UNIVERSITA' DEGLI STUDI DI MODENA E REGGIO EMILIA

Facoltà di Ingegneria – Sede di Modena
Corso di Laurea in Ingegneria Informatica

A BENCHMARKING ENVIRONMENT FOR VALIDATING A DATA WAREHOUSE MAINTENANCE COST-MODEL

Thesis of:
Gelati Gionata

1999-2000

TABLE OF CONTENTS

CHAPTER ONE

A USER-ORIENTED COST-MODEL

The purpose of this work is to develop a benchmarking tool which can be used to test the correctness of a published cost-model for data warehouse maintenance [ENG00].

A cost-model is a cost-based framework for estimating the performance of a system under given parameter settings. In order to give comprehensiveness and cohesion to the approach, a cost-model must define in a rigorous way both the metrics for assessing performance and the parameters used to model the environment which the system is to act in.

For a well-established cost-model is not enough to claim the soundness of its results based only on theoretical analysis. It is also necessary to consolidate the theory by benchmarking a real instance of the environment assumed by the cost-model.

Generally, benchmarking entails observing the behaviour of a real system under some definite conditions and measuring its performance by means of a specific metric. Therefore, a first step is to construct a system with architecture as assumed in the cost-model. A second step is to emulate the dynamics of the target system, and instrument it to allow measurement of the required metric.

Once the benchmark environment had been prepared and verified, we chose a set of significant scenarios to be tested for comparison with the behaviour predicted by the cost-model. Naturally, we have investigated the scenarios that have appeared to be more interesting as our aim has been to illustrate how the system developed can be used to analyse the cost-model itself.

In this dissertation, we describe the cost-model and show our contributions in developing the benchmarking tool. The tool architecture took as its starting point an initial prototype system developed as part of a Master's degree at the university of Skövde, Sweden [AND00].

Firstly, we introduce the general concepts related to data warehouse (DW) in order to shape its maintenance problem; for a more detailed explanation we refer the reader to Appendix-A. From this starting point, we focus on the main concepts of the cost-model and its implications, avoiding detailed formulas. At the end of this section, however, we present a table [ENG00] which summarises the details of the cost-model considered.

1.1 DATA WAREHOUSING AND THE PROBLEM OF MAINTENANCE

First of all, we need to clarify what a DW is. The diagram proposed in Figure 1 gives a first idea:

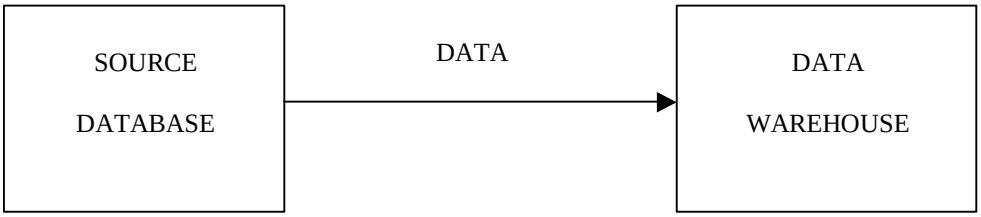


Fig. 1. A DW based on one source database

The figure shows a DW that collects data coming from a source database.

In general a DW collects information from many source databases. This is a typical system perspective of what a DW is. For the sake of completeness, we have to take into consideration a user perspective. Indeed, a DW is far from being a database which simply mirrors the data of its sources. It processes data in order to provide elaborated views for activities like on-line analytical querying and decision making. An elaborated view can be of different natures. It can be a clean or summarised version of a table or view coming from the source as well as an integrated view originating from different tables from different sources. Naturally, we can build new views starting from the views of the DW itself, just like any other database.

These features make the evaluation of more articulate queries possible and afford an analysis of historical trends. There are evident advantages in data availability and in response time for OLAP queries, too. Furthermore, since the queries are submitted to the central system, the sources suffer a smaller stress, being not directly involved in the process.

Figure 2 illustrates a richer diagram:

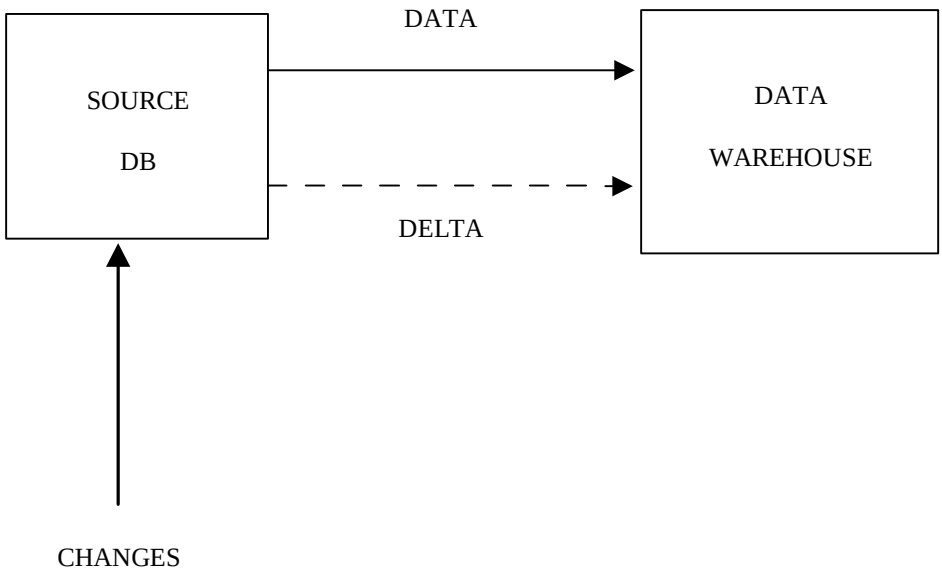


Fig. 2. Changes to the source database are propagated as deltas to the DW

In general, a source database undergoes changes during its life. A database user can perform some actions over the tables that modify their contents (examples are insert, delete, update). These changes make up the database maintenance.

At the same time, they have to be reflected to the DW side to preserve coherence between the information stored in the sources and the knowledge actually stored in the DW. The sketched line in the diagram underlines that changes have to be propagated from source to DW, since the latter cannot be directly updated by its users. This constitutes DW maintenance. The word "delta" in the diagram refers to the insert and deletes actions needed to refresh the contents of materialised views. Remember that even updates can be seen as a succession of insert/delete actions.

Deciding when and how a virtual view has to be refreshed to keep consistency with its sources is a matter of which policy we decide to take on. A policy covers the kind of behaviour that has to be carried out for handling changes. Choosing a policy is a rather complex process, involving a number of parameters related to the environment the DW operates in. We can group these properties in two dimensions: user requirements and source characteristics. We can enrich the diagram as shown in Figure 3:

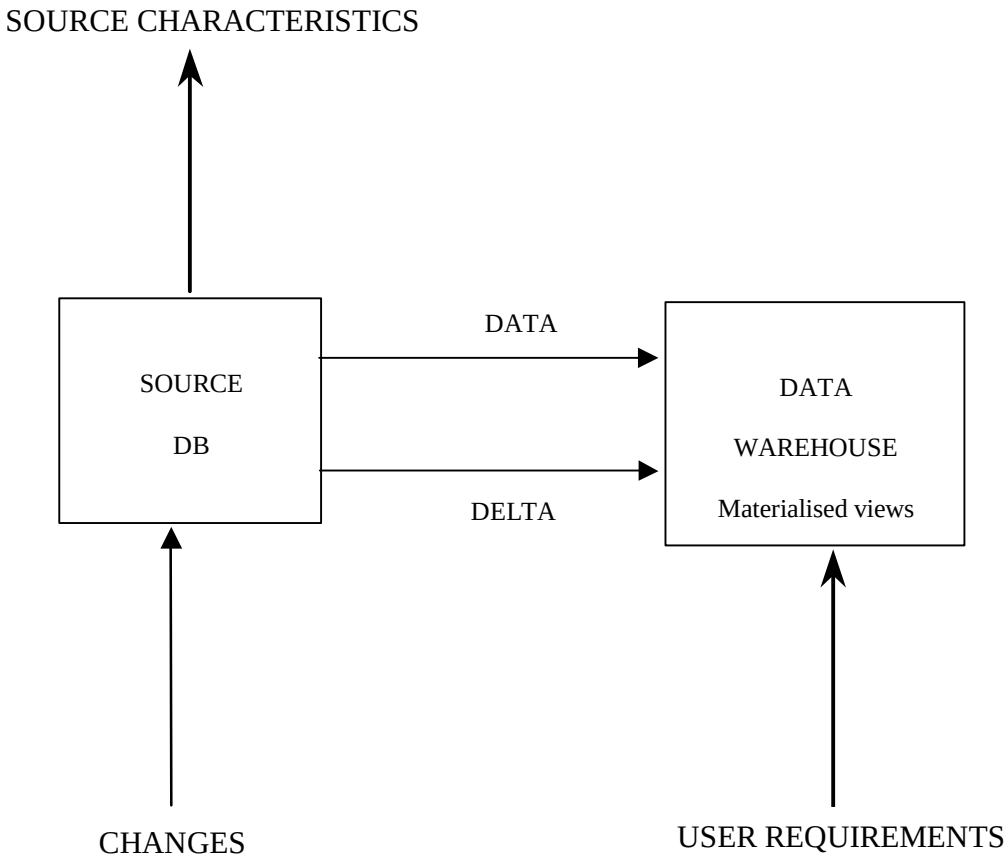


Fig. 3. Role of source characteristics and user requirements

While source characteristics express internal properties of the system, user requirements are external constraints. The latter define the quality of service (QoS) requirements.

The cost-model is fully grounded on this scenario. As a matter of fact, the need for a cost-model is born out of the DW maintenance problem. The aim is indeed to build a selection process which gives a minimal cost policy (not necessary a single one) against user QoS requirements and source characteristics. Six general policies are going to compete in the selection process. These are the result of combining three timings (immediate, periodic and on-demand) with two types of maintenance (incremental and recompute).

In the next sections we will discuss the fundamental concepts related to QoS and source characteristics.

1.2 A USER-ORIENTED VIEW OF THE COMPONENTS

In the previous section we have defined the data warehouse maintenance problem. Following a user-oriented view, the purpose is to find a set of maintenance policies that fulfil user requirements taking into consideration source characteristics.

We need to spell out the user-oriented QoS requirements and the capabilities of sources.

1.2.1 QUALITY OF SERVICE

Approaching the data warehouse maintenance problem, the authors of the cost-model have assumed a new outlook based on a user-centric view. This has allowed them to analyse existing work in a new light, since their perspective is rather original in the warehouse literature. This original angle is the foundation of their cost-model.

To discuss QoS in a systematic way we need to define measures. The authors move from measures proposed in previous work. In the literature we can find several measures related to a user-centric point of view: consistency, availability, currency, freshness, staleness. All the measures are time-based, time being the most evident and objective factor the user has to deal with. This is obviously true under the assumption that the returned data reflect a valid state of the view, i.e. data have meaning and coherence.

Engstroem et al. have pointed out that these ideas have been yet taken into account in previous research as [SEG90] and [HUL96], but they have been limited in their ability to express the real level of service perceived by the user. A precise examination is shown in [ENG00].

As outcome the authors refine the definition of view age and staleness. While the former focuses its attention on view-state and for this reason is system-oriented, the latter is an index of quality and suggests a user-oriented idea. Therefore, in this work we will deal only with staleness.

Staleness

Before introducing a formal statement, it is useful to look at the diagram of Figure 4:

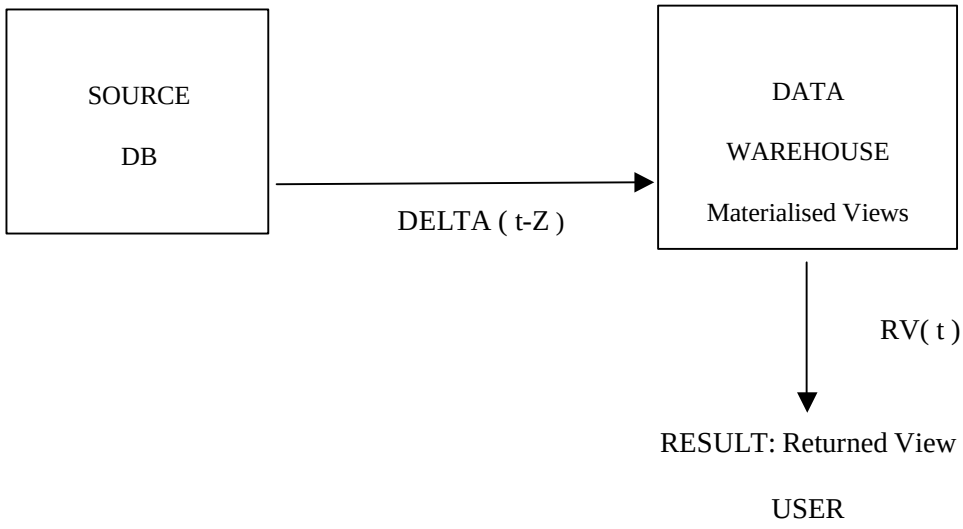


Fig. 4. A DW system diagram that shows delta changes and the returned view with their time information

The scheme is very similar to the previous ones. We have added a new trait: time. RV(t) means that the user has received at time t the view he has queried for. Delta (t-Z) means that the source has propagated to the DW the updates undergone till time t-Z, where Z is the current value for the maximum staleness for the returned view. Therefore the concept of staleness has some peculiarities. In particular:

1) it is a time-based measure, as we have stated in the previous section with regard to measures concerning quality of service;

2) it is related to a specific view;

3) it is independent from the kind of query performed, i.e. the maximum staleness is an upper bound for any performed query on that view;

4) from point 3) we can deduce that the view has any value of staleness of kind Z+d where d>0.

Through knowledge of staleness the user can appreciate the degree of coherence between source and data warehouse, i.e. how old the data reflected inside the returned view are. This implies that staleness is directly usable by the user to weigh up the quality of data he receives. It is a consequence of, and at the same time a confirmation of the user-centric approach adopted by the authors of the cost-model.

Figure 4 illustrates only partially a possible real system. Usually, the view integrates data coming from more than one source. Hence there is a need to widen the concept of staleness as shown in Figure 5:

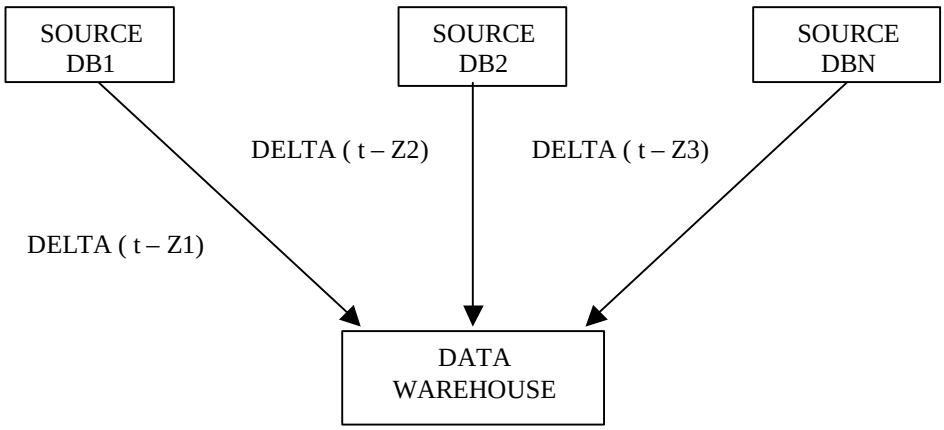


Fig. 5. A DW based on more sources with time information for delta changes: t is the current time, Zi is the maximum staleness of the source database DBi

In general, sources can have different values for maximum staleness. In this case, Z = max (Z1, Z2,..., Zn) represents the staleness. Z is the correct value if and only if, for any given query posed on a specific view, the DW reflects the changes undergone by each source the view is based on, prior to t-Z, where t is still the time when the result is returned to the user. That is to say the maximum staleness Z has to be guaranteed by every single source which contributes to a materialisation of the virtual view of the DW.

For a more rigorous definition of staleness we quote the definition proposed by [ENG00]:

' A view V is based on sources S(1),S(2),....,S(m), where m>=1. The state transitions of source S(i) occur on occasions stored in the vector T(i).The source is initialised at T0(i) and the nth state transition of S(i) occurs at Tn(i). For all n>=1 we have: Tn-1(i) < Tn(i) < Tn+1(i).Our weak consistency assumption implies the each state in the vector is a valid source state.A query over the view returned at t ret will be based on source states entered at Tq1(1), Tq2(2),...Tqm(m), where q1,q2,...,qm are states(represented by the integers greater than zero).Definition: A view guarantees maximum staleness Z iff for all query invocations over the view and all sources S(i) the following holds: Tqi+1(i) > t ret - Z. '

The most evident property of staleness is that its definition draws in neither the time a query is propounded nor the view state. The authors have been concerned about how old the data reflected in the view are and the time the user receives the result. No mention has been given of the time the user submits the query or of the last known instant at which a view reflected its sources (state of the view).

Furthermore, staleness is applicable to all kinds of policy. In particular, it is without restriction towards immediate, periodic and on-demand policies which will be of interest in this work. Seeing we are allowed to apply the same definition to different situations, it follows that values of different benchmarks can be compared and judged.

All these features together give a precise indication of the nature of the extended definition of staleness, attesting to its usefulness and objectiveness.

Response Time

A second measure used for QoS is response time (RT).

The usual definition of RT can be applied to the field of databases: it is conveyed as the difference between the time a result of a query is returned and the time the query was submitted.

The cost-model uses both staleness and RT to evaluate the level of QoS. This is due to the fact that the two measures are orthogonal and, from a user point of view, complementary.

First, they are orthogonal. Staleness must not be seen as a part of RT. As a matter of fact, while RT evaluates the delay expected in obtaining a service, staleness measures how up-to-date the data of the DW are.

Secondly, the measures are complementary: both measures are necessary to the user in order to understand the level of service. In more practical terms, RT indicates how much time the user has to wait between submitting a query and obtaining the result on his screen, and staleness indicates how fresh the data on screen are.

Figure 6(a) and figure 6(b) lay emphasis on the nature of the two measures:

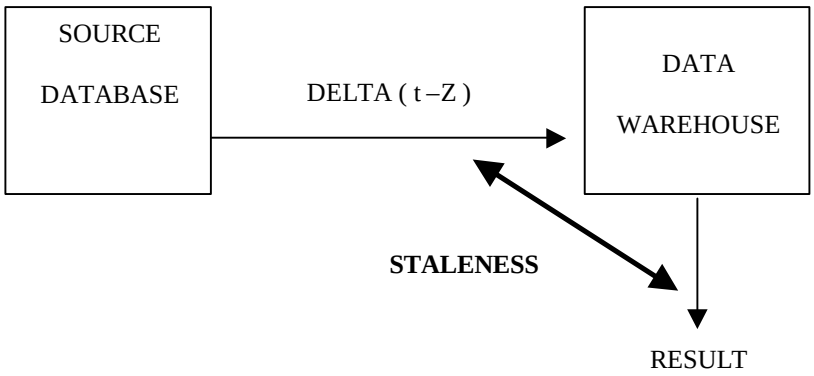


Fig. 6(a). Starting and ending moments to compute staleness following Engström's definition

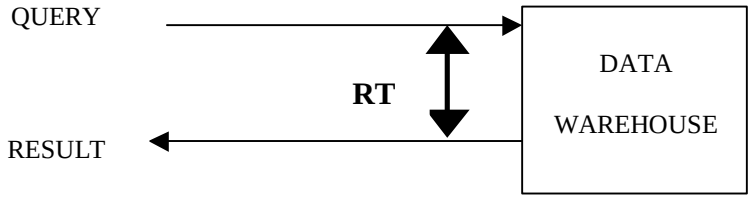


Fig. 6(b). Starting and ending moments to compute RT applying the general definition to our system

Guidelines for the level of requirements

At the moment we are in possession of two concepts, staleness and RT. They are suitable both for calculating system performance and for specifying the required level of service.

The latter use gives the user an active role to play. He is expected to supply two values to set the standard of service he desires. This choice could be rather difficult. If we suppose the user is a DW designer for instance, it is not viable to ask him to specify times for a system he has no experience with. In particular, he has no means to judge whether the values he has provided are optimistic, pessimistic or in some way realistic.

To give a guide to the user, the authors follow, once again, a user-centric approach. Assuming this point of view entails not only extending definitions like staleness and RT but also providing the user with an intuitive and productive interface to the cost-model. The change of perspective consists in granting more relevance to the level of optimisation the user desires for a certain measure rather than to its actual value.

Understanding how much importance is attached to a measure for an application brings the user to a higher level of abstraction, where qualitative remarks hide the mere presence of figures.

It follows that the user is now expected to give qualitative values which express the relative importance of his requirements. The authors have compiled a list of possible value for different levels: **ignore, consider, best possible**. In order to introduce a higher degree of flexibility the authors have defined a fourth value, **delta**, which allows the experienced user to specify a bearable gap from the best possible behaviour.

To have some evidence of the role the QoS specification plays in the selection process and to have a better idea of how these specifications affect the final result, we refer the reader to [ENG00].

Measures related to the system

As we have seen, staleness and RT are of use for evaluating system performance. Although they are very important markers of the QoS, we need to take account of other dimensions to supply a more complete evaluation of system performance. The authors put forward measures like **processing**, **communication delay** and **disk storage**. These concepts are applied to the DW and to the sources. They are necessary to strike a trade-off between the provided level of service and costs related to system features.

1.2.2 CATEGORISING SOURCE CAPABILITIES

The performance of a system depends on system properties. In the case of DW maintenance these properties are related to the sources. Therefore, there is a need to properly select and define the interesting characteristics of the sources. In this section we will establish the source characteristics to be used.

Categorisation of source capabilities is quite a new step in facing the DW maintenance problem with a clear and rigorous style. Some of the previous related work has made the assumption that each source is in a way active in the maintenance process, i.e. a source is able to make known the state of a transaction. The authors claim that not all the commercial DBMSs possess this property, making the existing methods non-applicable in a number of cases. An alternative that has been explored in some research is to extend a source by locating a wrapper process. Once again, the authors work through this alternative, widening the space of wrapping to the whole maintenance activity rather than studying the restricted area of source characteristics. The upshot is a set of relevant characteristics which deserve attention.

A first group concerns change detection capabilities. Assume we have a DW which reflects a single source and a DB user performs insert, delete and update actions, as the need arises. Being interested in the data stored in the DW, we would like to know information about the changes undergone by the database. One thing we would like to know could be whether changes have been made since the last time we have consulted the database. A source which is able to automatically discover and report that data have been modified is defined as **change active** (**CHAC**). Possible further information we would like to get is when the source last changed. This kind of information is usually provided on request and a source that implements such a service is called **change aware** (**CHAW**). All this would not be enough if we would like to be able to query the DW and demand the most recent data. For this, the DW must reflect changes to the source, i.e. the source must be able to propagate modification to the DW. This action may be performed automatically or under explicit request from the DW; in either case we call the source **delta aware** (**DAW**).

The most important property to note at this point is that these three capabilities are orthogonal: one ability does not imply the presence of one of the others or both of them.

Beside the change detection properties the authors introduce another capability, called **view-awareness** (**VA**). A source is said to be view-aware if it has the ability to single out each part of the view that has been modified from the rest of the view. As a consequence, a source with this capability must propagate as deltas only relevant changes and not all updates. It implies a smaller delay in terms of communication and processing time, when delivering changes.

At this point it is appropriate to emphasise that a typical warehouse architecture includes an integrator as shown in Figure 7:

Fig. 7. A more complete diagram including an Integrator between the DW and the source database. The Integrator acts as a bridge, receiving and passing on data

Consequently, there are three parties. Each of these parties can have the capability of being view-aware. For the maintenance problem, setting view-awareness on the DW side has no impact on the performance of the system. To understand this statement we have to recall that in this work we handle only materialised views.

Indeed, the capability of locating the changes does not affect the delay in getting the result for a materialised view stored in the DW, since it does not have to propagate the changes to anyone.

What does have an impact on performances is view-awareness as a property of the integrator or of the source.

Integrator view-awareness improves the propagation cost of the changes towards the DW, while source view-awareness gives the same kind of advantages between the source and the integrator.

The cost-model under consideration deals only with source and DW, leaving the interposing of an integrator for future work. Accordingly, only one interesting situation is left, namely the source being view-aware. In the future, each time we mention the ability of view-awareness, it has to be understood as source view-awareness.

There is one question left, regarding the wrapper process.

A wrapper is useful each time we want to extend the capabilities of a facility by means of a process which is responsible for implementing the desired service.

The point is what kind of implications has its location, as it can run either on the DW side (**remote** from the source) or the source side (**local** to the source). Once again, the difference lies in communication and processing cost. While view-awareness is definitely a clear benefit for this cost, the question about the localisation of the wrapper is a more tentative topic and it might not allow a precise decision to be taken in favour of one of the two solutions.

