Extending Logging for Database Snapshot Refresh

Bo Kähler and Oddvar Risnes

RUNIT - The Computing Research Centre at the University of Trondheim, N-7034 Trondheim-NTH, Norway

Abstract

A database snapshot mechanism represents a cost effective substitute for replicated data in a distributed database. The contents of a database snapshot can be periodically **refreshed** to reflect the current state of the database. In a distributed database system it is significant to reduce the cost of snapshot refresh. This can be obtained by a **differential refresh** strategy in which modifications to the base tables involved are detected.

The paper proposes two methods based on using a separate table for logging the modifications made to a base table; a **sequential** and a **condensed** logging approach. The methods have been compared for various update frequency and composition. The sequential log performs well for single snapshots if the modification set is small relative to the base table size, or if the snapshot is restrictive. In the case of large modification sets and **replicated** snapshots, the condensed logging method is to be preferred.

Introduction

Data replication is often introduced in distributed databases to improve performance and availability. By storing copies of data at sites where the data is frequently used, the need for costly, remote access is decreased and the probability of having a copy available is increased. In practice, the expected improvement in performance is hard to achieve due to the added cost of maintaining the replicated data.

Database snapshots, introduced in [ADIB 80], represent a cost effective substitute for replicated data in distributed databases. A database snapshot can be defined as a read-only replica of a selected portion of the database. Snapshots thus relax the requirement of being up-to-date. Instead, snapshots are **refreshed** i.e. made up-to-date only at specific points in time by some user-invoked or determined action. In a relational database system, a snapshot may in general be defined as a restricted join over a set of tables, similar to a general query.

Since a snapshot can be viewed as a system maintained table, refresh can easily be achieved by rebuilding the snapshot table from scratch at each refresh request. We call this a **full refresh** strategy. However, if few or no modifications are made to the base tables involved in the snapshot definition since the last refresh, much of the refresh processing will be redundant compared with the previous refreshes. In a distributed database, we may end up sending virtually the same snapshot as already stored remotely.

A **differential refresh** strategy is therefore based on detecting **modifications** made to each of the base tables involved in the snapshot. Then, by combining these modifications, refresh messages are computed and sent to the snapshot. In this paper, we propose two methods based on using a separate table to log modifications to the base tables; a **sequential** and a **condensed** logging approach. The differential snapshot refresh mechanism as proposed is designed to fulfill the following requirements:

1) The mechanism supports restrictive as well as full-copy snapshots. 2) The mechanism supports replicated as well as one-copy snapshots. 3) The mechanism supports independent snapshots on one table.

We will concentrate our discussion on snapshots based on a single table (i.e. no joins in the definition). We first describe the two logging methods designed to fulfill the first requirement. We then discuss how the logging approaches can be extended to support replicated as well as independent snapshots. We conclude with a cost analysis for each method under various update frequency and composition. In the analysis, the processing cost is considered as well as the message delivery cost.

The work presented here was done along with the development of the distributed relational database system MIMER* [MIME 85]. Among the characteristics of MIMER* is the enforcement of a primary key, optimistic concurrency control for transaction handling, and portability between dissimilar computer systems. We therefore discuss the various methods in light of this system.

Related Work

Most work on snapshots has been done within the framework of the R* distributed database management systems project at IBM Research in San Jose [LOHM 85].

The distributed DBMS INGRES/STAR [MCCO 86] supports something similar called **deferred updates** in which the local copy of the replicated data may be updated directly. The modifications are then added to an intention list and propagated to other sites holding copies of the same data. The updates are finally taken care of by transactions at the replica site. Deferred updates thus relax the requirement of mutual replica consistency. We will return to the deferred updates method later on since a table being a **read-only replica**, updated deferred, corresponds to a snapshot.

Also, some work on refresh processing applicable to snapshots has been performed in connection with the related issue **materialized views** or **view instantiation** [BLAK 86] . Here, a regular database view is instantiated in the form of a table, rather than as a definition which is evaluated each time it is referenced in a query. Maintenance of such instantiated views is suggested to be performed in a differential manner. The paper presents a method by which one can deduce the modifications necessary to the view table from the modifications to the tables referenced in the view. In contrast to our paper, it does not discuss how the base table modifications are determined.

In fact, most papers have treated the subject at the conceptual level. The only paper describing a specific implementation is [LIND 86] . This paper presents an algorithm which as its main objective has the minimization of the number of messages sent when refreshing a snapshot. The algorithm is based on annotating the base table with two columns, a previous tuple address and a timestamp. Unlike the methods proposed in our paper, the algorithm does not include the processing cost of refresh.

