Benchmarking of Data Warehouse Maintenance Policies

HS-IDA-MD-00-001

Ola Andersson

Department of Computer Science
University of Skövde, Box 408
S-54128 Skövde
Sweden

Submitted by Ola Andersson to the University of Skövde as a dissertation towards the degree of M.Sc. by examination and dissertation in the Department September 2000.

I certify that all material in this dissertation which is not my own work has been identified and that no material is included for which a degree has already been conferred upon me.

Ola Andersson

Abstract

Many maintenance policies have been proposed for refreshing a warehouse. The difficulties of selecting an appropriate maintenance policy for a specific scenario with specific source characteristics, user requirements etc. has triggered researcher to develop algorithms and cost-models for predicting cost associated with a policy and a scenario. In this dissertation, we develop a benchmarking tool for testing scenarios and retrieve real world data that can be compared against algorithms and cost-models. The approach was to support a broad set of configurations, including the support of source characteristics proposed in [ENG00], to be able to test a diversity set of scenarios.

Table of Contents

1 Introduction

Warehousing is a technique for retrieving and integrating information from distributed, autonomous, possible heterogeneous sources [ZHU95]. A data warehouse is often used as a decision support facility for an organization. The data in a warehouse is typically time-variant, non-volatile, subject oriented and summarized [CON98].

A typical data warehouse involves no or few updates from the users but complex queries [WID95]. There are many reasons to use a data warehouse instead of an operational database, also called OLTP systems ([CHA97]):

- data in the warehouse are of multidimensional type and need specialized data organization,
- using operational systems to answer complex OLAP queries would result in bad performance,
- a data warehouse might need data from heterogeneous sources,
- the information in an operational system may lack information important for decision making.

Information in a data warehouse is stored in materialized views [GUP97], which are redundant collections of data, as opposed to virtual views for which query modification is used to redirect queries directed to the view to the base tables. Because of the redundant data, a warehouse will have to be maintained in order to reflect changes made to its sources [AGR97]. The update of a warehouse to reflect source changes is referred to as a refresh [COL96].

A drawback of the data warehouse solution is the additional storage space required to store the information [LEE99]. One problem is the maintenance of the warehouse [AGR97].

A materialized view may have been derived from several sources. In such a case, data needs to be integrated by an integrator. The responsibility of the integrator is to get the source data into the warehouse [HAM95]. If this is not properly handled, the warehouse might become inconsistent with respect to its base tables. For instance, when an update occurs during a refresh, the view might become inconsistent [ZHU96].

1.1 Research Areas

Data warehousing became of interest to industry in the early 90’s but the research community paid little attention to the area at that time [WID95]. The importance to the commercial sector of data warehousing appears to originate from the need of enterprises to collect all information in one place for in-depth analysis, and to decouple it from its OLTP systems [WID95].

There are several related research problems associated with the warehouse approach. For instance, much work has been done on how data in a warehouse can be kept consistent with its base data. Many strategies have been proposed for how to refresh a warehouse and when to perform a refresh. These strategies are called maintenance policies.

1.2 Project Context

Because of the large set of maintenance policies, the selection of a suitable maintenance policy is often not trivial. This has been well understood by the research community and mathematical algorithms or models have been developed by research groups for use in calculating associated costs and gaining knowledge about maintenance policies. There are algorithms and cost-models that can calculate, for example, the following measures: staleness [HUL96], query response time [HUL96] and how often to refresh a view [SRI88].

These formulas or cost-models can be used by designers to help in the selection of refresh techniques for a specific scenario.

In [ENG00] the authors propose a cost-model which incorporates several user-oriented, as well as system oriented parameters. The cost-model is utilized in an automated selection process which has been implemented as a tool. If correctly configured, the tool can use the cost-model to propose a suitable policy, as well as to estimate the associated relative cost of a policy for a specific scenario. However, the results of the proposals of maintenance policies for given scenarios have not been yet compared with a real implementation.

The aim of this project is to produce a tool to support empirical analysis of the comparative costs of a selection of data warehouse maintenance policies. It is motivated by the need to investigate the cost model proposed by Engström *et al* [ENG00].

Some test cases have been performed to test the developed tool and compare the results between different test cases.

1.3 Outline of Report

Section two gives an introduction to the area and to the background of the project. In section three, the problem addressed by the project is elaborated. In section four, issues related to tool design and of solving the associated instrumentation problems are discussed. In section five, a prototype architecture for, and technical information regarding a benchmarking tool are presented. Experience with the tool is detailed in section six. In the final sections, conclusions and future work are presented.

2 Background

2.1 Maintenance

As stated earlier the information in a warehouse is a redundant collection of data and therefore needs to be maintained in order to reflect changes made to its sources [AGR97], a process referred to as a refresh.

A refresh can be carried out in a number of ways. It is important to propagate updates in a way that avoids inconsistencies in the warehouse while satisfying user and system requirements [COL97]. A maintenance policy describes how and when to refresh a warehouse from a source [CHA97].

The problem of updating a materialized view in a relational DBMS has several similarities with the problem of updating a materialized warehouse view. There exist several dissimilar properties as well. Views in a warehouse tend to be more complicated than in an ordinary DBMS [WID95]. Another important dissimilarity is that warehouse sources are autonomous [ZHU95] and heterogeneous [HAM95].

In commercial systems, a refresh is often performed off-line. Updates are periodic and performed in large batches [SIN97]. This technique has several drawbacks. For instance, the user cannot read data from the warehouse simultaneously with an update, because the data may be inconsistent. Therefore, maintenance is often performed overnight.

Because of the globalization of companies, schedulable maintenance intervals are shrinking; it is always daytime somewhere over the globe. Therefore, alternative techniques must be used to cover this situation. Another issue is that data that is retrieved is old [HAM95], on average a half-day old if the warehouse is refreshed every night. This is acceptable in many scenarios but there exist many situations in which such stale data is not tolerated, for instance in a stock market environment.

2.1.1 Incremental/recompute

There exist two major approaches to refreshing a materialized view: the recompute approach and the incremental approach [LEE99]. A recompute approach throws an entire materialized view away and propagates all tuples required for the materialized view from the source [ZHU98]. Incremental maintenance means that only the part of the view that is affected by source updates is updated [LEE99]. In other words, information in a view that is still current is kept and new information is integrated into it [ZHU98]. The set of tuples that must be propagated for a policy is referred to as a delta change.

Incremental maintenance can be advantageous if an application requires updates to be displayed quickly [ZHU98]. Incremental view maintenance has been analyzed in a number of studies (see, for example, [AGR97] and [LEE99]). There is a general opinion that an incremental strategy outperforms a recompute if the relative size of the change is small [COL97]. If the set of changes to a source between refreshes is comparatively large with respect to the size of the table then a recompute strategy may still be preferable.

There are situations in which a recompute strategy must be used even when in theory an incremental approach will outperform it. In particular, this is true if the warehouse and/or the source do not support the functionality required for incremental refresh.

In the literature, recompute is sometimes referred to as refresh but in this dissertation we will be using the term *recompute*.

2.1.2 Immediate

With an immediate maintenance policy, updates to a view are applied after each transaction [HAN87] or as a part of a transaction [COL96]. In other words, when an update arrives at a source it will also be sent to the warehouse either as a part of the updating transaction or immediately after that transaction. According to Colby [COL97] "*An application has to make a view immediate if it expects a very high query rate and/or real-time response requirements*". A disadvantage of the immediate maintenance policy is that each update incurs an overhead in updating the view [COL96], something that may significantly slow down the source (OLTP) system.

A technique called screening can be used to improve performance. This technique tests every tuple to see whether it affects a view [HAN87]. If a tuple passes the test, then it needs to be propagated to the warehouse; otherwise, the tuple need not be propagated because it will not affect the warehouse view.

The immediate maintenance policy cannot be used if the source does not have knowledge of the data warehouse view [COL96]. Many legacy systems cannot therefore utilize the immediate policy.

2.1.3 On-demand (deferred)

With an on-demand approach (sometimes called deferred) a view will be updated just before data is retrieved using it [HAN87]. In other words, when a warehouse database receives a query it first refreshes the warehouse and then answers the query.

For the on-demand maintenance policy to be effective, there must be a check on whether or not a view must be refreshed in order to answer the incoming query [COL97]. Colby further describes how this can be solved by a single time-stamp. On-demand leads to comparatively faster updates to the source but querying the warehouse is slower compared with an immediate maintenance policy [COL97].

2.1.4 Periodic (snapshot)

The periodic maintenance policy updates the warehouse at regular intervals. A periodic maintenance policy periodically refreshes database snapshots [HAN87]. *"Snapshot maintenance allows fast querying and updates, but queries can read data that is not up-to-date with base tables."* [COL97].

2.2 Scenario Properties

The choice of the most suitable maintenance policy depends upon a number of factors. It is impossible to make a complete analysis of every factor that may influence the choice of maintenance policy. Some of these factors play a significant role in deciding which maintenance policy to use and therefore are analyzed below.

2.2.1 Source characteristics

A source may have different capabilities for supporting the propagation of data to a warehouse. There are important relationships between these capabilities and the staleness of data and response-time for queries against a warehouse [ENG00b].

Zhou *et al.* propose three levels of source activeness: ‘Sufficient activeness’, ‘Restricted activeness’ and ‘No activeness’ [ZHO95]. The first level requires that the source can periodically send delta changes from the last refresh. Restricted activeness means that the source does not have ‘Sufficient activeness’ but can send messages to the integrator, triggered on a physical event at the source. No activeness means that the source has no triggering functionality and a polling approach needs to be adopted.