As a matter of fact, advantages and disadvantages related to one choice are complementary to those caused by the other one. The answer has to be found each time in regards to the peculiarities of a given system.

If we want to add a precise ability to a source and a wrapper is located on the DW side, the latter is forced to query the source remotely. The communication cost is penalised because a larger amount of data has to be sent. Furthermore, this calls the DW for an heavier processing delay and larger space for storage. On the other hand, if the wrapper is located at the source, the source is put up with an heavier processing delay and a larger space for storage, but this time there is no need for extra communication cost.

1.3 EVALUATION CRITERIA AND COST FORMULAS FOR DIFFERENT POLICIES

In the previous section we have sorted out all the elements used to write the cost-model formulas. In this section we will summarise the details of the cost-model.

1.4 OBSERVATION ABOUT SOURCE CHARACTERISTICS

To give a better profile of the implications of each capability related to a source, we will make some observations about its influence on the measures that we have introduced in the previous sections. The experiments cited are executed using the prototype the authors have written for implementing their cost-model. The prototype implements the formulas of the cost model and calculates a total cost for each single policy. It can be run from:

http://www.his.se/ida/research/groups/dbtg/demos/dwmp/

The scheme taken into consideration is composed by a DW which reflects data from one single source. A standard scenario is assumed for all experiments, as illustrated in Figure 8:

Fig. 8. Dialog of the prototype setting the default values for an experiment

We start from the point where the source has no capability. For each experiment we will set one single property to take note of how it impacts on the measures. We will follow this order: CHAW, CHAC, DAW, remote/local and VA.

If the source is CHAW, we can see some changes in periodic incremental policy and periodic recompute policy. More precisely, for periodic incremental we remark a smaller cost of source processing only. For periodic recompute the values of source and DW processing as well as communication cost sensibly decrease. Table 1 shows the figures:

POLICY	MEASURE	Not CHAW	CHAW
P1	Source processing (sec)	4.4389	2.21976
P2	Source processing	3.01	1.505
	W. processing	0.25	0.125
	Communication (sec)	0.07639	0.038197

Table 1. Change in figures when selecting the CHAW ability

As we can see from the table, the values in the presence of the CHAW capability are half of the corresponding values in the absence of this capability.

The introduction of CHAW affects the periodic incremental policy and the periodic recomputed policy in two different ways. In order to understand the reasons, we have to look at the nature of the cost-model.

In an incremental policy, if the source is CHAW we have to introduce in the source processing cost formula the factor $y(u-1)$ which tell us that cancellation is possible where the change size is zero.

On the other hand, $y(u-1)$ is removed as a factor from the DW processing cost formula because the source is extended locally. For the same reason there is no added communication cost.

For a recomputed policy the source and DW processing costs are complementary: the set of operations to complete is the same but it changes where the operations are done. The source being CHAW, we have to insert in both cost formulas the factor $[w+y(u-1)(1-w)]$ (that is strictly <1). Again, we may cancel recomputes when the source is not updated. For the same reason the communication cost decreases. As a matter of fact, we can remark a parallel between the changes of the costs: the source and DW processing costs decrease to half and the communication cost follows them. This is due to the specified factor.

Again, the source being CHAW, there is a benefit when there is a polling action but no change occurred in the last period; in this case no action has to be taken.

At this point someone could remark that it is strange that there is no benefit for on-demand policies. As we assume regular intervals between queries (updates as well as periodic polls) the update frequency has to be lower than the query frequency for this to be utilised. Otherwise there will always be at least one update between queries. If we had Poisson distributed updates and queries there would be a small fraction of queries which may utilise CHAW independently of inter-arrival times. This is a simplification in the model which gives an upper bound on staleness, for example.

If a source is change active no changes are reported in the table. If we read through, the detailed cost model formulas do not depend on this capability: once one of the three policies is given it doesn’t matter whether the source is able to detect and report that changes have been made.

As a matter of fact, if the chosen policy is immediate we will have an update for every change. If a polling frequency is given it is respected even if someone reports that changes have been made; on-demand policies transmit their query each time the user does. This statement underlines two concepts:

1. CHAC and CHAW are orthogonal capabilities. Specifying that a source is able to detect and report changes have been made is quite different from specifying that a source is able to tell when the last change was made;

2. CHAC does not affect the cost components. It is a requirement for immediate policies (if we do not have CHAC we have to do periodic polling which makes immediate incremental a variation of periodic incremental and so on).

If we check the DAW's checkbox we can take note of the changes for incremental policies shown in Table 2:

MEASURE	INCREMETAL POLICY	NOT DAW	DAW
Staleness (sec)			
	Immediate	469.106	25.2166
	Periodic	569.106	125.217
	On-Demand	469.645	25.6297
Source Storage (byte)			
	Immediate	1.0 E7	0
	Periodic	1.0 E7	25000
	On-Demand	1.0 E7	75000
Source Processing (sec)			
	Immediate	2.21976	6.25E-1
	Periodic	4.4389	6.25E-1
	On-Demand	1.11019	6.25^{E}-1

Table 2. Change in figures when selecting the DAW ability

Notice that there is a radical improvement in the values when we set the DAW ability. This is due to the formulas of the cost-model: thanks to DAW we are allowed to eliminate some contributes that are otherwise calculated in absence of the ability.

If we analyse the formulas proposed for the incremental policies, we find out that if the source is not delta aware then we have to compute as cost the quantity [r + a(M,K)] for all the measures which see a change in their value. K represents the size of the change, being obviously different for the three incremental policies.

This means that the DAW capability cancels the processing delay due to recomputing the view and the processing delay due to change detection. This is comprehensible since the delta of a relation was brought in to reach a higher efficiency for incremental view maintenance.

There is a further improvement in RT for on-demand incremental policy. As underlined in the detailed cost model for on-demand policies, RT is the delay due to refreshing the view, which is identical to the staleness cost.

A DAW source hasn't any influence on the costs sustained in the case of a recompute policy. This is clear since these policies have a processing cost for recomputing the view whether on the source side or on the DW side.

If we say the source to be wrapped remotely the dialog presenting the details of the cost model shows a lot of changed values as reported in Table 3 and Table 4:

	INCREMENTAL POLICY	LOCAL	REMOTE
Staleness (sec)			
	Immediate	469.106	1020.19
	Periodic	569.106	1120.19
	On-Demand	469.645	1020.32
RT (sec)			
	Immediate	0	0
	Periodic	0	0

	On-Demand	469.645	1020.32
Source Storage (byte)			
	Immediate	1.0E7	0
	Periodic	1.0E7	0
	On-Demand	1.0E7	0
DW Storage (byte)			
	Immediate	1.0E7	1.0E7
	Periodic	1.0E7	1.0E7
	On-Demand	1.0E7	1.0E7
Source Processing (sec)			
	Immediate	2.21976	1.25
	Periodic	4.4389	2.5
	On-Demand	1.11019	0.625
DW Processing (sec)			
	Immediate	0.125625	3.46945
	Periodic	0.125625	6.93859
	On-Demand	0.063125	1.73488
Communication (sec)			
	Immediate	1.45367E-4	0.38152
	Periodic	1.45367E-4	0.763039
	On-Demand	1.20367E-4	0.19076

Table 3. Change in figures according to the location of the wrapper for incremental policies

	RECOMPUTE POLICY	LOCAL	REMOTE
Staleness (sec)			
	Immediate	333.639	852.304
	Periodic	433.639	952.304
	On-Demand	333.639	852.304
RT (sec)			
	Immediate	0	0
	Periodic	0	0
	On-Demand	333.639	852.304
Source Storage (byte)			
	Immediate	0	0
	Periodic	0	0
	On-Demand	0	0
DW Storage (byte)			
	Immediate	1.0E7	1.0E7
	Periodic	1.0E7	1.0E7
	On-Demand	1.0E7	1.0E7
Source Processing (sec)			
	Immediate	1.505	1.25
	Periodic	3.01	2.5
	On-Demand	0.7525	0.625
DW Processing (sec)			
	Immediate	0.125	2.63
	Periodic	0.25	5.26
	On-Demand	0.0625	1.315

Communication (sec)			
	Immediate	0.038197	0.38152
	Periodic	0.0763939	0.763039
	On-Demand	0.0190985	0.19076

Table 4. Change in figures according to the location of the wrapper for recompute policies

The staleness increases.

For incremental policies this is due to the fact that the processing delay for change detection moves from the source side to the DW side. Its cost may not change; what is becoming heavier is the cost related to extraction, communication and storing a larger amount of data.

Apparently, the influence of these costs is stronger than the saving we make by eliminating the processing delay of incremental maintenance.

For recompute policies we have a similar phenomenon. This time the processing delay to recompute the view moves from the source side to the DW side. Costs related to extraction, communication and storage are heavier as the whole source has to undergo these operations.

As a consequence, the source processing delay is shorter. The fact that the values of source processing are the same by pairs (immediate incremental=immediate recompute, periodic incremental=periodic recompute, On-demand incremental=On-demand recompute) does not express a specific behaviour. They are equal because we haven't specified any other property; it is possible, therefore, that if we specify a further capability one cost may change while the other stays fix.

It is intelligible that the DW processing delay increases, too.

F being the view predicate selectivity, communication cost is F times bigger than the recompute policy value since the whole source has to be sent.

The capability of being view aware does not seem to have any influence if chosen alone. Hence, to get some evidence of VA effects it is interesting to choose it together with remote.

In staleness, the processing delay to recompute the view moves to the source side. The same delay moves from DW processing to source processing.

For communication the argument of d( ) changes from N to M, i.e. the argument is smaller.

This is true for both incremental and recompute policies.

CHAPTER TWO

PROBLEM ANALYSIS

In this chapter we move toward the issue of the implementation of the benchmark environment by identifying the key characteristics required of the components of the overall system, and issues to do with its development.

2.1 THE SYSTEM UNDER TEST

As we discussed in the chapter dedicated to the presentation of the cost-model, the system taken into consideration is shown in Fig.1:

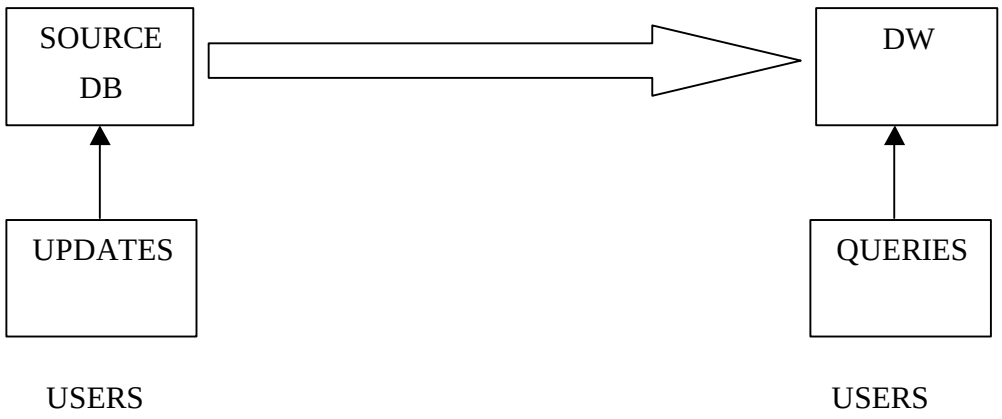


Fig.1 Basic system.

As a consequence, we have taken a platform an d created a real database and a data warehouse.

The choice of which particular platform or DBMS to use has not been an issue since the cost-model is intended to be applicable to any system configuration that complies with the above illustration.

In our case, the platform has been a Sun Sparc 5 with 128 MB RAM running under Solaris 8. The DBMS used was Interbase 6.0 and the source database has been naturally accessed by means of JDBC (Interclient1.6). One reason for the choice of InterBase was that it is o pen-source, leave some margin in the event that we needed better control over the database.

In the case of the data warehouse we had to ask ourselves what we really needed in order to model the data warehouse in our environment. From the cost -model, it was apparent that we did not need a complete and complex data warehouse system to extract data from the view it provides. What was actually required was simply the view itself for answering queries users may pose to the warehouse.

2.2 THE APPLICATION

Once the SUT was set up, we moved our attention to the second part of the benchmarking environment, i.e. the application that is charged with taking control over the execution of experiments and measuring its performance.

The first issue is what language could be most suitable to build such an application.

Our approach has been to provide a list of important features against which the language should match up:

1) **Portability**: as the cost-model is for general use whatever system we are working on, it is useful if the application can be run on different systems to be able to evaluate different arrangements.
2) **Concurrency**: whenever we want to benchmark a system that belongs to the real world we find that its behaviour is in most cases non -linear, many phenomena happening simultaneously. The imperative is hence to have good support for handling the concurrency of the several processes interacting in or on the system. The language must make available facilities (constructs) to help with concurrency.
3) **Connectivity**: since dealing with a database implies accessing, querying and updating its content, the language must allow handling connections to a DBMS. In view of the fact that we care about portability, changing DBMS should not be a hard-coded parameter inside the application. On the contrary, for the sake of flexibility we should be allowed to easily change the type of DBMS without changing the code.

Java was a natural choice as it satisfies all of the above. The two -stage Java compiler grants portability. Running threads and synchronizing them assist the control over concurrency. Finally, JDBC is a compliant facility for either opening connections or querying and updating data in a database.

2.3 BASIC CRITERIA BEHIND THE IMPLEMENTATION

Approaching the design phase, we h ave singled out the major principles which should direct the development of the application.

A first set of properties to preserve when drawing up the code naturally belongs to the sphere of Software Engineering (SE).

Java is first of all an OO language and thus all SE concepts related to this field are applicable.

Following those principles it is particularly serviceable to pursue three aims:

1) Preserving a clear distinction between classes that model the S.U.T. and classes that are designed to control a nd measure the system.

   This is a significant point, especially for an application designed for benchmarking. For such an application, it is important to keep the object of analysis separated from the part for analysing it in order to avoid artificially affecting the S.U.T.'s behaviour. In other words, partitioning the software into two sets of classes helps to preserve the objectiveness of the performance evaluation;

2) Reducing overheads as much as is feasible. Observing principles like minimum correlation or redundancy helps not only in writing a more readable code but also to remove all unnecessary activity, optimising the management of the resources required by the application itself, an important property in a benchmarking environment;

3) Catering for futu re enhancement, for example a move to using RMI for remote wrapping.

A further key principle to carry through is the correctness of concurrency handling.

This topic is particularly ticklish due to the difficulty apparent in human beings of thinking in a concurrent way. As a matter of fact, the non -linearity of the interactions between several phenomena that have to be taken into consideration often seems to confound every effort of the mind.

Acknowledging this complexity, there is a need for a clear app roach to the matter.

A valuable help towards clarity should come from the instrument we use to implement the application. It should make provision for constructs expressively studied for outlining the interactions involved in the system. The choice of Jav a fulfils this requisite. Java caters in fact for concurrency handling by means of threads and their timing (keyword 'synchronized').

Threads capture the different processes that accomplish the dynamics of the system, while declaring methods accessible fr om threads as 'synchronized' controls the interaction between threads whenever critical actions have to be performed. Critical actions are those we want to be atomically completed, i.e. only one thread at a time is allowed to access the section that enclos es the code to be executed.

While the first aspect is undoubtedly straightforward, the threads' synchronization has to be carefully analysed for all its implications. The risk is to unawares let the system behaving unpredictably outside the rules that sho uld actually direct it. A good practice is obviously to synchronize only methods that really implement critical actions and, very important, those methods should be as short as possible, including only the essential code lines to carry out those critical a ctions.

2.4 THE KEY CRITERIA FOR A DOMAIN-SPECIFIC BENCHMARK

Besides the basic principles, our application should meet some other criteria as it has been intended to support a domain -specific benchmark. These are:

1) **Relevance:** the application should proper ly and correctly keep track of the significant magnitudes that characterize the system and that allow us to compare relative performance when investigating different scenarios within our problem domain. In our case, it is important to assume the same metri cs the cost-model does;
2) **Scalability:** the benchmarking environment should be perfectly stable and able to suit systems of the same nature but different size. In our case, it should be applicable for source databases of whatever size and for data warehouse views with different selectivities on the source table(s);
3) **Simplicity:** both the features of the implementation and the outcomes given by the experiments must be presented in a clear way in order to favour comprehension and avoid possible criticism.

CHAPTER THREE

DESIGN OF THE APPLICATION ARCHITECTURE

In this chapter we explain the design approach to building the structure of the application. Note that this chapter introduces the set of classes to be defined and their relationships, only. How those classes have been effectively implemented will be the subject of the following chapter.

3.1 MODELLING ENTITIES OF THE OBSERVED SYSTEM

The benchmark software must model the various parts that comprise the observed system. For this reason, approaching the design phase of the tool, our starting point has been the pattern discussed by the cost-model as illustrated in Figure 1:

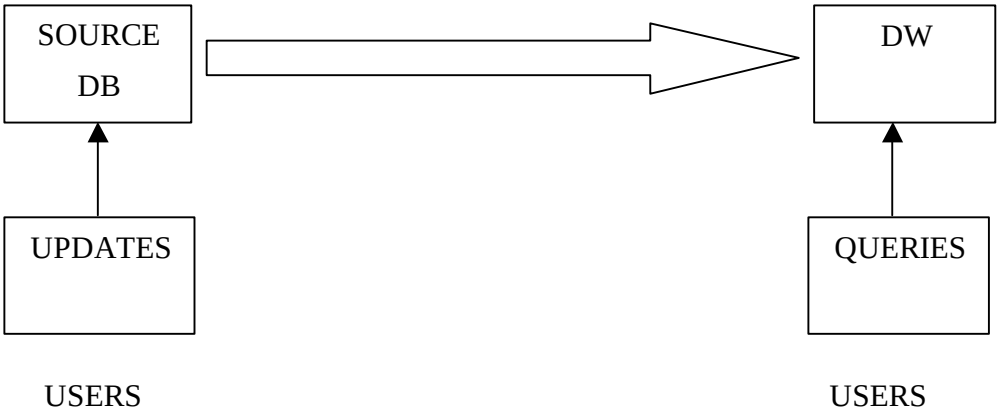


Fig.1. Entities considered by the cost-model

It seems clear we should model each of the following entities:

a) The source database: once we have created the database the application has to include a way to access it. Thus we have defined two classes: one holds all the methods needed to handle such access and the second specifies how these methods can be used for the particular case of Interbase as shown below:

Fig.2. Classes for accessing the database

In general, the source database consists of one or more tables. These kinds of objects must be represented in our application. It hence makes provision for two new data types: the table and its component, the row as illustrated in Figure 3:

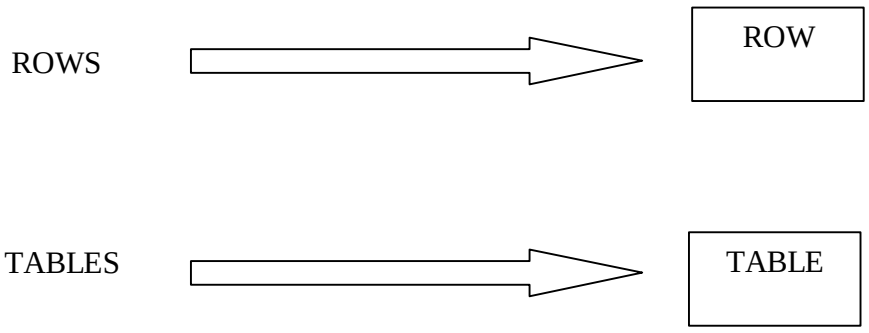


Fig.3. Row and tables as new data types

b) The Data Warehouse: the cost-model pictures a data warehouse as composed of two distinct parts: the view and the wrapper. The former is basically a table and the latter is charged to carry out maintenance of the view according to the desired policy, taking advantage of potential source characteristics.

Figure 4 shows the class that is committed to access to the data warehouse view and the one that wraps both source capabilities and maintenan ce policies:

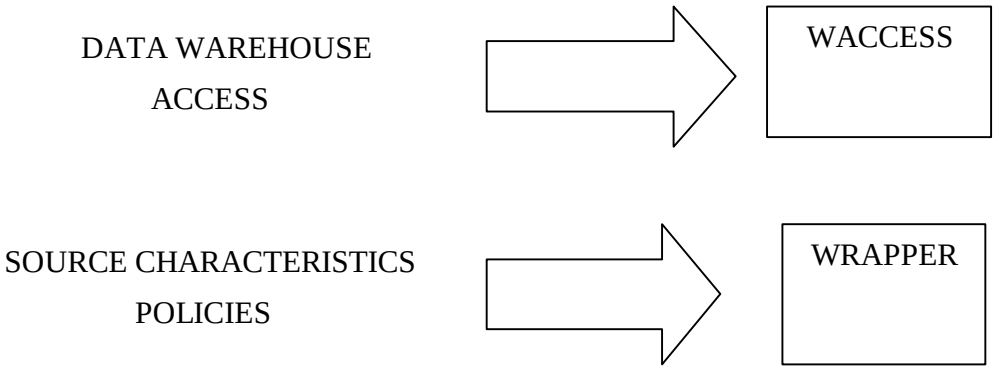


Fig.4. WAccess allows access to the data warehouse and its view.
Wrapper implements source characteristics and maintenance policies

c) Updates and queries: the process of updating and the process of querying are not physical entities of the system but at the same time affect the system. We must represent the dynamics it undergoes. Two classes have been defined to apply the stream of source updates and the stream of warehouse queries. We have called them Supdater and Wquerier as shown in Figure 5:

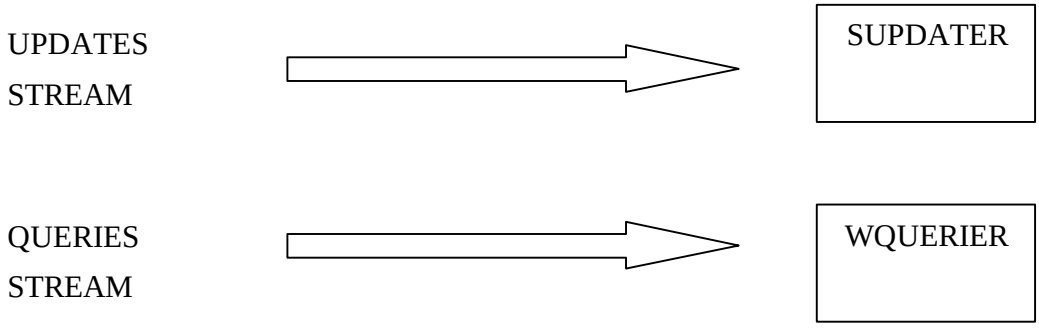


Fig.5. Classes to simulate users updating a database and querying a DW

3.2 BENCHMARK CLASSES

Once we have modelled the entities of the system, the second step has been the design of a set of classes for executing and controlling benchmarks.

There is a need for a class that directs all others for making sure that the benchmarks are executed in a proper and correct manner as illustrated below:

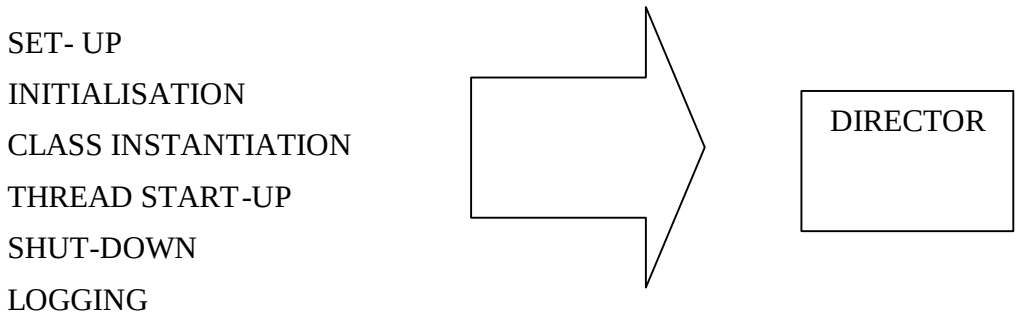


Fig.6. The classes that support an experiment

To make things clear, other classes help the director carrying out its duty.

