV’s to query complex SQL statements. The two algorithms developed were used to achieve time saving when dealing with the current records as well as accepting new records to guarantee database consistency.

This research demonstrates that by applying MV’s on complex queries, decision makers are assisted in getting the required results more efficiently. This research is a milestone for future research seeking development of all kinds of DML and DDL statements issued with any heterogeneous large datasets.

From this research, it is now clear that MV’s can be coupled with any SQL statements to work with the newly developed interface, efficiently and automatically, in any similar database system. To the best of our knowledge, this research is the first research to query complex datasets in heterogeneous database using MV’s. Finally, research to date has applied MV’s only to traditional databases. This current research applies MV’s to non-traditional databases where time efficiency is also improved. In this way, the research accomplished in this thesis adds scientific value, contributing an innovative approach to the computer science field.

We emphasize that as part of this research, we conducted a continuing broad and deep review of the academic literature regarding the subject of querying large heterogeneous data sets.

5.3 Future work

Our future work will consider deleting records and the effect of this process on the database. This may include two types of deletion, namely physical and logical deletion.

We believe that deletion process is an essential job, since keeping historical records is considered, in health care space, an important stage of handling data process.

Also, future work will take in consideration the archiving process of the database after using MV’s.

APPENDIX A

JAVA CLASSES

A.1 Database Javabeans

The following class is used for database connectivity

<u>Class Mydriver</u>

java.lang.Object

Mydriver

public class Mydriver

extendsjava.lang.Object

Constructor Summary

Constructors

Constructor and Description
Mydriver()

Method Summary

Methods

Modifier and Type	Method and Description
Void	connectme()
Void	dropall()
java.lang.String	dropit(int num)
Void	exe_query(java.lang.String myquery)
java.lang.String	mapit(java.lang.String s1, int num)

Void	play()
Void	registerme()
Void	sel_query(java.lang.String myquery)
Void	showdata()

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Mydriver

public Mydriver()

Method Detail

registerme

public void registerme()

connectme

public void connectme()

play

public void play()

sel_query

public void sel_query(java.lang.String myquery)

exe_query

public void exe_query(java.lang.String myquery)

mapit

public java.lang.String mapit(java.lang.String s1,

int num)

dropit

public java.lang.String dropit(int num)

showdata

public void showdata()

dropall

public void dropall()

A.2 GUI classes

The following class shows a sample of the graphical user interface, which is used to apply the methodology of this research.

Class Mygui

java.lang.Object

Mygui

All Implemented Interfaces:

java.awt.event.ActionListener, java.util.EventListener

public class Mygui

extendsjava.lang.Object

implementsjava.awt.event.ActionListener

Constructor Summary

Constructors

Constructor and Description

Mygui()

Method Summary

Methods

Modifier and Type	Method and Description
Void	actionPerformed(java.awt.event.ActionEvent e)
Void	init()

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Mygui

public Mygui()

Method Detail

actionPerformed

public void actionPerformed(java.awt.event.ActionEvent e)

Specified by:

actionPerformed in interface java.awt.event.ActionListener

init

public void init()

public class AmmarTable

java.lang.Object

java.awt.Component

java.awt.Container

java.awt.Window

java.awt.Frame

javax.swing.JFrame

AmmarTable

All Implemented Interfaces:

java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants

A.3 Swing classes

The following class presents the swing package, which is used to show the results of the repetitive complex queries in a table included in the GUI.
public class AmmarTable

extendsjavax.swing.JFrame

See Also:

Serialized Form

Nested Class Summary

Nested classes/interfaces inherited from class javax.swing.JFrame

javax.swing.JFrame.AccessibleJFrame

Nested classes/interfaces inherited from class java.awt.Frame

java.awt.Frame.AccessibleAWTFrame

Nested classes/interfaces inherited from class java.awt.Window

java.awt.Window.AccessibleAWTWindow, java.awt.Window.Type

Nested classes/interfaces inherited from class java.awt.Container

java.awt.Container.AccessibleAWTContainer

Nested classes/interfaces inherited from class java.awt.Component

java.awt.Component.AccessibleAWTComponent, java.awt.Component.BaselineResizeBehavior, java.awt.Component.BltBufferStrategy, java.awt.Component.FlipBufferStrategy