Engström *et al.* [ENG00b] have proposed three different change detection capabilities that a source may have. These are orthogonal, and therefore a source may have any of the combinations of the three source characteristics. Change aware (CHAW) means that the source can, on request, tell when it was last updated. Delta aware (DAW) means that the source knows which tuples have been affected (delta changes) since the last refresh. Change Active (CHAC) means that the source can report automatically that updates have occurred.

Different maintenance policies require different source characteristics. For example, to use an immediate maintenance policy the source needs to be CHAC. For an incremental approach the source needs to be DAW [ENG00b]. However, if a wrapper with the desired functionality can be utilized then this requirement can be relaxed. For instance, a wrapper can extend non-DAW sources to provide DAW but this will introduce additional costs.

2.2.2 Warehouse properties

A warehouse may have important properties that affect the choice of maintenance policy. The frequency of queries is one of many characteristics a warehouse will have. It is also important to consider the sizes of the materialized views in choosing an appropriate maintenance policy.

2.2.3 User requirements

The selection of maintenance policy is highly dependent on the user of the warehouse. For instance, users will dictate how stale (old) data may be to be tolerated in a query answer, what level of consistency is needed in an answer, and what kind of query response time is acceptable.

2.2.4 System constraints

System constraints are important as well. These may include space restrictions in the source and warehouse, and processing constraints on a refresh.

2.3 Related Work

Developers of data warehouses have a difficult task to select the most suitable policy for a specific scenario because of all the factors that affect the choice. A tool or mathematical calculation that provides the user with the cost of applying a specific maintenance policy for a specific case would be useful.

In [COL96] the paper describes the performance results from an actual implementation of maintenance policies. The tests have been performed on two different databases and similar results have been achieved. The authors state that they learned that: *"Immediate maintenance penalizes update transactions, while deferred maintenance penalizes read transactions"* and *" immediate and deferred maintenance have similar throughput at low updates ratios"* (that is, few updates and many queries) and they showed interesting results concerning when recompute is preferred over incremental update.

Another idea, of protecting a user from seeing inconsistent data without locking tables, has been analyzed in [LAB98] and [QUA97]. Both techniques keep consistent copies of data that the user can access while the warehouse is being refreshed. After the refresh has been performed, all new queries are directed to the new set of data.

The authors of [LAB98] have implemented an algorithm for consistency preservation and conducted an experiment that compared their own algorithm with the algorithm found in [QUA97]. The paper examines the performance of the two different algorithms in various settings. According to the authors, the results from this showed that the proposed algorithm in [LAB98] 'Multi-Version No Locking' (MVNL) has generally less storage overhead than the 'Two Version No Locking' (2VNL) which also has the drawback that the user may receive inconsistent data if user queries span more than an update period. The authors of [LAB98] also claim that their algorithm is easier to adopt than 2VNL.

In [SRI88] the authors have proposed a model for selecting an appropriate maintenance policy depending on the user viewpoint and the system viewpoint. The paper assumes that the user always wants the latest changes (staleness is low) and that the updates to a source are Poisson distributed.

Hanson [HAN87] describes a performance evaluation of views using query modification and immediate and deferred maintenance policies. The author has derived three scenarios, which are examined with a set of maintenance policies. The outcome from this performance study was that the best maintenance policy was dependent on a set of factors. However, the paper details general heuristics concerning how to decide which policy to use and some tricks that can be utilized to obtain better performance results. Hanson concludes that: "*The performances of different maintenance policies are highly dependent on the database and view structure, and the frequency of updates and queries*". [HAN87] further states that future research would be to develop a method to choose an appropriate maintenance policy.

Some other relevant work, in the area of web materialization, has been undertaken by Labrinidis and Roussopoulos in [LAB99]. They have considered where to materialize web views to increase performance. In [LAB99] the authors compare three ways of storing views in a web-server environment, to increase performance. These materializing alternatives are: virtual (use query modification), materialized in the database and materialized for the web-server stored as an HTML document.

The paper suggests a cost-model, which has been tested in practice. The cost-model takes staleness (which in this test requires that there is no stale data at all) and response time into consideration. The storage requirement has been ignored because of the small portion of data that is needed in a usual web server. The general result from this paper is that storing a materialized view at the web-server as an HTML file is often the best way to increase performance.

Even though these results provide information in different areas, similar problems exist.

2.4 A Data Warehouse Maintenance Cost-model

According to [ENG00], previous work in the area has considered individual aspects (for instance user requirements and source characteristics) independently of each other and because of this, there is a need for a model that connects all parts in a comprehensive model. The authors propose a cost-based framework for analyzing different maintenance policies against a set of requirements and source characteristics.

The authors have also derived an algorithm for selecting a maintenance policy depending on the user requirements and the source characteristics. This algorithm is designed with the goal of selecting a maintenance policy that meets user requirements while minimizing cost as indicated by the cost-model.

The formulas and the selection algorithm have been implemented as a prototype. The cost-model prototype can be tuned for various scenarios, taking account of source characteristics and wrapper localization. Wrapper localization indicates whether the wrapping is performed locally or remotely with respect to the source. A local wrapper process resides on the source node. When a source cannot provide DAW, delta extraction can be performed by a local wrapper, thereby minimizing communication costs. This contrasts with remote delta extraction, performed by a wrapper on the warehouse side. If a source is ‘view aware’, it means that the source has a query interface where a single query can return the warehouse view.

The prototype can also be configured for different user requirements and scenarios. Such configurations are general in the way that the unit of measurement for them is flexible as long as the unit is the same in all configurations. In the following table, the letter associated with a configuration parameter is the letter used in the cost-model for that parameter.

Configuration parameter	Explanation
Source size (N)	The size of the source
View predicate selectivity (f)	The part of source size that the materialized view in the warehouse is interested in
Size of each update (L)	Size of each update set of operations
Source update frequency (c)	The frequency of updates to the source
Polling frequency (p)	The frequency of polling for a periodic policy
Warehouse query frequency (q)	Frequency of queries to the warehouse

Table 2-1 – Cost-model parameters

The cost-model can calculate the cost for the following subset of maintenance policies:

Policy	Requirements
Immediate, incremental	Source is Change Active
Immediate, recompute	Source is Change Active
Periodic, incremental	
Periodic, recompute	
On-demand, incremental	
On-demand, recompute	

Table 2-2 – Policy requirements

The associated cost with a maintenance policy contains a weighted sum of a number of measures, summarized in the table below with an informal definition given for each one:

Measure	Explanation
Staleness	How old data is in the retrieved query answer with respect to the source
Response-time	The overhead of a maintenance policy when answering a query
Source processing	The time needed at the source to make the refresh
Warehouse processing	The time needed at the warehouse to make the refresh
Source storage	The storage requirement at the source
Warehouse storage	The storage requirement at the warehouse
Communication	Communication needed between source and warehouse

Table 2-3 - Cost-model measures

According to [HUL96] the response time for a materialized view is the same as the query execution time. One might make the observation that [HUL96] does not consider on-demand policies where the time of refresh is also added before the query can be answered. In [ENG00] the authors use a slightly different definition from Hull *et al.* by which response time does not include query execution time, because the time is equal for any choice of policy and will not affect the selection.

Hull et al. in [HUL96] define ‘worst-case staleness’ as the maximum time of the subtraction of the last time the view retrieved was reflected in the source from the time at which the query was returned. A similar definition of staleness is given in

[ENG00] where it is defined as: *"For a view with guaranteed maximum staleness Z, the result of any query over the view, returned at time t, will reflect all source changes prior to t-Z."*

In other words, the last time the warehouse is up-to-date is just before a new and relevant change occurs to a source. The time when a query answer becomes stale is when such an update occurs which is not incorporated into the query answer.

In [ENG00] both staleness and response-time relate to queries against the warehouse and both consider worst-case execution times. The source and warehouse processing attributes consider the processing for a time unit. To get a per query cost it can be divided by the time between two queries.

The cost-model prototype provides a user-requirement facility that proposes a suitable policy depending on the relative importance of the attributes staleness and response-time, and the scenario.

Output from the cost-model prototype consists of three parts:

1. The most suitable policy depending on the user requirements.
2. The log file of how the proposed policy was selected.
3. A cost-graph for which the user can enable and disable measurement in the summarized total cost.

The cost-model prototype assumes periodic updates, and assumes a single relation in the source that is a select view at the warehouse. Some basic values for calculating the cost have been made static for the first release of the prototype: the block size is 4000 bytes, a tuple is assumed to be 100 bytes large, and block read time is set at 0.01 seconds. It also assumes that the source database size does not change over time.

3 Problem Definition

3.1 Project Aim

The cost-model proposed in [ENG00] has never been tested in practice to see whether it provides correct results for a real scenario. Because it is strictly theoretical and assumes a simplified model of reality, it would be interesting to see how well the results from the cost-model correspond with a real world implementation.

The aim of this project is to produce a tool to support empirical analysis of the comparative costs of a selection of data warehouse maintenance policies. It is motivated by the need to investigate the cost model proposed by Engström *et al* [ENG00].

3.2 Associated Issues

The results from the tests have to be comparable and accurate. However, it is almost impossible to measure anything without introducing the probe effect, which means that the measurement technique affects the measurements. Therefore the benchmarking tool needs to be correct and every part inside and outside of the implementation needs to be under control, something that is impossible because operating systems and the DBMS may behave unpredictably for different reasons.

