Data Integration in Heterogeneous Environments: Multi-Source Policies, Cost Model, and Implementation

Henrik Engström
University of Skövde, Sweden
henrik@ida.his.se

Sharma Chakravarthy
The University of Texas at Arlington
sharma@cse.uta.edu

Brian Lings
University of Exeter, UK
B.J.Lings@exeter.ac.uk

Technical Report

HS-IDA-TR-02-003
Department of Computer Science
University of Skövde, Sweden

Abstract. *The research community is addressing a number of issues in response to an increased reliance of organisations on data warehousing. Most work addresses aspects related to the internal operation of a data warehouse server, such as selection of views to materialise, maintenance of aggregate views and performance of OLAP queries. Issues related to data warehouse maintenance, i.e. how changes to autonomous sources should be detected and propagated to a warehouse, have been addressed in a fragmented manner.*
We have shown earlier that a number of maintenance policies based on source characteristics and timing are relevant and meaningful to single source views. In this report we detail how this work has been extended for multiple sources. We focus on exploring policies for data integration from heterogeneous sources. As the number of policies is very large, we first analyse their behaviour intuitively with respect to broader source and policy characteristics. Further, we extend the single source cost model to these policies and incorporate it into a Policy Analyser for Multiple sources (PAM). We use this to analyse the effect of source characteristics and join alternatives on various policies. We have developed a Testbed for Maintenance of Integrated Data (TMID). We report on experiments conducted to validate the policies that are recommended by the tool, and confirm our initial analysis. Finally, we distil a set of heuristics for the selection of multi-source policies based on quality of service and other requirements.

1. Introduction

Organisations are becoming increasingly reliant on data originating from distributed, heterogeneous, and autonomous sources. These can be web sites or autonomous, commercial and operational databases. Information extracted from multiple sources, and stored in a database for local access, can be described as a data warehouse (DW) [Gup95]. The contents of such a warehouse can be described as a set of materialized views based on distributed, heterogeneous, and autonomous sources [Ham95]. Figure 1 shows the components of the DW architecture used in this report. Sources are wrapped to communicate with an integrator that updates the warehouse view. Queries are posed against this view.

Data warehouse maintenance, in a somewhat simplified way, can be seen as a generalization of view maintenance used in relational databases. A DW maintenance policy determines *when* and *how*

to refresh the content of a warehouse view to reflect changes to its sources. Various view maintenance policies have been suggested and analysed in the literature [Han87, Seg89, Seg91, Col97]. A policy can, for example, be to recompute a view or do incremental maintenance; and maintain the view immediately when changes are detected, on-demand when the view is queried, or periodically. When combined, this gives rise to six different single source policies [Eng02]: immediate incremental (I_I), immediate recompute (I_R), periodic incremental (P_I), periodic recompute (P_R), on-demand incremental (O_I), and on-demand recompute (O_R).

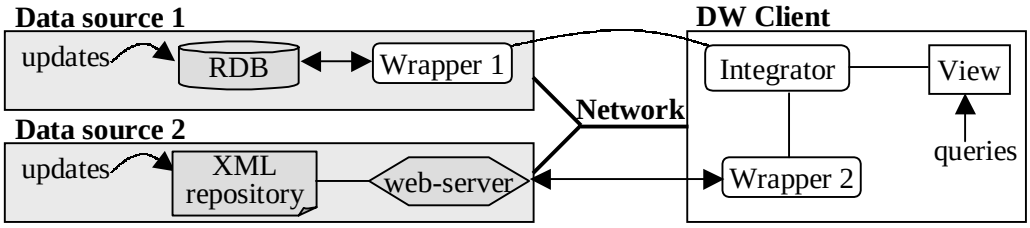

Figure 1. A data warehouse architecture

It has been shown that the selection of the most appropriate maintenance policy is a complex process affected by several dynamic and static system properties [Han87, Sri88, Col97]. There have been several approaches [Agr97, Hul96, Zhu96] to incrementally maintaining views and preserving consistency when sources are autonomous. Consistency, however, is not the only user requirement that needs to be considered. Data staleness (a measure of the elapsed time between receiving an answer to a query and the first source change to invalidate it) and response time are examples of other user requirements that have a significant impact on maintenance activity. Moreover, source capabilities play a crucial role in DW maintenance and cannot be overlooked.

For many DW scenarios, data from multiple sources needs to be combined and stored. It is possible that this combination can be performed within the DW through queries over warehouse data. A commonly used alternative is to combine (or join) data from different sources using an integrator, and then store it in the warehouse. The warehouse designer has the responsibility of deciding where, when, and how the data is to be combined. In this report we only consider the case where data is combined at the integrator level. We assume that the DW schema is given, and that the problem is to determine a maintenance policy to be used by the integrator for warehouse views defined over one or

more *supporting views.* A supporting view (virtual or materialised) is defined as a subset (e.g., select-project) of a data-set stored in a source.

To analyse this problem we first need to understand how heterogeneous views can be combined and what policies can be used to maintain them. We extend the policies proposed for single sources [Eng02] to multiple sources, something requiring more complex policy evaluation.

1.1. Problem Statement

DW maintenance for views on a single data source has been investigated earlier [Eng02]. For such views, the components of the maintenance problem have been characterised and a cost model developed that captures dependencies between policies, source & DW characteristics, and evaluation criteria. With a user-centric approach to specifying quality of service requirements for staleness and response time, the work clearly indicated the need to support several policies.

To generalise this work to multi source views in a DW context, there is a need to consider: new potential policies; additional source characteristics; and evaluation criteria. The approach we have taken is to analyse the policies for multiple sources systematically, developing heuristics for policy selection. The heuristics themselves have then been analysed against data from a multi-source testbed implementation of DW maintenance policies.

1.2. Roadmap

The remainder of the report is organized as follows. In section 2, we discuss a generalized join for heterogeneous sources and describe how multi-source policies are identified. In section 3 we analyse the policies and present a cost model which formulates the costs for policies based on source and join characteristics. This is used in a tool (a *Policy Analyser for Multiple sources* or PAM) for systematic comparison of policies. This tool is used to analyse policies and to produce heuristics for policy selection. In section 4 we compare this with the performance of the policies in an implemented *Testbed for Maintenance of Integrated Data* (TMID). Section 5 contains related work and in section 6 we draw conclusions and suggest future work.

2. Data Integration from Multiple Heterogeneous Sources

For single source views, a number of source capabilities have been identified [Eng00] that affect the relevance and cost of maintenance policies. Although a wrapper may compensate for most of these capabilities, it may impact on QoS and system cost. Supporting views for multiple source views are, by our definition, single source so the same capabilities relate. In addition, some maintenance policies require additional interaction with a source. In this section we analyse *join view maintenance*, and identify potential policies and relevant source capabilities. We analyse the effect of meaningful policies on consistency, an important QoS criterion for views based on multiple autonomous sources.

2.1. Generalized Join for Heterogeneous Sources

Operations on heterogeneous, autonomous sources are not well understood. In a relational DBMS, several relations can be integrated through joins and (for union-compatible relations) set-operations. Misra and Eich [Mis92] give a good overview of join processing in relational databases. Although they claim that the join operation is unique to relational algebra, they acknowledge that join-like operations exist for other data models.

The ODMG (http://www.odmg.org/) has proposed a standard which includes, among other things, an object model and a query language - OQL. In OQL objects are collected using a "select-from-where" based on SQL-92 SELECT. An OQL query can reference objects, sets of objects (e.g. class extents), arrays, lists, and bags of objects, among other things.

In contrast to SQL, OQL is not based on a formal calculus. This makes it hard to analyse the nature of queries and to perform query transformations. This is addressed by Fegaras and Maier [Feg95, Feg00] who propose a calculus which defines OQL queries as monoid comprehensions. They present an unnesting operator which uses the monoid comprehension calculus, and show that this operator can unnest any query. In the unnested form, all queries can be expressed using a low level algebra which operates on collections of objects. This algebra contains only two binary operators: *join* and *outer-join*. It is important to remark that join in Fegaras' and Maier's algebra is a richer operator than a relational join. As an example, the relational union-operator becomes a special case of join.

Joining of XML sources can be treated similarly [Feg01]. We adapt this generalised join for our work, as we are dealing with a heterogeneous environment.

Definition: A *join view* contains a subset of the Cartesian product of two supporting views, where elements in the view are concatenations of parts of objects, one object from each supporting view. A join-predicate p(x,y) determines the relevant subset. A join-predicate is an arbitrary function which takes an object from each set and returns true if the two objects should be combined and included in the join, false otherwise.

This join differs from operations in database environments in a number of ways. Firstly, a common data model cannot be assumed. Secondly, there may be semantic differences which require cleaning and transformation before data from different sources can be combined. The predicate must be powerful enough to capture such things.

It is assumed that selection and projection of sources are pushed down to supporting views with key attributes preserved. Each element of a join view will have a unique id formed from the ids of the two concatenated elements. We do not consider deleting duplicated attributes (as in natural join).

2.2. Multi-source Maintenance Policies

For a single source view, the task of the integrator is simply to update the warehouse view. With a join view the integrator has to determine *when* and *how* the join should be performed. Much attention has been paid in the literature to the fact that the joining of two autonomous data sources can be handled in a more or less consistency preserving way. Two fundamental approaches have been identified for handling the consistency problem: one is to maintain a copy of each supporting view, which is used to compute the view in a controlled way; the other is to maintain the view incrementally, by sending compensating queries to sources when changes are reported [Zhu96]. Much effort has been focused on producing algorithms for consistent warehouse maintenance for the latter case.

Apart from these specific algorithms to preserve consistency, we claim that the integrator can use the same policies as a single source. For example, it is possible to do a join *incrementally* or through *recomputation*. Also, the timing of a join can be *immediate* (when changes are reported from

wrappers), *periodic* or *on-demand*. In addition, each supporting view can be maintained by the appropriate single source by choosing one of our suggested [Eng02] six policies. Throughout this section we use the abbreviation *SVP* for Supporting View Policy, and *integrator policy* to denote the strategy used for joining. A join policy is a combination of integrator policy and SVP for each source. The combination of SVPs and integrator policies for a join view with two sources gives rise to 6*6*6*2*2=864 potential policies to choose from (six policies for *each* supporting view, six policies for the integrator and a choice to maintain a copy of each supporting view at the integrator). In addition, the periodicity of periodic policies can be varied for each periodic activity.

When a copy of the supporting view is maintained in the integrator we will refer to it as an *auxiliary view* [Qua96]. If no copy is maintained it implies that the supporting view is virtual.

We should clarify that the meanings of on-demand and immediate are different for a join view and a single source view. For example, immediate join is done immediately when changes from the wrapper are sent to the integrator. This is not necessarily immediately after changes are committed in the source. In a similar way, an on-demand SVP is maintained when the integrator sends a request, not necessarily when a request is sent from the warehouse to the integrator.

2.3. Identifying Candidate Policies for Further Analysis

Not all of the identified 864 combinations of SVPs and integrator policies are meaningful, however. It is trivial to show that it is not possible to use some combinations of integrator and wrapper timings. If a wrapper has on-demand timing, it means that it is awaiting requests from the integrator. In such a case it is not possible to have an immediate policy in the integrator, as it will lead to a deadlock. The integrator is waiting for updates from the wrapper, which is waiting for requests from the integrator.

In a similar way, if the integrator is maintaining the view incrementally, the SVPs also have to be incremental. With a recomputed SVP, the wrapper will propagate the whole supporting view to the integrator, which implies there is no delta for use in incremental maintenance.

When auxiliary views are maintained in the integrator, it is possible to use an incremental SVP and to recompute the view. Without auxiliary views, however, the whole supporting view needs to be retrieved to recompute the view. This implies that when integrator policy is recompute and no

auxiliary views are maintained, the only SVP that can meaningfully be used is O_R. Both P_R and I_R are asynchronous with joining and will hence force the supporting view to be stored. In practice, this will be the same as keeping auxiliary views. Policies both with and without auxiliary views are considered in the analysis.

Table 1 shows the set of policies after the reduction described here. With two sources these represent 134 different policies[1]. Each of these is considered in the analysis which follows. In the rest of this report we will name join policies according to the following:

<Integrator policy>-<aux/noAux>-<left SVP>-<right SVP>.

For example, I_R-aux-I_I-P_I denotes a policy which immediately recomputes the view when changes are reported from the wrappers, and uses auxiliary views. The wrappers use incremental maintenance. The left sends changes immediately when the source reports them, while the right sends changes periodically.

2.4. Consistency Implications

An important QoS characteristic for a view based on several autonomous sources is how consistent it is with its sources. View maintenance may be done to varying levels of consistency [Zhu96]. Table 1 shows all join policies considered in this report, and the consistency each provides. When auxiliary views are used for a binary join, it is relatively easy to ensure strong consistency (that consecutive view states are based on valid consecutive source states). The state of each auxiliary view will always reflect a valid state of the source, and state changes will reflect the evolution of the source. As both maintenance of auxiliary views and joining are done in the integrator, it is easy to ensure that updates of auxiliary views are not done concurrently with maintenance of the join view. When auxiliary views are *not* maintained in the integrator, and the view is recomputed, strong consistency can be guaranteed. This is because each delivered supporting view will reflect states from each source. Moreover, when the view is recomputed, the supporting views will always reflect the same or a later state. When there are no auxiliary views and the integrator policy is incremental,

[1] For each line in Table 1, any combination of the shown SVPs is possible. This means that the number of join policies is the square of the number of SVPs. Totally there will be 4+16+9+36+9+36+4+0+9+1+9+1=134 different policies.

consistency cannot be guaranteed unless maintenance algorithms are modified appropriately (see for example [Zhu96]). We do not consider such algorithms in this report.

Table 1. Join policies considered in this report and the level of consistency they provide.

Integrator policy	Auxiliary views	Possible SVPs (one for each supporting view)	Consistency implications
I_I	Yes	I_I or P_I	Strong consistency
I_R	Yes	I_I or I_R, P_I, P_R	Strong consistency
P_I	Yes	I_I, P_I, O_I	Strong consistency
P_R	Yes	I_I, I_R, P_I, P_R, O_I, O_R	Strong consistency
O_I	Yes	I_I, P_I, O_I	Strong consistency
O_R	Yes	I_I, I_R, P_I, P_R, O_I, O_R	Strong consistency
I_I	No	I_I or P_I	No consistency[2]
I_R	No	None	Not applicable
P_I	No	I_I, P_I, O_I	No consistency[2]
P_R	No	O_R	Strong consistency
O_I	No	I_I, P_I, O_I	No consistency[2]
O_R	No	O_R	Strong consistency

2.5. Source Capabilities for Participating in Joins

If the integrator keeps auxiliary views, it can join them without any additional interaction with a source. This means that source capabilities do not affect a join, only the maintenance of auxiliary views. However, when a view is incrementally maintained without auxiliary views, the integrator has to send compensating/join queries to find objects in one source that join with objects in the other source. Ideally, the join predicate computation should be supported natively in the source. The source will return the set of objects that match the predicate for a given object. If the source does not support queries, or does not understand the join predicate, it becomes the responsibility of the wrapper to find matching objects through operations on the supporting view. To be able to classify a source's ability to participate in join view maintenance we define the *semi-join aware* capability.

Definition: A source is semi-join aware (SAW) for a join with predicate p if for any object O it can return objects that are matched under predicate p.

[2] Unless algorithms are introduced to ensure consistency. The cost of this is, however, not included in this work.

Note that the SAW capability is defined with respect to a specific predicate. A source can be singleton or set-oriented SAW. Set-oriented SAW means that, in a single query, for a set of objects it can return all objects that participate in the join. As an example, a relational database is set-oriented SAW for a relational join if it supports set oriented selection (e.g. "select * from tab where joinAttribute in {12,34,17}"). If it only supports single valued selections, it will only support singleton SAW[3]. As with other capabilities[4], a source which is SAW should provide it without requiring extensions or resource wasting "work-arounds" [Eng00].

When a source is not SAW, the wrapper has to retrieve the supporting view and perform the join. As an example, one source may contain regular expressions while the other contains sequences of text. If the join predicate applies the regular expression to the sequences, a standard relational DBMS does not support such queries. If we want to find sequences that match a new regular expression, the wrapper has to retrieve all sequences and apply the expression to each of them. This is potentially different from querying a source that supports regular expressions in the join predicate. We have encountered the utility of the above for protein sequence identification. Protein sequences are, for example, available in web data sources such as SWISS-PROT (http://www.expasy.org/sprot/) and regular expressions are used to identify which protein family they belong to.

3. Analysis of Policies

The selection of a single source policy has been shown to depend on a combination of source characteristics and evaluation criteria [Eng02, Eng02b]. When deciding on policy timing (periodic, on-demand, or immediate) the relationship between periodicity, change frequency and update frequency plays an important role. For example, if system cost is the only concern, periodic recompute polices with low periodicity (in relation to change frequency) typically outperform immediate incremental policies. Further, source capabilities have a bearing on maintenance. To use immediate policies, for example, a source has to be CHAC and the choice between incremental and

[3] As is the case with other capabilities, intermediate situations are possible. For example, a relational database may support disjunctions ("select * from tab where joinAttribute=12 or joinAttribute=34 or joinAttribute=17") but limitation on query length typically gives a relatively low upper bound on the number of terms.

recompute is affected by the ability of the source to deliver deltas (DAW). When a source is not DAW, recomputing the view can give better performance than incremental maintenance, even when change size is small [Eng02].

All of the above observations have been established for the single-source model and tested in practice by measuring the performance of policies in a testbed [Eng02b]. Join policy selection inherits all these properties for each SVP involved, and adds the issue of when and how to perform the join, and how to "coordinate" SVP selection.

As is evident from the above discussion, the dependencies for a join view are complex and hence it is very difficult to intuitively understand all the implications of source characteristics and timing on maintenance. To support a detailed analysis of various policies, we have developed a cost model that captures dependencies between integrator, SVP, and the underlying sources. This cost model has been incorporated into PAM, which facilitates relative comparison of join maintenance policies. In this section, we demonstrate the utility of PAM to analyse the dependency of policy selection on source characteristics. We give a description of the cost model and PAM, and then present the results obtained using PAM to compare all suggested polices for a large number of situations in which evaluation criteria, view, and source properties have been varied.

3.1. A Cost Model for Join Views

The cost model formulates cost in terms of staleness, response time, storage, processing and communication for each policy and each combination of source capabilities. Costs are expressed using parameters such as source and view size, update size, update and query frequency, etc. To be applicable to various sources, views and data models, costs are expressed in terms of functions. We have chosen to use size in bytes in all cost-formulations and to "push" the object sizes into the cost components (see below). This reduces the complexity of our cost formulation (less number of variables) without losing flexibility.

Table 2 shows the cost components introduced to formulate the cost of integrator activity.

---

[4] Source characteristics defined in [Eng00] include: CHAC (change active – the ability to actively notify that changes have occurred), DAW (delta aware – the ability to deliver delta changes), and CHAW (change aware – the ability to indicate whether a source has changed).

Table 2. Cost components to formulate integrator costs

join(x,y)	The delay to perform a join with a left data set of size x and a right data set of size y. This includes possible partition costs.
merge(x)	The delay to merge a delta of size x into a view. This is required for incremental maintenance
q	The query frequency
p	The periodicity used in the integrator

The join-cost-function "join(x,y)" expresses the delay incurred for the join of two sets (in terms of their sizes, x being the size of the left supporting view and y being the size of the right supporting view). Different formulations of join cost can be used, based on the data model as well as the join strategy used. The arguments are expressed in terms of bytes; for example, a left supporting view of size 100Kb with a right supporting view of size 1 Mbyte. However, join cost is typically determined by the cardinality of the sets. This means that these cost-components will have to encapsulate the object size for each source. Let the object size be 1Kb for the left supporting view and 2Kb for the right supporting view. If the join cost is three times the sum of the cardinalities, then the join cost function can be formulated as:

$$join(x,y)= 3* (x/1024+y/2048)$$

The remaining components in Table 2 are the merge cost function, representing the cost of inserting a change (hereafter referred to as a delta) into a join view, query frequency and integrator periodicity. The last two are both represented as frequencies (sec^{-1}); integrator period is therefore obtained as 1/p.

To express the cost of maintaining each supporting view we use the components shown in Table 3. Most of these have a left and a right version distinguished using a subscript. We show only the left components in Table 3.

Table 3. Cost components used to formulate the left supporting view costs.

N_L, M_L	The size of the left source and the size of the left supporting view
Z_L, RT_L,SS_L,WS_L, SP_L, WP_L, COM_L	Staleness, response time, source storage, DW storage, source processing, DW processing and communication of the left supporting view. These are formulated and described in [Eng00b, Eng02].
p_L	The periodicity of the left wrapper
c_L	The frequency of changes from the left wrapper. This is determined by the change frequency of the source (if SVP is immediate) or the periodicity (if SVP is periodic)
q_L	The query frequency used for the left SVP. This is determined by the integrator.
r_L	The cost of recomputing the left supporting view in the source
L_L	The change size of the left supporting view. This is determined by the change size of the source and the relation between c_L and the change frequency of the source.
jsf_L	Join selectivity factor for the left supporting view. This factor is multiplied by the size of the left supporting view to derive the size of the join view.
e(x)	The cost of extracting the given (x) amount of data from the source. This is from the single source cost model [Eng02].
d(x)	The cost of transmitting the given (x) amount of data over the network. This is from the single source cost model [Eng02].

As we will see, the single source cost model can be used unchanged in most situations to model the cost of a supporting view. In a few situations we have to correct the cost (when auxiliary views are not used) or "disable" a source capability (recomputed policies without auxiliary views). The periodicity of the wrapper (p_L) is a parameter of the single source cost model that can affect the join cost, for example if the integrator policy is immediate. The change frequency from the wrappers, c_L, is determined by SVP policy:

- If the SVP is immediate: c_L= the change frequency of the source
- If the SVP is periodic: c_L= the periodicity of the wrapper (p_L)
- If the SVP is on-demand: $c_L=q_L$

In a similar way, with on-demand SVP query frequency is determined by integrator policy:

- If the integrator timing is on-demand: $q_L=q$
- If the integrator timing is periodic: $q_L=p$
- If the integrator timing is immediate q_L will not be used. It is not possible to combine on-demand SVP with an immediate integrator policy.

Whenever the wrapper reports a change (which depends on SVP) the size of each change, L_L, is given as follows:

- If the SVP is immediate: L_L=change size of the source
- If the SVP is periodic: L_L=(change size of the source)*(change frequency of the source)/p_L
- If the SVP is on-demand: L_L= (change size of the source)*(change frequency of the source)/q_L

The join selectivity factor needs a further explanation. It is common to define join selectivity as the relation between the cardinality of the view and the product of the cardinality of the two joined sets. If there are n objects in one set and m objects in the other, there will be: n*m*jsf objects in the view. In our formulation, we introduce left and right join selectivity factors (jsf_L and jsf_R) which, from the size in bytes of one set, express the size in bytes of the view. This means that jsf_L multiplied by the left size equals jsf_R multiplied by the right size. Again, we avoid using the object sizes and express sizes in bytes. With the example above, and with jsf denoting the "regular" selectivity factor (on cardinality), the following equality holds:

jsf_L*x=jsf_R*y=(1024+2048)*jsf*x/1024*y/2048

This means that, for this specific situation, jsf_L (similarly for jsf_R) can be computed in the following way:

jsf_L=(1024+2048)*jsf*y/(1024*2048)

The final components in Table 3 formulate the cost of extracting a certain amount of data from the source (e) and to send data over the network (d). These functions are taken from the single source cost model [Eng02] and are assumed to be identical for the two sources.

3.1.1. Recomputed Join With Auxiliary Views

Table 4 shows the cost of policies with a recompute integrator policy and with auxiliary views. In some cases the cost formulation depends on integrator timing. In such a case the cost is preceded with the integrator timing typed in a bold typeface.

Table 4. Recompute integrator policy with auxiliary views

Z	**Periodic:** $Max(Z_L, Z_R) + join(M_L, M_R) + 1/p$ **On-demand and Immediate:** $Max(Z_L, Z_R) + join(M_L, M_R)$
RT	**Periodic and Immediate:** 0 **On-demand :** $Max(RT_L, RT_R) + join(M_L, M_R)$
SS	$SS_L + SS_R$
WS	$WS_L + WS_R + M_L + M_R$
SP	$SP_L + SP_R$
WP	**Periodic:** $WP_L + WP_R + p \cdot join(M_L, M_R)$ **On-demand:** $WP_L + WP_R + q \cdot join(M_L, M_R)$ **Immediate:** $WP_L + WP_R + (c_L + c_R) \cdot join(M_L, M_R)$
COM	$COM_L + COM_R$

Staleness for recompute policies is determined by worst case staleness for the supporting views. In addition there is a delay to perform the join. For periodic policies, a full period is added to staleness (in the worst case, changes are delayed a full period in the integrator before they are reflected in the view).