Finally, our work is applicable to differential maintenance of complex objects [HASK 82] stored in relational CAD/CAM databases [FRØS 86] . In such environments, complex objects are retrieved for processing from a central database (check out). Instead of writing the entire changed object back in to the central store, a condensed differential log is maintained and later merged with the original tables, which have remained locked in the meantime.

Performance Objectives of Differential Refresh Strategies

Differential refresh strategies are based on detecting base table modifications. When considering snapshot definitions not involving joins, qualification of the snapshot is restricted to single tuple expressions. Modifications to a snapshot thus become a subset of the modifications to its base table.

The identification of this subset can be more or less accurate. Some methods are unable to determine if a tuple did satisfy a particular snapshot restriction prior to deletion. However, extraneous identification of deletions or updates (not present in the snapshot prior to refresh) can easily be discarded by the snapshot refresh processing at the snapshot site.

When calculating the overall performance impact of a snapshot there are several factors to consider. First, there is the possible overhead connected to the "normal" modification activity on the base table (i.e. the identification of inserts, updates and deletes). Then there is the cost of refresh processing at the site of the base table. Finally there is the cost of communicating the refresh messages to the snapshot site and the refresh processing done there. The latter two are largely proportional to the number of messages sent from the base table site.

Previous work on snapshots has focused on the minimization of refresh messages sent when refreshing a snapshot. However, as suggested in [SELI 79] and confirmed in [MACK 86], one can not neglect the cost of local processing in distributed queries. Earlier work has also suggested that the cost of maintaining a snapshot should fall on the snapshot refresher. While this is an appropriate strategy in many cases, there are situations in which this would lead to unacceptably high costs overall. As we shall see, this is especially important when we consider multiple snapshots defined on a table.

Sequential Logging

Already the early papers on snapshots [ADIB 80] suggested that the database log was used to detect base table modifications. The database log keeps a record of all modifications to the database since the last back-up. Extracting the modifications belonging to *one* table will thus be time consuming.

A slightly different variant is used in INGRES/STAR to support deferred updates of replicas [MCCO 86]. Deferred updates make use of a separate log for all of the base tables (in addition to the normal log). Any modification to a base table is written sequentially to this log together with the name of the table. The replicas are updated (or refreshed) on demand or periodically by a demon process. Naturally, normal processing becomes more expensive since each tuple modification is written twice.

The sequential change log approach we describe is similar to the solution of INGRES/STAR. However, we assume that each base table has its own change log organized as a sequential table. In this way, the log needs only to be kept to the next snapshot refresh time. The refresh is basically done by sending all qualifying modifications to the snapshot site and redoing them on the snapshot. After refresh, the log is erased.

	Value		
Resort	**Country**	**Price Level**	**Comment**
Brighton	England	8	unchanged
Cannes	France	8	was 7
Crete	Greece	4	inserted
-	-	-	deleted
Florence	Italy	5	unchanged
Tenerife	Spain	4	unchanged

Figure 1. Holiday Resorts Base Table: The comment column is added to the base table for clarity only.

The sequential logging approach does impose some overhead on the normal processing of a base table. When a tuple is inserted, updated or deleted, a log entry is written to the log reflecting the modification to the base table. The modification as such may either be logged as an entry containing the before and after images of the tuple changed, or the after image only. Figure 2 gives an example of a sequential change log with after images only. The example shows the changes made to a base table describing holiday resorts. Figure 1 displays the base table itself. In the case of insert and update, the value of the new tuple is added to the log. In the case of delete, the primary key of the tuple is added to the log. Each log entry is provided with a label identifying the type of modification.

Modif.		**Value**	
Type	**Resort**	**Country**	**Price Level**
UPD	Cannes	France	8
INS	Crete	Greece	3
DEL	Beirut	Lebanon	
UPD	Crete	Greece	4

Figure 2. Sequential Logging: After images only.

As the sequential log records all modifications made to the base table since the last refresh, snapshot refresh can be carried out quite cheaply in terms of processing costs. The sequential log is scanned, and for each log entry, a modification message is sent to the snapshot site iff the log entry qualifies.

As it is, only log entries of type insert can be checked with the snapshot definition to see if the new tuple qualifies for the snapshot. This verification can not be carried out for the update and delete log entries as their before tuple images (and thus their presence in the snapshot table) are unknown. As a consequence, all updates and deletions must be signalled to the snapshot. The refresh process at the snapshot site must therefore be prepared to handle modifications to tuples not present in the snapshot table. Extraneous updates and deletes do however not result in an incorrect refresh of the snapshot, they merely cause an unnecessary overhead.