To test and see whether the developed tool can be useful, a set of test cases was to be run and compared within this project. It is not possible to test some scenarios, because of their nature. One example is a scenario that uses more processing than available; another is a test that spans several months.

3.3 Objectives

This section describes the original objectives of the project in terms of the tool to be built, and the experiments envisaged.

A powerful benchmarking tool was required to enable testing of diverse scenarios with differing maintenance policies. There was also a need to be able to isolate parts that were going to be measured so that any results obtained would be consistent, accurate and comparable.

If the resulting tool is easy to use then anyone could perform experiments which they find interesting, something which is likely to increase the utilization of it. A few experiments were to be conducted to illustrate how the benchmarking tool could be used. Results from these test cases were to be analyzed and discussed.

This project could also be seen as providing a tool that could be used to test a whole set of cost estimation algorithms and cost-models. Even though the options for configuring the tool are based on the chosen cost-model, it is believed that they are quite general and so the tool could be used to test other algorithms and cost-models with no changes to the program code. This tool could be used with any cost-model or algorithm to compare predicted results with actual results from the developed tool.

4 Tool Design

4.1 Measurements

The cost-model can provide estimates for seven different measures. In the table below, these are listed together with an indication of whether the tool measures them.

Estimated by cost-model	**Measured in tool**
Staleness	Yes
Response-time	Yes
Source Processing	Yes
Warehouse Processing	Yes
Source Storage	No
Warehouse Storage	No
Communication	No

Table 4-1 – Cost-model measurement realization

Staleness and response-time are of major importance in the cost-model and need to be measured as they play an important role in the selection of maintenance policy, according to a number of authors who consider staleness/freshness and response-time. The processing attributes are also included because this is easily achievable and these measurements cannot easily be retrieved by other means. All these realized measurements consider time as the unit of measurement.

The tool does not take any other measurements. The storage attribute can be determined easily through the DBMS and communication costs were hard to realize because the environment used did not support retrieving information directly from the network card. Moreover, a careful analysis is required to avoid communication which is not generated by the benchmarking environment. This could for example be communication generated by system processes.

4.2 Tool Functionality

In this section the functionality of the tool is discussed.

4.2.1 Ease of use

To allow anyone to conduct experiments the tool has to be easy to use for both experienced and inexperienced users. A good graphical interface was considered useful to help the user become familiar with the tool, allowing easy configuration for different scenarios.

4.2.2 Portability

As it must be possible to run the tool on a diversity of platforms and connecting to a broad set of DBMS, it had to be realized in a programming language that is platform independent. Java was the obvious choice. Java standard libraries support the efficient development of graphical user interfaces. Also, being object oriented Java offers mechanisms for structured extension of the tool. The connection to the database had to be standardized and allow for a flexible change of DBMS. JDBC (Java Database Connectivity) is designed for Java access and was chosen for the tool. Many DBMS vendors provide JDBC packages for access to their DBMS.

4.2.3 Coverage of cost-model configuration parameters

Ideally, all possible parameters in the cost-model (can be found in table 2-1) should be configurable in the benchmarking tool.

4.2.4 Storing results from experiments

The result of an experiment can be displayed (a diagrammatic representation of results is often considered beneficial) or stored in some way. As a log file provides the ability to later display data in a broad variety of ways this was the choice for this project.

4.2.5 Implementation

The tool has a set of threads: one for simulating the user that asks queries to the warehouse, one for making changes in the source database and one for performing a refresh. In the absence of other evidence, all tasks are considered to have the same importance and every thread in the tool has therefore been given equal priority.

The start time of threads was randomized throughout the period of a task, to prevent different experiments following the same line of execution and delivering results which may be highly distorting, either optimistically or pessimistically.

4.3 Correct Results

Because of the importance of obtaining accurate, consistent data the implementation of each maintenance policy was crucial. If a policy is implemented badly the measurements will not be representative. A detailed analysis phase was needed both to consider design choices for each maintenance policy and to assure a flexible architecture for the tool.

The “probe effect” was minimized by considering longer periods for measurement instead of summarizing several short measurement intervals. At a first glance, it would seem useful to turn off/on different measurements for specific tests. However, as this would require that more operations were needed to see whether each measurement is going to be used or not it would also contribute to increased overhead. The tool therefore does not allow turning off measurements (through the user interface). Instead, the overhead of each measurement has been minimized as far as possible. The design has had to allow for more measurements to be added to the tool.

The instrumentation costs were found to be negligible after an early prototype showed that the basic ‘measurement operation’ (which stores the current time in milliseconds in an array) costs much less than the granularity for each measurement (a millisecond). This granularity level is enough for this project because database operations are in the order of seconds (again shown by a prototype program that performed a sample of database operations). If in the order of microseconds is needed then C/C++ and assembler may be a better choice than using Java for the instrumentation code.

4.4 Experiment Execution

It would probably be useful for a user who performs an experiment to be able to run a set of queries for each test run and as well to perform a number of repetitions of each test case without any user interaction in between. This would result in more representative results from the tool. The tool therefore supports such configuration.

The tool could be made general so that it could accept a large set of underlying database schemas. This could increase the number of tests it is possible to perform. However, since this would have required more effort than was considered viable in developing the tool for this project, a decision was taken to make the implementation specific to a table found in TPC-C[1]. However, an attempt has been made to reduce the effort needed to change the table used in the tool.

Attribute	Data Type	Primary key
Row	Integer	No
Order_ID	Integer	Yes
Order_District_ID	Integer	Yes
Order_Warehouse_ID	Integer	Yes
Order_Customer_ID	Integer	Yes
Order_Entry_D	Timestamp	No
Order_Carrier_ID	Integer	No
Order_Line_CNT	Integer	No
Order_All_LOCAL	Integer	No
Order_Total_Amount	Double	No

Table 4-2 - Order table

Table 4-2 shows the current table used for benchmarking. The size of a tuple in this table is 56 bytes if contained in DB2. The TPC-C specifies all parts except the

[1] Transaction Processing Performance Council offers specifications and description for benchmarking DBMS. More information regarding TPC can be found at: www.tpc.org

‘Order_Total_Amount’ and the ‘Row’ attribute, which have been added for this project. The ‘Row’ attribute contains the number of the transaction which inserted a tuple. The ‘Order_Total_Amount’ contains the total value of an order. It reflects data as being summarized (typical for a data warehouse) and a populated database provides more semantic information than without this attribute.

The start-time of tasks is randomized and this affects when a refresh is performed. For instance, if a periodic policy is used and a refresh is started almost immediately then more refreshes could be performed during an experiment than if the refresh was initially started late. This would affect results concerning staleness and more importantly processing time that, depending on the start time, might include more refreshes.

5 The Implemented Tool

In this section the implemented tool will be in focus. How to access its functionality and how to set up a test case with the user interface is described below. A more detailed study of how the different measurements are calculated and what the benchmarking tool performs is also provided.

5.1 Overview of Architecture

The benchmarking tool consists of a number of parts with different responsibilities. The main parts of the benchmarking tool are:

- The 'Source Updater' that simulates users and applications that update the source.
- The 'Warehouse Querier' that simulates usage of a warehouse, by submitting queries to the warehouse periodically.
- The 'Wrapper', that performs refreshes.

Two versions of the benchmarking tool have been built. This was necessary because the underlying DBMS was switched during the project from Oracle 8i to IBM DB2. This was easy to do since both products provided standardized JDBC drivers to their DBMS. This meant that parts had to be changed to comply with the new DBMS. It would be useful if the two versions could be merged into one, allowing the user to select through the user-interface which DBMS is going to be used.

5.2 Limitations

I have not been able to realize the CHAC source characteristic since I could not find any information that DB2 could effectively support such active participation able to report changes outside the DBMS environment. Triggers are available in DB2 but they cannot be used to send notifications to external processes. DB2 has a replication mechanism but this reacts only on a periodic basis and therefore cannot be used for providing CHAC, which demands immediate notification when changes occur to the source. However, it could be possible to use a replication schema if the periodicity of the replication mechanism is just a small fraction of the inter-arrival time of changes to the source.

Realizing remote wrapping has been ignored because of time limitations. Remote wrapping can, for example, play an important role in reducing communication costs [ENG00]. Nevertheless, there should be little difficulty in adding this type of functionality to the benchmarking tool.

These limitations have reduced the number of possible combinations for the experiments. However, the program can still be useful. Four policies, two source characteristics and a large set of configuration parameters to configure the tool for a specific scenario have been realized.

5.3 Extensibility

The tool has been designed to be easily extensible, both for new measurements as well as adding functionality to the tool, for instance to add functionality to perform batches of experiments.

Consider, for example, adding a communication measurement to the benchmarking tool. A class will be created which handles each measurement and keeps track of them all. When a measurement is reported the method writeToLog(String) will be invoked in the 'Time Recorder' object that writes the String to the log file.
If more elapse time measurements are needed all that is needed is that the part that will be measured can invocate methods of the Time Recorder object.

