A database snapshot captures a consistent database state as of a given time. The contents of a snapshot table 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. The thesis proposes two methods based on using a separate table for logging modifications made to a base table; a sequential and a condensed logging approach. The logging approach is compared with the extended base table approach used in the experimental System R*. Two enhancements to the differential refresh algorithm in R* are proposed, and alternative refresh strategies and realizations of the logging approach are discussed.
DATABASE SNAPSHOTS: A MECHANISM FOR REPLICATION OF DATA IN DISTRIBUTED DATABASES

December 20th, 1987

Oddvar Risnes

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

Kandidat: Oddvar Risnes
Institutt: Institutt for datateknikk og telematikk

AVHANDLING

Tittel på avhandlingen: DATABASE SNAPSHOTS: A MECHANISM FOR REPPLICATION OF DATA IN DISTRIBUTED DATABASES

SENSURKOMITEENS VURDERING

Avhandlingen er bedømt og godkjent for graden doktor ingeniør

Trondheim, den

Prof. P. M. G. Apers

Prof. Willy Jensen

Prof. Kjetil Benteland

underskrift

underskrift

underskrift
To my wife, Fiona
and my children, Evelyn, Chris, and Anne Kristin
who I love more than anything else in this world.
Preface

Some while ago my children were given a game called snap. It appeared to be an English card game that can be played by any number of players — at least two. The goal of the game is to be very fast at things — otherwise you have no chance at staying in the game. Each player, in turn, places a card, faced upwards, onto a pile that is gradually being built up. When, or if, the same card face is repeated in sequence, the first one to grab the pile saying snap wins the pile. The winner of the game is the one that eventually gets hold of all the cards. Therefore, be prepared to move like a shot. The game may be generalized to include any number of card decks. This makes the game more exciting, but requires more skill. Efficiency is, one may say, the key to success.

Snapshots described in this thesis have some striking similarities with the situation described above. The card deck is replaced with a database, and the game can be generalized to include several databases. Several users may take part in the game, and they all demand for efficiency and cry out if such is not supported.

The common understanding of a snapshot is a picture that captures a situation at a particular time. It therefore makes sense to apply snapshots to databases — to capture information describing a state as of a given time. This thesis discusses snapshots within this context.

It is common practice to include acknowledgements in a document of this kind. I have been fortunate to have received assistance from many sources, and it is a great pleasure to express gratitude for the help I have received in writing this thesis.

Most of all, I wish to thank my colleague, Bo Kähler, for his guidance and his valuable insight into all areas of database research. Many of the ideas presented in my thesis, relate to our many discussions on database snapshots, and distributed databases in general. I have also been fortunate enough to work closely with him on the distributed prototype MIMER*.

A special thanks goes to another of my colleagues, Svein-Olav Hvasshovd, for helpful criticism and comments throughout the writing of this thesis, and for his concern in the completion of my work. I am very grateful to my advisor, Professor Kjell Bratbergsgen, for bearing with me throughout my efforts in completing this research. Also, I wish to thank Per Hokstad at SINTEF for the help I received with the statistics on the first version of the cost model for the condensed log. My thanks also goes to all members of the Database Research Group at RUNIT, and my employer RUNIT, for sponsoring my research over the last months — directly and indirectly.

It would be remiss not to acknowledge the support that provided the stepping stone for my research. First of all, I express my gratitude to The Norwegian Research Council for
Science and the Humanities (NAVF) that sponsored the first part of my study. A special thanks goes to Professor Peter Stocker at the University of East Anglia for taking me on his research team for a year, and providing me with insight in distributed databases. Also, it would be very unfair to leave MIMER Information Systems AB in Sweden unmentioned. After all, the MIMER* prototype has provided me with the most detailed insight in solutions for distributed database systems — this work was partially funded by The Nordic Fund for Technology and Industrial Development.

My final thanks go to the library automation group at the University of Trondheim (BIBSYS) for allowing me to use their computer facilities during the final stages of my work.
Abstract

A database snapshot is a read-only table in which the data is extracted from other tables in the database. The snapshot mechanism provides for replication of data in general. In a distributed database, database snapshots can be used to provide for locality of data — thus reducing the need for costly, remote data access. A snapshot captures a consistent database state as of a given time, and relaxes the requirement of being up-to-date. The mechanism is not loaded with the maintenance overhead which is normally associated with replicated data, and therefore represents a cost effective substitute for replicated data in a distributed database.

The content 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, relevant to the snapshot, are detected and shipped to the snapshot table site for refresh. This allows for an incremental update of snapshots — the motivation behind the research in this thesis.

The thesis proposes two methods based on using a separate table for logging modifications made to a base table; a sequential and a condensed logging approach. The logging approach is compared with the extended base table approach used in the experimental system R*. The R* approach annotates the base table to detect the changes to be applied in snapshot refresh. In this thesis, two enhancements to the differential refresh algorithm in R* are proposed. The thesis discusses alternative refresh strategies and realizations of the logging approach. Two complementary refresh strategies are proposed for the sequential log, and two different realizations of the condensed log are described.

In addition to the differential refresh issue, the thesis defines statements for definition and manipulation of database snapshots in a distributed environment. The statements allow for snapshots to be defined independent of each other, and for snapshot replication at different sites, in which the replicas are refreshed collectively. This poses an additional requirement to the differential refresh approach, and the thesis discusses how differential refresh strategies can be extended to support snapshot replication in general.

An analytic model, describing the logging and the R* approach, is developed. The model is applied in a comparative cost analysis of differential refresh strategies. The analysis addresses processing and communication cost introduced at snapshot refresh, plus the overhead imposed on the normal update activity of base tables.

The conclusion of the thesis is two-fold. A sequential logging approach — supported with the two complementary differential refresh strategies — reduces the cost of snapshot refresh, when compared with a full refresh strategy, for a wide range of snapshots and modification cases. The overall cost of snapshot maintenance using a sequential ap-
proach, is far better than the R* approach when a small portion of the base table tuples have been changed. Besides, the sequential approach integrates snapshot replication most cost effectively, and may be used in support of complex snapshot definitions. However, the R* approach is a good alternative to sequential logging for supporting differential refresh of snapshots based on a single base table, provided the snapshots are not very restrictive.
Table of Contents

Chapter 1. Distributed Databases and Snapshots .............................................. 1
  1.1 Distributed Databases ........................................................................ 1
  1.2 Snapshots — an Alternative to DQP .................................................. 4
      1.2.1 The Introduction of Snapshots ................................................ 5
      1.2.2 Snapshots in Distributed Environment ..................................... 6
      1.2.3 Design Objectives .................................................................. 7
  1.3 Related Work ....................................................................................... 8
  1.4 Plan of Thesis ...................................................................................... 9

Chapter 2. Snapshot Definition and Semantics ..................................................... 11
  2.1 Introduction ......................................................................................... 11
  2.2 Snapshot Definition ........................................................................... 12
  2.3 Snapshot Refresh ............................................................................... 14
  2.4 DROP and ALTER Statements .......................................................... 16
  2.5 Replicated and Independent Snapshots .............................................. 17
  2.6 Summary ......................................................................................... 22

Chapter 3. Implementing Snapshots ................................................................. 25
  3.1 Introduction ......................................................................................... 25
  3.2 The Refresh Operation ....................................................................... 27
  3.3 Performance Objectives .................................................................... 29
  3.4 Full Refresh ....................................................................................... 31
  3.5 Differential Refresh .......................................................................... 32
  3.6 Implementing a Spectrum of Methods ............................................... 35
  3.7 Operational Issues ........................................................................... 36
      3.7.1 Data Availability ................................................................... 36
      3.7.2 Recovery ................................................................................ 37
      3.7.3 Refresh Initiation ................................................................... 37

Chapter 4. Differential Refresh Methods .......................................................... 39
  4.1 Introduction ......................................................................................... 39
  4.2 Sequential Logging ............................................................................ 42
      4.2.1 Basic Refresh Strategy .............................................................. 43
      4.2.2 Sort Merge Refresh ................................................................ 49
  4.3 Condensed Logging .......................................................................... 50
      4.3.1 Condensed Log Based on B-tree ............................................. 50
      4.3.2 Condensed Log Based on Linear Hashing ............................... 53
  4.4 The R* Method ................................................................................. 54
  4.5 Enhancements to R* .......................................................................... 62
      4.5.1 Enhancement One ................................................................... 62
      4.5.2 Enhancement Two ................................................................... 63
4.6 Enhancement to Logging ........................................... 65
4.7 Summary and Concluding Remarks .............................. 68

Chapter 5. Refresh Method Analysis ................................. 71
5.1 Introduction ............................................................ 71
5.2 Definitions ............................................................ 72
5.3 Analytic Model of Logging Methods ......................... 73
  5.3.1 Sequential Log ............................................... 73
  5.3.2 Condensed Log ............................................... 74
  5.3.3 Enhanced Log ............................................... 78
  5.3.4 Summary ..................................................... 79
5.4 Analytic Model of R* Methods ................................. 83
  5.4.1 Basic R* Algorithm ......................................... 83
  5.4.2 Enhancement One ............................................. 85
  5.4.3 Enhancement Two ............................................. 85
  5.4.4 Summary ..................................................... 86
5.5 Estimating Overall Cost of Logging Methods .............. 90
  5.5.1 Sequential Log ............................................... 91
  5.5.2 Sort Merge of Sequential Log ............................ 92
  5.5.3 Condensed Log Using B-Trees ............................ 93
  5.5.4 Condensed Log Using Linear Hashing ................... 95
  5.5.5 Enhanced Log ................................................ 95
  5.5.6 Summary ..................................................... 97
5.6 Estimating Overall Cost of R* Methods ...................... 97
  5.6.1 Basic R* Algorithm ......................................... 98
  5.6.2 Enhancement One ............................................. 98
  5.6.3 Enhancement Two ............................................. 98
  5.6.4 Summary ..................................................... 99
5.7 Analytic Results .................................................. 99
  5.7.1 Static Base Table ........................................... 100
  5.7.2 Incremental Base Table .................................... 104
  5.7.3 Dynamic Base Table ........................................ 108
5.8 Evaluation of Results and Concluding Remarks ........... 108

Chapter 6. Extending Refresh for Snapshot Replication .......... 113
6.1 Introduction .......................................................... 113
6.2 Extended Sequential Logging ................................... 115
6.3 Extended Condensed Logging ................................... 119
6.4 Impact on Enhancement to Logging ......................... 129
6.5 Extended R* Method .............................................. 129
6.6 Analysis .............................................................. 134
  6.6.1 Replicated Snapshots ...................................... 135
  6.6.2 Independent Snapshots .................................... 137
  6.6.3 Analytic Results ............................................ 139
6.7 Summary and Concluding Remarks ............................. 149

Chapter 7. Conclusion .................................................. 151

Bibliography ............................................................ 157

x Database Snapshots
List of Illustrations

Figure 1. Example Base Table and Snapshot Table ........................................... 14
Figure 2. Replicated Snapshot ................................................................. 19
Figure 3. Snapshot Definition and Manipulation Statements ....................... 22
Figure 4. Refresh Operation in Distributed Environment ......................... 28
Figure 5. Impact of Database Snapshot Mechanism .................................. 30
Figure 6. Example Base Table and Modification Set .................................. 40
Figure 7. Sequential After Image Log ....................................................... 43
Figure 8. Refresh Create for Sequential After Image Log ......................... 44
Figure 9. Refresh Update for Sequential After Image Log ....................... 45
Figure 10. Refresh Messages to Snapshot Table ........................................ 46
Figure 11. Sequential Before and After Image Log ...................................... 46
Figure 12. Refresh Create for Sequential Before and After Image Log ...... 47
Figure 13. Refresh Update for Sequential Before and After Image Log ...... 48
Figure 14. Refresh Messages to Snapshot Table (Revised) ....................... 48
Figure 15. Merge Rules for Modifications ................................................ 51
Figure 16. Condensed Log ........................................................................ 51
Figure 17. Annotated Base Table ............................................................... 55
Figure 18. Base Table Fixup ...................................................................... 56
Figure 19. Basic R* Refresh of Snapshot Table .......................................... 57
Figure 20. Refresh Create for Basic R* Method ......................................... 58
Figure 21. Fixup Operation ............................................................ 59
Figure 22. Refresh Update for Basic R* Method ......................................... 60
Figure 23. Refresh using Enhancement One ............................................... 63
Figure 24. Refresh using Enhancement Two ............................................... 64
Figure 25. Refresh Create for Enhanced Condensed Log ......................... 66
Figure 26. Refresh using Enhanced Condensed Log ................................... 67
Figure 27. Update Modification Outcome and Ratio .................................. 73
Figure 28. Size Estimates for Logging Methods ........................................ 80
Figure 29. Static Base Table ................................................................. 81
Figure 30. Incremental Base Table (p = 0.1, r = 0.9) ................................. 81
Figure 31. Dynamic Base Table (p = 0.2, q = 0.1, r = 0.7) ...................... 82
Figure 32. Size Estimates for R* Methods ................................................ 87
Figure 33. Static Base Table ................................................................. 88
Figure 34. Incremental Base Table (p = 0.1, r = 0.9) ................................. 88
Figure 35. Dynamic Base Table (p = 0.2, q = 0.1, r = 0.7) ...................... 89
Figure 36. Incremental Base Table (p = 0.5, r = 0.5) ................................. 89
Figure 37. Cost Estimates for Logging Methods ........................................ 96
Figure 38. Cost Estimates for R* Methods ................................................ 99
Figure 39. Static Base Table (Case 1) ....................................................... 101
Figure 40. Static Base Table (Case 2) ....................................................... 102
Figure 41. Static Base Table (Case 3) ....................................................... 103
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>Incremental Base Table (p = 0.1, r = 0.9, Case 1)</td>
<td>104</td>
</tr>
<tr>
<td>43</td>
<td>Incremental Base Table (p = 0.1, r = 0.9, Case 2)</td>
<td>105</td>
</tr>
<tr>
<td>44</td>
<td>Dynamic Base Table (p = 0.2, q = 0.1, r = 0.7, Case 1)</td>
<td>106</td>
</tr>
<tr>
<td>45</td>
<td>Dynamic Base Table (p = 0.2, q = 0.1, r = 0.7, Case 2)</td>
<td>107</td>
</tr>
<tr>
<td>46</td>
<td>Differential vs. Full Refresh (Static Base Table)</td>
<td>110</td>
</tr>
<tr>
<td>47</td>
<td>Sequential Log Supporting Multiple Snapshots</td>
<td>116</td>
</tr>
<tr>
<td>48</td>
<td>Extended Condensed Log</td>
<td>120</td>
</tr>
<tr>
<td>49</td>
<td>Refresh Create for Extended Condensed Log</td>
<td>123</td>
</tr>
<tr>
<td>50</td>
<td>Compressing the Condensed Log</td>
<td>124</td>
</tr>
<tr>
<td>51</td>
<td>Check for Shipment of Merged Modification</td>
<td>125</td>
</tr>
<tr>
<td>52</td>
<td>Merging Entries of Extended Condensed Log</td>
<td>127</td>
</tr>
<tr>
<td>53</td>
<td>Base Table used in Multiple Snapshots</td>
<td>130</td>
</tr>
<tr>
<td>54</td>
<td>Base Table Extended for Enhancement Two</td>
<td>133</td>
</tr>
<tr>
<td>55</td>
<td>Cost Estimates for Replicated Snapshots — Logging Methods</td>
<td>135</td>
</tr>
<tr>
<td>56</td>
<td>Cost Estimates for Replicated Snapshots — R* Methods</td>
<td>136</td>
</tr>
<tr>
<td>57</td>
<td>Cost Estimates for Independent Snapshots — Logging Methods</td>
<td>138</td>
</tr>
<tr>
<td>58</td>
<td>Static Base Table (Condensed Logging)</td>
<td>140</td>
</tr>
<tr>
<td>59</td>
<td>Static Base Table (Sequential Logging and R*)</td>
<td>141</td>
</tr>
<tr>
<td>60</td>
<td>Incremental Base Table (p = 0.1, r = 0.9)</td>
<td>142</td>
</tr>
<tr>
<td>61</td>
<td>Dynamic Base Table (p = 0.2, q = 0.1, r = 0.7)</td>
<td>143</td>
</tr>
<tr>
<td>62</td>
<td>Static Base Table (Independent Snapshots, Condensed Logging)</td>
<td>145</td>
</tr>
<tr>
<td>63</td>
<td>Static Base Table (Independent Snapshots, R* approach)</td>
<td>146</td>
</tr>
<tr>
<td>64</td>
<td>Incremental Base Table (Independent Snapshots, p = 0.1, r = 0.9)</td>
<td>146</td>
</tr>
<tr>
<td>65</td>
<td>Dynamic Base Table (Independent Snapshots, p = 0.2, q = 0.1, r = 0.7)</td>
<td>147</td>
</tr>
<tr>
<td>66</td>
<td>Differential vs. Full Refresh (Static Base Table)</td>
<td>148</td>
</tr>
</tbody>
</table>
Chapter 1. Distributed Databases and Snapshots

1.1 Distributed Databases

There appears to be a common understanding that distributed databases give better performance and availability than centralized databases, and at the same time provide the quality of centralized databases in terms of preserving a global coherent database image. The expectations of what a distributed database may provide has caused some people, trying to fulfill them, a great deal of frustration — yet others, hoping to use the features in their applications, may have, up to now, become disappointed.

At the present, the development of distributed database systems has taken a new direction in the sense that one attempts to provide users with a set of mechanisms that enables them to build distributed applications with varying degree of integration. In the beginning, this will mean that the applications take a larger share of the distributed management. In the future, the applications may rely fully on the distributed management system unless other factors qualify for a different solution.

A distributed database is commonly viewed as a collection of interrelated database fragments stored at different sites. The database fragments are connected and controlled by a distributed database management system so as to obtain a global coherent database image.

The fragments of a distributed database are typically organized as local databases, with access provided for by a conventional database management system at each site. The sites holding the various fragments of the distributed database are connected by a communication network, thus enabling the distributed database management system to enforce global integrity control.

The knowledge on how the distributed database is distributed, is stored in a global catalog or data dictionary. This information is vital to the distributed database management system, e.g. when finding the subset of the database that is to be in effect during the processing of a given query. The global catalog will contain information on the databases distributed over the actual network, and may in itself be organized as a distributed database.

The overall feature of distributed databases is a global database image which means i) that the user need not be aware of the distribution of the data, and ii) that the distributed database management system will maintain the global database. The major advantages of distributed databases, when compared with centralized databases, are well
recognized in the literature — see for instance [CERI 84] — and are not recapitulated here.

The field of distributed databases and database management systems has become a promising field of database research, and several research projects have been carried out in the past 10 years. Some prototypes have been built, others are under development. Yet others, although with limited capabilities, can now be purchased. Listed in alphabetic order, the systems are: ADD, DDM, DDTS, ENCOMPASS, JDDBS, INGRES/STAR, MICROBE, MIMER*, Multibase, MIMER*, ORACLE SQL/NET, Polytheme, POREL, PRECI*, PROTEUS, SCOOP, SDD-I, SIRIUS-DELTA, System R*, and VDN.

The design and implementation of distributed database management systems involves a wide range of topics — the major being:

1. **Concurrency control**, including updates to replicated data.
2. **Data distribution** and **catalog management**.
3. **Deadlock management**.
4. **Heterogeneity**.
5. **Query processing**.
6. **Reliability**, including handling of component failures and network partitioning.

Not all of the topics listed above have been given the same amount of attention in the various research projects. The aspects of concurrency control in distributed systems has definitely been given the most attention, closely followed by the various aspects of reliability and distributed query processing. An annotated bibliography on the topics listed above is given by Mohan in [MOHA84].

A wide acceptance of distributed databases depend largely on the performance of the distributed database management systems. Leaving the obvious factors out — such as the speed of the hardware components, communication network and storage — the performance is determined by the following issues:

1. The efficiency of the concurrency control mechanism.
2. The distribution and allocation schema applied for the data.
3. The quality of the distributed query processing algorithm.

Note that each of these issues can not be solved isolated from the others. The distribution of the data will influence on the strategy to be used by the query processing algorithm. Also, the allocation scheme may increase the complexity of the task for the concurrency control mechanism. What makes it really difficult to deal with these issues, is that they pull in different direction. For example, a high degree of data replication will improve the performance of data retrieval, and thus make the job easier for the distributed query processor. On the other hand, this will increase the load on that part of the concurrency mechanism responsible for ensuring mutual consistency between copies of replicated data.
Consequently, a fair amount of the recent research and development of distributed databases and systems, has looked into alternative ways of utilizing the functional potential in distributed databases.

Usually, a distributed database is perceived as a global coherent database, i.e. as a single system image, in which the local database fragments are managed by the distributed database system. A single system image stands opposed to a distributed database perceived as multiple local databases, in which an application program may connect to a remote database through a remote database access facility. In this case we have a multiple systems image view of the database. In between, we may find numerous variations.

This suggests that a distributed database management system may be based on one out of several different system architectures reflecting a multiple or a single system image, or somewhere in between. The factor determining where a system is to be found on the scale, is the degree of distribution transparency, i.e. the degree to which the distribution aspects of the database is hidden from the user.

In [STON 87], Stonebraker introduces a taxonomy of distribution transparency containing six transparency rules. A discussion of distribution transparency is also given in [KÄHL 87b] and [RISN 87]. The last two papers discuss the consequences of supporting the transparency rules as each level of transparency is subject to a new mechanism that must be supported by the distributed database management system.

The notion of distribution transparency is defined as follows:

1. **Schema Transparency**, which provides transparent access and manipulation of the entire database schema or system catalog, e.g. table definitions and user definitions.

2. **Transaction Transparency**, which provides transparent distributed integrity and concurrency control, and for recovery in case of failures.

3. **Location Transparency**, which hides the actual location of database objects from the applications and the users.

4. **Performance Transparency**, which ensures that a given query is solved in the most efficient way independent of the site at which the query is submitted.

5. **Replication Transparency**, which allows for several identical copies of data to be maintained at different sites for performance and availability reasons.

6. **Fragmentation Transparency**, which allows a table to be split and maintained in fragments at the sites where these fragments are most frequently used for performance and availability reasons.

For some of these rules it is meaningful to speak of **degrees of transparency**. In other words, there exists more than one solution — some only supporting transparency to some degree. To put it more correctly, some of the solutions transfer responsibility for distributed management to the application logic which obviously places a burden on the application designer. However, to some applications this may provide for an appropriate solution as support of full transparency may not be of paramount importance.
An example of such is the transaction transparency rule. Transaction transparency ensures that concurrently running transactions updating data at more than one site, are executed as atomic operations, and guarantees transaction serializability.

This issue has a theoretical solution called two-phase-commitment, often abbreviated 2PC. Systems using 2PC depend on a reliable connection between the transaction managers and a quick recovery period in case of a failure, or else data may become unavailable for some time. We will return to this problem later on.

Because of this, some systems support a less sophisticated transaction mechanism, often referred to as single site update transactions, see for example [RTI 86]. This mechanism allows for update at one site only, but allows for read from several sites. Updates, that logically should be contained in one transaction and span several sites, must therefore be broken down to several transactions. In other words, the responsibility for this now lies in the application logic.

There appears to be a trend towards a distributed system architecture supporting this looser notion of distribution — either distribution transparency may be provided to the full extent or only partially. This is reflected in the announcements made by the leading relation database management systems, see for example the 3-phase development plan of INGRES/STAR [MADA 86].

The reasons for this change in development, is that some distributed applications accept, or demand for, looser notion of distribution. Other issues may be more important than full distribution transparency. On the other hand, the applications do require mechanisms that allow for retrieval of data from remote sites without having to deal with the complexity of communication networks.

Above, we used transaction transparency to demonstrate that it is meaningful to speak of degrees of transparency. Replication transparency is another issue which, as one of its solutions, introduces the database snapshot mechanism.

1.2 Snapshots — an Alternative to DQP

Distributed Query Processing (DQP) and the quality of a DQP algorithm is one of the issues that determines the performance of a distributed database. If we assume a distributed database where, \(i\) data is distributed to several sites, \(ii\) no replication of data is used, and \(iii\) a large proportion of the applications contain transactions requiring access to data from more than one site, then the following situation may arise:

As the number of of transactions requiring remote data access increases, the performance obtained for the applications may decrease considerably. Even worse, transactions, and thus the applications, may fail to run as remote data becomes temporarily unavailable. Even if a good distributed design method has been applied, it may still be difficult to obtain high transaction rates in such an environment.

Therefore, 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 [DAVI 85]. All copies of the same logical data item must agree on the same current value of the data item. Maintaining replicated data may therefore become a costly operation; in general the cost will increase by the degree of replication.

Some applications may be willing to pay for the added cost of maintaining replicated data in return for a copy at hand. Applications may, however, not be quite as willing to accept the risk of leaving the distributed database in an inconsistent state — if availability is to be preserved at all times. In the event of network partitioning — where the communication network separates the sites into two groups — mutual consistency between copies may be hard to ensure, with the result that data may be left in an inconsistent state. The normal way of preventing this, is not to allow for updates in the case of network partitioning, or by restricting updates in some other way [DAVI 85].

There are solutions that try to avoid this problem. Existing database management systems allows users to operate only upon the latest (committed) database state. Many applications will tolerate (or require) access to data which are not 100% up-to-date as long as the data belongs to a previous 'transaction consistent' state of the database. In these applications, the relevant data can be copied into local database entries and operated upon locally. In this way, the state of the database may continue to evolve independent to the applications requiring the "freezed" portion of the database, and at the same time, the copied state is maintained for the same applications.

By having the portion of the distributed database that is frequently accessed by an application stored locally, the query processing becomes a local query processing problem. As the copy of the data is local domain only, the local query processing is not loaded with the maintenance overhead which is normally associated with replicated data. Instead, the local copy of the data is requested updated periodically, i.e. the content of the local copy is refreshed to reflect the current state of the database. We call this a database snapshot mechanism.

1.2.1 The Introduction of Snapshots

Database snapshots were first introduced in [ADIB 80]. The introduction of database snapshots was motivated by the observation that in many applications an up-to-date accuracy is not needed. In fact, some applications require data as taken at a certain point of time as long as the data represent a consistent state of the database, e.g. sales figures extracted for planning purposes or reporting. Other applications may tolerate "outdated" data for the sake of simplifying processing, e.g. figures extracted for statistics, or the data may be updated very seldom and an old version may be good enough for a long period of time, e.g. a product catalog.

A database snapshot is a read-only copy of some portion of the database. The snapshot contains data extracted from a transaction consistent state of the database as of the time the snapshot is taken. A snapshot will thus freeze portions of the database, and preserve this state for the future. Applications and users may thus be able to refer to previous, yet consistent, states 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 be defined as a general query and the snapshot refresh operation may easily be obtained by executing the query.

Database snapshots represent a way of providing replicated data to users and applications of a database without loading the same users and applications with the overhead of maintaining the replicated data. Instead, the database, as of a well-defined time, may be made available to these users and applications.

1.2.2 Snapshots in Distributed Environment

In a distributed database, the database snapshot mechanism becomes even more interesting as it may be used to provide for locality of data without imposing the overhead normally associated with data replication.

So far, the common methodology for obtaining data replication in a distributed database, has been to apply a horizontal and vertical fragmentation scheme to a table and then determine the physical allocation of each fragment. For a discussion of this, see [CERI 84]. This scheme allows for replication of table fragments. By storing fragments of a table at sites where the data of the fragment is frequently used, the need for costly, remote access is decreased, and the probability of having the appropriate data available is increased. However, the scheme suggests that mutual consistency between copies of data is to be maintained at all times.

Database snapshots represent a cost effective substitute for replicated data in distributed databases. It appears to be a useful alternative to distributed query processing in a distributed environment where a table is used at many different sites, and for which a local copy is important for efficiency purposes, or to ensure availability. The mechanism supports data distribution similar to the fragmentation scheme. Since a snapshot is a local domain only, transactions applying local snapshots are not loaded with the overhead of ensuring that the snapshot data is consistent with the original.

There may be many instances when consistency is not of paramount importance, or consistency can only be obtained at certain "checkpoints". The following statements, quoted from a recent ARPA publication [MOCK 87], emphasize this issue: "Access to information is more critical than instantaneous updates or guarantees of consistency". "When updates are unavailable due to network or host failure, the usual course is to believe old information while continuing efforts to update it". Other examples of when this is a relevant issue, are the stock lists issued to all operating NATO vessels at certain checkpoints [SJOF 84], and inventory lists extracted for product modelling in CAD/CAM applications [LIEN 87].

In most cases a single site will appoint itself naturally as the site holding the prime copy, e.g. the head quarters for the vessels in the NATO case. The sites holding replicas will then act as read-only sites, e.g. each vessel in the NATO case.
As processing using database snapshots becomes a local problem, emphasis is moved to methods that can remedy the temporary inconsistency in data, i.e. how to refresh the database snapshot. The read-only copies can easily be refreshed 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.

Communicating the refreshed snapshot may degrade the overall system performance in a distributed environment. The number of messages that need to be transferred to the snapshot site may be reduced by applying a differential refresh strategy. A differential refresh strategy is 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.

1.2.3 Design Objectives

The thesis will discuss database snapshots used in distributed databases. Without loss of generality, the discussion is kept within the scope of the relational model of data [CODD 70]. The snapshot mechanism as such, can also be applied e.g. to the hierarchical and the network model.

Within the scope of a relational distributed database, a database snapshot mechanism should be designed to fulfill the following requirements:

1. The mechanism should support snapshots based on one or several tables.

Facilities should be provided for defining snapshots based on a single table — either as a full copy of that table or as a restriction and/or projection. The same facility should also fulfill the definition of more complex snapshots such as a join of several tables or previously defined snapshots.

2. The mechanism should support one-copy snapshots as well as replicated snapshots.

Replication is often needed in distributed systems to ensure data availability or performance. Snapshots provide for such replication of data without imposing the overhead of maintaining replica consistency. Replication of one snapshot to several sites may therefore become a useful tool in distributed management when the same data is requested by more than one recipient. A replicated snapshot should reflect the same database state in each replica, i.e. the snapshots should be refreshed collectively.