Field Summary

Fields inherited from class javax.swing.JFrame

accessibleContext, EXIT_ON_CLOSE, rootPane, rootPaneCheckingEnabled

Fields inherited from class java.awt.Frame

CROSSHAIR_CURSOR, DEFAULT_CURSOR, E_RESIZE_CURSOR, HAND_CURSOR, ICONIFIED, MAXIMIZED_BOTH, MAXIMIZED_HORIZ, MAXIMIZED_VERT, MOVE_CURSOR, N_RESIZE_CURSOR, NE_RESIZE_CURSOR, NORMAL, NW_RESIZE_CURSOR, S_RESIZE_CURSOR, SE_RESIZE_CURSOR, SW_RESIZE_CURSOR, TEXT_CURSOR, W_RESIZE_CURSOR, WAIT_CURSOR

Fields inherited from class java.awt.Component

BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

Fields inherited from interface javax.swing.WindowConstants

DISPOSE_ON_CLOSE, DO_NOTHING_ON_CLOSE, HIDE_ON_CLOSE

Fields inherited from interface java.awt.image.ImageObserver

ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Constructor Summary

Constructors

Constructor and Description
AmmarTable()

Method Summary

Methods

Modifier and Type	Method and Description
static void	main(java.lang.String[] arg)

Methods inherited from class javax.swing.JFrame

addImpl, createRootPane, frameInit, getAccessibleContext, getContentPane, getDefaultCloseOperation, getGlassPane, getGraphics, getJMenuBar, getLayeredPane, getRootPane, getTransferHandler, isDefaultLookAndFeelDecorated, isRootPaneCheckingEnabled, paramString, processWindowEvent, remove, repaint, setContentPane, setDefaultCloseOperation, setDefaultLookAndFeelDecorated, setGlassPane, setIconImage, setJMenuBar, setLayeredPane, setLayout, setRootPane, setRootPaneCheckingEnabled, setTransferHandler, update

Methods inherited from class java.awt.Frame

addNotify, getCursorType, getExtendedState, getFrames, getIconImage, getMaximizedBounds, getMenuBar, getState, getTitle, isResizable, isUndecorated, remove, removeNotify, setBackground, setCursor, setExtendedState, setMaximizedBounds, setMenuBar, setOpacity, setResizable, setShape, setState, setTitle, setUndecorated

Methods inherited from class java.awt.Window

addPropertyChangeListener, addPropertyChangeListener, addWindowFocusListener, addWindowListener, addWindowStateListener, applyResourceBundle, applyResourceBundle, createBufferStrategy, createBufferStrategy, dispose, getBackground, getBufferStrategy, getFocusableWindowState, getFocusCycleRootAncestor, getFocusOwner, getFocusTraversalKeys, getIconImages, getInputContext, getListeners, getLocale, getModalExclusionType, getMostRecentFocusOwner, getOpacity, getOwnedWindows, getOwner, getOwnerlessWindows, getShape, getToolkit, getType, getWarningString, getWindowFocusListeners, getWindowListeners, getWindows, getWindowStateListeners, hide, isActive, isAlwaysOnTop, isAlwaysOnTopSupported, isAutoRequestFocus, isFocusableWindow, isFocusCycleRoot, isFocused, isLocationByPlatform, isOpaque, isShowing, isValidateRoot, pack, paint, postEvent, processEvent, processWindowFocusEvent, processWindowStateEvent, removeWindowFocusListener, removeWindowListener, removeWindowStateListener, reshape, setAlwaysOnTop, setAutoRequestFocus, setBounds, setBounds, setCursor, setFocusableWindowState, setFocusCycleRoot, setIconImages, setLocation, setLocation, setLocationByPlatform, setLocationRelativeTo, setMinimumSize, setModalExclusionType, setSize, setSize, setType, setVisible, show, toBack, toFront

Methods inherited from class java.awt.Container

add, add, add, add, add, addContainerListener, applyComponentOrientation, areFocusTraversalKeysSet, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getAlignmentX, getAlignmentY, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getComponentZOrder, getContainerListeners, getFocusTraversalPolicy, getInsets, getLayout, getMaximumSize, getMinimumSize, getMousePosition, getPreferredSize, insets, invalidate, isAncestorOf, isFocusCycleRoot, isFocusTraversalPolicyProvider, isFocusTraversalPolicySet, layout, list, list, locate, minimumSize, paintComponents, preferredSize, print, printComponents, processContainerEvent, remove, removeAll, removeContainerListener, setComponentZOrder, setFocusTraversalKeys, setFocusTraversalPolicy, setFocusTraversalPolicyProvider, setFont, transferFocusDownCycle, validate, validateTree