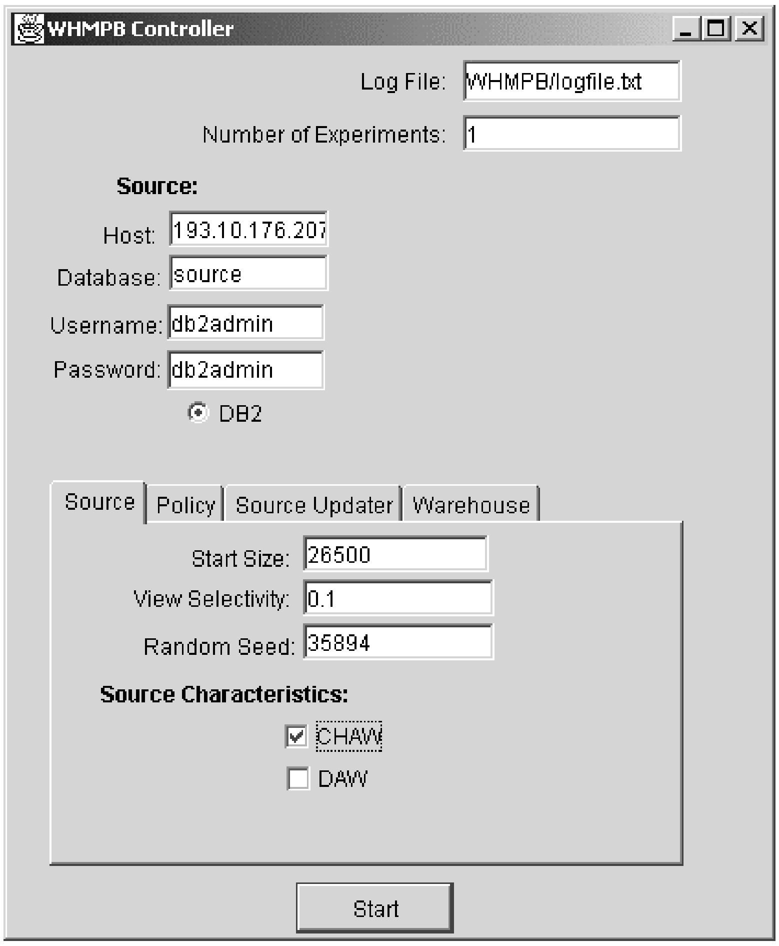
To add instrumentation to a section of code requires five additional method invocations:

- one to initialize the measurement (startNewMeasure),
- one to add the report text associated with the measurement that will be displayed in the log-file (setReportText),
- one to record the start of the measure (startSingleMeasure),
- one to stop measurement (stopSingleMeasure), and
- one to indicate that the measurement will be included in the next set of data written to the log file (setFinished).

5.4 User Interface and Configurations

The configuration and usage of the benchmarking tool to perform tests is described in this section.

Figure 5-1 shows the main user interface. Within this window all configuration parameters of the benchmarking tool are represented. In the right top of the window, the location and name of a log file is specified, in this case 'logfile.txt' located in the WHMPB directory. The result will be stored in plain text, and therefore can be edited with any text editor.


WHMPB Controller

Log File: WHMPB/logfile.txt

Number of Experiments: 1

Source:

Host: 193.10.176.207

Database: source

Username: db2admin

Password: db2admin

[✓ DB2]

Start Size: 26500

View Selectivity: 0.1

Random Seed: 35894

Source Characteristics:

[✓ CHAW]

DAW

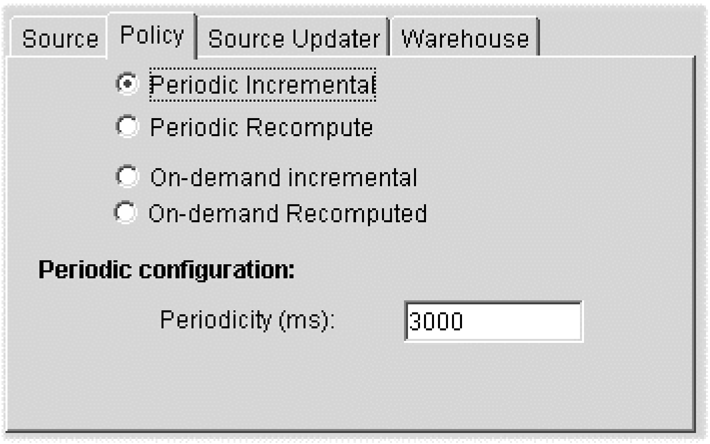
Figure 5-1 – Main window

The field under the log file specifies how many repetitions of the test-run will take place for the same test case. All log data is stored in the same file independently of the number of test-runs.

It was found that storing the warehouse view in a DBMS was costly with respect to processing and disk utilization. Both the source and warehouse for the experiments resided on the same computer (since only one DBMS were in use) and the benchmarking tool was executed on the same computer as the DBMS. We also found out that using JDBC gives a great overhead to recompute views as updates have to be done tuple by tuple. We thought it was beneficial to be able to perform experiments with more concurrent load and therefore a simple main memory warehouse was built. Figure 1 shows the main memory warehouse version. The ordinary DBMS version would require the user to specify warehouse parameters such as database server, database name, username and password. The fields for configuring the source database can be seen illustrated on the 'Source' tab panel. A radio-button is provided to select the DBMS being used (currently only DB2).

All parameter settings for configuring an actual test case are entered in the 'tab panels'. In the source panel, the parameters include how large the initial database will be. The view selectivity field is used to denote the fraction of the source that is of interest to the warehouse (a detailed explanation can be found in [ENG00]). The random seed field is used to denote which seed to use for generating the initial database.

The user can easily enable or disable source characteristics. Two source characteristics have been implemented, Change Active and Delta Aware. If checked this means that the source characteristic is enabled.



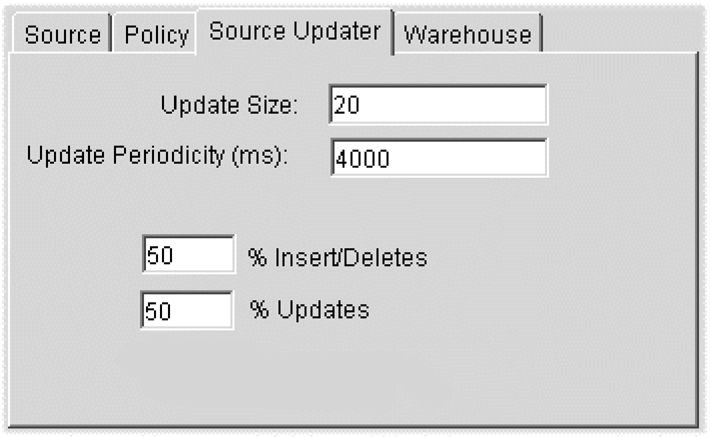
[✓ Periodic Incremental]
Periodic Recompute
On-demand incremental
On-demand Recomputed

Periodic configuration:

Periodicity (ms): 3000

Figure 5-2 – tab panel 'Policy'

The 'Policy' tab panel allows the policy to be configured. This version allows only one policy at a time to be active. If the selection is a periodic policy, the period of the refresh has to be specified in the 'Periodicity' field.

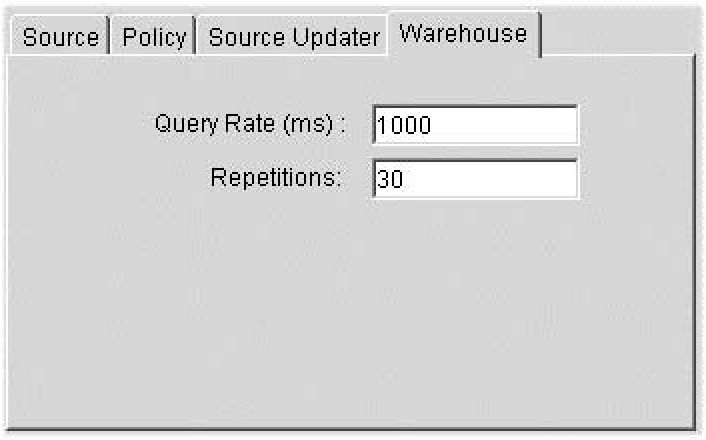


Update Size: 20
Update Periodicity (ms): 4000

50 % Insert/Deletes
50 % Updates

Figure 5-3 – tab panel 'Source Updater'

The ‘Source Updater’ tab panel allows source update behaviour to be configured. The update size determines how many tuples will be updated for each transaction. The ‘Update Periodicity’ determines periodicity of updates to the source. The percentage of inserts and deletes and percentage of the number of tuples to perform needs to be specified (when changing one parameter the other one is changed automatically to add up to 100%). The underlying cost-model assumes that the size of the source never changes. For this reason, an equal number of insertions and deletions are always performed by the tool. In the case illustrated in figure 5-3: 5 (25%) of 20 tuples will be inserted, an equal number deleted and 10 (50%) updated in each update transaction.



Query Rate (ms) : 1000
Repetitions: 30

Figure 5-4 – tab panel ‘Warehouse’

The tab panel ‘Warehouse’ shown in figure 5-4 allows the pattern of queries to the warehouse to be configured. The ‘Query Rate’ specifies with which periodicity queries are posed to the warehouse. The ‘Repetitions’ field determines how many queries are posed to the system for one test-run. These two numbers will in practice determine the length of the experiment.

In the rest of this section, technical issues regarding the development are discussed. Sections 5.5 and 5.6 can be skipped if technical parts are of no interest; it is possible to understand the rest of document even if this is done.

5.5 Source Characteristics

The source characteristics DAW and CHAW have been realized by a set of triggers. To realize the CHAW characteristic three triggers were implemented. A new table was created which contains an id and a timestamp; depending on the database in use the data type for the timestamp may be different. The timestamp object in DB2 supports granularities down to at least millisecond level (which is enough for this prototype).