Given a snapshot S_1 with snapshot restriction *PriceLevel* < 7, then the refresh messages sent to the snapshot will be all the log entries of the sequential log in Figure 2. As can be seen from the figure, the change of price level from 7 to 8 for Cannes is encounted in the refresh messages even though the tuple is not qualified for the snapshot before or after the update.

As pointed out in [LIND 86], the inaccuracy in selecting the relevant modifications potentially increases as the snapshot qualification becomes more restrictive. The situation may be remedied by saving the old value of each tuple prior to its modification. In the case of delete, the full tuple (as opposed to the primary key only) is written to the log. In the case of update, the before image is added to the log immediately followed by the new tuple value. Figure 3 displays the revised sequential log.

Modif.		Value	
Type	Resort	Country	Price Level
UPD	Cannes	France	7
-	Cannes	France	8
INS	Crete	Greece	3
DEL	Beirut	Lebanon	6
UPD	Crete	Greece	3
-	Crete	Greece	4

Figure 3. Sequential Logging: Both before and after images recorded.

The snapshot refresh process at the base table site may now discard all log entries not qualifying the snapshot restriction as follows:

Inserts not satisfying the restriction are not sent.

Updates in which the before *and* the after image of the updated tuple do not satisfy the restriction are not sent. All other updates will be sent as insert, update or delete message depending on which out of the before and after image that qualify. If both qualify, then the update is sent as an update message holding the new value. If the before image *only* qualifies, a delete message holding the primary key is sent. If the after image *only* qualifies, then an insert message holding the new tuple value is sent.

Deletes not satisfying the restriction are not sent. Those that qualify are sent as delete messages holding the primary key.

Figure 4 shows the refresh messages sent in the revised approach. A sequential log recording both before and after images requires more storage. However, the number of refresh messages may be greatly reduced in the case of restrictive snapshots.

S_1 : **Restriction** = PriceLevel < 7			
Modif.		Value	
Type	Resort	Country	Price Level
INS	Crete	Greece	3
DEL	Beirut	Lebanon	
UPD	Crete	Greece	4

Figure 4. Refresh Messages to Snapshot Table S_1

Still, in the revised approach, unqualified updates are sent as each modification is considered separately. Several modifications to a single tuple result in just as many entries in the log. A tuple being updated several times and finally deleted may therefore cause just as many update messages and one delete message. Ideally, only a delete message is needed. However, this can not be determined without scanning the entire log.

Condensed Logging

The number of messages in the sequential logging strategy can be reduced if the log is sorted tuple-wise while preserving the order of modifications per tuple. By doing this, the change history of each tuple (represented by a sequence of modifications) can be condensed into one resulting modification (an update followed by an update followed by a delete results in a delete etc.). Since only the resulting modification is needed in order to refresh the snapshot, only this is sent.

Instead of sorting the log at the refresh time, the tuple order may be preserved during normal processing of the base table. We then arrive at the condensed log approach. The condensed log is organized as an index (e.g. B-tree), ordered on some unique tuple identifier (e.g. primary key). Each entry of the index points to (or contains) the at any time resulting modification to a tuple since last refresh. The size of the modification log is then kept down to a minimum and the intermediate write and read of the full log is eliminated. The rules for adding a modification to a tuple are as given in Figure 5.

	Stored Entry			
Modif.	NONE	Insert	Update	Delete
Insert	Insert	-	-	Update
Update	Update	Insert	Update	-
Delete	Delete	*Remove*	Delete	-

Figure 5. Merge Rules for New Modifications

If no modification entry is found for the tuple, the modification is saved as it is, i.e. this is the first modification done to the tuple after a snapshot refresh.

If an entry already exists for the tuple, an insert will be stored as an update, since the existing entry must be a delete entry for semantic reasons. An update is merged with an update into an update entry, whereas it is merged with an insert entry into an insert entry. A delete modification is merged with an update entry into a delete entry. Deleting an inserted tuple results in a removal of the entire entry for that tuple.

To overcome the problem of incorporating all deleted and updated tuples in the refresh messages sent, regardless of their qualification, one can save the old value of the tuple prior to its **first** modification after a refresh. The condensed log will thus be quite similar to the sequential log containing before and after images. There are however some dissimilarities. The condensed log is sorted tuple-wise, and a modified tuple is represented by one log entry only. In the sequential log, the update entry contains the tuple value before and after the modification. In the condensed log, the before image of the update entry is the value prior to the first modification.

