A Snapshot Differential Refresh Algorithm

Bruce Lindsay, Laura Haas, C. Mohan,
Hamid Pirahesh, & Paul Wilms
IBM Almaden Research Center
San Jose, CA 95120-6099

Abstract

This article presents an algorithm to refresh the contents of database snapshots A database *snapshot* is a read-only table whose contents are extracted from other tables in the database The snapshot contents can be periodically *refreshed* to reflect the current state of the database Snapshots are useful in many applications as a cost effective substitute for replicated data in a distributed database system

When the snapshot contents are a simple restriction and projection of a single base table, *differential* refresh techniques can reduce the message and update costs of the snapshot refresh operation The algorithm presented annotates the base table to detect the changes which must be applied to the snapshot table during snapshot refresh The cost of maintaining the base table annotations is minimal and the amount of data transmitted during snapshot refresh is close to optimal in most circumstances

Introduction

A DBMS provides a mechanism for maintaining, accessing, and updating information representing the current state of some real world activity or process Application programs can interrogate the *current* state of the database and modify the database state to reflect changes in the real world system being modeled by the database and its applications However, many database applications need to *freeze* portions of the database state for analysis, planning, or reporting In order to support applications which require a stable version of parts of the database state, the relevant data can be *copied* into separate database entries This allows the database state to continue to evolve to track the real world state while maintaining the copied state for those applications that need it

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery To copy otherwise or to republish, requires a fee and/or specific permission

© 1986 ACM 0-89791-191-1/86/0500/0053 $00 75

The notion of a database *snapshot* was introduced in [ADIBA 80] Database snapshots are periodically refreshed, read-only replicas of selected portions of the database In a relational database, a snapshot is a read-only table whose value is defined by a query over one or more database tables Furthermore, a snapshot can be *refreshed* to cause its value to reflect the *current* (transaction consistent) state of the database tables referenced by the snapshot query The ability to refresh snapshots allows them to be used in place of the base tables from which the snapshot is derived whenever the application does not require access to the current state and does not need to update the current state Snapshots are especially interesting in a distributed database as a cost effective substitute for replicated data *Local* snapshots at several sites can be periodically refreshed from remote base tables This approach allows reads of snapshot data to be local while avoiding the complexity and overhead of maintaining multiple, transaction consistent replicas

Snapshots capture the (transaction consistent) current state of some portion of the database for subsequent processing Once a snapshot has been defined and initialized, its contents can be accessed using ordinary queries Indices can be defined on a snapshot to accelerate access to its contents and snapshots can serve as base tables for other snapshots

Periodically, the snapshot can be refreshed to bring it up to date In general, snapshot refresh requires evaluating the query defining the snapshot and replacing the contents of the snapshot with the results of the query evaluation If the snapshot is derived from a *single* base table, it can be refreshed by detecting the changes to the base table since the last time the snapshot was refreshed and applying only those changes to the snapshot By isolating the changes to the base table since the last refresh of the snapshot, the amount of information transferred to the snapshot during the refresh operation can be reduced if the base table has not changed substantially Also, processing only changes to the base table reduces the number of (recoverable) updates needed to bring the snapshot up to date This article presents an algorithm for determining the base table changes which need to be applied to the snapshot when the snapshot is derived from a single base table When the snapshot is derived from several tables, the snapshot query must, in general, be re-evaluated to determine the new snapshot contents We will first discuss the objectives for a differential snapshot refresh algorithm and consider some alternative refresh methods Then we will present a stepwise development of a differential snapshot refresh algorithm We conclude with an analysis of the effectiveness of the algorithm under various conditions

Snapshot Refresh Objectives

Snapshot refresh should make the snapshot reflect the current, transaction consistent state of the base table First, *all* changes to the base table, which have occured since the last refresh of the snapshot and which which affect the snapshot state must be detected and applied to the snapshot At the same time, we wish to *minimize* the impact of snapshots upon operations on the base table Ideally, base table operations (insert, delete, & update) would be unaffected by the presence of one or more snapshots on the base table In order to provide efficient support for remote snapshots, the refresh algorithm should transmit as little data as possible during the refresh operation