For response time, the cost model is intended to capture the *additional* response time incurred by maintenance activity. The time to execute the actual query, once the view is up-to-date, is not included. This explains why periodic and immediate policies have zero response time cost; whenever a request comes, the current state of the view will be used. On-demand policies, on the other hand, will initiate maintenance when a request is sent to the view. The integrator will then have to wait for both auxiliary views to be updated before it can recompute the view. This means that the total response time is the worst case response-time for the auxiliary view plus the time to recompute the view.

Source storage cost is simply the sum of the storage costs for the supporting views. Warehouse storage is the sum of the warehouse storage of the supporting views plus the additional space required to store the auxiliary views, M_L+M_R.

For source processing, there is no cost in addition to the sum of the costs of maintaining the supporting views. This is always the case when auxiliary views are kept in the integrator.

For all timings, warehouse processing cost contains a component for each auxiliary view. In addition there is a cost associated with performing the actual join. The timing determines how frequently the join is performed, and this obviously affects the cost. Periodic policies will incur a join for each period, giving a cost of: p*join(M_L, M_R). In a similar way, on-demand policies will incur a join cost

for each query, giving a cost of: q*join(M_L, M_R). Immediate policies require a join for each change reported from each wrapper, which means performing a join with frequency (c_L+c_R).

Finally, the communication cost when auxiliary views are maintained is the sum of the communication costs for the supporting views.

It is important to note that join cost can be formulated without considering which specific SVPs are used. This means that the cost-formulations shown above apply to all periodic recomputed joins irrespective of SVP. For example, P_R-aux-I_I-I_I and P_R-noAux-O_R-P_R have the same staleness cost-formulation, although actual staleness will differ significantly between SVPs.

3.1.2. Incremental Join With Auxiliary Views

When an incrementally join is performed with auxiliary views, the integrator will incur a different join cost. In worst case, incremental join will involve a join of each change from one supporting view with the auxiliary view for the other. There is then the cost of merging the changes to the view (the delta) into the view. Table 5 shows the cost formulations for this situation.

Table 5. Incremental integrator policy with auxiliary views

Z	**Immediate:** $Max\big(Z_L + join(L_L, M_R) + merge(jsf_L \cdot L_L), Z_R + join(M_L, L_R) + merge(jsf_R \cdot L_R)\big)$ **On-demand:** $Max(Z_L, Z_R) + join(L_L \cdot c_L/q, M_R) + join(M_L, L_R \cdot c_R/q) +$ $merge(jsf_L \cdot L_L \cdot c_L/q + jsf_R \cdot L_R \cdot c_R/q)$ **Periodic:** $Max(Z_L, Z_R) + join(L_L \cdot c_L/p, M_R) + join(M_L, L_R \cdot c_R/p) +$ $merge(jsf_L \cdot L_L \cdot c_L/p + jsf_R \cdot L_R \cdot c_R/p) + 1/p$
RT	**On-demand:** $Max(RT_L, RT_R) + join(L_L \cdot c_L/q, M_R) + join(M_L, L_R \cdot c_R/q) +$ $merge(jsf_L \cdot L_L \cdot c_L/q + jsf_R \cdot L_R \cdot c_R/q)$ **Periodic and Immediate:** 0
SS	$SS_L + SS_R$
WS	$WS_L + WS_R + M_L + M_R$
SP	$SP_L + SP_R$
WP	**Immediate:** $WP_L + WP_R + c_L \cdot \big(join(L_L, M_R) + merge(jsf_L \cdot L_L)\big) + c_R \cdot \big(join(M_L, L_R) + merge(jsf_R \cdot L_R)\big)$ **On-demand:** $WP_L + WP_R +$ $q \cdot \big(join(L_L \cdot c_L/q, M_R) + join(M_L, L_R \cdot c_R/q) + merge(jsf_L \cdot L_L \cdot c_L/q + jsf_R \cdot L_R \cdot c_R/q)\big)$ **Periodic:** $WP_L + WP_R +$ $p \cdot \big(join(L_L \cdot c_L/p, M_R) + join(M_L, L_R \cdot c_R/p) + merge(jsf_L \cdot L_L \cdot c_L/p + jsf_R \cdot L_R \cdot c_R/p)\big)$
COM	$COM_L + COM_R$

An immediate join policy is implemented independently for the two sources. For each change there are three different components. For staleness, the first component is the staleness of the auxiliary view. This depends on SVP, and is the delay in detecting and reporting changes from the wrapper and

updating the auxiliary view. In addition there is the cost of joining the changes (each change reported from the left supporting view has size L_L) with the other auxiliary view (for the right source this has size M_R). There is then the cost of merging the delta into the view.

The size of the view delta is determined by the join selectivity factor and the size of changes to the supporting view. Immediate policies will be initiated by changes from one source, and it is assumed that these are reported independently. This means that staleness will be the maximum of the staleness of each source, i.e. the delay to propagate and handle the changes from that source.

With on-demand and periodic policies, the joining of changes from both sources will be handled at the same time. This implies that merging can be done once, a potential advantage. Another difference is in the size of changes. This is determined by the size of changes propagated from the wrappers (e.g. L_L for the left source) and the relation between the change propagation frequency (c_L) and the join frequency (p for periodic and q for on-demand). For periodic join, for example, the size of the left change is L_L*c_L/p. This means that if the wrapper propagates changes with a higher frequency than the join periodicity then each round of maintenance will have more accumulated changes. If the periodicity is increased, the size will be reduced. As with recompute, staleness is determined by the worst case staleness of auxiliary views plus the time to update the view. Periodic policies have a full period added to the staleness.

Response time is zero for periodic and immediate policies. For on-demand integrator policies, response time is the worst case response-time for the auxiliary views plus the time to perform the join. Storage costs, source processing and communication have the same cost formulation as recomputed join. Warehouse processing is different. For all policies it is the processing cost for the supporting views plus the cost to do maintenance, multiplied by the associated frequencies (c for immediate, p for periodic and q for on-demand).

3.1.3. Recomputed Join Without Auxiliary Views

The cost formulations for recomputed join without auxiliary views are shown in Table 6.

Table 6. Recompute integrator policy without auxiliary views

Z	**Periodic:** $Max(Z_L, Z_R) + join(M_L, M_R) + 1/p$ **On-demand:** $Max(Z_L, Z_R) + join(M_L, M_R)$
RT	**Periodic:** 0 **On-demand :** $Max(RT_L, RT_R) + join(M_L, M_R)$
SS	$SS_L + SS_R$
WS	$WS_L + WS_R$
SP	$SP_L + SP_R$ *(sources should be treated as non-CHAW)*
WP	**Periodic:** $WP_L + WP_R + p \cdot join(M_L, M_R)$ *(sources should be treated as non-CHAW)* **On-demand:** $WP_L + WP_R + q \cdot join(M_L, M_R)$ *(sources should be treated as non-CHAW)*
COM	$COM_L + COM_R$ *(sources should be treated as non-CHAW)*

The only differences with Table 4 (with auxiliary view) are that immediate policies are excluded and storage cost in the warehouse does not have the M_L+M_R component. In addition, if a source is CHAW it cannot be utilised for the SVPs. When the integrator sends a request to the wrappers to maintain the view (all SVPs will be on-demand in this case), a CHAW source may return nothing if no changes have occurred. This is unacceptable when no auxiliary view is maintained, as the supporting view is needed to compute the join view. When the integrator maintains an auxiliary view it can use the CHAW capability to avoid propagating the supporting view when it is unchanged.

3.1.4. Incremental Join Without Auxiliary Views

For incremental policies without auxiliary views, the integrator will have to send join queries to a source to find objects that match changes from the other source. For these policies, source capabilities affect join cost. As with other policies, join timing makes a difference to the size of changes and the intensity with which maintenance is done. Immediate policies join and merge changes from each source independently, while periodic and on-demand joins handle both sources at the same time. On-demand join has non-zero response-time, and periodic join has a period added to staleness. We present the cost formulations for each timing in separate tables. As the sources may have different capabilities they are considered independently, but the table formulates the cost for the right source only. The left source cost is identical except that subscripts R and L are reversed.

A difference when no auxiliary views are maintained is that deltas propagated from sources do not have to be inserted into the auxiliary views. This implies that the SVP cost components in the cost-formulas below should not include the cost of incrementally inserting changes into the view. Using the cost formulation presented in [Eng02] this means that the i(x,y)-component is removed.

The cost formulations for immediate join are shown in Table 7.

Table 7. Immediate incremental integrator policy without auxiliary views

Z	The maximum Z from the left and right source. For the right source this is given by: **SAW:** $Z_R + d(L_R) + join(M_L, L_R) + d(jsf_R \cdot L_R) + merge(jsf_R \cdot L_R)$ **not SAW, Server Wrap:** $Z_R + d(L_R) + r_L + e(M_L) + join(M_L, L_R) + d(jsf_R \cdot L_R) + merge(jsf_R \cdot L_R)$ **not SAW, Client Wrap, VAW:** $Z_R + r_L + e(M_L) + d(M_L) + join(M_L, L_R) + merge(jsf_R \cdot L_R)$ **not SAW, Client Wrap, not VAW:** $Z_R + e(N_L) + d(N_L) + r_L + join(M_L, L_R) + merge(jsf_R \cdot L_R)$
RT	0
SS	$SS_L + SS_R$
WS	$WS_L + WS_R$
SP	$SP_L + SP_R$+ a cost per source. For the right source this is given by: **SAW:** $c_R \cdot join(M_L, L_R)$ **not SAW, Server Wrap:** $c_R \cdot \big(r_L + e(M_L) + join(M_L, L_R)\big)$ **not SAW, Client Wrap, VAW:** $c_R \cdot \big(r_L + e(M_L)\big)$ **not SAW, Client Wrap, not VAW:** $c_r \cdot e(N_L)$
WP	$WP_L + WP_R + c_L \cdot merge(jsf_L \cdot L_L) + c_R \cdot merge(jsf_R \cdot L_R)$ +a cost per source. For the right source this is given by: **Server Wrap or SAW:** 0 **not SAW, Client Wrap, VAW:** $c_R \cdot join(M_L, L_R)$ **not SAW, Client Wrap, not VAW:** $c_r \cdot \big(r_L + join(M_L, L_R)\big)$
COM	$COM_L + COM_R$+a cost per source. For the right source this is given by: **Server Wrap or SAW:** $c_r \cdot \big(d(L_R) + d(jsf_R \cdot L_R)\big)$ **not SAW, Client Wrap, VAW:** $c_R \cdot d(M_L)$ **not SAW, Client Wrap, not VAW:** $c_r \cdot d(N_L)$

For several criteria in Table 7 the cost is only formulated for the right component. The left component can be formulated by switching the L and R prefixes and the order of arguments in the join function. It has been omitted here to increase readability.

The staleness cost for immediate join policies is the maximum of the staleness for each source. These always include the staleness for the supporting view plus the cost of merging changes into the view (merge(jsf_R*L_R)). In addition to these common costs there is a component which depends on source capabilities. When the source is SAW, the cost added is for sending the changes from the right source (d(L_R)), joining with the left source (join(M_L, L_R)) and sending the join tuples (d(jsf_R*L_R)). If the source is not SAW, the cost of a join depends on wrapper localisation and view awareness (VAW). When the wrapper can be located in the server, staleness is the delay to send the changes, recompute and extract the supporting view (r_L+e(M_L)), join and send the changes. When the wrapper is located in the client, staleness is the delay to extract and send the supporting view (if the source is VAW) or extract and send the whole source (when the source is not VAW). As before, there is a delay to recompute the view and join it with changes.

Response time is zero with immediate join. Source and warehouse storage have no cost in addition to the SVP cost.

As with staleness, source and warehouse processing depend on SAW, VAW and wrapper localisation. In all cases the cost has a component from the SVP. When the source is SAW, source processing involves joining changes from one source with the other whenever the first is changed. When the source is not SAW and the wrapper is located in the server, the cost is to recompute the supporting view, extract it and perform the join. When the wrapper is located in the client, it is to recompute the view and extract it (source is VAW) or to extract the whole source (source is not VAW).

For warehouse processing there is always a cost component from the SVP and the cost of merging changes into the view. When sources are SAW and/or when the wrapper can be located in the server, there is no additional cost. When the wrapper is located in the client, there is the cost of recomputing the view (if the source is not VAW) and of performing the join.

Communication incurs a cost from the SVPs. In addition there is a cost that depends on source capabilities. If the source is SAW and/or when the wrapper is located in the server, changes have to be sent to the source and join tuple sent back. If the source is not SAW and the wrapper is located in the client, each change introduces a cost to send the whole source (when it is not VAW) or the whole supporting view (when the source is VAW).

The cost formulations for on-demand join, shown in Table 8, are similar to those for immediate join, with the differences discussed below.

Table 8. On-demand incremental integrator policy without auxiliary views

Z	$max(Z_L, Z_R) + merge(jsf_L \cdot L_L + jsf_R \cdot L_R)$+a cost per source. For the right source this is given by: **SAW:** $d(L_R) + join(M_L, L_R) + d(jsf_R \cdot L_R)$ **not SAW, Server Wrap:** $d(L_R) + r_L + e(M_L) + join(M_L, L_R) + d(jsf_R \cdot L_R)$ **not SAW, Client Wrap, VAW:** $r_L + e(M_L) + d(M_L) + join(M_L, L_R)$ **not SAW, Client Wrap, not VAW:** $e(N_L) + d(N_L) + r_L + join(M_L, L_R)$
RT	$max(RT_L, RT_R) + merge(jsf_L \cdot L_L + jsf_R \cdot L_R)$+a cost per source. For the right source this is given by: **SAW:** $d(L_R) + join(M_L, L_R) + d(jsf_R \cdot L_R)$ **not SAW, Server Wrap:** $d(L_R) + r_L + e(M_L) + join(M_L, L_R) + d(jsf_R \cdot L_R)$ **not SAW, Client Wrap, VAW:** $r_L + e(M_L) + d(M_L) + join(M_L, L_R)$ **not SAW, Client Wrap, not VAW:** $e(N_L) + d(N_L) + r_L + join(M_L, L_R)$
SS	$SS_L + SS_R$
WS	$WS_L + WS_R$
SP	$SP_L + SP_R$+ a cost per source. For the right source this is given by: **SAW:** $q \cdot join(M_L, L_R)$ **not SAW, Server Wrap:** $q \cdot \big(r_L + e(M_L) + join(M_L, L_R)\big)$ **not SAW, Client Wrap, VAW:** $q \cdot \big(r_L + e(M_L)\big)$ **not SAW, Client Wrap, not VAW:** $q \cdot e(N_L)$
WP	$WP_L + WP_R + q \cdot merge(jsf_L \cdot L_L + jsf_R \cdot L_R)$ +a cost per source. For the right source this is given by: **Server Wrap or SAW:** 0 **not SAW, Client Wrap, VAW:** $q \cdot join(M_L, L_R)$ **not SAW, Client Wrap, not VAW:** $q \cdot \big(r_L + join(M_L, L_R)\big)$
COM	$COM_L + COM_R$+a cost per source. For the right source this is given by: **Server Wrap or SAW:** $q \cdot \big(d(L_R) + d(jsf_R \cdot L_R)\big)$ **not SAW, Client Wrap, VAW:** $q \cdot d(M_L)$ **not SAW, Client Wrap, not VAW:** $q \cdot d(N_L)$

The staleness cost for on-demand is determined by the maximal staleness of the SVPs plus the delay for handling changes from the sources. This is different from immediate, which has a worst case of handling each source individually. Response time differs in the cost formulation only in that worst case response time is used instead of worst case staleness. This can be a significant difference with incremental maintenance. If SVPs are periodic, for example, max(Z_L, Z_R) will be at least the maximum period of the wrappers, while max(RT_L, RT_R) will be zero.

For other costs, immediate and on-demand differ only in that c_R (and c_L) are replaced by q, and merging is done once, with cost based on the sum of the change sizes; cost formulations are otherwise similar. For example, there is the same dependency on source characteristics.

The cost formulations for periodic join without auxiliary views are shown in Table 9.

Table 9. Periodic incremental integrator policy without auxiliary views

Z	$max(Z_L, Z_R) + merge(jsf_L \cdot L_L + jsf_R \cdot L_R) + 1/p$+a cost per source. For the right source this is given by: **SAW:** $d(L_R) + join(M_L, L_R) + d(jsf_R \cdot L_R)$ **not SAW, Server Wrap:** $d(L_R) + r_L + e(M_L) + join(M_L, L_R) + d(jsf_R \cdot L_R)$ **not SAW, Client Wrap, VAW:** $r_L + e(M_L) + d(M_L) + join(M_L, L_R)$ **not SAW, Client Wrap, not VAW:** $e(N_L) + d(N_L) + r_L + join(M_L, L_R)$
RT	0
SS	$SS_L + SS_R$
WS	$WS_L + WS_R$
SP	$SP_L + SP_R$+ a cost per source. For the right source this is given by: **SAW:** $p \cdot join(M_L, L_R)$ **not SAW, Server Wrap:** $p \cdot \big(r_L + e(M_L) + join(M_L, L_R)\big)$ **not SAW, Client Wrap, VAW:** $p \cdot \big(r_L + e(M_L)\big)$ **not SAW, Client Wrap, not VAW:** $p \cdot e(N_L)$
WP	$WP_L + WP_R + p \cdot merge(jsf_L \cdot L_L + jsf_R \cdot L_R)$ +a cost per source. For the right source this is given by: **Server Wrap or SAW:** 0 **not SAW, Client Wrap, VAW:** $p \cdot join(M_L, L_R)$ **not SAW, Client Wrap, not VAW:** $p \cdot \big(r_L + join(M_L, L_R)\big)$
COM	$COM_L + COM_R$+a cost per source. For the right source this is given by: **Server Wrap or SAW:** $p \cdot \big(d(L_R) + d(jsf_R \cdot L_R)\big)$ **not SAW, Client Wrap, VAW:** $p \cdot d(M_L)$ **not SAW, Client Wrap, not VAW:** $p \cdot d(N_L)$

The difference between periodic and on-demand is that staleness has an additional 1/p cost, response-time is 0 and q is replaced with p in the formulations for SP, WP and COM.

3.1.5. Assumptions

In modelling the cost of join-view maintenance for all 134 policies we have been forced to abstract away from some of the fine-level details. In this section we present all assumptions made related to the cost model. We leave it for future work to address whether these assumptions might usefully be relaxed.

When computing staleness we use the average size of changes. This implies that worst case cost is sometimes slightly underestimated. To give an example, if we have periodic maintenance every 11 seconds and the source is updated every 6 seconds with ten objects, the worst case size should be 20 (2 changes before each round of maintenance) but we will instead estimate 10*11/6=18 objects.

We do not consider the potential option to avoid maintenance when no changes have occurred. If, for example, the wrapper is sending changes every 6 seconds and the integrator performs a join every

5 seconds, there will be situations in which no changes have been reported from the wrapper. This means that the computation cost will be slightly overestimated.

For incremental policies without auxiliary views it is assumed that all changes reported from a source will have to be sent to the other for joining. This is a pessimistic assumption. For example, deletes are self-maintainable and do not have to be included in a join query. It would be possible to introduce a component to model the fraction of changes that require a join query. We feel this will make the model overly complex. We again make a pessimistic assumption that all changes will have to be joined with the other source.

Cost formulations for immediate integrator policies are optimistic in that they assume that activations from the two wrappers never overlap. This is not appropriate when changes are reported frequently.

For on-demand and periodic incremental integrator policies, we make a pessimistic assumption that no parallelism can be utilised when sending join queries. This means that these queries are sent to the two sources sequentially.

When integrator timing is not immediate, and the SVP is periodic or immediate, there is an increased DW storage overhead to cache changes sent to the integrator. We currently ignore this cost. The same applies to the temporary additional storage required for incremental policies not using an auxiliary view. When the source is not SAW, the supporting view needs to be stored temporarily to perform a join.

We moreover use an optimistic assumption for incremental policies without auxiliary views. It is assumed that no additional compensating queries are required to ensure consistency. This gives these policies a lower cost. Put alternatively, they give a lower degree of consistency.

In addition to consistency, consideration of multiple sources also introduces the issue of how to handle several source systems. It is possible that different systems have different costs. The

communication channels may, moreover, vary between sources. This means they can have different capacities and costs. In our cost-formulation the source system costs are added together and communication is treated as one component. The reason for this simplification is to limit the number of criteria; the set of criteria is already relatively large (seven different components). Any oversimplifications will be picked up in testbed validation of the model.

3.2. Maintenance Policy Comparison Using PAM

The purpose of the cost model developed for multi-sources is to allow analysis and comparison of policies, using a tool (or an optimiser), based on the specification of source and warehouse characteristics. Modelling of costs as composable functions allows one to use the tool for different heterogeneous sources by customising cost functions (e.g. join() and merge() described above) appropriately. PAM allows users to specify source and warehouse characteristics and policies. Using the cost model, PAM will compute the relative costs of policies. The characteristics of the view and sources correspond to the parameters of the cost model. PAM allows for automatic comparison of policies using any combinations of staleness, response time and system costs (storage, processing and communication). A user can select any parameters of the cost model and PAM will automatically vary these and produce all combinations of parameter configurations (cases). For each such case the policy with the lowest cost will be recorded. When all cases have been tested the result will be presented to the user. For each policy PAM will present the percentage of cases in which this policy has the lowest cost. This allows us to identify policies that are optimal in many cases and those that are never optimal. PAM allows a systematic exploration of a slice or all of the search space to investigate whether a policy is meaningful for certain parameter settings.

PAM can vary parameters over a range if reliable estimates are not available. If a user, for example, knows that query and update frequencies are unreliable, these can be marked as such and PAM will automatically vary them. Currently, three different values are used for numerical parameters and two different values for Boolean parameters. In the rest of this section we discuss details of the tool. PAM can be accessed from http://www.his.se/ida/~henrik/research/

3.2.1. The Functionality of PAM

The tool makes it possible to define a join view based on two sources. The characteristics of the view and sources correspond to the parameters of the cost model. Figure 2 shows the dialog in which the view and sources are specified.

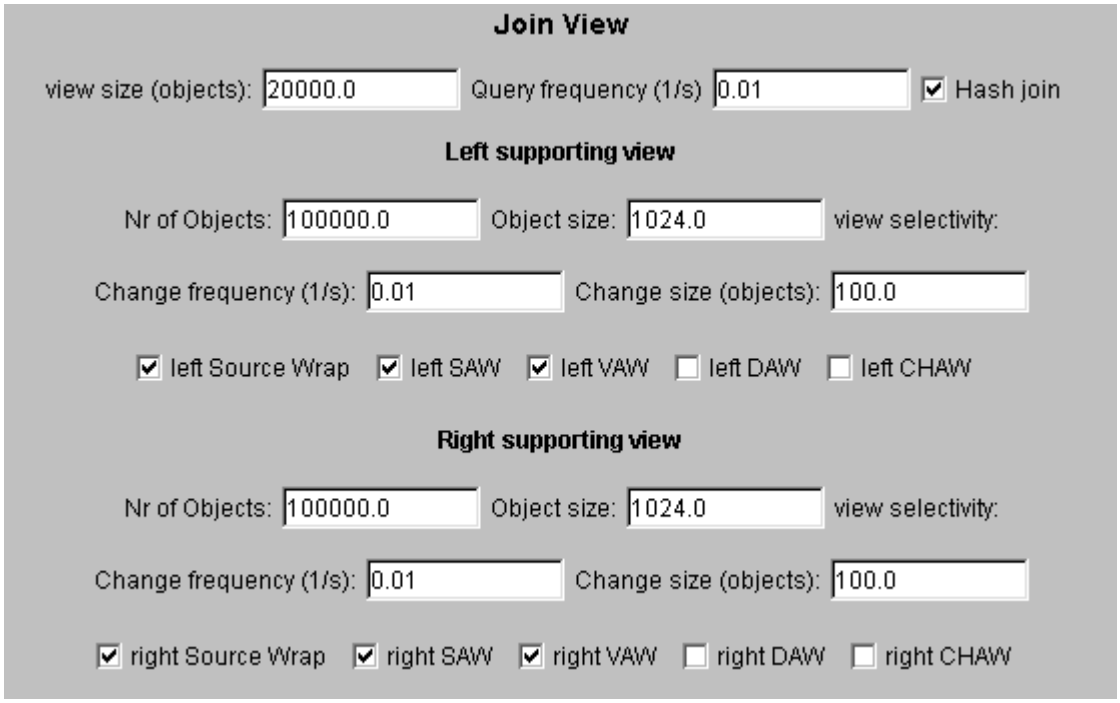

Figure 2. Configuring the parameters for a join view

The join is defined over two supporting views, referred to here as the left and right supporting view. For each of these it is possible to specify the number of objects in the source, the average size of objects, the view selectivity (the fraction of the source that constitutes the supporting view), the change frequency and the average number of objects affected by a change. Checking the corresponding boxes specifies the source capabilities provided. Each source can have the wrapper located in the server (checked) or in the client (unchecked), and can have any combination of SAW, VAW, DAW, and CHAW. Change activeness, CHAC, is not included in the interface to the tool. CHAC is a necessity for using an immediate SVP, but as this is not captured within the cost model the tool does not consider it. For a specific case it is obviously important to consider this aspect. If a source is not CHAC, no policies using immediate SVP could be considered.