Each operation that might change the content in the source (insert, delete and update) has a trigger that triggers upon execution of the operation. These triggers perform an update on the table CHAW with the current time as the timestamp.

To check whether a refresh is needed, a query is made against this table to establish when it was last modified. If the time retrieved is newer than the time when the last refresh was made the view needs to be refreshed, otherwise it does not.

The DAW characteristic was realized by three triggers, one for each operation that may modify the content of the source. A new table to store the delta changes was added to the source DBMS (called DAW table). It has exactly the same format as the Order table (see section 4.4) with one exception; it contains a flag indicating whether the tuple is an insert or a delete. If an insert operation is performed on the source, the triggers add the same tuple to the DAW table with the flag set, which indicates that the tuple is an insert to the source. This operation is performed after the insert has been made to the source. If a delete occurs, the old tuple is inserted in the DAW table but without the flag set to indicate a delete. An update operation activates a trigger that inserts the old tuple in the DAW table without the flag set, to indicate a delete, then performs an insertion to the DAW table with the new tuple and the flag set. The net effect of the deletion and the insertion is equal to performing an update.

Retrieval from the delta table when DAW is enabled causes, within the same transaction, all tuples to be retrieved and then deleted. It would be possible to use the DAW capability to simulate CHAW: if the delta table is empty then no operations have been performed in the source.

It is important to understand the level of overhead introduced when using triggers to realize the source characteristics. For this an experiment was set-up that consisted of a single trigger that triggered on insert into a table. This trigger simply inserts the same tuple into another table. A number of tuples were inserted into the base table and the trigger replicated these tuples in another table. This was compared to making two insertions in two different tables without using the trigger.

The results from this benchmark (illustrated in figure 5-5) showed that on average the system is penalized with 37% extra processing when using triggers compared with inserting two tuples in the system.

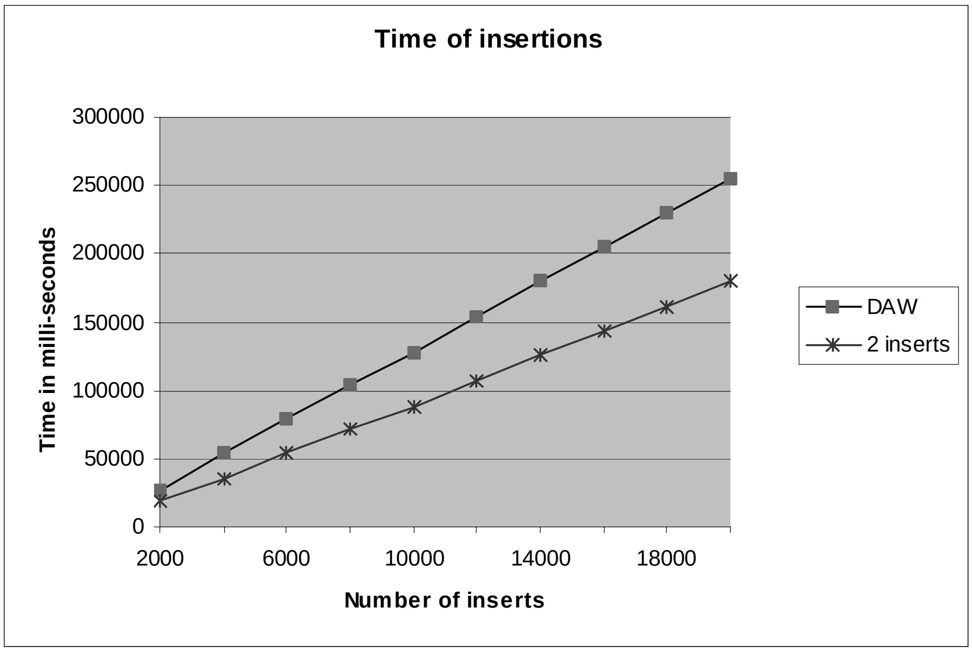


Figure 5-5 - Insertion time

This seems to be a high processing overhead, but it is important to remember that this penalty is not reflected in the set of measures. This is because it is associated with the update of the source database (the updater). It will, however, indirectly affect performance because the source will have a higher load, and processing time for refreshing the warehouse may be harder to get.

To see how much time is spent on making updates to the source, another measurement was introduced to log the time of performing updates for a test case. It gave some information regarding the overhead introduced by making updates to the source. This information could also been used to get a coarse picture if the computer was in an overload situation. During some test cases the updater used more processing time than was needed to refresh the warehouse. This suggests that it is likely to affect the performance and throughput of experiments with a high update load, but may also affect experiments with modest updates.

Parts of the code for the source updater could have been reconstructed, to improve performance. A number of approaches were considered one of which was to use database specific functionality instead of triggers to determine which tuples have been changed since the last refresh. However, in the absence of documentation that DB2 supported such delta extraction this approach was discarded. It would also be possible on update to the base table to insert the associated tuple(s) in the delta table as well and gain performance advantage. Since this would not mirror the chosen source characteristics this approach was not followed.

It was decided to use the trigger approach and after some adjustments to the code had been carried out the time for the updater was reduced by half. It is important that the experiments that are going to be conducted do not overload the system, something which could generate highly pessimistic and unreliable results.

In order to get an equal load independent of whether the source characteristics are used or not, the triggers will always be active. However, the functionality might not be used.

5.6 Program Execution

This section describes in more detail what function each of the different parts of the tool performs. It can be useful to see the architecture and the class responsibilities for the benchmarking tool found in the Appendix.

When the tool is started, the user interface is displayed. The user configures the tool for the actual test case that is going to be performed after which the user hits the ‘Start’ button. The actual configuration is stored in a Config object. Control is transferred to the Sync object, which starts the Time Recorder object, which keep track of all measurements. The Sync object sets the database up by deleting tables, starting triggers and generating base data. After this has been done, the object starts the SUpdater, Wrapper and WQuerier at randomly specified time points (throughout the period of the task).

The WQuerier and SUpdater send messages to the time recorder when certain operations are requested. These measurements are used for the staleness and response-time measurements. The Wrapper object refreshes the warehouse on-demand or periodically, each time measuring the time needed at the source and the time needed at the warehouse; these are summarized throughout the test-run and are the same as the source and warehouse processing. When all queries have been posed to the warehouse a method in Sync is invoked which terminates all processes, writes the log file and checks whether further test-runs are going to be performed. If they are, then it deletes the tables and generates the base data and starts again. If not, then the time-recorder is shut down.

5.6.1 Source updater

Source updates can be divided into two parts: one that generates the initial database that is used in the experiment, the other that periodically updates the source to simulate users and applications that make changes. The initially generated source data can be divided up into two parts: one that is of no interest to the warehouse view (base data) and one that is of interest to the warehouse (view data). Updates during the benchmarking experiments only make changes to tuples in the view data.

The update method works in the following way. The number of updates, inserts and deletes are calculated from the configuration provided by the user.

The actual updates can more easily be explained by imagining three pointers: one insert pointer, one update pointer and one delete pointer. Each points at a tuple that will be affected by the operation. The delete and update pointers identify an already existing tuple; the insert pointer points to a currently non-existing tuple which after the insert operation will be created.

The following operations are performed until the number of updated tuples is equal to the number of updates that are going to be performed:

If there are more insertions left to perform then a tuple is inserted, with the primary keys set to the value of the pointer. If there are more updates to perform then an update of the current tuple is set to a new (randomised) total amount of the order. If there are more deletes to perform the first inserted tuple in the view data is deleted.

For every performed operation the pointer points to the next tuple and the variable containing how many operations have been performed is incremented. If the update pointer goes outside its range, it starts with the first tuple in the view data (the tuple that in the next loop will be deleted).

When all updates have been performed for the current update set, the completion of the transaction is reported to the Source Access (SAccess) with information that the database holds the complete updates from transaction with an id of the value of 'row'. The update is reported to the Time Recorder object that records the time when the update transaction was completed.

5.6.2 Performing a refresh

The source has the ability to retrieve data in three ways:

- Get all data from 'Order' table
- Get all data, sorted on primary key from 'Order' table
- Get delta table from 'DAW' table

Refreshing of the warehouse can be divided into four cases, based on whether refreshes are incremental or recomputed, and what triggers a refresh. The source characteristics potentially make this 2^4 possibilities. However, the implementations of the incremental and recompute approaches are independent of what triggers a refresh (periodic or on-demand) and therefore the cases reduce by a factor of 2. Further, a recompute policy is independent of DAW and therefore the combinations are reduced to six different cases.

Whether the source has the CHAW characteristic makes no difference to the way in which data is retrieved from the source. If the source can provide CHAW and this is used, a check is made to see whether the source has been changed since the last refresh. If no changes have been made then there is no need to refresh the warehouse.

When the source cannot provide DAW and an incremental approach is used it needs to calculate the delta in some other way. I have selected to perform this by retrieving all tuples sorted on the primary key. The delta is extracted by comparing the retrieved data with the one stored in the wrapper (kept in main memory). The cost of performing the delta extraction is O(M+N) (where M is the number of tuples in the new version, and N is the number of tuples in the old version). To illustrate the operations of performing refresh, three examples are considered.