Methods inherited from class java.awt.Component

action, add, addComponentListener, addFocusListener, addHierarchyBoundsListener, addHierarchyListener, addInputMethodListener, addKeyListener, addMouseListener, addMouseMotionListener, addMouseWheelListener, bounds, checkImage, checkImage, coalesceEvents, contains, contains, createImage, createImage, createVolatileImage, createVolatileImage, disable, disableEvents, dispatchEvent, enable, enable, enableEvents, enableInputMethods, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange,

firePropertyChange, firePropertyChange, getBaseline, getBaselineResizeBehavior, getBounds, getBounds, getColorModel, getComponentListeners, getComponentOrientation, getCursor, getDropTarget, getFocusListeners, getFocusTraversalKeysEnabled, getFont, getFontMetrics, getForeground, getGraphicsConfiguration, getHeight, getHierarchyBoundsListeners, getHierarchyListeners, getIgnoreRepaint, getInputMethodListeners, getInputMethodRequests, getKeyListeners, getLocation, getLocation, getLocationOnScreen, getMouseListeners, getMouseMotionListeners, getMousePosition, getMouseWheelListeners, getName, getParent, getPeer, getPropertyChangeListeners, getPropertyChangeListeners, getSize, getSize, getTreeLock, getWidth, getX, getY, gotFocus, handleEvent, hasFocus, imageUpdate, inside, isBackgroundSet, isCursorSet, isDisplayable, isDoubleBuffered, isEnabled, isFocusable, isFocusOwner, isFocusTraversable, isFontSet, isForegroundSet, isLightweight, isMaximumSizeSet, isMinimumSizeSet, isPreferredSizeSet, isValid, isVisible, keyDown, keyUp, list, list, list, location, lostFocus, mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move, nextFocus, paintAll, prepareImage, prepareImage, printAll, processComponentEvent, processFocusEvent, processHierarchyBoundsEvent, processHierarchyEvent, processInputMethodEvent, processKeyEvent, processMouseEvent, processMouseMotionEvent, processMouseWheelEvent, removeComponentListener, removeFocusListener, removeHierarchyBoundsListener, removeHierarchyListener, removeInputMethodListener, removeKeyListener, removeMouseListener, removeMouseMotionListener, removeMouseWheelListener, removePropertyChangeListener, removePropertyChangeListener, repaint, repaint, repaint, requestFocus, requestFocus, requestFocusInWindow, requestFocusInWindow, resize, resize, revalidate, setComponentOrientation, setDropTarget, setEnabled, setFocusable, setFocusTraversalKeysEnabled, setForeground, setIgnoreRepaint, setLocale, setMaximumSize, setName, setPreferredSize, show, size, toString, transferFocus, transferFocusBackward, transferFocusUpCycle

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.awt.MenuContainer

getFont, postEvent

Constructor Detail

AmmarTable

public AmmarTable()

Method Detail

main

public static void main(java.lang.String[] arg)