Note that although object sizes are not used as explicit parameters of the cost model, they can still be used in the interface of the PAM tool.

The join view is specified through the size of the view, the query frequency and whether the join can be done using a hash-join technique. Size is given as the number of objects in the view. The size of these objects is the sum of the sizes of the supporting view objects. The hash-join property has been included to model the potentially big difference in processing cost between different join views. For example, equijoin can be implemented using hash-join, which has an O(n+m) cost [Mis92], while other types of join are significantly more expensive. The worst case join is a nested loop join which has an O(n*m) cost. If hash join is not checked in the dialog box, the tool will assume it is a nested loop join and apply the corresponding cost-functions.

In a second dialog, policies for analysis can be defined. Each defined policy is added as a panel (named "p1", "p2" and so on) as shown in Figure 3.

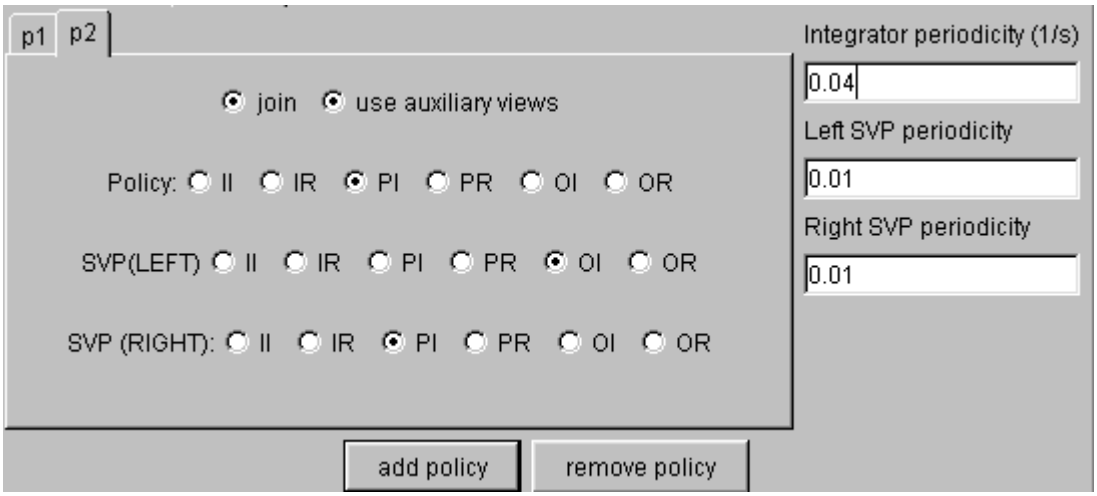

Figure 3. Specifying join policies

All 134 policies described in section 2.3 are supported, and the user can define the subset of policies to be analysed.

A join policy is defined by specifying a policy for each SVP ("SVP(LEFT)" and "SVP(RIGHT)" in the figure) and for the join ("Policy" in the figure). For each of these there is a choice between I_I, I_R, P_I, P_R, O_I and O_R. The user specifies which of these by marking the corresponding checkbox. The first field in the panel ("join") is used to specify whether the policy is a join policy or not. If it is not a join policy it will be treated as a single source over the left source. The second field ("use auxiliary views") specifies whether auxiliary views should be kept in the integrator.

The periodicities for periodic policies are specified to the right in the dialog. Individual values can be given for the integrator and each supporting view. These are obviously only applicable when a periodic policy is used.

Whenever a policy is changed, the tool will verify it according to the restrictions discussed in section 2.3. If an invalid combination is detected the user is notified through a dialog. An example of this is shown in Figure 4, where an on-demand SVP is combined with immediate join.

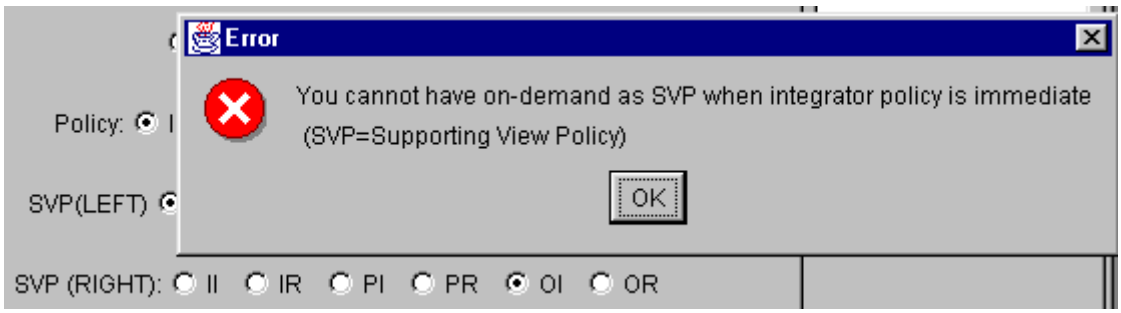

Figure 4. The tool enforces restrictions to join policies as discussed in section 2.3

Once the parameters and policies have been defined it is possible to compare the costs of policies. The tool supports various types of comparison. Figure 5 shows one example, where the cost of selected policies can be plotted as a function of a selected numerical parameter.

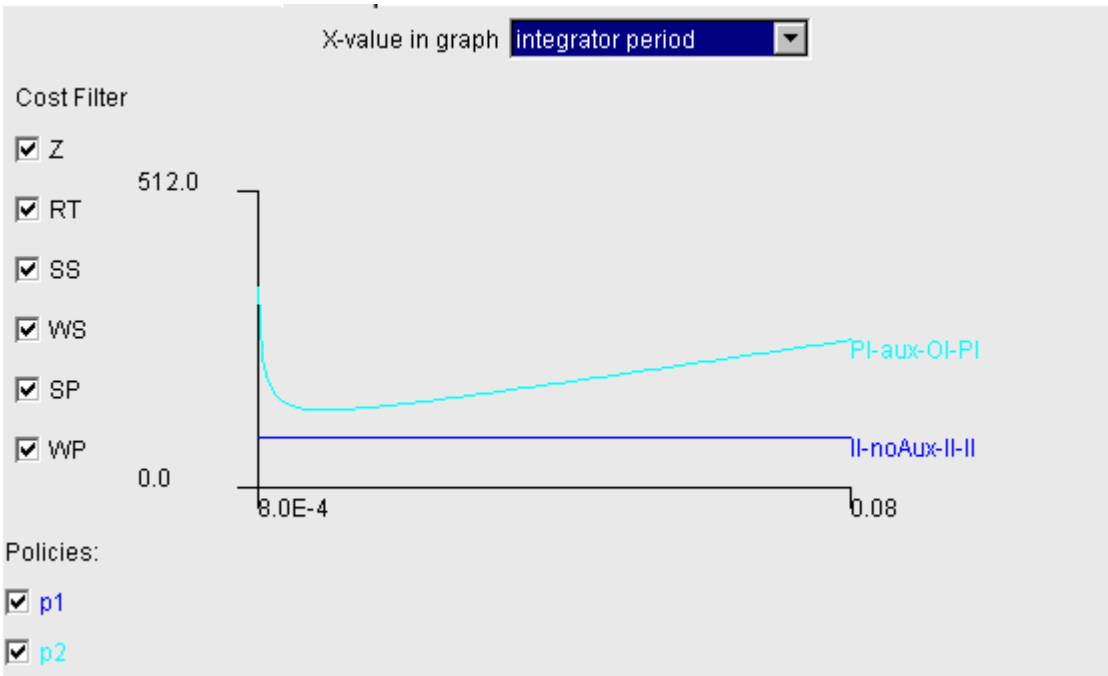

Figure 5. Analysing how policy costs depend on parameters

The user can select which policies to plot by checking the policy names at the bottom of the window. The computed cost is a weighted sum of evaluation criteria. Only criteria selected (to the left

of the diagram) will be included in the sum. The choice at the top lets the user select which parameter to vary in the plot. Any numerical parameter in the cost model can be selected and it will be varied from 0 to twice its specified value (in this case the periodicity 0.04 in Figure 3).

Different colours are used for the cost-curves in the diagram to distinguish policies. The name of the policy is, in addition, written to the right of the curve.

In addition to this single value comparison of policies, PAM allows for analysis of the cost of one policy as a function of two parameters. The cost is computed for a number of points in a 2-dimensional grid. These values are written to file and may then be plotted by an external application. Figure 6 shows an example of this where a spreadsheet program has been used to create the graph.

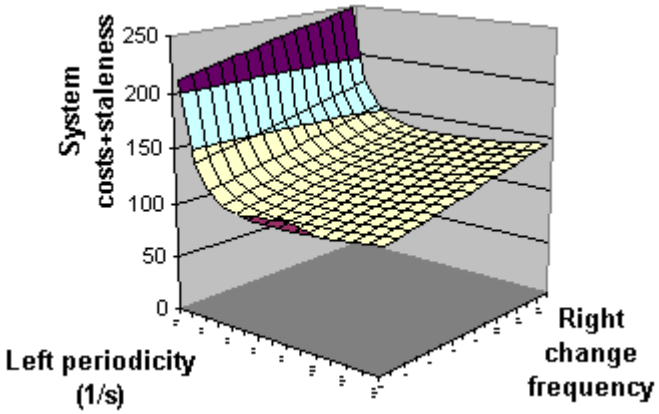

Figure 6. The cost (system+Z) of I_I-aux-P_I-I_I as a function of periodicity and change frequency

3.2.2. Automatic Comparison

Although the graphs plotted are useful for understanding dependencies on individual parameters, it may still be difficult to derive policies that are the most likely to be useful. This kind of analysis can be done using another capability of PAM, where the policy with the lowest cost is derived for a systematic set of parameter values. Figure 7 shows the dialog for initiating this type of analysis.

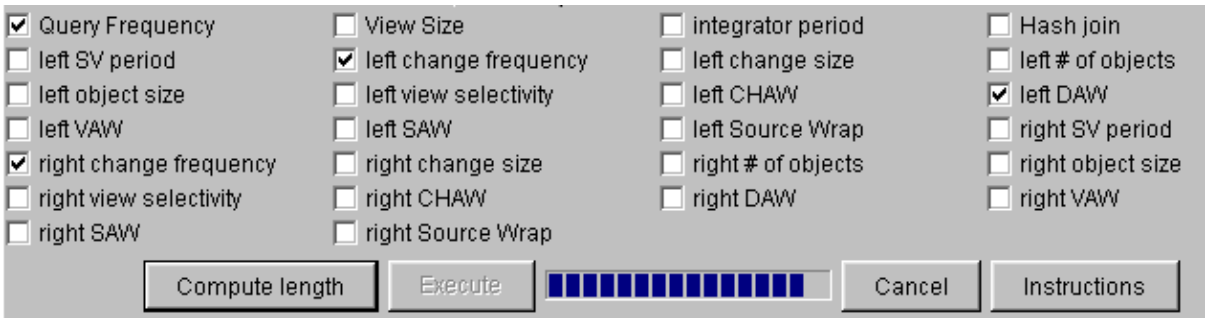

Figure 7. Exploring the search space for policy selection

The dialog contains all parameters of the cost model, numerical and Boolean, including the periodicities of periodic policies. A user can check which parameters to vary in the analysis. For each checked numerical value the tool will use the specified (base) value, and half and double that value. For the example shown here it means that query frequency will take the values 0.01, 0.005 and 0.02. The base value of 0.01 will have been specified in the dialog shown in Figure 2. Each Boolean value checked in Figure 7 will take the values true and false. When the user presses the "Execute" button, the tool will analyse all possible combination of these values. The cost of each policy (filtered according to the filter in the Graph-tab – see Figure 5) will be computed for each combination, and the policy with the lowest cost recorded. Once all combinations have been tested, the result will be presented to the user as shown in Figure 8.

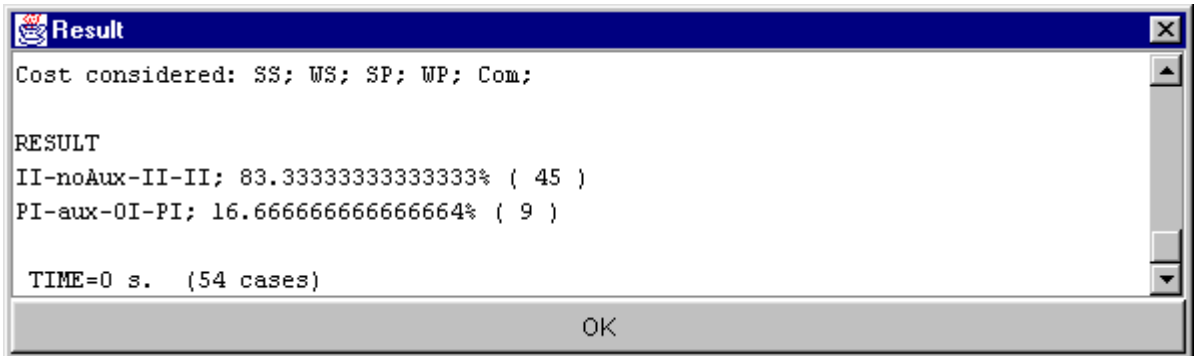

Figure 8. The result of a comparison

For each defined policy the tool presents the percentage of cases in which that policy had the lowest cost. It also presents the number of configurations this corresponds to (in parentheses). Finally, the execution time and total number of cases (parameter variations) is presented. The result of the analysis in Figure 8 is that I_I-noAux-I_I-I_I was found to have the lowest cost in 83% of the cases. There were 54 different cases in total, this being the result of combining three numerical parameters (3 values each) with a Boolean parameter (DAW). Obviously there is a practical limitation to the number

of parameters that can be included in an analysis. There is a combinatorial explosion which makes it impossible to include all parameters. For this reason, the tool provides users with the computed number of combinations and an estimation of the required processing time (which obviously depends on the computer on which the tool is executing). Figure 9 shows an example of this.

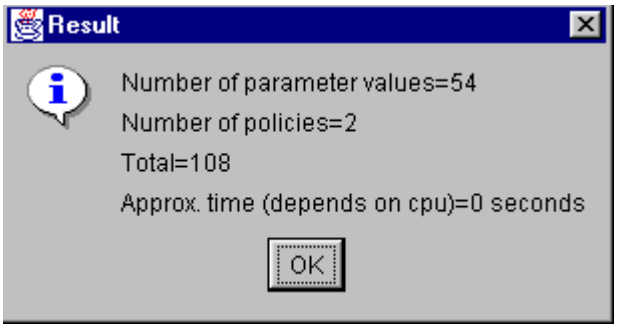

Figure 9. A dialog presenting the size of the selected search space

Although the analysis may include billions of individual costs it is still limited to a small fraction of the full search space. The given variation of numerical parameters implies that analysis will include samples in an interval about the given value. This may be useful for real scenarios where precise estimations are not available. In this way users can select unreliable parameters to be varied. Source capabilities, on the other hand, may well be known in advance and so fixed.

3.2.3. Modelling Costs

Supporting view costs have been modelled as in [Eng02]. For join cost (the join() function described above) PAM currently provides a choice between two cost-functions, one for hash-join and the other for nested loop join. Hash-join is modelled to have a cost of 3*(N+M)*k (where k is a constant and N and M are the number of objects). Nested loop join is modelled to have a cost of N*M*k. These algorithms represent the boundary conditions for join, and the user of PAM may select any of them to analyse the impact of the join algorithm. For incremental policies, merge cost, i.e. the cost of incorporating changes into the join view, is modelled as a scan through the view and a scan through the view delta. For our analysis we have set the constant k to 0.00004 in all of these formulas. This value was chosen to model a cost somewhere between that of disk-based (>1ms) and main memory operations (<1μs).

3.3. Results from PAM

PAM has enabled us to analyse and compare the set of possible multi-source maintenance policies. In this section we present the results of a systematic comparison of policies for a large number of situations.

3.3.1. Parameter Configuration

For our analysis, we use a join view with 20000 objects over sources each with 100000 objects (of size 1Kb). View selectivity of 0.2 results in supporting views of 20 Mbyte. The size of the join view is 40 Mbyte. Query frequency, update frequency, and periodicity are set to 0.01, which means these periods are 100 seconds long. Each change affects 100 objects in the view (= 0.1 Mbyte). The wrappers are located in the client and the sources are not CHAW.

We consider all possible policies, but as the left and right sources have identical configuration it is possible to reduce this set. Pairs of policies which are identical except for the order of SVPs are represented with only one policy. This means, for example, that only one of P_R-aux-I_R- P_R and P_R-aux-P_R-I_R is considered in the comparison. This leaves us with 84 different policies.

Ideally, all parameters of the cost model should be varied in the comparison, but this results in an extremely large search space. We have therefore been forced to vary only a subset of parameters. We have tried to identify parameters that are likely to have an impact on relative policy comparison. For the integrator, query frequency, view size, integrator period and join technique is varied. The following source parameters are varied independently for left and right sources: wrapper period, change frequency, source size, DAW and SAW capabilities. The source capabilities (four different) and join technique are Boolean valued which means they give rise to 2^5=32 different combinations. The remaining 9 parameters are numerical and take three different values, giving 3^9=19683 combinations. Totally there are almost 630000 different cases analysed in each evaluation. For each case, all 84 policies are compared.

3.3.2. Comparing All Policies

The first comparison, based only on **system cost,** showed that 76 of the 84 policies incurred the lowest cost in at least one of the cases explored. This means that only 8 policies were never selected. This is an important result in that it shows that almost all policies are meaningful. The evaluation

criteria used in this comparison do not include any quality of service aspects. This is advantageous for on-demand policies, which have an increased response time, and for periodic policies, which have an increased staleness. Another interesting result is that there is no clear winner when ranking policies by fraction of cases for which each is selected. First ranked is I_I-aux- P_I-P_I, selected in 7% of cases. The policies that follow in the ranking show great variation in SVP and integrator policy. Figure 10 (left) shows the 10 most-selected policies.

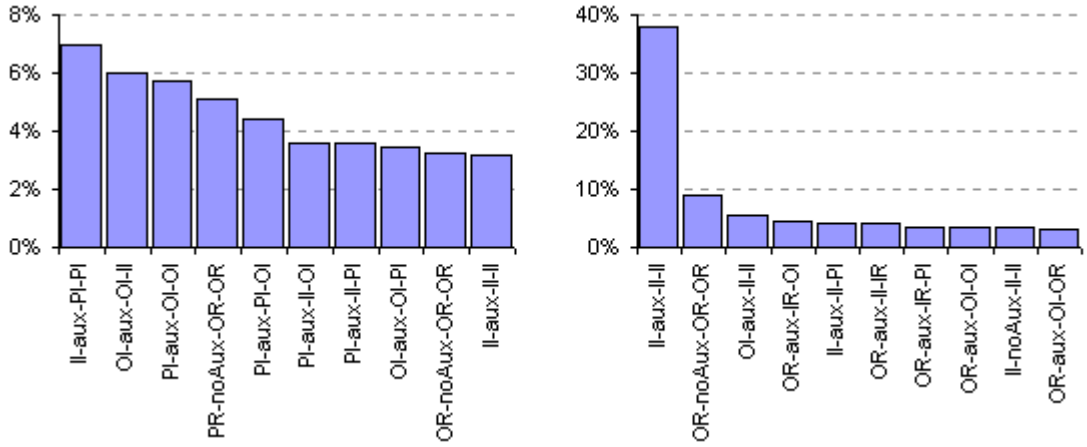

Figure 10. The 10 most-selected polices when evaluation is based on only system costs (left) and a combination of system costs and staleness (right)

Both recompute and incremental policies are selected. Among the policies which are never selected, a majority have recompute SVP using auxiliary views. As sources are not CHAW, these policies differ from recomputed policies without auxiliary views only in having increased storage cost; this explains why they are never selected. It is important to note that when comparison is based only on system cost, policies that do as little maintenance as possible benefit. In principle, the optimal policy is a periodic policy with an extremely low periodicity.

The second analysis therefore included a QoS component. Figure 10 (right) shows the 10 most-selected policies when comparison is based on combined **system** and **staleness** costs.

Here I_I-aux-I_I-I_I is a clear winner, having the lowest cost in almost 40% of the cases. It should be noted, however, that this policy requires both sources to be CHAC. If some source does not have CHAC capability, O_R-noAux-O_R-O_R seems to be the best choice. From Figure 10 we can also note that none of the policies shown uses periodic integrator policy, and very few have a periodic SVP.

Periodic policies have an increased staleness due to the periods. Of the 29 policies never selected, 28 have a periodic component.

When we compare the left and right diagrams in Figure 10 it is clear that the evaluation criteria have a big impact on the selection of policies. For this reason, the same comparison has been repeated for various combinations of evaluation criteria. The details of these comparisons can be found in Appendix A. We present a summary in Table 10. Starting with the right-most column, it is apparent that immediate SVPs are preferable for low staleness. This is true in all comparisons where staleness is included in the criteria. Again, it should be emphasised that this requires the source to be CHAC. Among the 22 policies selected when only staleness is considered, only 5 that did not have any immediate SVP.

Table 10. A summary of policy selection against different criteria

	System	System+ Z	System+ RT	All criteria	Only Z
Nr of policies selected	76	55	47	33	22
Most selected policy	I_I-aux-P_I-P_I	I_I-aux-I_I-I_I	I_I-aux-P_I-P_I	I_I-aux-I_I-I_I	I_I-aux-I_I-I_I
% of the cases	6.94%	37.98%	11.52%	47.59%	31.34%
Secondly most selected	O_I-aux-O_I-I_I	O_R-noAux-O_R-O_R	P_I-aux-O_I-O_I	I_R-aux-I_I-I_R	I_R-aux-I_I-I_R
% of the cases	6.03%	8.99%	7.79%	11.21%	16.67%

For response time it is clear that an on-demand integrator policy becomes less attractive. It was not meaningful to compare policies based only on response time, as all periodic and immediate policies have the same cost.

When analysing the above, some policies stand out as less useful in that they are found to be optimal in none or very few cases. These include:

- Incremental maintenance without auxiliary views; especially when using different SVPs
- Periodic integrator policy with at least one periodic SVP. These give a double penalty to staleness

Among the remaining policies, it is difficult to see any obvious winner. To do a deeper analysis, we selected the top two policies for each evaluation criterion shown in Table 10. We then repeated the comparison above, using only the six selected policies. The results of these comparisons against three different sets of criteria are shown in Table 11.

Table 11. Comparison of top six policies

System		System+Z		All	
P_I-aux-O_I-O_I	35.78%	I_I-aux-I_I-I_I	57.45%	I_I-aux-I_I-I_I	71.13%
O_I-aux-O_I-I_I	20.51%	O_R-noAux-O_R-O_R	17.82%	I_R-aux-I_I-I_R	22.44%
I_I-aux-P_I-P_I	19.60%	I_R-aux-I_I-I_R	12.82%	I_I-aux-P_I-P_I	3.38%
I_I-aux-I_I-I_I	9.24%	O_I-aux-O_I-I_I	9.75%	P_I-aux-O_I-O_I	1.53%
I_R-aux-I_I-I_R	7.62%	I_I-aux-P_I-P_I	1.76%	O_R-noAux-O_R-O_R	1.45%
O_R-noAux-O_R-O_R	7.25%	P_I-aux-O_I-O_I	0.40%	O_I-aux-O_I-I_I	0.08%

Again, there is a notable variation in selection. All policies are selected sometimes for each set of criteria. An interesting observation is that, when considering only six rather than all policies, rank order is not preserved. As an example, when only system cost is considered I_I-aux-P_I-P_I was ranked first among the 84 policies. P_I-aux-O_I-O_I was ranked third. As can be seen in Table 11, when only six policies are considered P_I-aux-O_I-O_I is clearly the most selected policy while I_I-aux-P_I-P_I is ranked third. An explanation for this is that when we remove 78 policies the cases where one of these had the lowest cost will now be replaced by one of the remaining six. P_I-aux-O_I-O_I replaces these policies in more cases than the other policies, and becomes the most selected policy. This illustrates the complexity of the selection process and why a careful analysis is required to understand these tradeoffs.