Thus, there are two classes charged with collecting and storing the initial configuration of the parameters required to set up an experiment:

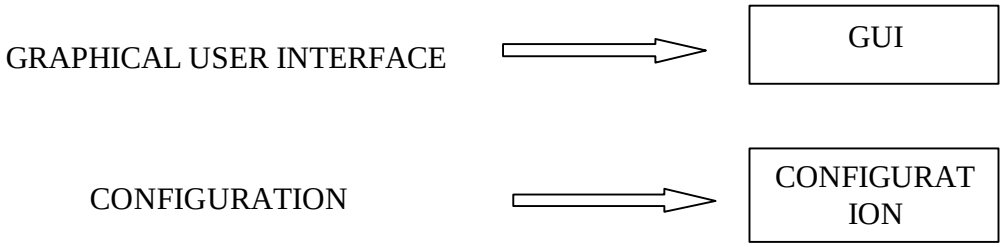


Fig. 7. Classes to set up our parameters

3.3 UNIFIED CLASS DIAGRAM

The unified diagram illustrating the final structure of the benchmark software is presented below:

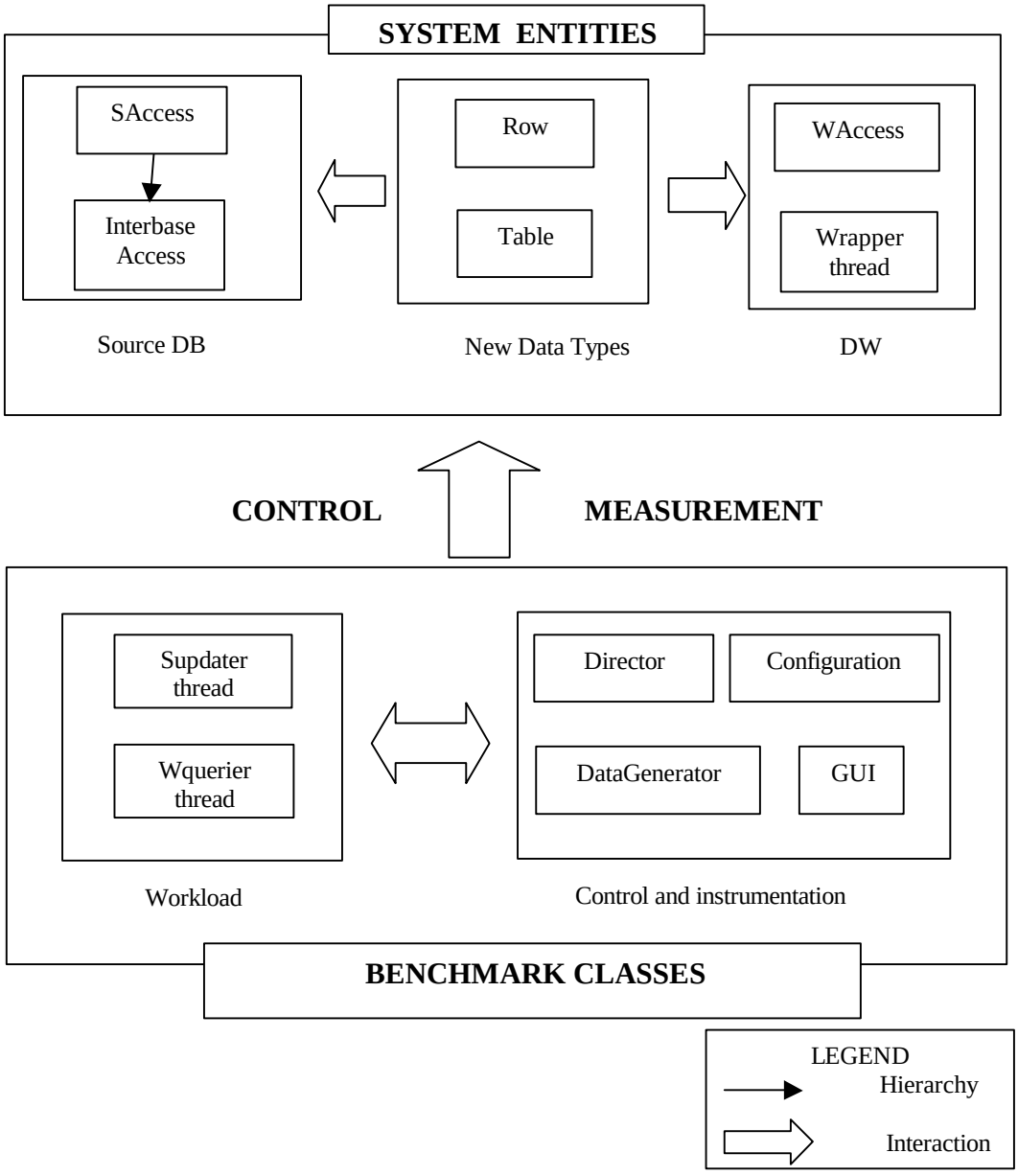


Fig.8. The final structure of the benchmark software

CHAPTER FOUR

APPLICATION IMPLEMENTATION

In this chapter we give details about the techniques we have drawn on when developing the code to build a real instance of the system assumed by the cost-model. The purpose has been to implement functional behaviour in a way which is realistic for a data warehousing scenario.

4.1 IMPLEMENTING THE CLASSES TO MANAGE THE SYSTEM ENTITIES

4.1.1 THE SOURCE DATABASE

In this section we discuss the implementation of the classes for interacting with the source database: DBAccess, InterbaseAccess, Row, Table.

Exploiting JDBC to manage access to the database

For accessing databases Java's designers have introduced Java DataBase Connectivity (JDBC). We have utilized it to manage the connection to the source database and to query and update it. This implies that we will only be investigating view aware sources with this tool.

Basically, there are three key elements to deal with: connections, statements and result sets. A result set collects the result of a query stored in a statement. A statement is used to pose the query to the database the connection refers to. A statement can contain an update string, in which case no result set is required.
If a connection is closed then all its statements are automatically closed.
If a statement is closed then all its result sets are automatically closed.

A crucial matter concerns whether we should have one single connection or more connections to the same database; whether we should create one single statement and change its string each time a new query or update has to be performed or use a new statement for each query or update.

A first approach suggests that a logical solution could be to have one single connection mainly because opening and closing more connections could bring in an overhead while executing an experiment. Practical experience has shown that when a single connection handles a number of queries or updates then the time to complete such actions is much higher than the time to perform the queries or updates opening a connection for each of them. An example can be taken from the data generator: to fill 1000 rows, in the case of opening and closing a new connection each time it takes approximately half of the time required in the case of one single connection. Apparently, opening and closing a connection is not as costly for JDBC as we thought at the beginning.

As far as statements are concerned, our choice has been that of declaring a statement each time we have to perform a kind of query, i.e. there is no general statement but each method declares locally its own. This is expedient to avoid problems when parsing the result of the query since it is not possible to go through a result set if the relative statement has been closed. The latter situation could be feasible seeing that the application has a high degree of concurrency.

Row and table as new data types

Since the tool manages source tables and rows, it is good programming practice to make them two new data types.

Thus, there are two classes to model the data type row and the data type table. We make use of their instances whenever we need to process tables (for example when comparing two tables) or to pass tables as method parameters (for example, to communicate a table from the source to the wrapper).

The structure of a row (and consequently of a table) has no particular constraints but three identifiable fields:

- the primary key: consists of a single “not null unique” attribute;
- a Guide Attribute column: directs the course of the update stream;
- a View Attribute column: used to select which rows belong to the DW view.

Apart from these fields, there are no constraints about the number and the nature of the rest of the row. At the same time the size of a row should be kept flexible to match different needs. We have thus found it opportune to define a fourth attribute governed by a parameter which dictates the size of a CHAR type.

4.1.2 THE DATA WAREHOUSE

In this section we discuss the implementation of the classes that represent the two entities of the data warehouse: WAccess and Wrapper.

The instance of the DW view

The view of the DW is like a table of a database. To keep things simple we have chosen to store the view in main memory avoiding the complexity of a real data warehouse system. Notice that this choice does not reduce the validity of the benchmark itself because we are not interested in the performance of the data warehouse but in the performance of the maintenance policies against the parameters defined in the cost model.

Having the whole view in main memory entails that:

a) the part of the warehouse processing cost related to recomputing or incrementally updating the view is zero;
b) the warehouse storage cost is zero;
c) there is no need to handle the access to the DW since the view is represented as an object of type Table in main memory.

The wrapper

The wrapper has two tasks: setting up source characteristics and managing maintenance policies.

Source characteristics: CHAC, CHAW and DAW

The source characteristic identified by the authors of the cost-model are ahead of actual commercialised DBMS in the sense that none of them provides CHAC, CHAW or DAW features as basic kernel properties. In particular, they are not available in Interbase.

As a consequence, the source DBMS has to be extended for providing such capabilities.

The key point is to introduce the smallest feasible overhead so as to prevent prejudicing DW performance.

For change activeness and change awareness we basically use communication between threads. In particular, the process engages the thread that performs updates on the source database and the wrapper thread:

?? each time the source updater commits an update packet, it notifies the wrapper of the sequence number of that update by invoking a method of the wrapper.

?? the wrapper method keeps up-to-date the variable that contains the order number of the latest update packet committed in the source.

By means of this simple method invocation, the tool provides change activeness, i.e. the ability to automatically detect and report that changes have been made. After all, the communication between the two threads turns out to be a mere method call whose overhead can be safely considered negligible.

The wrapper stores a second piece of information that is of interest for implementing source characteristics. It stores the order number of the latest update packet propagated to the DW. Then, by comparing the latest propagated and the latest committed update packet numbers, the wrapper knows whether the DW view reflects the most recent changes. This is the piece of information required by change awareness. Once again, the overhead introduced by this small control can be neglected.

For the sake of accuracy, we must underline that the definition of change awareness involves the time the last change was completed. This piece of information can be retrieved from the object 'Director', which keeps track of the time each change has been committed. On the other hand, for our application CHAW matters for knowing whether the DW view reflects the latest changes or not. If it does, no change transmission is required and therefore we save time when satisfying a query. We claim it is not necessary to perfectly suit to the formal definition, as we can simulate the benefits of having the desired source property with an effective and straightforward shortcut.

The DAW capability is more delicate since the data it mines are in the form of a table.

Consequently, we create a second table in the source database which we term the DAW table. It has the same fields as the source tables plus a further field (updateType) whose meaning will be made clear below.

To have delta awareness, the DAW table must keep track of each update to the source table. If the source table undergoes an insert then the new row has to be reported to the DAW table. If a delete happens then once again a row has to be reported to the DAW table, this time with the deleted old values. An update is treated as an insert plus a delete.

Notice that only changes to rows belonging to the DW view (VA equal to 1) are reported.

It is clear that when reporting a row to the DAW table we must specify the type of update that occurred to the source table. The updateType field helps for this.

All that is left is the mechanism that automatically reports the updated rows to the DAW table. In the field of DBMS there is a very popular and useful construct: the trigger. We create a trigger that fires each time the source table has been modified and among the changed rows gives an account only of those belonging to the DW view.

As we can notice, the DAW capability is provided in a more artificial way than the other two. The cost of an added table and of a trigger may not be negligible as claimed for the mechanism implemented to supply CHAC and CHAW.

Being aware of the possible overhead introduced, we claim there is still a feasible way to preserve the right comparative outcomes of different experiments. Our solution makes provision for making the trigger active no matter which source properties are chosen. In doing this, the tool impinges upon system performance bringing in always the same delay.

Maintenance timing implementation

The cost-model takes into consideration three commonly mentioned timings: immediate, periodic, on-demand.

To understand how the benchmark software implements them, it is convenient to focus on the subset of classes that handles the maintenance timing. Figure 1 shows the classes we have to consider:

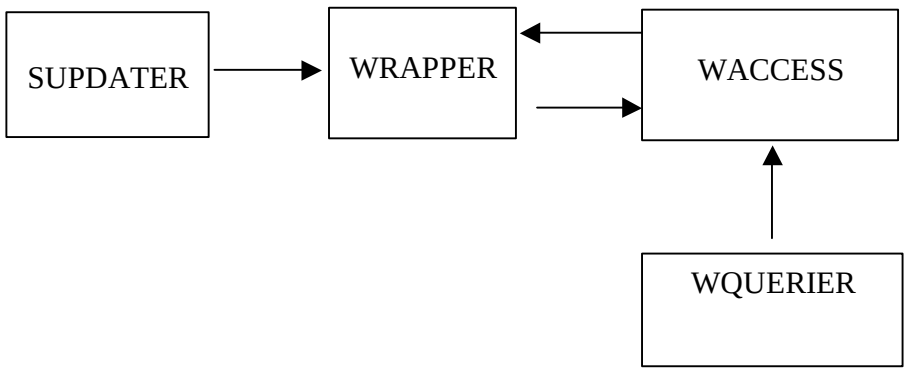


Fig.1. The subset of classes that handle the maintenance timing.

For an immediate policy, when an update is committed then it has to be *immediately propagated* to the DW view.

Thus, in order to put into effect an immediate policy, it is necessary to make available a mechanism that alerts when changes to the source database have taken place. It is basically the same mechanism that lies behind the implementation of the source capability CHAC.

As a matter of fact, if a source is able to automatically detect and report that changes have happened, then each time the source does so we report the changes to the DW view. In terms of classes, when the class which carries out the update stream commits an update, it should give notice to the wrapper that *immediately* transmits the data to the DW.

A periodic policy, instead, performs maintenance on a regular basis. Thus, we need an algorithm that triggers maintenance after a given amount of time.

For an on-demand policy maintenance is triggered when a query is submitted to the warehouse.

When the latter receives the query, it warns the wrapper to retrieve the most recent data from the source database. Once the warehouse receives the changes, the query can be finally satisfied.

Whatever the maintenance policy, the wrapper starts a thread to handle it. To control the state of the thread for the different policies we make use of Java methods such as wait() and notify(). Notice that it is necessary to declare as 'synchronized' the methods that make use of notify(). Its use has therefore been carefully constrained to avoid unpredictable dependencies between threads

Recompute and incremental

Considering how maintenance is to be performed, the authors of the cost-model have chosen either recompute or incremental approaches.

Recompute policies suppose that, when required, the source transmits the committed changes and when this happens the warehouse completely reviews its table.

At this stage CHAW comes out to be very important: if no alterations have been made since the last warehouse view update, we must not send data from the source to the DW.

Incremental approaches give enough information for the DW to update only the modified rows of the view.

Now, two capabilities become significant: CHAW for discerning whether modifications have in effect been completed since the last time the DW view reflected the source table, and DAW for making out delta changes, i.e. to know which subset of source table rows has been altered during the last transaction.

In the case of incremental policies, the wrapper transmits the delta changes to the DW in any case, even if the source is not extended to provide the DAW capability. There is hence a need for a method to compute delta changes at run time after the wrapper has retrieved the whole source table. The method itself is located in the wrapper.

Distinction between the wrapper and an integrator

The following concerns the wrapper and the role it plays in the tool.

At a glance, it may appear that the wrapper is, to a large extent, an integrator. In point of fact, they are unlike. Both collect data coming from the source database and store them. Further, either the wrapper or the integrator could do the extraction of delta changes. This is as far as their similarity goes. The dissimilarity turns up towards the DW side.

While the wrapper reports data to the DW following the timing laid down by the policy it is bringing into effect, an integrator is not totally free in choosing the time the maintenance of the DW has to be performed. The DW, in fact, can warn the integrator to be unwilling to slot in changes and can prevent the integrator from propagating its data. The DW is not able to notify the wrapper about this sort of situation.

Further, extracting delta changes by the integrator would reduce source computation but increase the communication cost as currently done.

Thus, while the wrapper looks like a mere executor of maintenance policies, an integrator is a more active unit inside the system, able to interact with other entities.

CHAPTER FIVE

WORKLOAD AND INSTRUMENTATION

In this chapter we give details about simulating workload on the system and the instrumentation that measures its performance. The workload requirement involves the simulation of a stream of updates and a stream of queries. Instrumentation deals with the metric defined by the cost-model.

5.1 SIMULATING A WORKLOAD

5.1.1 CHANGES TO THE SOURCE DATABASE

The DW maintenance problem arises from the fact that its source database undergoes some changes during its life. Consequently, how to simulate these changes during our experiments plays a relevant role. It is appropriate to remark that the matter we are dealing with in this section is slightly different from the one related to the DW. While the DW maintenance problem directs how to export a materialised view, the matter we are going to face is how to import a view, i.e. how to install a stream of updates.

In the literature, related work concerning performance studies or benchmarking has addressed the latter issue. One of the most recent is [COL97]. It suggests a lucid approach for implementing a stream of updates.

It brings into discussion elements of control over the variable size of updates, i.e. number of rows involved in a single update, and over the percentage of updates out of the total number of transactions.

While a good step in deepening the topic, we claim this is not to be sufficient to clarify how the stream of updates has to be understood and implemented.

Therefore, to our knowledge previous work has been rather indistinct on this point, implicitly assuming that incorporation of view update presents no complication.

Thus, in order to set up experiments in an objective and comprehensive way it is necessary to further investigate the nature of the workload submitted to the source database, i.e. the size and the frequency of updates.

5.1.2 APPLYING UPDATE STREAMS IN A DB SYSTEM

A source database is, in general, kept updated by a number of users. These users perform modifications independently from each other. Typically, we use statistical approaches to describe this kind of phenomena. As a consequence, we will use statistical distributions in order to review how to produce a stream of updates for our benchmark software.

Our remarks arise from the fundamental concepts developed from the careful analysis of Adelberg et al. in [ADE95].

5.1.3 A POINT ABOUT THE TOOL ARCHITECTURE

A first question addresses how to integrate the stream of updates in our application. As designers we must bear in mind that our purpose is to handle updates in the tool environment.

Adelberg et al. give valid reasons in favour of a single separate and independent process handling. In fact, with such a process we are able to keep under control the stream of updates, untying the install of the stream from the rest of the execution. Therefore, we implement a thread which is responsible for installing the stream of updates. Only one thread is started to accomplish this task.

5.1.4 CHARACTERISATION OF UPDATES

According to Adelberg et al., we can characterise an update statement along two dimensions.

A first dimension has to do with the attributes the update explicitly changes. Considering that most of the time it modifies only the value of some attributes of the set of rows which match the where clause, an update statement can be written in two different manners.

The first one provides a *partial update*, since the update explicitly involves only those attributes whose values have to be modified.

The alternative is for the update to specify each single attribute even if its value has not to be changed. Adelberg et al. define this as a *complete update*.

In this work we find it useful to use partial update, assuming partial updates bring some advantages in terms of flexibility when implementing the stream, as we will remark later on.

A second dimension characterises the frequency of updates. From a qualitative point of view, the frequency can be steady or aperiodic. With a regular frequency, an attribute undergoes an update every fixed period of time even if its value is not supposed to change. With an aperiodic frequency, instead, updates are performed unpredictably.

We believe the latter gives a more realistic description of the phenomenon.

5.1.5 VIEWS AND STALENESS

For the sake of completeness we find it interesting to briefly go into a particular point that Adelberg et al. bring to our attention. Thus, there follows some remarks upon views and staleness.

[ADE95] takes into consideration some view properties. In particular, it is worth noting the approach to the property termed staleness, since its definition is one of the main achievement of the cost-model.

[ADE95] states that "there are actually several options for defining staleness".

According to Adelberg et al. one possible assumption to start with is called *unapplied Update*: "a data object is always fresh unless an update has been received by the system but not yet applied to the data".

Our remark is that the definition proposed in [HEN00] assumes this statement and therefore the analysis of Adelberg et al. is applicable to our case.

5.1.6 UPDATE MODEL

Two more points are relevant to modelling updates.

Firstly, we apply an update *as soon as* it arrives at the source database.

Adelberg et al. use this mechanism when touching on the case of a Do First Update policy.

A Do First Update policy is used in a system that has to commit either transactions or updates. The policy gives clear priority to updates. This is our case since we deal with updates only.

A positive consequence is that no update queue is required since updates are executed whenever they arrive at the source database.

Secondly, we share the choice of Adelberg et al. to shape the update arrival as a *Poisson distribution*. A Poisson distribution is often used to represent phenomena like users independently modifying source data.

Summing up, our update model assumes:

- ?? partial updates: the update explicitly involves only those attributes whose values have to be modified;
- ?? aperiodic frequency: updates are performed unpredictably;
- ?? immediate execution: we apply an update *as soon as* it arrives at the source database;
- ?? Poisson distribution: we shape the update arrival as a *Poisson distribution*.

5.1.7 PRODUCING A POISSON DISTRIBUTION

The simulation of users updating the source DB or querying the DW is done by means of a Poisson distribution as suggested by [ADE95].

The Poisson distribution models the number of occurrences of a rare event, i.e. it is valid for events that have a low probability of one single happening and is often used in simulation. For instance, it is used to model phone calls arriving to a telephone exchange or requests to a Web-server.

We want now to discuss briefly beyond this assumption, assuming that our phenomenon satisfies the Poisson hypothesis.

The literature presents several algorithms for implementing a Poisson distribution about an expected value. One example is given by [Weiss98]: " we repeatedly generate uniformly distributed random numbers in the interval (0,1) until their product is smaller than (or equal to) exp(-a)", where constant a is the mean number of occurrences. Weiss' algorithm is shown in Fig.1:

```
public int poisson( double expectedValue)
{
double limit = Math.exp( -expectedValue);
double product = randomReal();
int count;

for( count=0;product >limit; count++)
product *=randomReal();
return count;
}
```

Fig.1. The algorithm for implementing a Poisson distribution as presented by Weiss

We claim that using this method in our application may introduce an unnecessary delay, seeing that what we are looking for is computing the inter-arrival time between two occurrences.

Fortunately, Poisson analysis supplies a direct formula for that purpose. Therefore effectively the algorithm used is (Figure 2):

```
//compute the delay following Poisson unknown=randomValue.nextDouble();

/* in this way we sleep a Poisson inter-arrival time after the latest occurrence */
seconds= ( (1/freq) *Math.log(1 / (1-unknown) ));

//we compute the delay in milliseconds
delay= (int) (seconds*1000);
```

Fig.2 The algorithm effectively in use

As we can notice, we can deem the overhead brought in by these few lines to be negligible.

5.1.8 ALTERNATIVES FOR IMPLEMENTING THE UPDATE STREAM

Up to this point our examination has presented two key aspects related to the implementation of updates stream.

The first concerns the structure of the tool, which is split in separate threads, and the second deals with the statistical properties of the stream.

As yet no importance has been given to the size of each update, i.e. the number of rows affected by an update statement (we remember that with the word 'update' we intend one of the three possible actions insert, delete, update). When necessary, we will refer to the update size with the letter L (cost-model parameter).

The related work to our knowledge takes a rather simple approach to the issue of update size. In fact, it is assumed that each update affects one single row of the table.

Yet, our cost-model makes provision for a parameter that specifies update size and consequently, we have to extend our benchmark software in order to address this issue.

As the literature is a little vague on this topic, we have explored a number of solutions, which we will catalogue and present in the following sections.

5.1.9 ENLARGED ISSUE

The question of update size involves a bigger issue.

As a matter of fact, the solution is strictly bound to the distribution of values in the source table and therefore to the scheduling of the update stream.

5.1.10 FEASIBLE APPROACHES

Our analysis has brought to our attention two possible approaches, as illustrated in Figure1:

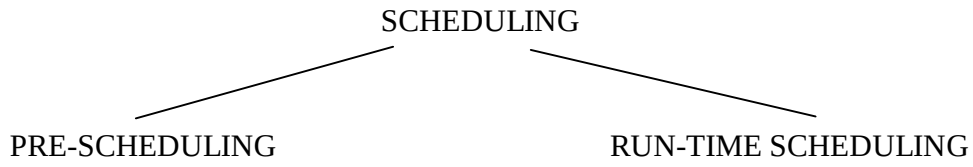


Fig.3 Approaches diagram

The goal of pre-scheduling is to prepare a sequence of updates before starting to run an experiment.

On the contrary, with run-time scheduling we dynamically schedule updates during the execution of an experiment.

Both approaches have been explored. As outcome we can group the solutions explored (with their shortened names) following the kind of scheduling they carry out:

1) Pre-scheduling: One Call for All (OCFAl), Guide Attribute (GAt) , Scheduling Table (ScheTa), String of Keys (SOK), String of Values(SOV);
2) Run-time Scheduling: One Call for All (OCFAl), Run-Time Guide Attribute (RuTGAt).

As we can see, the list of pre-scheduling alternatives is richer than the list for run-time scheduling. This is due to the fact that run-time scheduling must compel much stricter constraints of time and overhead than pre-scheduling , where we can take our time to generate a stream.

In the next sections we will present the different approaches, drawing attention to their properties, and to the advantages and the disadvantages of each of them.

One Call For All – OCFA

This represents a trivial solution.

The stream is composed by simple updates that affect one row per statement. Thus, each update turns out to be one JDBC call. It means that the benchmark tool always brings about the same delay.

This technique could be used for pre- as well as for run-time scheduling. In the case of run-time scheduling, we must account for a delay each time an update has to be performed since the preparation of the statement for its execution takes time.

Guide Attribute – GA

The basic concept is to add to the source table an attribute that plays a particular role. It will guide the search of the rows to be updated. Thus, we name this attribute a Guide Attribute (GA).