The snapshot refresh algorithm should support *multiple* snapshots on a single base table Each snapshot should be independently refreshable Also, each snapshot should be allowed to specify its own *restrictions* and projections on the base table This allows each (remote) snapshot to extract only needed data from the base table

Alternative Refresh Methods

Several alternatives are available for implementing snapshot refresh The simplest method is to transmit the (restricted & projected) base table to the snapshot each time the snapshot is refreshed The snapshot is first cleared and then the received data is inserted into the snapshot This method has the advantage of minimal impact on normal base table operations Unless a significant portion of the base table has been updated since the last refresh of the snapshot, this simple method will transmit, delete, and insert many unchanged entries One alternative is to transmit changes to the snapshot(s) as they occur at the base table This method, know as ASAP (As Soon As Possible) update propagation has several drawbacks Since the snapshot is, more or less, continuously being updated, it no longer captures the base table state as of a specific refresh time More seriously, if the snapshot is remote from the base table and communication between the base table and the snapshot is interrupted, the base table changes must be buffered or rejected Transmitting each base table change to the snapshot ASAP will increase base table update costs due to the cost of the communication and snapshot update associated with each ASAP base table update

Another alternative is to buffer the changes to the base table and transmit relevant buffered changes whenever a snapshot demands to be refreshed This method creates the need to buffer changes until all snapshots on a given base table have been refreshed Multiple changes to the same base table entry will be buffered and transmitted separately Of course, one could bound the buffering required and transmit the entire (restricted) base table if the last refresh of the snapshot precedes the earliest retained changes Operations on the base table might be unaffected if the database recovery log is used as the change buffer Otherwise, considerable space and time overhead will be required to recoverably buffer changes to the base table If the recovery log is used to buffer the information needed for snapshot refresh, considerable effort will be needed to cull the relevant, *committed* data from the log Only a small portion of the log will involve updates to the base table for a particular snapshot Unless the values of unchanged base table fields are written to the log, an access to the base table is required to determine whether the updated entry qualifies (or qualified) for the snapshot and to obtain the values for the all the snapshot fields Background processing of the recovery log can support snapshot refresh "as of" a given time Refresh to the current base table state requires that the log filter "catch up" and that uncommitted changes be handled very carefully

Instead of ASAP refresh or base table change buffering, one can *annotate* the base table to identify changed entries By judiciously adding information about changes to the base table, the amount of space used to support snapshot refresh is bounded at the cost of extra work to maintain the annotations As we shall see, both the space used and the extra work done can be made to be quite small In addition, multiple snapshots on a single base table do not require additional annotations and much of the extra work is amortized over the set of snapshots depending upon the base table

Differential Refresh: A Simple Solution

We first present a simple, but impractical, algorithm for differential snapshot refresh This simple algorithm will then be incrementally modified to produce more satisfactory algorithms The simple algorithm assumes that the entries of the base table are embedded in a *dense, ordered* space It is useful to think of the space as an address space in which each element either contains a base table entry or is marked as empty In addition, each element of the base table address space is extended to contain a *timestamp* field which records the time at which the address space element was last modified The time stored in the **TimeStamp** field is assumed to be any local, monotonically increasing value For example, the local standard time, or a local, recoverable counter could serve as the time base for the differential refresh algorithm

The snapshot table itself is stored more traditionally The entries in the snapshot table are extended to include a field (**BaseAddr**) containing the *address* of the corresponding entry in the base table Associated with the snapshot table is the (base table) time at which the snapshot was last refreshed (**SnapTime**) In addition, the base table restriction (**SnapRestrict**) which defines the contents of the snapshot is associated with the snapshot instance (We shall ignore base table projections to simplify the presentation)