3.3.3. Effect of Join Strategy and Source Capabilities

For a single source view it has been shown that the recompute view maintenance choice is dependent on the size of changes and the change detection capabilities of the source. If the source is not DAW, recompute view maintenance may be less expensive than incremental [Eng02].

For a join view this situation is naturally more complex, as each source can have any combination of capabilities and the join can be more or less expensive. In order to analyse the effect of the join algorithm, we drilled-down to understand the effect of hash-join on policy selection. To do this, we split the comparison shown in Table 11 into two cases, one where we always use hash-join, and the other where nested loop is used. The result of this analysis is shown in Table 12.

Table 12. System cost comparison of policies for 2 different join algorithms

Policies	Hash	Nested loop
P_I-aux-O_I-O_I	30.88%	40.68%
O_I-aux-O_I-I_I	13.93%	27.08%
I_I-aux-P_I-P_I	18.24%	20.96%
I_I-aux-I_I-I_I	7.21%	11.27%
I_R-aux-I_I-I_R	15.25%	0.00%
O_R-noAux-O_R-O_R	14.49%	0.00%

The value shown in Table 11 (left column) is the average of the two values shown in Table 12. It is clear that a requirement for recompute join to be selected is that the join can be computed at "low cost". With hash-join, recompute joins are selected in 30% (15.25+14.49) of the cases when the comparison is based on system costs. With nested loop join, incremental integrator policy is always selected.

For single source maintenance the choice between incremental and recompute is dependent on DAW. To further analyse how DAW capability impacts join selection, we further drilled-down the left column in Table 12. For each combination of DAW capability we repeat the comparison. The result of this is shown in Table 13.

Table 13. Selection based on system cost when join view is computed with hash-join

Policies	None DAW	Left DAW	Right DAW	Both DAW
P_I-aux-O_I-O_I	23.08%	28.44%	33.78%	38.24%
O_I-aux-O_I-I_I	1.30%	2.56%	26.34%	25.52%
I_I-aux-P_I-P_I	12.26%	18.36%	17.74%	24.60%
I_I-aux-I_I-I_I	0.00%	0.00%	17.21%	11.63%
I_R-aux-I_I-I_R	19.88%	41.11%	0.00%	0.00%
O_R-noAux-O_R-O_R	43.49%	9.54%	4.94%	0.00%

The average value for each row corresponds to the hash-join value in Table 12. We see that for O_R-noAux-O_R-O_R, which is selected in 14.49% of the cases when the join is hash-based, there is a clear correlation with DAW. If no source is DAW, it is the most selected policy. If one source is DAW (left or right) it ranks outside the top three. When both sources are DAW, it is never selected.

The other policy with a recompute component, I_R-aux-I_l-I_R, is selected frequently when the left source is DAW. The reason for this is that it makes use of the deltas from the left source to

incrementally update the left auxiliary view. When the left source is not DAW, other policies always have a lower cost.

To conclude, to be selected recompute integrator policies require hash-based joining. In addition, at least one of the sources should not be DAW. From now on, we exclude recompute policies and concentrate on analysing the choice among incremental policies.

First we repeated the analysis of the impact of joining techniques, shown in Table 12 above, but excluding recompute policies. Interestingly, no significant difference in policy selection is evident. In other words, joining technique does not impact on relative comparison of incremental policies. This is an important result in that it makes join-cost modelling less critical. For incremental policies, P_I-aux-O_I-O_I is the most selected followed by O_I-aux-O_I-I_I and I_I-aux-P_I-P_I. The least selected policy is I_I-aux-I_I-I_I. Here it is important to bear in mind that the comparison is based only on system cost. Immediate integrator policy has an inherent handicap in that it updates the view for each reported change from either wrapper. With periodic or on-demand timing, changes from both sources can be merged into the view at the same time, reducing cost.

When comparison is based on the combined staleness and system cost the situation is different. Then policies with periodic components are penalised with their increased staleness cost (due to the period) and I_I-aux-I_I-I_I is selected in more than 50% of the cases (as shown in the middle column of Table 11). O_I-aux-O_I-I_I is the second-most selected policy. It has different timings, and is selected when query frequency is lower than update frequency.

It is important to remember that to use a policy with immediate SVP the corresponding source has to be CHAC. For I_I-aux-I_I-I_I this means that both sources have to be CHAC, a strong requirement. For O_I-aux-O_I-I_I the right source has to be CHAC. If this is not the case, we have to choose another policy. In this case only two policies remain, namely P_I-aux-O_I-O_I and I_I-aux-P_I-P_I, and these are very similar: they differ only in join timings. When closely analysed, P_I-aux-O_I-O_I, which is integrator driven, has a slightly lower system cost while I_I-aux-P_I-P_I, which is wrapper driven, has a lower staleness.

Irrespective of criteria, SAW does not have any impact on selection between the six policies considered above. This is as expected, as SAW only impacts on incremental policies without auxiliary views. To analyse the impact of SAW we have compared P_I-aux-O_I-O_I with P_I-noAux-O_I-O_I. The P_I-aux-O_I-O_I policy was found to be preferable in all situations except when both sources are SAW. When further analysed, it became clear that the absolute difference in cost between policies varied significantly depending on the join technique and SAW. P_I-aux-O_I-O_I can have a significantly lower cost than P_I-noAux-O_I-O_I. This is an important result, as it shows that the difference between keeping auxiliary views and not can be much more than only an increase in storage space.

3.4. Developing Heuristics

The number of potential policies for maintenance of a join view is large, and no single policy can be identified as a clear winner. Although tools such as PAM may support policy selection, the goal is often not so much to find an optimal policy as to avoid really bad policies. In this section we present a set of heuristics primarily based on analysis using PAM as described above, and experiences from single source maintenance [Eng02, Eng02b]. We then present a selection process based on these heuristics.

Analysis indicates that selection of SVP cannot be made independently of integrator policy. To obtain low staleness, it is important that integrator activity and wrapper activity be synchronized. If wrappers detect changes eagerly and send them to the integrator, this work may be wasted if the integrator simply stores them for later use. For this reason, it is preferable to use policies which have integrator activity synchronised with wrappers. A policy using different SVPs can match the characteristic of each source to give "optimal" performance. The prediction of the cost-model suggests that the benefit of such policies is small in terms of absolute cost, however. As the heuristics are not aimed at finding the optimal policy, such policies are not considered. The implications of this exclusion are small, as we will see in the performance evaluation presented in section 4.

We identify three important timings: I-I-I (immediate integrator policy and immediate SVPs), P-O-O (periodic integrator policy and on-demand SVPs), and O-O-O (on-demand integrator policy and on-demand SVPs). All three have a synchronisation between integrator and wrappers, and together

they cover a wide set of evaluation criteria and source capabilities. I-I-I has been shown to be optimal in most situations where staleness is considered. However, it requires both sources to be CHAC. If that is not the case, and if response time is important, P-O-O is a possible alternative. If response time is not important, O-O-O is to be preferred as it typically gives lower staleness. When staleness is not considered, the optimal policy is to avoid maintenance. This is based on insights from single source maintenance [Eng02, Eng02b]. In such cases P-O-O is to be preferred, as maintenance intensity can be set through periodicity. The other timings are initiated by events that are typically not controlled by the system designer.

The choice between incremental and recompute view maintenance is dependent on the size of changes, change detection capabilities and joining technique. The size of changes propagated to the integrator will be dependant on the sizes of changes committed in the sources as well as the timings used in the wrappers. This is a complex trade-off. The heuristic we use is that incremental maintenance should be used unless both sources are DAW and the join can be performed at low cost. There is one exception to this, namely when staleness is not considered. As mentioned above, the timing will be set to have very infrequent maintenance. If an incremental policy is used this implies that there will be a large number of aggregated changes once maintenance is initiated. With a recompute policy, cost is independent of the delay since last maintenance, motivating the use of a recompute policy in these cases.

Keeping auxiliary views is always a good idea unless space is a problem. If space is a problem then incremental policies should only be used when consistency is not important, and when both sources are SAW.

The above heuristics have been captured in the selection process shown in Figure 11. Based on quality of service requirements (consistency, staleness and response time) and source properties (CHAC, DAW, SAW) a policy is derived in two phases. First, a timing (when) is derived and then a strategy (how), i.e. whether auxiliary views should be used and whether maintenance should be done incrementally or through recompute.

By way of example, consider two sources, one of which is DAW and CHAC and the other CHAC only. Strong consistency with low staleness is required, but no requirements are imposed on response time or DW storage overhead. First we derive the timing. In this case it will be I-I-I, which means integrator and SVPs will use immediate timing. We then derive a strategy, which will be incremental with auxiliary views. Combining these will give us the policy I_I-aux-I_I-I_I. If we compare this with Figure 10 (right) we see that it corresponds to the most selected policy.

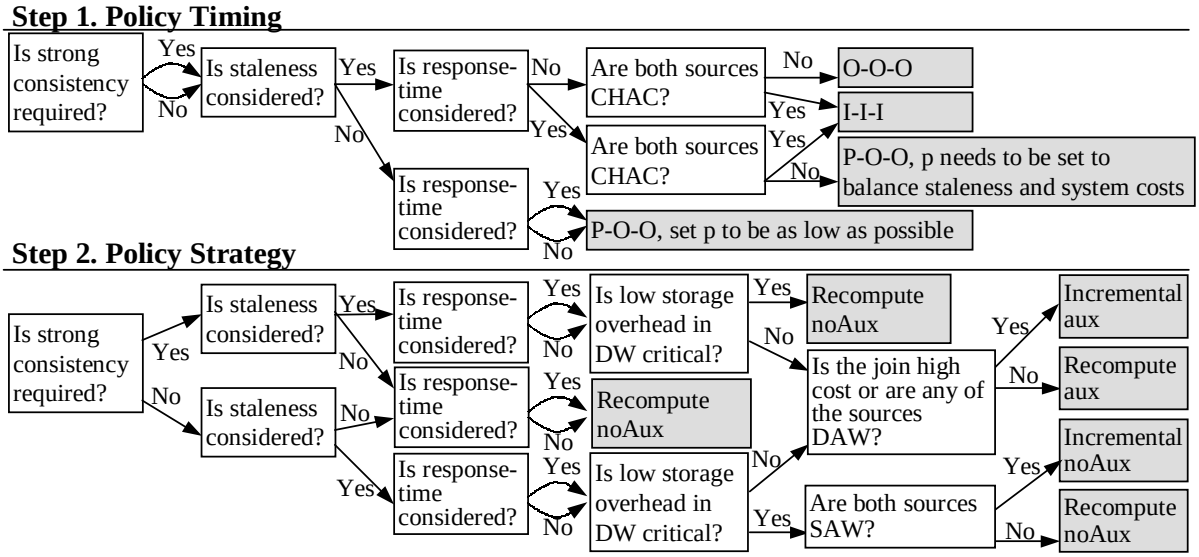

Figure 11. Two-stage selection process to select a join policy based on QoS

By using the selection process, one out of 12 possible policies (three timings, incremental or recompute, with or without auxiliary views) is selected. This is done in a straightforward and simple way. It remains to establish whether the developed heuristics lead to acceptable policy selection. We consider this in the next section, based on the performance of all policies measured in a testbed system.

4. Comparison of Selected Policies Using TMID

In this section we present TMID, a testbed implementation of view maintenance using sources such as a relational database and an XML repository. The system serves two purposes. Firstly, it is a proof of principle system to investigate the implementation and use of all 134 suggested policies. Secondly, it enables measurement of policy performance, both in terms of quality of service and system overhead. This makes it possible to empirically test the performance of a policy selected according to the

selection process of Figure 11 against all other possible selections. In this section we give an overview of TMID and present the results of experiments that have been conducted.

4.1. TMID

This section describes the architecture, functionality and implementation of TMID, a Testbed for Maintenance of Integrated Data. It describes how all identified join policies have been implemented and how various data sources can be used. The system is publicly available (at http://www.his.se/ida/~henrik/research/) and can be downloaded and used for multi-source view maintenance.

4.1.1. Overview

TMID is a fully functional data integration tool which can retrieve data from distributed, heterogeneous sources to be accessible for a client. The architecture corresponds to the one shown in Figure 1. The system is implemented as a distributed Java application and uses RMI for communication between components. Wrappers, integrators and other components can be located on any machine on the network. XML is the canonical data model used by the integrator. This means that data from various sources are wrapped and sent to the warehouse in XML format. The main components, from a maintenance perspective, are wrappers and the integrator. The wrappers can be tailored to handle different data sources.

Currently a data source can be an Interbase database or an XML web-server. Wrappers are responsible for providing the relevant data to the integrator according to the SVPs. They can be configured to make use of available source capabilities (such as DAW) or compensate for them (for example, DAW can be compensated for by comparing snapshots of the view). The integrator is the most complex component, in that it is responsible for synchronizing wrapper activity, and computing and maintaining the join view. Currently it is possible to use either hash-based or nested loop join in the integrator.

Although the system can be used for distributed data maintenance, the main purpose of TMID, so far, has been for measuring performance. Experiments can be defined and executed, and performance can be measured and recorded. For this purpose each data source will be updated by an updater component, and a querier will query the view. The three main collected metrics are: staleness,

response time and system overhead. By specifying a number of policies and source characteristics to be varied, the system can be configured to automatically execute experiments and log results.

Any combination of integrator and wrapper policies described in section 2.3 can be used for experiments. This includes incremental or recomputed join with periodic, immediate or on-demand timing and the possibility of using auxiliary views in the integrator. If auxiliary views are not maintained in the integrator, the incremental join algorithm will send join queries to the wrappers which will either forward them to the source (if it is SAW) or retrieve the supporting view and perform the join.

4.1.2. Architecture

TMID is highly dynamic in that components can be created, configured, and removed at runtime without having to reconfigure the participating hosts. It is even possible to add software components at runtime to interact with new data sources. All this is achieved through *Source Servers,* a program responsible for creating and killing nodes and configuring hosts. A source server needs to be started on each computer that should have nodes executing. If a data source does not have a source server running, it is possible to use it as a source but a wrapper has to be located on some other machine.

The main components of TMID are shown in Figure 12.

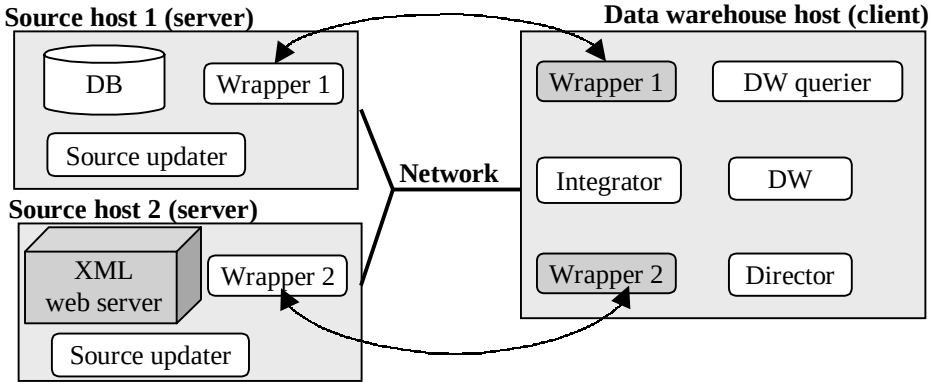

Figure 12. The main components of TMID when experiments are executed.

All communication between wrappers and integrator is handled using Java RMI, and data is represented as JDOM elements (http://www.jdom.org). Wrappers can be located either in the data source or in the warehouse client. The type of wrapper used will depend on the data source it communicates with. Currently there are wrappers for relational databases (Interbase and PostgreSQL) and XML web servers. Wrappers can be configured to use any combination of source capabilities. If a

source is specified as not having a capability (such as DAW or SAW) the wrapper will compensate for it.

The *integrator* is responsible for synchronising wrappers and joining supporting views. Depending on user settings it will also maintain auxiliary views. Through user settings it is also possible to choose whether a nested loop or a hash algorithm should be used for joins. Currently the integrator does not implement any query language to define join views. The join predicate is instead hard-coded into the component.

The *director* initiates experiments. This includes creating nodes, interconnecting them, starting experiments, collecting metrics, and finally recording results and removing all nodes. When experiments are executed, wrappers, updaters, integrator and querier communicate with the director component, which is responsible for time-stamping events. The delay caused by such message passing (remote method invocations) is measured and recorded. This makes it possible to estimate the precision of measurements.

Wrappers and integrator run as separate threads, and synchronization between these threads has been carefully analysed to avoid unnecessary blocking.

The *updaters* run as separate processes and execute updates to the data sources according to user settings. As with wrappers, there are different updaters for different data sources.

The *DW querier* component issues symbolic queries (with empty result) to the DW in a randomised way according to user settings.

In addition to this the system supports queries over the warehouse through Java RMI (resulting in a JDOM XML tree), or through a Tomcat web-server, which translates results using XSL into any presentation format.

4.1.3. Implementation of Policies

As mentioned above, all policies described in section 2.3 are supported in TMID. The integrator and wrappers cooperate to realise a policy. Before an experiment is started, each source wrapper will be initiated with the policy to use. This can be any of the six single source policies. For each periodic SVP the periodicity is specified. Periodic events will be generated internally in each wrapper, which means they will be completely independent of other events. An immediate SVP will be initiated by

signals from the source, which requires the source to be CHAC. An on-demand SVP will be initiated by requests from the integrator, which in turn may be activated periodically or on-demand.

A wrapper can be located in any node in the network. Typically, it is located either in the data source node or in the DW node. In the latter case it is possible to specify whether the wrapper should be run within the integrator process as a thread, or as a separate process. The main difference between these two is that with the former communication between wrapper and integrator can be done through shared memory, while the latter requires socket communication.

Each wrapper will operate according to the capabilities of the corresponding data source.

With periodic integrator policy there is a timer in the integrator which periodically initiates joining. If an SVP is on-demand the first thing the integrator does is to send a notification to the corresponding wrapper. If both SVPs are on-demand, requests are sent to both wrappers. The integrator then awaits results (or a notification that no maintenance is necessary). When the last wrapper (timings depend on source size etc) has notified the integrator, it invokes the join procedure. For periodic and immediate SVPs joins can be performed immediately. If changes are propagated from the wrappers while the view is being maintained, this is recorded as a potential inconsistency violation. This means inconsistency is detected but not prevented. A potential extension to guarantee strong consistency would be to queue changes to auxiliary views while the view is updated.

On-demand maintenance is similar to periodic except that joins are initiated by external requests. In the current system, these requests are generated by DW querier according to a given distribution (for example Poisson).

Immediate integrator timing means that joins are initiated whenever changes are propagated from the wrappers. These in turn may be using either immediate or periodic policies.

A hash algorithm or a nested loop algorithm can be used to compute the join. The nested loop algorithm is straightforward: each element in the left set is compared with each element in the right set using the defined join predicate. This algorithm can be used for any join.

The hash algorithm can be used for equijoins. It creates a hash map for one of the sets. This is done using a Java HashMap-object to map objects based on the join attribute. Once the hash map has been created, the integrator iterates through the other set and finds the set of matching objects through a look-up in the hash map.

For recomputes, a hash map is created for the smallest supporting view (something commonly done, according to Mishra and Eich [Mis92]).

When the join view is maintained incrementally, an algorithm is used based on the following:

- Deletion of an object causes all view-objects that use it to be deleted (based on ID)
- An update that does not alter the join attribute can be handled locally
- An update that alters the join attribute is treated as a delete followed by an insert
- An inserted object will be joined with the other source to find matching tuples

Among the objects in a delta from one source, some can be handled locally by using the view. The remaining objects need to be joined with the other supporting view. This can be done within the integrator if auxiliary views are used. Otherwise, the objects are sent to the corresponding wrapper where the join is conducted and the resulting view delta is sent back. The wrapper will perform the join differently depending on the SAW capability of the source.

The join algorithm makes use of several indexes implemented using hash maps. Firstly there is a mapping of view IDs to view elements. This is used for efficient merging of the view delta. There are also mappings of supporting view object IDs (left and right) to view objects. This is used to efficiently find view objects that are affected by a change (e.g. delete) to a supporting view object.

Changes to the supporting views and the join view are internally represented as hash maps. In this way only the net changes will be applied. The incremental join algorithm ensures that no inserted element is added if the same element is later deleted.

4.1.4. Implementation of Source Capabilities

A source can be more or less capable of participating in maintenance of join views. A source with SAW capability will have a query interface which understands the join-predicate. A relational source

is typically SAW for equijoin on an attribute. The query will be a select statement (select * from table where attrib='value'). It is possible to compose a select query for several join values (select * from table where attrib in {'value1', 'value2', ..}) to support set-oriented SAW. This approach will be limited to relatively few values as there is typically a restriction on the length of a query. For this reason we have chosen to use singleton SAW for relational data sources. Unless an index is defined for the join-attribute, the join may not be supported efficiently. We have, for this reason, defined an index on the join attribute. If a relational source is specified as non-SAW, the wrapper will retrieve the whole view and invoke the same join method as is used by the integrator.

Source capabilities VAW, CHAC, DAW, and CHAW for the relational source have been implemented as described in [Eng01].

The XML data source has been implemented to give full support for SAW. This includes an index on the join-attribute, and set-oriented compensating queries. We have not incorporated a fully-fledged query language for the XML-source. If the source is specified as VAW, a client can request that a subset of pages should be retrieved. If it is not VAW, the client has to retrieve all pages and inspect them to derive relevant pages. As for the relational source, CHAW and CHAC are realised through the updater. This means that an XML-source does not have any true support for CHAW and CHAC. The implemented solution is sufficient for experimental purposes. DAW is provided natively by the XML-source. It records all updates, and a client can request their retrieval. If a source is not DAW, the wrapper handles change detection by sorting the set of elements in the view. For the XML source, sorting is done using a Java TreeMap. The relational source uses "order by" to obtain a sorted view. To changes are derived using a merge algorithm [Elm94].

4.1.5. Collected Metrics

During experiments three different metrics are collected: the staleness (Z) and response time (RT) for each query, and an integrated cost (IC), which is the sum of source processing, warehouse processing and communication.

Staleness

Staleness is computed post-facto, based on the time when updates are committed, the time when query results are returned, and the last updates which a result utilised. To collect this information, the

director receives notifications from both the querier and the updaters. From an updater, this involves a remote procedure call. This avoids any problems with clock synchronisation, but delays involved are checked in order to ensure that they are within tolerance levels. The last update reflected in a query result is reported for each source to the querier when the result of a query is returned.

To compute staleness, the director maintains three arrays storing the required information:

- *finalResultTime* contains the time (in ms) when a query is satisfied;
- *query_packet* contains, for each query and source, the number of the latest update reflected in the answer to the query;
- *updateTime* contains the time a given update was committed.

Updates are assigned consecutive "packet" numbers starting from 0. Queries are also numbered sequentially from 0. The following expression will give the staleness for a given query q with respect to a source:

$$\text{staleness}[q] = \begin{cases} 0 & \textit{if query\_packet[q] is the last update or} \\ & \textit{finalResultTime[q]<updateTime[query\_packet[q]+1]} \\ \text{finalResultTime}[q] - \text{updateTime}[\text{query\_packet}[q]+1] & \textit{otherwise} \end{cases}$$

In other words, if no change has occurred after the one which is reflected in the result, staleness is 0. This is also the case if changes occur only after a result has been returned. Figure 13 shows a simple example of this with four queries.

finalResultTime

q0	3 s
q1	6 s
q2	10 s
q3	15 s

query_packet

q0	#0
q1	#0
q2	#1
q3	#2

updateTime

#0	0 s
#1	4 s
#2	6 s

staleness

q0	3<4 ⇒ 0
q1	6-4=2
q2	10-6=4
q3	no update #3 ⇒ 0

Figure 13. Computing staleness for queries - an example.

For join views, staleness is computed for each source individually. View staleness is taken as the greater of the two.