3. The mechanism should support independent snapshots based on one table.

Defining several snapshots based on one table provides a data definition facility similar to the fragmentation scheme. Each snapshot should be independently refreshable as well as being allowed to specify its own restriction and/or projection of the table. Unlike the restriction that is often imposed on horizontal table fragments, independent snapshots defined on one table may overlap.

Chapter 1. Distributed Databases and Snapshots 7
In the following we will discuss how a snapshot mechanism can be designed to fulfill these requirements, and how the refresh methods can be implemented to give efficient support of database snapshots in a distributed environment.

1.3 Related Work

Most work on database snapshots has been done within the framework of the R* distributed database management systems project at IBM Research in San Jose [LOHM 85]. The notion of snapshots was introduced by Adiba and Lindsay [ADIB 80], and a more recent paper describes an algorithm for implementation of the snapshot mechanism in a distributed environment [LIND 86].

The distributed DBMS INGRES/STAR [MCCO 86] supports a related mechanism called deferred update in which a 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 in a demon process at the replica site. Deferred updates thus relax the requirement of mutual replica consistency. A table that is replicated to one or several sites and updated at its primary site only by the deferred update mechanism, corresponds to a snapshot providing the demon process performs the updating periodically. We will return to the deferred updates method when discussing mechanisms for refreshing database snapshots in “Chapter 4. Differential Refresh Methods”.

Some work on refresh processing applicable to snapshots has also 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 materialized views is suggested to be performed in a differential manner. The paper presents a method in which the view table can be updated incrementally, based on modifications submitted to the tables referenced in the view. In contrast to this thesis, it does not discuss how the base table modifications are determined. We will return to this when discussing the implementation of snapshots in “Chapter 3. Implementing Snapshots”.

In [BLAK 86], a materialized view is updated immediately after each transaction. An alternative approach has been proposed recently in [HANS 87], called deferred view maintenance, that incrementally updates a materialized view just before data is retrieved from it. Modifications to the base tables of the view are kept in an adapted type of hypothetical relations [WOOD 83]. The main issue in [HANS 87] is a performance analysis of view materialization strategies. Unlike database snapshots, materialized views must be refreshed every time they are queried. Thus, deferred view maintenance not only have to consider modifications accumulated since the view was last queried, but must continuously include modifications while the view is in use. Therefore, view materialization addresses issues different from those relevant to snapshot refresh. Furthermore, the aspects of distribution are not discussed.

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 this thesis, the algorithm does not include the processing cost of refresh. The algorithm has been analyzed and compared with the algorithms proposed in this thesis. A detailed description of the algorithm is therefore given later in the thesis.

Finally, our work is applicable to differential maintenance of complex objects [HASK 82] stored in relational CAD/CAM databases [FRÖS 86]. In such environments, complex objects are retrieved for processing from a central database — an operation commonly known as check out. The retrieved objects are then kept in a local repository for modification. When the user at the work-station has completed her/his task, a request will be made to the central database to replace the checked out objects with the modified ones. This operation is called check in. The objects that were checked out have, in the meantime, remained locked, so that replacement with the new objects can be done. Instead of writing the entire changed objects back in to the central store, a log recording the modifications can be maintained and later merged with the original tables. In this way, the old objects may be refreshed in a differential manner so as to reflect the latest changes made to them.

1.4 Plan of Thesis

The requirements posed for the snapshot mechanism provide an outline for the thesis as follows:

First, the definition and semantics of database snapshots is discussed in detail in “Chapter 2. Snapshot Definition and Semantics”. Statements for defining and maintaining database snapshots — in accordance with requirements — are proposed for a distributed environment.

The implementation aspect of database snapshot is the issue of “Chapter 3. Implementing Snapshots”. The chapter describes various methods for maintaining database snapshots and discusses the implication of supporting snapshots in a distributed database — such as the performance impact, the integration of the snapshot mechanism in database management systems, and other operational issues.

“Chapter 4. Differential Refresh Methods” focuses on incremental methods for maintaining database snapshots, which is the major issue presented in this thesis. The chapter describes two different approaches that can be used to achieve this — a logging approach and the R* approach. For each, alternative refresh strategies are discussed, and algorithms are given to clarify the difference in nature for each of the alternatives proposed.

The two approaches to differential refresh of snapshots are analyzed next, in “Chapter 5. Refresh Method Analysis”. An analytic model is established for describing each approach, and includes the alternatives proposed in “Chapter 4. Differential Refresh Methods”. A cost analysis, based on the analytic model, is provided. The analysis compares the performance impact of snapshot maintenance for each of the differential refresh methods.
In "Chapter 4. Differential Refresh Methods" and "Chapter 5. Refresh Method Analysis", database snapshots are discussed within the context of being operated on as a single copy of some portion of the database. In "Chapter 6. Extending Refresh for Snapshot Replication", emphasis is on methods that can support replicated as well as several independent snapshots. This can be obtained by extending the differential methods described in "Chapter 4. Differential Refresh Methods". The snapshots mechanism will now fulfill the requirements posed above.

Finally, "Chapter 7. Conclusion" summarizes the research in this thesis and indicates directions for future research in the field of distributed databases and snapshots.
Chapter 2. Snapshot Definition and Semantics

2.1 Introduction

A database snapshot captures the current, transaction consistent state of some portion of the database. Once a snapshot has been defined and initialized, its content can be accessed as an ordinary table of the database and the portion of the database that contributed to the value of the snapshot may continue its own, independent evolution. Unlike ordinary tables, snapshot tables are read only.

In a relational database system, a database snapshot may be defined as a general query. The first definition of a snapshot was given in [ADIB 80]. The paper stressed that snapshots should be supported as a separate mechanism by the database management system. Statements were introduced for definition and manipulation of snapshots — a DEFINE SNAPSHOT statement to be used for creating database snapshots in which a query is associated with a snapshot name, and a REFRESH SNAPSHOT statement for refreshing the content of the snapshot. The statements were defined in an SQL syntax.

The syntax of SQL has since then been changed, and the application of snapshots in a distributed environment has introduced some new issues to the snapshot concept. A set of new statements to be used for defining and manipulating database snapshots is therefore proposed in this chapter. The statements not only adhere to the latest syntax of SQL [ISO 86], they are made to provide support for the distributed aspects of snapshots.

In defining the statements, a distributed environment as the one provided by the R* system [DANI 82] is assumed. One of the important features of R* is location transparency, i.e. data may be referenced independent of their actual storage location. This is accomplished by a special naming convention and by allowing tables to be moved from one location to another (table migration) without affecting the applications referring to them. The approach also makes provision for another feature; site autonomy, i.e. objects may be created and operated on locally without being conditioned by other sites. This is achieved by letting each site maintain its own system catalog. Information on local objects is then kept in the local catalog.

The R* naming convention, introduced by [LIND 81], provides for site transparency by assigning a fully qualified name to each object consisting of the object’s name and the
name of its birth site. Thus, the fully qualified name of a table consists of the table name followed by the name of its birth site. If the table is migrated to a new site, i.e. the table is moved from the present site to a new storage site, the fully qualified name remain the same. In this way, applications may proceed without modification. Instead, the system catalog at the birth site and the table’s new storage site, record the change taken place.

Alternative conventions for obtaining site transparency can be used — one of them being globally unique synonyms or links, see e.g. [RTI 86].

In the following sections, a database snapshot is first defined within the scope of being operated on as a single copy of some portion of the database. Statements for snapshot definition and refresh are given, and the semantics of database snapshots are defined. Statements supporting management of snapshots in a distributed environment are also given. We then discuss how these statements can be extended to support replicated copies of the same portion of the database as well as several independent snapshots, and describe the semantics of replicated snapshots in a distributed environment. Finally, a summary of the statements is given.

2.2 Snapshot Definition

In [ADIB 80], a snapshot is defined by a query which is associated with a snapshot name. We introduce a CREATE SNAPSHOT statement which adheres to the latest syntax of SQL [ISO 86]:

```
CREATE SNAPSHOT [creator.]snapshot-name [(column-name-list)]
    AS select-statement;
```

The snapshot is given a name — snapshot-name — that must be unique among all database objects (i.e. tables, views, synonyms, and other snapshots) created by the user. The snapshot-name may be qualified by the name of the creator of the snapshot. In a distributed environment, both the creator and the snapshot-name may — as explained above — be qualified by the name identifying the birth site of the creator and the snapshot, if different from the current site, see also [LIND 81].

The column-name-list defines the column names (attributes) of the snapshot. The list of names may be omitted — in which case the columns of the snapshot inherit the names of the columns from which they are derived. The data types of the columns are always determined by the columns on which they are defined.

The select-statement defines the content of the snapshot. The statement may be any query that can be expressed by the SELECT statement of SQL not including an ORDER BY clause. A snapshot may in other words be defined as a simple restriction or projection on a source table, view, or snapshot, as well as a complex join involving restrictions and projections of several tables. The select-statement may also include aggregation functions.

---

1 The birth site is the site at which the object is first created — hence the name.
The tables involved in the select-statement of a snapshot definition is commonly referred to as base tables. The definition of the snapshot, including its defining select statement, is stored in the system catalog at the birth site of the snapshot. The time of the snapshot creation is also recorded. This information will be used in future refresh operations on the snapshot.

The base tables involved in the snapshot definition may in some cases be affected by a snapshot definition. Ideally, operations on a base table should be allowed to proceed independent of the number of snapshots currently defined on the base table. The working load involved in restoring the content of a snapshot should fall on the snapshot table alone. However, in order to ensure efficiency, one may have to compromise. We will return to this discussion in “Chapter 3. Implementing Snapshots”.

The snapshot definition initializes the snapshot table by evaluating the select statement and storing the result in the snapshot table. In a distributed environment this operation may involve several database sites. As soon as the snapshot has been defined, it can be treated like an ordinary table. The snapshot may be used in any query and can take part in the definition of other snapshots or views. The user may grant (and revoke) privileges on the snapshot to other users. The user may also create indexes to improve the access performance of the snapshot.

A database snapshot resembles a materialized database view [BLAK 86]. There are, however, three major differences:

- First, a database view defines a dynamic window of some portion of the database (as of the current database state), whereas a snapshot creates and preserves a static copy of the database as at the time of the snapshot definition or snapshot refresh.

- Second, views may allow updates to be performed on the tables constituting the view, whereas snapshots are read-only. Updates to a snapshot, if allowed, would otherwise have to be propagated to the base table — which violates the intention of database snapshots as being loosely coupled to the master copy, i.e. the master copy should be allowed to evolve independent of any snapshots.

- Finally, a view depends, at any time, on the existence of the base tables referenced in the view definition. Whenever a base table is deleted, all view definitions based on the base table become invalid. A snapshot, on the other hand, may continue its lifetime as long as no refresh operation is invoked. A refresh operation on the snapshot will obviously fail, in which case the snapshot will be made invalid.

The content of a snapshot is defined by the select-statement, which refers to base tables by their birth name. A snapshot may therefore be refreshed as normal if some of its base tables have migrated to other sites. The snapshot definition is in other words independent of the actual storage location of its base tables, a property inherited from the location transparency.

An example illustrating the database snapshot mechanism is now given. The example applies a base table Resorts describing holiday resorts. The PriceLevel of a named Resort in a Country is given in scale ranging from 1 to 10 as shown in Figure 1.

Chapter 2. Snapshot Definition and Semantics 13
### BASE TABLE RESORTS

<table>
<thead>
<tr>
<th>Resort</th>
<th>Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beirut</td>
<td>Lebanon</td>
<td>6</td>
</tr>
<tr>
<td>Brighton</td>
<td>England</td>
<td>6</td>
</tr>
<tr>
<td>Cannes</td>
<td>France</td>
<td>7</td>
</tr>
<tr>
<td>Colombo</td>
<td>Sri Lanka</td>
<td>3</td>
</tr>
<tr>
<td>Florence</td>
<td>Italy</td>
<td>5</td>
</tr>
<tr>
<td>Tenerife</td>
<td>Spain</td>
<td>4</td>
</tr>
</tbody>
</table>

### SNAPSHOT TABLE $S_1$

<table>
<thead>
<tr>
<th>Resort</th>
<th>Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beirut</td>
<td>Lebanon</td>
<td>6</td>
</tr>
<tr>
<td>Brighton</td>
<td>England</td>
<td>6</td>
</tr>
<tr>
<td>Colombo</td>
<td>Sri Lanka</td>
<td>3</td>
</tr>
<tr>
<td>Florence</td>
<td>Italy</td>
<td>5</td>
</tr>
<tr>
<td>Tenerife</td>
<td>Spain</td>
<td>4</td>
</tr>
</tbody>
</table>

**SnapTime** = 00:00  
**SnapRestriction** = `PriceLevel < 7`

Figure 1. Example Base Table and Snapshot Table: Base table Resorts and snapshot table $S_1$ as at definition time of snapshot.

The following snapshot is defined on the base table (Figure 1 depicts the snapshot $S_1$ and the base table Resorts as at the time of the definition):

```
CREATE SNAPSHOT $S_1$ AS
SELECT *
FROM Resorts
WHERE PriceLevel < 7;
```

### 2.3 Snapshot Refresh

Not only should it be possible to define *what* part of the database that the snapshot should capture, it should also be possible to define *when* the snapshot is to be refreshed. A user\(^2\) may want to refresh a snapshot on *demand* or by requesting the snapshot refresh to be *periodic*. Other requirements for snapshot refresh may need different solutions and we will return to this at the end of this section.

\(^2\) With user we here — and in the rest of this chapter — mean an experienced user, preferably a database administrator, or an application program initiating the task in accordance with the design specification made by a system designer.
The demand option may easily be provided by means of a REFRESH SNAPSHOT statement:

    REFRESH SNAPSHOT snapshot-name [AT snapshot-time];

The named snapshot will be refreshed immediately or at the snapshot-time if a specific time (e.g. day and hour of the day) is given by the statement. The statement will cause the system to replace the present content of the snapshot by the result as obtained by executing the select-statement associated with the snapshot definition. The system uses the definition as found in the system catalog (at the snapshot's birth site) for the snapshot-name. In a distributed environment the refresh operation may involve several database sites depending on the object(s) referenced in the select statement of the snapshot. The snapshot, when refreshed, will capture the current database state (or the state at the given snapshot-time). The time associated with the snapshot in the system catalog will be set to the time of the refresh.

As an example of snapshot refresh, the previously defined snapshot is requested updated at 9:00 am — which may be expressed as:

    REFRESH SNAPSHOT S1 AT 9:00 am;

The user may, as mentioned above, request for a periodic refresh of a snapshot, e.g. sales figures to be extracted for planning purposes every week, bank accounts with overdraft every month, etc. It should be possible to define database snapshots with such properties. The CREATE SNAPSHOT statement can easily be extended with a refresh clause to support this facility. The full statement becomes;

    CREATE SNAPSHOT [creator.]snapshot-name [(column-name-list)]
       AS select-statement
       [REFRESH EVERY snapshot-period [STARTING AT snapshot-time]];

where the snapshot-period may be expressed as a number of hours, days, weeks, etc. The outcome of the definition is a snapshot that will be refreshed by the system at the end of every period starting from the present time or at the snapshot-time if this option is used. The snapshot is first initialized at the time the definition statement is submitted.

The refresh period as defined for the snapshot will also be stored in the system catalog, and the snapshot will be included in a system refresh check list at the snapshot's birth site. This list will enable the system to refresh the snapshot at the end of a refresh period. We will return to this in "Chapter 3. Implementing Snapshots" where we discuss the operational issues of database snapshots.

The refresh operation as defined above is demand or clock driven. One may also envisage an event driven refresh process if the database system allows for user-defined events. An example of event driven refresh will then be;

    REFRESH SNAPSHOT S1 AT EVENT CatalogRelease;

in which CatalogRelease may be defined as the event of releasing the new edition of the holiday resorts catalog.
Events introduce the need for an added option in the REFRESH SNAPSHOT statement, and the REFRESH clause of the CREATE SNAPSHOT statement. Events generally introduce a completely new issue that must be handled by the database management system. How should events be defined, and how do we integrate them in database management systems? The support of events is in itself a research topic, and is for this reason, not studied any further in this thesis. For a discussion of events, see [ROBI 79] and [BREU 79].

2.4 DROP and ALTER Statements

In order to make the snapshot mechanism complete, it should be possible to drop a snapshot by issuing a DROP SNAPSHOT statement:

```
DROP SNAPSHOT [creator.]snapshot-name;
```

The snapshot identified by the `snapshot-name` will be deleted from the database and the definition of the snapshot is removed from the system catalog. The snapshot will also be removed from the system refresh check list if previously added to this list.

Redefining a database snapshot by adding a new column (or removing an existing one), or changing the select statement defining the selected portion of the database, will require full reprocessing of the snapshot, i.e. a new snapshot content will have to be initialized. There is, however, one type of modification to a snapshot that may be done without any processing at all — namely changing the present snapshot refresh period. This change will not cause any refresh operation until the time is due. An ALTER SNAPSHOT statement is therefore included to support this:

```
ALTER SNAPSHOT [creator.]snapshot-name
    REFRESH EVERY snapshot-period [STARTING AT snapshot-time];
```

The statement will add (or modify) the snapshot refresh period to (in) the system catalog, and include the snapshot on the system refresh check list, if not included already. The outcome of the modification will either be an instant refresh of the snapshot, or at the time given by the `snapshot-time`, cf. the CREATE SNAPSHOT statement.

In a distributed environment, a user may want to move to another database site and take the snapshot definition with him. By issuing a migrate statement on snapshots similar to migrate of table, site transparency can be maintained. However, support of a snapshot migrate statement is not straightforward. Ideally, we would like the definition of the snapshot to be moved along to the snapshot table's new storage site. This can not be guaranteed to complete successfully as the `select statement` (defining the snapshot) may reference local synonyms or other local objects that are not recognized at the new storage site. Furthermore, the statement may now violate authorization constraints.

To support a snapshot migrate statement, we therefore have to separate the definition of the snapshot (i.e. its `select statement`) from the snapshot table itself. Now, only the table is moved to the new storage site and the definition remains at the birth site. The disadvantage of this method is that snapshot refresh is conditioned by the birth site which violates the intention of snapshots as providing for high availability of data.
This problem can be avoided if the migration is done stepwise as follows: First, create the snapshot at the new site, then drop the snapshot at the old site. From an operational point of view this is a better solution since the data will not be unavailable to the user in the meantime.

The stepwise approach does not, however, provide for site transparency. Applications referring to old snapshot will have to be changed. This situation can be amended by defining the old snapshot name as a synonym containing the new snapshot's name. Applications may now refer to the new snapshot indirectly by the synonym name, and no change to the application programs becomes necessary.

## 2.5 Replicated and Independent Snapshots

So far we have discussed snapshots as being a single copy of some portion of the database. We will now turn to the situation in which a snapshot is replicated and stored at several sites. Although this may be accomplished for a snapshot based on several base tables, it is most likely to be used for snapshots that are based on one base table only. The latter case is very close to the data distribution as obtained by a fragmentation scheme applied on a table, cf. [CERI 84].

We distinguish between two forms of snapshot replication. A **replicated snapshot** is a snapshot with the same content stored at several sites, and the snapshots are refreshed collectively. 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, or a library may wish to replicate the bibliographic catalogue to each of its branches.

Replicated snapshots may be obtained by issuing identical CREATE SNAPSHOT statements at the sites where copies are wanted, i.e. each replica is treated as an independent snapshot. It will, however, be difficult to keep these snapshots "tied together". After all, they are defined as independent snapshots and there is no guarantee that all definitions are the same. Below, we will see how to tackle this problem.

The other form of replication may be obtained, as just described, by having several **independent snapshots** defined on a base table (or several base tables for that matter). Each snapshot will have their own content and their private refresh cycle. The snapshot may overlap in the sense that they partially extract the same data from the base table. For instance, given a company that has several departments — each responsible for selling a subset of the products that are for sale by the company, each department may define its own private snapshot by extracting the products sold by the department from the company's product catalog table.

Independent snapshots may easily be supported by the previously given statements. Each snapshot is defined as a separate snapshot using the CREATE SNAPSHOT statement, and the snapshots may be refreshed on demand or at regular time intervals.

The main difference between the two forms of snapshot replication identified above is that in the first the company probably wants all replicas of a replicated snapshot refreshed collectively. In contrast, independent snapshots will have their own independent
refresh frequency, e.g. whenever new products are introduced for sale by a particular department, that department may want to refresh its product information.

Replicated snapshots are tied together by a common definition — both with respect to what data they should contain, and when the content should be refreshed. In order to keep the replicas consistent with one another, we introduce a refresh manager who is responsible for creating and maintaining the content of all replicas. The refresh manager will reside at the site where the frame for the replicated snapshot is defined. The frame is defined by a DEFINE SNAPSHOT statement as follows:³

```
DEFINE SNAPSHOT [creator.]snapshot-frame [(column-name-list)]
    AS select-statement
    [REFRESH EVERY snapshot-period [STARTING AT snapshot-time]];  
```

The snapshot-frame represents a coming cluster of snapshots — all with the content as defined in the select-statement and refreshed simultaneously. Note that the definition may be entered at a site different from any of the base tables. Furthermore, no snapshot needs to be defined at the site of the refresh manager.

A snapshot-frame has the same properties as a snapshot defined by a CREATE SNAPSHOT statement, i.e. the definition is entered in the system catalog at the birth site of the frame, and the snapshot frame is included in the system refresh check list. A refresh cycle may not necessarily be defined for a given frame, in which case the snapshot replicas may be refreshed collectively on demand by issuing a REFRESH SNAPSHOT statement on the snapshot-frame. If a refresh cycle is given to the frame, it may be changed by the ALTER SNAPSHOT statement. The frame may also be dropped by the DROP SNAPSHOT statement.

A snapshot confirming to the snapshot-frame may now be created by issuing a CREATE SNAPSHOT statement according to the following Format 2 specification of the statement:

```
CREATE SNAPSHOT [creator.]snapshot-name
    AS snapshot-frame;
```

For a snapshot defined as a replica based on a snapshot frame, the snapshot-frame is kept together with the snapshot-name in the local system catalog, and the name and site of the snapshot replica is added to a replica list associated with the system catalog entry of the snapshot-frame. This enables the refresh manager to refresh this copy along with the others that are defined over the same snapshot-frame.

Each replica is defined by a Format 2 CREATE SNAPSHOT statement, and a replica may be removed independent of the others by issuing a DROP SNAPSHOT statement. Dropping a replica will cause the removal of that snapshot from the replica list associated with the snapshot-frame.

³ A snapshot frame represents a definition to be used for creating snapshot replicas. A DEFINE SNAPSHOT statement is therefore used to emphasize the definition issue, and to distinguish frames from snapshots defined by CREATE SNAPSHOT statements.
An example illustrating the use of replicated snapshots is now given. The example is defined over the base table `Resorts` as given in Figure 1. We assume that the base table resides at Site 1 and that the snapshot-frame is defined at the same site as follows:

```sql
DEFINE SNAPSHOT S@Site1 AS
SELECT *
FROM Resorts@Site1
WHERE PriceLevel < 7
REFRESH EVERY MONTH
STARTING AT 1987-01-01;
```

Note that fully qualified names for the snapshot frame and the base table have been used in the example, although this is not needed if the statement is issued at Site 1.

Replicas confirming to the snapshot-frame are now created at Site 2, Site 3, and Site 4 as follows (fully qualified names are used for clarity):

```sql
CREATE SNAPSHOT S2@Site2 AS S@Site1;
CREATE SNAPSHOT S3@Site3 AS S@Site1;
CREATE SNAPSHOT S4@Site4 AS S@Site1;
```
Figure 2 depicts the base table, the snapshot frame and the snapshot replicas. As can be seen from the figure, the refresh manager must know what to refresh, when and to whom. This information is kept in the system catalog at the birth site of the snapshot frame — at Site 1 in the example above. At the replica sites, a reference to the snapshot-frame is kept so as to provide for drop of snapshot replicas independently. The snapshot-frame itself may also be dropped — in which case all replicas will be removed as well. Note that an ALTER SNAPSHOT statement can not be issued on a snapshot replica. The property of a snapshot replica is in other words determined by the snapshot-frame. In some cases, as we shall see below, a REFRESH SNAPSHOT statement can be issued on one particular snapshot replica.

The syntax notation for replicated snapshots as proposed above, enables dynamic creation and deletion of snapshots replicas. A new replica may be added to the group and an existing replica may be removed at any time. For example, a new replica is introduced at Site 5, and the replica at Site 2 is removed:

```
CREATE SNAPSHOT S_5@Site5 AS S@Site1;
DROP SNAPSHOT S_2@Site2;
```

The initial content of snapshot S_2 is determined by the same criteria as in the case of snapshot recovery which is discussed below.

As pointed out above, replicated snapshots are refreshed collectively. In a distributed environment the obvious question is: How tolerant should the refresh of replicas be to site failures — either prior to, or during the refresh operation?

In order to perform the refresh operation, the refresh manager must be able to access the sites holding the base tables as referenced in the snapshot-frame. If a replica site becomes unavailable for refresh, there are two possible strategies.

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 sites that are available at time of refresh 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 as at the time the refresh operation was invoked, or each replica may reflect the most up-to-date state of the base table.

As an example, consider the case described in Figure 2. Given that all snapshot replicas defined on the Resorts table — S_2, S_3, and S_4 — are in the same state. A new holiday resort — Crete,Greece,5 — is added to the base table. At the following refresh of the replicated snapshot, Site 2 holding S_2 is unavailable. If we apply the second strategy, the remaining replicas are refreshed at Site 3 and Site 4.

Following this, another holiday resort — Mallorca, Spain, 5 — is added. When Site 2 holding snapshot S_2 becomes available, should it immediately be refreshed, and if so,
should both Crete and Mallorca be added to the snapshot, or should only Crete be added awaiting the next refresh for Mallorca to be added to all replicas?

According to the definition of replicated snapshots — being snapshots with the same content and refreshed collectively, the latter approach — in which only Crete is added at the following refresh of snapshot $S_2$ — should be chosen. Information necessary to regenerate the state at the time of refresh invocation must therefore be kept around for some time, marked appropriately so new changes can be distinguished from old ones. We will return to this in the discussion of refresh mechanisms for replicated snapshots in “Chapter 6. Extending Refresh for Snapshot Replication”.

By default, the strongest notion of replica consistency will be enforced, which is mutual consistency between the snapshot replicas. This means that all replicas must be refreshed collectively, and in the case of site failure, none of the replicas are refreshed.

In cases where a looser notion of replica consistency is requested, this must be identified along with the definition of the snapshot frame. This can be done by the following option in the DEFINE SNAPSHOT statement;

[REPLICA MODE [NOT] CONDITIONAL]

where replica refresh as at invocation time is denoted by a conditional replica mode, and replica refresh reflecting the most up-to-date state of the database is denoted by an unconditional replica mode (i.e. MODE NOT CONDITIONAL). The conditional mode is thus restricted by the database state as of a particular time — hence the name. In the unconditional mode, a replica will reflect the current database state.

If the REPLICA MODE option is left out, mutual replica consistency is enforced. In cases where replica refresh is unconditional, refresh of one particular snapshot replica, using the REFRESH SNAPSHOT statement, is allowed.

The situation when creating a new copy of a replicated snapshot is similar. The content of the new copy should reflect the same database state as the other snapshots defined on the same snapshot-frame. The content can either be copied from one of the existing copies, or be established from the snapshot-frame definition as of the time the refresh operation was last invoked for the existing replicas. When the first copy is requested, the initial content is established by snapshot refresh.

Similar consideration 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. If several independent snapshots are defined on one base table, some refresh mechanisms will rely on a special way of marking the base table tuples. Again we will return to this in “Chapter 6. Extending Refresh for Snapshot Replication”.

The recovery aspect of database snapshots in distributed databases is discussed independent of the replica consistency issue in “Chapter 3. Implementing Snapshots”.

Chapter 2. Snapshot Definition and Semantics 21
Figure 3. Snapshot Definition and Manipulation Statements

2.6 Summary

Statements for supporting database snapshots in a distributed environment have been proposed. The statements are listed in Figure 3.

The statements support snapshots based on one or several tables, and allow for replicated snapshots as well as several independent snapshots on the same base table. Thus,
the statements fulfill the design objectives posed in “Chapter 1. Distributed Databases and Snapshots”.

In this chapter we have discussed the snapshot mechanism at a conceptual level. In the rest of this thesis, we will discuss methods that can support the mechanism in a distributed environment.
Chapter 3. Implementing Snapshots

3.1 Introduction

A successful implementation of database snapshots in a distributed environment rely mainly on the support of an efficient snapshot refresh operation. The refresh operation should also fulfill the semantics of snapshot refresh, which is to refresh the content of the snapshot to reflect the current transaction consistent state of the database — in accordance with the snapshot definition, cf. “Chapter 2. Snapshot Definition and Semantics”. Snapshot refresh can either be requested on demand, or initiated by the system at the end of each refresh period defined for the snapshot. To refresh a given snapshot correctly, all changes to the database that are relevant to the snapshot since the last refresh of the snapshot must be detected and applied to the snapshot.

Snapshot refresh methods can be divided into two classes; full refresh strategies and differential refresh strategies. A full refresh is, as the name implies, a full refresh of the snapshot content achieved by reproducing its content at each refresh. The refresh operation will thus send a new snapshot each time the operation is invoked. The objective of differential refresh strategies is to reduce the refresh processing by detecting the modifications that have been made since the last refresh of the snapshot, and then transfer these to the snapshot.

In order to detect these modifications, the differential refresh method makes use of additional information that is either stored in the base tables referenced in the snapshot definition or kept in a domain elsewhere in the database. This approach will obviously introduce some overhead to the normal operations on base tables. Ideally, snapshot refresh should be carried out without causing any impact on the normal base table operations. That is, insert, delete and update operations on the base table should be unaffected by the presence of one or more snapshots based on the base table. At least, this type of impact should be minimized.

The number of recorded modifications may, for some refresh methods and in some cases, become quite large — even exceeding the number of tuples in the base tables. As it is, not all modifications may be relevant to the snapshot. However, it is not always that easy to detect those modifications that are. Detecting the modifications relevant to one particular snapshot depends largely on the complexity of the select statement in the snapshot definition.