The pre-scheduling of the stream lies in working out an ordered sequence of updates. Since the goal is to predetermine this sequence, the application must know how many update packets it needs to complete it. Once the application has this knowledge, it can start to assign values to the GA of each tuple before running an experiment. These values are the order numbers in the sequence of updates.

More precisely, the table is initialised with all GAs equal to zero.

Then, for a given order number in the sequence, we set some GAs to that order number, i.e. the nth update will affect rows with GA equal to n. Furthermore, since we want an update to affect L rows, we set exactly L rows to have the same GA value.

A feasible way to store the sequence is using a simple array. When the experiment starts we go through the array of objects that stores the data for the update stream.

Consequently, performing updates means following the order number of an update packet and executing a statement that involves the rows with GA value equal to the actual order number.

Notice that updates and deletes affect a subset of rows in one transaction, while inserts are executed one by one.

In spite of being a clear approach to a pre-planning of the update list, this technique delivers a limited stream, since each row can undergo only one update during the experiment.

Scheduling Table – ScheTa

The starting point for this technique is the creation of a new table in the source database besides the one we use to store the interesting data.

This added table determines the sequence of updates; thus, some fields are defined for this purpose: a sequence number field to group the rows updated by one statement (there are L rows with the same number), a type field to show the type of update, the key field to specify the key of the row to be updated in the source table and various fields to take in values we want to change (in the case of update or insert).

The advantage is that we prepare all data, including the new values for some fields, needed to replace the original when pre-scheduling. Secondly, the stream is not bounded by the source table size since we have direct control over the size of the added table. Finally, there is a homogeneous delay in performing inserts, deletes and updates since in each of the three cases we have one call for each affected row.

On the other hand, the second table renders an overhead due to accessing it. Better performance could be reached using an index on the key of the added table. In addition, managing an added table hints that the stream is anyway bounded by its size.

For the sake of clarity, we present an example. If we set the table which contains the interesting data to have the following fields:

ID	***VAL***
1	Crow
20	Owl
45	Eagle
2	Chaffinch

Table 1. The source table taken as example.

then the added table that keeps track of the pre-scheduled update stream will have the following structure:

No	*Type*	*Key*	*Val*
1	Ins	64	Seagull
1	Ins	26	Robin
2	Del	1	Nil
2	Del	2	Nil

Table 2. The corresponding added table.

Once the table has been filled, the pre-scheduling phase is accomplished.

When the experiment starts, it is sufficient to read from the added table which update is going to happen and to arrange the corresponding statement according to the data stored in the added table fields.

String Of Keys– SOK

This technique focuses on primary keys.

The aim is to put up sets of randomly chosen existing keys and perform on each set a particular update.

Each set will contain a number of keys equal to the desired size of updates L and will be represented as a string. According to the Java syntax, a typical string will have the following format: key[1]+","+key2[2]+","+…+key[L].

The choice of representing a set of keys by means of a string is profitable for creating the statement to execute an update. As a matter of fact, an update will basically affect the tuples whose primary keys belong to a set. Thus, the statement to execute that update will consists of two parts: the clause that specifies the type of update to perform together with the new values to be inserted in the case of update or insert; and a where clause that specifies the set of rows to be affected by means of their primary keys. In Java we can build such a statement by simply concatenating two predefined strings.

Assuming that after each scheduled update the algorithm keeps the set of existing keys up-to-date, with this method a row can undergo more updates of different types over its life.

On the other hand, though the update stream is not bounded by any of the parameters of the application, the update stream is naturally bounded by the finite number of strings we can store before the execution of an experiment.

String Of Values – SOV

If we apply the method above directly to the key attribute then the tuples affected by an update could be accessed benefiting from indexing on the primary key.

If instead there is a need to avoid this, we can apply the method to a simple attribute, whose values are assigned in a 'not null unique' way without declaring this in the schema.

Run-Time Guide Attribute – RTGA

This technique stems from the homonym pre-scheduling technique.

At the beginning there is still pre-planning to assign initial values to GAs as seen in the GA approach. This time we set the GA values to belong to a specific range from 0 to an upper bound. Hence, we firstly compute the upper bound of the range as the size of the source table N divided by the size of an update L. Then, we randomly scatter each GA value over the table in such a way that all values appear L times.

Once this phase is completed, the algorithm that produces the run-time stream of updates starts. Firstly, it chooses the type of the next update to be performed among the three possible. The choice has to be made on the basis of a parameter that specifies the probability that the next update will be a true update, an insert or a delete.

The update statement to be executed is then completed with a where clause which includes, in the case of an update or delete, the GA value of the rows that have to be affected by the update or, in the case of an insert, the value of the key for the new row. In this way the distribution of the GA values along the table changes throughout the execution of an experiment due to the series of updates. Notice that update and delete statements are performed in a single transaction while inserts are carried out one by one. We claim these are realistic circumstances since a single update or a single delete usually affects a subset of rows, while an insert usually slots in one new row.

Given that, after each update, the algorithm keeps the set of available keys, the set of existing GA values and, for every key, its GA value up-to-date, it produces an infinite stream.

This run-time technique has a nice property. Since the distribution of GA values changes during execution due to the random choice of a GA value for a randomly selected type of update, it is possible that the updates affect a different number of rows each time, i.e. we can have changes of varying size. However, the average size of an update will be the specified parameter L. Consequently, the table spreads out and shrinks during the experiment around the initial size. We claim these are very realistic and therefore interesting dynamics.

We will present evidence of the dynamic behaviour of the system in subsequent sections.

5.1.11 THE QUERY STREAM

Simultaneously to the update stream to the source database the application has to simulate the stream of queries posed to the DW. The latter stream benefits from the same statistical properties as the update stream. Hence, we make use of the same algorithm employed to produce the Poisson distribution for the updates.

As far as the characterisation of queries is concerned, there is a need to clarify the role of a query inside the whole system.

First, the user is interested to know the time a query is posed and when the same is finally answered. Hence the application must keep track of this data.

Secondly, we need to decide the type of query to perform. If we bear in mind that the DW view instance is in main memory then any kind of query on the view has theoretically no processing cost to add to the system. Therefore, the model of the query we use could be a null query in the sense that we know when it is posed and answered but there is no actual statement for the query.

5.2 BENCHMARK CLASSES: INSTRUMENTATION

5.2.1 DIRECTOR

To mount all parts, the application has a director. The director must accomplish three main tasks: creation of the real instance of the system, management of the execution of an experiment and storing, computing and logging of the required data (staleness, RT,……).

In the light of the discussion so far, the creation of a real instance of the system means that the director is charged to create and fill only the source table. The class that provides all the methods to correctly perform this activity is called ‘Data generator’. We will discuss its content in the next subsection.

To accomplish the second task, the director has to match up the different entities needed to execute an experiment. Thus, it performs a number of actions for setting up, running and closing down a benchmarking session.

Firstly, it creates *instances* of all objects that will act together to model the system and to measure its performance. Notice that *only one* object of each class is instantiated (except for Row and Table). At the same time, an object is able to invoke methods belonging to other classes thanks to Java’s *call-by-reference*

parameter passing.

Then, the various threads are started in the proper order.

Finally, a method handles the shutting down of a test session: measures are computed and logged to a file and it takes care to stop all threads and with them the entire execution.

5.2.2 DATA GENERATOR

The generation of the initial set of data contained in the source table is one of the actions required to set up an experiment. The class that is charged with this task is named a 'DATA GENERATOR'.

How the data generator accomplishes its task is directly dependent on the technique we have adopted to apply the stream of updates. As a matter of fact, filling the table implies selecting values for each of the fields including the ones that have a special meaning for the update algorithm as discussed in the section dedicated to the stream of updates.

Basically, the input parameters are the initial size of the table, the DW view selectivity and the update size.

The initial size of the table is important. It dictates on the range within which we pick the primary keys to insert in the table. In order to avoid any possible kind of ordering or clustering, we set the range of possible primary key values to be significantly larger than the number of rows in the table.

The view selectivity parameter is important to distributing the value of the 'view attribute' (VA). If the view selectivity is 0.1 then we *randomly* arrange 10% of the inserted rows to have a special value of VA and therefore to belong to the DW view. The view will select only the rows with the special VA value.

Further, the data generator defines the range of values of the guide attribute (GA). The lower value of the range be zero; we need to compute only the upper bound of the range.

To do this, we have to consider that the update size parameter conveys the desired average number of rows affected by a generic update statement and that we want to scatter the rows affected by the same update statement along the whole table,

avoiding clustering.

The first step of our answer is to compute the maximum value of the GA range:

$$\text{MAX GA VAL} = \frac{\text{Size of the table}}{\text{Size of one update}} = \frac{\text{number of tuples in the table}}{\text{number of tuples affected by an update}}$$

Thus, the range is formally defined as [0…. MAX_GA_VALUE] and MAX_GA_VALUE gives also the size of the range (for the sake of precision the actual size is MAX_GA_VALUE +1).

To avoid clustering, when inserting a row we *randomly* choose a GA value in the range.

Thanks to the formula we have used, each value will occur in the table distributed about an expected value given by the specified update size parameter. Consequently, a generic update statement will involve rows with a specific GA value .The result is that an update will affect, on *average*, update size rows, as we desire.

5.2.3 THE METRIC

The metric we assume is composed of several measures: staleness, RT, source and warehouse processing costs, communication delay.

We compute staleness according to the definition given by the cost-model.

RT is the delay between the submission of the query and the result ‘on screen’.

For immediate or periodic policies the cost-model assumes this response delay to be zero, since the query should be satisfied using the DW view displayed at the moment the query is posed. However, we have found it useful to measure the real delay to answer a query to prove in point of fact that that delay is negligible.

An on-demand policy has instead a true RT because there is a delay due to retrieving data from the source database as soon as a query is asked. The whole process begins when a hypothetical user submits a query to the DW. The DW then needs the latest source data to refresh its view; hence the wrapper requests data from the source database and as soon they are received and processed by the wrapper, the latter propagates them to the DW completing the maintenance. The overall delay in the process is calculated as the sum of processing and communication costs.

While staleness and RT are measures related to a single query, the processing costs and the communication delay are related to the total maintenance cost of a policy. Our choice has been to refer to source and warehouse processing costs and communication delay as a whole-integrated cost. Computing the integrated cost is much easier and more precise than separately computing each contribution and then aggregating.

Even though the cost-model explicitly mentions source and warehouse storage costs we come to the decision of leaving them out of the scope of our benchmarks since they are related more to a system's hardware than to system performance.

Notice that evaluations based on benchmark results are still comparable to choices based on the cost-model formulas if we set the weights of storage costs that turn up in the selection algorithm to zero.

In the end, a number of data values are required for computing the required magnitudes: the time a query is posed, the time a query is satisfied together with the latest update packet used in its answer, and the time an update is committed.

These pieces of information are spread over different objects of the application and for the sake of clarity our choice has been to collect all of them into one single class object, the 'Director'. The classes that are able to recognize the relevant events notify the Director each time a specific event occurs. When a notification arrives, the Director stores the current time and, in the case of query satisfaction, the related packet number.

Our choice has been guided by consideration of future extension of the application. As a matter of fact, we have left open the possibility of introducing remote wrapping by means of Java RMI, where it is necessary to have one single object to keep track of the times since clocks on different machines are in general not synchronized. We claim that adding this feature to the application does not affect the final outcome given that centralizing the measurement involves only local communication as one machine locates all of the objects. Local communication entails negligible overhead in the considered case of local wrapping.

CHAPTER SIX

RUNNING THE BENCHMARK SOFTWARE

In this chapter we will present some evidence of how the benchmark software works. Our purpose is twofold: to show the use of the interface and to check that the tool actually behaves in accordance with the claims made for its design.

6.1 A GLANCE AT THE GUI

The user interface looks like a folder.

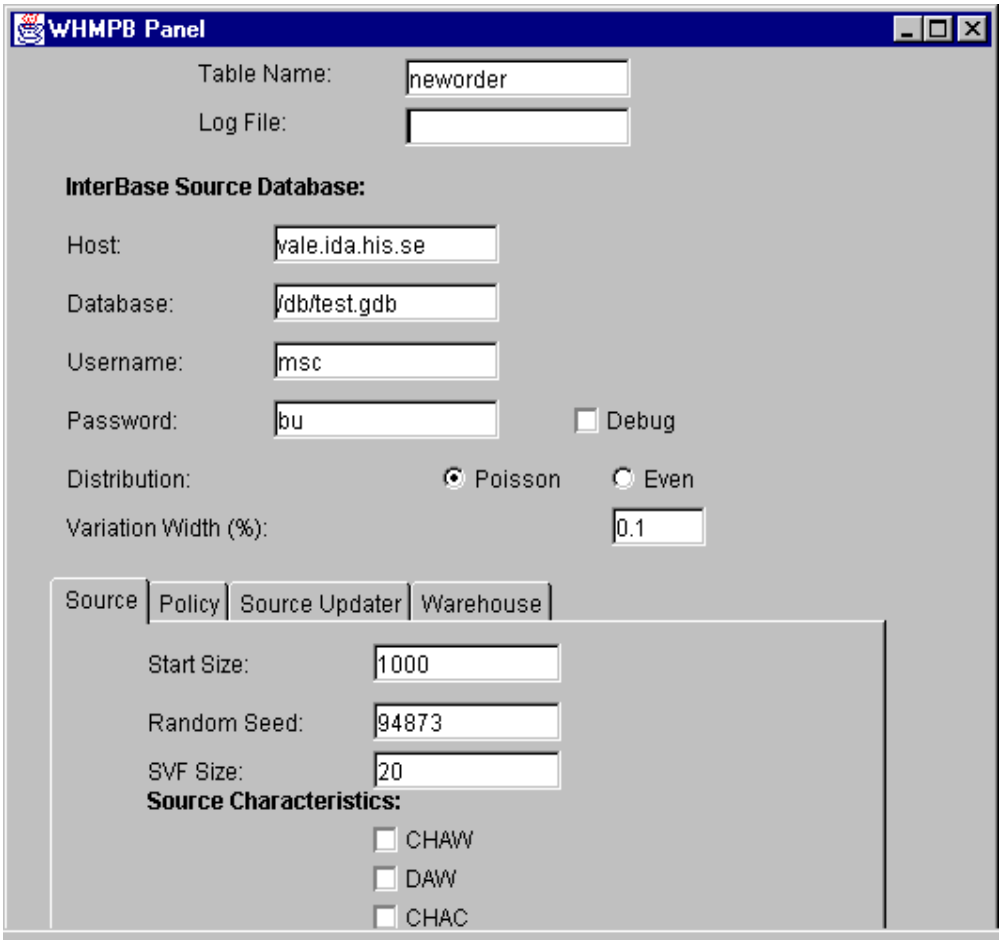


Fig.1 The look of the user interface

The upper part is used to specify the basic information required when starting or closing down a session of experiments. Hence, we find plain fields filled with the necessary data for opening a connection to the source database and logging outcomes to file. In addition, the user can request the debugger and specify which distribution should be used to simulate the stream of updates and queries. Some default values are defined, but users can specify their own paths for every field.

The lower part consists of four panels. Each panels relates to an entity modelled as a class in the benchmark software, whose behaviour is parameterised and needs to be set down by users. These entities are the source database, the source updater, the warehouse querier and the maintenance policies settled by the wrapper.

Notice that information concerning the DW is hidden both in the upper and lower part because we have chosen to have a "phantom" database on the DW side and therefore no parameters are vital.

The source database panel allows users to state the initial size of the source table and the random seed to generate its content. There are a number of check boxes for setting the source characteristics.

Figure 2 prints out the panel:

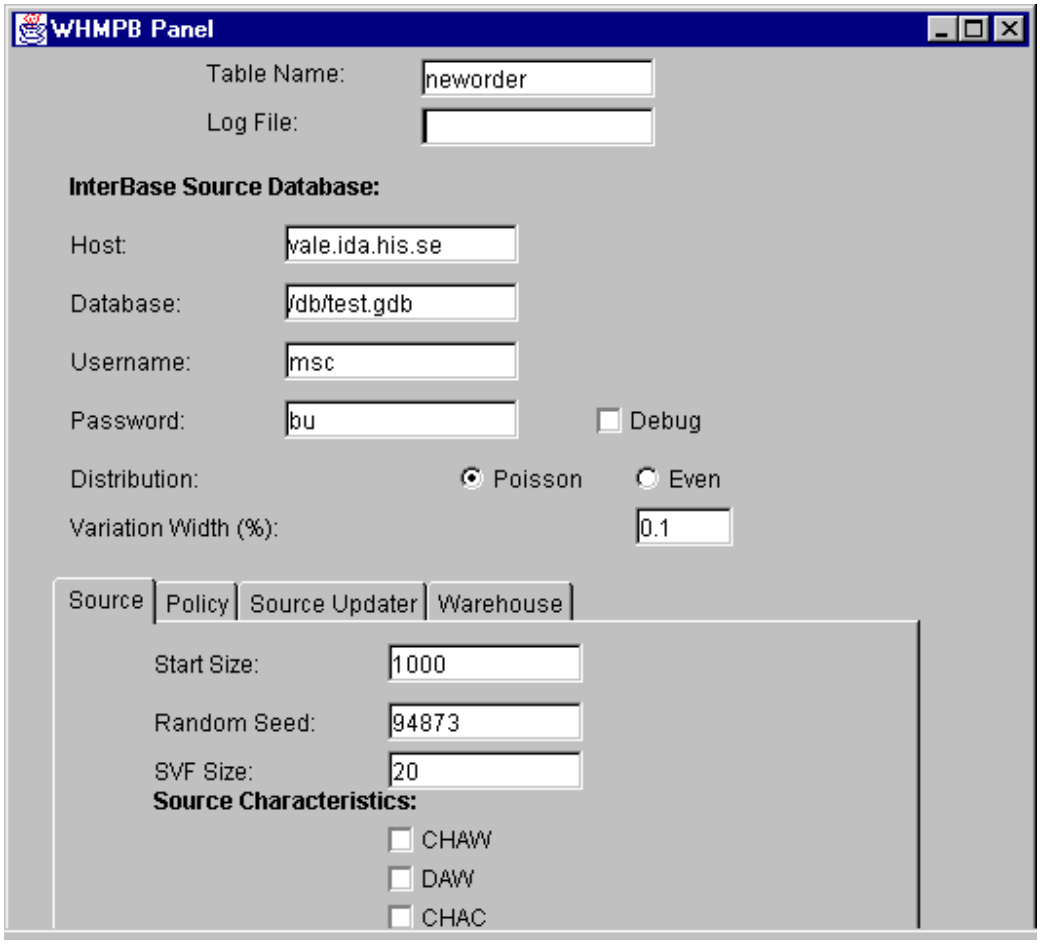


Fig.2. A source panel where all source abilities are checked

The source updater panel shows fields for giving a value to parameters referring to the update stream such as the update size, the mean delay between updates, and the percentage of updates, inserts and deletes as reported in Figure 3:

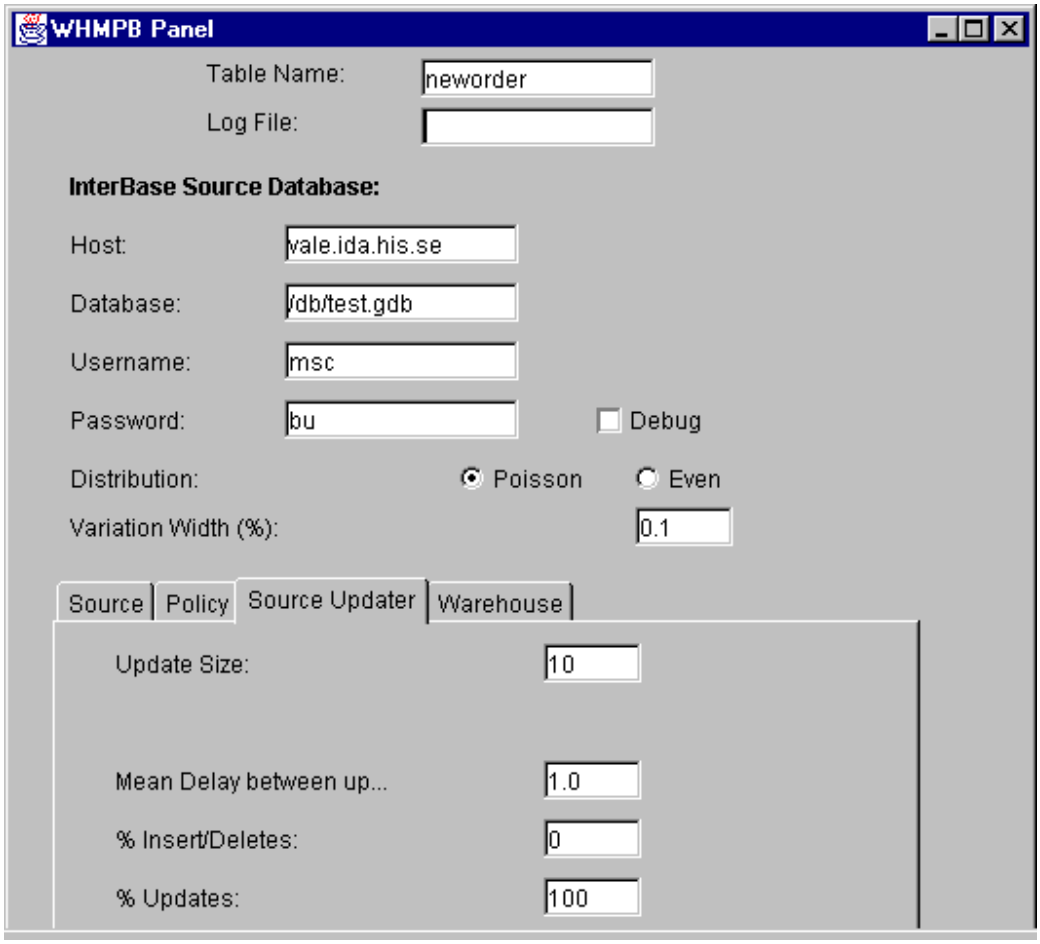


Fig.3. The panel for setting parameters of the source updater

The warehouse query panel is used to give guidelines for the stream of queries. Thus, we notice a box for specifying the mean delay between two queries, a box for the total number of queries and a final one for the selectivity of the DW view as shown in Figure 4:

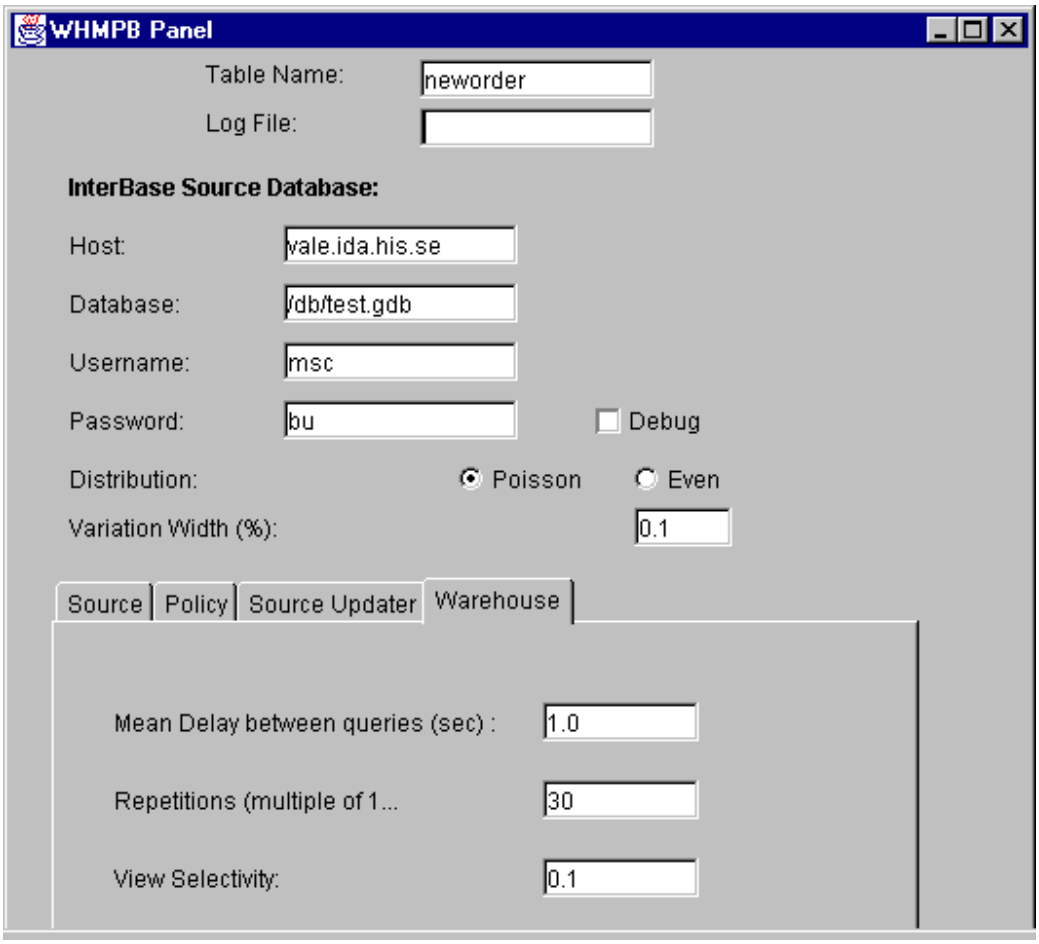


Fig. 4. The panel that shows the boxes for the DW's parameters

Finally, Figure 5 depicts the panel for selecting the maintenance policy to be used during an experiment:

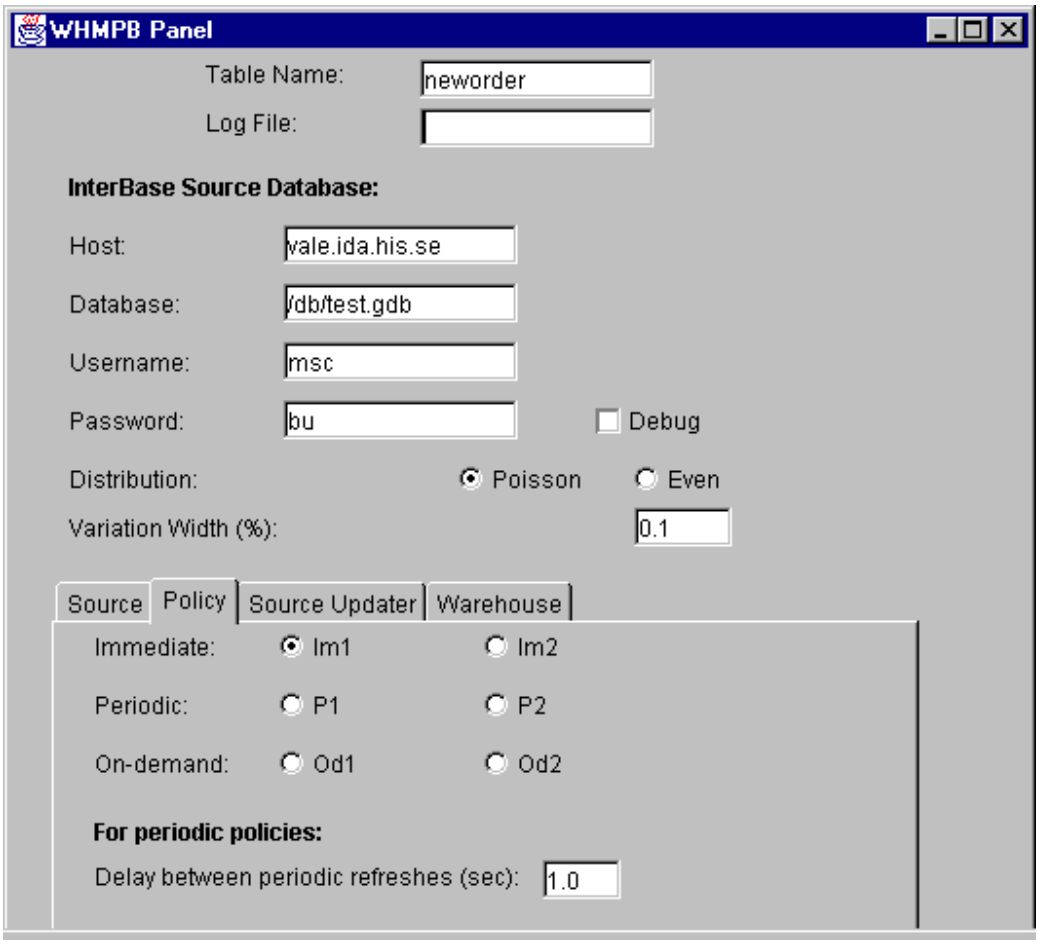


Fig. 5. The panel for selecting the policy. In case of periodic policies, users are allowed to give a range for the periodicity

In the upper part, there are some buttons to select one of the maintenance policies considered in the cost-model. In the lower part, there is a box to specify the delay between two successive refreshes of the DW view in the case the user chooses one of the two periodic policies.

6.2 EASY TO USE FEATURES

The GUI is helpful to get control of all parameters of the application. Nevertheless, after the user becomes familiar with the tool environment it could be that going through the GUI panels each time he has to perform an experiment turns out to be a repetitive and wasteful process.

Once the user is ready to use the environment in a more advanced way then he needs some more advanced facilities to set experiments. We have therefore made provision for file handling. The user can specify a configuration file that the application will parse in order to read the values assigned to the parameters.

Finding that in practice this was not flexible enough, we now also allow the user to specify parameters directly in the command line.

The result has been that the user can finally profit from a good combination of easy-to-use characteristics. He could specify an initial configuration file and then write a batch file to execute more experiments in sequence. It is even possible for those experiments to have a different configurations. This can be achieved in two ways: either we make the application to read from another configuration file or we simply specify only the changed parameters from the initial configuration.

Last but not least, we have added two more services:

a) the option of choosing whether the application should read the configuration and run an experiment immediately or whether it should read and put the GUI on screen;

b) the option of visualizing all the system output or only error messages.

We believe that these are important for a benchmarking environment since having the opportunity to strike out all possible elements that could affect the overhead during the execution has always been one of our goals.

CHAPTER SEVEN

VALIDATION

AND

VERIFICATION

In the benchmarking and simulation field, a benchmark environment must be carefully assessed before allowing anyone to employ it for test purposes. Precisely, there are two aspects to this: validation and verification.

7.1 VALIDATION

Validation aims to attest that the model of the system we assume is a correct representation of the system itself. In our case, this is straightforward since the model of the system has been taken from the cost-model. Transferring it to our application has been a natural action as we have simply adopted the existing model.

7.2 VERIFICATION

Verification is to establish whether the benchmark software operates according to its specification.

Consequently, we have developed a suite of tests to verify the conduct of the application in a few, key aspects. These relate to the handling of maintenance policies, concurrency, the simulation of updates and the computation of measures.

7.2.1 THE SUITE OF "SANITY" CHECKS

The suite consists of two types of "sanity" checks.

A first type of "sanity" check has been intended to test the protocols of the tool throughout the execution of an experiment, i.e. it concerns how the algorithms of the application work in terms of the sequences of operations they execute under given conditions. The most comprehensive method for conducting these tests is to make the several components of the application log their progress throughout the execution of an experiment.

This sort of check has been useful not only in analysing the final state of the application but in progressively controlling the development of the software when existing parts have had to be modified or new parts have had to be added.

A second type of "sanity" check has been instead intended to test the behaviour of the system under load, i.e. as a proof of integrity that concerns the effects of the applied algorithms on the system such as, in our case, the changes occurring to the source database or the computed measures.

Typically, purpose-built tools monitor what is happening inside the system, reporting a numerical analysis of the observed magnitudes, frequently supported by plots.

Ideally, the combination of all “sanity” checks helps us to verify all key aspects we can identify and helps to give a clear picture of what the application is actually doing and how it is carrying out its tasks. Once this picture is complete and in a way satisfactory, the benchmark software can be regarded as suitable and safe for use. The ultimate purpose of the whole verification process is to give evidence that nothing really suspicious is happening throughout the execution of an experiment.

7.2.2 THE PROTOCOLS

The first set of “sanity” checks has been primarily intended to try out how maintenance policies and source characteristics are handled and whether the application computes staleness in the correct way. In a second stage, we have added some sanity checks to control other parts of the application such as initialization, the Poisson inter-arrival time and the way it keeps track of the relevant instants of the execution.

Our inspection has been made possible thanks to messages printed on screen. A message lets the user know the class which is sending it and the reason it is sent. A line on the screen shows the name of the class and the action that class is performing. In this way, we can follow the succession of actions undertaken throughout the execution of an experiment.

7.2.2.1 SOURCE CHARACTERISTICS

The application implements all three source capabilities.

While CHAC is compulsory to put into effect immediate policies, CHAW and DAW directly affect the sequence of actions demanded when applying a policy. In this section, we show the role that CHAW and DAW play when carrying out a policy. The importance of CHAC will come out in section 5.2.2.2.

Chaw

It is important to know whether changes have been committed since the latest DW view’s refresh. Two cases are possible: if no changes have occurred then there is no need to update the DW view as shown below in Figure 1:

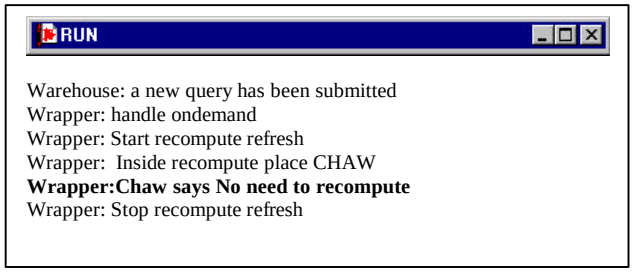


Fig.1. CHAW means maintenance can be avoided if the source database hasn't been updated

since the last time the DW view reflected its source.

(Example taken from an on-demand recompute policy)

Otherwise, if change has occurred then we start the maintenance (Figure 2):

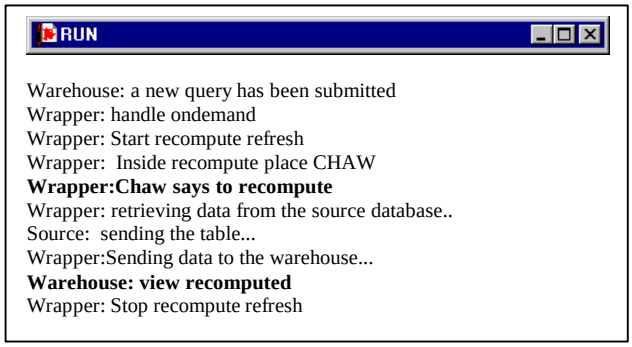


Fig.2. CHAW means maintenance will start if changes have occurred.

(Example taken from an on–demand recompute policy)

Daw

Delta awareness is useful with incremental policies.

If the ability is set then the DW has to retrieve the DAW table instead of the source table; otherwise, the wrapper must compute delta changes each time the DW view needs to be refreshed. A DAW table is in either case propagated to the DW. Figures 3 and 4 report the behaviour in both cases:

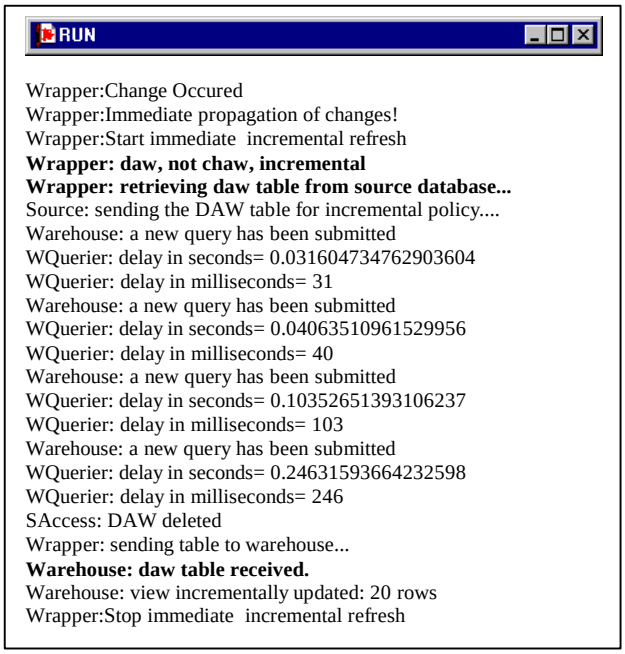


Fig.3. If the source is DAW, the DAW table is sent when maintenance is needed.

(Example taken from an immediate incremental policy)

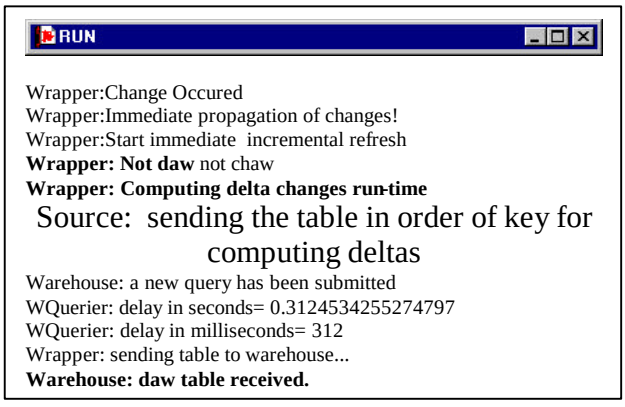


Fig. 4. If the source is not DAW then delta changes must be computed at run-time.

(Example taken from an immediate incremental policy)

7.2.2.2 MAINTENANCE TIMING

The application is able to handle all six maintenance policies, resulting from the combination of three maintenance timings (immediate, periodic and on-demand) and two methods of refreshing (incremental and recompute).

All of them need to be verified as the maintenance policies embody one of the major features of the application.

Immediate

An immediate policy reports source changes to the DW as soon as they are committed.

In order to put into effect this type of policy we have made provision for the mechanism used for change activeness. As a matter of fact, we can read in the first line of the following excerpt that the source updater notifies the wrapper whenever an update to the source is committed. From this moment on, a maintenance loop fires going through the stages of the selected policy according to the settled source characteristics.

This behaviour is shown in Figure 5:

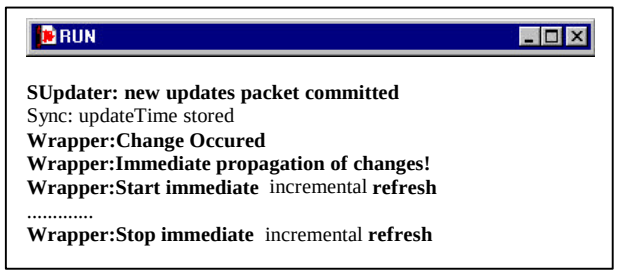


Fig.5.CHAC allows activation of immediate maintenance.

Periodic

A periodic policy performs maintenance on a regular basis.

The sequence of actions to be completed should be as follows: the wrapper tells when a new maintenance loop has to fire and according to the chosen policy, data are retrieved from the source database and sent to the DW, where the view will be either incrementally updated or recomputed. Figure 6 shows the output:

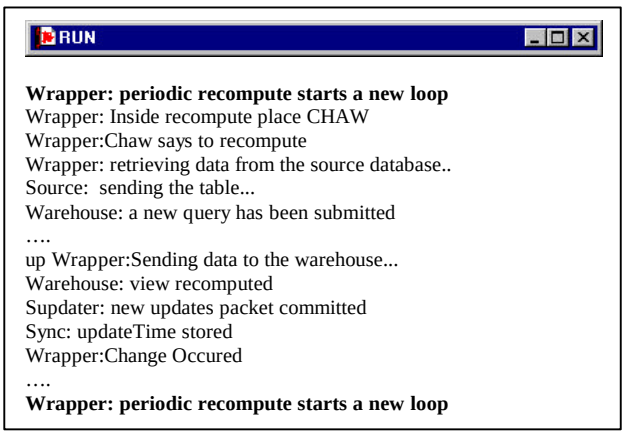


Fig.6. A periodic policy starts maintenance on a regular basis.

On-demand

An on-demand policy retrieves the latest changes from the source when a query is submitted to the DW. As soon as the DW receives a query, it notifies the wrapper and a new maintenance loop starts as shown below:

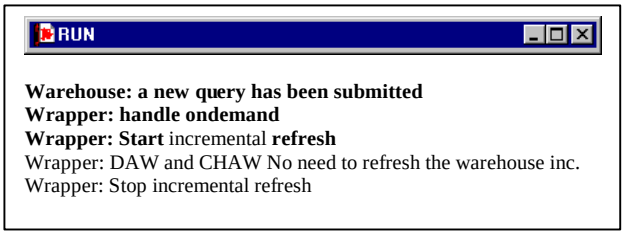


Fig. 7. The wrapper handles maintenance whena new query is posed to the DW.

7.2.2.3 WAYS OF PERFORMING MAINTENANCE

Incremental

An incremental refresh affects only the modified rows of a view.

The number and type of actions to be performed entirely depends on source capabilities: CHAW could tell that no refresh is needed, DAW helps to recognize delta-changes without computing them at run-time.

Two probing examples are reported in Figure 8 and Figure 9:

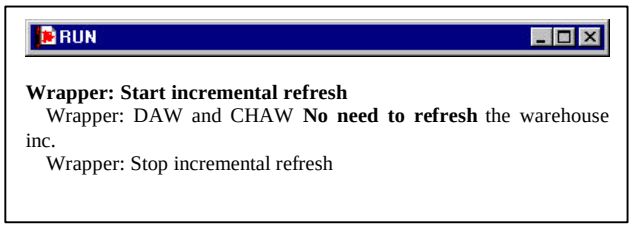


Fig.8. Case of CHAW when no refresh is required.

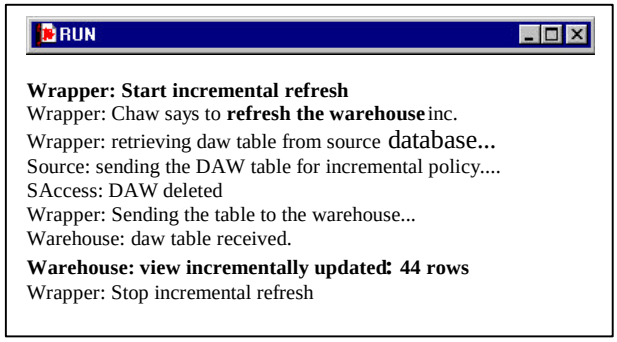


Fig.9. Case of CHAW when maintenance has to be started.
Here DAW helps with delta changes.

Recompute

A recompute policy implies that when the need to refresh arises, we recompute the whole view each time.

If CHAW is selected and no changes occurred since the last recompute then no action has to be undertaken, as previously stated for incremental policies.

An example of a recompute policy is reported in Figure 10:

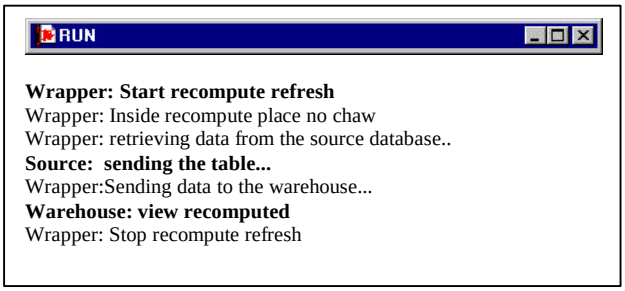


Fig.10. The wrapper retrieves the whole table from the source and sends the whole table to the DW which recomputes the view.

7.2.2.4 OTHER IMPORTANT CHECKS

Initialisation

The first step is to initialise all components necessary for an experiment: one object of each class is instantiated and all settings are laid out, as shown below:

Fig. 11. Actions completed to get ready for experiments.

Poisson distributions

As motivated in the previous chapter, source updating and DW querying are simulated as Poisson processes. Effectively the formula we use to compute inter-arrival times between two updates or queries give as outcome a series of varying delays as we can see in the following sequences:

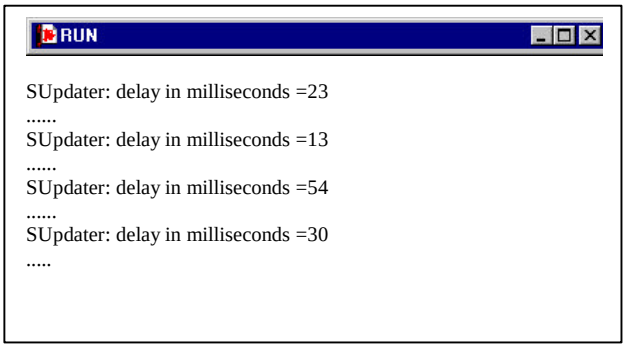


Fig. 12. Example of inter-arrival times between two updates.

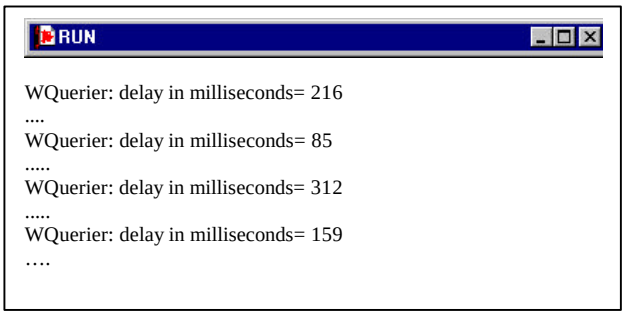


Fig. 13. Example of inter-arrival times between two queries.

Log File

A log file stores both the conditions under which an experiment took place and the outcomes. It is useful to control the regularity of an execution (for instance, the actual length of an experiment and the total number of committed updates) and whether errors occurred.

For the sake of clarity, the log file organizes the necessary data to have a clear picture of the series of values we ought to consider to compute measures (in particular staleness). Thanks to this procedure, we have been able to compute them independently and to compare the outcomes with the results given by the application itself.

Figure 14 shows how we have organized the data:

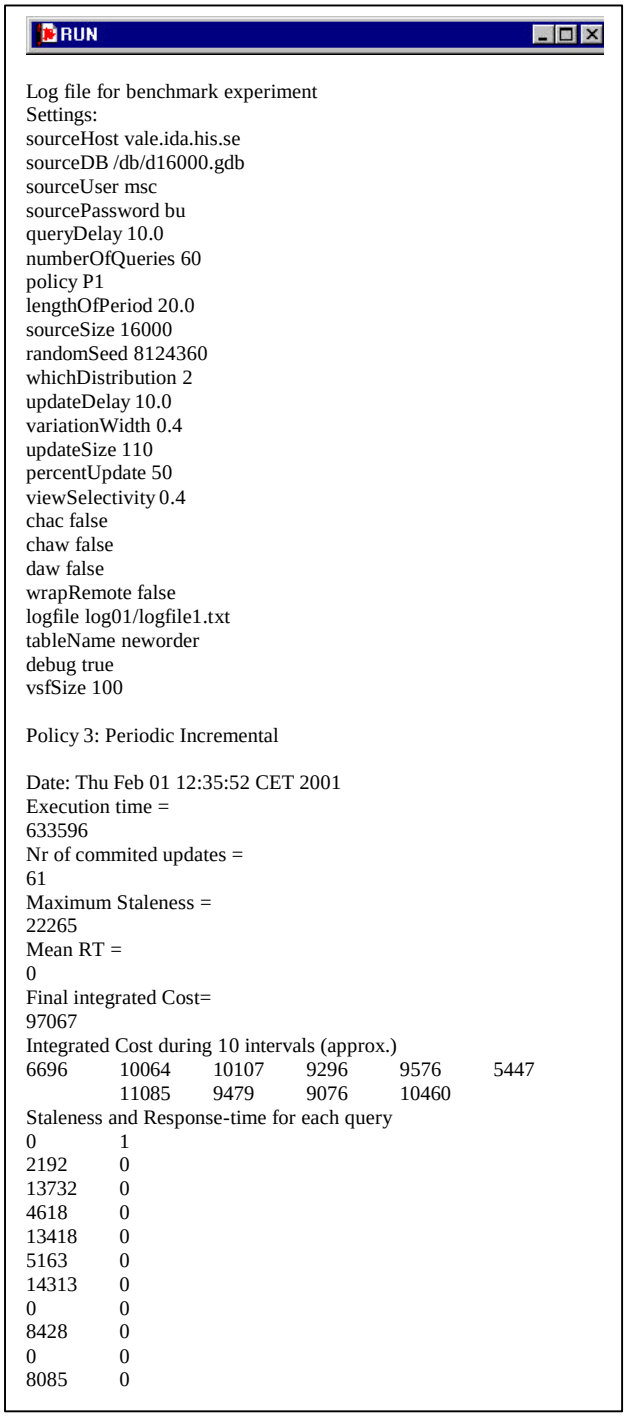


Fig. 14 The content of the log file helps to control the regularity of an execution

7.2.3 BEHAVIOUR UNDER LOAD

While the “sanity” checks concerning the protocols allow us to verify the behaviour of the software, sanity checks concerning the system under load are based on observing the effects of the actions of the software upon the system. The set of tests we have designed has therefore focused on aspects like the workload and the computation of measures.

7.2.3.1 THE STREAM OF UPDATES

One of the most relevant and discussed parts of the benchmark software has been the application of the stream of updates. We have already identified a number of feasible alternatives that could have been applied. Our choice fell on the technique called ‘run-time GA’ that potentially offers an infinite stream of updates.

The algorithms have been designed to have appropriate statistical properties. However, we still believed it necessary to conduct appropriate sanity checks. Hence we monitored the state of the source table as it was updated.