Response Time

To compute response time, the querier sends a notification to the director immediately before it issues a query and as soon as the response is received from the warehouse. The Director will record

the time of querying and the time of the response (finalResultTime in Figure 13). When experiments are finished, the response time for each query, i, is computed by subtracting queryTime[i] from finalResulTime[i].

Integrated Cost

The integrated cost is measured in the wrappers and in the integrator by recording the time spent on maintenance. This will include communication with the data source, internal processing, communication with the integrator and the time to update the view. Communication between wrapper and integrator is synchronous, which means the RMI-call will not return until the integrator has finished its part of maintenance. Obviously this measure may include delays caused by blocking due to resource contention, which means it is not measuring actual hardware utilisation. It is still believed the metric is useful as the testbed is only used for relative policy comparison, and this overhead is common to most configurations and any deviations are likely to be small.

Additional Recorded Information

In addition to the measures described above, additional useful information is collected. This is for such purposes as detecting errors and estimating the reliability of results. The following measures are reported after each experiment:

- The progress of the total IC during experiments. This indicates whether the duration of an experiment is sufficient to obtain representative results. For example, the overhead of the first half of the experiment should be close to the overhead of the second half.
- Total execution time, the number of updates, the size of updates and number of maintenance activities. This information is collected to ensure the correct behaviour of the system.
- Time spent performing updates, RMI overhead (including the time spent communicating with the director), errors and overflows (in wrappers, integrator, updater and querier).

Taken as a whole, these measures indicate whether an experiment is representative, and what confidence intervals can be placed on the output measures. All information collected is saved in an experiment log (an XML-file). In addition to the information above, the log also details the configuration used for the current experiment. Depending on user settings, the log can also contain

debug messages produced from each component in the environment. This is useful for analysing the behaviour of the system, and for tracing potential errors.

4.1.6. Execution of Experiments

An experiment can be configured and run using a graphical tool. Figure 14 shows a screen-dump.

A user can define data sources, hosts and the various nodes participating in an experiment. Each defined host needs to have a source server running. Hosts acting as a database server need to have the corresponding database installed and running. This is the case for *Server1* in the figure, which uses an Interbase database (stored in the file /export/db/database.gdb). All other components (including XML sources) will be automatically created and configured according to user specifications. It is possible to locate any node on any host. In the figure, Server2 is an XML data source and the node "xmlServer" is the actual data source. This will be installed automatically when an experiment is executed. The wide boxes represent abstract data sources which are handled by some data server (database or XML server currently).

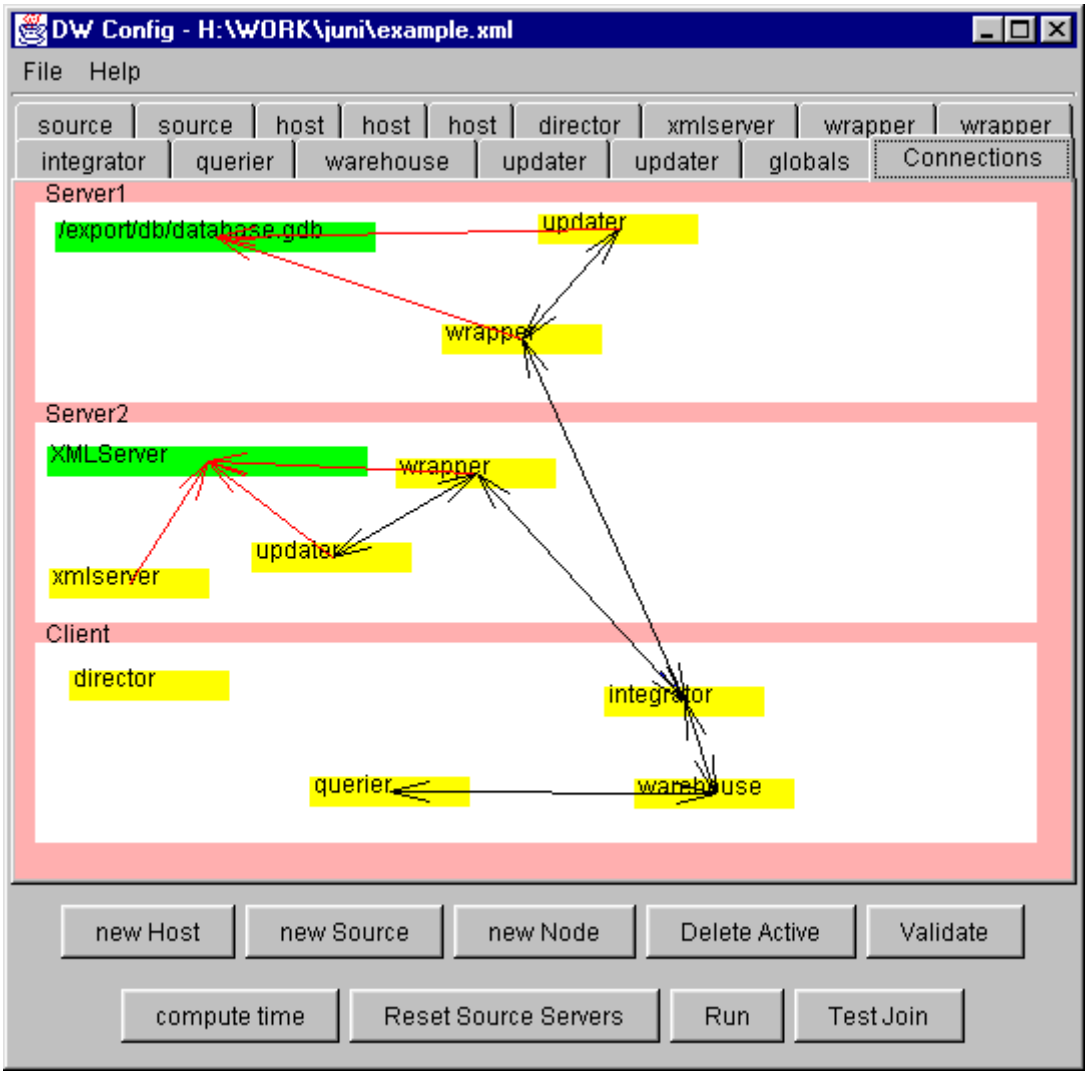

Figure 14. A tool for configuring and running experiments in TMID

The figure shows a panel which illustrates the localization of components and their connections. There is a choice to run each node (e.g. wrapper, querier, integrator) either as an individual process or as a thread within the source server. A user can specify which nodes need to communicate with each other, and these will be given remote references to each other at initialisation.

Each node has a number of parameters that can be configured. This is done using individual panels, not shown in Figure 14. It is possible to specify a range of values for most of these (for example, different query frequencies). When the "run" button is pressed, each possible combination will give rise to a separate experiment. This means that, through one operation, a user can run a large number of experiments. Between experiments all processes are killed and the source servers restarted. This is important to avoid dependencies between experiments (such as memory allocations).

The length of an experiment is specified as the number of queries to be posed. The tool can predict the total length of all experiments by computing the parameter combinations, and for each the product of the number of queries and the average query delay. Query delay is computed according to a random distribution, which means the computed length is approximate.

Once a configuration file has been created, it is possible to run experiments in batch mode (without a GUI). These can be started from any machine (which does not need to run a source server) and the results of experiments will be sent back to it and stored in XML-files. It is possible to interrupt running experiments by resetting the source servers (either through the GUI or in batch mode).

4.1.7. Analysis

An experiment consists of several benchmark tests, and analysing several files each with a large number of measures is an error-prone task. We have therefore developed a program which takes any number of logs and summarises information into a single file. This can then be processed, for example using a spreadsheet program. Individual queries are excluded; only summarised measures are shown. A setting is included if it is varied in the experiments. In other words, properties which are not varied are not included in the summary file. These settings can, however, be found in *any* of the log-files on which the summary is based. Figure 15 shows an example of such a summary, translated into a spreadsheet.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
logFilename	Policy	Left source is DAW	Right source is DAW	ComputedExecTime	ActualExecTime	IC left wrapper	IC Integrator	IC right wrapper	Max staleness Left source	Max staleness Right source	Max Staleness	Average Response time	IC per query	Max rmi overhead	Errors	Overflows
log001.xml	II-noaux-II-PI	Y	Y	1500	1536	200	3566	681	0	26274	26274	31	4448	87	0	0
log002.xml	PI-aux-PI-OI	Y	Y	1500	1536	181	169	460	36669	22788	36669	45	811	123	0	0
log003.xml	OI-aux-OI-OI	Y	Y	1500	1537	256	187	1452	0	563	563	355	1895	103	0	0
log004.xml	PR-aux-OR-OR	Y	Y	1500	1536	10387	291	12460	33429	27431	33429	58	23139	142	0	0
log005.xml	OI-aux-II-II	Y	Y	1500	1536	191	187	1427	225	655	655	177	1805	92	0	0

Figure 15. An example of summary results from several experiments

As can be seen from the figure, it is possible to observe staleness with respect to each source individually. The IC is shown for each wrapper and the integrator as well as the total sum. It is important that the summary indicates whether errors or overflows have occurred.

4.2. Experiments

Two different configurations have been used for experiments: one with computers using the Solaris operating system and the other with computers running Linux.

The first configuration comprises three Sun Solaris machines with different hardware and operating system characteristics. The machines are connected via a 10Mbit hub. All machines are running Java SDK 1.3.0 from Sun.

The left source is an XML server running under Solaris 8 on a 502 MHz Sun UltraSPARC-IIe with 128 Mbytes RAM, 0.2 Mbyte cache and an IDE disk. The source contains 2000 XML-pages each with a size of 2.5Kb. It is updated on average every 30 seconds and each update involves 50 pages (half of these are updates the others are inserts and deletes). In the base configuration the source is neither SAW nor CHAW. The view contains 60% of these pages.

The right source is an Interbase 6.0 relational database server running under Solaris 8 on a 360 MHz Sun Ultra Sparc IIi with 384 Mbytes RAM, 0.2 Mbyte cache and an IDE disk. The database contains on average 10000 tuples and each of these is 1Kb in size. It is updated on average every 30 seconds and each update involves 100 tuples (half of these are updates, the others are inserts and deletes). Database access is handled with JDBC using the native Interbase Interclient 1.6 driver. In the base configuration, the source is set to be SAW but not CHAW. The view contains 60% of the tuples. The client DW runs under Solaris 7 on a 167 MHz Sun Ultra Sparc I with 256 Mbytes RAM and 0.5 Mbyte cache and a SCSI disk.

The second configuration comprises three Linux PCs with an AMD Duron 800 MHz processor and an IDE disk. All computers run Mandrake Linux 8.1 and use Java SDK 1.4 from Sun. They are connected via a 10 Mbit hub. Both data sources are XML web servers running on machines with 515 Mbytes RAM. Each data source contains 4000 pages, each with a size of 2Kb. It is updated on average every 30 seconds, and each update involves 100 pages (half of these are updates the others

are inserts and deletes). In the base configuration, both sources are SAW but not CHAW. The view contains 60% of the pages in the source. The client DW has 320 Mbytes RAM; this is the only difference from the source machines.

In both configurations all sources are VAW and the wrappers are located in the DW host. Each experiment runs for 50 queries, and a query is issued on average every 30 seconds. This means the duration of an experiment is approximately 25 minutes plus the initialisation and finalisation time. The first 10% of queries are excluded from each result, as these are considered to be issued during the warm up period. Each experiment is repeated three times, using different random seed initialisations, and the median value of the three experiments is used.

4.2.1. Analysing SAW and the Impact of Joining Strategy

Theoretical analysis showed that incremental polices not using auxiliary views are rarely useful. Moreover, the choice between incremental and recompute join strategies was shown to depend on join strategy. To investigate these observations the following set of experiments has been conducted:

- Compare I_I-aux-I_I-I_I and I_I-noAux-I_I-I_I with all combinations of SAW, DAW and join strategy.
- Compare how the absence and presence of SAW affects I_I-noAux-I_I-I_I when the relational source does not have an index on the join attribute.
- Compare P_R-aux-O_R-O_R and P_I-aux-O_I-O_I for all combinations of join strategy and DAW.

The same sets of experiments has been conducted using both configurations. Here we present the results from the Solaris environment. The results from the Linux environment appear in Appendix C. The same trends reported below can be observed in this environment. This is an important result. As the goal with this work is to be useful in heterogeneous environments and variations in data model, source system (etc.) should not affect major trends. Where results differ in any respect in the Linux environment this is highlighted below.

Table 14 shows the costs when I_I-aux-I_I-I_I and I_I-noAux-I_I-I_I are compared for all combinations of join technique, DAW and SAW. Total cost (IC) is presented, as well as the components from the wrappers (to detect and propagate changes) and integrator (to perform the join, including join queries to the wrapper if no auxiliary views are maintained).

Table 14.. Analysing the impact of SAW and auxiliary views. The cost shown is the per query system cost in seconds.

					Hash-based join				Nested loop join			
XML source DAW	XML source SAW	RDB source DAW	RDB source SAW	Policy	XML wrapper cost	Integrator cost	RDB wrapper cost	Total cost (IC)	XML wrapper cost	Integrator cost	RDB wrapper cost	Total cost (IC)
Yes	Yes	Yes	Yes	II-aux-II-II	0.26	0.18	1.47	1.92	0.31	3.411	1.581	5.36
Yes	Yes	Yes	Yes	II-noAux-II-II	0.19	0.69	1.43	2.33	0.192	0.68	1.453	2.33
Yes	Yes	Yes	No	II-aux-II-II	0.26	0.19	1.45	1.9	0.306	3.423	1.531	5.3
Yes	Yes	Yes	No	II-noAux-II-II	0.21	8.87	1.47	10.6	0.195	11.17	1.501	12.8
Yes	Yes	No	Yes	II-aux-II-II	0.29	0.22	13.1	13.6	0.36	4.501	14.09	19.3
Yes	Yes	No	Yes	II-noAux-II-II	0.27	0.94	13.5	15	0.259	1.043	13.32	14.7
Yes	Yes	No	No	II-aux-II-II	0.32	0.2	13.2	13.7	0.32	4.425	14.48	19.4
Yes	Yes	No	No	II-noAux-II-II	0.25	10.1	15.3	25.7	0.255	13.06	15.93	29.4
Yes	No	Yes	Yes	II-aux-II-II	0.26	0.19	1.46	1.9	0.292	3.482	1.559	5.23
Yes	No	Yes	Yes	II-noAux-II-II	0.2	4.4	1.45	6.05	0.202	5.07	1.439	6.71
Yes	No	Yes	No	II-aux-II-II	0.27	0.18	1.46	1.93	0.297	3.414	1.526	5.38
Yes	No	Yes	No	II-noAux-II-II	0.19	12.5	1.51	14.2	0.198	15.79	1.516	17.4
Yes	No	No	Yes	II-aux-II-II	0.34	0.21	13.3	13.8	0.371	4.692	14.23	19.4
Yes	No	No	Yes	II-noAux-II-II	0.25	4.69	13.3	18.8	0.291	5.518	12.99	19.3
Yes	No	No	No	II-aux-II-II	0.3	0.2	13.2	13.7	0.377	4.446	14.12	19.3
Yes	No	No	No	II-noAux-II-II	0.25	13.9	15.1	29.3	0.29	18.25	15.73	34.3
No	Yes	Yes	Yes	II-aux-II-II	5.83	0.18	1.47	7.49	6.068	3.447	1.504	11
No	Yes	Yes	Yes	II-noAux-II-II	5.75	0.89	1.43	8.06	5.622	0.829	1.437	7.93
No	Yes	Yes	No	II-aux-II-II	5.8	0.18	1.48	7.46	6.005	3.597	1.556	11.1
No	Yes	Yes	No	II-noAux-II-II	5.75	9.22	1.57	16.2	5.702	11.56	1.531	18.6
No	Yes	No	Yes	II-aux-II-II	7.29	0.24	14.6	21.9	7.588	4.182	15.19	26.6
No	Yes	No	Yes	II-noAux-II-II	7.34	1.29	14.6	22.8	7.209	1.134	14.31	22.9
No	Yes	No	No	II-aux-II-II	7.07	0.25	14.3	21.7	7.484	4.26	15.24	26.7
No	Yes	No	No	II-noAux-II-II	7.32	9.82	15.4	32.8	7.067	12.74	15.84	35.5
No	No	Yes	Yes	II-aux-II-II	5.88	0.19	1.45	7.51	6.079	3.615	1.59	11.1
No	No	Yes	Yes	II-noAux-II-II	5.74	4.58	1.43	11.7	5.856	5.178	1.443	12.3
No	No	Yes	No	II-aux-II-II	5.87	0.18	1.44	7.5	5.999	3.502	1.636	11.1
No	No	Yes	No	II-noAux-II-II	5.89	12.6	1.51	19.9	5.736	15.83	1.57	22.7
No	No	No	Yes	II-aux-II-II	7.26	0.24	14.4	21.8	7.556	4.251	15.19	26.6
No	No	No	Yes	II-noAux-II-II	7.42	5.09	14.7	26.9	7.487	6.083	14.4	27.9
No	No	No	No	II-aux-II-II	7.3	0.2	14.8	22	7.262	4.361	15.25	27.2
No	No	No	No	II-noAux-II-II	8.21	14.1	16.4	38.2	8.434	19.37	16.98	43.9

First of all we notice that I_I-noAux-I_I-I_I has a higher cost than I_I-aux-I_I-I_I in most of the 32 situations explored. The comparative cost of the I_I-noAux-I_I-I_I policy is slightly lower cost when nested loops join is used and both sources are SAW, but can be significantly higher when some source is not SAW, or when hash join is used. This confirms the observation made in the theoretical analysis that auxiliary views should always be maintained unless the storage space in the warehouse is a very limited resource. In the Linux configuration it was found that using auxiliary views was always advantageous. We note from Table 14 that the difference in cost between non-SAW and SAW is biggest in the

relational source. A possible explanation is that this source contains more elements (10000) and that this makes it advantageous to send a few join queries when the source is SAW. This suggests that the trade-off (in using auxiliary views or not) involves the relative difference in size between source and changes. This is the case for incremental maintenance in general, and is captured in the cost model.

A second experiment explored the impact of having an index on the join attribute. The index on join attribute was removed and the experiments where both sources are DAW were repeated. The result of this is shown in Table 15.

Table 15. The impact of auxiliary views, join technique and SAW when the relational source does not have an index on the join attribute. The cost shown is the per query system cost in seconds.

Join Technique	**SAW**	**I_I-aux-I_I-I_I**	**I_I-noAux-I_I-I_I**
Hash	Yes	1.92	29.0
Hash	No	1.90	10.5
Nested loop	Yes	5.31	28.5
Nested loop	No	5.35	12.8

First of all we note that when auxiliary views are maintained, the result is unaffected by index and SAW capability. Without auxiliary views there is an interesting outcome. When the source is SAW the cost for hash join is almost three times the cost when the source is not SAW. As the relational source is singleton SAW, we must query the relational database to get the matching tuples for each inserted object in the other source. Without an index, each such query implies a scan through the relation. There is no way the database server can make use of the hash-based join. If the source is set-oriented SAW, the whole set of objects could be presented to the server, and it could in theory select a more appropriate joining technique.

Our cost-formulation does not capture this fine-grained detail concerning join. The model will therefore be misleading for a SAW source which does not utilise the potential optimisation techniques (that is a singleton SAW source without an index on the join attribute, or a set-oriented SAW source without query optimisation). To reformulate it, a source should not be classified as SAW if it does not handle joins efficiently.

To conclude, the SAW capability is meaningful but careful analysis is required to realise maintenance in the most efficient way. The performance evaluation confirms the result from PAM that using auxiliary views is advantageous in most situations. If both sources supports SAW then the cost can be slightly lower by not using auxiliary views. Reducing storage overhead in the DW is another reason to avoid auxiliary views. In such case the wrapper should handle the joining unless SAW is supported efficiently by the source.

The third set of experiments is designed to identify conditions under which recompute policies are advantageous. Analysis showed that this is only when no source is DAW and the join can be computed at low cost. In experiments P_I-aux-O_I-O_I was compared with P_R-aux-O_R-O_R for all combinations of DAW capabilities and join technique. In both cases the period was set to 200s. This value was selected to avoid overflows when nested loop join was used (nested loop join takes almost 3 minutes for the configuration used). Table 16 shows the results of these experiments.

Table 16. Comparison of recompute and incremental policies. The cost shown is the per query system cost in seconds.

Join Technique	**XML source DAW**	**RDB source DAW**	**P_I-aux-O_I-O_I**	**P_R-aux-O_R-O_R**
Hash	Yes	Yes	0.61	4.20
Hash	Yes	No	2.81	4.11
Hash	No	Yes	1.43	3.70
Hash	No	No	4.53	3.59
Nested loop	Yes	Yes	3.90	25.63
Nested loop	Yes	No	5.78	25.67
Nested loop	No	Yes	4.50	33.65
Nested loop	No	No	8.26	26.07

The experiments confirm the analytical results. The recompute policy has a lower cost than the incremental in only one of the eight cases. When nested loop join is used, the recompute policy has a cost that is always more than three times the incremental cost. These results are for a specific configuration, and the relative difference will obviously depend on the size of the views and the size of the changes. With bigger change size, incremental policies become less attractive. Colby et al. [Col97] suggest that once changes involve 10-20% of the tuples then recompute policies should be a viable alternative. The figures in Table 16 suggest that this percentage is highly dependant on join

technique for join views (as well as DAW capability). The cost of the incremental policy with hash join is relatively close to that for its recompute counterpart. This may imply that if change size is increased then incremental policies will be less favourable.

When nested loop join is used, on the other hand, the incremental policy is far better than its recompute counterpart. This implies that change size can increase more for this type than for hash-join. This observation has been confirmed for a different join scenario [Ast02].

4.2.2. Comparing all Policies

The 84 policies analysed using PAM can all be defined and used in TMID. A number of parameter configurations have been selected for which these policies have been compared, using different metrics. Analysis using PAM, and results from previous experiments, both show that recompute join requires low cost joining to be effective. For this reason we only consider hash-based joins in this section. It should be remembered that recompute join is more costly when nested loop join is used.

The comparison has been repeated for combinations of DAW capability where both, one or no source is DAW.

Table 17 shows four different combinations of costs for all policies when no source is DAW. The policies are ordered according to total ('All') cost.

Table 17. The cost of all policies when no source is DAW. The cost shown is the per query system cost in seconds.