The simple differential refresh algorithm is initiated by sending the last snapshot refresh time (**SnapTime**) and snapshot restriction (**SnapRestrict**) to the base table Each element of the base table address space is then examined If the **TimeStamp** of the element is greater than **SnapTime**, the element must be transmitted to the snapshot table If the element is empty, or if its value does not satisfy **SnapRestrict**, only the element address and "empty" status are transmitted to the snapshot Otherwise, the base table address, status, *and* value are sent to the snapshot After scanning the base table, and transmitting changed elements, the current (base table) time is sent to the snapshot to become the new **SnapTime** of the snapshot Figure 1 and Figure 2 illustrate the representation of the base table and the snapshot The figures also depict refreshing the

Simple Base Table

Addr	Status	Time Stamp	Value Name	Value Salary
1	ok	3 00	Bruce	15
2	ok	3 45	Laura	6
3	ok	3 50	Hamid	15
4	empty	4 00	-	-
5	ok	2 30	Mohan	9
6	ok	2 00	Paul	8
7	empty	4 10	-	-

Refresh Messages to Snapshot Table

SnapTime = 3 30 **BaseTime** = 4 30
SnapRestrict = Salary < 10

Base Addr	Status	Value Name	Value Salary
2	ok	Laura	6
3	empty	-	-
4	empty	-	-
7	empty	-	-

Figure 1 Simple Base Table and Refresh Messages

snapshot, giving the messages and the before and after images of the snapshot

At the snapshot, each transmitted element is received and processed If the element status is "empty", the snapshot entry with the matching **BaseAddr** is deleted from the snapshot table (if such an element exists) Non-empty elements cause an update if a matching **BaseAddr** is found Otherwise, a new entry, with the indicated **BaseAddr** is inserted into the snapshot Clearly, a snapshot index on **BaseAddr** will accelerate snapshot refresh processing

This simple refresh algorithm clearly detects all changes to the base table and faithfully informs the snapshot When the snapshot restriction reduces the base table, the algorithm sends superfluous entries to the snapshot When a base table entry which does not satisfy the snapshot restriction is deleted, inserted, or updated, the entry's address and status is transmitted to the snapshot This is because modified base table entries which do not currently satisfy the snapshot restriction may have satisfied the restriction before their modification (E g , Hamid has had a raise in the example)

Differential Refresh Empty Regions

The simple refresh algorithm requires an impractical representation for the base table Entry addresses are usually the offset of

Snapshot Table before Refresh

SnapTime = 3 30
SnapRestrict = Salary < 10

Base Addr	Value Name	Value Salary
3	Hamid	9
4	Jack	6
5	Mohan	9
6	Paul	8
7	Bob	7

Refresh Messages to Snapshot Table

SnapTime = 3 30 **BaseTime** = 4 30
SnapRestrict = Salary < 10

Base Addr	Status	Value Name	Value Salary
2	ok	Laura	6
3	empty	-	-
4	empty	-	-
7	empty	-	-

Snapshot Table after Refresh

SnapTime = 4 30
SnapRestrict = Salary < 10

Base Addr	Value Name	Value Salary
2	Laura	6
5	Mohan	9
6	Paul	8

Figure 2 Simple Snapshot Refresh - Before and After

the entry within a file or a page number and index of the entry within the page Thus, not all addresses have entries and maintaining a status for every possible address is not feasible for most database storage systems If we assume that the database system *does* assign some sort of address for every actual entry in a table, and that the addresses are totally ordered, then it is possible to maintain summary information about which addresses are not in use For each unused address region we can store its limits and the time at which the region was created or changed size As before, actual base table entries contain a **TimeStamp** field to record the time of their last modification

Maintaining the information about which regions of the base table address space are empty will require extra work when base table entries are inserted or deleted In the simple algorithm only the **TimeStamp** and status of the modified base table entry needed to be updated Now, when an entry is inserted or deleted from the base table, empty regions must be split or coalesced and the empty region timestamp must be set