Two elements came out to be relevant in monitoring the behaviour of our algorithm and the actual effects on the source table:
1) The distribution of GA values;
2) The size of the table.

The former is a hint of the quality of the statistical properties of the algorithm.

As a matter of fact, the distribution of GA values is obtained from all other data the algorithm considers and since it has been the main property we have looked for, it is the measure to observe.

In particular, two plots have been of interest: the number of rows with a given GA value and the number of GA values that have a specific cardinality.

The plot that illustrates the number of rows with a given GA value is constructive to evaluate the fairness of the algorithm, i.e. whether it is favouring a certain grouping of the values in an identifiable sub-range or homogeneously spreads the values across the whole range. This is a key property since the GA is used as a guide for choosing the rows to be affected by a generic update statement.

The plot that shows the number of GA values with a specific cardinality is significant to complete the view over the statistical properties of the algorithm. From it, we can gain an idea of the density of the distribution, i.e. where it is most common to find GA values.

The size of the table reflects the net result of all activities of the update algorithms. The property to be preserved is that, given an equal percentage of insert and delete transactions, size is allowed to vary up and down from the initial expected value and the overall average size should be equal to the expected value.

We need then a tool to read those pieces of information and to plot them. The tool will access the source database and collect the required data by querying the source table. The two main queries have to find out how many rows there are at the time in the table and, for each of those rows, its GA value. These queries are asked on a regular basis specifying a polling frequency.

Once again we have availed ourselves of a Java tool that uses JDBC to read the desired data from the source, and canvases to plot diagrams.

Notice that using this Java tool simultaneously with the benchmark software we do not really affect the phenomenon we wish to analyse. The influence the checking tool exerts on the benchmarking environment could impinge on a number of aspects (the measures the application is charged to compute). In fact, no influence is exerted upon the key points of the algorithm such as the choice of the kind of update and the choice of which rows that update should affect.

We have chosen to perform our analysis observing the state of affairs at important point during an execution; more precisely we have singled out four junctures.

The first one concerns the initial table size and distribution. What we are probing is how the application works in generating a new table, i.e. how the data generator fills the rows in a new table and which values turn up in the special field GA.

Scrutinising how the data generator fills the table is strictly related to the technique we utilise to produce the stream of updates as the choice of values to insert into the initial table is driven by the specific technique we have decided to apply.

Figure 15 shows the initial situation:

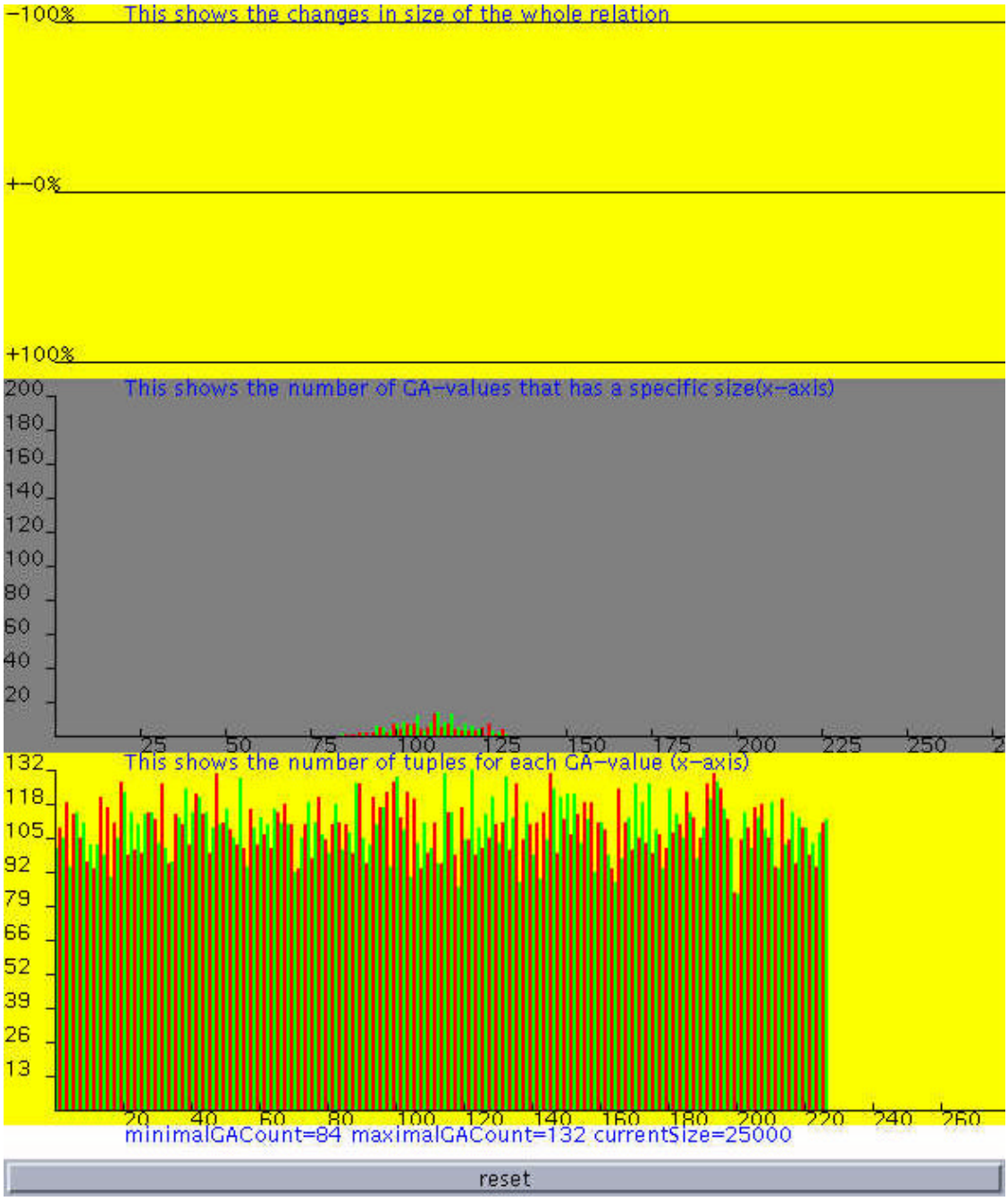


Fig. 15. The initial distribution of GA values given by the data generator. (Example with a table size of 25000 rows and update size of 110 rows)

The most interesting indicator is the distribution of the GA values before the source updater is started. We can detect whether the data generator actually scatters the GA values along the computed range and whether the distribution is homogeneous over the individual values. The middle plot shows how the concentration of GA values has the typical shape of the normal distribution with mean approximately equal to update size (110 in this case).

A second indicator worth mentioning is whether the size of the table has significantly enlarged in comparison with its initial value. Figure 16 depicts the data logged for a specific experiment:

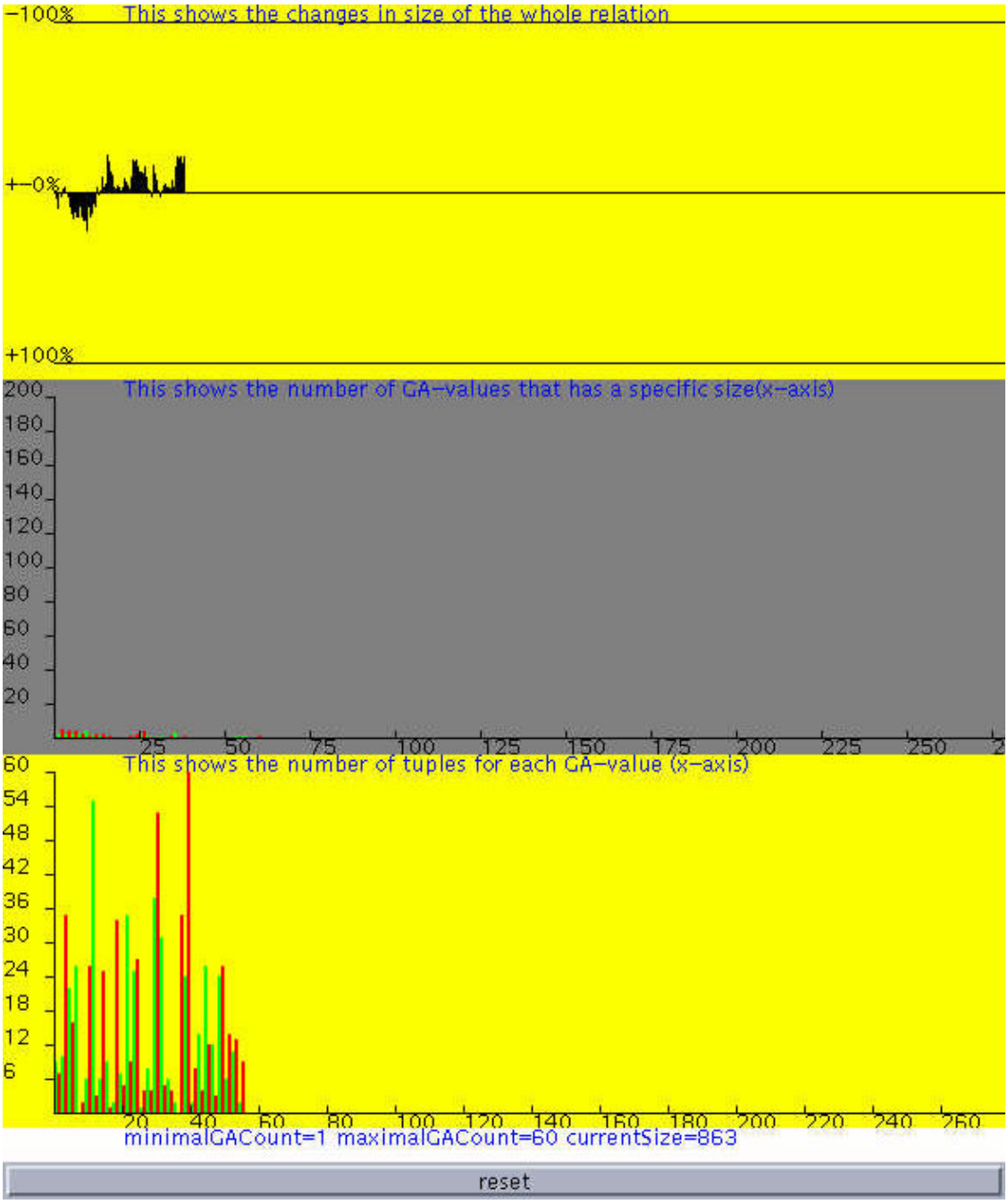


Fig.16. A screen dump when the table size is smaller than at the beginning. (Example taken from an experiment with table size 1100 rows and update size 20 rows)

The upper plot shows that we are in a phase where the table size is smaller than the initial one assumed as the normal level.

The central and lower plots are evidence for the distribution of the GA values and precisely show that most of the GA values turn up a small number of times. More precisely, most of the values are less than the specified update size (20 rows in this case). The two charts provide two different perspectives of the same phenomenon and are therefore complementary.

From the central chart we can remark that the bars are grouped towards the very left part of the x-axis: lots of GA values turn up only a few times.

From the lower chart we can inspect the actual distribution of those GA values and remark that there is no evidence of bias.

This fact is fully in accordance with what we have forecast for such a situation. As the table shrinks the possibility that a delete affects a number of rows smaller than the specified average size of an update should strengthen while an insert always adds a number of rows equal to the specified average size of an update. In this way, as the table becomes smaller and smaller it is more and more probable that the table is likely to revisit its initial value.

The third notable stage during an execution is attained when the size of the table has significantly increased in comparison to its initial value. Figure 17 illustrates this situation:

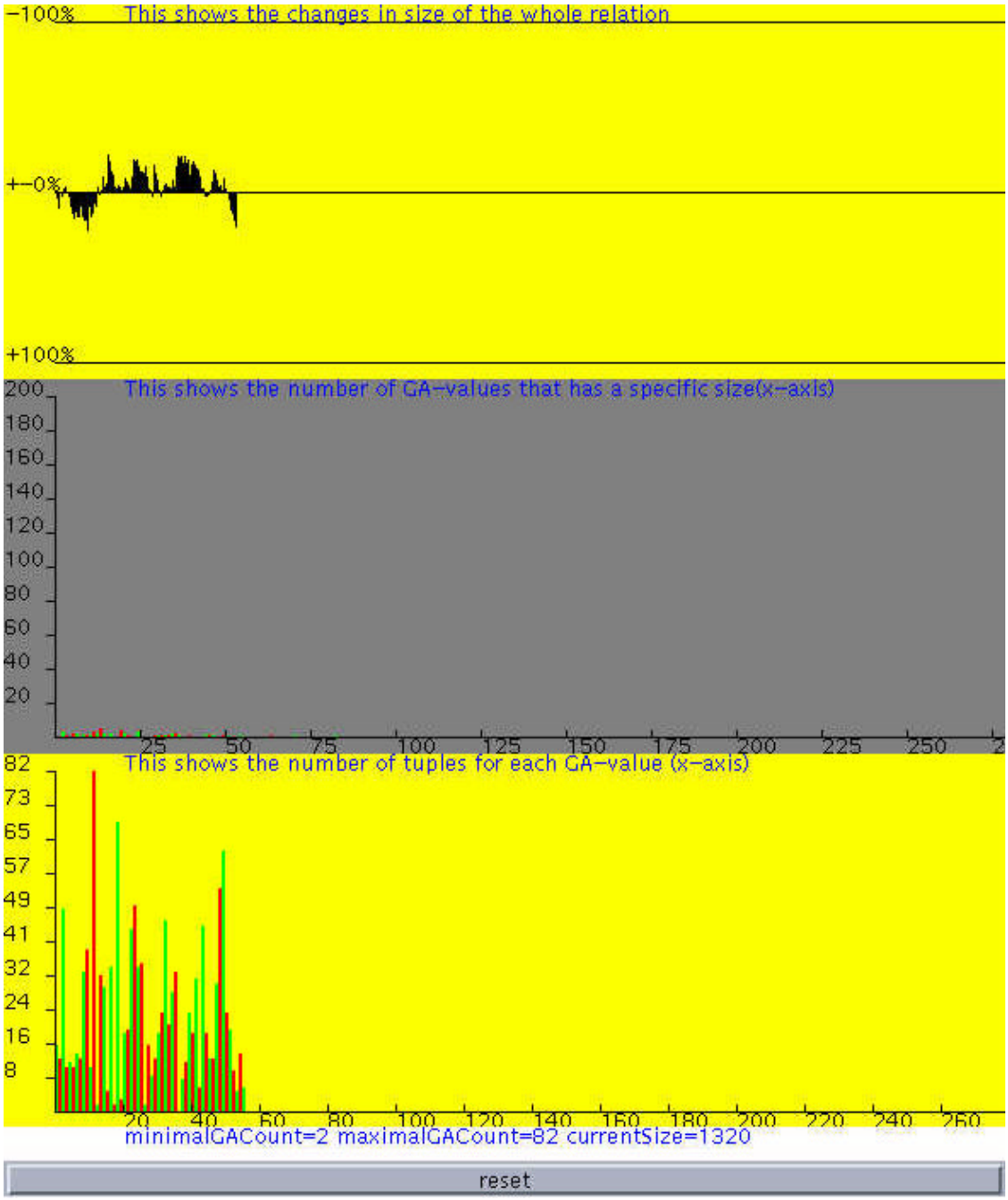


Fig.17. A screen dump when the size of the table is bigger than its initial value.
(Example taken from an experiment with table size 1100 rows
and update size 20 rows)

As before the first plot shows we are in the case of a bigger table than the starting one.

The other two plots show that the distribution of the GA values has changed: most of the values turn up higher than the specified update size (20 rows in this case).

In the central chart, the bars are now spread around the mark 25 and not only towards the very left part of the x-axis. This means that many more GA-values appear with higher than the average frequency in the table.

The lower chart again shows the actual distribution of GA values and we can have the clear perception that on average the bars are higher than the ones plotted when the size was smaller.

All these facts are once again in accordance with our predictions.

As the source table swells, there is a higher probability of having a delete hitting a number of rows greater than the specified average size of an update. In this way, as the table increases it is more and more probable that the table is likely to revisit its initial value.

Another situation worthy of note situation is when during an execution the table achieves the same size as it had at the beginning. Figure 18 illustrates this situation:

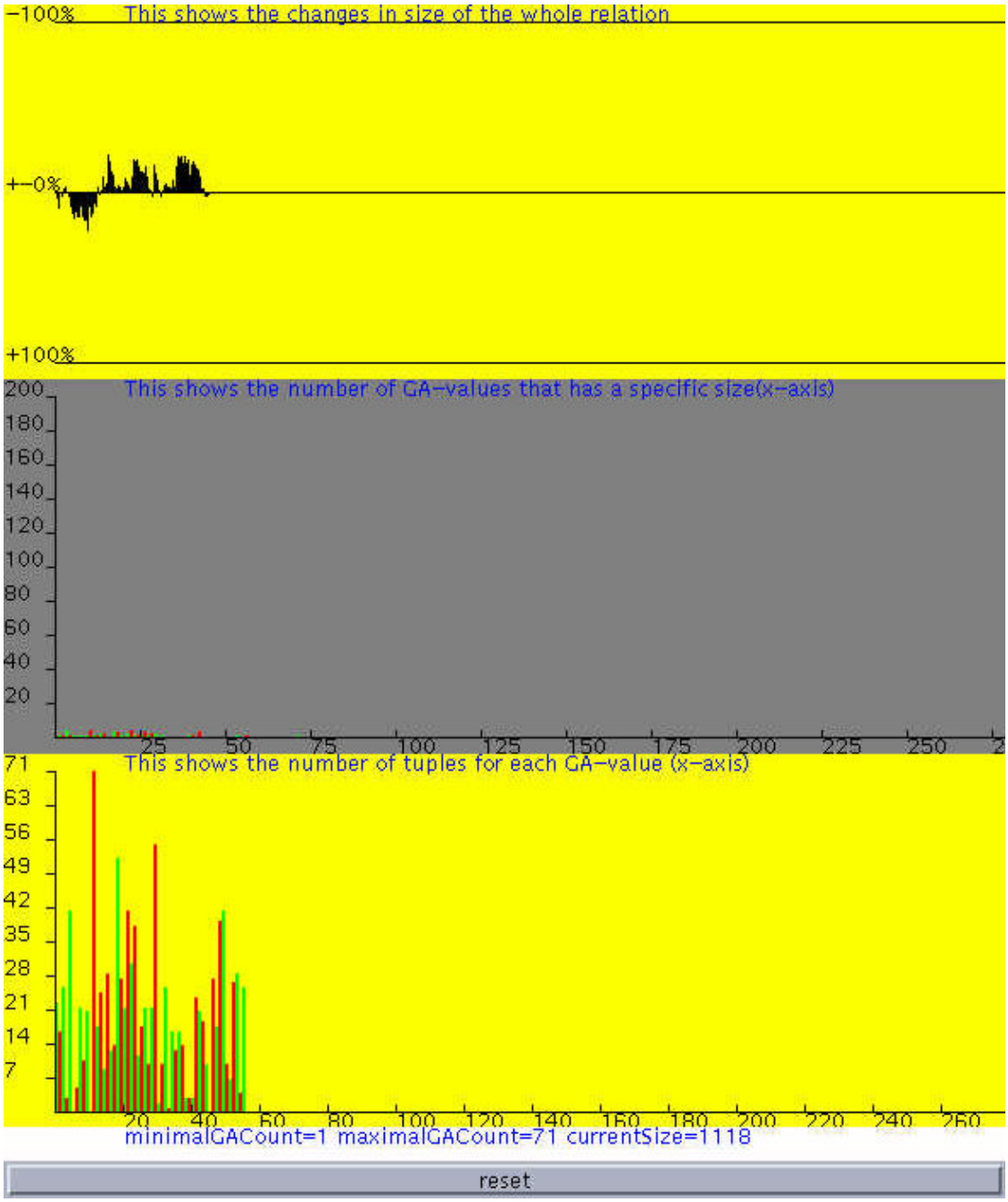


Fig.18: The dump of the screen at a point when the size has returned to its initial value.
(Example taken from an experiment with table size 1100 rows and update size 20 rows)

The purpose is to check whether the distribution of the GA value is similar to the initial one where it was built in an artificial way by the data generator. It should present a distribution in the middle of the previous two cases. On average, each value should turn up update size times (20 in this case).

We can notice that the central and lower diagrams agree with our expectations. Together they indicate that the distribution is again more or less homogeneous (note that the scale of the chart adapts to the maximum value) and balanced.

In the light of this and other examples, the technique we have implemented seems to have two nice properties:

1) Phases where the table shrinks and phases where the table swells seem to alternate following a periodic wave;

2) The variation of the size of the table is always naturally kept within limits. Effectively it looks improbable that the size of the table will double or degenerate to a very small value.

7.2.3.2 IS AN EXPERIMENT REPEATABLE?

The application utilizes the random generation of numbers to a large extent.

All the random generators are created starting from the same seed. The reason lies in that we want an experiment to be repeatable: two tests whose configuration is exactly the same including the seed should give the same outcomes.

The sanity check for this property is quite simple. There is a need to run some experiments and analyse the difference between the values of the measures we receive as outcome. Table1 reports the integrated cost and the staleness of two experiments with exactly the same configuration together with the variation between the outcomes of the two tests:

	IC (sec)	Z (sec)	Variation (sec)
Test 1	401.16	21.307	0.01047
Test 2	405.36	20.641	-0.03126

Table 1. The variation between the outcomes of two experiments with exactly the same configuration

As we can see, the variation is very small and therefore negligible.

7.2.3.3 THE INTEGRATED COST

We have defined the integrated cost as the sum of the source processing cost, the warehouse processing cost and the communication cost. It is an overall measure related to an experiment and it rises each time the maintenance activity produces a new refresh of the DW view.

Given a set of experiments based almost on the same scenario but varying only in some particular parameters, the integrated cost should display certain properties.

First, given a fixed scenario, we expect the integrated cost to increase as the size of the table swells and consequently the amount of data to process and propagate soars. More precisely, we expect the relationship between the size of the table and the integrated cost to be linear: the larger the amount of data the higher the cost to handle them. Linearity is required since the application should always perform the same actions on different numbers of rows without incurring any particular overheads.

To evaluate the truth of our expectations, we have performed an analysis on the values of the magnitude Integrated Cost as stored to the log file. On the basis of this knowledge, we have then plotted graphs to verify the shape of the curve. Below, we report one table with the configuration we have used and the relative values for the integrated cost and the subsequent final graph:

DAW	LENGTH (sec)	Policy	TABLE SIZE (rows)	IC (sec)	STALENESS (sec)
FALSE	20	P2	2000	152.437	19.509
FALSE	20	P2	4000	184.66	20.838
FALSE	20	P2	8000	244.52	20.33
FALSE	20	P2	16000	401.16	21.307

Fig.19. The linearity of integrated cost against table size

We claim that the relationship can be seen as linear.

Further, we expect the Integrated Cost to have an express connection with the maintenance period when a periodic policy has been chosen.

Given all other parameters fixed, the expectation is that the IC has to rise if we have a lower periodicity as refreshes to the DW view will be performed.

For the sake of simplicity, we have applied the sanity check assuming a periodic recompute policy. In addition, we have assumed that the source does not exhibit any of the source characteristics the cost-model has defined. This means that, on a regular basis, the wrapper always retrieves the whole table even if no changes to the source table have been committed.

Under these conditions, the magnitude of the IC should solely depend on the periodicity that directly determines how many times during an experiment the maintenance is performed.