APPENDIX B

SQL SAMPLE CODE

B.1 MV's manipulation

The following SQL functions present samples of how to manipulate the MV's in the database.

```
- Function: dropall1(integer)

-- DROP FUNCTION dropall1(integer);

CREATE OR REPLACE FUNCTION dropall1(xx integer)

  RETURNS integer AS

$BODY$

DECLARE

row     record;

BEGIN

    FOR row IN

        SELECT

table_schema,  table_name

        FROM

information_schema.tables

        WHERE

table_type = 'BASE TABLE'

        --AND

            --table_schema = _schema

        AND
```

```
table_name ILIKE ('v%')
    LOOP
        EXECUTE 'DROP TABLE ' || quote_ident(row.table_schema) || '.' ||
quote_ident(row.table_name);
        RAISE INFO 'Dropped table: %', quote_ident(row.table_schema) || '.' ||
quote_ident(row.table_name);
    END LOOP;
return 1;
END;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE
  COST 100;

CREATE OR REPLACE FUNCTION cre_map(myq text)
  RETURNS text AS
$BODY$
declare
cre text='create table '||'mv'  ;
cre1 text;
ins text='insert into mapq values (';
co int;--counter
myd date=now();
des text='select * from mv';
BEGIN

select count(*) from mapq into co;

--get the last number of mapq row to create temp mv's where mapq is the table which

--has the queries and the mv's which mapped to each one

if co<1 then

        co=1;

        else

        co=co+1;


end if;

cre1 =cre|| co ||' as ' ||myq;

des=des||co;

execute cre1; --create the mv

 Execute ins||co||','||quote_nullable(myq)||','||quote_nullable(des)||')'; --map the mv with the

query

 -- call

 --selectcre_map('select c0 from bigone where c0<100 union select c0 from bigone4

where c0<100');

RETURN cre1;

END;

 $BODY$

  LANGUAGE 'plpgsql' VOLATILE;

CREATE TABLE mapq
```

```
(
query_id integer NOT NULL,
query_txt text,
view_des text,
myd date default now(),
  CONSTRAINT pk_mapq PRIMARY KEY (query_id)
)
  -- this is a function which adds to any sql query the date criteria
--this is used for my 2nd algorithm which needs to filter new records
CREATE OR REPLACE FUNCTION ctext(mytxttext,mydate text)
  RETURNS text AS
$BODY$
declare
cre text='create table '||'tempmv'  ;
mo text;
tal text;
ins text='insert into mv';
sql text=' select * from tempmv';
xxint;
len integer;
BEGIN
 --assuming that every query has where clause
select count(*) from mapit into xx;
```

```
--get the last number of mapit table to create temp mv's where mapit is the table which
--has the queries and the mv's which mapped to each one
xx=xx+1;
tal=replace(mytxt, 'where', 'where timestamp >'|| quote_nullable(mydate)||' and ');
mo =cre|| xx ||' as ' ||tal;
 Execute ins||xx||sql||xx; --insert all the new records into the mv

RETURN mo;
END;
 $BODY$
  LANGUAGE 'plpgsql' VOLATILE ;
-- This function would create the tempmv for each query then
-- it would insert new records into the approp mv

CREATE OR REPLACE FUNCTION tem_mv(myidinteger,mytxt text, mydate text)
  RETURNS text AS
$BODY$
declare
cre text='create table '||'tempmv'  ;--to create the temp mvs
mo text;
sen text;
ins text='insert into mv';
sql text=' select * from tempmv';
```

```
len integer;

BEGIN

sen=replace(mytxt, 'where', 'where date >'|| quote_nullable(mydate)||' and ');

mo =cre|| myid ||' as ' ||sen;

 Execute ins||myid||sql||myid; --insert all the new records into the mv



RETURN mo;

END;

 $BODY$

  LANGUAGE 'plpgsql' VOLATILE;
```

B.2 Prototype functions

The following SQL code shows samples of how to manipulate the prototype data
Including DML.

```
-- Function: ins_big(integer)

-- DROP FUNCTION ins_big(integer);

CREATE OR REPLACE FUNCTION ins_big(xx integer)

  RETURNS integer AS

$BODY$

declare

x integer=2;

BEGIN

            loop

   Insert into bigone3 values(x,

'c1 text ','c2 text ','c3 text ','c4 text ','c5 text ','c6 text ','c7 text ','c8 text ','c9 text ',
```