The refresh algorithm for empty regions is very similar to the simple refresh algorithm The timestamps on empty regions and base table entries are compared to the **SnapTime** If an empty region has a high timestamp, its boundary addresses are transmitted to the snapshot and all snapshot entries with **BaseAddr** in the empty region are deleted from the snapshot Updated and inserted base table entries are handled as in the simple refresh algorithm If each empty region specification is stored in an "empty" address of the region, then empty regions and actual entries can be processed in base table address order This allows empty regions which are separated by entries which do not satisfy the snapshot restriction to be combined before transmitting the empty region to the snapshot (Of course, the combined empty region is not transmitted unless one of the empty regions, or one of the intervening unqualified entries has a timestamp greater than **SnapTime**)

The lumping together of empty entries reduces the number of items transmitted to the snapshot because multiple deletions may create a single empty region Also, by merging empty regions separated by unqualified entries, a single empty region transmission "covers" all the base table updates in the combined region

Associating Empty Regions with Actual Entries

The next step in the development of the snapshot differential refresh algorithm is to associate the empty region information with the base table entry which follows the empty region We shall add a field (**PrevAddr**), containing the address of the preceding base table entry, to all base table entries All addresses between an entry and the **PrevAddr** of the entry are empty

When modifying the base table, extra effort will be required when entries are deleted or inserted When an entry is deleted, the **PrevAddr** *and* **TimeStamp** fields of the succeeding base table entry must be updated with the **PrevAddr** from the deleted entry and the current time When an entry is inserted, the **PrevAddr** of the new entry must be set to the value of the **PrevAddr** from the next entry in the base table, *and* the **PrevAddr** in the next entry must be set to the address of the new entry The **TimeStamp** of the new entry is set to the current time, but the **TimeStamp** of the next entry does not need to be updated Care must be taken to avoid anomalies when concurrent updates to the base table are allowed Concurrent inserts or deletes at neighboring addresses will require updates to the same successor record and these updates must be synchronized carefully with the insert or delete

The snapshot refresh algorithm remains similar to the preceding version of the algorithm The principal difference is that a high **TimeStamp** on a base table entry indicates that the preceding empty region has grown *or* that the entry has been inserted or updated, *or* both As before, empty regions, separated by entries which do not satisfy the snapshot restriction can be combined When transmitting an entry to the snapshot we will transmit the address of the preceding qualified entry and the value of the entry Figure 3 is a pseudo code representation of the base table refresh algorithm Figure 4 depicts the snapshot table side of the refresh algorithm

```
BaseRefresh( BaseTable, SnapTime, SnapRestrict )
  LastQual = 0,     /* Addr last qualified */
  Deletion = False, /* Deletions detected? */
  /* Scan BaseTable in address order */
  forever do,
    /* Get next BaseTable entry */
    <Address, PrevAddr, TimeStamp, Value> = Next( BaseTable )
          Next( BaseTable),
    if (End_of_Scan) then
      break,
    if (SnapRestrict( Value )) then do,
      /* Qualified BaseTable entry */
      if (TimeStamp > SnapTime) | (Deletion) then
        /* Updated or preceding deletions */
        Xmit( Address, LastQual, Value ),
      LastQual = Address,
      Deletion = False,
      end, /* Qualified BaseTable entry */
    else do,
      /* Unqualified BaseTable entry */
      if (TimeStamp > Snaptime) then
        /* Updated entry ==> may have */
        /* qualified before update    */
        Deletion = True,
      end, /* Unqualified BaseTable entry */
  end, /* of BaseTable scan */
  /* Handle deletions at end of BaseTable */
  Xmit( NULL, LastQual, NULL ),
  /* Transmit new SnapTime */
  Xmit( current_time ),
end, /* of Base Refresh */
```