The first (figure 5-6) describes the operation for refresh when both CHAW and DAW are enabled.

```
source.connect()
Check CHAW to see if it needs to refresh {
    Get last id of the last update transaction
    TimeRecorder.startMeasure(sourceProcessing)
    table=source.getDAW() // Get delta table from Source Access
    /*  getDAW() performs:
        Source Access: Get all tuples in the delta table
        Source Access: Remove the delta table
        Source Access: Commit
        Source Access: Return results
    */
    TimeRecorder.stopMeasure(sourceProcessing)
    TimeRecorder.startMeasure(warehouseProcessing)
    warehouse.addDAW(table) //Insert the tuples into the warehouse.
    warehouse.commit()
    Set last update number to the warehouse
    TimeRecorder.stopMeasure(warehouseProcessing)
}
source.disconnect()
```

Figure 5-6 - Incremental refresh when both CHAW and DAW are enabled

The second example in figure 5-7 describes the operation for a recompute when the source cannot provide CHAW.

```
source.connect()
Get last id of the last update transaction
TimeRecorder.startMeasure(sourceProcessing)
table=source.getTable()// Get table from Source Access
TimeRecorder.stopMeasure(sourceProcessing)
source.disconnect()
TimeRecorder.startMeasure(warehouseProcessing)
warehouse.recompute(table) //Insert the tuples into the warehouse.
warehouse.commit()
Set last update number to the warehouse
TimeRecorder.stopMeasure(warehouseProcessing)
```

Figure 5-7 - Recompute not CHAW

The final example (figure 5-8) shows the incremental approach when the source can provide neither CHAW nor DAW.

```
source.connect()
Get last id of the last update transaction
TimeRecorder.startMeasure(sourceProcessing)
// Do delta extraction without DAW enabled.

        t=Get sorted table
        newTable = new Table()
        o= oldTable.next() // One tuple in from the old table
        n= t.next() // One tuple in the new table
        while not browsed through either t or oldtable {
                if (o < n)              // Not in new table, therefore delete it.
                        …
                if (o > n)              // Not in old table , therefore insert it.
                …
                if (o == n)
                        if (o.equal(n)) …
                                // Exists both in old and new table. Do nothing
                        else    // Exists in both tables but an update have changed
                        ...     // some parts in the tuple, delete the old value and make
                                // an insert on the new value.
        }
        // The rest of the tuples in the oldTable is marked as deletions in the delta table
        // the rest of the tuples in the t table is marked as insertions in the delta table.

TimeRecorder.stopMeasure(sourceProcessing)
source.disconnect()
TimeRecorder.startMeasure(warehouseProcessing)
warehouse.addDAW(newTable) //Insert the tuples into the warehouse.
warehouse.commit()
Set last update number to the warehouse
TimeRecorder.stopMeasure(warehouseProcessing)
```

Figure 5-8 - Incremental not DAW, not CHAW

5.6.3 Posing warehouse queries

The returned answer to an ordinary DBMS question is of no interest in the benchmarking other than to see how stale the data is that is retrieved. We therefore simulate queries to the warehouse instead and retrieving information about which update transactions have been incorporated into the answer. This means that the benchmarking tool does not pose database queries; instead it notifies the Warehouse Access object (WAccess, that handles the warehouse database access) that a query is being posed. WAccess sends a notification to the Wrapper object that a query has been posed. Depending on the policy either the wrapper hands control back to WAccess (periodic policy) or triggers a refresh and control is not returned until the complete refresh has been performed. When WAccess gets control back, it passes back the last update transaction being reflected in the warehouse to the Warehouse Querier. This notifies the Time Recorder object about the last update transaction to be incorporated into the warehouse.

5.6.4 Time Recorder

The Time Recorder object keeps track of all measurements that are included in the log file. The time recorder stores all measurements in an array during an experiment and then after the experiment has finalized the data is written to a log file. Three types of measurement are supported.

- Staleness and response-time measurements use the following methods:
    - Supdate(transactionId), invoked by SUpdater to tell when an update has been incorporated in the database.
    - queryStart(Id), triggered when a query starts executing.

    - queryStop(query id, lastid) which is invoked by the Warehouse Querier.

- Single measures (elapsed time measurements)
    - startNewMeasure() start a new ‘Single Measure’
    - startSingleMeasure(measurementId) starts measurement of the specified measurementId
    - stopSingleMeasure(measurementId) stops the measurement
    - setReportText(measurementId, logfileText) sets the text, apparent in the log file, associated with the measurement
    - setFinished(measurementId, boolean) tells the Time Recorder object to record the measurement in the log file next time it writes information to the log file.

5.7 Measurements

The measuring part of the tool uses the Java built-in function `currentTimeMillis()` that gets the number of milliseconds since midnight on January 1, 1970 UTC. See “Java 2 Platform API Specification” for more information regarding this method, which can be found at:
http://java.sun.com/products/jdk/1.2/docs/api/java/lang/System.html.

5.7.1 Response time

In the implementation, the response-time measurement is realized by making each query to the warehouse send a message to the TimeRecorder object that records the start-time of the query. When the data is available in the warehouse, another message to the TimeRecorder object is sent; this stores the stop time for that query. Response time is simply calculated as the difference between these times:

$$\mathrm{ResponseTime} = \mathrm{QueryStopTime} - \mathrm{QueryStartTime}$$

An illustration of this measurement is displayed in section 5.7.3.

5.7.2 Staleness

The latest time the warehouse is up-to-date is just before a new change occurs to the source. The time when a query answer becomes stale is when a new update occurs at the source that is not in the warehouse (recall that all updates are to be in the view data).

To measure staleness, each query stores a timestamp recording when the query answer was returned. All timestamps of updates to the source are recorded in a table with an incrementally growing id. For each query posed to the warehouse the last id in the answer is stored in an array. The staleness measurement is calculated as:

$$\mathrm{Staleness} = \mathrm{Querytable.StopTime} - \mathrm{SourcetableTime}[\mathrm{lastid} + 1]$$

An illustration of this measurement is displayed in section 5.7.3.

5.7.3 Example

Here we present an example of how the response-time and the staleness measurements are calculated. Every query posed gives one row in the table:

Query ID	Last ID	Query Start Time	Query Stop Time
1	1	2000-09-02 23.50.01.84300	2000-09-02 23.51.23.73000
2	1	2000-09-03 00.01.52.93000	2000-09-03 00.02.33.18000
3	2	2000-09-03 01.32.48.21000	2000-09-03 01.33.16.44000

Table 5-1 - Query information

Table 5-1 contains the 'Query ID', which is an incrementally increased value for each query. 'Last ID' denotes the last complete update in the query answer. The 'Query Start Time' contains the time at which the query was posed and the 'Query Stop Time' denotes the time at which the query answer was returned. In the tool all times are represented in milliseconds, but to ease the explanation these are converted to ordinary date and time format.

Each update transaction to the source is stored as a row in the table. The column 'ID' in table 5-2 denotes the id of the update transaction. The 'Time' column records when an update was performed and became visible in the source database.

ID	Time
1	2000-09-02 23.48.23.43000
2	2000-09-02 23.52.37.93000
…	…

Table 5-2 – Source updating table

The calculation of staleness and response-time is illustrated below.

To calculate the response-time for the second query we get:

> 2000-09-03 00.02.33.18000 – 2000-09-03 00.01.52.93000
>
> = 00-00-00 00.00.40.25000

In other words, a response time for this query of 40 seconds and 250 milliseconds was obtained.

The staleness for the same query is calculated as:

> 2000-09-03 00.02.33.18000 - 2000-09-02 23.52.37.93000
>
> = 00-00-00 00.09.55.25000

This means that the query answer is 9 minutes and 55 seconds stale.

To put it into words, we are taking the time at which the query result was returned (2000-09-03 00.02.33.18000) and subtracting the time when it became stale. This time is the same time as the insertion time of the next update transaction into the source database that is not incorporated into the warehouse view (2000-09-02 23.52.37.93000).

5.7.4 Source and warehouse processing

The processing measurements are connected to a whole experiment instead of being connected to each query like the staleness and response-time measurements. These measurements include the following operations:

Configuration	Source Processing	Warehouse Processing
Incremental DAW disabled	Do delta extraction Retrieve all tuples Extract changes	Insert delta in warehouse
Incremental DAW enabled	Get delta table Retrieve all delta tuples Delete all delta tuples	Insert delta in warehouse
Recompute	Get source Table Get all tuples	Insert all tuples into warehouse

Table 5-3 – Parts included in processing measurement

6 Tool Experience

In this section, some experience with the tool is described, and the results from the analysis of some benchmarking scenarios presented.

6.1 Test Cases

The following configurations are used in the experiments. The source initial size is set to 40,000 tuples (each tuple is 56 bytes) and 5 percent of the table (2000 tuples) is of interest to the warehouse.

The source is updated with a periodicity of 43.5 seconds and each update considers 100 tuples. This corresponds to about 66,200 tuples updated during an eight-hour working day or about 2.3 tuples updated each second.

The periodicity at which the warehouse is refreshed if a periodic policy is used is set to 32 seconds. The queries are posed with a periodicity of 25 seconds. Each test case poses 7 questions to the warehouse, is repeated 5 times and the average result from all test-cases is calculated and used throughout this section.

DB2 automatically built an index on the table’s primary key, and therefore the experiments have been executed with an index at the source. All tests below use a main memory warehouse. The source is modified with only inserts and deletes, and source characteristics have been varied between test cases.