Modif. Type	Value		
	Resort	Country	Price Level
DEL	Beirut	Lebanon	6
UPD	Cannes	France	7
-	Cannes'	France	8
INS	Crete	Greece	4

Figure 6. Condensed Logging: The table is ordered on primary key. The two entries for Cannes represents an update.

Figure 6 shows a condensed version of the previous sequential log (cf. Figure 3). Only one entry is found for Crete since the last update was merged into an insert. The one update entry for Cannes shows the before and after image of the tuple. Notice that the log entries are stored in the order of the primary key of the base table tuples, and the modification type. The modification type is significant for storing update modifications. In the example given, symbolic names have been used for the type. In an implementation, codes will be used so that the before images of updates will always precede the after images.

By adding the overhead of merging tuple modifications to normal processing, the local refresh evaluation is able to determine if a modified tuple was included in the snapshot since only before images of tuples satisfying the definition criteria can be stored in the snapshot. Unqualified deletes and updates can be discarded applying the rules as described for the revised sequential approach. Unlike the sequential approach, extraneous refresh messages will be avoided. For very restrictive snapshots this may result in large savings percentage wise. Returning to the example as shown in Figure 4, the last refresh message will not be sent in the condensed log approach. Instead, the first refresh message will reflect the later change of price level from 3 to 4.

Operational Aspects

The logging of the modifications as shown above can be done in much the same way as the DBMS maintains index tables. In this way one avoids altering the base table definition which leads to recompilation (provided precompilation is used in the DBMS) of all queries referring to that table, i.e. the mechanism does not affect the query compiler or interpreter of the DBMS.

In MIMER*, a special *write set* log is kept for optimistic concurrency control in addition to the normal log. In the case where the user wants to read modified tuples inside the modifying transactions, the *write set* is consulted prior to the table for reading. In a prototype, we intend to implement the *write set* as a condensed log, since the problem of locating previously written or modified tuples is similar to the problem of locating modifications on a log. *Write sets* for tables acting as base tables can thus be used for snapshot change logging at little extra cost.

Independent and Replicated Snapshots

The nature of snapshots, and the reason for using them, imply that more than one snapshot is likely to be defined on a base table when the mechanism itself is applicable. For instance, a company that has several departments may wish to replicate identical copies of a snapshot on the telephone directory to each departmental computer. This is called a **replicated** snapshot. Assume that each department is responsible for selling a subset of the products for sale by the company. Therefore, each department defines a snapshot, extracting the products sold by the department from the product catalog table. We call such snapshots **independent**.

The main difference between the two forms is that the department probably wants all replicas of a replicated snapshot refreshed **concurrently**. In contrast, independent snapshots will have their own **independent refresh** frequency (e.g. whenever a *new* product is sold by a particular department).

For replicated snapshots the obvious question is: How tolerant should the refresh of replicas be to site failures - either prior to, or during the refresh processing?

If a site becomes unavailable for refresh, there are two possibilities. The first is to abandon the refresh, waiting until *all* involved sites are available. Unfortunately, this will decrease the overall availability of "up-to-date" information in snapshots, as one site may prevent the remaining sites from being updated.

Another possibility is to continue refreshing the remaining sites, if a looser notion of replica consistency can be tolerated. In this case, what base table state should the refresh reflect for the sites coming up again later on? Again, there are two alternatives: Either, *all* replicas should reflect the base table state at the time of *invoking* the refresh operation, or each replica may reflect the most up-to-date state of the base table.

As an example, consider some product change causing updates to a product catalog. Given that all snapshot replicas on the catalog - say $S_1,S_2,...,S_n$ - are in the same state. A new product - say P_1 - is added to the catalog. At the following refresh of the replicated snapshot, the site holding S_1 is unavailable. Despite that, the remaining replicas are refreshed.

Following this, a new product P_2 is added. When the site holding snapshot S_1 becomes available, should it immediately be refreshed, and if so, should both P_1 *and* P_2 be added to the snapshot, or should only P_1 be added awaiting the next refresh for P_2 to be added to *all* replicas?

If the latter approach is chosen, information necessary to regenerate the state at the time of refresh invocation must be kept around for some time, marked appropriately so new changes can be distinguished from old ones.

Similar considerations must be taken for independent snapshots. Whereas the problem of replicas is *common refresh invocation time/differing refresh times*, independent snapshots have *dissimilar refresh invocation times/differing refresh times.*

In the following sections we will discuss how the logging approaches can be extended to support both replicated and independent snapshots.