Figure 3 Base Table Refresh for Empties Associated with Entries

```
SnapRefresh ()
  Send( SnapTime, SnapRestrict) to BaseTable,
  <Address, PrevAddr, Value> = Receive_from_BaseTable,
  while (Address ≠ NULL) do,
    /* Delete empty region from snapshot */
    DELETE FROM Snapshot WHERE
      (BaseAddr > PrevAddr) AND
      (BaseAddr ≤ Address),
    /* Insert entry into snapshot */
    INSERT INTO Snapshot
      <Address, Value>,
    <Address, PrevAddr, Value> = Receive_from_BaseTable,
  end, /* of while more entries */
  /* Delete at end of snapshot */
  DELETE FROM Snapshot WHERE
    BaseAddr > PrevAddr,
  /* Get new SnapTime */
  SnapTime = Receive_from_BaseTable,
end, /* of SnapRefresh */
```

Figure 4 Snapshot Table Refresh for Empties with Entries

Base Table before Refresh

Addr	Prev Addr	Time Stamp	Value		Comment
			Name	Salary	
1	0	3 00	Bruce	15	unchanged
2	NULL	NULL	Laura	6	inserted
3	1	NULL	Hamid	15	updated - was 9
4	~~3~~	~~2 30~~	~~Jack~~	~~6~~	deleted
5	4	2 30	Mohan	9	preceeding delete
6	5	2 00	Paul	8	unchanged
7	~~6~~	~~1 00~~	~~Bob~~	~~8~~	deleted

Refresh Messages to Snapshot Table

SnapTime = 3 30 **BaseTime** = 4 30
SnapRestrict = Salary < 10

Base Addr	Prev Addr	Value	
		Name	Salary
2	0	Laura	6
5	2	Mohan	9
NULL	6	NULL	NULL

Base Table after Refresh

Addr	Prev Addr	Time Stamp	Value		Comment
			Name	Salary	
1	0	3 00	Bruce	15	unchanged
2	1	4 30	Laura	6	inserted
3	2	4 30	Hamid	15	updated
5	3	4 30	Mohan	9	preceeding delete
6	5	2 00	Paul	8	unchanged

Figure 5 Base Table Fix up - Before & After

Batch Maintenance of Empty Regions and Timestamps

The current version of the refresh algorithm has a serious impact on operations which insert or delete from the base table Is it possible to reduce the impact on base table operations by postponing the maintenance of the **PrevAddr** and **TimeStamp** fields until a snapshot must be refreshed? The answer is "Yes, but at the cost of extra complexity and overhead during snapshot refresh" However, it is the snapshot refresh operations which *should* bear the costs associated with maintaining the snapshot First we will specify how base table operations manage the extra fields Then we will discuss how the empty region and timestamp fields can be updated to allow refresh to detect the changes which need to be sent to the snapshot

Let us assume that the DBMS supports the notion of NULL fields in table entries Delete operations on the base table will be unaffected by the snapshots - the base table entry is simply deleted Insert operations will set the **PrevAddr** and **TimeStamp** fields to NULL and insert the entry into some empty address of the base table Update operations will simply set the **TimeStamp** field to NULL This approach does not require multiple entry updates during operations on the base table and has little effect upon the performance and complexity of the base table operations In particular, the synchronization problems mentioned in the previous section do not arise

Given that base table operations will *not* maintain the information to track update times or empty region boundaries, we must define an algorithm which will restore the **PrevAddr** and **TimeStamp** fields to the values needed to support the previously presented refresh algorithm Given such an algorithm, we can then run the previous version of the refresh algorithm to isolate the changes

Snapshot Table before Refresh

SnapTime = 3 30 **SnapRestrict** = Salary < 10		
Base Addr	**Value**	
	Name	**Salary**
3	Hamid	9
4	Jack	6
5	Mohan	9
6	Paul	8
7	Bob	8

Refresh Messages to Snapshot Table

SnapTime = 3 30 **BaseTime** = 4 30 **SnapRestrict** = Salary < 10			
Base Addr	**Prev Addr**	**Value**	
		Name	**Salary**
2	0	Laura	6
5	2	Mohan	9
NULL	6	NULL	NULL

Snapshot Table after Refresh

SnapTime = 4 30 **SnapRestrict** = Salary < 10		
Base Addr	**Value**	
	Name	**Salary**
2	Laura	6
5	Mohan	9
6	Paul	8