Policy	IC	IC+Z	IC+RT	All	Policy	IC	IC+Z	IC+RT	All
O_R-aux-I_I-I_R	19.2	30.8	19.6	31.2	O_R-aux-I_R-P_I	20	58.3	20.4	58.8
O_R-aux-I_R-I_R	18.8	30.9	19.1	31.2	P_R-aux-P_R-O_R	16.3	58.7	16.4	58.8
I_R-aux-I_R-I_R	19.3	31.2	19.3	31.2	P_R-aux-I_I-O_I	22.2	59.2	22.2	59.2
I_R-aux-I_I-I_R	20.2	33.4	20.3	33.5	I_R-aux-I_I-P_I	22.8	59.5	22.8	59.5
I_I-aux-I_I-I_I	21.8	35.6	21.9	35.7	P_R-aux-I_I-P_R	18.1	59.5	18.1	59.6
O_R-aux-I_I-I_I	21.9	36.9	22.3	37.2	I_R-aux-P_I-P_I	19.5	59.7	19.6	59.7
O_I-aux-I_I-I_I	22.4	37	22.6	37.3	I_I-aux-P_I-P_I	23.4	60.8	23.5	60.9
I_R-aux-I_I-I_I	23	38.7	23.1	38.8	P_R-aux-I_I-I_I	21.9	61.1	22	61.1
I_I-noAux-I_I-I_I	26.5	40.5	26.6	40.6	O_I-noAux-O_I-I_I	27.4	49.7	39	61.3
O_R-aux-I_R-O_R	18.5	30.5	28.7	40.7	P_R-aux-P_I-O_R	17.3	61.3	17.4	61.4
O_R-aux-I_I-O_R	19.4	31.7	29.6	41.9	P_I-aux-O_I-O_I	26.6	61.4	26.7	61.4
O_I-aux-O_I-I_I	22.7	38.3	29.6	45.3	I_R-aux-I_R-P_I	20.2	61.6	20.2	61.6
O_R-noAux-O_R-O_R	22.5	35.1	33.9	46.5	P_R-aux-O_R-O_R	22.8	62.1	22.9	62.1
O_R-aux-O_R-O_R	22.7	35.4	34.1	46.9	O_I-aux-P_I-P_I	22.3	62.3	22.6	62.6
O_R-aux-I_R-O_I	20.5	35.5	32.8	47.8	O_R-aux-P_R-O_I	20.2	50.6	32.5	62.8
O_R-aux-I_I-P_R	18.3	48.5	18.7	48.9	P_I-aux-P_I-O_I	19.3	63.7	19.3	63.7
I_R-aux-I_I-P_R	19	48.9	19.1	48.9	I_I-noAux-P_I-P_I	26.8	63.9	26.9	64
I_R-aux-I_R-P_R	18.3	48.9	18.4	48.9	P_I-aux-I_I-I_I	22	64.2	22	64.3
O_R-aux-I_I-O_I	21.8	36.5	34.3	49	I_I-noAux-I_I-P_I	25.8	64.4	25.9	64.4
O_R-aux-P_I-P_R	17	49.1	17.3	49.4	P_R-aux-O_I-O_I	26.6	64.4	26.6	64.5
O_R-aux-O_I-O_R	23.9	37.9	35.9	49.9	O_R-aux-P_I-O_R	19	54.8	29.3	65.1
O_R-aux-P_I-P_I	19.7	49.6	20.1	49.9	P_R-aux-P_I-O_I	19.2	65.2	19.3	65.2
I_R-aux-P_I-P_R	18	49.9	18	50	P_R-aux-I_R-P_R	17.7	65.3	17.8	65.4
P_R-aux-I_R-O_I	19.9	50.7	20	50.7	P_I-noAux-O_I-I_I	27.7	66.8	27.8	66.9
P_R-aux-I_I-O_R	18.3	50.7	18.4	50.8	P_I-noAux-I_I-I_I	28.8	67.3	28.8	67.4
P_R-aux-I_R-O_R	18	51.8	18.1	51.9	P_R-aux-P_R-O_I	23.6	68.2	23.6	68.2
O_R-aux-I_R-P_R	18.2	51.6	18.5	51.9	P_I-noAux-O_I-O_I	30.2	68.4	30.3	68.4
P_R-aux-I_R-I_R	18.8	52.5	18.9	52.6	P_R-aux-P_I-P_R	17.3	70.1	17.3	70.2
P_I-aux-I_I-O_I	20.4	52.8	20.5	52.9	O_I-aux-O_I-P_I	21.9	63.9	29.3	71.2
O_R-aux-P_R-O_R	18	43.7	28.1	53.8	O_I-noAux-O_I-O_I	31.2	53.1	49.5	71.4
I_I-aux-I_I-P_I	21.5	53.8	21.5	53.9	P_I-noAux-P_I-O_I	27.3	72.4	27.4	72.5
O_I-aux-O_I-O_I	25.8	41	39	54.1	P_I-noAux-P_I-I_I	28.1	73.2	28.1	73.3
O_R-aux-O_I-O_I	26.2	41	39.7	54.5	O_I-noAux-P_I-I_I	29.7	68.2	35.4	73.9
O_I-noAux-I_I-I_I	28	50.1	33.4	55.4	O_I-noAux-P_I-P_I	27.9	68.4	33.6	74.1
O_R-aux-I_I-P_I	21.2	55.7	21.5	56	P_R-aux-I_R-P_I	20.9	74.4	21	74.4
P_R-aux-I_I-I_R	19.3	56.1	19.4	56.1	P_R-aux-P_I-P_I	25.9	75.3	26	75.4
P_R-noAux-O_R-O_R	23.1	56.2	23.2	56.2	P_I-noAux-P_I-P_I	26.9	75.7	26.9	75.7
P_R-aux-O_I-O_R	23.8	57.4	23.9	57.4	P_R-aux-P_R-P_R	16.5	77.1	16.6	77.2
O_R-aux-P_I-O_I	20.6	45.7	32.6	57.6	P_R-aux-I_I-P_I	20.9	78.5	20.9	78.6
I_R-aux-P_R-P_R	21.7	57.7	21.8	57.8	P_I-aux-P_I-P_I	23.6	79.7	23.6	79.8
O_R-aux-P_R-P_R	20.8	57.9	21.2	58.3	P_I-aux-I_I-P_I	21	80	21	80
O_I-aux-I_I-P_I	21.6	58.5	21.8	58.7	O_I-noAux-O_I-P_I	27.6	69	39.6	81

First of all we see that the policies are relatively evenly distributed from the minimum value (31.2s) to the maximum value (81.0s). This means there are no obvious properties that determine cost. It is still possible to identify properties which are rare at one end of the table and common at the other. We note that recomputed SVPs are common in the beginning of the table (low cost) and rare at the end. The explanation is that when sources are not DAW incremental policies are more expensive than when they are DAW. In addition, the cost of recomputing the join is relatively low when a hash-based

join algorithm is used. The average delay (not shown) to recompute the view when the supporting views have been made available (with or without auxiliary views) was 0.4 seconds for the experiments presented. This should be compared with the delay (on average 0.2s) to incrementally maintain the view when there are auxiliary views. The difference (0.2s) is very small compared with the total cost. This means that, with hash-join, there is little difference between recomputing the join and incrementally maintaining it.

Another observation from Table 17 is that keeping auxiliary views when join policy is incremental seems to be advantageous. There are only four policies without auxiliary views among the top 42 policies (the left column). Two of these are incremental, and for both of these the corresponding policy that uses auxiliary views (I_I-aux-I_I-I_I and O_I-aux-O_I-I_I respectively) has a lower cost. This confirms analytical results. Among the low ranked policies (right column in Table 17) there are many policies which combine a periodic SVP with periodic integrator policy, also as predicted.

It should be noted that the comparison is based on all costs. If the comparison were based on other metrics different policies would emerge as optimal. Table 18 shows the top five policies for the four metrics presented in Table 17.

Table 18. The five lowest-cost policies for different metrics when no source is DAW

IC	**I**C+Z	**I**C+RT	**A**ll
P_R-aux-P_R-P_R	O_R-aux-I_R-O_R	P_R-aux-P_R-O_R	O_R-aux-I_I-I_R
P_R-aux-I_R-O_R	O_R-aux-I_I-I_R	P_R-aux-P_R-P_R	O_R-aux-I_R-I_R
P_R-aux-I_R-P_R	O_R-aux-I_R-I_R	O_R-aux-P_I-P_R	I_R-aux-I_R-I_R
P_R-aux-P_I-P_R	I_R-aux-I_R-I_R	P_R-aux-P_I-P_R	I_R-aux-I_I-I_R
P_R-aux-P_I-O_R	O_R-aux-I_I-O_R	P_R-aux-P_I-O_R	I_I-aux-I_I-I_I

As predicted, periodic policies are advantageous when staleness is not included in the cost. Immediate policies are advantageous whenever staleness is included. On-demand integrator policies are negatively affected when response-time is combined with IC. Interestingly, when all costs are considered two on-demand policies have the lowest cost. Both have immediate SVPs, which add nothing to the response time and still give low staleness. This means that response-time for these policies will only be the delay to recompute the view, which is low with hash-based join. And as predicted by the cost model, on-demand integrator polices seem to be beneficial compared with

immediate in that they can handle changes from left and right sources at the same time (avoiding several merges into the view).

We note that irrespective of metric, recompute SVPs dominate the top policies. When both sources are DAW the result is quite different, as can be seen from Table 19.

Table 19. The cost of all policies when both sources are DAW. The costs shown are the per query system costs in seconds.

Policy	IC	IC+Z	IC+RT	All	Policy	IC	IC+Z	IC+RT	All
O_I-aux-O_I-O_I	0.91	1.07	1.55	1.72	I_R-aux-I_R-P_I	6.51	34.3	6.54	34.3
O_R-aux-O_I-O_I	1.01	1.48	1.74	2.21	P_I-aux-P_I-P_I	0.82	35.8	0.86	35.9
O_R-aux-I_I-O_I	1.06	1.57	1.75	2.27	P_R-aux-I_R-O_I	6.01	37.5	6.04	37.6
I_I-aux-I_I-I_I	1.9	2.73	1.93	2.77	P_I-aux-P_I-O_I	0.89	37.6	0.94	37.6
O_I-aux-I_I-I_I	1.79	2.65	1.96	2.83	O_R-aux-P_R-O_I	6.16	37.3	6.93	38
O_I-aux-O_I-I_I	1.9	2.7	2.25	3.06	P_I-aux-I_I-P_I	0.87	38.1	0.9	38.2
I_R-aux-I_I-I_I	2.22	3.11	2.25	3.14	I_R-aux-P_I-P_R	11.4	38.9	11.5	39
O_R-aux-I_I-I_I	1.91	3.05	2.17	3.3	O_R-aux-P_I-P_R	11	38.8	11.3	39.1
I_I-noAux-I_I-I_I	5.86	6.75	5.89	6.78	O_I-noAux-P_I-P_I	4.92	35.4	8.68	39.2
O_R-aux-I_R-O_I	6.02	11.2	6.75	11.9	P_R-aux-I_I-P_I	0.92	39.4	0.95	39.4
O_I-noAux-I_I-I_I	5.88	11.4	9.76	15.3	O_I-noAux-O_I-P_I	4.77	35.8	8.55	39.6
O_I-noAux-O_I-I_I	5.87	11.5	9.92	15.5	O_R-aux-I_R-O_R	18.3	30.6	28.3	40.7
O_I-noAux-O_I-O_I	5.78	11.7	10.8	16.7	O_I-noAux-P_I-I_I	5.92	37.4	9.81	41.2
O_R-aux-I_I-I_R	11.8	21.7	12.1	22	P_R-aux-I_I-O_R	11	44.3	11.1	44.4
P_I-aux-I_I-I_I	1.84	22.1	1.87	22.2	O_R-aux-I_I-P_R	11	44.6	11.3	44.9
O_I-aux-O_I-P_I	0.92	22.3	1.27	22.7	P_I-noAux-P_I-O_I	5.8	45.3	5.83	45.4
O_I-aux-I_I-P_I	0.88	23.3	1.09	23.5	P_R-aux-O_I-O_R	11	46.1	11	46.2
I_R-aux-I_I-I_R	12.4	24.3	12.4	24.3	O_R-noAux-O_R-O_R	22.8	35.1	34.1	46.4
P_R-aux-I_I-I_I	1.94	25.9	1.96	26	O_R-aux-P_I-O_R	11.1	36.9	20.7	46.5
I_R-aux-I_I-P_I	1.35	26	1.39	26.1	P_R-aux-I_R-I_R	18.4	47.2	18.4	47.2
P_R-aux-I_I-O_I	1	26.3	1.04	26.3	I_R-aux-P_R-P_R	17.3	47.6	17.3	47.7
I_I-aux-P_I-P_I	1.03	26.5	1.06	26.6	I_R-aux-I_I-P_R	11.3	47.6	11.4	47.7
P_I-aux-I_I-O_I	0.91	27.2	0.94	27.2	P_I-noAux-P_I-I_I	5.72	47.8	5.75	47.8
I_I-aux-I_I-P_I	0.97	27.7	1	27.7	O_R-aux-O_R-O_R	23.2	36.9	34.6	48.4
P_I-aux-O_I-O_I	0.91	27.7	0.95	27.8	P_R-aux-P_R-O_I	6.21	49.1	6.24	49.1
O_R-aux-I_I-P_I	1.02	27.7	1.27	28	I_R-aux-I_R-P_R	19.1	49.7	19.2	49.7
O_I-aux-P_I-P_I	0.9	28	1.08	28.1	P_R-aux-I_I-I_R	11.8	50.8	11.9	50.9
O_R-aux-P_I-O_I	1.01	27.6	1.71	28.3	P_R-aux-P_I-O_R	11.3	50.9	11.3	51
P_R-aux-P_I-O_I	0.97	28.5	1	28.5	P_I-noAux-P_I-P_I	4.99	51.4	5.03	51.4
P_R-aux-O_I-O_I	1.01	28.6	1.04	28.7	O_R-aux-P_R-P_R	21.5	51.5	21.9	51.9
I_R-aux-P_I-P_I	1.38	29	1.42	29.1	O_R-aux-I_R-P_R	18.2	51.8	18.5	52.2
O_R-aux-P_I-P_I	0.96	29.6	1.23	29.9	P_R-aux-I_I-P_R	11.2	52.2	11.2	52.3
O_R-aux-I_R-I_R	18.4	30	18.7	30.4	P_R-aux-I_R-O_R	18.7	55.9	18.8	56
I_R-aux-I_R-I_R	19.1	30.9	19.2	30.9	P_R-aux-P_I-P_I	0.92	56	0.96	56.1
I_I-noAux-I_I-P_I	4.95	31.2	4.98	31.3	P_R-aux-I_R-P_I	6.09	56.2	6.13	56.3
O_R-aux-I_I-O_R	11.1	21.7	20.7	31.3	P_R-noAux-O_R-O_R	22.7	58.7	22.8	58.7
O_R-aux-O_I-O_R	11	21.9	20.5	31.5	P_R-aux-P_I-P_R	10.9	59.1	11	59.2
P_I-noAux-O_I-O_I	5.84	31.6	5.87	31.6	P_R-aux-P_R-O_R	16.6	60.1	16.7	60.2
O_R-aux-I_R-P_I	6	31.4	6.28	31.7	O_R-aux-P_R-O_R	18.3	51.5	28.4	61.6
I_I-noAux-P_I-P_I	5.1	32.3	5.13	32.3	P_R-aux-O_R-O_R	23.1	61.7	23.2	61.8
P_I-noAux-O_I-I_I	5.88	32.8	5.9	32.8	P_R-aux-P_R-P_R	21.6	66	21.7	66
P_I-noAux-I_I-I_I	6.07	33.2	6.1	33.3	P_R-aux-I_R-P_R	18.1	72.1	18.2	72.1

A handful of policies have a significantly lower cost than the rest. All of these have either O_I or I_I as SVPs. Among the ten most expensive policies, nine have a periodic integrator policy and seven

have at least one periodic SVP. Again it should be noted that policies are ordered according to total cost. Table 20 shows the top-5 policies under different evaluation criteria.

Table 20. The five lowest cost policies for different metrics when both sources are DAW

IC	**I**C+Z	**I**C+RT	**A**ll
P_I-aux-P_I-P_I	O_I-aux-O_I-O_I	P_I-aux-P_I-P_I	O_I-aux-O_I-O_I
P_I-aux-I_I-P_I	O_R-aux-O_I-O_I	P_I-aux-I_I-P_I	O_R-aux-O_I-O_I
O_I-aux-I_I-P_I	O_R-aux-I_I-O_I	P_I-aux-P_I-O_I	O_R-aux-I_I-O_I
P_I-aux-P_I-O_I	O_I-aux-I_I-I_I	P_I-aux-I_I-O_I	I_I-aux-I_I-I_I
O_I-aux-P_I-P_I	O_I-aux-O_I-I_I	P_R-aux-I_I-P_I	O_I-aux-I_I-I_I

It is interesting to note that all SVPs are incremental. This confirms the theoretical analysis that when sources are DAW incremental SVPs are preferable. Recomputed integrator policies are represented among the lowest cost policies. This indicates that when joining is low cost it may be advantageous to maintain auxiliary views incrementally and to recompute the join, even when both sources are DAW. This was not identified in the original analysis of the cost model, but it is possible to show this effect using PAM.

Finally we note that all policies in tables 18 and 20, independent of evaluation criteria, use auxiliary views. It is clearly advantageous to maintain auxiliary views in the integrator.

Results when only one source is DAW, shown in Appendix B, suggest that using an incremental SVP for the DAW source and a recompute SVP for the non-DAW source gives low costs. This is as predicted by the cost model. It is important to remember that this is true only when the join can be computed at low cost. With nested loop join, incremental policies are always preferable.

4.2.3. Analysing the Heuristics

As mentioned above, all policies are supported in TMID. In the previous section we presented the results of experiments in which the performance of each policy has been measured. This enables us to compare the outcome of the selection process in Figure 11 with “real” results. For various evaluation criteria there will be a policy with the lowest “real” value and a policy with the highest “real” value. We then compare these values with the value of the policy derived from the selection process. This will be as a measure of the quality of the policy decision. In this section we analyse the quality of the selection process for each set of experiments presented in the previous section. For each set, we analyse four different evaluation criteria (IC, IC+Z, IC+RT, All) with different combinations of

CHAC and DAW. We remind the reader that immediate policies require the source to be CHAC. If this is not the case, we have to exclude all policies with immediate SVP. This can also have an effect on the policy delivered by the policy selection process.

Table 21 shows the results of this comparison for the first set of experiments. Here no source is DAW but both are CHAC

Table 21. A comparison of the policy selected through heuristics against all other policies when no source is DAW but both are CHAC. The cost shown is the per query system cost in seconds.

Evaluation criteria	IC	IC+Z	IC+RT	All
Selected policy	P_R-aux-O_R-O_R	I_R-aux-I_R-I_R	P_R-aux-O_R-O_R	I_R-aux-I_R-I_R
Cost of selected	0.6s*	31.2s	0.6s*	31.2s
Lowest cost of all 84 policies	16.3s	30.5s	16.4s	31.2s
Highest cost of all 84 policies	31.2s	80.0s	49.5s	81s
Average cost of all 84 policies	22.1s	55.0s	24.8s	57.7s

* This value is for a periodicity which is 50 times lower than update frequency. The selection process suggests that the periodicity should be set as low as possible.

First of all, when comparison is based on IC and IC+RT the selected policy is P_R-aux-O_R-O_R with periodicity set as low as possible. In the original experiment (shown in Table 17), periodicity was set equal to query and update frequency. To give a fair comparison we have therefore repeated the experiment using this policy with periodicity set to 50 times lower than it was originally, in which case the IC was found to be 0.6s per query. This explains why the cost of this policy is lower than the lowest cost of Table 17.

We note that the cost of the selected policy, for all criteria, is very close to the lowest cost policy and is significantly lower than the average policy cost. In two cases the selected policy uses immediate SVPs. As mentioned earlier, this assumes that sources are CHAC. Table 22 shows the result when sources are non-CHAC and hence policies with immediate SVPs have been excluded.

Table 22. A comparison of the policy selected through heuristics against all other policies when no source is DAW or CHAC. The cost shown is the per query system cost in seconds.

Evaluation criteria	IC	IC+Z	IC+RT	All
Selected policy	P_R-aux-O_R-O_R	O_R-aux-O_R-O_R	P_R-aux-O_R-O_R	P_R-aux-O_R-O_R
Cost of selected	0.6s*	35.4s	0.6s*	62.1s
Lowest cost	16.3s	35.1s	16.4s	46.5s
Highest cost	31.2s	79.7s	49.5s	81s

* This value is for a periodicity which is 50 times lower than update frequency. The selection process suggests that the periodicity should be set as low as possible.

Firstly, it should be noted that the highest and lowest cost is now for all policies not having any immediate SVP. This explains why the lowest and highest costs differ from Table 21. The result of the comparison shows that all selected policies have a very low cost except when all criteria are considered, in which case P_R-aux-O_R-O_R is selected. In this situation the selection process also suggests that a periodicity should be selected to "balance staleness and system costs". In the experiments, the periodicity is identical to the change and update frequencies. This means that a full period of 30s will be added to staleness. For a single source it has been shown [Eng02b] that if sources are CHAW it is possible to shorten the length of the period so reduce the combined staleness and system costs. This has been tested in TMID, by setting the period to 10 seconds. The total cost ('All') for P_R-aux-O_R-O_R will then be 36.1s. This is lower than the lowest cost of all policies in Table 22, not using immediate SVPs. If the sources are not CHAW, however, it is not possible to reduce the cost. Actually the cost will be 78.6s when the period is 10 seconds. In such cases O_R-aux-O_R-O_R would have been a better selection. Even with response-time included, this policy has a very low cost (46.9s as can be seen in Table 17).

Up to now we have considered a situation in which no source is DAW. If both sources are DAW a different policy will be selected and the costs of policies will also be different. Table 23 shows the result when both sources are CHAC and DAW.

Table 23. A comparison of the policy selected through heuristics against all other policies when both sources are DAW and CHAC.

Evaluation criteria	IC	IC+Z	IC+RT	All
Selected policy	P_R-aux-O_R-O_R	I_I-aux-I_I-I_I	P_R-aux-O_R-O_R	I_I-aux-I_I-I_I
Cost of selected	0.6s*	2.73s	0.6s*	2.77s
Lowest cost of all 84 policies	0.82s	1.1s	0.86s	1.72s
Highest cost of all 84 policies	23.2s	72.1s	34.6s	72.1s
Average cost of all 84 policies	7.9s	33.7s	9.2s	35.0s

* This value is for a periodicity which is 50 times lower than update frequency. The selection process suggests that the periodicity should be set as low as possible.

Again we note that the selected policies have a cost close to the lowest, and that it is much lower than the average. Table 24 shows the results when immediate SVPs are excluded (sources are not CHAC).

Table 24. A comparison of the policy selected through heuristics against all other policies when both sources are DAW but not CHAC

Evaluation criteria	IC	IC+Z	IC+RT	All
Selected policy	P_R-aux-O_R-O_R	O_I-aux-O_I-O_I	P_R-aux-O_R-O_R	P_I-aux-O_I-O_I
Cost of selected	0.6s*	1.07s	0.6s*	27.8s
Lowest cost	0.82s	1.1s	0.86s	1.72s
Highest cost	23.2s	66.0s	34.6s	66.0s

* This value is for a periodicity which is 50 times lower than update frequency. The selection process suggests that the periodicity should be set as low as possible.

The result in this case is similar to when no source is DAW. The selection gives very good policies except when all costs are combined. The same problem of setting periodicity arises. For incremental policies, we have shown [Eng02b] that if sources are DAW the periodicity can be increased to reduce the combined system and staleness cost. If the period is set to 10 seconds (instead of 30 seconds) in this situation, the total costs for P_I-aux-O_I-O_I will be close to 10 seconds, a clear improvement.

The above gives a few examples of how the quality of the heuristics has been evaluated. Table 25 summarises this evaluation for a large number of situations. Experiments from both system configurations (Linux and Solaris) and with different combinations of CHAC and DAW are included.

The quality measure is intended to denote how close a selected policy is to the best policy, and is computed simply as 100*(Max-x)/(Max-Min), where x is the cost of the selected policy and Max and Min are the highest and lowest costs, respectively, over all policies. This means that if the selected policy has a cost equal to the best policy then quality is 100%. If it has a cost equal to the worst policy then quality is 0%. The table illustrates a large number of comparisons.

Table 25. The quality (as a percentage) of the policy selected through the selection process in comparison to the best and worst policies in different configurations.

System	Left DAW	Right DAW	CHAC	IC	IC+Z	IC+RT	All
Solaris			*	100	99	100	100
Solaris				100	99	100	55
Solaris	*	*	*	100	98	100	99
Solaris	*	*		100	100	100	59
Solaris	*		*	100	90	100	91
Solaris	*			100	94	100	57
Linux			*	100	99	100	100
Linux				100	100	100	53
Linux	*	*	*	100	100	100	100
Linux	*	*		100	100	100	53
Linux	*		*	100	91	100	100
Linux	*			100	99	100	60

Here we will give the details for one such measure, the third line in the table when all criteria are used. This can be computed using the information in Table 19 (which is also used for Table 23).

Among all policies, O_I-aux-O_I-O_I had the lowest cost: 1.72s per query. P_R-aux-I_R-P_R had the highest cost, 72.1s per query. When all criteria are considered and the sources are CHAC and DAW, the selection process suggests that I_I-aux-I_I-I_I should be used. This policy had a cost of 2.77s per query. The quality of this selection will therefore be 100*(72.1-2.77)/(72.1-1.72)=99%.

It is clear that the suggested selection process avoids poor policies in all situations. Actually, the heuristics are very good except when all criteria are considered and the sources are not CHAC. Immediate SVPs are not available in this case, and the selection process suggests that a periodic policy should be used. On-demand integrator policies are avoided as they give a response time penalty. For the situations considered in this section response time is, however, low compared with staleness for periodic policies. This means that an on-demand policy would have been preferable in these situations. Note that the selection process suggests selecting a periodicity to balance system costs and staleness. In the above experiments, periodicity was set to the average change frequency. As discussed above, a more suitable periodicity can easily be selected in many situations. For recomputed policies over CHAW sources and incremental policies over DAW sources, periodicity can be increased to reduce staleness without increasing system overhead [Eng02b]. In such cases the cost of a periodic policy becomes lower than the cost of the corresponding on-demand policy. Table 26 shows the total cost of periodic policies as the period is changed, when both sources are DAW and CHAW.