The result of the sanity check is shown in Table 2 and plotted in Figure 20:

DAW	LENGTH (sec)	IC (sec)	Policy	TABLE SIZE (rows)
FALSE	10	800.464	P2	16000
FALSE	20	401.16	P2	16000
FALSE	40	203.98	P2	16000
FALSE	60	140.41	P2	16000

Table 2. The Integrated Cost when the length of maintenance period increases

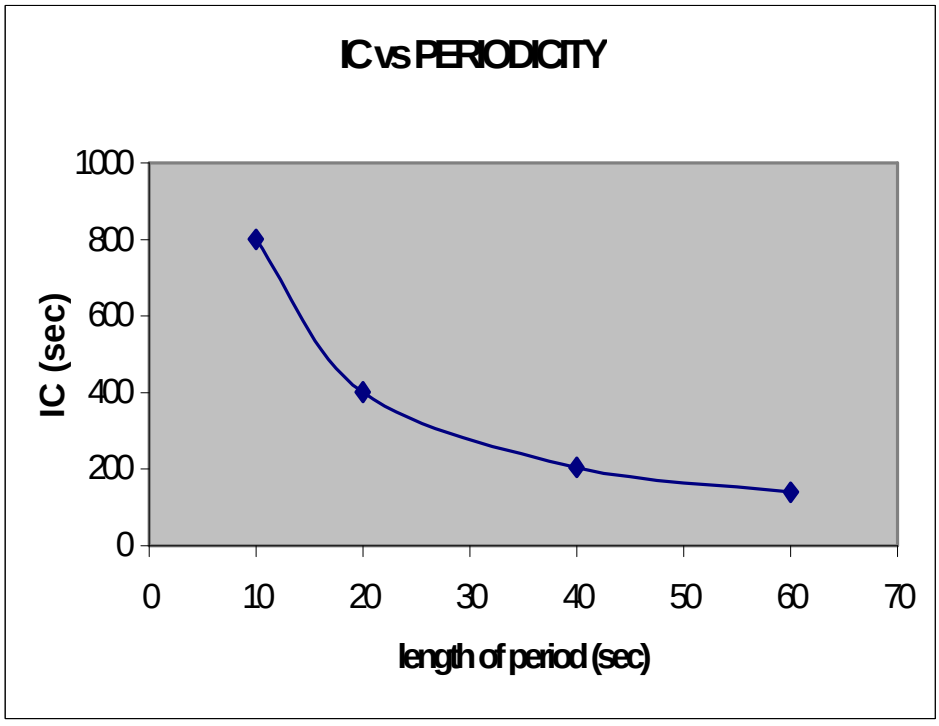


Fig.20. The inverse proportionality graphically translates the property we were looking for

7.2.3.4 STALENESS

Staleness is a measure that deserves attention.

We can apply the same consideration as we have just done for the IC in the case of a periodic policy. Under the same conditions, we expect the staleness to worsen (higher value) as the delay between refreshes spreads. This is due to the fact that if the wrapper refreshes the DW view after a longer period of time then for a longer period of time data that are becoming older and older are used to answer queries.

Essentially, we expect a linear correspondence between periodicity and staleness, as the result of our sanity check can show:

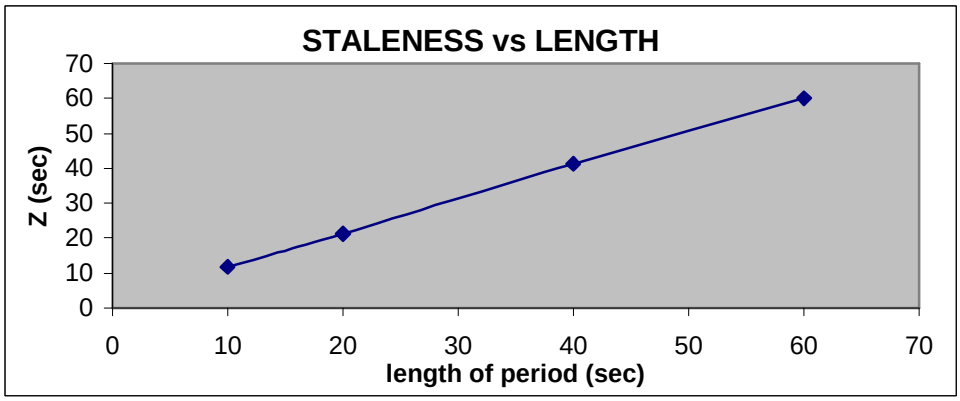


Fig.21. The linearity of staleness against the length of the period of a periodic policy.

The example is taken from the same sanity check used for the IC.

7.3 CONCLUSION ABOUT THE “SANITY” CHECKS

The set of “sanity” checks concerning the protocols has been completely successful.

In practice, their role has extended beyond the final verification of the actions of the application; they have in fact been very useful throughout the whole drafting of the code.

In particular, we have found them an invaluable instrument for reviewing concurrency, as we have been able to follow on screen how the different processes of the system could interfere with each other.

The set of “sanity” checks concerning the behaviour under load has given precious indications about the workload and the dynamics of the system, summed up in the measures. Those indications have been fully in line with our predictions. Finally, we claim that the set of tests has successfully proved the properties of the benchmark environment, which can now be used with confidence for investigating the validity of the cost-model.

CHAPTER EIGHT

APPROACHING BENCHMARKS

In this chapter we present how we have intended to approach a benchmarking session and we report a few examples of benchmarking to show how the environment could be used to test the cost-model, which is the ultimate purpose behind this work.

Approaching a benchmarking session we need to address a number of points concerning the hardware and software environment in which tests are to take place. Many have already been addressed, but for complete transparency, a few more points should be covered in order to allow arbitrators to make an objective assessment of the results.

Then, a few example runs are presented to demonstrate how users can choose significant scenarios and test the cost-model.

The purpose has been to introduce the intended use of the benchmark tool rather than to assess the overall behaviour of the cost-model. A systematic analysis is beyond the scope of this work.

8.1 PLATFORM

The application has been developed in Java and it is therefore possible to run it on any suitable platform. The only constraint is its use of Interbase as the DBMS; the handling of the connection and the syntax of the update and query statements are hard-coded. However, the use of JDBC allows the specific DBMS in use to be changed, and all DBMS-specific code is kept to one class.

Our choice has been to use a Sun Microsystems Sparc 5 with 128Mbytes RAM, running under Solaris 8. Apart from InterBase 6.0 (server and client), JDK 2.1 and our application no other software has been running on top of Solaris 8; no other users were granted access to the system.

8.2 TUNING LOW-LEVEL PARAMETERS

The installation of the DBMS has brought to our attention the importance of some low-level parameters related to the configuration of the kernel of Interbase. In particular, we claim a disclosure should be made on the parameter block size. It is very important to set the DBMS' block size to the same value as the operating system's block size. The consistency among these parameters assures that the two worlds will use the same unit when interacting and communicating.
In our case, the value assigned to both parameters has been 64kb.

A similar consistency should be kept for the parameters of the prototype the authors have developed to compute the cost of different policies with the cost-model formulas. As we want to compare the outcomes coming from tests on the real system with the outcomes computed by the prototype, the two should take into consideration the same scenarios and therefore the consistency of parameters such as the block size, the velocity of the disk drive and others must be kept under control. Thus, we have exactly mirrored the parameters of the operating system in the prototype.

8.3 DEALING WITH OUTCOMES

A benchmark is intended either to assess the working of a system or to investigate its properties. No matter what a benchmark is intended for, the ultimate purpose is to provide information in order to critically assess aspects of the performance of a system.

For the sake of fairness and correctness, those observations should stem from the evaluation of outcomes which can be regarded as truthfully representative of the scenario we are examining. Any criticism that comes from results that have not been carefully analysed does not constitute a defensible proof.

Thus, the quality of outcomes should always be inspected. As far as our tasks are concerned, we have tried to address the issue of whether a run can be said to give a representative picture of the characteristic behaviour of the system.

Much has been done to reduce the probe effect of the benchmarking environment. Further, the sanity checks have demonstrated that a single experiment is repeatable and that the results of the application are seed independent. Hence, it is possible to repeat experiments and check that captured behaviour is representative rather than unusual.

We have respected the following list of criteria to guarantee the integrity of an experiment:

1) consistency with the outcome of similar tests (concerns the probe effect);
2) absence of reported errors related to any part of an execution: the error file should be empty, meaning that none of the stages to set up and execute an experiment failed;
3) absence of overlapping updates or queries during the application of the streams: during the application of a stream it could happen that the time required to carry out the activities to commit an update or to answer a query is more than the delay till the next update or query. In this case, the simulation of the stream is artificially disturbed. Nevertheless, in particularly demanding configurations some overlap is inescapable and tolerable. Thus, the constraint is weakened to an insistence that the error file should report few overlapping updates or queries.

8.4 EXAMPLE RUNS

The goal has been to investigate example scenarios to demonstrate how we may analyse some aspects of the behaviour of the cost-model. We have therefore spent some time on devising a suitable set of configurations to test. The key has been to look for interesting cost-model predictions by utilising the available tool (*DWMP*[1]).

8.4.1 INCREMENTAL VERSUS RECOMPUTE: THE ROLE OF *DAW*

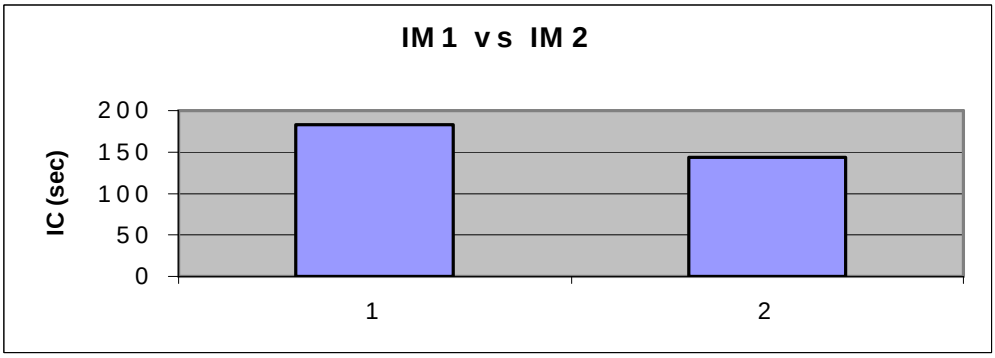
Incremental and recompute represent the two ways in which the DW view can be refreshed: the former selects only changes to the source database and propagates those to the DW while the latter always retrieves the whole table and replaces the old DW view.

On the basis of our understanding of the cost-model, we have found it informative to compare the costs of these two methods in terms of *integrated cost*.

In a first set of tests, we have taken a base-level case of a source with none of the specified capabilities. This leads to a straightforward examination of the performance of an incremental and recompute policy for a given scenario.

Based on the cost-model, recompute policies are expected to be more efficient than incremental ones for any given configuration of parameters under these conditions.

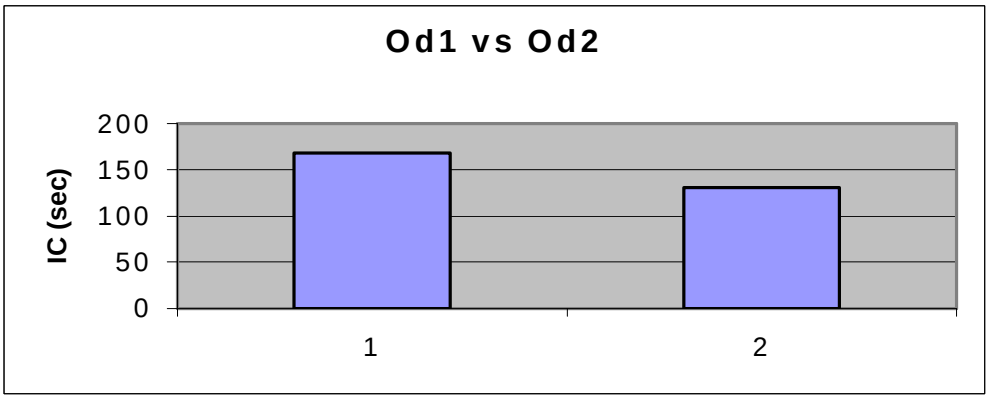
Table 1 and Table 2 show the outcomes for an immediate policy and an on-demand policy, giving details of the configuration parameters used.



DAW	IC (sec)	Z (sec)	POLICY	TABLE SIZE (rows)
FALSE	182.873	3.416	IM1	16000
FALSE	143.754	2.811	IM2	16000

Table 1. Comparing the integrated cost of two *immediate* policies.

[1] The tool can be run from: http://www.his.se/ida/research/groups/dbtg/demos/dwmp/



DAW	IC (sec)	Z (sec)	POLICY	TABLE SIZE (rows)
FALSE	167.817	2.268	OD1	16000
FALSE	130.299	1.5	OD2	16000

Table 2. Comparing the integrated cost of two *on-demand* policies.

The prediction has been confirmed in these tests.

The analysis was then extended to investigate how the introduction of source capabilities could affect the cost of policies. More precisely, we have drawn our attention to the case in which the incremental policy benefits from the DAW capability and the recompute operates with no advantageous source capability.

Apparently, the advantage of having a source with DAW capability is relevant as it entails that only delta changes have to be propagated rather than the whole source.

In order to have direct control of experiments we have chosen to build a scenario with periodic policies; we could then exert full control over the number of times maintenance is performed. This fact has been relevant, as the analysis has focused on integrated cost - which increases each time a refresh of the DW view is completed.

The configuration tested had the main settings shown in Table 3.

CHAW	DAW	POLICY	TABLE SIZE (rows)
FALSE	FALSE	P2	16000
FALSE	TRUE	P1	16000

Table 3. Configuration settings for checking periodic with DAW

We ran a group of tests varying only the frequency of refreshes. The outcomes are shown in Table 4.

POLICY	DAW	LENGTH OF PERIOD (sec)	IC (sec)
P1	TRUE	10	53.098
P1	TRUE	20	38.634
P1	TRUE	40	27.512
P1	TRUE	60	26.000
P2	FALSE	10	133.410
P2	FALSE	20	66.86
P2	FALSE	40	33.996
P2	FALSE	60	23.402

Table 4. Performance: periodic with and without DAW

On the whole our set of tests agreed with expectations, but one outcome was unexpected. From Table 4 we see that the incremental policy performed worse than the recompute policy when the period was set to 60 seconds. All tests with a higher frequency showed incremental to be better than recompute for the given scenario.

A graph helps to shed light on this piece of evidence:

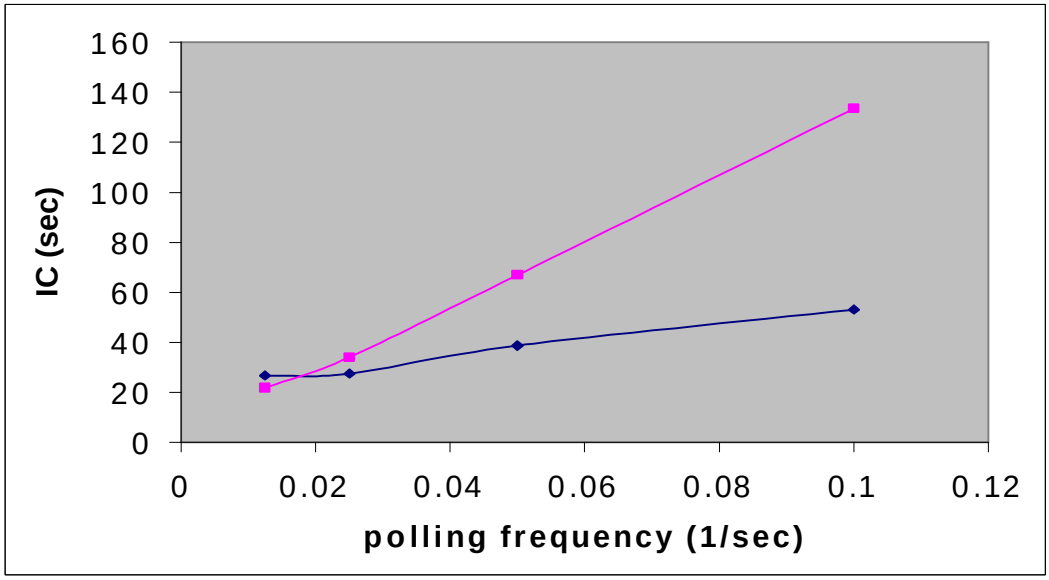


Fig.1 Plot of integrated cost against polling frequency for P1 and P2 (P1 blue, P2 purple)

When maintenance becomes less frequent, the cost related to the recompute policy gradually draws nearer to the cost of the incremental policy. Finally, under a certain frequency there is a crossover and the recompute policy performs better than the incremental one.

At first, this was considered to be behaviour which needed explanation. However, when examining the cost-model itself using the DWMP tool (see Figure 2), this 'unexpected' behaviour turned out to have been predicted.

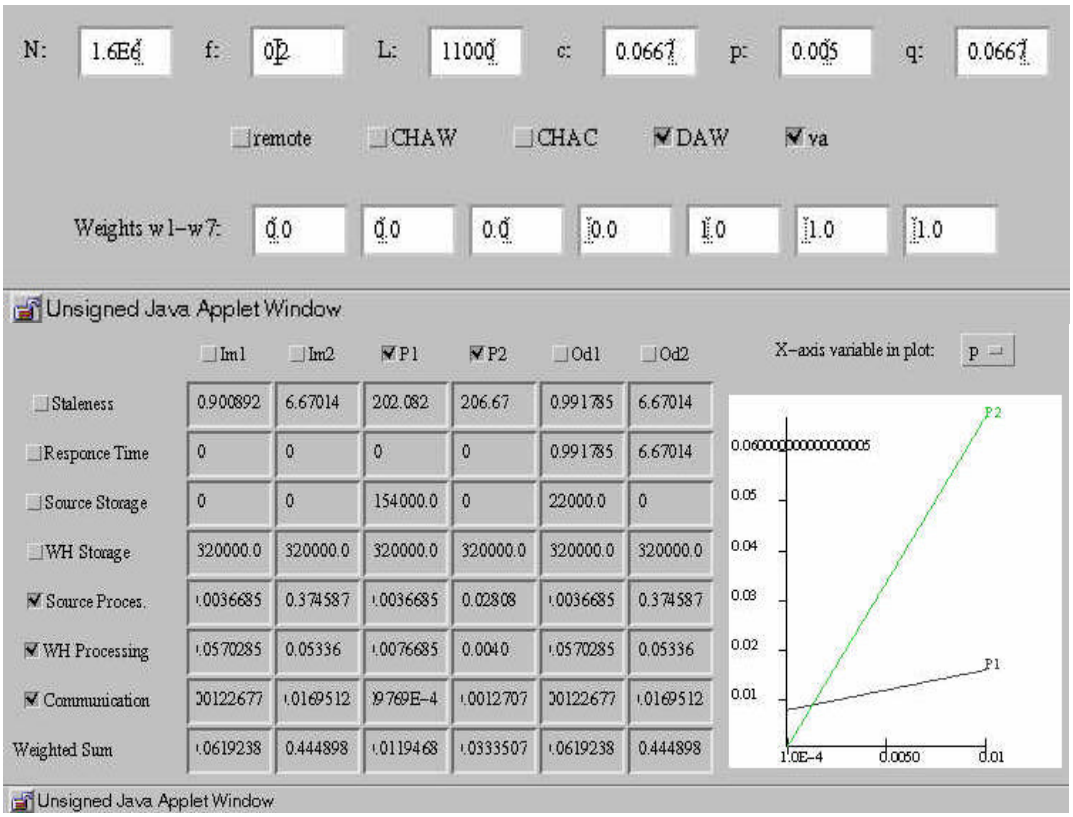


Fig.2. The prototype's dialog box that displays the parameters and the graph

8.4.2 COMPARING TWO POLICIES: IMMEDIATE VERSUS ON-DEMAND

A typical activity that a designer would like to do is to compare the behaviours of two policies in order to understand which one should be chosen under definite conditions.

Our choice for this task was made with the aid of DWMP. The two policies belong to the incremental set and concern two different timings: immediate and on-demand.

Our investigation could have been conducted considering different perspectives, i.e. taking into consideration different measures. The choice made was to consider staleness and integrated cost for their significance inside the cost-model.

The graphs shown in Figure 3 represent the outcomes obtained from the cost-model formulas.

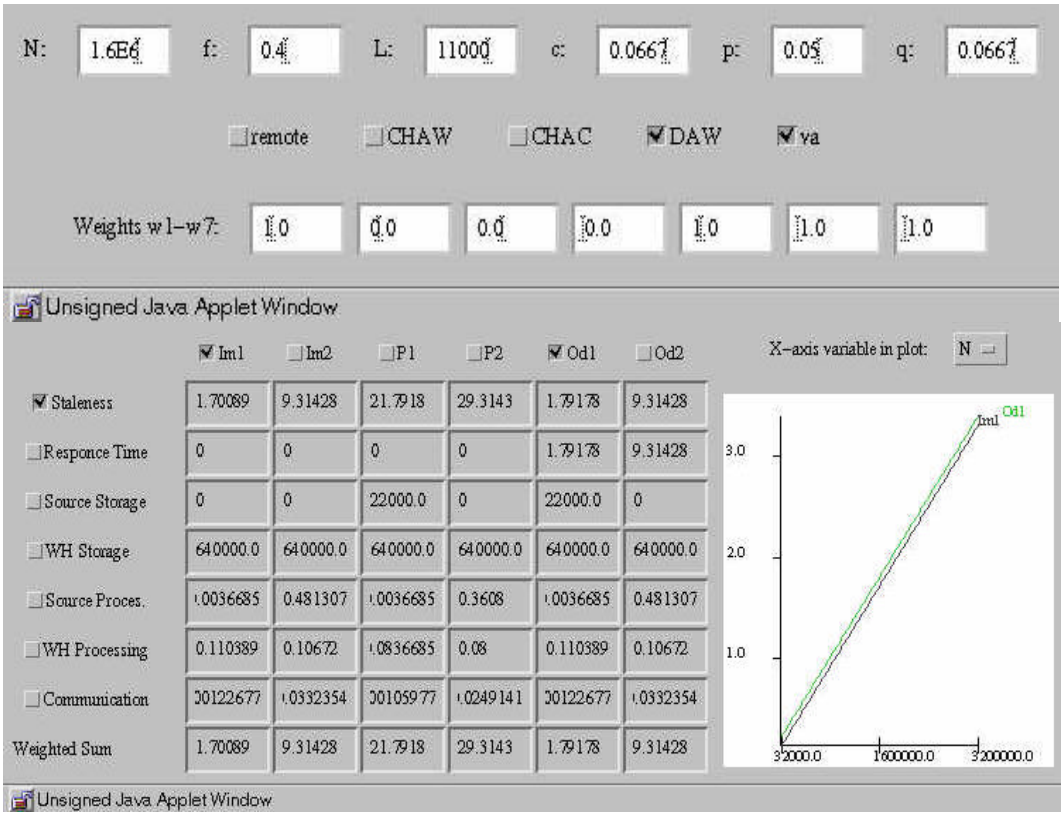


Fig.3. The comparison of the two policies using staleness

As far as staleness is concerned, the on-demand policy holds a small but useful advantage over the immediate one. According to the cost-model, this can be accounted for as follows: the on-demand timing retrieves the latest data from the source each time a query has to be satisfied and therefore the data have a high degree of freshness. On the other hand, the immediate timing could deliver answers to queries that are relatively old, depending on the gap in time between updates and queries.

In our case, the difference between the two magnitudes is rather close due to the choice of an equal frequency for updates and queries. This proves that even in extreme cases on-demand has an advantage over immediate.

As illustrated in Figure 4, integrated cost is not a discriminating factor.

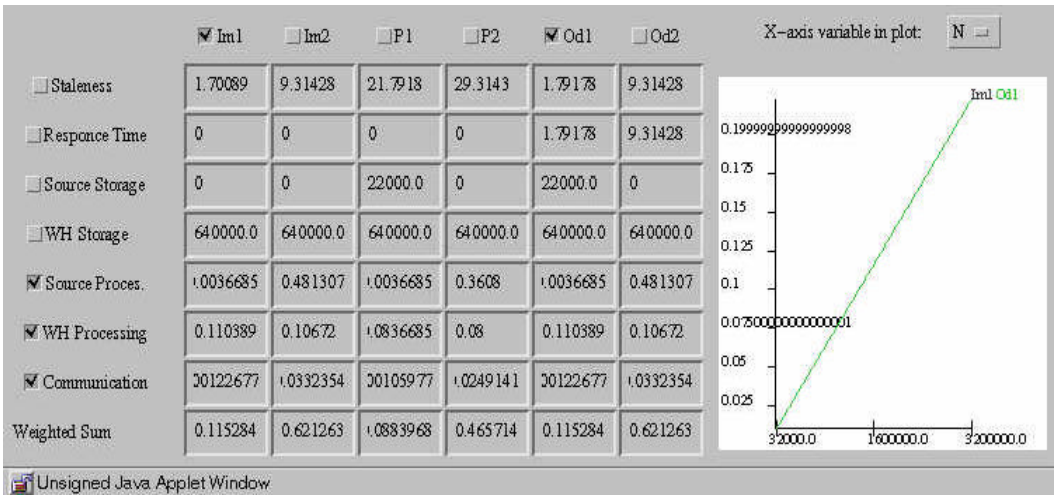


Fig.4. Comparison of IM1 and OD1: predicted

The two plotted lines are on top of each other. This is consistent with the choice of retaining the same frequency for the updates and the queries. As a matter of fact, given that the experiments have the same length, on average the same number of refreshes to the DW view has to be completed for both policies giving on average the same costs due to source processing, communication and warehouse processing.

The results given by the set of tests based on a matching configuration is reported in Table 5.

DAW	IC (sec)	Z (sec)	Table Size (rows)	Policy
TRUE	289.1	1.148	16000	OD1
TRUE	586.97	1.825	16000	IM1

Table 5. Comparison of IM1 against OD1: empirical

Note that the variation in staleness between the two cases is small, the better value being obtained for the on-demand policy.