Extended Sequential Logging

To support the independent snapshots, the sequential log (cf. Figure 3) can no longer be discarded after a refresh. The log entries will have to be kept untill *all* snapshots defined on the base table have been refreshed correctly. In order to avoid full scan of the base table for one particular snapshot refresh, a mechanism is used to identify the last refresh time of each snapshot.

Each snapshot will be associated with a **refresh mark** on the log, identifying the most current log entry refreshed for the actual snapshot. In the event of a new refresh, only log entries since the refresh mark need to be considered. Entries seen by *all* snapshots defined on the table (below the lowest refresh mark of all snapshots) can be discarded.

By associating a refresh mark with each snapshot, reflecting the time of its previous refresh, the algorithm lends itself to support the independent snapshots. The extension will also support replicated snapshots in the case where it is sufficient to allow each replica to reflect the state of the base table *at the time of the refresh* rather than refresh invocation time. Each replica is simply treated as an independent snapshot. Obviously, a mark must be kept associated with each replica.

To support replicas reflecting the table state *at refresh invocation time*, refresh of a previously unavailable replica must consider all entries from its refresh mark up to the highest refresh mark of any of its *sibling* replicas. This mark can be maintained in the log table itself, instead of keeping it in the dictionary (so as to avoid the search).

As a consequence of the extensions proposed above, the log may become quite large if a snapshot site is unavailable for long periods of time. The log reflects the modifications performed since the oldest refresh of a snapshot and up to the present. On the other hand, very little overhead is added per replica or snapshot. In addition, the method easily incorporates both types of replica support.

Extended Condensed Logging

The extensions to the condensed logging method are much along the lines of those proposed for the sequential method. Modifications for each tuple are recorded as previously described for this method, but instead of discarding them at refresh time, they are associated with a timestamp reflecting the time of refresh. Log entries holding the same timestamp thus corresponds to modifications recorded between two refresh marks on the sequential log. After refresh of one snapshot, new modifications will be timestamped NULL (and real timestamps are filled in at the next refresh). New modifications will in other words be recorded independently of the older timestamped versions.

In this manner, the log will at any time consist of a sequence of regular modifications for each tuple. Each modification reflects the changes made to the tuple between two refreshes.

Figure 7 shows a log table for this scheme. A timestamp column has been added to the table, as can be seen by the figure. The table has been refreshed three times (at the time 2:00, 3:00 and 4:00). Modifications are recorded for Crete twice (at time 3:00 and 4:00). Notice that the tuples are stored in the order of the primary key, the timestamp and the modification type like for the condensed log described previously.

Associated with each snapshot is a timestamp of its last refresh. The refresh of a particular snapshot may then proceed as follows. For each tuple that has been modified, its true modification is found by merging all of its log entries having a timestamp *newer* than the one associated with the snapshot. In other words, log entries having timestamp larger than the one associated with the snapshot (given that the tuple is qualified by the snapshot restriction), are selected for refresh processing. Naturally, modifications having a timestamp *older* than the oldest timestamp of any snapshot defined on the table, can be discarded.

Clearly, the extended mechanism supports independent snapshots, and like with sequential logging, the log has sufficient information to support both requirements for

Time Stamp	Modif. Type	Value		
		Resort	Country	Price Level
3:00	DEL	Beirut	Lebanon	6
NULL	DEL	Brighton	England	8
2:00	UPD	Cannes	France	7
2:00	-	Cannes	France	8
4:00	UPD	Cannes	France	8
4:00	-	Cannes	France	9
3:00	INS	Crete	Greece	4
4:00	UPD	Crete	Greece	4
4:00	-	Crete	Greece	5
NULL	INS	Mallorca	Spain	5

Figure 7. Condensed Log Supporting Multiple Snapshots: The refresh process may have to merge several versions of tuple modifications to obtain the resulting modification.

replicated snapshot. The timestamp associated with each snapshot replica plays the same role as the refresh mark in the sequential log.

The performance of the refresh of a particular snapshot depends on how many refresh cycles that have to be considered for each tuple (cf. Figure 7). This again depends on the number of snapshots defined on the table, and on how widely the refreshes are scattered. The refresh performance can be improved somewhat as can be seen from the example shown in Figure 8.