Table 26. The total cost in seconds for periodic policies when the period is varied

Period (s.)	P_I-aux-O_I-O_I	P_R-aux-O_R-O_R
5	4.3	29.6
10	9.0	36.1
20	18.3	48.8
40	38.9	63.8

It is clear that the total cost can be reduced significantly by reducing the period in this case. If the sources are not CHAW, the recompute policies will have a higher cost when the period is 5s than when it is 40 seconds. This is due to increased system overhead. If sources are not DAW, an incremental policy will be dependent on CHAW in the same way as a recompute policy.

The usefulness of the selection process becomes clear when it is compared with an ad-hoc policy decision. A common approach to DW maintenance is to periodically reload a view. This corresponds to the policy P_R-noAux-O_R-O_R. Table 27 shows the quality of that policy for the same situations as Table 25.

Table 27. The quality (as a percentage) of P_R-noAux-O_R-O_R in comparison to the best and worst policies in different configurations.

System	Left DAW	Right DAW	CHAC	IC	IC+Z	IC+RT	All
Solaris			*	54	48	80	50
Solaris				54	53	80	72
Solaris	*	*	*	2	19	35	19
Solaris	*	*		2	11	35	11
Solaris	*		*	3	27	51	27
Solaris	*			3	27	51	33
Linux			*	15	50	55	42
Linux				15	50	55	55
Linux	*	*	*	0	13	30	13
Linux	*	*		0	13	30	13
Linux	*		*	2	17	39	19
Linux	*			2	19	39	22

It is clear that this policy choice is far from optimal in most situations, and in many cases is very close to worst case performance.

It should be noted that hash-based join has been used for all experiments presented. It remains to evaluate the quality of the heuristics for situations where more expensive join techniques are used.

4.3. Summary and Conclusions

TMID has been used to conduct a number of experiments. By measuring performance we have been able to verify most of the results from the PAM analysis. For example:

- It is advantageous to use auxiliary views in the integrator. Sending queries to the wrapper gives poor performance unless both sources are SAW.
- When nested-loop join is used, incremental maintenance is always better than recompute.
- When hash-join is used, the integrator cost to recompute the join is comparable with the cost of performing incremental maintenance.
- The lowest-cost policy varies depending on view and source characteristics. Having different SVPs is often advantageous. When one source is DAW but not the other, the best policies are those that combine incremental and recompute SVPs, and recompute the join (assuming joins are low cost).
- Staleness is high if integrator and wrappers all use periodic policies.

However, some interesting differences from the analytical model were also identified:

- Even if the data source is singleton SAW, it can be advantageous to perform the join in the wrapper. If the wrapper is located in the data source, the cost of extracting the view may be low compared with sending singleton join queries to the source, unless these are handled efficiently. This was the case, for example, when the join attribute for a relation (in the database server) did not have an index. In such a case, each join query (select * from tab where joinAttribute=x) causes the database to do a table scan.
- For the configuration used in the experiments, the response-time was relatively small for on-demand policies with auxiliary views using immediate or periodic SVPs. This made on-demand policies comparable with other timings even when response-time was included in the criteria. It should be noted that this is not true for recompute policies when nested loop join is used. It may be different for other configurations.

By comparing the performance of policies derived through the selection process (Figure 11) with the best and worst policies, we have established that it always avoids “bad” policies and mostly chooses “good” policies.

The results have been established using two different hardware and software configurations, indicating that they are not limited to a specific environment.

To conclude, the experiments have shown that all theoretically interesting policies can be implemented and are meaningful. Also, predictions based on the cost model and using the PAM tool have been shown to be accurate. This implies that these predictions may be useful when a detailed analysis is required for a specific scenario. It is clear from the experiments that selection of an optimal policy is a complex task, and many dimensions can be varied. For the situations explored, we have shown that the suggested selection process (based on derived heuristics) yield policy decisions of good quality. More work is needed to test the quality of the heuristics for other situations.

5. Related Work

There have been several previous studies of view maintenance policies. Hanson [Han87] presents a performance comparison of view maintenance policies in a centralised database environment. Srivastava and Rotem [Sri88] identify the trade-off between quality of service and system overhead. Colby et al. [Col97] extend this and suggest support for multiple view maintenance policies for different groups of views. All these studies are for specific database environments, which means they do not have to address differences in source capabilities and distribution.

In the ADMS prototype [Rou86, Rou95] materialised views are used to optimise performance and system utilisation in a client-server database. Segev et al. [Seg89, Seg91] present pioneering work on materialised views for distributed databases, which resemble the ideas of data warehousing. None of these handles autonomous data sources with varying degrees of support for view maintenance. Moreover, they do not use or discuss the wide set of policies presented in this report. Zhou et al. [Zho95, Hul96, Hul96b] address DW maintenance as part of the Squirrel project, where the system supports virtual and materialised views as well as their hybrids. In this report we address view maintenance, which by definition implies materialised views.

Gupta et al. [Gup93, Gup96] present techniques for incremental view maintenance and define the concept of self-maintainable views. Quass et al. [Qua96] extend on this and suggest algorithms for generating auxiliary views that make warehouse views self-maintainable. These use referential integrity constraints and other properties (e.g. that some views are never updated) to reduce the size of auxiliary views. Our approach may be extended to include some of these ideas to reduce the storage overhead of auxiliary views in the integrator.

DW quality issues have been addressed in a number of studies within the DWQ project [Jar97, The99]. Issues related to view consistency have been studied thoroughly by Zhuge et al. [Zhu95, Zhu96] who define four levels of consistency. Strong and complete consistency have received most attention. In our formulation, complete consistency requires all sources to be either CHAC or to be DAW with an additional property that deltas should contain all changes that have occurred. A source that propagates *only net changes* (e.g. a tuple which is inserted and then immediately deleted corresponds to no change) will not be sufficient to ensure complete consistency. Only specialised applications are likely to require that all state changes of all sources should be reflected. For this reason we have focused on strong consistency.

There are several studies on how to select and efficiently maintain views within a warehouse environment (for example [Har96, Gup99]). This issue is fundamentally different from the one addressed here, as such views are defined within the DW over local data for query optimisation purposes.

6. Conclusions and Future Work

6.1. Summary

In this report we have extended work on single source DW maintenance policies to multiple sources, with the aim of developing a set of usable heuristics for policy selection. As the state space of applicable policies is very large, we have used both analytic reasoning and cost model analysis to develop a set of heuristics for refresh policy selection. The significance and impact of source capabilities was evident from earlier work. We have therefore extended a single source cost model to

multiple sources in a composable way that can include various join alternatives (hash and nested loop join are considered in this report) and an additional source capability (SAW).

A policy analyser (PAM) has been developed using the cost model, and has proved helpful in analysing the space of selected policies in detail. In particular, it has allowed us to drill down once best policies were chosen, using various combinations of costs and QoS. This allowed us to further substantiate the findings of the earlier analysis of policies, and highlighted dependencies related to source characteristics.

Finally, a testbed (TMID) has been built and used to test the policies using an XML web server and/or relational database as sources. The results obtained from TMID confirm our earlier analysis, and the cost model predictions, and in particular demonstrate the utility of the heuristics proposed for policy selection.

6.2. Future Work

The work in this report focuses on a selected part of the problem space, and more studies can be conducted to look at other aspects. First of all, we have limited the set of policies based on intuitive analysis. Some policy combinations that we have excluded have a potential to be meaningful in some situations. For example, using an auxiliary view for one supporting view but not for the other may be meaningful when only one source is SAW. Another possible extension of our work is to look at other multi-source operations such as set-operations and outer joins. We have, moreover, assumed that the views are sets (each objects has an ID). This may be relaxed to allow duplicates in supporting views and in the join view. How to handle maintenance of views based on three or more sources is another problem which could usefully be studied. This would include an analysis of the policies to consider. A related issue is how shared sources should be handled, when several views are defined over the same source.

It might also prove enlightening to extend the tools presented in this report (PAM and TMID) to support a wider set of scenarios.

6.3. Limitations

We have identified the problem of maintaining views over multiple, distributed, heterogeneous and autonomous sources. We have defined the search space in terms of source capabilities, system properties and policies. As no previous work, that we are aware of, has approached this problem, most of the map is still white. Although the results presented here are based on extensive analytical and empirical studies, there are many limitations. We are not making any claims for this study to be exhaustive, it should rather be seen as a first systematic look at the problem space.

First of all, we have chosen to concentrate on the joining of two sets. This implies that we have not addressed other operations over other types and collections. It should also be noted that we do not address arbitrary n-ary operations. Secondly, the cost model is formulated at a high level of abstraction. Details related to data models, query languages, optimisers and physical organisation are not captured. In addition, we make a number of simplifying assumptions when formulating costs, which have been presented above.

The comparison of policies in PAM and TMID are both limited to a selection of parameter values. Although care has been to cover a wide range, there are always other parameters that can be tested.

Finally, the suggested heuristics have been developed to offer easy-to-use guidance in policy selection. They have been shown to result in quality decisions in a number of situations, but more analysis is needed to understand their limitations. For example, they do not capture the well-known heuristic that, if the size of changes is large[5] (>20%), then recompute policies are preferable. Although we believe it to be typical for change size to be small in comparison with view size, the heuristics could obviously be improved to handle situations where this assumption is false.

6.4. Conclusions

This work demonstrates the complexity of the problem of choosing maintenance policies either for a single source or for multiple sources. We have presented a large number of maintenance policies and shown them to be meaningful. Our hope is that the derived set of heuristics can be used directly.

[5] Note that we have shown in this report that the trade-off between incremental and recompute for join views also depends on join cost. This implies that when join cost is high, larger relative change size can be allowed.

The PAM tool can also be used for analysing dependencies to the level of detail required, and the testbed (TMID) for real-world applications.

It should be noted that we have been able to extend our single source DW maintenance framework to handle multi-source views. This has been done in a straightforward way, not requiring any changes to the single source model. The single source cost model was found to be very useful when formulating costs for supporting views. This indicates that the approach is extensible and sound.

References

[Agr97] D. Agrawal, A.E. Abbadi, A.K. Singh, T. Yurek, "Efficient View Maintenance at Data Warehouses", SIGMOD Conference, 1997

[Ast02] K. Ásþórsson, "Evaluation of view maintenance with complex joins in a data warehouse environment", MSc Dissertation, HS-IDA-MD-02-301, 2002

[Col97] L.S. Colby, A. Kawaguchi, D.F. Lieuwen, I.S. Mumick, K.A. Ross, "Supporting Multiple View Maintenance Policies", SIGMOD Conference, 1997

[Elm94] R. Elmasri, S.B. Navathe, "Fundamentals of Database Systems", Benjamin/Cummings, ISBN 0-8053-1753-8, 1994

[Eng00] H. Engström, S. Chakravarthy, B. Lings, "A User-centric View of Data Warehouse Maintenance Issues", British National Conference on Databases, 2000

[Eng00b] H. Engström, S. Chakravarthy, B. Lings, "A Holistic Approach to the Evaluation of Data Warehouse Maintenance Policies", Technical report HS-IDA-TR-00-001, University of Skövde, Sweden, 2000

[Eng02] H. Engström, S. Chakravarthy, B. Lings, "A Holistic Approach to the Evaluation of Data Warehouse Maintenance Policies", EDBT, 2002

[Eng02b] H. Engström, S. Chakravarthy, B. Lings, "Implementation and Comparative Evaluation of Maintenance Policies in a Data Warehouse Environment", British National Conference on Databases, 2002

[Feg95] L. Fegaras, D. Maier, "Towards an Effective Calculus for Object Query Languages", Sigmod Conference, 1995

[Feg00] L. Fegaras, D. Maier, "Optimizing Object Queries Using an Effective Calculus", ACM Transactions on Database Systems, 2000

[Feg01] L. Fegaras, R. Elmasri, "Query Engines for Web-Accessible XML Data", VLDB, 2001

[Gup93] A. Gupta, I.S. Mumick, V.S. Subrahmanian, "Maintaining Views Incrementally ", SIGMOD Conference, 1993

[Gup95] A. Gupta, I.S. Mumick, "Maintenance of Materialized Views: Problems, Techniques, and Applications", IEEE Data Engineering Bulletin 18(2), 1995

[Gup96] A. Gupta, H.V. Jagadish, I.S. Mumick,"Data Integration using Self-Maintainable Views", EDBT, 1996

[Gup99] H. Gupta, I.S. Mumick, "Selection of Views to Materialize Under a Maintenance Cost Constraint", International Conference on Database Theory, 1999

[Ham95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, Y. Zhuge, "The Stanford Data Warehousing Project", IEEE Data Engineering Bulletin 18(2), 1995

[Han87] E.N. Hanson, "A Performance Analysis of View Materialization Strategies", SIGMOD Conference, 1987

[Har96] V. Harinarayan, A. Rajaraman, J.D. U1lman, "Implementing Data Cubes Efficiently", SIGMOD Conference, 1996

[Hul96] R. Hull, G. Zhou, "A Framework for Supporting Data Integration Using the Materialized and Virtual Approaches", SIGMOD Conference, 1996

[Hul96b] R. Hull, G. Zhou, "Towards the Study of Performance Trade-offs Between Materialized and Virtual Integrated Views", VIEWS'96, 1996

[Jar97] M. Jarke, Y. Vassiliou, "Data Warehouse Quality Design: A Review of the DWQ Project", Conference on Information Quality, Cambridge, 1997

[Mis92] P. Mishra, M.H. Eich, "Join Processing in Relational Databases", ACM Computing Surveys, 1992

[Qua96] D. Quass , A. Gupta , I.S. Mumick , J. Widom, "Making Views Self-Maintainable for Data Warehousing", PDIS, 1996

[Rou86] N. Roussopoulos, H. Kang, "Principles and Techniques in the Design of ADMS±", IEEE Computer 19(12), 1986

[Rou95] R. Roussopoulos, C.M. Chen, S. Kelley, A. Delis, Y. Papakonstantinou, "The ADMS Project: Views "R" Us", IEEE Data Engineering Bulletin 18(2), 1995

[Seg89] A. Segev, J. Park, "Updating Distributed Materialized Views", IEEE TKDE 1(2), 1989

[Seg91] A. Segev, W. Fang, "Optimal Update Policies for Distributed Materialized Views", Management Science 37(7), 1991

[Sri88] J. Srivastava, D. Rotem, "Analytical Modeling of Materialized View Maintenance", PODS, 1988

[The99] D. Theodoratos, M. Bouzeghoub, "Data Currency Quality Factors in Data Warehouse Design", Design and Management of Data Warehouses, workshop at CAiSE, 1999

[Zho95] G. Zhou, R. Hull, R. King, J.C. Franchitti, "Data Integration and Warehousing Using H2O", IEEE Data Engineering Bulletin 18(2), 1995
[Zhu95] Y Zhuge, H. Garcia-Molina, J. Hammer, J. Widom, "View Maintenance in a Warehousing Environment", SIGMOD Conference, 1995
[Zhu96] Y. Zhuge, H. Garcia-Molina, J.L. Wiener, "The Strobe Algorithms for Multi-Source Warehouse Consistency", PDIS, 1996

Appendix A – Comparison of policies using PAM

This appendix contains the detailed result of the PAM analysis summarised in Table 10 in section 3.3.2.

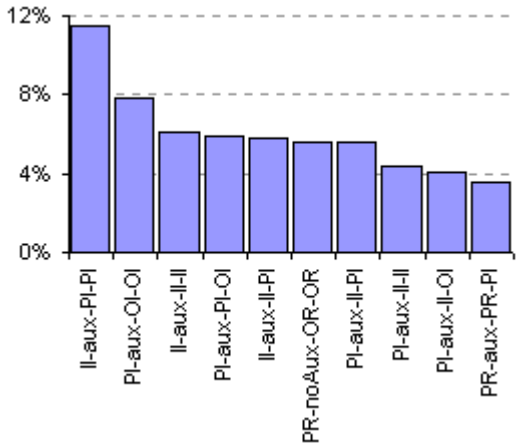

Figure 16. The 10 most-selected polices when evaluation is based on system costs combined with response time

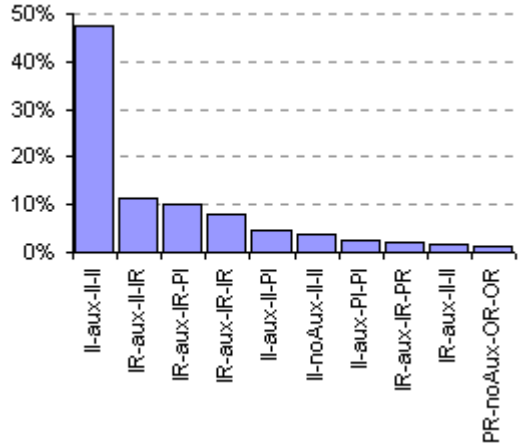

Figure 17. The 10 most-selected polices when evaluation is based on all criteria

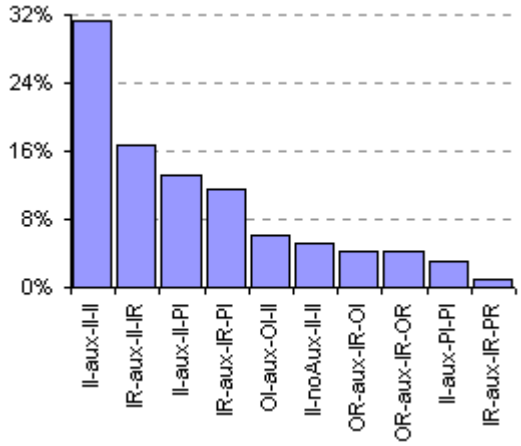

Figure 18. The 10 most-selected polices when evaluation is based on only staleness

Appendix B – Detailed result of experiments in TMID

This appendix contains additional details of the experiments described in section 4.2.2. All have been conducted in a Solaris configuration.

Table 28. The cost of all policies when the left source is DAW and the right is not DAW. The cost shown is the per query system cost in seconds.

Policy	IC	IC+Z	IC+RT	All	Policy	IC	IC+Z	IC+RT	All
O_R-aux-I_I-I_R	11.8	21.8	12.1	22.1	P_I-aux-O_I-O_I	12.9	50.7	12.9	50.8
I_R-aux-I_I-I_R	12.3	24.3	12.4	24.4	I_I-aux-I_I-P_I	13.1	50.8	13.1	50.8
O_I-aux-O_I-I_I	13.6	26.5	14	26.9	O_I-aux-O_I-P_I	13	50.5	13.4	50.9
O_R-aux-I_I-I_I	13.8	26.6	14.1	26.9	I_R-aux-I_R-P_R	18.4	51	18.4	51
I_R-aux-I_I-I_I	14.3	26.9	14.4	27	I_R-aux-P_R-P_R	18.4	51	18.4	51
I_I-aux-I_I-I_I	13.7	27	13.8	27.1	P_I-aux-I_I-I_I	13.6	51.1	13.7	51.1
O_I-aux-I_I-I_I	13.7	27.8	13.9	28	P_R-aux-O_I-O_I	13.1	51.5	13.2	51.5
I_R-aux-I_R-I_R	19	30.2	19.1	30.2	O_I-aux-P_I-P_I	13	52.9	13.3	53.1
O_R-aux-I_R-I_R	18.7	30.1	19	30.4	I_I-noAux-I_I-P_I	17.5	53.2	17.5	53.2
O_R-aux-O_I-O_R	11.1	22	20.8	31.7	O_I-noAux-O_I-O_I	17.7	37.5	33.4	53.3
O_R-aux-I_I-O_R	11.3	22.2	21	31.9	P_R-aux-O_R-O_R	22.8	53.3	22.8	53.3
I_I-noAux-I_I-I_I	18.5	31.9	18.6	32	P_R-aux-P_I-O_I	13	54	13.1	54
O_I-aux-O_I-O_I	12.9	25.5	24.2	36.8	O_R-aux-P_R-P_R	16.2	53.8	16.7	54.2
O_R-aux-O_I-O_I	12.9	26	24.3	37.4	P_R-aux-I_R-P_R	17.3	54.5	17.4	54.5
O_R-aux-I_I-O_I	13	26.2	24.4	37.6	I_I-noAux-P_I-P_I	17.4	54.5	17.5	54.5
O_R-aux-I_I-P_R	11	38	11.3	38.3	P_R-aux-I_R-O_I	20.1	54.9	20.2	55
O_R-aux-I_R-O_R	18.3	29.4	28.4	39.5	P_R-aux-I_R-I_R	18.6	55.4	18.7	55.4
I_R-aux-P_I-P_I	13.8	41.5	13.8	41.5	O_I-noAux-P_I-I_I	19.5	50.5	24.5	55.5
O_R-aux-P_I-O_R	11.1	33	20.7	42.6	P_R-aux-P_I-O_R	11.4	55.7	11.4	55.8
O_I-noAux-I_I-I_I	19.7	38	24.7	43.1	O_R-aux-P_R-O_R	18.1	46	28.1	55.9
I_I-aux-P_I-P_I	13.1	43.3	13.2	43.3	P_I-noAux-O_I-O_I	17.7	56.1	17.8	56.2
O_I-noAux-O_I-I_I	19.7	38.8	24.9	44	P_R-aux-I_I-P_R	10.9	57.2	11	57.2
O_R-aux-P_I-P_R	11	44	11.3	44.3	I_R-aux-I_R-P_I	20.2	57.4	20.2	57.5
P_R-aux-I_I-O_R	11.4	44.7	11.4	44.7	P_I-noAux-O_I-I_I	20.5	58.9	20.5	58.9
O_I-aux-I_I-P_I	13	44.6	13.2	44.8	P_I-noAux-I_I-I_I	19.9	60.4	20	60.4
I_R-aux-I_I-P_I	13.5	45.5	13.6	45.5	P_I-noAux-P_I-O_I	17.7	60.4	17.8	60.5
P_R-aux-I_I-I_R	11.8	45.5	11.9	45.5	O_R-aux-I_R-P_I	20.4	60.5	20.8	60.8
P_I-aux-P_I-O_I	13	45.7	13.1	45.7	P_R-aux-P_I-P_R	11.1	60.9	11.2	61
I_R-aux-P_I-P_R	11.4	45.7	11.5	45.7	P_R-noAux-O_R-O_R	22.5	61.2	22.5	61.3
O_R-noAux-O_R-O_R	22.4	34.8	33.8	46.2	O_R-aux-P_R-O_I	20.4	49.1	32.7	61.5
O_R-aux-P_I-P_I	13.1	46.1	13.4	46.4	O_I-noAux-O_I-P_I	18.3	57.6	23.1	62.4
I_R-aux-I_I-P_R	11.5	46.5	11.5	46.6	P_R-aux-P_R-O_I	18.6	62.4	18.7	62.5
O_R-aux-I_R-O_I	20.6	35	32.7	47.1	P_I-noAux-P_I-I_I	19.8	64.5	19.9	64.6
O_R-aux-O_R-O_R	22.8	36.1	34.4	47.7	P_R-aux-P_R-O_R	16.3	65	16.4	65
P_R-aux-O_I-O_R	11	48.1	11.1	48.2	P_I-aux-I_I-P_I	12.9	65.8	12.9	65.9
P_R-aux-I_I-I_I	13.8	48.3	13.8	48.3	P_I-aux-P_I-P_I	12.9	66.5	13	66.5
O_R-aux-I_R-P_R	18.5	48.1	18.8	48.5	O_I-noAux-P_I-P_I	18.5	62	23.2	66.7
P_R-aux-I_I-O_I	13	48.9	13.1	49	P_I-noAux-P_I-P_I	18.5	66.8	18.5	66.9
O_R-aux-P_I-O_I	13	37.6	24.4	49	P_R-aux-P_I-P_I	12.9	68.9	13	68.9
P_R-aux-I_R-O_R	17.4	49.1	17.5	49.1	P_R-aux-I_R-P_I	19.9	70.1	19.9	70.2
P_I-aux-I_I-O_I	13	49.6	13	49.7	P_R-aux-I_I-P_I	13	75.1	13.1	75.1
O_R-aux-I_I-P_I	13.2	50.4	13.5	50.7	P_R-aux-P_R-P_R	20.1	75.9	20.2	76

Appendix C – Result of TMID experiments on Linux

This appendix contains the detailed results from the experiments in a Linux environment.

Table 29 shows a comparison of I_I-aux-I_I-I_I and I_I-noAux-I_I-I_I when SAW and DAW capabilities are varied for the two sources. Both hash-based and nested loop joining have been used.

Table 29. Analysing the impact of SAW and auxiliary views. The cost shown is the per query system cost in seconds.