Snapshots defined as simple restrictions such as the one in Figure 1 on page 14, can be refreshed by considering only the changes to the base table. An insert into the base table
can be mapped to an insert into the snapshot table — provided the new record satisfies the snapshot restriction. Delete operations may be mapped similarly. Delete of a record from the base table can be mapped to a similar delete on the snapshot table — again providing that the record deleted satisfies the snapshot restriction. Finally, by treating an update as a delete followed by an insert, the mapping of updates become as for these two operations.

In cases where the select statement of the snapshot definition involves a join of two or more base tables, the mapping of the modifications to the snapshot table may become quite complex. This problem has been analyzed for a related topic — materialized views [BLAK 86]. The paper split the set of modifications as performed by one transaction into two sets; relevant and irrelevant updates. By detecting the irrelevant updates, re-evaluation can be avoided, or at least the number of tuples considered can be reduced.

The paper assumes that the modification set — being the database modifications performed by one transaction — can be detected for the base tables that are involved in the view definition. The net effect of a transaction on a base table can be represented by the set of tuples that have been inserted, and the set of tuples that have been deleted. Formally, given a base table \( R \) and a transaction \( \tau \), there exists sets of tuples \( I_\tau \) and \( D_\tau \) such that \( R, I_\tau, \) and \( D_\tau \) are disjoint, and \( \tau(R) = R \cup I_\tau - D_\tau \).

Updates to each base table performed by the one transaction are assumed identified in this manner. Then, an algorithm is given for combining these modifications if a view is defined as a join of two or more tables. The algorithm makes use of the combinatorial property of the equi-join operation when applied on the relevant and irrelevant updates of the modification set, e.g. a join of an irrelevant set with a table produces irrelevant entries. A detailed description may be found in the paper. The algorithm for join views proposed in [BLAK 86] does not always perform correctly, and an amendment is given in [HANS 87].

Not only join views have to be treated specially, projection views must in some cases be solved differently to selection views. The difficulty arises when the base table is exposed to delete operations and the view does not project the primary key. In this case, the view may include a multiplicity counter which records the number of base table tuples that contribute to the tuple in the view. An alternative solution is to include the primary key in the view. For a detailed description of this we refer to [BLAK 86].

The algorithms proposed for maintaining materialized views can also be applied to database snapshots. This is illustrated for a snapshot corresponding to a selection view, as follows: If we replace the base table \( R \) in the formal expression above, and assume a snapshot \( S \) instead, then the net effect of refreshing the snapshot after a modification period \( \tau \) — given that \( I_\tau \) and \( D_\tau \) are the set of tuples inserted and deleted, over the period \( \tau \), that qualify for the snapshot, and that the sets are disjoint — may be expressed as follows: \( \tau(S) = S \cup I_\tau - D_\tau \).

As for snapshots corresponding to projection views, an additional solution can be used. Instead of using a multiplicity counter, the number of occurrences can be calculated from the base table thus preventing deletion of a tuple from the snapshot when other identical values are encountered.
The issue of how to mark updates made to a base table for future detection so as to materialize the view, is not addressed in [BLAK 86], nor does it discuss how updates are detected for a sequence of transactions which will be the case in database snapshots. We will in the following — "Chapter 4. Differential Refresh Methods" and "Chapter 6. Extending Refresh for Snapshot Replication" — address these issues. As the problems related to combining modifications from several base tables are the same for database snapshots and materialized views, we restrict the discussion to refresh methods supporting snapshots based on a single base table.

The remainder of this chapter is organized as follows: First, we discuss how the snapshot mechanism can be supported by a general refresh operation in a distributed environment. Then we address the performance objectives for the refresh operation since efficiency is a major issue in the support of database snapshots. We then describe the main classes of snapshot refresh methods, and conclude by suggesting a spectrum of algorithms (strategies) that may have to be enforced by the snapshot refresh operation in order to support all types of snapshots efficiently. Finally, we discuss some of the operational aspects of database snapshots in a distributed environment.

3.2 The Refresh Operation

Snapshot refresh will be supported by a separate refresh operation. According to the semantics, snapshot refresh should reflect the current transaction consistent state of the database. By treating the refresh operation as an ordinary transaction, data as of a consistent state is retrieved from the database and the snapshot table will be left in a consistent state after completion of the transaction.

The refresh operation is invoked either by some event (e.g. a clock) or by a direct request from a user or an application program. In the latter case, the refresh may be one out of several requests issued within the current transaction. As the refresh operation may involve a series of database updates, it is recommended to issue the refresh as a single request within one transaction. An approach where the snapshot operation may be separated from the transaction in which it is initiated, and executed as a stand-alone transaction, may provide us with a better solution. Without loss of generality, we have treated the refresh operation as one transaction in the discussion that follows.

In "Chapter 2. Snapshot Definition and Semantics", we introduced a refresh manager to ensure that replicated snapshots are refreshed collectively. The refresh manager will act as a coordinator in the transaction refreshing the replicas belonging to one snapshot cluster. The mechanism appears to be useful in refresh of independent snapshots as well, especially in view of snapshot support within a distributed environment.

---

4 In the follow-up project of R*, the Starburst project at IBM San Jose [SCHW 86], some research along this line is being done. However, details of this have not been published. In the relational DBMS TechRa [KVAT 87], some operations are deferred until the end of a transaction so as to minimize the consequence of a user-invoked rollback. Details of this have not been published either.
In a distributed environment, the snapshot table will often reside at a site different from the base table(s). Given a single snapshot defined on one base table, then a refresh operation applied on the snapshot will proceed as follows:

First, the snapshot definition is retrieved from the system catalog at the snapshot site. Then the select-statement, defining the content of the snapshot, is shipped to the base table site, and used for selecting modifications made to the base table that apply to the snapshot. These will then be returned to the snapshot site where the snapshot table will be updated. Finally, the operation is completed by a 2-phase commitment protocol with the control at the snapshot site.

In a distributed system, a refresh operation, performed as described, is a distributed transaction consisting of local transactions at the base table and snapshot table site, see Figure 4. As shown in the figure, the refresh manager may take control of the distributed transaction and perform the 2-phase commitment protocol that completes the refresh operation. The refresh manager will thus supervise the update transaction at the snapshot site and the read transaction at the base table site.

If the snapshot is defined as a distributed query involving several base tables, then several base table sites may have to take part in the refresh operation and in the 2-phase commitment protocol coordinated by the refresh manager. In this case, several local read transactions — perhaps one for each base table referenced in the query — and one local update transaction are needed to fulfill the refresh operation.

In the case of replicated snapshots, the refresh manager is responsible for update of several snapshot replicas. Again, the snapshot may be defined on several base tables so that the refresh operation may involve one or several local read transactions as well as several update transactions — one per snapshot replica.

In Figure 4, the refresh operation is generalized. Three types of transactions take part in the operation in a distributed environment: i) A snapshot manager being the coordinator, ii) local read transactions selecting the modifications that apply to the snapshot, and iii) local write transactions updating the actual snapshot table(s). The figure shows the transactions as processes and not their allocation to sites. The figure may leave the
impression that data is transmitted through the refresh manager. This may not necessarily be true as will be explained below.

The refresh manager will, by default, reside at the site where the snapshot definition or the snapshot frame is created, i.e. their birth site. Thus, in the case of independent snapshots, the refresh manager resides at the snapshot site whereas in the replicated snapshot case, the refresh manager will reside at the site where the snapshot frame is defined.

The refresh manager plays the role as the coordinator in the refresh of independent snapshots. In the refresh of replicated snapshots, the role may be extended to collecting and distributing the refresh messages. This will be the case if the snapshot frame is defined as a join of several base tables residing at different sites. Here, the selection of modifications that apply to the snapshot replicas will involve tracing of base table modifications at more than one site. Then these modifications are combined using the algorithms proposed by [BLAK 86], before refresh messages are sent to the replica sites. If the replicated snapshot is based on one base table only, the collection can be done locally if the manager resides at the base table's site.

Note that for replicated snapshots, the performance of the refresh operation may be influenced by the choice of frame site. If the definition refers to one base table only, the snapshot frame should be defined at this site. If the definition involves several sites, the site holding the largest fragment of base tables is probably the best choice. Defining the snapshot frame at a site different from any of its base tables and snapshot replica sites is not a good choice from a performance point of view, so the system may dynamically choose a different one by using a strategy similar to DQP, cf. [CERI 84].

### 3.3 Performance Objectives

There are several factors to consider in calculating the overall performance impact of a database snapshot mechanism. Most of the factors stem from the cost introduced by the refresh operation when refreshing the content of a database snapshot. In a distributed environment where the snapshot table resides at a site different from the base table site, the shipment of data to the snapshot site is an added cost that may increase the cost of the refresh operation significantly.

At snapshot refresh, the modifications to the base table must somehow be reflected in the snapshot table. If we for a moment disregard the refresh of replicated snapshots, a refresh operation will most likely involve processing at the base table and the snapshot sites as well as transfer of refresh messages to the snapshot site.

Four cost factors that may contribute to the overall cost of a database snapshot mechanism are identified. These cost factors stem from the various steps involved in the maintenance of database snapshots, and are reflected in Figure 5.

First, there is the possible overhead connected to the "normal" modification activity on the base table. Operations on the base table such as insert, update and delete, must be identified for future refresh on the base table. We call this process **identify modification**.
Figure 5. Impact of Database Snapshot Mechanism: Steps involved in the maintenance of database snapshots. Note that the identify modification process takes place prior to the actual refresh operation.

Secondly, there is the cost of refresh processing at the site of the base table, i.e. finding the modifications appropriate and creating the refresh messages that need to be shipped to the snapshot site. We call this process refresh create.

Thirdly, there is the cost of communicating the refresh messages to the snapshot site. We use the term refresh messages for these.

Finally, there is the cost of refresh processing performed at the snapshot site, i.e. updating the snapshot by applying the appropriate operations on the snapshot table as denoted by the refresh messages received from the base table site. We name this process refresh update.

Note that the cost of the last two steps are largely proportional to the number of messages sent from the base table site. Some of the cost factors may not be relevant to all of the refresh methods. We will return to this when discussing differential refresh strategies in “Chapter 4. Differential Refresh Methods” and “Chapter 5. Refresh Method Analysis” where we discuss the cost of snapshot maintenance in detail.

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. We have therefore addressed both issues in the cost analysis provided in “Chapter 5. Refresh Method Analysis”. 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 snap-
shots defined on a table.

When performing the refresh operation on a database snapshot, only the three last steps
as depicted in Figure 5, are actually involved. The identify modification process is acti-
vated at the time a base table is modified, and thus takes place prior to the refresh op-
eration itself. When comparing the various refresh strategies, there may be cases in
which some of the processing cost can be amortized at off peak periods. Some of the
refresh create processing can be done prior to refresh if modifications are identified in a
manner that allows for this. We will return to this in “Chapter 5. Refresh Method
Analysis” where we discuss the consequence of relaxing the cost function by neglecting
some of the processing cost.

So far, we have only discussed refresh of single snapshots. In the case of snapshot rep-
lication, the refresh operation must either refresh a set of snapshots collectively (replic-
cated snapshots), or the operation is performed for several snapshots individually
(independent snapshots). The same cost factors as identified in Figure 5, apply to
maintenance of these snapshots, but the weight of each factor in the overall cost calcu-
lation is different.

In the single snapshot case discussed above, the cost of identifying modifications is taken
by that snapshot alone. In the snapshot replication case, this cost can be considered as
“shared” by the snapshots currently defined over the base table. Still, in the snapshot
replication case, a large proportion of the cost is associated with each snapshot (or
snapshot replica), such as the cost involved in updating the snapshot table and the cost
of communicating the refresh messages to the snapshot site. In “Chapter 6. Extending
Refresh for Snapshot Replication”, these issues are discussed in detail.

### 3.4 Full Refresh

The simplest implementation of database snapshots is to define a snapshot as an ordi-
ary table that is refreshed using a full refresh strategy. As the name suggests, the
snapshot is refreshed by creating the full content of the snapshot table at each refresh.
The new content of the snapshot is determined by executing the select statement asso-
ciated with the snapshot definition. This will produce a sequence of records which are
shipped to the snapshot site. At the snapshot site, the snapshot table is first cleared (i.e.
all its records deleted), and then the new content, received from the base table site, is
inserted into the snapshot table on a tuple basis. The process is thus similar to the one
involved in an “INSERT INTO table select-statement” in SQL.

In some database management systems such as DB2 [IBM 86], queries are compiled
into a sequence of lower level database operations and stored in an access module for
future execution of the same query. The select statement associated with the snapshot
may thus be stored as an access module in such systems. The snapshot definition stored
in the system catalog will then simply hold a reference to the corresponding access
module. At snapshot refresh, the access module is invoked. The access module may
perform the operations that are needed to refresh the content of the snapshot including
call to a remote access module that will handle the insert of records in the remote snapshot table. The original query defining the snapshot will also be kept in the access module in the event that recompilation may become necessary. In actual fact, this is how queries referencing data at more than one site are executed in System R* [LOHM 85].

The full refresh method has the advantage that it causes minimal impact on the base tables involved in the snapshot definition. Base table operations may simply proceed as before. The refresh method can therefore easily be implemented in any database system. The disadvantage of the method is that unless a significant portion of the base tables have been updated since the last refresh, the method will transmit many unchanged entries (since the full snapshot is sent each time). This is a major drawback in a distributed environment.

In the extreme case — which demonstrates the pessimistic approach taken by the full refresh strategy — the entire snapshot content is sent even if no modification has taken place. Obviously, this situation may be avoided by keeping some simple statistics on the base tables such as the time of the last update operation. Although this only helps for this special case, it is a very simple measures that may be useful to some applications.

3.5 Differential Refresh

The differential refresh method attempts to reduce the number of messages sent to the snapshot site, and thus the number of tuples that need to be changed in the snapshot table at refresh time, by detecting the modifications made to the base tables since the last refresh. If the snapshot is derived from a single base table, the modifications made to the base table since the last refresh can be detected easily. Then, only these are examined by the differential refresh method. Thus, modifications to a snapshot become a subset of the modifications to its base table.

The identification of this subset can be more or less accurate — depending on how modifications are marked by the identify modification process in Figure 5. With accuracy we mean a method that can extract only those modifications that are of relevance to the snapshot — in other words identifying the smallest subset needed. Some refresh methods are unable to determine whether a deleted tuple satisfied a particular snapshot restriction prior to deletion or not. However, extraneous identification of deletions can easily be discarded by the refresh update process at the snapshot site (as a delete of a non-existing tuple causes no harm).

The following three approaches support differential refresh of a snapshot:

1. The ASAP (As Soon As Possible) approach.
2. The logging approach.
3. The extended base table approach.

The ASAP approach passes changes onto the snapshot table as they occur at the base table. The method has two major weaknesses.
First, the method can not support snapshots that adhere to the general snapshot definition. An ASAP snapshot is continuously being updated as opposed to capturing the base table state as of a specific refresh time. The method can therefore only be used for special snapshot cases.

Secondly, the method will increase the normal processing cost of a base table considerably. Each update operation on the base table will have to pay for the cost of transmitting the modification to the snapshot site, plus the cost of updating the snapshot table. If the snapshot site is unavailable due to communication failure, the ASAP strategy may remedy this situation in two different ways. Either the base table change will have to be buffered safely so that the update can be added to the snapshot table when the system recovers from the communication failure, or the base table update operation will have to be rejected. In other words, the method has the same sort of performance penalty as transactions enforcing mutual consistency of replicated data. This was the problem we hoped to avoid by introducing snapshots.

In the logging approach, the changes to the base table are kept in a log. At snapshot refresh we only transmit modifications relevant to the snapshot table. The log may be organized in two fundamentally different ways — as a sequence of database operations submitted to the base table, or as an after image log containing the modified base table entries. Both of the logging methods can support snapshot refresh as of a given refresh time.

A log keeps the modifications submitted to a base table in a separate domain at the base table site. The modifications are added to the log as part of the identify modification process, see Figure 5. A logging method will for this reason impose some overhead to the normal operations on the base table. Unlike the ASAP strategy, most of the cost is associated with the refresh operation itself as will be explained below. The logging approach possesses another good property from a transaction management point of view: The base table does not have to be consulted after the modifications have been logged, i.e. the refresh operation can be carried out by referring to the log alone. We will return to this later in the chapter when we discuss the operational issues of snapshots.

A call sequence log may typically be organized as a sequence of SQL statements updating the base table. At refresh, the statements kept on the log are transferred to the snapshot site for execution on the snapshot table. It is essential that the statements are carried out on the snapshot table in the same order as submitted to the base table. The end result will otherwise be incorrect.

An after image log holding the modified entries can be processed similarly — only that entries need not be shipped to the snapshot site log if they do not satisfy the snapshot restriction. As in the call sequence log, the modifications will, in principle, have to be added to the snapshot table in the same order as they were submitted to the base table. We will return to this in “Chapter 4. Differential Refresh Methods” where we also shall see that some preprocessing may be done on the after image log so as to reduce the log size before the refresh operation is applied. Similar reduction can also be applied to the call sequence log for simple predicate cases.

The call sequence log has an advantage — when compared with the after image log — that the modifications may be recorded with a higher density as one SQL statement potentially involves several base table entries. On the other hand, the call sequence log
has a serious weakness caused by cursor concept in the SQL language. An update may be issued on a base table tuple identified by the current position of the cursor. In order to identify this modification correctly as an update statement in the call sequence log, the primary key of the referenced tuple (or an equally unique tuple identifier) must be inserted in the logged update statement. This substitution is illustrated by the following update example:

```
UPDATE Resorts
SET PriceLevel = 8
WHERE CURRENT OF CI;
```

Assuming that the cursor CI refers to the holiday resort Cannes in France, the update statement will be logged as:

```
UPDATE Resorts
SET PriceLevel = 8
WHERE Resort = 'Cannes'
AND Country = 'France';
```

A log organized as a sequence of SQL database operations thus requires the aid of a query transformer. SQL operations that modify the base table must be written to the log at the time the operation is submitted. In the case of cursor statements, a transformed update statement is added to the log. This transformation — as well as the logging operation — will add to the normal base table operation cost.

The transformation can be speeded up if the database system supports a precompilation scheme such as in DB2 [IBM 86]. The transformation must, however, be compiled into the access module handling the cursor modification statement. The approach has a disastrous side effect, as a database table no longer is independent of the snapshot mechanism. Introducing a snapshot referencing a database table for the first time, will cause recompilation of all access modules that implements statements on that table. Even if this may be a one time cost, the impact is too great on an operational database.

In a call sequence log one is therefore left with the option of substituting SQL cursor statements. The approach requires the SQL statements on a symbolic form which obviously reduces the applicability of SQL precompilation. Operations other than those directly affected by the snapshot may therefore be degraded performance wise. Although a call sequence log records the modifications at a higher level of abstraction, the method is excluded from the analysis in "Chapter 4. Differential Refresh Methods" for this reason.

In the third approach, the extended base table approach, modifications are identified in the base table directly instead of keeping them in a log. This strategy — which has been adopted by System R* [LIND 86] — is based on annotating (or expanding) the base table with columns so that the changes can be identified within the base table itself. The additional columns are maintained as part of the normal base table operations, which can be performed at little extra cost. Like the logging approach, snapshot refresh as of a given refresh time may be supported.

In R*, the refresh operation detects the modifications by scanning the base table, and only those changes that are of relevance to the snapshot table will be sent to the snap-
shot site. The update of the snapshot table is driven by the refresh messages received from the base table site. Nearly all the cost in maintaining snapshots is associated with the refresh operation. The overhead on the normal base table operations is virtually negligible although some cost involving the base table is introduced at refresh time. We will return to this in the discussion of the R* method in later chapters, and will from now refer to the method as the R* approach.

Differential refresh methods based on a type of after image log or an annotated base table, is the topic of "Chapter 4. Differential Refresh Methods". Common to both methods is that they provide efficient refresh of snapshots at the expense of some overhead on the base table operations. The base table operations insert, update and delete, must either write the modification to a log or maintain the additional columns of a base table. In any case, adding these function to the base table operations can be done quite easily in most database systems.

3.6 Implementing a Spectrum of Methods

The discussion in the previous section assumed differential refresh methods supporting snapshots that are based on single base tables. The modifications relevant to the snapshot table are found by detecting the changes made to the base table since the last refresh. When a snapshot is derived from several base tables, the changes needed to the snapshot table can not be detected that easily. A general snapshot mechanism can therefore not rely on a simple differential refresh strategy alone. Other refresh strategies must be included to complete the snapshot mechanism. One can envisage a system supporting a spectrum of refresh methods.

In refresh of snapshots derived from several base tables, the select statement associated with the snapshot definition must, in general, be re-evaluated in order to come up with a refreshed image of the snapshot table. As described at the beginning of this chapter, modifications made to several base tables can be merged so as to reflect modifications that are relevant to a snapshot based on these tables. Algorithms are given in [BLAK 86] and [HANS 87]. By applying these algorithms to the base tables and the select statement of the snapshot, the number of tuples that need to be considered can be reduced, and the number of refresh messages may be reduced significantly compared with a full refresh strategy.

As mentioned earlier, snapshots that include projections may in some cases be treated specially. Again, algorithms are provide in [BLAK 86] and [HANS 87].

Despite a range of good differential refresh strategies, generalized to handle the merge of modifications from several base tables, some cases may still be solved less costly by applying a non-incremental method, usually called a full refresh strategy. For example, if most of the base table tuples have been deleted, then a full refresh may be the least expensive. This suggests that the differential and full refresh may represent the ends of a spectrum of refresh methods.

The database administrator may thus choose from a set of refresh methods, depending on the update frequency of the base table, the composition of the modifications submitted, the base table size, and the selectivity of the snapshot restriction at hand. Even
the system optimizer may dynamically decide to change method, e.g. if the base table is empty, it may decide on using full refresh.

The rationale for such a spectrum of refresh methods will be better understood after we have compared the performance of the differential refresh methods in “Chapter 5. Refresh Method Analysis”.

3.7 Operational Issues

We now comment on some of the operational issues of database snapshots in a distributed environment.

3.7.1 Data Availability

The main objective for introducing snapshots in a distributed environment, is to increase the availability of data by allowing for data replication. A snapshot provides applications with a local copy of data and allows the “donator” to operate its own data without being conditioned by the snapshot site. This property is commonly referred to as site autonomy. Snapshots increase the degree of site independence by reducing the need for remote data access.

In a distributed database, site autonomy can only be partially achieved as the 2-phase commitment of a distributed transaction requires cooperation of the sites participating in the transaction. A site failure during 2-phase commitment may lead to a situation in which the distributed transaction can not be terminated. The resources held by the transaction can not be released, since the locks are required for providing atomicity of the distributed transaction. This situation may arise when a participant has agreed on committing the transaction, but before it has received the coordinator’s decision.

It therefore seems that loss of site autonomy can not be avoided at refresh of database snapshots. As the objective of snapshots is to increase availability of data, the snapshot mechanism should, ideally, not degrade the availability of the base tables “donating” the data. On the other hand, one may argue that snapshots reduce the need for distributed transactions, so overall, the risk of loosing site autonomy is decreased.

Some refresh methods do however allow for site autonomy even at snapshot refresh, and thus maintain the objective of site independence instead of weaken it. This can only be achieved if the local transaction(s) retrieving data from the base table(s) in the refresh operation are read-only, see Figure 4. In this case, the base table read transaction(s) can release their locks and forget the transaction immediately after having agreed to the commitment. In fact, a local read transaction is not troubled with whether the distributed transaction will commit or abort. This type of 2-phase-commitment, named presumed abort protocol, is proposed in [MOHA 83].

By using a presumed abort protocol, the base tables will be released even in the event of site failure during 2-phase-commitment, and the base tables are no longer conditioned by the snapshot site. The protocol used in this case is also called a 1.5PC protocol.
to reflect that the second phase is reduced compared with a full 2PC protocol.

A full refresh strategy fulfills the read-only requirement as base tables are only opened for read. As for the differential refresh methods, site autonomy may be lost at refresh time using the R* method whereas this can be avoided by the logging methods. The logging methods will in some cases include updates at the base table site(s). Unlike the R* method, base table are not exposed to modification and will therefore not be blocked. A site failure will therefore only cause minimal impact. We will return to this in “Chapter 4. Differential Refresh Methods” after having described the various differential refresh methods.

3.7.2 Recovery

During the refresh operation, the snapshot table is updated to reflect the current state of the base table(s). If the refresh operation is aborted, then the changes to the snapshot table must be undone, and the same applies for any alterations made at the base table site(s). A standard recovery log will take care of this.

Normally, a recovery log supports both the undo and the redo operation. In order to do so, the before image as well as the after image of modified data entries are kept in the log. Since a snapshot is a read-only copy of some portion of the database, its content may always be reproduced. The after images may therefore be omitted from the recovery log.

Furthermore, a recovery log is normally compressed before the archive version, often named database log, is produced. In the database log, before images are no longer needed. This implies that for snapshots, it is sufficient to log the initiation of the refresh operation in the database log.

3.7.3 Refresh Initiation

Snapshots defined with a refresh period are kept in a system refresh check list at the snapshot’s (or the snapshot frame’s) birth site, cf. “Chapter 2. Snapshot Definition and Semantics”. The list is used by the system to initiate the refresh operation when refresh of a particular snapshot is due.

Obviously, the refresh operation can not be performed if some of the sites involved in the snapshot are unavailable at the time the operation is initiated. Also, the initiation fails to take place if the snapshot site is down at the actual time. The refresh operation must then be invoked at a later stage.

The refresh check list can be designed to handle this situation by including all periodic snapshots defined at the site, and their next expected refresh time. If refresh initiation fails, then the system will detect this by comparing the expected next refresh time with the current time, i.e. the system checks for overdue refreshes.
After completing a refresh operation, the next expected refresh time is filled in for the snapshot in the check list. A periodic snapshot will thus always be kept on the list; either its refresh is due at the time recorded in the list, or the refresh is overdue. The list may also be used to assist recovery, as abort of the refresh operation may be treated as an initiation failure.
Chapter 4. Differential Refresh Methods

4.1 Introduction

The objective of differential refresh methods is to reduce refresh processing by updating the database snapshot in an incremental manner. Basically, this means detecting the changes relevant to the snapshot and submitting them to the snapshot table. This implies that we must be able to trace modifications made to the original data — i.e. the base table(s) — since the last refresh of the snapshot.

As discussed in “Chapter 3. Implementing Snapshots”, modifications relevant to the snapshot can be detected easily if we assume snapshots based on a single base table (i.e. no joins in the definition). We also discussed how changes to several base tables can be combined to support differential refresh of more complex snapshot definitions. In the following, we emphasize methods that support refresh of database snapshots defined as single base table selections, i.e. selection statements made up of selection clauses on one base table and projections that include the primary key of the base table.

The differential refresh methods discussed in this chapter identify the changes made to base tables on a tuple basis — either in the base table directly, or indirectly by recording the tuple modifications as log entries in a log. The following refresh methods are discussed:

- **Sequential Logging**, in which modifications are recorded in a log in the sequence submitted to the base table.

- **Condensed Logging**, in which modifications concerning one base table tuple are merged into one entry in the log.

- **The R* Method**, in which modified or new base table tuples are marked so as to distinguish them from unchanged entries of the base table.

The methods not only differ in the way modifications are identified for future refresh, but also in selecting modifications relevant to the snapshot table. As we shall see, the identification of the modification set relevant to the snapshot table may be more or less accurate.\(^5\)

\(^5\) The modification set to be applied on the snapshot table will never be selected too small. On the contrary, the set may for some of the methods contain extraneous changes, but the result


**BASE TABLE BEFORE MODIFICATION**

<table>
<thead>
<tr>
<th>Resort</th>
<th>Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beirut</td>
<td>Lebanon</td>
<td>6</td>
</tr>
<tr>
<td>Brighton</td>
<td>England</td>
<td>6</td>
</tr>
<tr>
<td>Cannes</td>
<td>France</td>
<td>7</td>
</tr>
<tr>
<td>Colombo</td>
<td>Sri Lanka</td>
<td>3</td>
</tr>
<tr>
<td>Florence</td>
<td>Italy</td>
<td>5</td>
</tr>
<tr>
<td>Tenerife</td>
<td>Spain</td>
<td>4</td>
</tr>
</tbody>
</table>

**SET OF MODIFICATIONS**

1. UPDATE Resorts SET PriceLevel = 8 WHERE Resort = 'Cannes';
2. INSERT INTO Resorts VALUES ('Crete', 'Greece', 3);
3. DELETE FROM Resorts WHERE Resort = 'Beirut';
4. INSERT INTO Resorts VALUES ('Miami', 'United States', 7);
5. UPDATE Resorts SET PriceLevel = 4 WHERE Resort = 'Crete';

**BASE TABLE AFTER MODIFICATION**

<table>
<thead>
<tr>
<th>Resort</th>
<th>Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brighton</td>
<td>England</td>
<td>6</td>
</tr>
<tr>
<td>Cannes</td>
<td>France</td>
<td>8</td>
</tr>
<tr>
<td>Colombo</td>
<td>Sri Lanka</td>
<td>3</td>
</tr>
<tr>
<td>Crete</td>
<td>Greece</td>
<td>4</td>
</tr>
<tr>
<td>Florence</td>
<td>Italy</td>
<td>5</td>
</tr>
<tr>
<td>Miami</td>
<td>United States</td>
<td>7</td>
</tr>
<tr>
<td>Tenerife</td>
<td>Spain</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 6. Example Base Table and Modification Set

Naturally, the discussion will lay emphasis on the accuracy of selecting modifications for shipment, see Figure 5 on page 30. This is mainly due to the significant overhead associated with communicating refresh messages in a wide area network.

First we will describe the two logging approaches proposed in this thesis. We discuss realization of the methods, and alternative refresh algorithms. Next, we describe the R* approach. By relating the logging approaches to the R* approach, we see that some enhancement — in terms of refresh messages sent — can be made in each approach. The enhancements will reduce the number of refresh messages sent to the snapshot table. On the other hand, the enhancement may increase the overall cost of snapshot refresh.

of submitting them to the snapshot table will always provide a snapshot in accordance with its snapshot definition.
First, we propose two enhancements to the basic R* method. We then describe an enhanced version of logging, named enhanced log for short.