Figure 6 Snapshot Refresh - Before and After

which will be sent to the snapshot Of course, we would like to be able to perform both functions (fix up the base table extra fields and transmit changes) in a single pass over the base table Before combining the algorithms, let us first examine how the extra fields can be updated to reflect the current state of the base table

The algorithm to fix up the extra fields will scan the base table in address order Because only snapshot refresh events need to occur at distinct times, we can use the current (base table) time to update the **TimeStamp** field during the fix up process In order to have a transaction consistent view of the base table during the fix up process, we must obtain a table level lock on the base table during the fix up (and refresh) procedures

As the base table is scanned, we must detect and reflect all inserts, updates, and deletes in the base table since the last time the fix up algorithm was run An entry with a NULL **PrevAddr** was *inserted* since the last time the fix up algorithm was run The **TimeStamp** of the inserted entry should be set to the current time and the **PrevAddr** should be set to the address of the previous entry in the base table An entry with a non-NULL **PrevAddr** and a NULL **TimeStamp** was *updated* since the last time the fix up algorithm was run The **TimeStamp** of the updated entry should be set to the current time

Detecting *deleted* entries is somewhat more complex If a non-NULL **PrevAddr** is not equal to the address of the last *non-newly-inserted* entry that was encountered, then one or more entries were deleted between the current entry and the last non-inserted entry Both the **PrevAddr** *and* the **TimeStamp** of the current entry must be updated If the **PrevAddr** is equal to the last non-newly-inserted entry encountered, but is not equal to the address of the previous (newly inserted) entry, only the **PrevAddr** of the current entry needs to be updated The notion of detecting deletions from the base table by detecting anomalies in the empty region information in the **PrevAddr** fields is central to the differential refresh algorithm Figure 5 and Figure 6 depict the representation of the base and snapshot tables and present an example of the base table fix up and snapshot refresh operations Figure 7 is a pseudo code representation of the base table fix up algorithm

The final step in our development of the differential refresh algorithm is to combine the fix up algorithm with the refresh algorithm of the preceding section The combination is straightforward For each base table entry, we first update the extra fields, if needed Then, if necessary, the entry is transmitted to the snapshot It is possible to further optimize the basic differential refresh algorithm The reader is invited to discover improvements which reduce the message traffic and the number of updates to the base table during the fix up phase of the algorithm

```
BaseFixup( BaseTable )
  ExpectPrev = 0       /* Expected PrevAddr          */
  LastAddr   = 0,      /* Last Address in BaseTable  */
  FixupTime  = Now,    /* New value for TimeStamp    */
  /* Scan BaseTable in address order */
  forever do,
    /* Get next BaseTable entry */
    <Address, PrevAddr, TimeStamp> = Next( BaseTable ),
    if (End_of_Scan) then
      break,
    if (PrevAddr = NULL) then
      /* Inserted BaseTable entry */
      UPDATE CURRENT (PrevAddr = LastAddr, TimeStamp = FixupTime),
    else do,
      /* non-inserted entry */
      if (TimeStamp = NULL) then
        /* Updated BaseTable entry */
        UPDATE CURRENT (TimeStamp = FixupTime),
      if (PrevAddr ≠ ExpectPrev) then
        /* Deleted entry(s) preceding current entry */
        UPDATE CURRENT (PrevAddr = LastAddr,
                        TimeStamp = FixupTime),
      else
        if (PrevAddr ≠ LastAddr) then
          /* Entries inserted before current entry */
          UPDATE CURRENT (PrevAddr = LastAddr),
      ExpectPrev = Address,
      end, /* non-inserted entry */
    LastAddr = Address,
  end, /* of BaseTable scan */
end, /* of Base Fixup */
```