Against the forecast, integrated cost is likely to play a significant role in the choice of best performance. Once again, on-demand policy gives the better performance, and this time by far. As a matter of fact, the cost related to the immediate policy is about double that of its competitor. Several similar experiments have been conducted and all have testified to divergence from the cost-model.

A possible explanation could be found in the way the DBMS applies updates. An immediate policy by definition tries to retrieve the source data, either from the source table or from the DAW table, each time an update is committed. Whatever the type of update, it usually affects numerous rows of the table and the common procedure is to execute all of the statements required to complete the update and only then to commit all changes to the source table and the DAW table. Accordingly, an immediate policy always tries to retrieve the data that have just been committed.

However, the cost-model assumes that it is always feasible to immediately retrieve data as soon as a request is raised. It seems that in actuality one perhaps should account for an additional delay: that caused by queuing a query whilst waiting for the updates to be committed. This yields a larger integrated cost, as there is a hold-up between the submission of the query to retrieve data and the beginning of the search for those data.

An on-demand policy, on the other hand, is not affected by this issue. As we have designed it, the stream of queries is independent of the stream of updates, leading to two unsynchronized processes. It is hence unlikely that maintenance will fire exactly at the moment updates are about to be committed. This situation could happen, but with a low probability.

At this point, a researcher would need to investigate this issue further. For this dissertation, we simply note it as potentially useful evidence in the process of testing the cost-model and questioning its assumptions.

8.4.3 RANKING POLICIES

A natural broadening of the comparison between two policies is to compare the whole set of policies considered by the cost-model. The final aim is to rank them and get an overall view of the benefits and drawbacks that the selection of each policy brings about.

We have chosen the scenario of Table 6 by exploring the cost-model through DWMP.

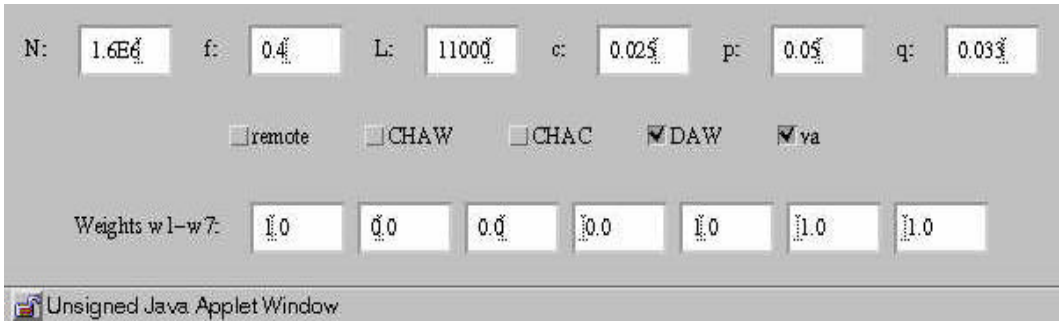
DAW	Update Frequency (1/sec)	Query Frequency (1/sec)
TRUE	0.0667	0.033

Table 6. Chosen configuration for exploring all policies

Since policies can be ranked according to many criteria, we simply make a number of independent observations about the results obtained.

Observation about staleness

The details dialog reported below shows a graph where staleness has been plotted against size of source table:



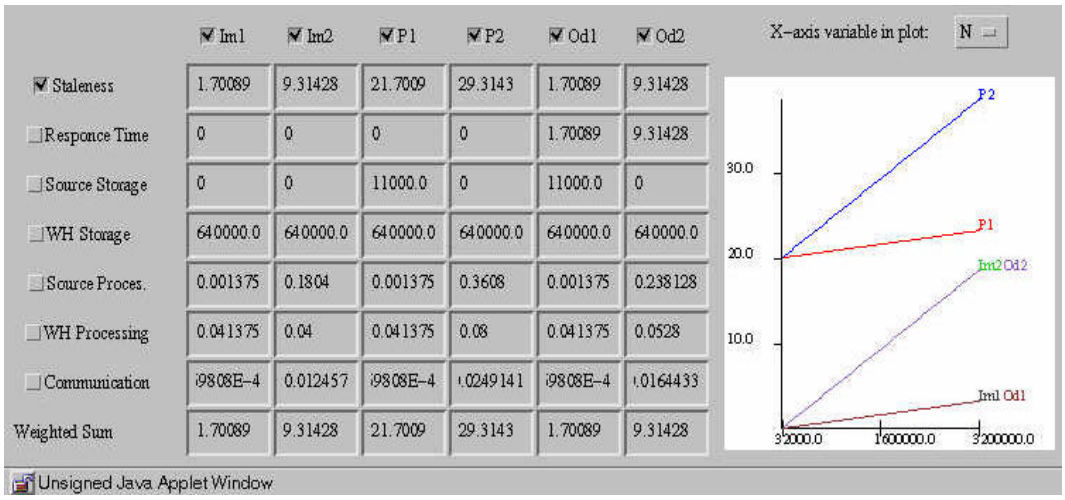


Fig.5. The plot that orders the policies against the staleness

We can read that for the size we have used (16000), the expected rankings are:

1. Im1
2. Od1
3. Im2
4. Od2
5. P1
6. P2

The small variation with empirical results (see Table 7) again largely concerns immediate policies.

Policy	Z (sec)
Od1	0.4
Im1	1.2
Od2	2.1
Im2	3.3
P2 10	9.4
P1 10	9.5
P1 20	12.1
P2 20	19
P1 60	56.3
P2 60	56.7
P1 180	163
P2 180	163

Table 7. The policies in order of increasing staleness - empirical.

Observation about polling frequency

The examination of staleness leads to another interesting discussion, this time concerning periodic policies and their polling frequency. From Table 7 it is clear that we have run a set of experiments using those policies, with a different periods. Staleness has been plotted against polling frequency in Figure 6.
:

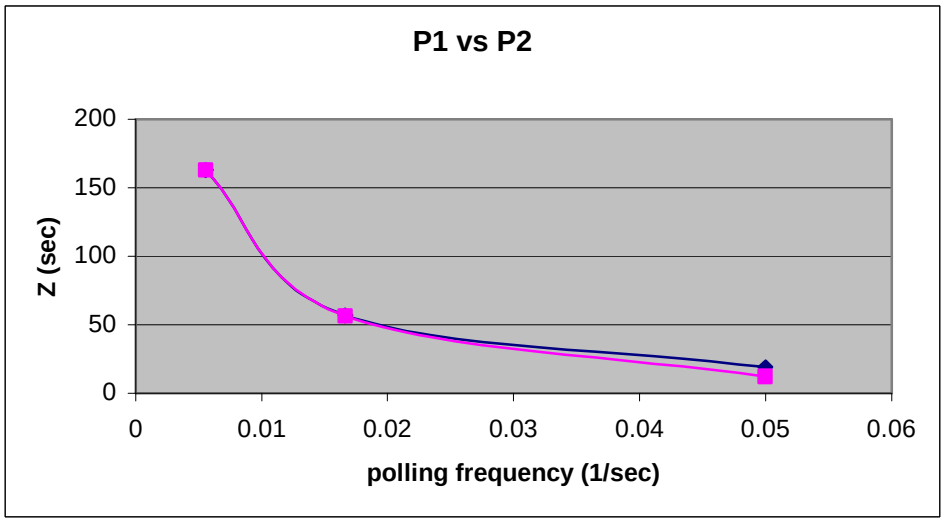


Fig.6. Staleness against polling frequency (P1 purple, P2 blue).

The first feature to note is that there is a similarity between the shape of the expected graph, plotted by DWMP, and the graph based on the outcomes of our tests. This is a further proof of the inverse relationship between staleness against frequency of periodic policy as discussed when dealing with "sanity" checks.

The shape of the two curves reveals one detail: the curves diverge as polling frequency increases. In fact, the incremental policy has a perceptible advantage at high frequencies. This can be explained by looking at the way in which the DW is refreshed, as follows.

As the data have to be retrieved from the source on a frequent basis, the incremental policy propagates to the DW the delta changes that have occurred in a narrow period of time while the recompute policy always propagates the whole table. Thus, the number of rows sent to incrementally refresh the DW view is likely to be smaller then the number of rows in the relation. The difference is translated in terms of the delay required to process and transmit the information. The incremental approach being faster in executing maintenance, the DW finally satisfies the queries with fresher data in comparison with the recompute policy.

For long refresh periods, the incremental policy has to retrieve a larger number of rows as the DAW table keeps track of the updates that the source has undergone in a longer period of time. Little by little the size of the DAW table rises and for the lowest frequencies it has the same order of magnitude as the source table. This means that the delay to process and propagate the data is similar for both the incremental and recompute approach, and the DW replies are based on data of similar age.

Further evidence for the above interpretation comes when considering integrated cost. According to our observations, integrated cost of the two policies should be significantly different for short periods as the recompute policy implies transmission of a much larger number of rows, but the gap should progressively diminish as the period increases. Figure 7 plots the two curves for integrated cost.

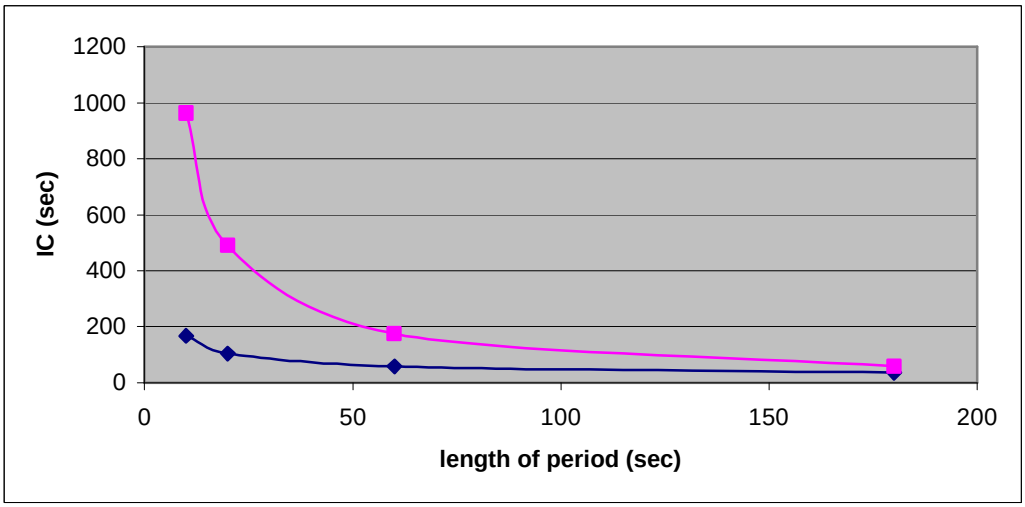


Fig.7. Integrated costs against period (P1 blue, P2 purple).

Observation about integrated cost

The ranking of the integrated cost acquired from our tests follows the prediction of the cost-model, as we can see from Figure 8 and Table 8.

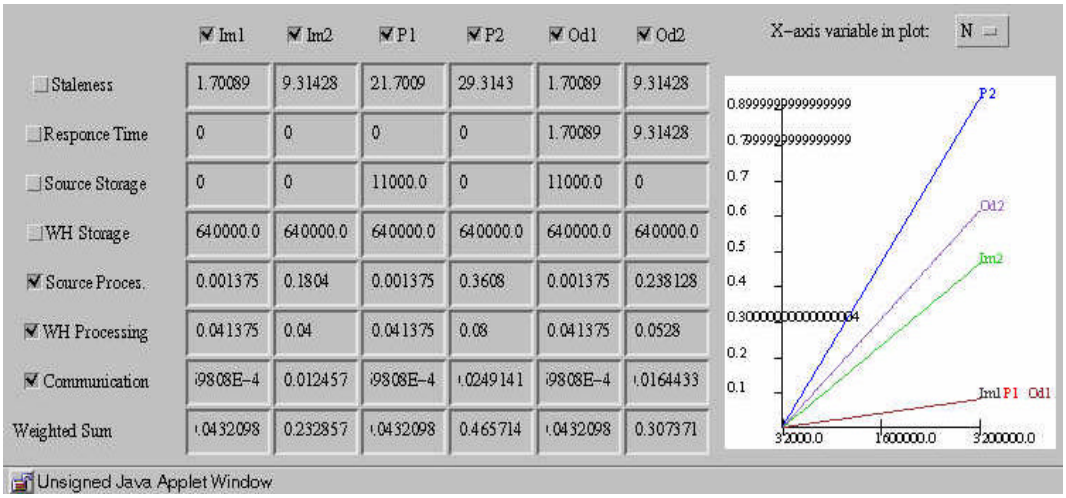


Fig.8. Ranking of policies against integrated cost (DWMP).

Policy	IC (sec)
OD1	96
P1 20	104
IM1	113
IM2	231
OD2	477
P2 20	489

Table 8. Ranking of policies against integrated cost (empirical).

Note that only the performance of periodic policies with length of period equal to 20 seconds (the chosen run parameter) has been reported.

A first comment concerns the point we have discussed when comparing incremental policies to recompute policies in section 8.4.1. At that stage, we found that there is a crossover when maintenance becomes less frequent. Here the order of the policy shown in Table 8 (above) reveals that we are operating in the area of the diagram where the incremental policies perform better than the recompute ones.

This is evidence that, for realistic parameter values, behaviour follows initial expectations.

A second comment concerns the DAW capability. In previous experiments, we have shown that in the case of lack of delta awareness incremental policies perform worse than recompute. In order to examine the influence of DAW on the performance of incremental, we set the source to be extended with DAW capability when configuring the tests above.

The result is that the incremental policies benefited from the delta awareness and therefore performed better than the recompute ones, confirming the prediction of the cost-model.

8.5 CONCLUSIONS

In this chapter we have shown how the benchmarking tool environment can be tuned and effectively utilised to investigate the scenarios we believed to be deserving of interest.

As envisaged, the tuning phase turned out to be somewhat delicate. Despite the help of the user interfaces to set up the required configurations, the process requires care in the choice of settings to preserve consistency between the prototype and the benchmark environment, and hence the objectivity of the final results.

The investigation of a few scenarios selected from our understanding of the cost-model and from use of the DWMP prototype has provided examples of how the benchmarking phase could bring out interesting properties of the behaviour of the cost-model. It is clear that other observations could have been developed given the outcomes of our experiments as they can be looked at from several perspectives.

In the end, the experiments conducted using the benchmarking tool have provided evidence of all three classes of outcome that we can expect from it.

Firstly, some tests have confirmed predictions of the cost-model. This has added to assurance about the assumptions of the cost-model tested in those experiments.

Secondly, some tests have revealed unexpected behaviour which is, however, predicted by the cost-model. This reflects behaviour that has been difficult to foresee because of the apparent intricacy of the considered scenario. Once they have been brought to light, we have been able to account for them, developing our knowledge of the details of the cost-model.

Thirdly, some tests have revealed aspects of behaviour which currently appear to be in conflict with the cost-model. In these cases, results predicted from the prototype have not been borne out by the relevant experiments. These need further exploration.

CONCLUSIONS AND FUTURE WORK

In this dissertation, we have introduced to the cost-model published by Engström et al. and a prototype tool developed by one of the authors to explore implications of the cost-model. It is claimed that the cost-model can be used to predict the behaviour of a defined Data Warehouse architecture with respect to maintenance costs associated with a variety of policies, and that the tool developed can be used to explore optimal policy choice.

We have addressed the issue of building a benchmarking tool for conducting empirical tests in order to investigate the behaviour of DW systems conforming to the architecture assumed in the cost-model. We have analysed the problem in terms of the main criteria for constructing such a tool. We have used these to design the application architecture, maintaining a clear separation between the modelled system and the instrumentation required to measure its performance. We have discussed the implementation of the application, accounting for all the features of the cost-model. All six of the maintenance policies and all three of the source characteristics have been effectively implemented. We have discussed how to effectively simulate workload on the system and how to measure the performance of the system, drawing attention to the overhead introduced. We have also included some features for possible future extensions.

We have tested the protocols and the behaviour of the system under load when running the tool by means of a suite of "sanity" checks. All tests have given positive answers about the tool implementation.

We have provided some examples of how the tool can be used to run benchmarks. From those experiments, we have shown how one may compare the behaviour predicted by the cost-model with that observed using the benchmarking tool. We have made some initial and tentative observations of possible elements of consistency and variance between the two.

There are a number of interesting extensions that are of interest to the originators of the cost-model. The benchmarking tool described here may be extended to a distributed system, where the source and the DW are located on separate hosts by means of Java RMI. The cost-model is to be extended to the case of multiple sources, and non-relational sources (e.g. XML sources) will be considered. With such enhancements in mind, the tool has been designed to support ease of extension.

ACKNOWLEDGMENTS

In particular, I would like to thank Dr. Brian Lings of the University of Exeter (UK) for the support throughout the development of the project and for the precious suggestions and corrections of the dissertation and Henrik Engström, of the Univeristy of Skövde (Sweden), for the inspiring ideas and the invaluable technical support.

APPENDIX A

DATA WAREHOUSES

Since the 1970s, companies have been investing in latest-technology computer systems to automate their business processes. The purpose has clearly been to improve their competitive edge in the market, offering a more proficient and cost-effective service to customers. Therefore, in previous decades we witnessed the spread of operational databases and companies accumulated large amounts of data in their systems. Nowadays, such computer systems are commonplace and a further technology seems to be required to improve companies' competitive edge. This technology uses operational data to support decision-making processes; the new perspective is to build a source of knowledge from archives of data, in order to provide a widely integrated view of a company's data profile.

A.1 DATA WAREHOUSING: MAIN CONCEPTS

In recent times the most authoritative voice in data warehousing has been Bill Inmon (named "the father of data warehousing"). According to his definition, a data warehouse is a "subject-oriented, integrated, non-volatile, time-variant collection of data in support of management's decisions".

"Subject-oriented" refers to the data warehouse standpoint of the company. It focuses indeed on company's subjects (suppliers, customers, sales, stock) rather than on their respective application areas (supplier and customer invoicing, product sales, stock control).

"Integrated" means that a data warehouse must bring standardisation to data coming from diverse sources with different formats.

"Non-volatile" reminds that a data warehouse is not kept up-to-date in real time.

"Time-variant" underlines that a data warehouse perfectly reflects its sources only at some point in time, its view being based on a series of snapshots.

Many other definitions of data warehouse have been given and all suggest that the task of a data warehouse is to collect and organise in a homogeneous view data scattered on different operational resources, allowing users to submit queries, print reports and carry out analysis. The purpose is to increase the value of a company's data asset, including benefits such as potential high returns on investment, company advantages and increased productivity of corporate decision-making.

In summary, a data warehouse is data management and data analysis technology, making available a decision-support environment.

Figure 1 summarises the role of a data warehouse:

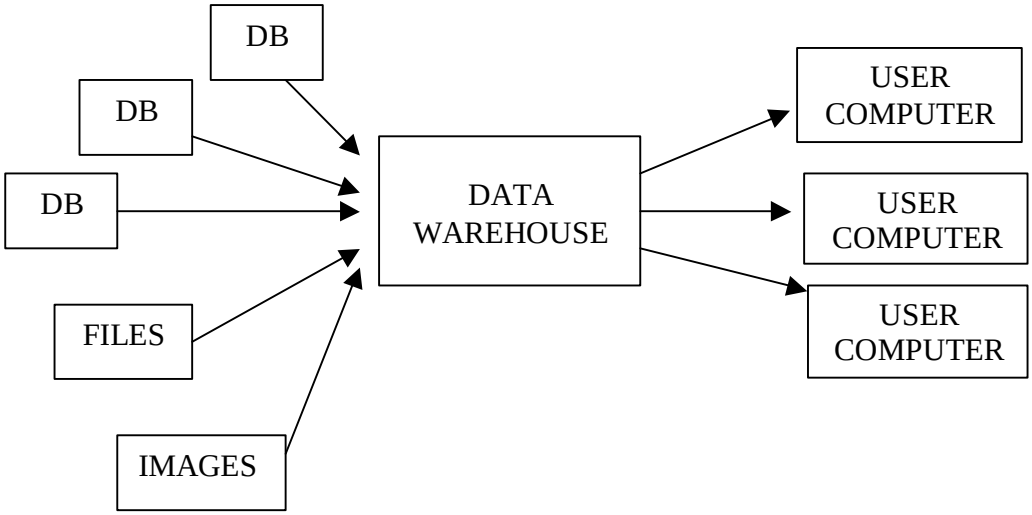


Fig. 1. Flow of information in a DW system.

In the following sections we will briefly discuss some features of data warehousing to give a better idea of its nature.

A.2 DATA WAREHOUSING COMPONENTS

A data warehouse consists of a number of components:

a) Warehouse DBMS(s);

b) Metadata of warehouse contents;

c) Warehouse data management and analysis tools;

d) User help materials.

As stated, a data warehouse extracts and stores data. For doing this, a data warehouse has its own DBMS(s).

Since data are likely to be heterogeneous, in order to handle them a data warehouse must know their type by means of some information pertaining to origin, format and limits on use. Consequently, there is a need for a metadata repository, which is a separate database that keeps up with the data currently stored in the data warehouse. The aim is to isolate the data warehouse from modification in the schemes of source databases. Figure 2 illustrates a typical architecture:

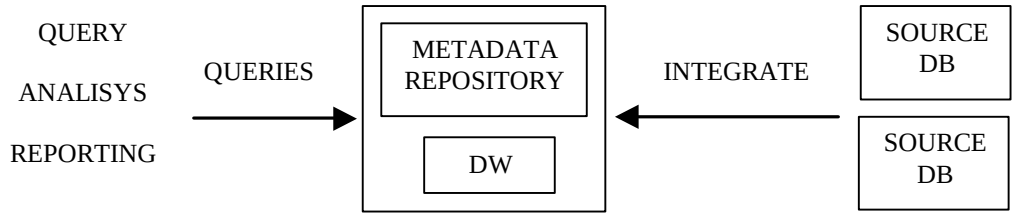


Fig. 2. A DW is often assisted by a metadata repository.

Then, the task is to organise these data for making available an integrated view which users can use for query, reporting and analysis. Hence, tools to transform, aggregate or disaggregate data are embedded in the system.

Finally, all facilities characteristic of a data warehouse should be supported by intuitive GUI, on-line help utilities and training products to make the support-decision environment user-friendly.

A.3 DATA WAREHOUSING REQUIREMENTS

Data warehouses and traditional database applications are designed keeping in mind that they meet different needs and requirements.

A data warehouse is a more dynamic system in comparison to a database. Users are allowed to build their own views starting from the same data. Consequently, one requirement is the aggregation and disaggregation of data according to personal necessity.

For supporting the use of a data warehouse's capabilities, there are two final requisites. It would be useful if a data warehouse were able to provide tools for presenting graphical output as well as exporting data into domain-specific programs.

A.4 DATA WAREHOUSING CHALLENGES

In actually implementing a data warehouse some difficulties arise in meeting all of the above requirements.

Firstly, timing and domain inconsistency happens depending on how and when the data warehouse is kept up-to-date.

Further, inconsistency can develop due to importing data from and exporting data through tools interacting with the data warehouse.

Finally, we are noticing a continuing lack of tools that fulfil companies' growing requirements. Indeed, more and more often companies actually have to develop such tools internally, experiencing long and expensive processes.

For a more complete discussion of data warehousing concepts we refer readers to [CON99] and [ELM97].

REFERENCES

[ENG00] H.Engström, S.Chakravarthy, B.J.Lings, "A Holistic Approach to the Evaluation of Data Warehouse Maintenance Policies", 2000

[ADE95] B.Adelberg, H. Garcia-Molina and B.Kao, "Applying update streams in a soft real-time database system". In SIGMOD 1995.

[COL97] Latha S. Colby, Akira Kawaguchi, Daniel F. Lieuwen, Inderpal Singh Mumick, Kenneth A. Ross, "Supporting Multiple View Maintenance Policies". In SIGMOD 1997.

[CON99] T.R. Connolly, C. E. Begg, "Database Systems: A Practical Approach to Design, Implementation, and Management", International Computer Science Series, Addison Wesley, 1998, pages 912-933

[ELM97] R.A. Elmasri, S.B. Navathe, "Fundamentals of Database Systems", World Student Series, Addison Wesley, 1999, pages 842-854

BIBLIOGRAPHY

J. Gray (ed.), "The benchmark Handbook for database and transaction Processing Systems", Second Edition, Morgan Kaufmann Publishers, San Francisco, CA (1993)

S.W.Dietrich, M. Brown, E. Cortes-Rello and S. Wunderlin, "A practioner's approach to the Database Performance Benchmarks and Measurements", The Computer Journal, vol. 35., NO.4, 1992

Eric N. Hanson, "A performance Analysis of View Materialization Strategies", SIGMOD Conference, 1987

M.T. Ozsu, P. Valduriez, "Principles of Distributed Database systems", Prentice Hall, 1999