In order to illustrate the various differential refresh methods and how they differentiate, the holiday resorts table defined in Figure 1 on page 14, has been used as an example throughout the discussion in this chapter. We assume that the base table has been exposed to modification since the time snapshot $S_1$ was defined on the base table. Figure 6 displays the base table before modification, the set of modifications submitted to the base table expressed as SQL statements, and the base table after modification.

The example is chosen so as to emphasize the difference between the various refresh methods. By only referring to the examples, some of the methods may seem significantly better than others. This is not necessarily the case as we shall see in the cost analysis given in "Chapter 5. Refresh Method Analysis". The analysis is performed under various update frequencies on the base table and for various modification compositions. In the analysis, the processing cost is considered, as well as the message delivery cost. We also discuss the overall consequence of changing the cost function, e.g. by neglecting the processing cost partly or totally.

Note that in the discussion that follows, we describe the refresh methods within the scope of a single snapshot per base table. In "Chapter 6. Extending Refresh for Snapshot Replication", we will see how the logging approaches can be extended to support replicated as well as several independent snapshots.

The work presented here has been done along with the development of the distributed relational database system MIMER* [KÄHL 85]. Among the characteristics of MIMER [MIME 85] is the enforcement of a primary key, optimistic concurrency control for transaction handling, and portability between dissimilar computer systems. Some of the refresh methods proposed in the following have been influenced by this ongoing development. The various methods are therefore discussed in light of this system, although the discussion as such does not address this database system alone.
4.2 Sequential Logging

An early paper 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 therefore requires a sequential scan of the log. Also, probably only a subset of the tables act as base tables for snapshots. Since it is necessary to keep the log on-line for snapshot refresh, information concerning modification to database tables not involved in any snapshot definition would have to be kept on-line much longer than necessary.

An alternative representation of the database log, suitable for differential snapshot refresh, may be found. For instance, the database log can be processed by a pending process which extracts the changes to the base tables and thus maintains two separate logs. This will affect the system recovery process. As the objectives of database log and a log for snapshot refresh are different, this type of integration may cause conflicts. As we shall see in “Chapter 6. Extending Refresh for Snapshot Replication”, it may become necessary to record extra information in the log to support several independent snapshots on one base table.

A logging variant similar to this 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 defined as replicated (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, a log needs only to be kept to the next snapshot refresh on its base table.

The refresh of database snapshots is similar to the maintenance of replicated data using the deferred update policy. There is one major conceptual difference. A snapshot is a read-only copy of some subset of a base table whereas a deferred replica is a copy of a full base table. A deferred update policy is for this reason not selective in the sense that some updates to a base table are not relevant to the replica. In a differential snapshot refresh method, this property must be included in the refresh operation.

Refresh of snapshots based on a sequential log can be done in two different ways:

- **Basic refresh**, in which the sequential log is scanned once, and modifications qualifying the snapshot restriction are sent to the snapshot table.

- **Sort merge refresh**, in which the sequential log is sorted on the ascending base table primary key as the first step of the refresh create process. Then, refresh messages are produced, and only changes relevant to the snapshot table are sent.

In both refresh strategies, the log is erased after refresh.
<table>
<thead>
<tr>
<th>Modif. Type</th>
<th>Resort</th>
<th>Value Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPD</td>
<td>Cannes</td>
<td>France</td>
<td>8</td>
</tr>
<tr>
<td>INS</td>
<td>Crete</td>
<td>Greece</td>
<td>3</td>
</tr>
<tr>
<td>DEL</td>
<td>Beirut</td>
<td>Lebanon</td>
<td></td>
</tr>
<tr>
<td>INS</td>
<td>Miami</td>
<td>United States</td>
<td>7</td>
</tr>
<tr>
<td>UPD</td>
<td>Crete</td>
<td>Greece</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 7. **Sequential After Image Log**: An entry records the after image of a modified base table tuple. Each entry is given a label reflecting the type of modification, i.e. INS for insert, UPD for update, and DEL for delete.

### 4.2.1 Basic Refresh Strategy

The sequential logging approach does impose some overhead on the normal processing of a base table. In Figure 5 on page 30, we named this process **identify modification**. 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. The accuracy of the refresh create process increases if the log includes before images of changed base table tuples.

Figure 7 gives an example of a sequential change log with after images only. The log is organized as a table. The example shows the log entries corresponding to the set of modifications listed in Figure 6. The modifications are recorded as follows:

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. In Figure 7, symbolic labels have been used as opposed to a numeric coding scheme which is more likely in an implementation situation.

A sequential log records all modifications made to a base table since the last refresh. Snapshot refresh can therefore be carried out quite easily by scanning the sequential log. For each log entry, a refresh message is sent to the snapshot site if the log entry satisfies the snapshot restriction.

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. This is shown in Figure 8 which gives a pseudo code representation of the **refresh create** process at the base table site.

Note that delete log entries are always sent in their present form, and that each message will cause delete of the snapshot tuple denoted by the value of the log entry. If no such tuple exists in the snapshot table, the delete operation will obviously fail, though causing no harm to the snapshot table.
RefreshCreate( SeqLog, SnapRestrict, Channel );
begin
Open( SeqLog, 'Seq', 'Read' );
Open( Channel, 'Send' );
Get( SeqLog, LogRec );
repeat
begin
  case LogRec↑.ModifType of
    INS: if SnapRestrict( LogRec↑.Value ) then Xmit( Channel, LogRec );
    UPD: if SnapRestrict( LogRec↑.Value ) then
      begin
        (* Qualified log entry, may cause insert or update *)
        Xmit( Channel, LogRec );
      end;
    else
      begin
        (* Unqualified log entry, may cause delete or is superfluous *)
        Xmit( Channel, <'DEL', LogRec↑.PKey > );
      end;
    DEL: Xmit( Channel, LogRec )
  end;
end;
until Eof( SeqLog );
Close( Channel );
Close( SeqLog );
end;

Figure 8. Refresh Create for Sequential After Image Log: The value of a tuple after modification is recorded in a log entry, named Value in the algorithm. PKey is the primary key of a tuple.

Update log entries are subject to some modification as illustrated by the pseudo code in Figure 8. In the case that the update entry satisfies the snapshot restriction, the log entry is sent as it is. Each update refresh message will cause insertion of a new tuple to the snapshot table, if not present in the snapshot table already. If an entry exists, the present entry is updated. In the case that the update entry is unqualified, a delete refresh message is sent so as to remove the corresponding tuple from the snapshot table, cf. the discussion of deletes above.

Not knowing the before image of the modified tuples — and thus their presence in the snapshot table — causes what we call inaccurate selection of modification entries. For example, the algorithm depicted in Figure 8 sends delete messages of tuple not present in the snapshot table — either caused by a delete log entry, or an unqualified update entry. Consequently, the refresh update process at the snapshot site must be prepared to handle modifications to tuples not present in the snapshot table.
RefreshUpdate( Snapshot, Channel );

begin
  Open( Channel, 'Receive' );
  Receive( Channel, LogRec );
  repeat
    begin
      case LogRec↑.ModifType of
        INS: INSERT INTO Snapshot
              VALUES ( LogRec↑.Value );
        UPD: if EXISTS( SELECT *
                        FROM   Snapshot
                        WHERE PKey = LogRec↑.PKey )
            then
              (* Existing tuple updated * )
              UPDATE Snapshot
              SET   Value = LogRec↑.Value
              WHERE PKey = LogRec↑.PKey
            else
              (* New tuple inserted * )
              INSERT INTO Snapshot
              VALUES ( LogRec↑.Value );
        DEL: DELETE FROM Snapshot
              WHERE PKey = LogRec↑.PKey
      end;
      Receive( Channel, LogRec );
    end
    until EoF( Channel );
  Close( Channel );
end;

Figure 9. Refresh Update for Sequential After Image Log: The primary key of tuples and log entries are named PKey. Note that the use of EXISTS is not standard.

Extraneous deletes do however not result in an incorrect refresh of the snapshot, they merely cause an unnecessary overhead. Figure 9 defines the refresh update process corresponding to the refresh create process in Figure 8. Note that the outcome of the DELETE statement is none when the tuple is not present in the snapshot table. Also note that the logic for handling update refresh messages first examines the snapshot table. If a tuple with the given key is found, the tuple is updated. Otherwise, a new tuple is inserted.
<table>
<thead>
<tr>
<th>Modif. Type</th>
<th>Resort</th>
<th>Value Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEL</td>
<td>Cannes</td>
<td>France</td>
<td>3</td>
</tr>
<tr>
<td>INS</td>
<td>Crete</td>
<td>Greece</td>
<td>4</td>
</tr>
<tr>
<td>DEL</td>
<td>Beirut</td>
<td>Lebanon</td>
<td>7</td>
</tr>
<tr>
<td>UPD</td>
<td>Crete</td>
<td>Greece</td>
<td>3</td>
</tr>
</tbody>
</table>

**Figure 10. Refresh Messages to Snapshot Table:** Messages sent to snapshot $S_1$ based on the sequential log shown in Figure 7.

To demonstrate the refresh operation based on a sequential log with after images only, we use the example base table and the set of modifications listed in Figure 6. A refresh of snapshot $S_1$ — with the snapshot restriction $PriceLevel < 7$ — will produce the set of refresh messages listed in Figure 10. As can be seen from the figure, Cannes is included in the refresh messages even though the tuple does not qualify 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 11 shows the revised sequential log.

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 messages depending on which out of the before and after images 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.

<table>
<thead>
<tr>
<th>Modif. Type</th>
<th>Resort</th>
<th>Value Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPD</td>
<td>Cannes</td>
<td>France</td>
<td>7</td>
</tr>
<tr>
<td>-</td>
<td>Cannes</td>
<td>France</td>
<td>8</td>
</tr>
<tr>
<td>INS</td>
<td>Crete</td>
<td>Greece</td>
<td>3</td>
</tr>
<tr>
<td>DEL</td>
<td>Beirut</td>
<td>Lebanon</td>
<td>6</td>
</tr>
<tr>
<td>INS</td>
<td>Miami</td>
<td>United States</td>
<td>7</td>
</tr>
<tr>
<td>UPD</td>
<td>Crete</td>
<td>Greece</td>
<td>3</td>
</tr>
<tr>
<td>-</td>
<td>Crete</td>
<td>Greece</td>
<td>4</td>
</tr>
</tbody>
</table>

**Figure 11. Sequential Before and After Image Log:** An update entry holds the before image as well as the after image of the tuple modified. Alternatively, update entries can be stored as separate delete and insert entries.
RefreshCreate( SeqLog, SnapRestrict, Channel );
begin
Open( SeqLog, 'Seq', 'Read' );
Open( Channel, 'Send' );
Get( SeqLog, LogRec );
repeat
with LogRec do
begin
case ModifType of
  INS, DEL: if SnapRestrict( Value ) then Xmit( Channel, LogRec );
  UPD: begin
    if SnapRestrict( BImage ) and SnapRestrict( AImage )
    then Xmit( Channel, < 'UPD', AImage > );
    if SnapRestrict( BImage ) and not SnapRestrict( AImage )
    then Xmit( Channel, < 'DEL', BImage > );
    if not SnapRestrict( BImage ) and SnapRestrict( AImage )
    then Xmit( Channel, < 'INS', AImage > );
  end
end
Get( SeqLog, LogRec );
end
until Eof( SeqLog );
Close( Channel );
Close( SeqLog );
end;

Figure 12. Refresh Create for Sequential Before and After Image Log: The before image of an update entry is assumed named BImage, and the after image AImage. The order in which the tests are applied in the pseudo code is not necessarily efficient.

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

Figure 12 is a pseudo code representation of the refresh create process at the base table site for the revised sequential log. It shows how the create process only selects modifications that satisfy the snapshot restriction. The corresponding refresh update process is given in Figure 13. Note that all modifications now refer to tuples present in the snapshot table. That is, each modification submitted to the snapshot table will refer to a tuple in the snapshot table — either existing prior to refresh, or a tuple that has been inserted during the refresh update process.
RefreshUpdate( Snapshot, Channel );
begin
  Open( Channel, 'Receive' );
  Receive( Channel, LogRec );
  repeat
    begin
      case LogRec↑.ModifType of
        INS: INSERT INTO Snapshot
             VALUES ( LogRec↑.Value );
        UPD: UPDATE Snapshot
             SET     Value = LogRec↑.Value
                     WHERE  PKey = LogRec↑.PKey;
        DEL: DELETE FROM Snapshot
             WHERE  PKey = LogRec↑.PKey
      end;
      Receive( Channel, LogRec );
    end
    until Eof( Channel );
  Close( Channel );
end;

Figure 13. Refresh Update for Sequential Before and After Image Log

Figure 14 shows the refresh messages sent in the revised approach. The number of refresh messages may be greatly reduced compared with an after image log, especially in the case of restrictive snapshots. On the other hand, a sequential log recording both before and after images requires more storage.

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 inserted in the base table and later updated may therefore cause just as many update messages. Ideally, only one insert message is needed. For example, Crete is inserted first with PriceLevel 3, and is later changed to 4. In Figure 14, two messages are sent whereas only one insert reflecting the latest value is necessary. Similarly, updates followed by delete should result in one delete message, and insert followed by delete should cause no message.

<table>
<thead>
<tr>
<th>Modif. Type</th>
<th>Resort</th>
<th>Value Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>INS</td>
<td>Crete</td>
<td>Greece</td>
<td>3</td>
</tr>
<tr>
<td>DEL</td>
<td>Beirut</td>
<td>Lebanon</td>
<td></td>
</tr>
<tr>
<td>UPD</td>
<td>Crete</td>
<td>Greece</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 14. Refresh Messages to Snapshot Table (Revised): Messages sent to snapshot $S_1$ based on the revised sequential log shown in Figure 11.
4.2.2 Sort Merge Refresh

One way of avoiding the shipment of unnecessary refresh messages to the snapshot table is to sort the sequential log tuple-wise at refresh time. The sort operation will become part of the refresh create process. The log entries are sorted on some unique tuple identifier — e.g. the primary key of the base table — while preserving the log order for each tuple. The result is equivalent to a sequential logging scheme applied on a tuple basis, i.e. a sequential log per base table tuple.

For each tuple that has undergone one or several changes, we may now find the outcome of the modification history and whether the tuple was qualified by the snapshot to begin with or not. Thus we may avoid the shipment of intermediate changes, and a final change is only sent if of relevance to the snapshot. If we deal with updates only, the change history for one tuple can be found by taking the before image of the first modification, and the after image of the last modification recorded for the tuple. In the case that the first recorded entry is an insert, or the last is a delete, the result of the modification history is found by applying a set of rules. We will return to this in the next section describing the condensed logging approach where we formalize the merging of log entries.

By sorting the log entries in a fashion just described, significant reductions in the number of refresh messages can be obtained. We will return to this in “Chapter 5. Refresh Method Analysis”. Not only will the number of refresh messages be reduced compared with the basic refresh strategy, the cost of updating the snapshot table — see Figure 13 — will decrease as the snapshot table will be updated in primary key order. On the other hand, the cost of sorting the log may outweigh this reduction. By using the replacement selection algorithm described in [KNUT 73] and a pattern merge algorithm — i.e. a sort merge strategy — sorting of the log may prove worthwhile. Again, we will return to this in “Chapter 5. Refresh Method Analysis”.

Before ending this section, we have two comments to add to the sequential logging approach:

First, a snapshot mechanism based on a sequential logging scheme can be implemented using existing database system primitives. Adding the modifications to a sequential log is a simple operation. Obviously, the mechanism introduces some overhead to the normal processing of base tables as well as an increase in storage consumption. On the other hand, there is no need to consult the base tables at snapshot refresh. The cost of refresh is in other words dependent on the length of the sequential log which implies that the mechanism may perform well as long as the log is of moderate size.

Secondly, the sort merge strategy may easily be adapted to the sequential logging scheme. A sort merge strategy is often supported by the database management system already, e.g. to assist initial load of tables.
4.3 Condensed Logging

The number of messages sent at refresh 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 as discussed for the sequential log in the previous section, the tuple order may be preserved during normal processing of the base table. We then arrive at the condensed log approach.

The condensed log keeps only one entry for a modified or new base table tuple. This implies that the condensed log must rely on an access method providing direct access. Since the log is continuously increasing in size, the access method must also enable dynamic growth. The following two alternatives are discussed for a condensed log:

- Condensed log based on B-tree.
- Condensed log based on linear hashing.

In the following we have described the condensed logging scheme as being based on B-trees, although most of the description applies to the condensed logging scheme in general. Furthermore, we relate the logging scheme to the sequential logging approach discussed in the previous section. We refer to the basic refresh strategy of sequential logging unless otherwise stated.

4.3.1 Condensed Log Based on B-tree.

By organizing the condensed log as a B-tree, we obtain a tuple ordering of the log. The log may be ordered on some unique tuple identifier, e.g. the base table’s primary key. Each entry records the net effect of modifications submitted to a base table tuple since the last refresh. The entry may either contain both before and after images, or the before image only. In the latter case, a pointer to after images in the base table is used. The size of the modification log is thus kept down to a minimum, and the intermediate write and read of the full log is eliminated in comparison with a sequential log.

The rules for adding a modification of a tuple to the condensed log, are as given in Figure 15. If no modification entry is found for the tuple, the modification is saved as it is, i.e. this is the first modification of the tuple since the last snapshot refresh. If a log entry already exists for the modified tuple, the modification is merged with the stored entry as follows:

An insert modification will be stored as an update, since the existing entry must be a delete entry for semantic reasons. Inserting a tuple that already exists is considered as an illegal operation and is prevented by the insert operation on the base table.
Figure 15. Merge Rules for Modifications: A modification entry is merged with an entry stored in the condensed log if the tuple has been modified previous to the current modification.

An update modification is merged with an update into an update entry, whereas it is merged with an insert entry into an insert entry. Updating a non-existing entry is an illegal operation.

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. Deleting a non-existing tuple is illegal.

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 therefore 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 (since the last refresh). This is taken care of when merging a new entry with a stored one. Also, merging a delete entry with a stored update entry will ensure that the before image of the update entry is kept.

Figure 16 shows a condensed version of the previous sequential log (cf. Figure 11). The condensed log is here organized as a separate log table containing the information of each modification. As the example shows, only an insert entry is recorded for Crete since the update is merged with the inserted entry. The one update entry for Cannes shows the before and after image of the tuple. Note that the log entries are stored in 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,

<table>
<thead>
<tr>
<th>Modif. Type</th>
<th>Resort</th>
<th>Value</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEL</td>
<td>Beirut</td>
<td>Lebanon</td>
<td>6</td>
</tr>
<tr>
<td>upd</td>
<td>Cannes</td>
<td>France</td>
<td>7</td>
</tr>
<tr>
<td>-</td>
<td>Cannes</td>
<td>France</td>
<td>8</td>
</tr>
<tr>
<td>ins</td>
<td>Crete</td>
<td>Greece</td>
<td>4</td>
</tr>
<tr>
<td>ins</td>
<td>Miami</td>
<td>United States</td>
<td>7</td>
</tr>
</tbody>
</table>

Figure 16. Condensed Log: The log table is ordered on primary key. Note that the two entries for Cannes represent an update.
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. The refresh create process is thus identical to the one shown in Figure 12 on page 47, except that the messages will be sent in key order of the condensed log.

Unlike the basic sequential approach, extraneous refresh messages will be avoided as a modified tuple has one recording only. Returning to the example as shown in Figure 14 on page 48, 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.

The refresh update process will only change tuples already present in the snapshot table, or add new tuples to the snapshot table. The process is carried out in the same way as in the revised sequential log, see Figure 13 on page 48. Because of the difference in nature of the two logging methods, there are two major differences in the way the snapshot table is actually updated: In the sequential log using the basic refresh strategy, the snapshot table is updated in a "random" fashion and a tuple may be updated several times. In the condensed log, the snapshot table is modified in a "sequential" fashion and a tuple is therefore updated once. The refresh update process can therefore be carried out with less cost in the condensed logging scheme.

A reduction in the cost of refresh update and shipping of refresh messages in a condensed logging scheme is achieved at the expense of added normal processing overhead. The log order of the condensed log must be preserved, as well as the condensed property of the log. This cost will then outweigh the benefits for cases where the number of modifications are small, i.e. the condensed logging scheme may first be profitable for large modification sets.

In the examples above, the condensed log is stored as a separate log table. The refresh operation need therefore only consult the log at snapshot refresh (and not the base table). Alternatively, the logging of the modifications can be done in much the same way as the database system maintains index tables. With this approach, the base tables will be consulted at refresh which may degrade the availability of the base tables as explained in "Chapter 3. Implementing Snapshots".

The logging operation introduces an extra working load on the normal base table operations. Large savings may be obtained if the logging operations can be merged with other operations during transaction management and still provide the refresh operation with a "clean" log table. In MIMER [MIME 85], a special write set is kept for optimistic concurrency control in addition to the normal recovery 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 the MIMER* prototype [KÄHL 85], the write set will be implemented as a condensed log, as the problem of locating previously written or modified tuples is similar to
the problem of locating modifications on a log. Also, the merging of updates will provide one updated image for each tuple, which simplifies the read operation. Write sets can therefore, by keeping them after transaction commitment, be used for a snapshot change log and will thus moderate the increase in normal processing overhead.

4.3.2 Condensed Log Based on Linear Hashing

One of the advantages using B-trees for a condensed log, is that the log can be stored using the same access method as used for ordinary base tables. B-tree, or a variant thereof, is the default access method in several relational database management systems. The disadvantage is that the condensed log introduces a large logging overhead without providing a corresponding benefit. Much of the effort is put into preserving the order of the condensed log, although this property is only taken advantage of once — when scanning the log for snapshot refresh.

An order preserving logging strategy for the condensed log is not strictly necessary as long as we maintain the log history for each updated base table tuple. The condensed property of the log can be maintained just as easily using a hash based access method for the log table. In fact, this strategy will add a smaller overhead to the normal processing cost as we avoid maintenance of an index structure.

The hash based method must, however, provide a dynamic storage structure that can grow with an increasing set of modifications. Such an access method is known as linear hashing or dynamic hashing [LITW 80].

The principles for adding and merging entries on the condensed log using this access method, are the same as discussed above. The refresh create process will also perform as before. The log is scanned sequentially, and refresh messages will be produced as before. The algorithm for the refresh update process remains the same although there is one significant difference in the way changes are submitted to the snapshot table. The snapshot table is updated in a "random" fashion, as in the sequential logging case. This may not be quite as efficient from the refresh update process point of view. Overall, the snapshot mechanism may benefit from this strategy. We return to this in "Chapter 5. Refresh Method Analysis".
4.4 The R* Method

An algorithm [LIND 86] supporting differential refresh of single base table snapshots, has been implemented in the experimental distributed database system R*. The objective of the R* approach is to minimize normal processing overhead, as well as the number of messages sent at refresh.

In System R*, all tuples are identified (throughout their lifetime) with a linearly ordered tuple identifier (address). This characteristic is used to detect changes in the base table.

The relational scheme of a base table is extended with two extra columns (e.g. by issuing an ALTER TABLE statement on the base table). One column contains the address of the previous tuple in the table, PrevAddr. The other field, TimeStamp, contains a timestamp that reflects the time when the tuple was last updated. Figure 17 shows a base table with the columns added. The example illustrates the same base table before and after states as depicted in Figure 6 on page 40.

The columns added to the base table impose almost no impact on the normal processing of the base table. Tuples are inserted, updated or deleted as usual. However, when a tuple is inserted, the value of the two extra fields is set to NULL. For updates, only the value of the TimeStamp is set to NULL, the PrevAddr field is left unchanged. In the case of delete, the base table tuple is simply deleted. Later, at refresh time, the correct value of the extra fields will be filled in.

Modifications to a base table prior to a refresh may now be identified by scanning the table — provided the previous refresh left all tuples with a correct pointing address field and non-NULL timestamp. The modifications since the last refresh are detected as follows:

Inserts are identified by a NULL value in the PrevAddr field. The TimeStamp field is also NULL.

Updates are identified by a non-NULL value in the PrevAddr field, and a NULL value in the TimeStamp field.

Deletions are not detected quite as easily. In order to identify a deleted tuple, the content of the PrevAddr field must be examined. Address fields pointing to an empty address (i.e. an address holding no tuple), or pointing to a newly inserted tuple, signals that one or more deletions have taken place in the region between the given PrevAddr and the address of the tuple considered. The algorithm is unable to determine exactly which tuples that were deleted, but detects "neighbouring" deletes as one delete region.

At refresh time, the address fields of the base table are corrected, and timestamp fields containing NULL are filled in with the refresh time. This process is named base table fix up. To a snapshot, it is sufficient to use the refresh time to notify the change in the base table rather than the time the change actually took place. The snapshot is only interested in the changes made to the base table since the last refresh time and up to the current time. The present refresh time can thus be used to distinguish "new" changes from the "old" ones.
BASE TABLE BEFORE MODIFICATION

<table>
<thead>
<tr>
<th>Addr</th>
<th>Prev Addr</th>
<th>Time Stamp</th>
<th>Resort</th>
<th>Value Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>nil</td>
<td>0:00</td>
<td>Beirut</td>
<td>Lebanon</td>
<td>6</td>
</tr>
<tr>
<td>3:</td>
<td>1</td>
<td>0:00</td>
<td>Brighton</td>
<td>England</td>
<td>6</td>
</tr>
<tr>
<td>5:</td>
<td>3</td>
<td>0:00</td>
<td>Cannes</td>
<td>France</td>
<td>7</td>
</tr>
<tr>
<td>7:</td>
<td>5</td>
<td>0:00</td>
<td>Colombo</td>
<td>Sri Lanka</td>
<td>3</td>
</tr>
<tr>
<td>9:</td>
<td>7</td>
<td>0:00</td>
<td>Florence</td>
<td>Italy</td>
<td>5</td>
</tr>
<tr>
<td>11:</td>
<td>9</td>
<td>0:00</td>
<td>Tenerife</td>
<td>Spain</td>
<td>4</td>
</tr>
</tbody>
</table>

BASE TABLE AFTER MODIFICATION

<table>
<thead>
<tr>
<th>Addr</th>
<th>Prev Addr</th>
<th>Time Stamp</th>
<th>Resort</th>
<th>Value Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td>1</td>
<td>0:00</td>
<td>Brighton</td>
<td>England</td>
<td>6</td>
</tr>
<tr>
<td>5:</td>
<td>3</td>
<td>NULL</td>
<td>Cannes</td>
<td>France</td>
<td>8</td>
</tr>
<tr>
<td>7:</td>
<td>5</td>
<td>0:00</td>
<td>Colombo</td>
<td>Sri Lanka</td>
<td>3</td>
</tr>
<tr>
<td>8:</td>
<td>NULL</td>
<td>NULL</td>
<td>Crete</td>
<td>Greece</td>
<td>4</td>
</tr>
<tr>
<td>9:</td>
<td>7</td>
<td>0:00</td>
<td>Florence</td>
<td>Italy</td>
<td>5</td>
</tr>
<tr>
<td>10:</td>
<td>NULL</td>
<td>NULL</td>
<td>Miami</td>
<td>United States</td>
<td>7</td>
</tr>
<tr>
<td>11:</td>
<td>9</td>
<td>0:00</td>
<td>Tenerife</td>
<td>Spain</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 17. Annotated Base Table: The base table is extended with two columns recording the changes made to the table.

The following fixup operations are performed on the base table during the scan:

Inserted tuples must be given a PrevAddr that is the address of the previous tuple in the base table. The TimeStamp is set to the current time which is the refresh time.

Updated tuples have only the TimeStamp set to the current time.

Deleted entries are no longer present, but still cause some fixup operations. A tuple that follows a deleted tuple will need to have the PrevAddr field adjusted. For example, tuple at address 3 in Figure 17 follows a delete, and the PrevAddr must be set to nil. The TimeStamp is also changed to the current time. This is done to notify that previous tuples have been deleted. A tuple following the deleted region is thus treated as an updated tuple.

Similar corrections are made to tuples that follow tuples inserted since the last refresh. For example, the tuples at addresses 9 and 11 will get their PrevAddr changed. The TimeStamp is not updated as the insert can be detected from the new record.

The result of applying the base table fixup on our example base table at time 9:00 is shown in Figure 18. Note that new timestamps have been given to the Brighton tuple.
**BASE TABLE AFTER REFRESH AT 9:00**

<table>
<thead>
<tr>
<th>Addr</th>
<th>Prev Addr</th>
<th>Time Stamp</th>
<th>Resort</th>
<th>Value Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>nil</td>
<td>9:00</td>
<td>Brighton</td>
<td>England</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>9:00</td>
<td>Cannes</td>
<td>France</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>0:00</td>
<td>Colombo</td>
<td>Sri Lanka</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>9:00</td>
<td>Crete</td>
<td>Greece</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>0:00</td>
<td>Florence</td>
<td>Italy</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>9:00</td>
<td>Miami</td>
<td>United States</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>0:00</td>
<td>Tenerife</td>
<td>Spain</td>
<td>4</td>
</tr>
</tbody>
</table>

**Figure 18. Base Table Fixup:** NULL values are replaced with real values.

as it follows a deleted entry. Also note that the new timestamps emphasize the latest changes made to the base table.

Before going into details as to how the refresh operation is performed, we describe how snapshot tables are organized. In the R* method, snapshot tables are also extended. Unlike base tables, only a BaseAddr column is added to the snapshot table. The column is used to correlate the snapshot and the base table entries. The BaseAddr keeps the address of the corresponding tuple in the base table, i.e. its “home” address. As an example, the snapshot $S_1$ — at the time of definition — is shown in Figure 19. Note that the tuple addresses in the snapshot table is a local domain which has nothing to do with the BaseAddr values.

The address of base table tuples plays an important part in creating the refresh messages that are shipped to the snapshot table. Basically, each refresh message carries the value of the modified tuple — the tuple either being updated or inserted. In addition, each refresh message contains two addresses; BaseAddr and PrevAddr. The BaseAddr is the “home” address of the tuple being sent. The PrevAddr is the address of the previous tuple in the base table that also satisfies the snapshot restriction. Thus, each refresh message informs the snapshot of its previous qualified tuple. As refresh messages are sent in base table order, any tuple that may come between the PrevAddr and the one sent, no longer qualifies and can therefore be removed from the snapshot table by the refresh update process.