'c10 text ','c11 text ','c12 text ','c13 text ','c14 text ','c15 text ','c16 text ','c17 text ','c18 text ','c19 text ','c20 text ','c21 text ','c22 text ','c23 text ','c24 text ','c25 text ','c26 text ','c27 text ','c28 text ','c29 text ','c30 text ','c31 text ','c32 text ','c33 text ','c34 text ','c35 text ','c36 text ','c37 text ','c38 text ','c39 text ','c40 text ','c41 text ','c42 text ','c43 text ','c44 text ','c45 text ','c46 text ','c47 text ','c48 text ','c49 text ','c50 text ','c51 text ','c52 text ','c53 text ','c54 text ','c55 text ','c56 text ','c57 text ','c58 text ','c59 text ','c60 text ','c61 text ','c62 text ','c63 text ','c64 text ','c65 text ','c66 text ','c67 text ','c68 text ','c69 text ','c70 text ','c71 text ','c72 text ','c73 text ','c74 text ','c75 text ','c76 text ','c77 text ','c78 text ','c79 text ','c80 text ','c81 text ','c82 text ','c83 text ','c84 text ','c85 text ','c86 text ','c87 text ','c88 text ','c89 text ','c90 text ','c91 text ','c92 text ','c93 text ','c94 text ','c95 text ','c96 text ','c97 text ','c98 text ','c99 text ','c100 text ','c101 text ','c102 text ','c103 text ','c104 text ','c105 text ','c106 text ','c107 text ','c108 text ','c109 text ','c110 text ','c111 text ','c112 text ','c113 text ','c114 text ','c115 text ','c116 text ','c117 text ','c118 text ','c119 text ','c120 text ','c121 text ','c122 text ','c123 text ','c124 text ','c125 text ','c126 text ','c127 text ','c128 text ','c129 text ','c130 text ','c131 text ','c132 text ','c133 text ','c134 text ','c135 text ','c136 text ','c137 text ','c138 text ','c139 text ','c140 text ','c141 text ','c142 text ','c143 text ','c144 text ','c145 text ','c146 text ','c147 text ','c148 text ','c149 text ','c150 text ',151 text ','c152 text ','c153 text ','c154 text ','c155 text ','c156 text ','c157 text ','c158 text ','c159 text ','c160 text ','c161 text ','c162 text ','c163 text ','c164 text ','c165 text ','c166 text ','c167 text ','c168 text ','c169 text ','c170 text ','c171 text ','c172 text ','c173 text ','c174 text ','c175 text ','c176 text ','c177 text ','c178 text ','c179 text ','c180 text ','c181 text ','c182 text ','c183 text ','c184 text ','c185 text ','c186 text ','c187 text ','c188 text ','c189 text ','c190 text ','c191 text ','c192 text ','c193 text ','c194

```
text ','c195 text ','c196 text ','c197 text ','c198 text ','c199 text ','c200 text ','c201 text
','c202 text ','c203 text ','c204 text ','c205 text ','c206 text ','c207 text ','c208 text ','c209
text ','c210 text ','c211 text ','c212 text ','c213 text ','c214 text ','c215 text ','c216 text
','c217 text ','c218 text ','c219 text ','c220 text ','c221 text ','c222 text ','c223 text ','c224
text ','c225 text ','c226 text ','c227 text ','c228 text ','c229 text ','c230 text ','c231 text
','c232 text ','c233 text ','c234 text ','c235 text ','c236 text ','c237 text ','c238 text ','c239
text ','c240 text ','c241 text ','c242 text ','c243 text ','c244 text ','c245 text ','c246 text
','c247 text ','c248 text ','c249 text ','c250 text ');
x=x+1;
exit when x=1000000;
end loop;
       RETURN xx;
END;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE
  COST 100;



-- Function: ins_p1(integer)
-- DROP FUNCTION ins_p1(integer);
CREATE OR REPLACE FUNCTION ins_p1(xx integer)
  RETURNS integer AS
$BODY$
```

```
declare
x integer=500000;
BEGIN
            loop
   Insert into p1 values(
x,
'apple',
x+5,
now()
);
x=x+1;
exit when x=600000;
end loop;
    RETURN xx;
END;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE
  COST 100;
-- Function: ins_p2(integer)
-- DROP FUNCTION ins_p2(integer);
CREATE OR REPLACE FUNCTION ins_p2(xx integer)
  RETURNS integer AS
$BODY$
```

```
declare
x integer=500000;
BEGIN
            loop
   Insert into p2 values(
x,
'banana',
x+5,
now()
);
x=x+1;
exit when x=600000;
end loop;
    RETURN xx;
END;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE
  COST 100;
-- Function: test()
-- DROP FUNCTION test();
CREATE OR REPLACE FUNCTION test()
  RETURNS trigger AS
$BODY$
```

```
DECLARE
new_namevarchar;
new_phonenumvarchar;
BEGIN
IF(TG_OP='INSERT') THEN
INSERT INTO phonebook(name,phonenum) VALUES(NEW.name,NEW.phonenum);
END IF;
RETURN NEW;
END;$BODY$
  LANGUAGE 'plpgsql' VOLATILE
  COST 100;
ALTER FUNCTION test() OWNER TO postgres;
-- DROP TABLE p1;
CREATE TABLE p1(
pidbigint,
pname text,
pricebigint,
pdate date DEFAULT now()
)
WITH (
  OIDS=FALSE
);
-- Table: p2
-- DROP TABLE p2;

CREATE TABLE p2

(

pidbigint,

pname text,

pricebigint,

pdate date DEFAULT now()

)

WITH (

  OIDS=FALSE

);

--For algorithm 1

--This function is to map mv to complex query

-- and to create the mv @ the same time
```