Assume that the snapshots S_1 and S_2 are defined on the base table. S_1 was refreshed last time at 1:00, whereas S_2 was refreshed at 2:00, 3:00 and 4:00. Since these are the only snapshots defined on the table, *none* of the snapshots on the table has a last refresh time of 2:00 or 3:00. This means that the three sets of entries marked 2:00, 3:00 and 4:00 can be merged tuple-wise into one set marked 4:00. In our example, the Cannes entries are merged to reflect the price level change from 7 to 9. The Crete entries are merged into a single insert entry showing a price level of 5. This figure also serves as an example of the general merging process used for logging new modifications.

In general, it is never necessary to have more log entries per tuple than the number of snapshots defined on a table. The merging of entries must be done tuple-wise, since not all tuples may have been modified in one refresh cycle. This will add to the cost of refresh processing. However, if refreshes are performed as in the example above, possibly S_2 being refreshed more than three times, and S_1 being replicated, significant savings can be obtained for refreshing S_1.

Analysis of The Logging Methods

The cost of the two logging methods is analyzed in the following. The analysis is carried out under various update frequency and for various modification compositions. In the analysis, refresh processing cost is considered as well as message delivery cost.

First we define a modification set as being the modifications made to a base table since last refresh. The modification set consists of inserts, deletes and updates. For a modification set of M modifications, p is the ratio of inserts, q the ratio of deletes, and r the ratio of updates. Naturally, the sum of p, q and r adds up to one.

We assume that the M modifications are uniformly distributed over the tuples of the base table. The base table consists of N tuples prior to any modification.

Time Stamp	Modif. Type	Value		
		Resort	Country	Price Level
4:00	DEL	Beirut	Lebanon	6
NULL	DEL	Brighton	England	8
4:00	UPD	Cannes	France	7
4:00	-	Cannes	France	9
4:00	INS	Crete	Greece	5
NULL	INS	Mallorca	Spain	5

Figure 8. Extended Condensed Logging: Tuples having a timestamp to which no snapshot time is associated are merged "upwards", compressing the log.

The following main classes of base table modifications have been identified:

A **static base table** in which the base table is only updated, i.e. $p=0$, $q=0$, $r=1$.

An **incremental base table** in which the base table is updated and new tuples may be inserted, i.e. $p>0$, $q=0$, $r<1$.

A **dynamic base table** in which the base table is updated, new tuples inserted and tuples deleted, i.e. $p>0$, $q>0$, $r<1$. We assume however that the number of inserts outweighs the number of deletes, i.e. $p>q$.

In the case of a single snapshot, a sequential log will contain M entries assuming that updates are stored as one log entry. The number of log entries in a condensed log is determined by the number of tuples added to the base table, minus the number of deletions of newly inserted tuples (cf. the merge rules given in Figure 5), plus the number of tuples out of the original N that have been updated or deleted since the last refresh which is expressed as the difference between the N and those not changed;

$$L = pM - q(pM/(2N+(p-q)M))M + N(1-u)$$

where u is the probability of not changing an original entry given as:

$$u = \mathbf{PROD}_{i=1}^{M}(p+(q+r)(1-1/(N+(i-1)(p-q))))$$

Given the length of the sequential and condensed log, we may set up expressions for the cost of refreshing database snapshots. A snapshot is defined as a restricted subset of a base table with selectivity s. The cost formulas are given for replicated snapshot refreshed simultaneously. The snapshots will therefore have common modification set. The number of replicas is m.

The cost for sequential logging is given by C_{SL} as an expression of the cost of logging entries, the cost of retrieving qualified entries at refresh, plus the cost of shipping the qualified entries and refreshing them correctly at the snapshot site;

$$C_{\mathrm{SL}} = Mt_{\mathrm{SL}} + (M/b)t_{\mathrm{d}} + ms(1+r(1-s))M(t_{\mathrm{s}}+2t_{\mathrm{d}})$$

in which t_{SL} is the cost for adding entries to the sequential log, b is the average number of entries per disk page, t_{d} is the cost of disk read or write, and t_{s} is the cost of shipping one message to the snapshot site.

The cost for condensed logging is expressed similar to the one for sequential logging, given as C_{CL};

$$C_{\mathrm{CL}} = Mt_{\mathrm{CL}} + (L/b)t_{\mathrm{d}} + ms(Lt_{\mathrm{s}} + min(L,N/b)2t_{\mathrm{d}})$$

in which t_{CL} is the unit cost for adding entries to the condensed log.

The cost of adding an entry to the sequential log, t_{SL}, is one read followed by a write to the log, i.e. $t_{\mathrm{SL}} = 2t_{\mathrm{d}}$. In situations where the base table is updated frequently, the cost can sometimes be reduced (by the effect of having the last log page held in memory). As for condensed logging, the cost of adding an entry to the log depend on the depth of the index tree. For a small log, $t_{\mathrm{CL}} = 3t_{\mathrm{d}}$ is a good estimate, whereas $t_{\mathrm{CL}} = 4t_{\mathrm{d}}$ is used for a larger log. (The depth of index trees can be kept relatively small in MIMER [MIME85]).