The refresh create process will perform the base table fixup and create the refresh messages during one scan of the base table. Each tuple in the base table is evaluated with respect to the snapshot restriction. Combining this with the detection of the base table modification allows the algorithm to send the information necessary to perform the actual refresh processing at the snapshot site. Figure 20 gives a pseudo code representation of the refresh create process. The algorithm is taken from [LIND 86].

In the algorithm defined in Figure 20, the base table is opened for sequential scan. During this scan, each tuple is given a corrected timestamp and previous address, before the snapshot restriction is applied. In the case a tuple satisfies the restriction, a refresh message is sent if the tuple itself has been modified, or if the tuple follows a deleted region.
**SNAPSHOT TABLE AT CREATE TIME**

<table>
<thead>
<tr>
<th>Addr</th>
<th>Base Addr</th>
<th>Resort</th>
<th>Value Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>1</td>
<td>Beirut</td>
<td>Lebanon</td>
<td>6</td>
</tr>
<tr>
<td>3:</td>
<td>3</td>
<td>Brighton</td>
<td>England</td>
<td>6</td>
</tr>
<tr>
<td>5:</td>
<td>7</td>
<td>Colombo</td>
<td>Sri Lanka</td>
<td>3</td>
</tr>
<tr>
<td>7:</td>
<td>9</td>
<td>Florence</td>
<td>Italy</td>
<td>5</td>
</tr>
<tr>
<td>9:</td>
<td>11</td>
<td>Tenerife</td>
<td>Spain</td>
<td>4</td>
</tr>
</tbody>
</table>

**REFRESH MESSAGES TO SNAPSHOT TABLE**

<table>
<thead>
<tr>
<th>Addr</th>
<th>Base Addr</th>
<th>Prev Addr</th>
<th>Resort</th>
<th>Value Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>nil</td>
<td>Brighton</td>
<td>England</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td></td>
<td>Colombo</td>
<td>Sri Lanka</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td></td>
<td>Crete</td>
<td>Greece</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td></td>
<td>Tenerife</td>
<td>Spain</td>
<td>4</td>
</tr>
</tbody>
</table>

**SNAPSHOT TABLE AFTER REFRESH AT 9:00**

<table>
<thead>
<tr>
<th>Addr</th>
<th>Base Addr</th>
<th>Resort</th>
<th>Value Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:</td>
<td>3</td>
<td>Brighton</td>
<td>England</td>
<td>6</td>
</tr>
<tr>
<td>5:</td>
<td>7</td>
<td>Colombo</td>
<td>Sri Lanka</td>
<td>3</td>
</tr>
<tr>
<td>6:</td>
<td>8</td>
<td>Crete</td>
<td>Greece</td>
<td>4</td>
</tr>
<tr>
<td>7:</td>
<td>9</td>
<td>Florence</td>
<td>Italy</td>
<td>5</td>
</tr>
<tr>
<td>9:</td>
<td>11</td>
<td>Tenerife</td>
<td>Spain</td>
<td>4</td>
</tr>
</tbody>
</table>

**Figure 19. Basic R* Refresh of Snapshot Table:** The BaseAddr of the snapshot table is the tuple's address in the base table.

Before sending a qualified changed tuple, the address of the previous tuple satisfying the snapshot restriction is included as the PrevAddr in the refresh message, see Figure 20. Then the address and the value of the modified tuple is included.

In the case of an unqualified base table tuple, the algorithm still has to check if modification has taken place. The tuple may have satisfied the snapshot restriction the last time the refresh operation was invoked. If this is the case, the tuple is in actual fact deleted as far as the snapshot table is concerned, and the deletion must be notified in the next refresh message sent by referring to the last tuple that satisfies the snapshot restriction — hence the Deletion flag used in the algorithm defined in Figure 20.
RefreshCreate( BaseTable, SnapRestrict, LastRefreshTime, Channel );

begin
    LastQual := nil; (* Address of last qualified entry *)
    Deletion := false; (* Flag set if delete region detected *)
    ExpectPrev := nil; (* Expected PrevAddr applied in Fixup *)
    LastAddr := nil; (* Last base table addr applied in Fixup *)
    RefreshTime := CurrentTime; (* New value for TimeStamp *)

    DECLARE C1 CURSOR FOR
        SELECT Address, PrevAddr, TimeStamp, Value
        FROM BaseTable
        ORDER SEQUENTIAL;

    OPEN C1;
    Open( Channel, 'Send' );
    FETCH C1;
    repeat
        begin
            Fixup( );
            if SnapRestrict( Value ) then
                begin
                    (* Qualified base table entry *)
                    if TimeStamp > LastRefreshTime or Deletion
                        then
                            (* Updated or inserted entry,
                            possibly being a successor of deleted region *)
                            Xmit( Channel, < Address, LastQual, Value > );
                            LastQual := Address;
                            Deletion := false;
                        end
                    else
                        begin
                            (* Unqualified base table entry that may have qualified before *)
                            if TimeStamp > LastRefreshTime then Deletion := true;
                        end;
            end;
            FETCH C1;
        end
        until EndOfScan( C1 );

    (* Handle delete region at end of base table *)
    Xmit( Channel, < NULL, LastQual, NULL > );
    Close( Channel );
    CLOSE C1;
end;

Figure 20. Refresh Create for Basic R* Method: The ORDER SEQUENTIAL clause appended to the cursor definition is not SQL standard, but is used here to emphasize that the base table is scanned sequentially. Fixup is a local procedure, defined in Figure 21, which replaces the NULL values in the base table with real values.

and update refresh messages may therefore signal delete of one or several tuples from the snapshot table.

58 Database Snapshots
Fixup();
begin
  if PrevAddr = NULL then
    (* Insert entry *)
    UPDATE BaseTable
    SET PrevAddr = LastAddr,
        TimeStamp = RefreshTime
    WHERE CURRENT OF C1
  else
    begin (* Non-inserted entry *)
      if TimeStamp = NULL then
        (* Updated entry *)
        UPDATE BaseTable
        SET TimeStamp = RefreshTime
        WHERE CURRENT OF C1;
      if PrevAddr ≠ ExpectPrev then
        (* Current entry follows a deleted region *)
        UPDATE BaseTable
        SET PrevAddr = LastAddr,
            TimeStamp = RefreshTime
        WHERE CURRENT OF C1
      else
        if PrevAddr ≠ LastAddr then
          (* Entries inserted before current entry *)
          UPDATE BaseTable
          SET PrevAddr = LastAddr
          WHERE CURRENT OF C1;
          ExpectPrev := Address;
        end; (* Non-inserted entry *)
        LastAddr := Address;
    end;
end;

Figure 21. Fixup Operation: The fixup operation corrects the TimeStamp and PrevAddr fields in the base table. The procedure is applied within the refresh create process of R*, see Figure 20.

A detected delete may however cause shipment of qualified base table entries that have not been changed. An example of this is the first refresh message in Figure 19. The Brighton tuple satisfies the snapshot restriction, but has not been changed since the last refresh. Still, as the previous tuple is deleted, Brighton is sent as the next refresh message so as to inform the snapshot about the deletion.

The algorithm will delete neighbouring deletes — including unqualified updates — and treat them as one deleted region. No example demonstrating this effect has been shown in Figure 19. Assuming that the price level of the Brighton tuple has been updated to 8 prior to refresh at 9:00, then a scan of the base table shown in Figure 17, will proceed
RefreshUpdate(Snapshot, Channel);
begin
  Open( Channel, 'Receive' );
  Receive( Channel, Address, PrevAddr, Value );
repeat
  begin
    (* Entries between PrevAddr and Address no longer qualify *)
    DELETE FROM Snapshot
    WHERE BaseAddr > PrevAddr
    AND BaseAddr $ Address;

    (* Insert new entry possibly replacing old deleted entry *)
    INSERT INTO Snapshot
    VALUES ( Address, Value );
    Receive( Channel, Address, PrevAddr, Value );
  end
until Address = NULL;

(* Entries at end of snapshot is removed *)
DELETE FROM Snapshot
WHERE BaseAddr > PrevAddr;
Close( Channel );
end;

Figure 22. Refresh Update for Basic R* Method: Update of a snapshot entry is performed by deleting the old entry prior to insert of its new value.

as follows: Brighton is detected as unqualified, then Cannes, but Colombo as the third entry satisfies the snapshot restriction. Thus, Colombo will be sent as the first refresh message informing the snapshot table that all the previous tuples can be removed. This feature may reduce the number of messages sent, especially for restrictive snapshots.

Despite this, the algorithm sends redundant messages, i.e. messages bearing unnecessary information. Since the before image of each deleted tuple is not saved, it is impossible to detect whether a deleted base table tuple actually was in the snapshot table or not. As a consequence, all deletions must be signalled to the snapshot. Unfortunately the same applies for updates and inserts not satisfying the snapshot restriction. An unqualified insert may precede a qualified, but unchanged tuple (refresh message containing Tenerife in Figure 19). Likewise, an unqualified update may cause a shipment of the next qualified, but unchanged tuple (refresh message containing Colombo in Figure 19). In the next section we will discuss enhancements that eliminate these refresh messages.

The refresh update process is defined in Figure 22. The algorithm, taken from [LIND 86], treats all refresh messages as inserts. Updates are thus performed as delete followed by insert of the new tuple value. A dummy message is received at the end to ensure that entries deleted at the end of the base table, are removed from the snapshot table as well.

Before ending the description of the basic R* method, some comments on its performance and applicability to other database systems than R* are now made.
First, the inaccuracy in selecting entries from the base table potentially increases as the snapshot qualification becomes more restrictive, i.e. the unqualified portion of the base table becomes larger. Note however, that the coalescing effect greatly reduces the number of superfluous messages since only tuples satisfying the snapshot restriction are sent. Another way of seeing this is to observe that the chance of coalescing increases as the criteria becomes more restrictive. On the other hand, for very restrictive snapshots, we may end up sending all of the base table tuples satisfying the snapshot restriction. This implies that the full refresh strategy may be better for these cases.

Another performance observation should be made: In order to detect base table modifications the entire table is scanned. Thus, if only a few updates are made between refreshes, the overall processing cost increases with the table size. We will return to these issues in "Chapter 5. Refresh Method Analysis".

Finally, the algorithm may be used in database systems other than R*. As described, the method relies on the existence of an ordered address field mapped to each tuple. In portable database systems, running with highly varying file systems and address representation, provisions must be made to represent the address in a machine independent way. If the file system provides an order, but tuples "change address" due to space relocation or reorganization, the scheme is not applicable.

This can be overcome if the database system imposes the existence of a unique primary key. As keys assign an order and can be represented machine independently, the primary key can be used instead of the address. The penalty of this is that the size of a primary key may well exceed the size of an address. Thus the scheme may require a significantly larger amount of space to store the base table. Alternatively, a surrogate key [HALL 76] could be used, but this would impose some processing overhead.
4.5 Enhancements to R*

The basic R* refresh algorithm — described in the previous section — includes unqualified updates as well as unqualified inserts in the refresh messages sent to the snapshot table. In this section, we propose two enhancements reducing the number of unnecessary refresh messages.

4.5.1 Enhancement One

The first enhancement is based on a more efficient usage of the NULL value settings in the base table. The basic R* algorithm does not distinguish between updates and inserted entries when the entry is otherwise unqualified, see the test performed in Figure 20 on page 58. Entries inserted into the base table have their timestamp set to the current time and will thus be considered as part of a delete region by the algorithm. This situation can be avoided by introducing some minor modifications to the basic algorithm.

First, the Fixup procedure defined in Figure 21 is modified to include the situation that the entry being corrected is an update entry. This event is then signalled to the RefreshCreate procedure, e.g. as a flag named Updated. Unqualified update entries may thus be distinguished from the inserts by the following test in the RefreshCreate procedure, cf. Figure 20:

if Updated and (TimeStamp > LastRefreshTime) then Deletion := true;

Unqualified inserts can now be discarded from the refresh process. A refresh of snapshot $S_t$ using Enhancement One as defined above, will cause the set of refresh messages listed in Figure 23. Tenerife that was sent previously, is now excluded from the refresh messages.

The enhancement can be embedded in the basic R* method quite easily and with no added overhead. The effect of the enhancement is significant if the modifications submitted to the base table contain a large proportion of tuple inserts.

The modification to the test in the RefreshCreate procedure given above, can be simplified to;

if Updated then Deletion := true;  

— as we only want to distinguish updates from inserts. The update flag is set when the timestamp of a non-inserted tuple is changed. In the following, we assume the first form of the test statement.

---

6 Naturally, assignment of Boolean variables should be used from a language puritanical point of view.
REFRESH MESSAGES TO SNAPSHOT TABLE

<table>
<thead>
<tr>
<th>Base Addr</th>
<th>Prev Addr</th>
<th>Resort</th>
<th>Value Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>nil</td>
<td>Brighton</td>
<td>England</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>Colombo</td>
<td>Sri Lanka</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>Crete</td>
<td>Greece</td>
<td>4</td>
</tr>
<tr>
<td>NULL</td>
<td>11</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Figure 23. Refresh using Enhancement One

4.5.2 Enhancement Two

Despite the use of Enhancement One, unqualified updates are still sent to the snapshot table. This situation can only be remedied by introducing some sort of before image in the base table itself. Recording the full before image for every updated entry is an intractable solution. Instead, we may keep a dense recording of the before image by marking each base table entry as satisfying the snapshot restriction at last refresh time or not.

Enhancement Two relies on a third column, Qual, added to the base table, see Figure 24. Qual is a flag that is set if the tuple satisfies the snapshot restriction at the refresh time (or create time). The flag will be reset at successive refresh on the base table, but not until the tuple has been checked for qualification in the ongoing refresh. In this way, the enhanced refresh create algorithm may detect if an unqualified tuple, that has been updated, in actual fact qualified previous to the modification or not. The test in the RefreshCreate procedure becomes:

```
if Updated and (TimeStamp > LastRefreshTime) and Qual then Deletion := true;
```

The effect of Enhancement Two is illustrated in Figure 24 along with the revised base table. By using the dense before image column, the update of Cannes can be disregarded as a delete region and thus avoiding shipping Colombo.

Enhancement Two, defined by the revised if statement, incorporates Enhancement One. The test for unqualified inserts and updates can be expressed in a different way which opens for provision of multiple snapshots. It all has to do with the qualification flag which must be set to unqualified for tuples added to the base table. This reflects that the presence of the tuple in the snapshot table, so far, is unknown. At the next refresh of the base table, the new tuple is either selected for shipment, or the new tuple is unqualified in which case it was unqualified before. Thus, the test necessary to provide for the enhancement can be simplified to:

```
if (TimeStamp > LastRefreshTime) and Qual then Deletion := true;
```

and the Fixup procedure may remain as defined originally, without the Update flag.
**BASE TABLE BEFORE REFRESH**

<table>
<thead>
<tr>
<th>Addr</th>
<th>Prev Addr</th>
<th>Time Stamp</th>
<th>Qual</th>
<th>Resort</th>
<th>Value Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td>1</td>
<td>0:00</td>
<td>yes</td>
<td>Brighton</td>
<td>England</td>
<td>6</td>
</tr>
<tr>
<td>5:</td>
<td>3</td>
<td></td>
<td>no</td>
<td>Cannes</td>
<td>France</td>
<td>8</td>
</tr>
<tr>
<td>7:</td>
<td>5</td>
<td>0:00</td>
<td>yes</td>
<td>Colombo</td>
<td>Sri Lanka</td>
<td>3</td>
</tr>
<tr>
<td>8:</td>
<td>NULL</td>
<td>NULL</td>
<td>-</td>
<td>Crete</td>
<td>Greece</td>
<td>4</td>
</tr>
<tr>
<td>9:</td>
<td>7</td>
<td>0:00</td>
<td>yes</td>
<td>Florence</td>
<td>Italy</td>
<td>5</td>
</tr>
<tr>
<td>10:</td>
<td>NULL</td>
<td>NULL</td>
<td>-</td>
<td>Miami</td>
<td>United States</td>
<td>7</td>
</tr>
<tr>
<td>11:</td>
<td>9</td>
<td>0:00</td>
<td>yes</td>
<td>Tenerife</td>
<td>Spain</td>
<td>4</td>
</tr>
</tbody>
</table>

**REFRESH MESSAGES TO SNAPSHOT TABLE**

<table>
<thead>
<tr>
<th>Base Addr</th>
<th>Prev Addr</th>
<th>Resort</th>
<th>Value Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>nil</td>
<td>Brighton</td>
<td>England</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>Crete</td>
<td>Greece</td>
<td>4</td>
</tr>
<tr>
<td>NULL</td>
<td>11</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Figure 24. Refresh using Enhancement Two: A Comp column is added to the base table for recording the qualification of each tuple at last refresh. A tuple is marked with yes if qualified at last refresh. Inserted tuples are not given a mark in the example, but a ‘no mark’ could be used without disturbing the enhanced algorithm.

Enhancement Two can, like the previous enhancement, be incorporated in the basic R* algorithm quite easily. In addition to the modified test for unqualified entries, the Comp column must be reset by the following statement:

```sql
if Qual ≠ SnapRestrict( Value )
then
  UPDATE BaseTable
  SET Qual = SnapRestrict( Value )
  WHERE CURRENT OF CI;
```

before the next base tuple entry is fetched. The statement may appear unnecessarily complex at first, but the reason will become clear in the discussion below.

In order to support the enhancement to the basic R* method, the relational scheme of base tables must be modified to include the third column. The performance impact on the normal base table operations is virtually nil compared with the basic method as the column is basically maintained at refresh time (tuple insert is the exception). The flag is updated at refresh so as to be used at the next snapshot refresh. The base table fixup must, however, be performed as an atomic action or else the Qual column will be left in an inconsistent state.

The maintenance of the Qual column will obviously add to the processing cost at snapshot refresh. This cost can be kept down to a minimum by applying the base table up-
date as suggested above. The flag is only changed if its new value differs from its old value. This event takes place only when the base table tuple has been modified, in which case the fixup operations have changed the tuple anyway (by replacing the NULL values with real values). The cost can for this reason be neglected. Also, at snapshot creation time, the Qual column is established together with two columns introduced by the basic R* method.

4.6 Enhancement to Logging

In the R* approach, emphasis is on reducing the number of refresh messages sent to the snapshot table. This is obtained by the coalescing effect of neighbouring deletes. The question is: Can a similar reduction be made using the logging approach.

In contrast to the R* approach, the logging approach records the modifications in a separate table. The refresh process is based on finding the modifications in the log that are relevant rather than examining the base table by a sequential scan. Tuples deleted from the base table are recorded in the log as delete entries. Detecting neighbouring deletes is not possible without confirming with the actual state of the base table. Otherwise we can not determine if any two deletes are separated by an unmodified entry or not.

An enhancement to the logging approach — with respect to the number of refresh messages sent — is possible at the expense of searching the base table (in addition to the log) at refresh time.

The enhancement is in other words based on a sequential scan of the base table and the log. This can best be done by scanning the base table sequentially — in which tuples satisfying the snapshot restriction are selected. For each qualified tuple, we have to consult the log to verify if the tuple is new, changed, or possibly a successor of a deleted region that concerns the snapshot table. Basically, this requires a search for log entries within key boundaries. Consequently, some of the logging alternatives can be ruled out.

The enhancement is only useful when we have a key ordered log, i.e. a sequential log sorted by the refresh create process, or a condensed log based on an order-preserving access method such as B-trees. For simplicity, we discuss the enhancement within the scope of a condensed log based on B-trees.

Figure 25 is a pseudo code representation of the refresh create process for the enhanced condensed log. The base table entries satisfying the snapshot restriction are scanned in sequential order. For each qualified tuple, the condensed log is consulted to verify if the tuple is new or changed, or possibly a successor of a deleted region that concerns the snapshot table. A refresh message — containing the tuple and the primary key of the previous tuple that qualified — is sent to the snapshot table, see Figure 26.

From the point of view that the number of modifications is less than the number of entries in the base table for low modification activity, the condensed log should be scanned and the base table consulted instead. On the other hand, an index may be used to search for the base table tuples that satisfies the snapshot restriction. Besides, the logic of the algorithm as given in Figure 25 is much simpler than one with the scan performed the
RefreshCreate( BaseTable, SnapRestrict, CondLog, Channel );
begin
  LastKey := NULL; (* Primary key of last qualified tuple *)
  DECLARE C1 CURSOR FOR
      SELECT Tuple
      FROM BaseTable
      WHERE SnapRestrict(Tuple)
      ORDER SEQUENTIAL;

  OPEN C1;
  Open( Channel, 'Send' );
  FETCH C1;
  repeat
    begin
      (* The qualified base table tuple is either
         i) inserted/updated and recorded in the log, or
         ii) a successor of a deleted region between
             the last qualified tuple and the current one. *)
      if EXISTS( SELECT *
                 FROM CondLog
                 WHERE PKey = Tuple↑.PKey
                 OR ( PKey > LastKey AND
                      PKey < Tuple↑.PKey AND
                      SnapRestrict( BImage ) )
            then Xmit( Channel, < LastKey, Tuple > );
            LastKey := Tuple↑.PKey;
            FETCH C1;
        end
    until EndOfScan( C1 );

    (* Handle delete region at end of base table *)
    if EXISTS( SELECT *
               FROM CondLog
               WHERE PKey > LastKey
               AND SnapRestrict( BImage )
        then Xmit( Channel, < LastKey, NULL > );
        Close( Channel );
        CLOSE C1;
  end;

Figure 25. Refresh Create for Enhanced Condensed Log: The base table qualifying the
snapshot restriction is scanned in sequential order. The condensed log — con-
taining before images only — is checked for changes. The way EXISTS is used
is not standard.

other way around. This does not mean that the other logging alternatives could be used,
the requirement for a key ordered log is still essential to the algorithm.

The refresh update process of the enhanced log is identical to the one in R* — see
Figure 22 on page 60 — except that the addresses are replaced with primary key values.

66 Database Snapshots
REFRESH MESSAGES TO SNAPSHOT TABLE

<table>
<thead>
<tr>
<th>Prev PrimaryKey</th>
<th>Resort</th>
<th>Value Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>Brighton</td>
<td>England</td>
<td>6</td>
</tr>
<tr>
<td>Colombo</td>
<td>Crete</td>
<td>Greece</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 26. Refresh using Enhanced Condensed Log

In the enhanced version of the condensed log, after images of the changed tuples are no longer needed. The condensed log therefore records the before images of modified tuples, i.e. the tuple value as at the time of the last refresh. Inserts and deletes are recorded as before even though inserts represent after images. This is done so as to distinguish inserted tuples from unchanged base table tuples. The condensed log will therefore contain before images reflecting the value of modified base table tuples as at the time of the last refresh. The merging of new modifications with stored ones is much simpler than depicted in Figure 15 on page 51. The stored entry, if any, is kept intact unless the situation at hand is a delete of a previously inserted entry, in which case the outcome is no entry kept in the log.

Figure 26 shows the refresh messages sent using the enhanced condensed log. Note that no "end" message is needed as in the R* based refresh methods. The difference between the Enhancement Two of R* and the enhanced condensed log — with respect to the number of refresh messages sent — is that unqualified deletes never cause a shipment of an unchanged entry to the snapshot table.

Obviously, the cost of normal processing is much higher in the enhanced log than in the R* method. The cost is as for the basic condensed log. Some reduction in the refresh create processing may be obtained compared with the R* method. The refresh processing is however more expensive than for the basic condensed log. For a detailed discussion we refer to "Chapter 5. Refresh Method Analysis".

The enhancement can also be applied to a sequential log, sorted by the sort merge refresh algorithm. As a result of this process, we may obtain a log with the same property as the ordered condensed log. This sorted, condensed, sequential log can then be applied in the algorithm defined in Figure 25. Clearly, the enhancement will increase the refresh create processing for this alternative even further. On the other hand, the reduction in the number of refresh messages may outweigh this increase when compared with the sort merge refresh strategy.
4.7 Summary and Concluding Remarks

We have described a series of differential refresh methods having in common that modifications made to the base table since last refresh, are recorded on a tuple basis. Two main approaches are discussed. The logging approach, in which modifications are recorded in a separate log. The main advantage of this approach is that it allows the refresh create process to operate on a domain separated from the base table. The other approach, the R* approach, is based on marking the changed tuples — its main advantage being the minimal overhead imposed on the normal base table operations.

Keeping modifications in a separate domain is important for maintaining the availability to data. Data availability can not be ensured in the event of site failure during 2-phase-commitment. Since refresh of snapshots in the R* approach must be terminated by a full 2-phase-commitment, the base table may be locked for some time in the case of site failure. For details, see discussion in "Chapter 3. Implementing Snapshots".

Another disadvantage of the R* approach is that the method can not support differential refresh of complex snapshot restrictions. As discussed in "Chapter 3. Implementing Snapshots", the methods proposed in [BLAK 86] can be used to differential refresh of snapshots defined as a join of several base tables. The composition algorithm does however require a recording of all modifications submitted to the base table since the last refresh.

In the R* approach, deletes are never recorded and consequently, the method is ruled out when it comes to supporting join based snapshots. The logging approach is not constrained by this. Here, all modifications are recorded — either every one of them, or in a condensed form.

Whereas the R* approach may endanger the availability of data, the method does not degrade the performance of base table operations. For some of the logging alternatives, the logging of modifications represents a rather significant overhead. This is especially true for the condensed logging method. In the sequential log, this overhead is not so dominating.

The two main approaches support differential refresh strategies. As the examples hopefully have demonstrated, the algorithms differ in their accuracy of selecting entries for updating the snapshot.

As for the sequential logging methods, the sort merge refresh strategy is much more selective than the basic strategy. The condensed logging scheme provides for the same accuracy as the sort merge of sequential log.

The number of refresh messages may be reduced even further by eliminating the shipment of deleted messages. This is the main objective of the R* approach and the enhanced log. Deletes are instead sent as part of insert and update messages. The two enhancements proposed for R* refresh cause a smaller shipment of refresh messages than the basic refresh algorithm. In "Chapter 5. Refresh Method Analysis", expressions for estimating the number of refresh messages in each approach will be given. These are then used to analyze the difference in overall cost of snapshot maintenance using the differential refresh methods described in this chapter.
The working load on the refresh create and update process vary a great deal in the refresh strategies described. The refresh update process does in general benefit from having the refresh message received in key order sequence — as in the R* approach, the sorted condensed log, and the sort merge refresh of sequential log. The working load on refresh create is, in the logging approach, proportional to the number of log entires. In the R* approach, the base table size determines this cost. An exception to this rule is the sort merge of sequential log which may increase the cost significantly.

In the discussion of the sort merge strategy, we assumed that the entire sequential log is sorted. For very restrictive snapshots, the overhead of the sort merge step becomes extremely high compared with basic refresh. Since most of the log entries are disqualified by the snapshot restriction, it will be worth while discarding irrelevant entries at an early stage, before the sort step.

If we assume a sequential log with both before and after images, we may expel all entries disqualified by the snapshot restriction. In other words, we sort only those entries qualified by the snapshot restriction, and then apply the merge rules — given in Figure 15 on page 51 — to these to produce one condensed log history for each base table tuple modified.

Indeed what this approach really does, is a sorting of the modifications we intended to ship to the snapshot table using the basic refresh strategy. By sorting these modifications, we may merge entries reflecting changes to the same base table tuple into one condensed message, and thus obtain the condensed property of the condensed logging approach at a more reasonable cost.
Chapter 5. Refresh Method Analysis

5.1 Introduction

In this chapter, we analyze the differential refresh methods, described in "Chapter 4. Differential Refresh Methods", by comparing the performance impact of each snapshot mechanism. A cost analysis addressing each of the differential refresh approaches, including the enhancements proposed in this thesis, is performed. For some of the methods, alternative implementations of the refresh operation or modification identification have been analyzed.

An analysis of the sequential and condensed log is given in [KÄHL 87a] on a smaller scale. The paper compares the overall cost of snapshot maintenance when using the basic refresh strategy of sequential logging and a condensed log based on B-trees. The analysis, which is based on an analytic model, is included here not only for completeness, but because of adjustments made to the analytic model since then — adjustments that have influenced the results.

In addition to the analytic model used for comparing the two logging methods, the model now includes the enhancement to logging. Furthermore, an analytic model has been developed for the R* method, including the two enhancements proposed in this thesis.

In the following, we first describe the analytic model defined to express the length of each log and the number of log entries satisfying the snapshot restriction. Then, a corresponding model for the R* method is defined. The model expresses the number of base table tuples exposed to fixups in the R* algorithm, and the number of tuples that are relevant to the snapshot table. Formulas expressing these figures are given for both the logging and the R* methods, and some curves are provided to illustrate the same figures.

A cost model estimating the overall cost of snapshot maintenance is defined next. The cost model includes the processing and communication cost factors identified in "Chapter 3. Implementing Snapshots", see also Figure 5 on page 30. The cost model disregard cost that may be caused by recovery operations, i.e. we assume that the snapshot is successfully refreshed at each snapshot refresh. We also assume that the working load associated with keeping a recovery log is covered by the normal base table operations, i.e. no cost of significance is introduced by the recovery log to the snapshot mechanism.
A cost comparison of the refresh methods is done using this cost model and by applying the formulas defined in the analytic model. The analysis is carried out under various update frequencies of the base table, for various modification compositions, and for various snapshot selectivities. Alternatively, the cost comparison can be done by using simulated values for the number of refresh messages sent to the snapshot table.

Finally, we discuss the results from the cost comparison and emphasize the factors that may influence the results. We will also comment on the effect of neglecting the processing cost partly or totally. A simulation model has also been devised, and we correlate these results with the findings of the analytic model. The chapter ends with a comparison of differential and a full refresh strategy, and summarizes when differential refresh methods, and which ones, are profitable.

5.2 Definitions

We define a modification set as being the modifications submitted to a base table, on a tuple basis, since last refresh. The modification set consists of tuple 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, i.e. each tuple is equally likely to be referenced by the read operation that precede a delete or an update, and that inserts are evenly distributed over the base table. The base table consists of \( N \) tuples prior to any modification.