The first test case considers the scenario above when the source can both provide CHAW and DAW. The staleness measurement is of interest for this test case. The result from this test case is shown in table 6-1, with measurements in milliseconds. The table also contains the ranking of the policy compared to the others (in order of staleness) printed within parentheses.

Experiment 1, CHAW and DAW	Staleness
Periodic Incremental	2680 (3)
Periodic Recompute	5978 (4)
On-demand Incremental	0 (1)
On-demand Recompute	0 (1)

Table 6-1 – Experiment 1

The on-demand policies have the lowest staleness, probably because the refresh is performed synchronously with queries to the warehouse. This can also explain why there is no difference between on-demand incremental and recompute. The periodic incremental policy has lower staleness than recompute. Intuitively it may be because the source can provide delta awareness and only the delta changes have to be propagated to the warehouse.

When performing the same experiment without DAW, staleness for the periodic incremental policy increases while that for periodic recompute decreases, shown in table 6-2.

Experiment 2, CHAW	Staleness
Periodic Incremental	6162 (4)
Periodic Recompute	2367 (3)
On-demand Incremental	0 (1)
On-demand Recompute	0 (1)

Table 6-2 – Experiment 2

The periodic incremental policy increases the staleness more than twofold compared to when the source could provide DAW. However, the test gave unexpected results for the periodic recompute policy, which performs the same operations independent of the DAW source characteristics. An analysis of the reason behind this can be found in section 6.2. Basically, it is possible that the difference is due to very large experimental error and that makes staleness vary even when executed code is the same; this requires further investigation. Compared with the first test case the periodic incremental policy increases staleness when the source cannot provide DAW. If experimental error is found not to be the cause then one possible reason is that the delta cannot be retrieved from the source, and to be able to perform an incremental policy the delta needs to be calculated. In this case that means retrieving the whole view at the source and filtering out the differences since the last refresh.

The second test case considered source processing against source characteristics. The results from the test cases when the source supports CHAW but not DAW are shown in table 6-3.

Experiment 3, CHAW	**Source Processing**
Periodic Incremental	5988 (3)
Periodic Recompute	5465 (2)
On-demand Incremental	5250 (1)
On-demand Recompute	6044 (4)

Table 6-3 – Experiment 3

The differences between the results are not significant. This may again be due to experimental error, however the following observations can be made. The periodic policies include fewer refreshes (because it has a higher inter-arrival frequency than that by which an on-demand policy refreshes the warehouse). A periodic recompute might therefore demand less source processing time than on-demand recompute. But as the refresh is performed more seldom the changes will be greater. All together, therefore, source processing time might be expected to be approximately the same for both the on-demand and periodic policies.

The results of the source processing measurement for a test case which had neither DAW nor CHAW are shown in table 6-4.

Experiment 4	Source Processing
Periodic Incremental	12885 (1)
Periodic Recompute	13563 (2)
On-demand Incremental	18295 (4)
On-demand Recompute	14632 (3)

Table 6-4 – Experiment 4

The source processing needed is increased significantly for every policy. This is as expected, because the refresh has to be performed every single time the source is accessed because of the absence of CHAW. The periodic policies have the lowest source processing probably resulting from the fact that the refresh is performed more seldom than queries to the warehouse (recall that the periodicity of queries is set to 25 seconds and the periodicity of periodic policies at 32 seconds). The periodic incremental approach should require higher processing than the recompute version. The difference between them is small (5 %) and therefore the ordering may in practice not be significant.

6.2 Limitations of Tool and Environment

All results are dependent on several factors outside of our control. For instance, the operating system and the DBMS cannot be controlled. This could significantly affect the test results.

The following factors have been identified as potential error sources:

- Indeterminism of DBMS and operating system:
    - System tasks

    - Context switches
    - Other services running
- Benchmarking tool is incorrect
- Java interpreter
    - Indeterminism of execution
- Communication channel between tool and DBMS
- Environment
    - Single processor for tool
    - Limited memory, processing power
    - Swapping

6.2.1 Indeterminism of DBMS and Operating System

System related tasks could burden the system significantly. Depending on when context switches are performed the results might vary. Other services running in parallel could burden the system to a varying degree, resulting in the benchmarking tool getting variable processing capacity. DB2 automatically built an index in the source, which could affect the result considerably, especially when performing many insertions and deletions.

Although we could not remove all indeterminism, we have tried to reduce it as much as possible, for instance by performing tests over a longer time to reduce the effects of system tasks. However, as the measuring time increases more system related operations are performed during the execution time. Counter to this, as longer times are considered the system tasks will hopefully be equal in all tests, if the system behaves independently of test-cases and test-runs. Therefore, no other user activities were performed during test runs. The system ran the same background tasks for all test runs.

6.2.2 Incorrect tool

There might be errors in the tool developed, and no method had been used to prove the correctness of the tool. Therefore a number of faults might exist in the tool. The measurement part could include faults that make the time measurement not trustworthy. Some part might be incorrectly programmed or programmed in such a way as to imply that results from a set of test cases could not be compared against each other.

However, every effort has been made to check the appropriateness of each design decision.

During the tests it was shown that using an incremental approach when the source could provide DAW often turned out highly pessimistic. It seems that there are parts in retrieving the delta table that take a considerable time. It could be the penalty of deleting the delta table; since this locks the table, no other insertions to the source can occur and therefore throughput is reduced. When comparing with recompute and incremental approaches without DAW source characteristics, the penalty was often apparent.

Another issue concerns using an incremental approach without DAW, when the retrieved source data is sorted on primary key. In these experiments it can be argued that this had little affect on the result because new tuples were inserted ordered on increasing primary key[2]. When performing updates of tuples in the source, this can burden the retrieval time of the sorted data, since the data is not already sorted (as in the case with only insertions and deletions).

In a scenario with both CHAW and DAW enabled, the benchmarking tool checks the CHAW table before any refresh is performed to see whether there is a need to refresh the view. When DAW is enabled, an empty answer would be retrieved if no changes have been made and therefore a checkup using the CHAW feature is redundant. The CHAW checkup takes some time (not included in the source processing measurement) that may be reduced if the source can provide DAW.

6.2.3 Java interpreter

The Java interpreter uses more processing power than would a compiled and executable program (with no interpreter). This may be apparent in the results.

6.2.4 Communication

The communication channel has an important role. Suppose we use a 10 Mbit Ethernet network with the benchmarking tool and the DBMS residing on a separate subnet with no other computers, connected together through a switch. The cost of sending 5000 tuples in pure communication cost time is about 210 milliseconds. If several nodes exist on the same subnet, the communication cost might even be higher

[2] Something considered a straightforward solution at the time, which could easily be changed later.

due to communication collisions. This is an important aspect when considering communication intense test cases.

6.2.5 Environment

The benchmark tool has been executed on a single processor machine; therefore only one database access at a time is possible. In a real scenario, there might be several processors generating and retrieving data from the warehouse simultaneously. Limited main memory may cause the system to swap main memory parts to disk, which can take considerable time.

During the execution of the benchmark tests, the processing power might not be enough for all threads. This can result in an important thread being unable to execute, and results might be misleading.

7 Conclusions

The aim of this project was to produce a tool to support empirical analysis of the comparative costs of a selection of data warehouse maintenance policies. The background to this project has been described, and its formal aims presented. We have outlined the design of a tool to assist with benchmarking for the purpose of investigating the utility of a given cost model for data warehouse refresh. Provisional experiments with the tool have been reported to demonstrate its use and illustrate issues of interpretation of results.

In section 7.1, we reflect on the success of the project and highlight those problems that have been encountered. In section 7.2, future work is discussed.

7.1 Summary

The developed tool has been found to be useable, and to be easily configurable to let anyone conduct their own experiments. Some parts have been made static because of the cost of making it flexible, for instance the table in use for the project. Some parts have been made flexible to be configured for each test case: source characteristics, policy, database size, periodicity of tasks, are some of the configurations possible with the tool.

The tool can handle a diversity of scenarios, and has been designed to be simply modifiable for testing a broad set of algorithms and other cost-models. A set of tests have been performed some of which are reported in section 6. Before completely trusting the results from the benchmarking tool, more comparisons and benchmarks need to be performed.

Some problems of using triggers to realise the source characteristics have been discussed.

The architecture of the tool allows extensions as well as changing parts easily. For instance, an elapsed time measurement can be added with five lines of code.

The tool neither imitates the real world perfectly nor the cost-model. Somewhere a line has to been drawn for where to imitate the cost-model and where to imitate the real world.

7.2 Future Work

Some desired functionality of the benchmarking tool has not been implemented due to a tight schedule. If the immediate maintenance policy and the change active characteristic could be implemented this would improve the benchmarking tool considerably.

The tool could offer a batch functionality that allows several test runs of different test cases to be performed without any user interaction in-between.

It would also be a useful feature to let users configure the benchmarking tool for the type of schema in use for the benchmarking. This would increase the number of experiments possible. When users have prepared the database with username, password and a database name, the tool would set up all parts according to the user specification and then perform the benchmark tests.

Finally, it would be useful to be able to minimize the overhead needed to realize the given source characteristics, either by using alternative techniques or by minimizing the penalty of using triggers.

Other extensions include Poisson distributed queries and updates.