Although the normal processing cost is higher for the condensed logging approach, the actual refresh cost is reduced in comparison with the sequential logging approach. The amount of messages sent is reduced, and thus the amount of remote refresh processing. The remote refresh processing in itself is simpler in the condensed logging approach due to the fact that the refresh messages are sent in primary key order.

We have analyzed the logging methods for the classes of base table modifications identified above. In the analysis, we have assumed the message transfer cost to be equal to disk I/O cost, i.e. $t_{\mathrm{s}} = t_{\mathrm{d}}$. In other words, we assume a wide area network and that the shipping cost includes the cost of copying data to and from communication buffers, etc., as well as transmission time. Furthermore, we assume that $b=10$.

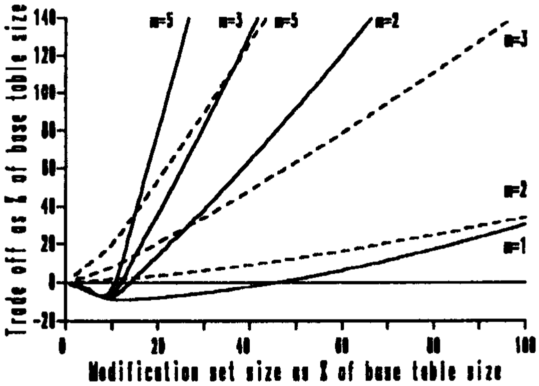

Figure 9. **Static Base Table:** Cost trade off in I/O units as % of base table size. The curves are drawn for different number of replicas and selectivity - $s=1.0$ solid lines, $s=0.25$ dashed lines. Positive values favour condensed logging.

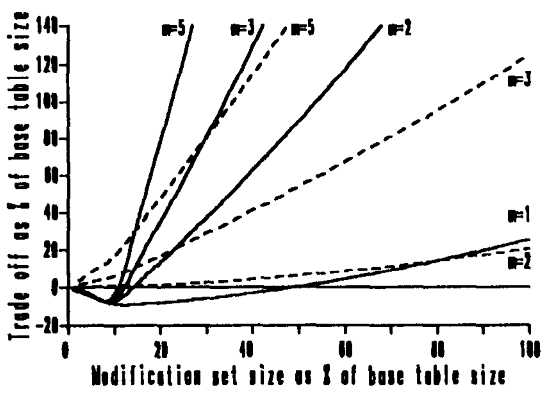

Figure 10. Incremental Base Table (p=0.1, r=0.9): Cost trade off in I/O units as % of base table size. The curves are drawn for different number of replicas and selectivity - *s=1.0* solid lines, *s=0.25* dashed lines. Positive values favour condensed logging.

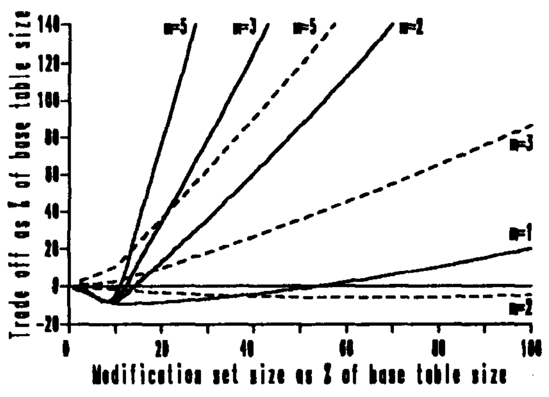

Figure 11. Dynamic Base Table (p=0.2, q=0.1, r=0.7): Cost trade off in I/O units as % of base table size. The curves are drawn for different number of replicas and selectivity - *s=1.0* solid lines, *s=0.25* dashed lines. Positive values favour condensed logging.

The result from the analysis is displayed in Figure 9, Figure 10, and Figure 11. Each figure displays the trade off between disk and message I/O for condensed and sequential logging. Each curve represents the cost difference between sequential and condensed log, i.e. C_{SL}-C_{CL} for different sized modification set given as a percentage of the base table size.

As can be seen from the figures, the condensed log is less costly for fully replicated snapshots when the modification set exceeds 10 to 15 percent of the base table size. For restrictive snapshots, the condensed log is the least costly even for small modification sets as long as base table updates outnumbers base table deletes and inserts, cf. Figure 11.