A base table may not always be exposed to all three types of modification. The following main classes, reflecting the composition of the modifications to base tables, are defined:

- A static base table in which the base table is updated only, 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 that the number of inserts outweighs the number of deletes, i.e. \( p \geq q \).

The three classes represent the most common update situations on base tables.

A snapshot, based on a single base table, may either be defined as a full copy or as a restricted subset. The selectivity of a snapshot, defined as the ratio of the base table tuples satisfying the snapshot restriction, is denoted by \( s \). In the case of a full base table copy, the selectivity is defined by \( s = 1.0 \). We assume that the selection of the base table tuples for the snapshot table, is uniformly distributed.
5.3 Analytic Model of Logging Methods

In the logging approach, the modification set is recorded in a separate domain; the log. In order to calculate the overall impact of a snapshot mechanism based on a logging scheme, the number of log entries is needed as well as the number of refresh messages sent to the snapshot site, cf. Figure 5 on page 30. In the analytic model, described in the following, the number of entries stored in each logging approach is estimated. Estimates for the number of log entries that are relevant to the snapshot table are also given.

First, we give the estimates for the sequential logging method. Then, we describe an analytic model for the condensed log and finally, we will apply this model to the enhanced log. A summary of the estimates is provided at the end of this section, and curves are drawn for various selectivities of the snapshot table and modification compositions.

5.3.1 Sequential Log

In the case of a sequential log, the length of the log equals the number of modifications submitted to the base table since the last refresh — namely $M$. The number of entries relevant to the snapshot table can be expressed as a function of the selectivity ($s$), the modification type ratios ($p$, $q$, $r$), and the sequential log length ($M$).

We will consider a sequential log with both before and after images as it provides for more accuracy than a log with after images only. The refresh create process, defined in detail in Figure 12 on page 47, ensures that only entries satisfying the snapshot restriction are sent to the snapshot table.

In the case of a restrictive snapshot, some tuples will neither qualify before, nor after the completion of the modification set. The modification set consists of a mix of inserts, deletes, and updates which are assumed uniformly distributed over the base table. As the selection of base table tuples for the snapshot table is also uniformly distributed, we may assume that the same applies for the selection of log entries.

The number of log entries that apply to the snapshot table in the sequential log case, can be estimated as follows: The number of insertions that qualify the snapshot restriction is $spM$. Similarly, $sqM$ deletions will qualify.
In the case of updates, the equivalent number of entries that qualify the snapshot restriction can be found by examining the before and after images of the log entries. The ratio of before images that satisfy the snapshot restriction is $s$, so is the ratio of after images. In Figure 27, the four combinations expressing the qualification of the before and after images, are displayed. Also the outcome, in terms of what type of modification that will be issued on the snapshot table, is given for each combination as well as the ratio of updates that cause each combination. The number of updates relevant to the snapshot table can be found by adding the contribution from the first three combinations listed in the figure, i.e. $(s^2 + s(1-s) + s(1-s) + s(2-s)r)M$ — which equals $s(2-s)rM$ — of the updates qualify for the snapshot.

The expression above can be found more directly using probabilities, but the classification as shown will be most useful when discussing the enhanced log later on.

The total number of modifications that are relevant to the snapshot table, expressed by $M_s$, is thus:

$$M_s = spM + sqM + s(2-s)rM$$

Since the sum $p + q + r$ is one, the expression can be written as:

$$M_s = s(1+r(1-s))M$$

### 5.3.2 Condensed Log

Similar expressions are found for the condensed log. Whereas the sequential log may record several modifications to one base table tuple, the condensed log only records one. We have to estimate the "collision" effect in order to calculate the length of a condensed log. One way of doing this, is to estimate the "inverse" effect on the base table. In determining the number of entries in a condensed log, the following property is used: The probability that a given tuple of the base table is never referenced by any of the modifications. The number of tuples — out of the original $N$ tuples in the base table — that have been changed, can easily be found by applying this property.

Several conditions must hold in order to fulfill the property defined above. The probability may therefore be expressed by the probability of these conditions, and is defined as follows:

*The probability that a given tuple is not modified by a given modification equals:*

1. the probability that the modification is an **insert** of a new tuple, or
2. the probability that the modification is a **delete** of another base table tuple, or
3. the probability that the modification is an **update** of another base table tuple.

Since each of these conditions represents statistically disjunct events, the probability that a given tuple is not changed by the modifications, may be expressed as the sum of the probability for each of the conditions listed above.
The probability that modification is insert equals the ratio of inserts $p$. The probability that modification is delete of another base table tuple than the one considered, equals the probability of delete times the probability of “hitting” the other base table tuples. The probability that modification is update of another base table tuple is similar. In order to find exact expressions for the last two cases, we have to express the probability of “hitting” one particular base table tuple.

Initially, the base table holds $N$ tuples. As a uniform distribution of the modifications is assumed, the probability of “hitting” one tuple of the base table may be expressed as $1/N$. Since the base table is exposed to deletes and inserts continuously, the number of tuples present in the base table will vary as modifications are submitted. We therefore define $N_i$ as being the number of tuples present after submitting the $i^{th}$ modification. $N_0$ is defined as $N$.

$N_i$ expresses the expected number of tuples in the base table after the $i^{th}$ modification. This can be expressed by the ratio of each modification type (insert, delete, and update), and the number of modifications $i$ entered:

$$N_i = N + ip - iq = N + i(p-q)$$

Note that updates introduce no increase/decrease to the number of tuples present. The probability of “hitting” one tuple of the base table in the $i^{th}$ modification is thus $1/N_{i-1}$.

Next, we define $u_i$ as the probability that a given tuple is not modified by the $i$ modifications submitted to the base table. At the time of the first modification, this probability equals the probability that a given tuple is not changed by the first modification. By applying the rules for the combined event as defined above, $u_1$ becomes:

$$u_1 = p + q(1-1/N) + r(1-1/N)$$

which can be simplified as:

$$u_1 = p + (q+r)(1-1/N)$$

The probability that a given tuple is not modified after the $i^{th}$ modification may be expressed as a combined probability. The event suggests that the given tuple has not been changed by any of the previous $(i-1)$ modifications, and will not be changed by the $i^{th}$ modification either. The two events are statistically independent of one another. The probability that a given tuple is not modified after $i$ modifications may thus be expressed as the product of the two:

$$u_i = u_{i-1}(p + (q+r)(1-1/N_{i-1}))$$

The probability that a given tuple is not modified by the $M$ modifications of the modification set may thus be expressed by a product as follows (the expression for $N_i$ has been substituted in the formula):

$$u_M = \prod_{i=1}^{M} \left( p + (q+r)(1-1/(N+(i-1)(p-q))) \right)$$
The expected number of “old” base table tuples that are not changed by the modification set becomes $N_{bM}$. The number of “old” base table tuples updated one or several times, deleted, or possibly both, is thus given as $N(1-u_M)$.

We can now establish an expression for the condensed log length. The condensed log consists of entries that reflect the new tuples added to the base table, and entries that reflect updates or deletes of “old” base table tuples. According to the merge rules of stored and new modification entries, cf. Figure 15 on page 51, some of the newly added tuples may be deleted later on — in which case the net result is no entry in the condensed log.

The number of entries on the condensed log is thus defined according to the merge rules, and consists of:

Case 1: One entry for each new tuple inserted into the base table. The number of such entries is proportional to the insert ratio of modifications, $p$.

Case 2: One entry for each “old” base table tuple that has been changed one way or the other (i.e. updated one or several times, deleted, or possibly deleted after first being updated). The number of such entries is expressed by means of the $u_M$ probability.

Case 3: Minus one for each removed entry. This number depends on the probability of deleting a tuple that has previously been inserted into the base table during the same modification set.

Case 4: Minus one for each insertion that really replaces an “old” base table tuple that has been deleted earlier in the same modification set. Since the delete operation has already been accounted for, such insertions should not cause an extra entry.

The negative contribution from the last case is neglected since an insert of a previously deleted entry is considered unlikely. If the case was to be considered, the probability of the event would depend on the density of the key, i.e. the number of key values stored in the base table to the number of key values possible.

The number of new tuples (Case 1) is $pM$, and the contribution from Case 2 is $N(1-u_M)$.

The negative contribution from Case 3 can be expressed by the ratio of new tuples to the total number of tuples present in the base table. In average, $pM/2$ tuples are added to the base table, and $N+(p-q)M/2$ tuples are kept in the base table. The probability that a delete operation cause the removal of a new tuple may thus be expressed as $q'$:

$$q' = q(pM/2)/(N+(p-q)M/2)$$

which can be simplified as:

$$q' = q(pM/(2N+(p-q)M))$$

In total, $qM$ deletes are submitted in the modification set. The negative contribution of Case 3 thus amounts to $q'qM$. An alternative expression can be found by applying
the same sort of reasoning as for the \(u_M\) expression, but cause no significant change to the results.

The condensed log length, expressed by \(L\), is thus given as:

\[
L = pM - q'qM + N(1-u_M)
\]

As in the sequential log, the number of modifications that are relevant to the snapshot table depends on the selectivity of the snapshot and the composition of the modification set. The formula estimating the number that qualify is, however, more complex than in the sequential case.

The entire log may record a complex modification history although the final, merged recording is the only one kept in the log. The final recording of an entry reflects either an insert, a delete, or an update operation to be performed on the snapshot table. We therefore divide the log into the following modification categories:

1. \(L_i\) being the **net number of inserts** which equals the total number of inserts minus those deleted later on in the modification set. Note that added tuples may have been updated later on in the modification set.

2. \(L_d\) being the **net number of deletes** which equals the number of "old" base table tuples that have been deleted as deletion of new ones do not appear in the log. Note that deleted tuples may have been updated prior to delete.

3. \(L_u\) being the **net number of updates** which equals the number of "old" base table tuples that are updated only, i.e not deleted later on in the modification set.

Each of these numbers can be estimated by using the formulas applied in the expression for \(L\). The following expressions are given:

\[
L_i = pM - q'qM
\]

\[
L_d = qM - q'qM
\]

\[
L_u = N(1-u_M) - L_d
\]

The sum of the three expressions is naturally \(L\).

The number of modifications that are relevant to the snapshot table, in the case of a condensed log, can be found by applying the same sort of reasoning as for the sequential log. The proportion of net inserts that qualify for the snapshot is \(s\), so is the proportion of net deletes, whereas the proportion of net updates is \(s(2-s)\) like in the sequential log. The total number of modifications that are relevant to the snapshot table, \(L_s\), is thus:

\[
L_s = sL_i + sL_d + s(2-s)L_u
\]
5.3.3 Enhanced Log

The third and last logging alternative is the enhanced log which really is an extension based on a condensed log or a sequential log with similar property.

As described in "Chapter 4. Differential Refresh Methods", recording of after images for updates is no longer needed in the enhancement to logging, but inserts and deletes are still recorded as in the condensed log. The length of the enhanced log, in terms of the number of entries in the log, is therefore equal to the condensed log length $L$.

The refresh algorithm of the enhanced log attempts to reduce the number of refresh messages sent to the snapshot table by sending only the updates and inserts that are relevant to the snapshot. Deletes are not sent as such, but are included in the other messages if a delete has occurred in the actual region. In some cases, this will cause shipment of entries that are not changed, but do qualify for the snapshot. This takes place if a delete, which is relevant to the snapshot, has occurred in the region preceding unchanged, but qualified entries.

The number of refresh message in the enhanced log is therefore reduced compared with the condensed log. In order to find an estimate for the number of refresh messages, we distinguish between the refresh messages carrying insert or update, and those that carry the information that a region has been changed only. By this we mean what the messages semantically are carrying. From the message itself, this distinction can not be made.

The expressions for the net number of inserts, $L_i$, and updates, $L_u$, as found for the condensed log, are used to determine the number of refresh messages of the first category. The proportion of the inserts that qualify is $s$, whereas the proportion of the updates that qualify can be found as follows. First note that the enhanced log does not contain the after image of an updated entry. The after image is instead retrieved from the base table. From Figure 27, we see that $s^2$ of the updates cause real updates of the snapshot table, and that $s(1-s)$ of the updates cause inserts. We let $U$ denote the number of inserts and updates that qualify for the snapshot, i.e.:

$$U = sL_i + (s^2 + s(1-s))L_u = s(L_i + L_u)$$

Now we find an expression for the number of entries belonging to the second category, i.e. those unchanged entries of the base table that qualify for the snapshot, and are successors of a deleted region. Region changes that cause such shipment, are either modification of type delete, or updates that modify an entry so it no longer qualifies for the snapshot. The number of deletes is $sL_d$ as in the condensed log. The number of updates, that from the snapshot's point of view are true deletes, is $s(1-s)L_u$ (see Figure 27). We let $V$ denote this number of deletes "as seen by" the snapshot, i.e.:

$$V = sL_d + s(1-s)L_u$$

Let us now assume that we perform the modifications on the snapshot table and analyze the situation from the snapshot table's point of view. We will perform $U$ insert/update modifications and $V$ deletes. During this process, some of the snapshot table tuples will be left unchanged. From the condensed log we found that the number of old, unmodified base table tuples is $Nu_M$. As we assume that selection of tuples to the snapshot table
is uniformly distributed, $s \text{Nu}_M$ of the snapshot tuples will be left unchanged. We may therefore define the following statistical model with the three disjunct events:

- $B_1$: event is tuple deleted from snapshot table.
- $B_2$: event is tuple inserted/updated in snapshot table.
- $B_3$: event is tuple not changed in snapshot table.

Clearly, the same events may be recognized by scanning the base table and the enhanced log during the refresh create process before we actually perform the operations on the snapshot table. The number of such events are $V$, $U$, and $s \text{Nu}_M$ respectively. The total number of events, $W$, is the sum of these three expressions:

$$W = U + V + s \text{Nu}_M$$

By substitution and rearrangement, the $W$ expression can be written as:

$$W = s(N + L_i + (1-s)L_a)$$

Note that all of the events listed are relevant to the snapshot table one way or the other. Given the total number of events and the three classes of events as defined above, the probability of each event may be defined. Thus, the probability of event $B_i$ is $s \text{Nu}_M/W$ and expresses the probability of finding an unchanged tuple that is qualified by the snapshot (amongst those deleted, inserted, updated and unchanged in the snapshot table). This probability can then be used to express the need for sending refresh messages caused by a delete. As we have $V$ number of deletes, the number of unchanged, qualified tuples sent to the snapshot because of being successor of a relevant a delete, is $V(s \text{Nu}_M/W)$.

We may now establish an expression for the number of refresh messages sent in the enhanced log approach. Before we do so, we recall from the algorithm in Figure 25 on page 66, that an extra refresh message at the end is needed if the end region of the base table has been changed. The probability of this is expressed by the $B_i$ event. Thus, the total number of shipments in the enhanced log, $E_r$, is given as:

$$E_r = U + V(s \text{Nu}_M/W) + V/W$$

which by substitution can be rearranged to:

$$E_r = s(L_i + L_a) + (L_d + (1-s)L_a)(s \text{Nu}_M + 1)/(N + L_i + (1-s)L_a)$$

### 5.3.4 Summary

The estimates for the number of entries in each log and the corresponding number of refresh messages produced during refresh processing are summarized in Figure 28. The formulas are expressed as a function of the modification type ratios ($p,q,r$), the modification set size ($M$), the base table size prior to modification ($N$), and the selectivity of
Log Method Summary

<table>
<thead>
<tr>
<th>Log Method</th>
<th>Log Entries</th>
<th>Refresh Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential Log</td>
<td>$M$</td>
<td>$M_s$</td>
</tr>
<tr>
<td>Condensed Log</td>
<td>$L$</td>
<td>$L_a$</td>
</tr>
<tr>
<td>Enhanced Log</td>
<td>$L$</td>
<td>$E_a$</td>
</tr>
</tbody>
</table>

\[ M = \text{"number of modifications in modification set"} \]
\[ M_s = s(l + r(l-s))M \]

\[ L = pM - q'qM + N(1-u_M) \]
\[ L_a = sL_a + sL_a + s(2-s)L_a \]

\[ E_a = s(L_a + L_a) + (L_a + (l-s)L_a)(sN_u_M + l)(N + L_a + (l-s)L_a) \]

\[ s = \text{"selectivity of snapshot"} \]
\[ p = \text{"ratio of insert modifications"} \]
\[ q = \text{"ratio of delete modifications"} \]
\[ r = \text{"ratio of update modifications"} \]

\[ N = \text{"number of base table tuples before modification"} \]
\[ q' = q(pM)(2N + (p-q)M) \]

\[ M = \prod \frac{p + (q+r)(l-i)(N + (l-i)(p-q)))}{i=1} \]

\[ L_a = pM - q'qM \]
\[ L_a = qM - q'qM \]
\[ L_a = N(1-u_M) - L_a \]

Figure 28. Size Estimates for Logging Methods: Estimates for the number of log entries and the number of refresh messages sent to the snapshot site using a logging method.

The formulas will be used in estimating the overall cost of snapshots based on a logging strategy.

Figure 29, Figure 30, and Figure 31 illustrate the number of refresh messages sent to a snapshot table for the three classes of modification to base tables. The number of refresh messages is given, for some snapshot selectivities, as a function of the modification set size. The modification set size is again expressed as a percentage of the base table size. As can be seen from the figures, the number of refresh messages sent in the case of sequential logging increases linearly with the modification set size.

In the condensed log, the "collision" effect of modifications keeps the number of refresh messages down. This effect increases with the modification set size. A further reduction is obtained using an enhanced log. Note however, that for a full snapshot ($s = 1.0$), en-
Figure 29. **Static Base Table**: The number of log entries qualified for a snapshot with selectivity $s$ given as % of base table size. Dotted lines show sequential logging, dashed lines condensed logging, and solid lines enhanced logging. Note that for $s = 1.0$, the curves for condensed and enhanced logging are the same.

Figure 30. **Incremental Base Table** ($p = 0.1, r = 0.9$): The number of log entries qualified for a snapshot with selectivity $s$ given as % of base table size. Dotted lines show sequential logging, dashed lines condensed logging, and solid lines enhanced logging. Note that for $s = 1.0$, the curves for condensed and enhanced logging are the same.
Figure 31. Dynamic Base Table \((p = 0.2, q = 0.1, r = 0.7)\): The number of log entries qualified for a snapshot with selectivity \(s\) given as \% of base table size. Dotted lines show sequential logging, dashed lines condensed logging, and solid lines enhanced logging.

Enhanced logging has no effect compared with condensed logging except for dynamic base tables, cf. Figure 31. Note also that the effect is more dominant for larger modification sets, given a selective snapshot.

Finally, observe that the reduction obtained for condensed and enhanced logging is relatively smaller for incremental and dynamic base tables than for static base tables, given that the snapshot is selective. The main reason for this is that the number of selected log entries decreases using a sequential log relative to the static case — see the formula for \(M_s\) in Figure 28. At the same time, the number of selected entries increases slightly using a condensed and enhanced log — cf. the formulas relative to a static model for \(L_s\) and \(E_s\) in Figure 28. This trend becomes more noticeable for increasing insert and delete ratios. Still, the reduction in number of refresh messages is significant for large modification sets and for less restrictive snapshots.

As an example of refresh message reduction using condensed or enhanced logging — as opposed to sequential logging — we apply the results shown for the static base table in Figure 29. Given a snapshot with selectivity \(s = 0.50\), and a modification set equal half the number of tuples in the base table, then a shipment equivalent to 37.5 percent of the base table tuples occurs for a sequential log. The corresponding number for condensed logging is 29.5 percent, and 24.7 percent for enhanced logging. In other words, a 21.3 percent reduction is obtained using condensed versus sequential logging, and a 34.1 percent reduction using enhanced logging. The improvement, going from a condensed log to an enhanced log, is 16.2 percent.
5.4 Analytic Model of R* Methods

Unlike the logging approaches, the R* method records the modifications in the base table directly. We recall from "Chapter 4. Differential Refresh Methods" that the R* method identifies the changes made to the table by placing a NULL timestamp, or a NULL previous address in the corresponding columns of the base table. In order to calculate the overall impact of a snapshot mechanism based on the R* method, we need to estimate the number of NULL value settings in the base table, and the number of refresh messages to be shipped.

As NULL values are replaced with real values during refresh processing, the number of NULL value settings gives an estimate for the base table fixup cost. The same NULL values are also used to determine the number of refresh messages that must be shipped to the snapshot table.

In the following, we first give estimates for number of base table tuples that need fixup, and the number of refresh messages in the basic R* algorithm. We then estimate the corresponding numbers for the Enhancement One and Enhancement Two algorithms. At the end of this section, we summarize the expressions given, and show some curves of the number of refresh messages sent for some snapshot selectivities and modification compositions.

5.4.1 Basic R* Algorithm

In the R* method, the modifications are recorded in the base table itself. The way this is done, cause only the after image of each tuple to be accessible after the modification has taken place. The base table will thus, after submission of $M$ modifications, contain $N_M = N + (p-q)M$ tuples with an up-to-date image.

During the refresh create process, base table tuples having either a NULL timestamp, or a NULL previous address, are updated. These tuples have either been inserted, updated or both. We may therefore apply the net numbers as found for the condensed log to express the number of fixups. From the condensed log we have $L_I$ net number of inserts, and $L_u$ net number of updates. We also recall from the fixup operations, defined in Figure 21 on page 59, that base table tuples following an inserted or deleted entry will have their previous address fields changed. This contribution can be estimated by using the same argumentation as applied in finding the number of refresh messages in the enhanced log. We define the following three disjunct events:

- $B_1$: event is tuple deleted from base table.
- $B_2$: event is tuple inserted/updated in base table.
- $B_3$: event is tuple not changed in base table.

Note that deletes have been included in this model even though there is no direct evidence that a tuple is deleted. However, deletes may be detected by consulting the previous address column of the base table, and are therefore implicitly given.
The number of such events are \( L_d, (L_i + L_u) \), and \( Nu_M \) respectively. The total number of events, \( W \), is then:

\[
W = L_d + L_i + L_u + Nu_M = N + L_i
\]

The probability of event \( B_3 \) is thus \( Nu_M/W \), and expresses the probability of finding an unchanged tuple amongst those deleted, updated, inserted and unchanged in the base table. As the net number of deletes performed on the base table is \( L_d \), the number of unchanged base table tuples that will get the previous address updated, is given as \( L_i(Nu_M/W) \).

We may view unchanged entries that follows an inserted base table tuple in exactly the same way. This gives an expression for the number of fixups taken place for unmodified base table tuples that follow an inserted tuple: \( L_i(Nu_M/W) \).

The total number of base table tuples that will be modified by the so called fixup operation, expressed as \( R_F \), is thus:

\[
R_F = L_i + L_u + (L_i + L_d)Nu_M/(N + L_i)
\]

The number of tuples that qualify for shipment is determined next. Like in the enhanced log, we split the refresh messages into two categories; those that concern inserts or updates, and those that are shipped for the sake of deleting a region only. Candidates for the latter class are those unchanged base table tuples that are qualified by the snapshot, and follow a changed region "as seen by" the snapshot.

First, we find the number of refresh messages that concern qualified inserts and updates. As the before image of updates is unknown, both the proportion of inserts and updates that qualify for the snapshot, are given by the selectivity \( s \). Like in the enhanced log, we let \( U \) denote the number of refresh messages of the first category, being:

\[
U = s(L_i + L_u)
\]

The deletes, "as seen by" the snapshot, are not only the real deletes from the base table, but also those updates that may have qualified before and no longer qualify for the snapshot. As it is, the basic algorithm selects all base table tuples with a changed timestamp and that no longer satisfy the snapshot restriction, as being deletes to the snapshot table (see Figure 20 on page 58). In other words, the basic algorithm not only selects the \( L_d \) deletes, and the updates that to the snapshot are deletes, \( (1-s)L_u \), but selects also the new tuples added to the base table that do not satisfy the snapshot, \( (1-s)L_i \). Obviously, the last group could have been discarded.

The total number of entries interpreted as deleted is thus expressed by \( V \) as follows:

\[
V = L_d + (1-s)L_u + (1-s)L_i
\]

We may now define the similar three events as in the enhanced log, and as in finding the \( R_F \) expression. The total number of events, \( W \), in this case is:

\[
W = U + V + sNu_M = N + L_i - (1-s)Nu_M
\]
The probability of finding an unchanged entry, \( B_\gamma \), is thus \( sN_u_M/W \).

The total number of refresh messages sent by the basic \( R^* \) algorithm, \( R_B \), is therefore:

\[
R_B = s(L_d + L_u) + s(L_d + (1-s)L_u + (1-s)L_d) \frac{N_u_M}{(N + L_d + (1-s)L_u)} + 1
\]

Note that the formula encounters an extra refresh message which is always sent to handle deletes in the end region of a base table. This takes place in all of the \( R^* \) methods.

### 5.4.2 Enhancement One

Enhancement One differs from the basic \( R^* \) algorithm in the way the deleted regions are detected. The number of base table tuples that need fixup operation is the same as in the basic algorithm, but the number of refresh messages is smaller due to the enhancement.

As in the basic algorithm, we split the refresh messages that are sent to the snapshot table into two categories. The number of refresh messages belonging to the first category are those that concern qualified inserts and updates. This amounts to \( U = s(L_d + L_u) \), as in the basic algorithm.

The number of refresh messages of the second category is smaller than in the basic algorithm, as the enhanced algorithm disregard new tuples added to the base table in the detection of deleted regions. The number of entries interpreted as deleted to a snapshot, \( V \), is therefore:

\[
V = L_d + (1-s)L_u
\]

This gives a smaller proportion of deletes than in the basic algorithm. Therefore, a smaller proportion of the unchanged, qualified base table tuples will be sent because of a deleted region.

Again we define the three disjunct events. The total number of events, \( W \), in this case is:

\[
W = U + V + sN_u_M = N + sL_d - (1-s)L_u
\]

The total number of refresh messages sent by Enhancement One, \( R_1 \), is therefore:

\[
R_1 = s(L_d + L_u) + s(L_d + (1-s)L_u) \frac{N_u_M}{(N + sL_d - (1-s)L_u)} + 1
\]

### 5.4.3 Enhancement Two

Enhancement Two applies a qualification flag to record the membership of each base table tuple, i.e. the flag is set if the tuple is reflected in the snapshot table. Maintaining this flag can be done together with the maintenance of the timestamp and previous address fields. In this way, the enhancement will require the same amount of fixup oper-
ation as the basic R* algorithm. The number of refresh messages is reduced when compared with both the basic and the Enhancement One algorithm.

Again we split the refresh messages into two categories. First, Enhancement Two will only send updates that have true relevance to the snapshot table, i.e. the selection is accurate as in the condensed log. The situation is actually identical to the one accounted for in the enhanced log where \( U \) denoting the number of inserts and updates, is defined as:

\[
U = s(L_a + L_u)
\]

Secondly, the number of refresh messages of the second category is smaller than by applying Enhancement One, as we may disregard updates with a before image not qualified by the snapshot by checking the flag. The number of entries interpreted as deleted to a snapshot, \( V \), is then:

\[
V = L_d + s(1-s)L_u
\]

This gives a smaller proportion of deletes, and consequently a smaller proportion of the unchanged, qualified base table tuples will be sent as caused by a deleted region.

\( W \) in this case is:

\[
W = U + V + sN_{M} = s(N + L_a + (1-s)L_u) + (1-s)L_d
\]

The total number of refresh messages sent by Enhancement Two, \( R_2 \), is therefore:

\[
R_2 = s(L_a + L_u) + s(L_d + s(1-s)L_u) N_{M} / (s(N + L_a + (1-s)L_u) + (1-s)L_d)) + 1
\]

### 5.4.4 Summary

The estimates for R* methods are summarized in Figure 32. The figure lists the formulas for expressing the number of base table tuples that need fixup operation, and the number of refresh messages that are needed for each of the algorithms based on the R* method. As in the logging approach, the formulas are expressed as a function of the modification type ratios \((p, q, r)\), the modification set size \((M)\), the base table size prior to modification \((N)\), and the selectivity of the snapshot \((s)\). The expressions are used in estimating the overall cost of snapshots based on the R* method.

Figure 33, Figure 34, and Figure 35 illustrate the number of refresh messages sent to a snapshot table for the three classes of base table modification. The number of refresh messages is given, for some snapshot selectivities, as a function of the modification set size. The modification set size is again expressed as a percentage of the base table size. As can be seen from the figures, the three algorithms select the same number of tuples for a full-copy snapshot \((s = 1.0)\), whereas the methods differ for selective snapshots.

In the static base table case, shown in Figure 33, Enhancement One has no effect since no insertion takes place. Enhancement Two, on the other hand, will reduce the number
### R* Method Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Fixups</th>
<th>Refresh Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Algorithm</td>
<td>$R_F$</td>
<td>$R_B$</td>
</tr>
<tr>
<td>Enhancement One</td>
<td>$R_F$</td>
<td>$R_1$</td>
</tr>
<tr>
<td>Enhancement Two</td>
<td>$R_F$</td>
<td>$R_2$</td>
</tr>
</tbody>
</table>

#### Definition of Symbols and Formulas

\[
\begin{align*}
R_F &= L_4 + L_u + (L_4 + L_d)Nu_M(N + L_4) \\
R_B &= s(L_4 + L_u) + s(L_d + (1-s)L_u + (1-s)L_4) Nu_M/(N + L_4(1-s)Nu_M) + 1 \\
R_1 &= s(L_4 + L_u) + s(L_d + (1-s)L_u) Nu_M/(N + sL_d(1-s)Nu_M) + 1 \\
R_2 &= s(L_4 + L_u) + s(L_d + (1-s)L_u) Nu_M/(s(N + L_4 + (1-s)L_u) + (1-s)L_d)) + 1 \\
L_i &= pM - q'qM + N(1-u_M) \\
L_4 &= sL_4 + sL_d + s(2-s)L_u \\
s &= \text{"selectivity of snapshot"} \\
p &= \text{"ratio of insert modifications"} \\
q &= \text{"ratio of delete modifications"} \\
r &= \text{"ratio of update modifications"} \\
N &= \text{"number of base table tuples before modification"} \\
q' &= q(pM/(2N + (p-q)M)) \\
\prod_{i=1}^{M} u_M &= \prod_{i=1}^{M} (p + (q+r)((i-1)(N + (i-1)(p-q)))) \\
L_4 &= pM - q'qM \\
L_d &= qM - q'qM \\
L_u &= N(1-u_M) - L_d
\end{align*}
\]

**Figure 32. Size Estimates for R* Methods:** Estimates for the number of base table tuple fixup and the number of refresh messages sent to the snapshot site using an R* based method.

of refresh messages somewhat for very restrictive snapshots, and a significant reduction may be obtained for less restrictive snapshots.