					Hash-based join				Nested loop join			
Left source DAW	Left source SAW	Right source DAW	Right source SAW	Policy	Left wrapper cost	Integrator cost	Right wrapper cost	Total cost (IC)	Left wrapper cost	Integrator cost	Right wrapper cost	Total cost (IC)
Yes	Yes	Yes	Yes	II-aux-II-II	0.265	0.034	0.247	0.546	0.268	0.566	0.261	1.095
Yes	Yes	Yes	Yes	II-noAux-II-II	0.249	0.808	0.235	1.292	0.247	0.826	0.238	1.311
Yes	Yes	Yes	No	II-aux-II-II	0.269	0.034	0.247	0.55	0.258	0.585	0.255	1.098
Yes	Yes	Yes	No	II-noAux-II-II	0.241	6.409	0.246	6.896	0.247	6.734	0.247	7.228
Yes	Yes	No	Yes	II-aux-II-II	0.336	0.044	6.458	6.838	0.335	0.616	6.567	7.518
Yes	Yes	No	Yes	II-noAux-II-II	0.324	1.285	6.457	8.066	0.321	1.283	6.533	8.137
Yes	Yes	No	No	II-aux-II-II	0.332	0.044	6.458	6.834	0.321	0.629	6.598	7.548
Yes	Yes	No	No	II-noAux-II-II	0.294	6.904	6.562	13.76	0.32	7.305	6.566	14.191
Yes	No	Yes	Yes	II-aux-II-II	0.26	0.034	0.246	0.54	0.266	0.574	0.25	1.09
Yes	No	Yes	Yes	II-noAux-II-II	0.258	6.671	0.237	7.166	0.254	6.917	0.246	7.417
Yes	No	Yes	No	II-aux-II-II	0.268	0.033	0.248	0.549	0.269	0.575	0.253	1.097
Yes	No	Yes	No	II-noAux-II-II	0.266	12.61	0.265	13.141	0.256	13.308	0.25	13.814
Yes	No	No	Yes	II-aux-II-II	0.319	0.045	6.481	6.845	0.321	0.63	6.571	7.522
Yes	No	No	Yes	II-noAux-II-II	0.339	7.301	6.503	14.143	0.337	7.559	6.56	14.456
Yes	No	No	No	II-aux-II-II	0.349	0.038	6.473	6.86	0.343	0.603	6.622	7.568
Yes	No	No	No	II-noAux-II-II	0.322	12.923	6.616	19.861	0.302	13.554	6.575	20.431
No	Yes	Yes	Yes	II-aux-II-II	6.665	0.045	0.344	7.054	6.771	0.619	0.321	7.711
No	Yes	Yes	Yes	II-noAux-II-II	6.641	1.235	0.312	8.188	6.73	1.29	0.32	8.34
No	Yes	Yes	No	II-aux-II-II	6.65	0.045	0.354	7.049	6.761	0.6	0.338	7.699
No	Yes	Yes	No	II-noAux-II-II	6.692	7.073	0.331	14.096	6.692	7.451	0.322	14.465
No	Yes	No	Yes	II-aux-II-II	7.673	0.044	7.315	15.032	7.771	0.651	7.521	15.943
No	Yes	No	Yes	II-noAux-II-II	7.697	1.949	7.381	17.027	7.668	2.141	7.458	17.267
No	Yes	No	No	II-aux-II-II	7.699	0.039	7.31	15.048	7.806	0.654	7.553	16.013
No	Yes	No	No	II-noAux-II-II	7.756	7.838	7.441	23.035	7.83	8.095	7.445	23.37
No	No	Yes	Yes	II-aux-II-II	6.662	0.047	0.329	7.038	6.736	0.605	0.334	7.675
No	No	Yes	Yes	II-noAux-II-II	6.76	7.213	0.305	14.278	6.733	7.493	0.312	14.538
No	No	Yes	No	II-aux-II-II	6.711	0.042	0.348	7.101	6.75	0.601	0.336	7.687
No	No	Yes	No	II-noAux-II-II	6.746	13.147	0.348	20.241	6.781	13.696	0.342	20.819
No	No	No	Yes	II-aux-II-II	7.77	0.044	7.316	15.13	7.813	0.681	7.49	15.984
No	No	No	Yes	II-noAux-II-II	7.827	8.071	7.462	23.36	7.784	8.163	7.461	23.408
No	No	No	No	II-aux-II-II	7.747	0.054	7.404	15.205	7.862	0.645	7.506	16.013
No	No	No	No	II-noAux-II-II	7.825	13.843	7.466	29.134	7.789	14.703	7.397	29.889

Table 30 shows the result of comparing recomputed and incremental policies for different join techniques and for all combinations of DAW capability in sources.

Table 30. Comparison of recomputed and incremental policy. The cost shown is the per query system cost in seconds.

Join Technique	**Left source DAW**	**Right source DAW**	**P_I-aux-O_I-O_I**	**P_R-aux-O_R-O_R**
Hash	Yes	Yes	0.856	3.596
Hash	Yes	No	1.840	3.593
Hash	No	Yes	1.697	4.083
Hash	No	No	4.193	4.093
Nested loop	Yes	Yes	1.464	7.137
Nested loop	Yes	No	2.502	7.052
Nested loop	No	Yes	2.361	7.152
Nested loop	No	No	4.216	7.068

Table 31, 32, and 33 shows the result for a set of experiments where all policies are compared with different DAW capabilities in the sources.

Table 31. The cost of all policies when no source is DAW. The cost shown is the per query system cost in seconds.

Policy	IC	IC+Z	IC+RT	All	Policy	IC	IC+Z	IC+RT	All
O_R-aux-I_R-I_R	14.6	24.4	14.7	24.6	P_R-aux-I_R-O_R	15.1	48.5	15.1	48.5
I_R-aux-I_R-I_R	15	24.7	15	24.7	P_R-aux-I_I-O_R	15.4	48.5	15.4	48.5
I_R-aux-I_I-I_R	15.2	25	15.2	25	P_R-aux-P_R-O_I	13.1	49.1	13.2	49.1
O_R-aux-I_I-I_R	14.8	24.9	14.9	25	P_I-noAux-P_I-O_I	14	49.1	14	49.1
I_I-aux-I_I-I_I	15.1	25.3	15.1	25.3	P_R-aux-P_I-O_R	13	49.1	13	49.1
O_I-aux-I_I-I_I	15.1	25.5	15.2	25.5	P_I-noAux-I_I-I_I	17.6	49.2	17.6	49.2
I_R-aux-I_I-I_I	15.5	25.9	15.5	25.9	O_I-noAux-P_I-P_I	15.4	47	17.6	49.2
O_R-aux-I_I-I_I	15.3	25.8	15.4	25.9	P_R-aux-I_I-O_I	15.4	49.2	15.4	49.3
I_I-noAux-I_I-I_I	17.1	27.3	17.1	27.3	P_R-aux-I_R-O_I	16.5	49.3	16.5	49.4
O_I-aux-O_I-I_I	14.8	25.4	21.6	32.3	I_I-aux-P_I-P_I	19.6	49.5	19.6	49.5
O_I-noAux-I_I-I_I	17.9	30	20.2	32.3	P_I-aux-I_I-O_I	15.1	49.7	15.1	49.7
O_R-aux-I_R-O_I	15.7	26.9	22.6	33.9	P_R-aux-I_I-I_R	14.9	49.7	14.9	49.7
O_R-aux-I_R-O_R	15.4	27.3	22.1	34	P_R-aux-I_R-I_R	14.7	50	14.7	50
O_R-aux-I_I-O_R	15.6	27.7	22.4	34.5	P_R-aux-I_I-I_I	15.1	50	15.1	50
O_R-aux-I_I-O_I	16	27.8	23	34.8	O_R-aux-P_I-P_I	20.9	50.2	21	50.3
O_I-noAux-O_I-I_I	16.1	27	24.2	35.1	P_I-aux-I_I-I_I	15	50.3	15	50.3
O_R-aux-O_R-O_R	23.2	25	34.1	35.9	O_R-aux-P_R-O_I	14.8	44.2	21.4	50.8
O_R-noAux-O_R-O_R	23	25.2	34	36.1	P_R-aux-P_I-O_I	13.7	50.9	13.7	50.9
O_R-aux-O_I-O_R	23.3	25.6	34.5	36.8	P_R-aux-P_R-O_R	12.9	51.2	12.9	51.2
O_I-aux-O_I-O_I	23.7	26	34.8	37.1	O_I-noAux-O_I-P_I	15.7	43.9	23.7	51.9
O_R-aux-O_I-O_I	23.7	26.2	34.9	37.4	O_I-noAux-P_I-I_I	18.4	49.9	20.5	52
I_R-aux-I_R-P_I	15	39.6	15	39.6	P_R-noAux-O_R-O_R	23.3	52.1	23.3	52.2
O_I-noAux-O_I-O_I	24.2	28	36.1	39.9	P_R-aux-I_R-P_R	15.2	52.5	15.2	52.5
I_R-aux-P_R-P_R	13.3	40.6	13.3	40.6	P_R-aux-O_R-O_R	23.7	53	23.7	53
I_I-aux-I_I-P_I	15.2	40.6	15.2	40.6	P_R-aux-O_I-O_R	23.6	53.4	23.6	53.4
O_I-aux-P_I-P_I	13.1	41.5	13.2	41.6	O_R-aux-P_I-O_I	16.7	47.1	23.6	54.1
O_R-aux-I_R-P_R	14.2	41.7	14.3	41.8	P_R-aux-P_I-P_I	13.7	54.4	13.7	54.4
O_R-aux-I_I-P_R	15.4	41.9	15.5	42	P_R-aux-O_I-O_I	24.3	55	24.3	55
O_R-aux-P_I-P_R	13.1	42.2	13.2	42.3	O_I-aux-O_I-P_I	16.1	47.8	23.5	55.2
I_R-aux-I_I-P_I	16	42.5	16	42.5	P_I-aux-P_I-O_I	13.3	55.4	13.3	55.4
O_I-aux-I_I-P_I	15	42.6	15	42.6	P_I-aux-O_I-O_I	24.1	55.6	24.1	55.6
I_I-noAux-I_I-P_I	17	42.7	17	42.7	P_I-noAux-O_I-O_I	25.1	55.7	25.1	55.7
I_R-aux-P_I-P_I	13.5	42.8	13.5	42.8	O_R-aux-P_I-O_R	16	49.6	22.7	56.4
I_I-noAux-P_I-P_I	14.1	42.8	14.1	42.8	P_R-aux-I_I-P_I	15.1	58.4	15.1	58.5
I_R-aux-I_I-P_R	16	42.8	16	42.9	P_R-aux-P_R-P_R	15.9	60.5	15.9	60.5
O_R-aux-P_R-P_R	13.2	42.8	13.3	42.9	P_I-aux-I_I-P_I	16.9	61.3	16.9	61.4
O_R-aux-I_R-P_I	15.5	43	15.6	43.1	P_R-aux-I_R-P_I	14.8	62	14.8	62
I_R-aux-I_R-P_R	14.8	43.4	14.8	43.4	P_I-noAux-P_I-P_I	15.3	64.7	15.3	64.7
I_R-aux-P_I-P_R	13.8	43.5	13.8	43.5	P_I-noAux-P_I-I_I	20.5	69.2	20.5	69.2
O_R-aux-P_R-O_R	14.2	40.3	20.4	46.5	P_I-aux-P_I-P_I	20.7	69.5	20.7	69.5
O_R-aux-I_I-P_I	17	46.9	17.1	47	P_R-aux-I_I-P_R	15.6	70.4	15.6	70.4
P_I-noAux-O_I-I_I	16.7	47.2	16.8	47.2	P_R-aux-P_I-P_R	19	72.3	19	72.3

Table 32. The cost of all policies when left source is DAW and right source is not DAW. The cost shown is the per query system cost in seconds.

Policy	IC	IC+Z	IC+RT	All	Policy	IC	IC+Z	IC+RT	All
I_I-aux-I_I-I_I	6.84	11.6	6.85	11.6	P_I-noAux-O_I-O_I	7.8	36.5	7.81	36.5
I_R-aux-I_I-I_R	6.88	11.7	6.89	11.7	I_I-noAux-P_I-P_I	7.45	36.6	7.46	36.6
O_I-aux-I_I-I_I	6.8	11.7	6.85	11.7	P_R-aux-P_I-O_R	6.75	36.7	6.76	36.7
O_R-aux-I_I-I_R	6.68	11.6	6.8	11.7	O_I-aux-O_I-P_I	6.89	36.4	7.26	36.7
O_R-aux-I_I-I_I	6.95	11.8	7.06	11.9	I_I-noAux-I_I-P_I	8.35	37	8.36	37
I_R-aux-I_I-I_I	7.13	11.9	7.14	11.9	O_R-aux-I_R-P_I	14.8	37.4	14.9	37.5
O_R-aux-I_I-O_R	6.68	6.98	12.3	12.6	P_R-aux-O_I-O_R	7.21	37.6	7.22	37.7
O_I-aux-O_I-I_I	6.83	12.4	7.18	12.7	P_R-aux-I_I-I_R	6.65	37.8	6.67	37.8
I_I-noAux-I_I-I_I	8.05	12.8	8.06	12.9	P_R-aux-I_I-P_I	7.06	37.8	7.07	37.9
O_R-aux-I_I-O_I	6.89	7.32	12.7	13.1	O_R-aux-P_I-O_I	6.9	32.8	12.7	38.7
O_I-noAux-I_I-I_I	8.03	15	9.13	16	O_I-noAux-P_I-P_I	7.84	37.5	9.04	38.7
O_I-noAux-O_I-I_I	7.98	15.2	9.37	16.6	P_R-aux-I_I-I_I	6.92	39.4	6.93	39.4
O_R-aux-O_I-O_R	7.05	12.2	12.8	18	P_R-aux-O_I-O_I	7.37	40	7.38	40
O_I-aux-O_I-O_I	7.08	12.7	13	18.6	O_I-noAux-O_I-P_I	8.02	38.6	9.56	40.1
O_R-aux-O_I-O_I	7.28	12.8	13.2	18.8	O_R-aux-P_R-P_R	12.7	40.8	12.8	40.9
O_I-noAux-O_I-O_I	7.81	14	14.3	20.5	P_I-noAux-O_I-I_I	8.14	42.2	8.15	42.2
O_R-aux-I_R-I_R	14.7	24.5	14.8	24.6	I_R-aux-P_R-P_R	13.4	42.3	13.4	42.3
I_R-aux-I_R-I_R	15	24.7	15	24.7	O_R-aux-I_R-P_R	14.8	42.5	14.9	42.6
I_I-aux-I_I-P_I	6.74	31.9	6.75	31.9	P_I-noAux-P_I-I_I	8.05	43.3	8.06	43.3
O_R-aux-P_I-O_R	6.69	27.8	12.3	33.4	I_R-aux-I_R-P_I	15.3	44	15.3	44
O_R-aux-I_R-O_I	15.7	26.8	22.8	33.8	P_R-aux-P_I-O_I	6.79	45.2	6.8	45.2
O_R-aux-I_R-O_R	15.4	27.1	22.3	33.9	I_R-aux-I_R-P_R	15.6	45.4	15.6	45.4
I_R-aux-I_I-P_R	7.04	34.1	7.05	34.1	P_I-noAux-P_I-O_I	7.43	45.8	7.45	45.8
O_I-aux-I_I-P_I	6.81	34.2	6.86	34.2	P_R-aux-I_R-O_I	14.7	47.1	14.7	47.2
O_R-aux-P_I-P_I	6.9	34.1	7	34.2	P_R-aux-I_I-P_R	6.67	47.7	6.68	47.7
O_R-aux-I_I-P_I	6.99	34.3	7.09	34.4	P_I-aux-I_I-P_I	6.81	47.9	6.83	47.9
I_I-aux-P_I-P_I	6.67	34.9	6.68	34.9	P_R-aux-I_R-O_R	15.5	49.1	15.5	49.2
P_I-aux-I_I-O_I	6.77	35.3	6.78	35.3	P_R-aux-I_R-I_R	14.6	50.4	14.6	50.4
O_R-aux-P_I-P_R	7.02	35.3	7.13	35.4	P_R-aux-O_R-O_R	23.6	52.3	23.6	52.3
P_I-aux-I_I-I_I	6.81	35.5	6.82	35.5	P_I-aux-P_I-O_I	6.65	52.7	6.66	52.7
O_R-aux-O_R-O_R	23.1	24.8	34	35.7	P_R-noAux-O_R-O_R	23.3	53.3	23.3	53.3
I_R-aux-P_I-P_I	7.81	35.8	7.82	35.8	P_R-aux-P_I-P_I	6.73	53.3	6.74	53.4
P_I-noAux-I_I-I_I	7.74	35.9	7.75	35.9	O_R-aux-P_R-O_I	15.5	46.9	22.2	53.6
P_R-aux-I_I-O_I	7.2	35.9	7.21	35.9	O_R-aux-P_R-O_R	14.8	48.5	21.4	55
P_I-aux-O_I-O_I	7.07	36	7.08	36	P_I-noAux-P_I-P_I	7.39	55.2	7.41	55.2
I_R-aux-P_I-P_R	6.82	36.1	6.83	36.1	P_R-aux-P_I-P_R	6.41	55.2	6.42	55.2
I_R-aux-I_I-P_I	7.29	36.1	7.3	36.1	P_R-aux-I_R-P_R	15.7	55.3	15.7	55.3
O_I-noAux-P_I-I_I	8.08	35.1	9.18	36.2	P_I-aux-P_I-P_I	6.71	55.6	6.72	55.6
P_R-aux-I_I-O_R	6.97	36.2	6.99	36.2	P_R-aux-P_R-O_R	22	56.2	22	56.2
O_R-aux-I_I-P_R	6.66	36.1	6.77	36.2	P_R-aux-P_R-O_I	13.4	58.8	13.4	58.8
O_I-aux-P_I-P_I	6.71	36.3	6.76	36.3	P_R-aux-I_R-P_I	14.8	61.6	14.8	61.6
O_R-noAux-O_R-O_R	23.1	25.6	34	36.5	P_R-aux-P_R-P_R	16.4	62.9	16.5	63

Table 33. The cost of all policies when both sources are DAW. The cost shown is the per query system cost in seconds.

Policy	IC	IC+Z	IC+RT	All	Policy	IC	IC+Z	IC+RT	All
I_I-aux-I_I-I_I	0.56	0.56	0.57	0.57	O_I-noAux-P_I-P_I	1.32	29	2.07	29.8
O_I-aux-I_I-I_I	0.53	0.53	0.57	0.57	O_I-noAux-P_I-I_I	1.3	29.1	2.04	29.8
I_R-aux-I_I-I_I	0.72	0.72	0.73	0.73	O_R-aux-I_R-P_I	6.96	30.4	7.07	30.5
O_I-aux-O_I-I_I	0.56	0.56	0.83	0.83	I_R-aux-P_I-P_R	7.3	31.2	7.32	31.2
O_R-aux-I_I-I_I	0.59	0.9	0.68	0.99	I_R-aux-I_R-P_I	7.05	32.1	7.06	32.1
O_R-aux-I_I-O_I	0.64	0.71	0.94	1.01	O_R-aux-P_I-P_R	6.61	32.8	6.71	32.9
O_I-aux-O_I-O_I	0.83	0.83	1.28	1.28	O_R-aux-I_R-O_R	15.4	27.7	22.1	34.5
O_R-aux-O_I-O_I	0.87	0.87	1.36	1.36	P_I-noAux-P_I-I_I	1.29	34.6	1.3	34.6
I_I-noAux-I_I-I_I	1.29	1.39	1.3	1.4	I_R-aux-I_I-P_R	7.03	35.4	7.04	35.4
O_I-noAux-I_I-I_I	1.31	1.86	2.07	2.62	O_R-aux-P_I-O_R	6.64	30.1	12.2	35.7
O_I-noAux-O_I-I_I	1.29	2.06	2.24	3.02	O_R-aux-P_R-O_I	7.14	35.5	7.52	35.9
O_I-noAux-O_I-O_I	1.54	2.29	2.68	3.43	P_R-aux-I_I-O_R	6.78	36	6.8	36
I_R-aux-I_I-I_R	6.88	11.6	6.89	11.6	O_R-noAux-O_R-O_R	23	25.3	33.9	36.2
O_R-aux-I_I-I_R	6.69	11.7	6.8	11.8	O_R-aux-O_R-O_R	23.2	25.5	34.1	36.4
O_R-aux-I_I-O_R	6.64	6.97	12.2	12.6	O_R-aux-I_R-P_R	14.5	37.1	14.6	37.2
O_R-aux-I_R-O_I	6.88	13.8	7.28	14.2	P_R-aux-I_I-I_R	6.67	37.3	6.69	37.3
O_R-aux-O_I-O_R	7.04	12.2	12.8	18	P_R-aux-O_I-O_R	7.15	38	7.16	38
P_I-noAux-I_I-I_I	1.28	21.5	1.29	21.5	P_R-aux-I_R-O_I	6.89	38.5	6.9	38.5
O_I-noAux-O_I-P_I	1.32	22.2	2.29	23.2	P_I-aux-P_I-P_I	0.52	40.7	0.53	40.8
O_R-aux-I_I-P_I	0.58	23.1	0.66	23.2	P_I-aux-P_I-O_I	0.54	41.3	0.56	41.3
P_I-aux-I_I-O_I	0.55	24.3	0.56	24.3	O_R-aux-P_R-P_R	13.2	42.2	13.3	42.3
I_R-aux-I_R-I_R	15	24.5	15	24.6	P_R-aux-P_R-O_I	6.82	42.6	6.83	42.7
O_R-aux-I_R-I_R	14.7	24.5	14.8	24.6	P_R-aux-P_I-P_I	0.57	42.7	0.58	42.7
P_R-aux-I_I-O_I	0.63	24.7	0.64	24.7	I_R-aux-P_R-P_R	13.3	43.1	13.3	43.1
I_I-aux-P_I-P_I	0.54	25	0.56	25	I_R-aux-I_R-P_R	15.6	43.4	15.6	43.4
O_R-aux-P_I-O_I	0.63	24.7	0.93	25	P_I-noAux-P_I-P_I	1.27	43.7	1.28	43.7
I_I-noAux-I_I-P_I	1.28	25.6	1.29	25.6	P_R-aux-I_I-P_R	6.62	44.4	6.63	44.4
P_R-aux-I_I-I_I	0.58	26	0.59	26	P_R-aux-P_I-O_R	6.6	44.4	6.61	44.5
P_I-aux-O_I-O_I	0.77	26.1	0.79	26.1	P_R-aux-I_I-P_I	0.59	46.1	0.6	46.1
O_R-aux-P_I-P_I	0.6	26.1	0.68	26.2	P_R-aux-I_R-I_R	14.6	47.8	14.7	47.8
P_I-noAux-O_I-I_I	1.29	26.4	1.3	26.4	P_I-aux-I_I-P_I	0.51	48.2	0.53	48.2
P_I-noAux-O_I-O_I	1.5	26.6	1.51	26.6	P_I-noAux-P_I-O_I	1.31	50.8	1.32	50.8
I_R-aux-P_I-P_I	0.72	26.9	0.73	26.9	P_R-aux-I_R-O_R	14.6	52.5	14.7	52.5
P_R-aux-O_I-O_I	0.9	26.9	0.91	26.9	P_R-noAux-O_R-O_R	24.1	53.5	24.1	53.5
I_I-aux-I_I-P_I	0.54	27.6	0.55	27.6	P_R-aux-P_I-O_I	0.6	53.6	0.61	53.6
O_I-aux-O_I-P_I	0.55	27.8	0.82	28	P_R-aux-O_R-O_R	23.1	53.6	23.1	53.6
O_I-aux-I_I-P_I	0.55	28	0.59	28.1	O_R-aux-P_R-O_R	16	49.9	23	56.9
I_R-aux-I_I-P_I	0.73	28.1	0.74	28.1	P_R-aux-P_R-P_R	12.7	57	12.7	57
I_I-noAux-P_I-P_I	1.26	28.6	1.27	28.6	P_R-aux-I_R-P_I	6.88	57.3	6.89	57.4
O_I-aux-P_I-P_I	0.52	28.8	0.56	28.8	P_R-aux-P_R-O_R	13	58.9	13	58.9
O_R-aux-I_I-P_R	6.69	29.4	6.8	29.5	P_R-aux-I_R-P_R	14.6	60.9	14.6	60.9
P_I-aux-I_I-I_I	0.51	29.6	0.52	29.6	P_R-aux-P_I-P_R	6.53	61.1	6.54	61.1