BIBLIOGRAPHY

[BAHLO09]	Bahloul S (2009) Access Control to Materialized Views: An Inference-Based Approach. In: Proceedings of the 2011 Joint EDBT/ICDT Ph.D. Workshop ACM New York, NY, USA© 2011.
[CALVA11]	Calvanese D, Giacomo G, Lenzerini M, Rosati R (2011) View-based query answering in Description Logics: Semantics and Complexity. In: Journal of Computer and System Sciences.
[GONG08]	Gong A, Zhao W (2008) Clustering-based Dynamic Materialized View Selection Algorithm. In: Fuzzy Systems and Knowledge Discovery, 2008. FSKD '08. Fifth International Conference.
[GLASE05]	John Glaser, PhD, “State of Information Technology to Support Clinical Research”, presentation at Clinical Research Forum, March 29, 2005, p. 13.
[HU91]	Prokosch HU et al: WING - Entering a New Phase of Electronic Data Processing at the Giessen University Hospital. Meth Inform Med 30 (1991) 289-298.
[KABRA06]	Kabra G, Ramamurthy R, Sudarshan S (2006) Redundancy and Information Leakage in Fine-Grained Access Control. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data ACM New York, NY, USA ©2006.
[KARDE10]	Karde P, Thakare V (2010) Selection & Maintenance of Materialized View and Its Application for Fast Query Processing: A Survey. In: International Journal of Computer Science & Engineering Survey (IJCSES) Vol.1, No.2, November 2010.
[LI11]	Li F, Ishikawa Y (2011) Using Materialized Views to Enhance a Traceable P2P Record Exchange Framework. In: Journal Of Advances In Information Technology, VOL. 2, NO. 1, FEBRUARY 2011.
[LI10a]	Li X, Qian X, Jiang J, Wang Z (2010) Shuffled Frog Leaping

Algorithm for Materialized Views Selection. In: 2010 Second International Workshop on Education Technology and Computer Science.

[LI10b] Li D, Han L, Ding Y (2010) SQL Query Optimization Methods of Relational Database System. In: 2010 Second International Conference on Computer Engineering and Applications.

[LIJUA09] Lijuan Z, Xuebin G, Linshuang W, Qian S (2009) Efficient Materialized View Selection Dynamic Improvement Algorithm. In: Fuzzy Systems and Knowledge Discovery, 2009. FSKD '09. Sixth International Conference.

[MISTR00] Mistry H, Roy P, Sudarshan S (2001) Materialized View Selection and Maintenance Using MultiQuery Optimization. In: SIGMOD '01 Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data ACM New York, NY, USA ©2001.

[RYENG11] Ryeng N, Hauglid J, Nørvåg K (2011) Site-Autonomous Distributed Semantic Caching. In: SAC'11 March 21–25, 2011, TaiChung, Taiwan. Copyright 2011 ACM 978-1-4503-0113-8/11/03.

[SEGOU05] Segoufin L, Vianu V (2005) Views and queries: determinacy and rewriting. In: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems ACM New York, NY, USA ©2005.

[SHUKL00] Shukla A, Deshpande P, Naughton J (2000) Materialized View Selection for Multi-cube Data Models. In: Advances in Database Technology — EDBT 2000 Lecture Notes in Computer Science, 2000, Volume 1777/2000, 269-284, DOI: 10.1007/3-540-46439-5_19.

[YU03] Yu J, Yao X, Choi C, Gou G (2003) Materialized View Selection as Constrained Evolutionary Optimization. In: Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on.

VITA AUCTORIS

Ammar Soliman Albalkhi was born in 1969 in Damascus, Syria. He graduated from Alhashimi High School in 1987 after which he went to the University of Damascus where he obtained an honor B. Sc. (Bachelor of Engineering) degree in Electronic Engineering in 1997. While obtaining his undergraduate degree, Ammar worked as a developer at NICE (National Information Center). After obtaining his undergraduate degree, he worked as a trainer/developer at New Horizons until he gained his second B.Sc. (Bachelor of Science) degree in Computer Science, in 2009, at the University of Windsor. He is currently a candidate for the Master's of Science degree in Computer Science at the University of Windsor and will graduate in the winter of 2012.