In the incremental and dynamic base table case, see Figure 34 and Figure 35 respectively, there is no significant difference between Enhancement One and the basic algorithm. Then again, the insert ratio is fairly low in these two cases ($p = 0.1$ and $p = 0.2$). Enhancement Two shows about the same reduction in these cases as in the static base table case.
Figure 33. **Static Base Table**: The number of tuples qualified for a snapshot with selectivity $s$ given as % of base table size. Dashed lines show the basic $R^*$ algorithm and solid lines Enhancement Two. Note that for $s = 1.0$, the curves for the basic algorithm and Enhancement Two are the same. Enhancement One selects the same number of tuples as the basic algorithm.

Figure 34. **Incremental Base Table** ($p = 0.1, r = 0.9$): The number of tuples qualified for a snapshot with selectivity $s$ given as % of base table size. Dotted lines show the basic $R^*$ algorithm, dashed lines Enhancement One and solid lines Enhancement Two. Note that for $s = 1.0$, the curves for the three algorithms coincide. For $s = 0.05$, Enhancement One and basic algorithm give virtually identical results.
Figure 35. Dynamic Base Table ($p = 0.2$, $q = 0.1$, $r = 0.7$): The number of tuples qualified for a snapshot with selectivity $s$ given as % of base table size. Dotted lines show the basic R* algorithm, dashed lines Enhancement One, and solid lines Enhancement Two. Note that for $s = 1.0$, the curves for the three algorithms coincide. For $s = 0.05$, Enhancement One and basic algorithm are virtually identical.

Figure 36. Incremental Base Table ($p = 0.5$, $r = 0.5$): The number of tuples qualified for a snapshot with selectivity $s$ given as % of base table size. Dotted lines show the basic R* algorithm, dashed lines Enhancement One and solid lines Enhancement Two. Note that for $s = 1.0$, the curves for the three algorithms coincide. For $s = 0.05$, Enhancement One and basic algorithm give virtually identical results.
An increase in the ratio of inserts will favour the enhancements to the basic R* algorithm. Figure 36 shows the number of tuples qualified when every other modification is a tuple insertion, i.e. \( p = 0.5 \). Again, there is no significant difference in the number of tuples selected — for very restrictive snapshots — using the basic algorithm and Enhancement One.

As an example of the message reduction using Enhancement Two in comparison with the basic R* algorithm, we apply the results shown for the static base table in Figure 33. Given a snapshot with selectivity \( s = 0.50 \), and a modification set equal half the number of tuples in the base table, then 28.3 percent of the base table tuples will be shipped using the basic algorithm. The corresponding number for Enhancement Two is 24.7 percent. In other words, a 12.7 percent reduction is obtained using the enhanced refresh algorithm for this case.

Finally, we comment on the estimate for the enhanced log given in Figure 28, and the estimate for the R* Enhancement Two in Figure 32. Both methods obtain a reduction in the number of refresh messages by sending deletes together with insert and update messages. The methods differ in their ability to determine if a real deleted tuple belongs to a snapshot or not. This can only be done in the enhanced log method. Consequently, the enhanced log may send a marginally smaller number of refresh messages in the case of dynamic base tables. For the other two types of base table update, the two methods select virtually the same number of refresh messages.

### 5.5 Estimating Overall Cost of Logging Methods

We will now find expressions that estimate the overall cost of maintaining database snapshots based a logging strategy. The cost will include both the logging cost and the cost associated with the refresh of a database snapshot, i.e. all of the cost contributing factors as identified in Figure 5 on page 30. In other words, both the refresh processing cost as well as the communication cost are included.

The cost estimates presented here rely on the size estimates for the various log strategies. These are listed in Figure 28.

In “Chapter 4. Differential Refresh Methods”, several implementations of the logging methods are discussed.

First, the sequential log can either be processed in a sequential manner at snapshot refresh (basic refresh), or we may sort the log as the first step in the refresh create process (sort merge refresh). In the last case, we obtain a log with the same condensed property as the condensed log, and thus a reduction in the number of refresh messages sent.

Secondly, the condensed log keeps the log in a condensed form throughout the modification set. This can be obtained either by keeping the log sorted at all times, e.g. by using B-trees, or by applying a direct access method that can handle a continuously growing log, e.g. a linear hashing access method.

Finally, the enhanced log may be realized either on top of a sorted sequential log, or a condensed log using B-trees. The enhancement will reduce the number of refresh messages, but requires a condensed, sorted image of the log in order to be useful.
The following assumptions are made for the cost estimates given. By processing cost we mean the cost of reading and writing log entries to disk storage. Basically, this means that the processing cost is measured in terms of disk page I/O, and that the CPU cost is either included in this disk unit cost, or ignored. The cost of reading or writing a disk page is defined by \( t_d \). In order to simplify the cost estimation, we assume that the unit disk I/O cost is the same at the base table and the snapshot table site.

As for the communication cost, the unit cost of shipping one refresh message is defined by \( t_r \). This unit cost include the cost of copying data to and from communication buffers, etc., as well as transmission time.

### 5.5.1 Sequential Log

The overall cost of snapshot maintenance, using the sequential logging approach, is determined by the cost of logging entries, the cost of retrieving qualified entries in the refresh create process, plus the cost of shipping the qualified entries to the snapshot site, and the cost of updating the snapshot table.

In the sequential log, each modification submitted to the base table is appended to the current sequence of log entries. Each entry on the log keeps a record of the before and after image of the modified base table tuple. We define \( t_{SL} \) as the cost of adding an entry to the sequential log.

We assume that several log entries of the sequential log will fit into one disk page. Let \( b \) denote the maximum number of entries per disk page. A sequential log of \( M \) entries will thus occupy \( \text{CEIL}(M/b) \) disk pages.\(^7\) The cost of retrieving qualified log entries during the refresh create process is proportional to the cost of reading these disk pages.

The number of log entries relevant to a snapshot table with selectivity \( s \) is expressed by \( M_s \), see Figure 28. We assume that each entry is sent as an individual refresh message at cost \( t_r \).

At the snapshot site, the snapshot table is updated in the same sequence as the refresh messages are received. Since the modifications are uniformly distributed over the snapshot table, we may assume that each modification will cause one read of a disk page in order to find the snapshot tuple to be updated, followed by a write. In other words, the cost is \( 2t_d \) per received refresh message.

The cost of sequential logging, \( C_{SL} \), may thus be expressed as:

\[
C_{SL} = Mt_{SL} + \text{CEIL}(M/b)t_d + M_t t_r + M_s 2t_d
\]

The cost of adding an entry to the sequential log, \( t_{SL} \), is typically one read followed by a write to the log, i.e. \( t_{SL} = 2t_d \). In situations where the base table is updated frequently,

\(^7\) CEIL is the ceiling function.
the cost can sometimes be reduced (by the effect of having the last log page held in memory).

5.5.2 Sort Merge of Sequential Log

The objective of performing a sort merge of a sequential log, is to obtain a reduction in communication cost at the expense of an increased refresh create cost.

The sequential log may be sorted using the replacement selection algorithm described in [KNUT 73], and a merge pattern algorithm. We let $k$ denote the merge ratio of the merge pattern algorithm, i.e. the number of files that can be merged in one step. The same $k$ does also denote the sort space available during the initial sort step.

Given a sequential log of $M$ entries, with a maximum of $b$ entries per disk page, then the number of merge steps, $x$, equals the depth of a $k$-tree, i.e.

$$x = \log_k(\lceil M/(2kb) \rceil)$$

The entire volume of the sequential log may, because of the sort merge process, be read and written to disk several times during the refresh create process. First, the log is read and initial files for sort merge are established. Next, these files are read and a new, smaller set of files are produced. This is repeated $x$ number of times until one sorted file is obtained. Finally, the sorted log is scanned to produce the refresh messages. In total, the volume is read/written $2(1 + x) + 1$ number of times.

So far we have assumed that the entire log volume is sorted. As pointed out in “Chapter 4. Differential Refresh Methods”, the overhead of the sort merge step becomes extremely high for very restrictive snapshots. We also showed that the number of entries can be reduced prior to sorting by selecting only those entries qualified by the snapshot restriction. Thus, only $M_t$ number of entries need to be sorted, and the sort merge step may proceed as follows:

First, the entire log volume is read. During the scan of the log, only entries satisfying the snapshot restriction are selected for initial sorting, i.e. the volume is reduced to $\lceil M_t/b \rceil$ pages. Clearly, the number of merge steps becomes less, especially for very restrictive snapshots, as $M_t$ replaces $M$ in the formula given for $x$.

A unit sort cost, $t_{SM}$, expressing the number of times the sort volume of $M_t$ log entries is read/written by the merge process, is therefore defined as follows:

$$t_{SM} = (\log_k(\lceil M_t/(2kb) \rceil) + 1)2t_d$$

In the sort merge refresh strategy, the cost of logging is determined by the sequential property of the log. The cost of establishing the refresh messages is higher than in the basic refresh strategy due to the sort operation just described. On the other hand, the cost of communicating refresh messages may be far less as the number of such may be reduced. As the condensed property of the condensed log is obtained, the number of refresh messages is $L_r$. Finally, the snapshot table is updated in primary key order as the refresh messages are sent based on a key ordered sequential log.
The overall cost of snapshot maintenance using a sort merge strategy on a sequential log, $C_{SM}$, is thus given as follows:

$$C_{SM} = M t_{SL} + \text{CEIL}(M/b) t_d + \text{CEIL}(M_d/b) t_{SM} + L t_s + \text{MIN}(L, \text{CEIL}(sN/bf)) t_d$$

Note that the size estimate of the condensed log is used in the term expressing the shipping cost. Note also that the update of the snapshot table cause $L$ number of disk read/writes for small number of refresh messages, and that the snapshot update cost never exceed the cost of reading and writing the entire snapshot table. The $f$ in the last term of the $C_{SM}$ expression denotes the load factor of the snapshot table. Also, the maximum number of tuples per page in the snapshot table is $b$.

When comparing the cost formulas for sequential logging, $C_{SL}$ and $C_{SM}$, we may conclude that the sort merge processing step is profitable if the reduction in communication and snapshot update cost outweighs the increased cost of the sort merge. We will return to this in the cost analysis later on.

A final remark should be made to the cost expression given. The unit sort cost, $t_{SM}$, increases by two for each new step of the sort merge process. The number of merge steps is determined by the number of relevant entries in the log, and the merge ratio $k$. In other words, an increase in log size can be met by an increased merge ratio so as to keep the unit sort cost down. To be precise, not only the merge ratio may be increased to obtain this effect, we may increase the internal sort space by using a larger bucket size, i.e. the internal sort space may be sized to accommodate a higher number of log entries at a time. This can be visualized as a higher number of entries per page, $b$, than what is used in the sequential log itself.

### 5.5.3 Condensed Log Using B-Trees

In the following, we discuss a condensed log based on a B-tree implementation. The B*-variant of B-trees is chosen. This variant is used as storage structure in several relational database management systems, e.g. in MIMER [MIME 85]. The condensed log may then be stored using the same access method as used for the tables in the database.

The cost estimate for condensed logging is expressed similar to the one for sort merge of a sequential log, and is defined by $C_{CB}$ as:

$$C_{CB} = L t_{IB} + (M-L) t_{UB} + \text{CEIL}(L/bf) t_d + L t_s + \text{MIN}(L, \text{CEIL}(sN/bf)) 2t_d$$

in which $t_{IB}$ is the unit cost of inserting an entry into a condensed log based on a B*-tree, $t_{UB}$ is the unit cost of changing an existing entry on the log, $b$ is the maximum number of entries per page, and $f$ is the load factor of the B*-tree.

We have split the logging cost in two: The cost introduced when adding the first modification of a base table tuple to the condensed log, and the cost involved when changing an existing log entry. The last two terms of the cost expression are equal to those of the sort merge expression defined above.
The unit cost \( t_{UB} \) depends on two factors: The depth of the B* index tree, and the number of page splits that have taken place during the writing of the condensed log. The unit cost \( t_{UB} \) depends on the depth of the index tree only.

A modification is entered to the condensed log as follows. First, the log is checked to verify if this is the first update to the given base table tuple or not. Independent of the outcome, the index tree is traversed top down leading to the actual data page — causing a corresponding number of disk page reads. Next, a new entry is added the log if this is the first modification to the base table, or else the existing entry on the log is updated. Both cases cause one disk page write. In the case where a new entry is inserted, additional disk page writes may become necessary if the actual data page is full, i.e. page split. Thus, the unit cost \( t_{UB} \) consists of index reads, data page read and write, and possibly additional read/writes in the case of page splits. The unit cost \( t_{UB} \), on the other hand, consists of index reads, and data page read and write.

For a small log, \( 3t_d \) seems a good estimate for \( t_{UB} \), and \( 4t_d \) for a larger log. In some database management systems — for instance MIMER [MIME 85] — the depth of index trees can be kept relatively small. In situations where the base table is updated frequently, the cost can be reduced by keeping the root page of the log index in memory.

As for the unit cost of log insert, \( t_{IB} \), the cost of page splits will cause a noticeable overhead. After all, the condensed log is continuously increasing which means a page split for approximately every \((bf)\) entry added to the log.

The number of page splits that have taken place during the \( M \) modifications submitted, equals the number of index and data pages contained in the B*-tree of the log minus the one page we started out with. This number can be determined exactly, given the number of records in the B*-tree. However, as the depth of the index tree can be kept low, we may use the following approximation for page split overhead,

\[
\text{CEIL}(L/bf) + \text{CEIL}(\text{CEIL}(L/bf)/bf)\]

where \( L \) is the number of entries in the condensed log, \( b_i \) denotes the maximum number of index references in an index page of B*-trees, and \( f \) is the load factor. Only the leaf level index pages, i.e. those referring to the data pages, are counted for in the approximated expression. The added cost of page split per inserted entry is therefore in average \((1/bf)(1 + (1/bf))\). As each page split cause two extra writes, the average unit cost can be expressed as:

\[
t_{IB} = t_{UB} + (1/bf)(1 + (1/bf))2t_d
\]

When relating the cost expression \( C_{CB} \), with the one found for the basic refresh of a sequential log \( C_{SL} \), the cost contributing factors are very different. The normal processing cost is much higher in the condensed logging approach. However, the actual refresh cost is reduced in comparison with the basic sequential logging approach. For a start, the number of entries kept in the condensed log, \( L \), is less. The cost of refresh create is thus proportional to \( L \). Furthermore, the amount of refresh messages sent is reduced which decrease the cost of refresh update processing. The update processing in itself is simpler in the condensed logging approach due to the fact that the refresh messages are received in primary key order. The snapshot table is updated in key order sequence, and the cost
of updating the snapshot table will therefore not exceed the cost of reading and writing the entire snapshot table.

5.5.4 Condensed Log Using Linear Hashing

An alternative implementation of the condensed log is to use a dynamic, hash based access method as described in “Chapter 4. Differential Refresh Methods”. A key ordering of the condensed log is not needed to maintain the condensed property of the log method. However, the hash method must provide a dynamic storage structure that can grow with an increasing modification set. Such a storage structure is known as linear hashing or dynamic hashing [LITW 80].

The overall cost of snapshot maintenance using a condensed log based on linear hashing is defined by $C_{CH}$ as;

$$C_{CH} = L_t t_{lh} + (M-L)t_{uh} + \text{CEIL}(L/b)f t_d + L_s t_s + L_w 2t_d$$

in which $t_{lh}$ is the unit cost of inserting an entry into a condensed log based on linear hashing, $t_{uh}$ is the unit cost of changing an existing entry on the log, $b$ is the maximum number of entries per page, and $f$ is the load factor of linear hashing.

Again, we distinguish between the cost of placing a new modification on the log and the cost of changing an existing log entry. This is done for the same reason as in the B-tree implementation, as inserting an entry using linear hashing may cause an overhead introduced by dynamic reorganization of the file. The unit cost of insert is therefore usually much higher than the unit cost of update. Values for $t_{lh}$ and $t_{uh}$ are given in [LITW 80].

Note that in this implementation of the condensed log, we may obtain a reduction in the logging cost. On the other hand, the refresh update cost is increased. The refresh messages are no longer sent in primary key order, and the update of the snapshot table becomes “random” as in the basic refresh of sequential log.

5.5.5 Enhanced Log

The enhanced log may be viewed as an extension to some of the basic methods described above. We recall from “Chapter 4. Differential Refresh Methods” that the enhancement requires a sequential scan of the base table as well as the log. Thus, the method is only useful when the log is sorted on the base table’s primary key. This excludes the basic refresh of sequential log, and the linear hashing realization of the condensed log.

We may therefore apply the enhancement on a sequential log that has been sorted by the sort merge strategy, or the B-tree based condensed log. The overall cost may be expressed as a modification to the expressions given for these two “basic” methods.

First, the logging is performed by the “basic” method so no change is introduced.
Log Method Summary

<table>
<thead>
<tr>
<th>Log Method</th>
<th>Implementation</th>
<th>Overall Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential Log</td>
<td>Basic Refresh</td>
<td>$C_{SL}$</td>
</tr>
<tr>
<td></td>
<td>Sort Merge Refresh</td>
<td>$C_{SM}$</td>
</tr>
<tr>
<td>Condensed Log</td>
<td>B-tree</td>
<td>$C_{CB}$</td>
</tr>
<tr>
<td></td>
<td>Linear Hashing</td>
<td>$C_{CH}$</td>
</tr>
<tr>
<td>Enhanced Log</td>
<td>Sort Merge</td>
<td>$C_{ES}$</td>
</tr>
<tr>
<td></td>
<td>B-tree</td>
<td>$C_{EB}$</td>
</tr>
</tbody>
</table>

Definition of Symbols and Formulas

\[ C_{SL} = M_{SL} + \text{CEIL}(M/b)t_d + M_t + M_s2t_d \]
\[ C_{SM} = M_{SM} + \text{CEIL}(M/b)t_d + \text{CEIL}(M_s/b)t_{SM} + L_t + \text{MIN}(L_s, \text{CEIL}(sN/bf))2t_d \]
\[ C_{CB} = L_{t_B} + (M-L)t_{UB} + \text{CEIL}(L/bf)t_d + L_t + \text{MIN}(L_s, \text{CEIL}(sN/bf))2t_d \]
\[ C_{CH} = L_{t_H} + (M-L)t_{BU} + \text{CEIL}(L/bf)t_d + L_t + L_s t_{2d} \]
\[ C_{ES} = C_{SM} + \text{CEIL}((N + (p-q)M)/bf)t_d - (L_s-E_s)t_d \]
\[ C_{EB} = C_{CB} + \text{CEIL}((N + (p-q)M)/bf)t_d - (L_s-E_s)t_d \]

\[ b = "\text{maximum number of entries in data page}" \]
\[ b_t = "\text{maximum number of entries in index page}" \]
\[ f = "\text{load factor}" \]
\[ t_d = "\text{unit disk I/O cost}" \]
\[ t_s = "\text{unit message delivery cost}" \]
\[ t_{SL} = "\text{unit cost of sequential logging; typically 2t_d}" \]
\[ t_{SM} = (\log_2(\text{CEIL}(M_s/2kb)) + 1)2t_d \]
\[ t_B = t_{UB} + (1/bf)(1 + (1/bf))2t_d \]
\[ t_{UB} = "\text{unit cost of B-tree update; 3t_d for small logs, 4t_d for larger logs.}" \]
\[ t_{IH} = "\text{unit cost of Linear Hashing insert}" \]
\[ t_{HU} = "\text{unit cost of Linear Hashing update}" \]

Figure 37. Cost Estimates for Logging Methods: The overall cost of database snapshots using a logging method expressed as a function of the log lengths and the number of entries that qualify for the snapshot given in Figure 28.

Secondly, the refresh create process will issue a full scan of the base table, as well as a repeated, limited scan of the log. If we assume that this cause no re-reading of log pages, the additional cost, when compared with the “basic” method, is the cost of reading the base table.

Thirdly, the number of the refresh messages sent is smaller compared with the “basic” methods. Instead of sending $L_s$ messages, $E_s$ messages are sent, see Figure 28.
Finally, although the number of refresh messages is less than in the "basic" method, the same number of snapshot table changes are required. The refresh update cost is therefore unchanged.

The overall cost of an enhanced log based on a sort merge refresh of a sequential log, is therefore;

\[ C_{ES} = C_{SM} + \text{CEIL}(N + (p-q)M)/bf)t_d - (L_xE_x)t_s \]

If the "basic" method is a B-tree based condensed log, we have:

\[ C_{EB} = C_{CB} + \text{CEIL}(N + (p-q)M)/bf)t_d - (L_xE_x)t_s \]

As can be seen from the formulas, the enhanced logging method is profitable only if the reduction in the communication cost outweighs the cost of reading the base table.

5.5.6 Summary

The cost estimates are summarized in Figure 37. The length of the various logs and the number of entries that qualify for shipment are taken from Figure 28.

There is an additional cost element not counted for in any of the estimates given. This is the overhead imposed at the snapshot site when the snapshot table is expanded due to insertion of new tuples. This overhead is identical in the alternatives discussed as we basically assume the same access method used for the snapshot table in each of the alternatives. This is done so as to obtain a fair cost comparison, and because of this, we may omit this cost element.

The net addition to the snapshot table is \( s(p-q)M \) tuples. Thus, the number of data page writes will amount to \( \text{CEIL}(s(N + (p-q)M)/bf) \) instead of \( \text{CEIL}(sN/bf) \) as in the given cost formulas.

5.6 Estimating Overall Cost of R* Methods

The overall cost of snapshot maintenance based on an R* method is calculated similarly. There is one major difference when compared with the cost calculation shown for the logging approaches: The processing overhead on the normal base table operations is virtually nil and is therefore ignored. The overall cost of database snapshots using an R* method therefore stems from the refresh processing alone.

The cost estimates rely on the size estimates given for the various refresh algorithms based on the R* method, see Figure 32. The cost estimates include both processing and communication cost. The assumptions are the same as in the cost calculation of the logging approach.
5.6.1 Basic R* Algorithm

As the cost of marking the modifications made to the base table is ignored, the factors contributing to the overall cost of snapshot maintenance are the following:

First, the refresh create process will issue a full scan of the base table, and replace NULL values with real values (i.e. base table fixup). As modifications are uniformly distributed over the base table, we may assume that fixups are evenly spread over the base table. Each update may thus cause a separate write, but the total number of writes will not exceed the number of pages allocated to the base table. The cost of the refresh create step, $C_{RC}$, is thus:

$$C_{RC} = \left\lceil \frac{(N + (p - q)M)}{bf} \right\rceil t_d + \min(R_F, \left\lceil \frac{(N + (p - q)M)}{bf} \right\rceil )t_d$$

In the formula, $b$ is the maximum number of entries per page in the R* method, and $f$ is the load factor.

Secondly, refresh messages are shipped to the snapshot table. Size estimates are given in Figure 32.

Finally, the refresh update is performed in much the same way as in the condensed log approach. Even though the number of refresh messages is less in the R* method, the same number of changes will be needed to the snapshot table. A refresh message in the R* approach may semantically contain several modifications. The update cost is therefore proportional to the number of refresh messages sent in the condensed log approach, $L_u$. As in the condensed log based on a B-tree, the number of read/writes will not exceed the number of pages allocated to the snapshot table.

We may now establish an expression for the overall cost of database snapshots using the basic R* refresh algorithm. The cost is expressed by $C_{RB}$ as follows:

$$C_{RB} = C_{RC} + R_Bt_d + \min(L_u, \left\lceil \frac{sN}{bf} \right\rceil )2t_d$$

5.6.2 Enhancement One

The cost calculation of Enhancement One is identical to the one given for the basic algorithm. The only difference is that the number of refresh messages using Enhancement One is smaller. The cost is expressed by $C_{R1}$ as follows:

$$C_{R1} = C_{RC} + R_1t_d + \min(L_u, \left\lceil \frac{sN}{bf} \right\rceil )2t_d$$

5.6.3 Enhancement Two

Enhancement Two introduces an even smaller communication overhead than Enhancement One, without increasing the other cost factors. The overall cost of snapshot maintenance using Enhancement Two, $C_{R2}$, is thus:

$$C_{R2} = C_{RC} + R_2t_d + \min(L_u, \left\lceil \frac{sN}{bf} \right\rceil )2t_d$$
R* Method Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Overall Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Algorithm</td>
<td>$C_{RB}$</td>
</tr>
<tr>
<td>Enhancement One</td>
<td>$C_{R1}$</td>
</tr>
<tr>
<td>Enhancement Two</td>
<td>$C_{R2}$</td>
</tr>
</tbody>
</table>

Definition of Symbols and Formulas

$$
C_{RB} = C_{RC} + R_B t_d + \text{MIN}(L_v, \text{CEIL}(sN/bf))2t_d
$$

$$
C_{R1} = C_{RC} + R_1 t_d + \text{MIN}(L_v, \text{CEIL}(sN/bf))2t_d
$$

$$
C_{R2} = C_{RC} + R_2 t_d + \text{MIN}(L_v, \text{CEIL}(sN/bf))2t_d
$$

$$
C_{RC} = \text{CEIL}((N + (p-q)M)/bf)t_d + \text{MIN}(R_F, \text{CEIL}((N + (p-q)M)/bf))t_d
$$

$b$ = "maximum number of entries in data page"

$f$ = "load factor"

$t_d$ = "unit disk I/O cost"

$t_s$ = "unit message delivery cost"

Figure 38. Cost Estimates for R* Methods: The overall cost of database snapshots using an R* method expressed as a function of the number of entries that qualify for the snapshot given in Figure 32.

5.6.4 Summary

The cost estimates are summarized in Figure 38. The number of entries that qualify for shipment are taken from Figure 32.

Like in the logging methods, there is an additional cost element which stems from insertions of tuples in the snapshot table. In order to ensure a fair cost comparison between logging methods and R* based methods, we assume the same access method to be used for the snapshot table. Thus, the overhead will be identical to the one introduced by the logging methods. Again, we may omit this cost element.

5.7 Analytic Results

The objective of the analysis is to detect how the various refresh methods differ in terms of cost. To obtain a fair comparison we therefore assume that the base table and the snapshot table are stored using the same access method. To simplify matters, we assume a B*-tree structure as suggested for the condensed log using B-tree.
Some of the refresh methods will perform relatively better for a small modification set than for a large one. This is the case using the sort merge alternative of sequential logging. The number of merge steps increases as the log size increases unless measures are taken. A similar progressive increase in cost may occur using B-trees for a condensed log. For these reasons, the following parameter values are chosen for the analysis:

1. The number of base table tuples prior to modification; \( N = 100,000 \).

2. The maximum number of entries (or tuples) per data page; \( b = 10 \).

3. In a B*-tree, the maximum number of index entries per index page; \( b_i = 146 \), and the load factor; \( f = 0.83 \). These figures are taken from [MIME 85].

4. In linear hashing, the following figures are used, taken from [LITW 80]; \( f = 0.61 \), \( t_{th} = 3.14 t_d \), \( t_{uh} = (1.07 + 1) t_d \).

5. In the sort merge refresh strategy, the merge ratio is \( k = 10 \) unless otherwise stated.

We have analyzed the refresh methods, including alternative implementations, under various base table modification conditions and for different snapshot selectivities.

In the analysis, we have assumed the message transfer cost to be equal to disk I/O cost, i.e. \( t_r = t_d \). 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.

The result from the analysis is given as a trade off in cost between each alternative, and the basic refresh of the sequential logging approach — hereafter referred to as basic sequential logging. As we have assumed that \( t_r = t_d \), the cost is measured in terms of disk and message I/O units. Each curve represents the cost difference between sequential logging and the alternative at hand, i.e. \( C_{SL} - C_r \) where \( C_r \) is the refresh method being compared. The curves are drawn for increasing modification sets, and the modification set size is given as a percentage of the base table size.

The results are presented for each class of base table modification. First, we present a cost comparison of the refresh methods, given a static base table. From the examples shown, we will see that some of the refresh performs significantly worse than others. We have therefore omitted these from the cost comparison performed for the other two modification classes.

The results given here will be evaluated in the next section.

### 5.7.1 Static Base Table

A static base table is only exposed to tuple updates, i.e. no tuple insertions and deletions occur.

Figure 39 shows the performance of a sort merge of a sequential log compared with the basic refresh strategy of the same log. The figure gives the trade off in I/O units of applying a sort merge operation or not, and shows the effect of using a sort space of 100
Figure 39. Static Base Table (Case 1): Sort Merge of sequential log. The curves give the cost trade off in I/O units as % of base table size. The section of a curve above the reference line means that sort merge is profitable, whereas negative values mean that the basic sequential logging wins.

and 300 tuples respectively. The effect of increasing the sort space is marginally for very restrictive snapshots, but cause a slight improvement for the less restrictive snapshots.
Figure 39 shows that the sort merge strategy is profitable when the modification set exceeds 9 to 16 percent of the base table size, given a small sort space, and 9 to 14 percent for the larger sort space.

In Figure 40, the two alternative implementations of the condensed log are compared with the basic sequential log. Both alternatives perform significantly worse than sequential logging, especially for restrictive snapshots. In general, linear hashing performs best out of the two alternatives, mainly because of less logging overhead. A similar trend will be found when applying the condensed logging alternatives to incremental and dynamic base tables. In both cases, condensed logging performs relatively worse, compared with sequential logging, than in the static base table case.

Figure 41 shows the performance of the R* based methods compared with basic sequential logging. Only the basic R* algorithm and Enhancement Two are shown on the figure, as Enhancement One causes the same number of refresh messages as the basic algorithm, cf. Figure 33 on page 88. The R* based algorithms clearly loose for small modifications sets, mainly due to the full scan of the base table. However, for modifications sets exceeding 8.5 to 11.5 percent of the base table size, the R* methods performs significantly better than basic sequential logging. Note also that Enhancement Two performs slightly better than the basic R* algorithm, except for full-copy snapshots where they perform identically.