The sequential log performs well for one-copy snapshots if the modification set is small relative to the base table size. The cost savings are significant for very restrictive one-copy snapshots (not shown on the figures). This is mainly due to less overhead during normal processing.

The results as given in the figures do also apply for independent snapshots when viewed as a batch of *m* snapshots refreshed over a common long refresh cycle.

In the analysis, a uniform distribution of the base table modifications is assumed. In many situations only some of the base table tuples are exposed to modification. The condensed log will therefore become smaller than in the case of a uniform distribution. The condensed logging approach may thus perform best even for relatively small modification sets.

The analysis indicates that the condensed log performs well for replicated snapshots despite the normal processing overhead. In the sequential log approach, the normal processing overhead is kept relatively low. It would therefore be interesting to study a hybrid solution in which the logging is done sequentially and the change history is kept sorted in a condensed log which is maintained by the refresh process.

Conclusion

In this paper, we have discussed two methods to support differential or incremental updating of snapshots based on using a separate table for logging modifications made to the base table.

If a snapshot is replicated to many sites, or if many independent snapshots are defined on a table, then the condensed logging approach is to be preferred.

Since the two methods only support snapshots without joins, a non-incremental method, usually called a full refresh strategy, is used to complete the snapshot mechanism. One can envisage a system supporting all of the methods. The database administrator may then choose to change the refresh mechanism from sequential to condensed logging when most of the modifications refers to only a few tuples. Even the system optimizer may dynamically decide to change method, e.g. if the table is empty, it may decide on using full refresh.

Finally, both logging approaches can be used to support other facilities like instantiated views and deferred update mechanisms in addition to snapshots.

Several different mechanisms supporting loosely concurrent replicated data are planned for implementation in MIMER*. We will for that reason primarily select the logging methods, since they also provide a basis for deferred updates and view instantiation. As the next step, we will look at specific implementations of the various methods in MIMER*.

Acknowledgement

The authors especially want to thank Dick Cooper for the fruitful discussions on snapshots and data replication that lead to this paper, and to Per Hokstad for the help with the statistics applied in cost analysis.

The authors would in addition thank Heidi Berg-Hoff, Henrik Dvergsdal and Svein-Olaf Hvasshovd for helpful criticism, comments and discussions on the subject of this paper.

This research was supported by The Nordic Fund for Technology and Industrial Development (Nordisk Industrifond) under contract P661 DATA.

Bibliography

[ADIB 80] M.E. Adiba and B.G. Lindsay, *Database Snapshots*, Proc. of the 6th International Conference on Very Large Databases, Montreal, Canada (October 1980) pp. 86-91.

[BLAK 86] J.A. Blakely, P-Å. Larson, and F.W. Tompa, *Efficiently Updating Materialized Views*, Proc. of ACM SIGMOD International Conference On Management of Data, Washington DC (May 1986) pp. 61-71.

[FRØS 86] Aa. Frøseth, T. Lien and M. Sæterhaug, *An Interface to an Information Handling Tool*, SINTEF Report STF14 A86007, Trondheim, Norway (January 1986).

[HASK 82] R.L. Haskin and R.A. Lorie, *On Extending the Functions of a Relational Database System*, Proc. of ACM SIGMOD International Conference on Management of Data, Orlando, Florida (June 1982) pp. 207-212.

[LIND 86] B.G. Lindsay et al., *A Snapshot Differential Refresh Algorithm*, Proc. of ACM SIGMOD International Conference on Management of Data, Washington DC (May 1986) pp. 53-60.

[LOHM 85] G.M. Lohman et al.,*Query Processing in R**, In Kim, Reiner, and Batory (editors), Query Processing in Database Systems, Springer Verlag (March 1985, Berlin) pp. 31-47.

[MACK 86] L.F. Mackert and G.M. Lohman, *R* Optimizer Validation and Performance Evaluation for Distributed Queries*, Proc. of the 12th International Conference on Very Large Databases, Kyoto, Japan (September 1986) pp. 149-159.

[MCCO 86] R. McCord and K Ong, *Deferred Copy Specification*, Relational Technology Inc. (May 20, 1986).

[MIME 85] *Database Management and Program Interface MIMER/DB*, MIMER Information Systems AB, (1985, Uppsala, Sweden).

[SELI 79] P.G. Selinger et al., *Access Path Selection in a Relational Database Management System*, Proc. of ACM SIGMOD International Conference on Management of Data, Boston, Massachusettes (May 1979) pp.23-34.