Figure 7 Base Table Fix Up Algorithm

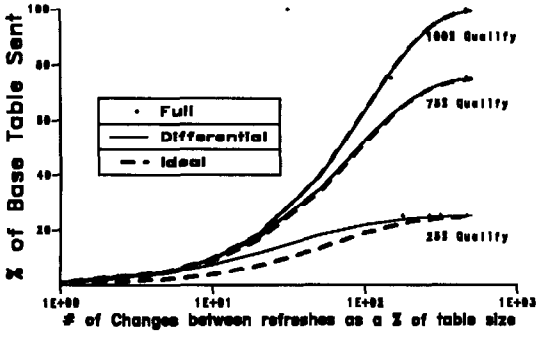

Figure 8 Comparison of % of tuples of the Base table that need to be sent for the ideal, differential, and full Refresh algorithms The curves are drawn for different % of tuples qualifying for snapshot

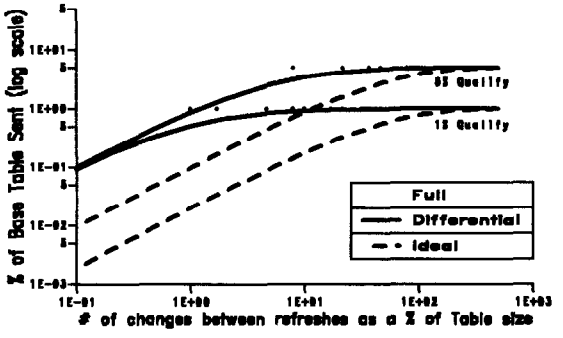

Figure 9 Shows the part of Figure 8 for 1% and 5% of tuples qualifying for snapshot Note that the vertical axis is in logorithmic scale

Analysis of Differential Refresh Algorithm

The differential snapshot refresh algorithm is designed to reduce the cost of maintaining (remote) snapshots The differential refresh algorithm exchanges communication and snapshot update overhead for base table accesses and updates during refresh When an efficient method for applying the snapshot restriction is available (e g , an index), the base table sequential scan may be more costly than simply re-populating the snapshot by executing the snapshot query The expected costs of differential refresh and full refresh can be computed when the snapshot is defined and the appropriate refresh method can be selected

How effective is the differential refresh method in reducing the communication and update overhead of snapshot refresh? As we noted earlier, the notion of empty regions and combining empty region information with the actual entries of the base table causes superfluous messages to the snapshot when unqualified entries are inserted, deleted, or updated How important is this effect? Intuitively, we can see that as the snapshot qualification becomes more restrictive, the "distance" between qualified entries will become larger Any insert, delete, or update between two qualified entries causes the second entry to be transmitted Therefore, when qualified entries are widely separated, it is more likely that base table modifications in the interval will cause an unnecessary message to be transmitted to the snapshot

In order to quantify the performance of the differential refresh algorithm, we will compare it to an *ideal* refresh algorithm and to the *full* refresh method The *ideal* algorithm transmits only actual base table changes to the (restricted) snapshot and only the most recent change to each entry (since refresh) The ideal algorithm uses old and new values of changed entries to insure that changes to unqualified entries are not transmitted The *full* refresh method simply transfers all qualified entries to the snapshot where the received entries replace the previous contents of the snapshot

Two parameters affect the performance of the refresh algorithm the amount of update activity on the base table since the last refresh, and the degree to which the base table is restricted by the snapshot When there is no restriction, the differential refresh algorithm performs as well as the ideal refresh and is superior to full refresh until the entire base table has been updated

As the snapshot qualification becomes more restrictive, the relative number of superfluous messages for the differential refresh algorithm increases For a given restriction, the percentage of superfluous messages decreases as the number of base table modifications increases For restricted snapshots, if few base table modifications occur between refreshes, few messages are sent Therefore, we see that the differential refresh algorithm is robust in the sense that it is most precise when much data needs to be transmitted to the snapshot When little data needs to be transmitted, it is less accurate in transmitting only necessary information

We have compared the number of messages sent as the restriction and the amount of base table update activity are varied Both simulation and analysis show that the above hypothesis is true Figure 8 and Figure 9 show the number of messages, as a percentage of the base table size, which are sent by the ideal algorithm, the full refresh method, and the differential refresh algorithm The figures give the message traffic as a function of the update activity between refreshes for different snapshot restrictions Figure 8 shows the message traffic when the snapshot includes more than 25% of the base table Figure 9 shows, on an expanded scale, the message traffic for more restrictive snapshots