8 References

[AGR97] D. Agrawal, A. El Abbadi, A. Singh, T. Yurek, “Efficient View Maintenance at Data Warehouses”, SIGMOD, 1997

[CHA97] S. Chaudhuri, U. Dayal, ”An Overview of Data Warehousing and OLAP Technology”, SIGMOD Record 26(1), 1997

[COL96] L.S. Colby, T. Griffin, L. Libkin, I.S. Mumick, H. Trickey, “Algorithms for Deferred View Maintenance”, SIGMOD, 1996

[COL97] L.S. Colby, A. Kawaguchi, D.F. Lieuwen, I.S. Mumick, K. Ross, “Supporting Multiple View Maintenance Policies”, SIGMOD, 1997

[CON98] T. Connolly, C. Begg, “Database systems A Practical Approach to Design, Implementation, and Management”, ADDISON-WESLEY, 1998

[ENG00] H. Engström, C. Chakravarthy, B. Lings, “A Holistic Approach to the Evaluation of Data Warehouse Maintenance Policies”, Technical report, University of Skövde, 2000

[ENG00b] H. Engström, S. Chakravarthy, B. Lings, ”A User-Centric View of Data Warehouse Maintenance Issues”, British National Conference of Databases, 2000

[GUP97] H. Gupta, "Selection of Views to Materialize in a Data Warehouse", International Conference on Database Theory, 1997

[HAM95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge, “The Stanford Data Warehousing Project”, IEEE Data Engineering Bulletin 18(2), 1995

[HAN87] E.N. Hanson , "A Performance Analysis of View Materialization Strategies", SIGMOD Conference, 1987

[HUL96] R. Hull, G. Zhou, “Towards the Study of Performance Trade-offs Between Materialized and Virtual Integrated Views”, VIEWS’96, 1996

[LAB98] A. Labrinidis, N. Roussopoulos “A Performance Evaluation of Online Warehouse Update Algorithms”, Technical Report, Dept of Computer Science, Univ. of Maryland, 1998

[LAB99] A. Labrinidis, N. Roussopoulos “On the Materialization of WebViews”, Preceedings of the ACM SIGMOD Workshop on the Web and Databases, 1999

[LEE99] M. Lee, J. Hammer, "Speeding Up Warehouse Physical Design Using A Randomized Algorithm", Proceedings of the International Workshop on Design and Management of Data Warehouses, 1999

[QUA97] D. Quass, J. Widom, "On-Line Warehouse View Maintenance" SIGMOD Conference, 1997

[SIN97] A. Sinha, “Implementation of the 2VNL Data Warehousing Technique in an Object-Relational DBMS”, Project Report CS395T-Database Mining and Monitoring, University of Texas at Austin, 1997

[SRI88] J. Srivastava, D. Rotem, “Analytical Modeling of Materialized View Maintenance”, ACM, 1988

[WID95] J. Widom, “Research Problems in Data Warehousing”, Conference on Information and Knowledge Management (CIKM), 1995

[ZHO95] G. Zhou, R. Hull, R. King, J.C. Franchitti, "Data Integration and Warehousing Using H2O", IEEE Data Engineering Bulletin 18(2), 1995

[ZHU95] Y. Zhuge, H. Garcia-Molina, J. Hammer, J. Widom, “View Maintenance in a Warehousing Environment”, SIGMOD Conference, 1995

[ZHU96] Y. Zhuge, H. Garcia-Molina, J.L. Wiener, “The Strobe Algorithms for Multi-Source Warehouse Consistency”, PDIS, 1996

[ZHU98] Y. Zhuge, H. Garcia-Molina, “Performance Analysis of WHIPS Incremental Maintenance”, Submitted for publication, 1998

Acknowledgements

I would like to thank the following people for their support during this project. Henrik Engström and Brian Lings for supervising and guide me through this project. I also want to thank Sharma Chakravarthy for his ideas regarding design aspects. All friends especially Malin Andersson without the support from them I would never finished this project. Special thanks to Erik Olsson, Marcus Thuresson and Adam Rehbinder for being kind and supportive.

Appendix

Class diagram of architecture

Here follows the class diagram describing the benchmarking tool architecture. The UML notation is used. Class responsibility is also provided in this section.

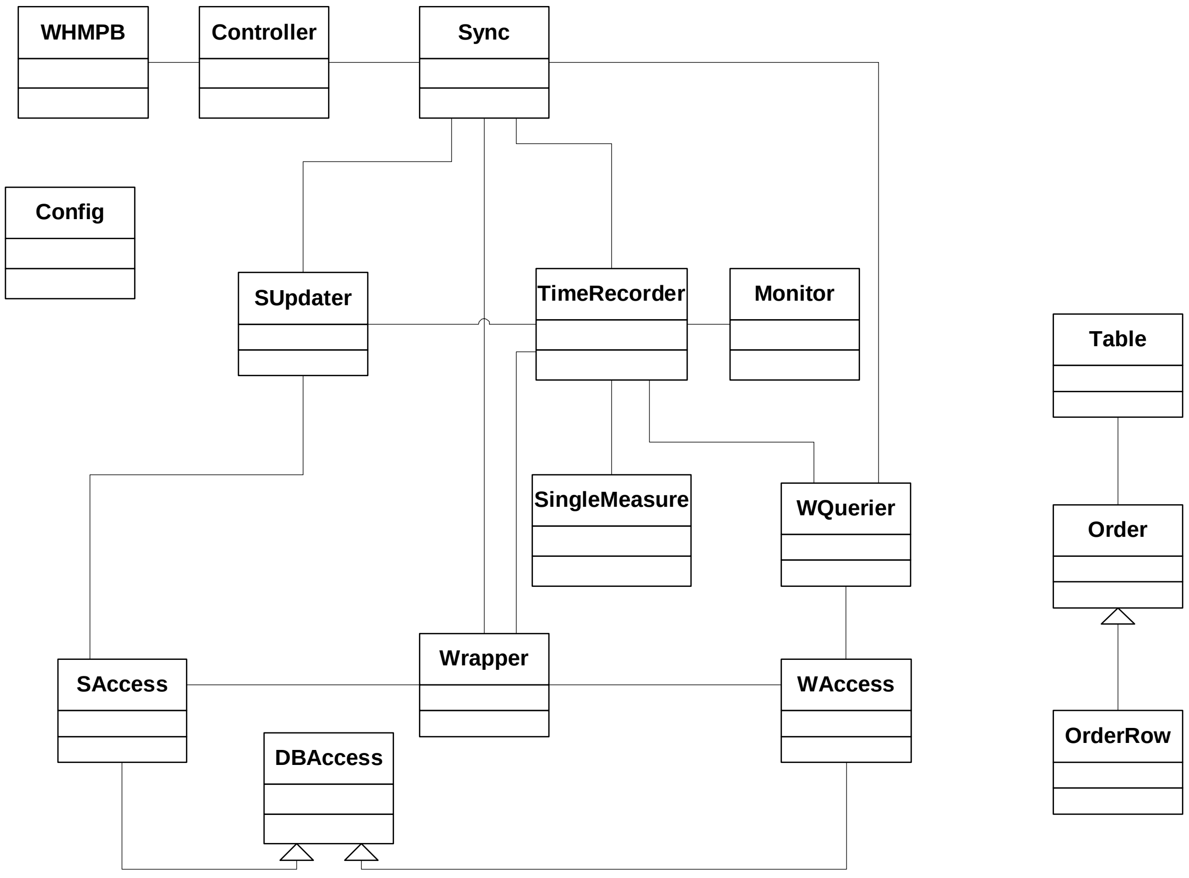


Figure Appendix-1

Class responsibilities

Controller

Presents the GUI to the user, handles the configuration and parameters for the benchmark. Stores the actual configuration in a Config Object.

Config

Store all parameters for an experiment.

DBAccess

General database functionality, connection to database, commit etc. Is a generalization of SAccess and WAccess, Database specific.

Monitor

Handles one log file for the experiments. It contains the structure and arrangement of the data will be written to the log file.

OrderRow

Represent one tuple in the order table. The OrderRow has the ability to convert internal attribute to SQL strings (inserting and deleting) therefore database specific.

Row

A generalization of OrderRow to let a broader set of underlying database schemas be used.

SAccess (Source Access)

Handles the source side DBMS access for performing refreshes. Provides and initializes the source characteristics. Database specific.

SingleMeasure (Single Measure)

Keeps track of simple measurements, which means measures that can be started and stopped and the elapse-time is the result of the measurement. This type of measurement is for instance, the total time of the experiment, the source processing, and warehouse processing measurements.

SUpdater (Source Updater)

Generate the initial data in the source database by sending updates/inserts/deletes to the Source Access to make updates to the source. Database specific.

Sync

Coordinating benchmarking experiments by creating, starting and setup objects for the current experiment.

Table

The Table class contains a set of 'Row'. The class is used to store the deltas or the whole table when making a refresh. When the main memory warehouse is used, an object of this class keeps all tuples.

TimeRecorder

Used to record the time when certain operations are performed. For instance, when queries are asked and when the answer is returned. It also calculates the measurements which is later stored in the Monitor object. Can calculate response-time, staleness, and several Single Measures.

WAccess (Warehouse Access)

Handles all database access to the warehouse. Exists in two versions one to connect to an ordinary database, the other to connect to a main memory database. Database specific.

WHMPB (WareHouse Maintenance Policy Benchmark)

This class starts a controller object. This class is used to start the benchmarking tool.

WQuerier (Warehouse Querier)

This class imitates the user by asking queries to the warehouse. This class records the last tuple in the answer and notifies the Time Recorder object when queries are asked and when they are answered.

Wrapper

Perform refreshes of a warehouse. Support both incremental and recompute approaches. Either the triggering of refresh is periodic or on-demand.