Conclusions

The differential snapshot refresh algorithm has been implemented as part of the R* experimental, distributed database management system [HAAS82] R* supports general snapshots in that any query can be used to define the contents of a snapshot When the snapshot is defined, an analysis of the query determines whether the differential refresh algorithm or full refresh is to be used to refresh the snapshot

R* supports *query compilation* [LOHMAN85] to allow efficient execution of queries which are executed repeatedly (like snapshot refresh) The query compilation process creates an efficiently executable representation of the query which is stored in the database, to be loaded and executed when the query is activated During query compilation, the query optimizer selects an execution strategy for the query and creates compile time bindings to the objects and access paths to be used during execution Separation of query compilation from query execution in R* amortizes the compilation and binding cost over multiple executions of the query

The R* implementation for snapshots takes advantage of the compilation facility to compile the snapshot refresh operations, for both the full refresh and differential refresh methods It was not totally straightforward to exploit compilation for snapshots because the compilation must be done during the execution of the CREATE SNAPSHOT statement and the execution is in response to a REFRESH SNAPSHOT statement Considerable cleanup of internal interfaces was necessary to permit the "recursive" activations of compiler and execution facilities for snapshots

Compilation of the differential refresh algorithm is complicated by the fact that the algorithm cannot be reduced to a standard query statement due to the fact that the algorithm uses entry addresses which are not available at the query language level Special runtime routines were needed to implement the differential refresh algorithm On the other hand, the normal distributed query execution facilities in R* *block* the entries to be transmitted and the execution of both the full and differential refresh methods take advantage of the blocking to reduce the cost of the refresh operation

The differential refresh algorithm also requires extra fields in the base table In the R* implementation, the extra fields are added automatically to the base table when the first snapshot using differential refresh is created Fortunately, R* already had support for adding fields to an existing table without accessing all the entries of the table The extra fields are given "funny" names to distinguish them from user defined fields while allowing them to be recorded in the system catalogs (schema) Detecting the presence of the "funny" named fields allows the system to compile the extra code to manage the fields when entries in the base table are updated No special efforts were needed to handle deletions and insertions to the base table Deletions just delete the entry Insertions, by omitting values for the extra fields, cause them to be set to NULL, just as for NULLable user fields

The implementation of snapshots in R* was somewhat difficult A complex modification, to an already quite complex system, was necessary Paul Wilms, Bruce Lindsay, and Dean Daniels implemented the high level controls for snapshot creation and refresh Laura Haas implemented the changes to the compiler to support snapshots and C Mohan implemented the runtime support Hamid Pirahesh provided the performance analysis of the differential refresh algorithm Dale Skeen has given helpful advice on how to present the differential refresh algorithm The efforts of all who contributed to the development of the differential refresh algorithm, its implementation, and its presentation are gratefully acknowledged

Bibliography

[ADIBA 80] M E Adiba and B G Lindsay, *Database Snapshots,* **Proceedings 6th International Conference on Very Large Data Bases,** Montreal, Canada (October 1980) pp 86-91

[HAAS 82] L M Haas, Pl Selinger, E Bertino, D Daniels, B Lindsay, G Lohman, Y Masunaga, C Mohan, P Ng, P Wilms, and R Yost, *R* A Research Project on Distributed Relational DBMS,* **IEEE Database Engineering, Vol 5,** No 4 (also available as IBM Research Report RJ3653, October 1982) (December 1982) pp 28-32

[LOHMAN 85] G Lohman, C Mohan, L Haas, D Daniels, B Lindsay, P Selinger, and P Wilms, *Query Processing in R*,* in **Query Processing in Database Systems,** W Kim, D Reiner, and D Batory (Eds), Springer-Verlag, 1985 (also available as IBM Research Report RJ4272, April 1984)