When comparing Figure 39 and Figure 41, we see that the R* methods performs better than the sort merge strategy for modifications set exceeding approximately 15 percent.
Figure 41. Static Base Table (Case 3): R* Method. The curves give the cost trade off in I/O units as % of base table size. The section of a curve above the reference line means that the R* method wins, whereas negative values means that the basic sequential logging wins. Solid lines are Enhancement Two, dashed lines are Basic algorithm. Dotted line is for selectivity s = 1.0.

of the base table size. The sort merge strategy performs significantly better for the small modification sets.
Figure 42. Incremental Base Table ($p = 0.1, r = 0.9$, Case 1): Sort Merge of sequential log. The curves give the cost trade off in I/O units as % of base table size. The section of a curve above the reference line means that sort merge is profitable, whereas negative values means that the basic sequential logging wins. The curves are for a sort merge ratio $k = 10$.

Enhanced logging is not shown, simply because the method has no cost reducing effect on the underlying methods, unless for very large modifications sets (1.5 to 2 times the base table size).

5.7.2 Incremental Base Table

We now present the results for an incremental base table, i.e. a table in which tuples may be added as well as updated. Only the sort merge strategy and the $R^*$ methods are shown, as the condensed and enhanced log give no improvements compared with a static base table.

Figure 42 shows the performance of sort merge versus the basic refresh algorithm. The figure pictures approximately the same trade off as for the static base table: The sort merge is profitable when the modifications set exceeds 9 to 16 percent of the base table size.

The benefit of using sort merge is slightly less than in the static base table case caused by the fact commented on in Figure 30 on page 81. The number of refresh messages using sort merge is the same as in the condensed log which increases for an incremental base table. At the same time, the number of refresh messages sent using basic refresh decreases. Thus, the cost difference of communicating refresh messages becomes smaller for the two alternatives. This difference shrinks as the insert ratio increases, but the sort merge refresh strategy will still reduce the overall cost of snapshot maintenance for
Figure 43. Incremental Base Table ($p = 0.1$, $r = 0.9$, Case 2): R* Method. The curves give the cost trade off in I/O units as % of base table size. The section of a curve above the reference line means that the R* method wins, whereas negative values mean that the basic sequential logging wins. Solid lines are Enhancement Two, dashed lines Enhancement One, and dotted lines basic R* algorithm. Curves for $s = 1.0$ are not shown, but lies between those given for $s = 0.25$ and $s = 0.5$ for the basic algorithm up to 20% of base table size. Above 20%, this curve inclines the most.

modification sets exceeding 10 to 17 percent of base table size (numbers taken for insert ratio $p = 0.5$).
Figure 44. **Dynamic Base Table** ($p = 0.2$, $q = 0.1$, $r = 0.7$, Case 1): Sort Merge of sequential log. The curves give the cost trade off in I/O units as % of base table size. The section of a curve above the reference line means that sort merge is profitable, whereas negative values means that the basic sequential logging wins. The curves are for a sort merge ratio $k = 10$.

This implies that the reduction in refresh messages can not be the only distinguishing cost factor. The reduction in cost of updating the snapshot table plays an important role. In the sort merge strategy, the snapshot table is updated in a key ordered fashion, whereas the basic refresh cause a "random" update of the snapshot tables tuples. This will reduce the cost for sort merge in comparison with the basic refresh strategy. We will return to this when evaluating the results.

Figure 43 shows the performance of the R* based refresh algorithms related to the basic sequential log. Again, the R* methods wins for modification sets exceeding 9 to 11 percent of base table size, depending on which out of the refresh algorithms used. The figure shows that applying the Enhancement Two algorithm will provide us with the largest profits, whereas the basic and Enhancement One perform approximately the same. This stems from the fact that the only distinguishing feature of the R* based algorithms, is the number of refresh messages sent. Figure 34 on page 88 shows this in detail.

When comparing the sort merge and the R* methods, the situation is much the same as in the static base table case.
Figure 45. Dynamic Base Table ($p = 0.2, q = 0.1, r = 0.7, \text{Case 2}$): R* Method. The curves give the cost trade off in I/O units as % of base table size. The section of a curve above the reference line means that the R* method wins, whereas negative values means that the basic sequential logging wins. Solid lines are Enhancement Two, dashed lines Enhancement One, and dotted lines Basic R* algorithm. Curves are not shown for $s = 1.0$, but follow the same pattern as described in Figure 43.
5.7.3 Dynamic Base Table

A dynamic base table is exposed to all three modification types. However, we assume that the deletes never outnumber the inserts. As in the incremental base table case, we only show the cost comparison figures for the sort merge and the R* methods.

Figure 44 illustrates the performance of the sort merge versus the basic refresh of sequential logging. Again, the figure depicts much the same situation as described for the other two modification cases.

Figure 45 shows the performance of the R* based algorithm. Like in the incremental case, Enhancement Two performs best — see also Figure 35 on page 89. The situation is very similar to the one described in the incremental case.

5.8 Evaluation of Results and Concluding Remarks

Several factors may influence the results given in the previous section. The most important ones, provided we keep the cost function unchanged, are the following:

1. The relationship between disk I/O and message unit cost.
2. The disk page size, i.e. the maximum number of entries per disk page.
3. The size of the base table prior to modification.

We will also study the consequence of changing the cost function, but first we comment on each of the issues listed above.

The relationship between disk I/O and message unit cost is so far assumed to be 1:1. If we increase the communication cost by assuming a slower communication network, say twice the disk I/O unit cost, i.e. \( t = 2t_d \), this will influence the results as follows:

- The sort merge refresh strategy becomes even more profitable, but the basic refresh strategy of sequential logging will still perform best for small modification sets.
- The condensed log performs relatively better, but is still very costly compared with the sequential log.
- The enhanced log does improve, but is still not profitable even for moderate modification sets.
- The basic R* algorithm provides no saving versus basic sequential logging for small modification sets. On the contrary, the basic algorithm performs somewhat worse. In the enhanced R* algorithms, a slight improvement is obtained even for small modification sets.

If we decrease the communication cost by assuming a faster communication network, say \( t = 0.5t_d \), then we will find that the sequential log is even more appropriate for small modification sets, but that the sort merge and the R* methods still perform best for sets above the previously given percent ratio. The cause for this is that the refresh update cost is more significant when relating the basic refresh method to the sort merge or the
R* methods. This explains why the cost difference does not change dramatically when varying the communication cost.

Increasing the maximum number of entries per data page, \( b \), will in general reduce the number of disk I/O and thus, the message cost becomes more dominant. The basic sequential logging will therefore lose overall. For \( b = 20 \), the sort merge alternative is profitable for modification sets down to 3 to 6 percent of the base table size, whereas the R* methods are still in the 10 percent range.

Finally, an increase of the base table size will influence the balance between sort merge and the basic refresh strategy for less restrictive snapshots. The cost of sorting the sequential log, when most of the log is selected, will lengthen the use of basic sequential logging.

Another issue that may affect the result is the following: 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. This will increase the “collision” effect, and bring the number of selected entries down for the condensed log and the R* method. In other words, this will favour the sort merge refresh strategy and the R* methods.

We will now study the consequences of changing the cost function. Neglecting the processing cost and only considering the communication cost, will obviously disfavour the basic sequential logging. All of the methods suffering from a high processing overhead will now become interesting. The relationship between the different methods can be read directly from the curves illustrating the number of entries qualified for shipment, see Figure 29, Figure 30, and Figure 31 describing the logging approach, and Figure 33, Figure 34, and Figure 35 describing the R* approach. The enhanced log and the R* Enhancement Two performs about equally — except for dynamic base tables where the enhanced log wins — and sometimes far better than the others.

Another change of the cost function is to neglect some of the processing cost. This may be relevant if some of the processing can be taken off peek periods. For instance, the sort merge step of the sort merge refresh strategy can be performed in this fashion. A key ordered sequential log with condensed property can be created at of peek periods, and used together with the modifications entered to the sequential log since then, at the next refresh of a snapshot. This will obviously favour this strategy to any of the others.

A simulation model has been devised for all of the logging and R* methods. The simulation model makes extensive use of the random function to control the distribution of modifications, the selection of tuples for modification and the location for inserting a new tuple in the base table. The model has been designed to simulate various snapshot selectivities — also controlled by a random number sequence.

The simulation shows results very close to the analytic model. However, some variations of the figures do occur, mainly due to small transitions in the select ratio as modifications are submitted. The results therefore show a stronger correlation for full copy snapshots \( s = 1.0 \) as this transition is avoided.
Figure 46. Differential vs. Full Refresh (Static Base Table): The curves give the cost trade off in I/O units as % of base table size. The section of a curve above the reference line means that differential refresh is profitable, whereas negative values means that a full refresh wins.

In the analysis we have so far focused on differential refresh strategies in order to find the applicability of each refresh strategy. We will now comment on the applicability of each method when compared with a full refresh strategy.
In a full refresh of a snapshot, the content of the snapshot table is reproduced by selecting all relevant tuples from the base table. In this respect, the cost of refreshing a snapshot table using a full refresh is similar to the R* approach. Unlike R*, a full refresh will cause shipment of a full snapshot table.

In the following scenario, we will focus on finding the applicability of a differential versus full refresh using the sequential approach. The cost using the sort merge refresh strategy has been applied in this comparison as it provides for larger savings than basic refresh for large modification sets. We have further excluded the cost of logging, as we focus on the choice between using a differential to a full refresh strategy at refresh time.

Also, in this scenario, we compare the R* approach to a full refresh strategy. Again, we apply the differential strategy providing the largest savings — the Enhancement Two refresh algorithm. Figure 46 shows the trade off in the choice between differential and full refresh. As can be seen from the figure, the sequential approach is profitable for a larger range than the R* approach.

In the R* approach, the differential refresh strategy will often perform worse than a full refresh strategy, if the snapshot is very restrictive. For less restrictive snapshots, the R* method will in general perform better, even if the number of modifications approach the number of tuples in the base table. The cause for this is the “collision” effect of modifications.

In the sequential approach, a differential refresh pays off for a much wider range. A reduction in the number of refresh messages is obtained at the expense of increased refresh create processing cost. For large modification sets, the refresh processing cost increases using sort merge, whereas the corresponding cost for full refresh is constant. In other words, what is gained in reduced communication cost, is lost in processing cost when compared with a full refresh strategy. For large modification sets, a full refresh strategy may therefore pay off.

The savings using differential versus a full refresh strategy is demonstrated using the sort merge refresh method as an example. Given a full-copy snapshot, i.e. $s = 1.0$, a modification set equal to 10 percent of the base table size, and a static base table containing 100,000 tuples. The I/O cost of refreshing a snapshot using the sort merge strategy, assuming the parameter values given in the previous section, amounts to 35,590. The corresponding figure using full refresh is 136,144. In other words, applying a differential refresh strategy will for this case give us a 74 percent reduction when related to full refresh.

In view of this, and the results given above, the following conclusion can be made. The results indicate that:

1. Sequential logging and its basic refresh strategy performs best for small modification sets.

2. The sort merge refresh strategy does improve the performance of snapshot maintenance when the modification set size exceed a certain ratio of the base table size. This can be determined exactly, given the modification set size ($M$), the composition ($p,q,r$), the base table size ($N$), and the selectivity of the snapshot ($s$).
3. The R* method, and especially Enhancement Two, is a good alternative to sequential logging when the modification set size exceed approximately 10 percent of the base table size.

The result further shows that for the sequential logging approach, we are able to dynamically decide whether to use the basic or the sort merge refresh strategy, or even the enhancement to logging — depending on the modification set size and composition. The two refresh strategies proposed for the sequential logging approach do in other words complement each other.
Chapter 6. Extending Refresh for Snapshot Replication

6.1 Introduction

In “Chapter 4. Differential Refresh Methods”, we described a series of differential refresh algorithms within the scope of one snapshot per base table. In the following sections we will discuss how these methods can be extended to support snapshot replication as defined in “Chapter 2. Snapshot Definition and Semantics”. Snapshot replication can be obtained in two different ways: i) by replicating the same snapshot to several sites, or ii) by defining several independent snapshots on the same base table.

In the case of single snapshots — i.e. at most one snapshot per base table — the cost associated with snapshot refresh must be taken for that snapshot alone, cf. “Chapter 3. Implementing Snapshots”. In a case where multiple snapshots are defined on the same base table, some of the processing cost as shown in Figure 5 on page 30, can be amortized over the snapshots defined:

First, the cost involved in the identification process can be “shared” by the snapshots defined on that base table. Secondly, the cost introduced by the refresh create process can be “shared” amongst the replicas of replicated snapshot whereas this is not possible for independent snapshots having individual refresh cycles.

As for the last two cost contributing factors identified in Figure 5, a “sharing” of cost can not be obtained. Separate refresh messages are sent to each snapshot even in the case of replicated snapshot. This cost, and the local refresh update cost, must therefore be taken by each independent snapshot or snapshot replica. We will return to this issue when analyzing the performance impact of the various methods.

In “Chapter 2. Snapshot Definition and Semantics”, refresh of replicated snapshots was discussed at a conceptual level. One of the issues emphasized there was the refresh or replicated snapshots in the event of site failure, i.e. what actions are taken when at least one of the replicas based on the same snapshot frame is unavailable for refresh.

As long as the distributed system operates “normally” — in the sense that all replica sites are available at the time the refresh operation is performed — refresh of all replicas will be identical to the case of refreshing one independent snapshot, except that refresh messages will be sent to all replicas. In the case that some of the replica sites are unavailable at refresh time, the outcome of the refresh operation depends on the level of
replica consistency defined for the frame. The REPLICA MODE option of the DEFINE SNAPSHOT statement — see Figure 3 on page 22 — is used to denote this.

By default, the strongest notion of replica consistency will be enforced, which is mutual consistency between the snapshot replicas. This means that all replicas must be refreshed collectively or not at all. The approach may in practice decrease the “up-to-date” information of snapshots, as one snapshot replica may prevent the others from being refreshed.

A looser notion of consistency may be specified for a frame using the REPLICA MODE option. One may choose using a conditional or an unconditional replication mode.

In the conditional mode, each replica should reflect the base table state as at refresh invocation time, i.e. the time when the operation was either requested (demand driven) or initiated by the database system (clock or event driven).

In the unconditional mode, each replica could be allowed to reflect the state of the base table as at time of refresh, i.e. when the refresh operation is carried out for that replica which will be sometime after the refresh invocation because of the site failure.

The conditional mode is in other words restricted by the state of the base tables as of a particular time — hence the name. In the unconditional mode, the current transaction consistent state of the base tables is used. The conditional mode is therefore in line with the original intention of snapshots whereas the unconditional mode does, when enforced, not support snapshots as of a given time.

In the following, we first describe how the refresh methods defined in “Chapter 4. Differential Refresh Methods” can be extended to support snapshot replication by studying each of the logging approaches and the R* approach separately. The discussion will address the basic as well as the enhanced refresh algorithms. A major part of the discussion is dedicated to the replica consistency issue. As we shall see, each approach can be extended for a replicated scheme. This extension can be done more easily to some of them than others.

Following this, we compare the extended version of the refresh approaches. An analysis similar to the one in “Chapter 5. Refresh Method Analysis” is performed. The cost function is extended to handle the added aspect of snapshot replication. The analytic model for the logging approach — defined in Figure 28 on page 80, and the R* approach — defined in Figure 32 on page 87,— are applied in the cost comparison.

We conclude with some remarks on the implementation aspects of the extended methods, and relate the differential strategy to a full refresh strategy.
6.2 Extended Sequential Logging

In "Chapter 4. Differential Refresh Methods", two alternative refresh algorithms are defined for the sequential logging approach — a basic refresh, and a sort merge refresh strategy. Both are based on a sequential log where modifications are recorded in the order submitted to the base table. We showed that a representation of the log with both before and after images provides for higher accuracy than a log with before images only — and without an increase in the logging cost. We also showed that the sort merge strategy provides higher accuracy than the basic refresh strategy. This property will thus, under certain circumstances, reduce the overall cost of snapshot maintenance when compared with a basic refresh strategy, see "Chapter 5. Refresh Method Analysis".

Since both refresh strategies rely on the sequential property of the log, the discussion that follows will focus on how the basic organization of the log can be extended to meet the snapshot replication requirement posed in "Chapter 1. Distributed Databases and Snapshots". At the end of this section we will study the implication of the extension with regards to the refresh strategies — especially the sort merge refresh strategy.

First we study the implication of the requirement for several independent snapshots defined on the same base table. The modifications submitted to the base table are added to the sequential log as before, but will now be used in refreshing each of the snapshots individually. We exclude from our discussion a logging scheme that involves one log per snapshot defined on a base table as the approach will increase the cost significantly for the identify modification process (see Figure 5 on page 30). Instead, we will see how one log can be used to support multiple snapshots.

As the sequential log is used as basis for refresh of each independent snapshot, the sequential log (cf. Figure 11 on page 46) can no longer be discarded after a refresh. The log entries will have to be kept until all snapshots defined on the base table have been refreshed correctly. In order to avoid full scan of the log table for one particular snapshot refresh, a mechanism can be used to identify the last refresh time of each snapshot.

Refresh of a snapshot — given that several independent snapshots are defined on the same base table — can basically be carried out as before except for one major difference: In the single case, a new sequential log is started after completing the refresh operation, and the old log is thrown away. In the extended sequential log, we start a new section of the sequential log, and leave the existing sections intact.

By keeping a reference — e.g. a page address — to the beginning of the section just started, the next refresh of the same snapshot can start its sequential search at this point of the log and continue its search onwards. We call this position on the log a refresh mark.

An example illustrating the sectioning of the sequential log is given in Figure 47. Three independent snapshots — $S_1$, $S_2$ and $S_3$ — are defined on the same base table. Each snapshot is associated with a refresh mark identifying the start of its section on the sequential log. As can be seen from the figure, the entire log, as of present, is relevant to the snapshot named $S_1$, whereas only the entries from page address 9 onwards must be considered in a refresh of snapshot $S_2$. In a way, each snapshot has its own log kept within the extended sequential log — its log beginning at the refresh mark associated
Figure 47. Sequential Log Supporting Multiple Snapshots: A sequential log stored over consecutive pages is shown. EndPage refers to the page holding the last recorded log entry. The region of the sequential log that needs to be searched at next refresh, is illustrated for the snapshots $S_1$, $S_2$ and $S_3$. with the snapshot. For simplicity, we have assumed that new sections of the log start at a new page (as shown in the figure).

An extended sequential logging scheme, applied on a base table, integrates the change history for all snapshots defined on that base table. Clearly, we do not want to keep the change history longer than necessary. Therefore, entries that are seen by all snapshots defined on the base table can be discarded, i.e. section(s) of the log below the lowest refresh mark of the snapshots can be thrown away.\(^8\)

Examples demonstrating the “delete” of old sections are now given. Consider a refresh of snapshot $S_1$ at time 9:00, and a log as given in Figure 47. After completing the refresh, a new section of the log will start at the address following EndPage (cf. Figure 47), and this address becomes the new refresh mark for snapshot $S_1$. The lowest refresh mark on the log will now be page address 6 (associated with snapshot $S_3$). Entries at page addresses below page 6 can therefore be removed from the extended log.

\(^8\) Instead of deleting old sections of the log, the pages may be released for future logging of entries. This can be controlled by a free chain, and the log may thus be used in a wraparound fashion. This technique is e.g. applied for the recovery log in System R [GRAY 81].

116 Database Snapshots
If one of the other snapshots — $S_2$ or $S_3$ — are refreshed at 9:00 instead of $S_1$, no section can be discarded from the log as can be seen by studying the example in Figure 47.

We will now discuss how the extension to the sequential log confirms with the requirements for independent and replicated snapshots. By associating a refresh mark with each snapshot, reflecting the log position at its previous refresh time, the sequential logging method lends itself to support independent snapshots. Refresh of an independent snapshot is based on extracting log entries from the given refresh mark and onwards.

The extension can also support replicated snapshots, and we discuss the three levels of replica consistency individually.

First, if mutual consistency is to be maintained for the snapshot replicas — of the same frame — at all times, the situation is similar to the one of independent snapshots. That is, the collective refresh is undertaken for the snapshot frame, and it is sufficient to keep the refresh mark with the frame definition, cf. the role of the snapshot manager in “Chapter 2. Snapshot Definition and Semantics”.

In the case where a looser replica consistency is allowed, the refresh of replicas may either be conditional or unconditional.

In the unconditional mode, where each replica is allowed to reflect the state of the base table at the time of the refresh, each replica may simply be treated as an independent snapshot. Obviously, a mark must be kept associated with each replica in order to support this strategy. This mark can be maintained in the sequential log instead of keeping it in the system catalog so as to avoid the catalog look up.

This approach can be illustrated using the example given in Figure 47. Assume that the three snapshots represent identical, replicated snapshots and that the snapshot replica named $S_1$ has been unavailable since 2:00, and replica $S_3$ since 3:00. Snapshot replica $S_2$, on the other hand, was available both at refresh time 3:00 and 4:00. Given that both snapshot $S_1$ and $S_2$ become available sometime after 4:00, then both snapshots will consider the log entries from their own refresh mark up to EndPage. In other words, the two replicas will include the later changes made to the base table — changes that have not been reflected in snapshot $S_2$. After completion, log entries below page address 9 can be discarded. As a result of this refresh approach, snapshot replicas $S_1$ and $S_3$ reflect a more recent database state than $S_2$.

A similar solution may be applied to the conditional mode. A refresh mark is kept for each replica to support replicas reflecting the base table state at refresh invocation time. However, 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 can best be illustrated using the same example as used for the unconditional replica mode.

Again, we let the three snapshots represent identical, replicated snapshots and that the snapshot replica named $S_1$ has been unavailable since 2:00, and replica $S_3$ since 3:00. Snapshot replica $S_2$, on the other hand, was available both at refresh time 3:00 and 4:00. Given that both snapshot $S_1$ and $S_3$ become available sometime after 4:00, then both snapshots must consider the log entries from their own refresh mark up to, but not including, $S_2$'s refresh mark. In other words, pages 1 to 8 must be read by snapshot $S_1$, and pages 6 to 8 must be read by snapshot $S_3$. After completion, log entries below page
address 9 can be discarded. All three snapshots will now reflect the base table state as at refresh time 4:00, and the sequential log will only hold the most recent modifications submitted to the base table.

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. Refresh of a previously unavailable replica may therefore take some time. A solution to this problem is to keep the refresh messages of the siblings in stable storage. This will reduce the log size and decrease the refresh cost when the site becomes available later on. Alternatively, the system may decide on a full refresh strategy (see “Chapter 3. Implementing Snapshots”) either based on the base table itself, or by sending a copy of an up-to-date sibling replica from another site. We will return to this when we discuss refresh of replicas within the frame of the R* approach.

Similar considerations must be taken for independent snapshots defined on the same base table. If one snapshot is refreshed very seldom, the system may consider a full refresh strategy as more appropriate for this snapshot. Thus, the log size may be reduced for the remaining snapshots defined on that base table. A full refresh strategy for the one snapshot may also appear to be less costly for this snapshot than a differential refresh, cf. the discussion on this subject in “Chapter 5. Refresh Method Analysis”.

We will now comment on the implication of the log extension with regards to the two refresh strategies. The extensions to the sequential logging method as proposed above, can be embedded quite easily in the basic refresh algorithm defined previously. The refresh create and refresh update processes for the extended sequential log are identical to the ones defined before, except for the restricted search region defined by the snapshot refresh marks, cf. Figure 12 on page 47 and Figure 13 on page 48. The sequential log is opened for sequential read within the relevant page address boundaries.

Depending on the solution applied to collective refresh of snapshots, the unconditional replication mode may affect the refresh update process somewhat. In the event of site failure, refresh of some replicas may be deferred. In the example given above, snapshot S₁ and S₃ are refreshed at a later stage with the result that these replicas now reflect a more recent database state than snapshot S₂.

A collective refresh of the replicas — employed as a common refresh operation — will thus cause extraneous refresh messages to some snapshot replicas — S₁ and S₃ in our example. This will affect the refresh update process in the sense that the algorithm depicted in Figure 13 on page 48 must be prepared to handle duplicate inserts, and delete of tuples no longer present in the snapshot table. As discussed in “Chapter 4. Differential Refresh Methods”, this does not cause incorrect refresh, only an extra overhead. Alternative solutions may be applied that avoid this, e.g. refreshing each replica individually.

When comparing the extended sequential log with the single case, the extended log will obviously contain more log entries as entries must be kept until all snapshots have seen them. On the other hand, very little overhead is added per replica or snapshot. Indeed, the cost of logging the modifications is “shared” by the snapshot defined on the base table, and the length of the log that needs to be consulted at next snapshot refresh is no
longer than the equivalent log in the single snapshot case. The cost of shipping the refresh messages will therefore be the same as in the single case.

As for the sort merge refresh strategy, the extended sequential log does not affect the refresh algorithm as such. The selection of log entries relevant to a given snapshot is carried out in the same way as in the basic refresh strategy. The added aspect of having several independent snapshots, opens the possibility for another solution. So far, we have assumed that the refresh create process uses the full extended log as basis for the sort merge step. That is, the relevant section(s) of the log is (are) scanned at each refresh as in the basic strategy.

Alternatively, we may keep the sorted version of the log instead of wasting the result obtained by the sort merge step. However, this requires a full sort merge of the relevant section(s) of the log as opposed to a restricted sort merge. That is, log entries not satisfying the snapshot being refeshes, can not be discarded prior to sorting. Also, to avoid changing previously defined refresh marks, the sort is performed for each section of the extended log separately.

In practice, this means sorting the new section first at the next refresh of an independent snapshot. Several log entries to the same base table tuple are then merged into one entry as defined by the merge rules in Figure 15 on page 51. The key ordered version of the section replaces the previous unsorted section. Then, the relevant sections of the log — now being sorted — are merged and refresh messages can be produced. The modification to the sort merge algorithm, to support this alternative solution, is feasible. However, in the cost analysis described later on, the original strategy is assumed.

6.3 Extended Condensed Logging

Two alternative realizations of the condensed log are discussed in “Chapter 4. Differential Refresh Methods” — a condensed log based on B-tree, and a condensed log based on linear hashing. Both support differential refresh, but the refresh operation for the two alternatives differs in one respect: The B-tree solution enables a sequential update of the snapshot table, whereas the linear hashing submits the snapshot updates in a "random" fashion. The logging overhead in the two alternatives is also different. Linear hashing introduces a smaller logging overhead. The linear hashing alternative does therefore perform significantly better — especially for restrictive snapshots — than the B-tree based approach, see discussion in “Chapter 5. Refresh Method Analysis”.

Another observation made in the cost analysis, is that the condensed log performs significantly worse than any other method. Despite this, we have included the approach here not only to make the discussion of each approach complete, but the replication aspect opens for a cost effective application of condensed logging.

The main disadvantage of the condensed log, discussed so far, is the considerable logging overhead. In the snapshot replication case, this cost can be "shared" amongst several snapshots. Thus, the weight of this cost factor is reduced compared with the other factors in the cost function. The quality of the other properties of condensed logging may thus have its proper effect when it comes to snapshot replication.
<table>
<thead>
<tr>
<th>Time Stamp</th>
<th>Modif. Type</th>
<th>Resort</th>
<th>Value Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:00</td>
<td>DEL</td>
<td>Beirut</td>
<td>Lebanon</td>
<td>6</td>
</tr>
<tr>
<td>2:00</td>
<td>UPD</td>
<td>Cannes</td>
<td>France</td>
<td>7</td>
</tr>
<tr>
<td>3:00</td>
<td>-</td>
<td>Cannes</td>
<td>France</td>
<td>8</td>
</tr>
<tr>
<td>4:00</td>
<td>UPD</td>
<td>Cannes</td>
<td>France</td>
<td>8</td>
</tr>
<tr>
<td>3:00</td>
<td>-</td>
<td>Cannes</td>
<td>France</td>
<td>9</td>
</tr>
<tr>
<td>4:00</td>
<td>UPD</td>
<td>Cannes</td>
<td>France</td>
<td>9</td>
</tr>
<tr>
<td>2:00</td>
<td>INS</td>
<td>Crete</td>
<td>Greece</td>
<td>6</td>
</tr>
<tr>
<td>3:00</td>
<td>INS</td>
<td>Crete</td>
<td>Greece</td>
<td>3</td>
</tr>
<tr>
<td>4:00</td>
<td>UPD</td>
<td>Crete</td>
<td>Greece</td>
<td>3</td>
</tr>
<tr>
<td>2:00</td>
<td>INS</td>
<td>Miami</td>
<td>United States</td>
<td>7</td>
</tr>
</tbody>
</table>

**Figure 48. Extended Condensed Log:** A timestamp is added to separate modifications belonging to different sections of the log. The example corresponds to a refresh situation as depicted for the three snapshots defined in Figure 47.

Despite the difference in realization of the two condensed logging alternatives, the log can be extended to support the replication requirement in both of them. To limit the discussion, we have assumed the B-tree version of the log in the following. First, we will see how the log can be extended to accommodate independent snapshots. The extension to the log is described and a new algorithm supporting refresh of independent snapshots is proposed. Then, we elaborate on the replica consistency issue, before we end by commenting on the approach using the linear hashing alternative.

We begin by studying the implication of supporting several independent snapshots. Like in the sequential log, the modifications recorded in the condensed log will be used as basis for refresh of possibly several independent snapshots. The change history must therefore be kept until the modifications have been reflected in all snapshots defined on the base table.

Unlike the sequential log, a condensed log does not preserve the overall order in which modifications are submitted to the base table. Instead, modifications are merged into one log entry recording the image of the changed base table tuple as at the time of the last refresh (before image), and the image as after the last modification (after image) — see Figure 16 on page 51 for details.

A refresh mark was used in the sequential log to denote the changes made after a certain refresh time. In the condensed log, a timestamp (or an equally continuous increasing value) can be used to fulfill the same identification. We define a LogTime as the current time for new modifications submitted to the base table. The LogTime is the time of the very last refresh performed on the condensed log, and is used to indicate the beginning of a new section of the log.

Modifications are added to the condensed log in the same manner as in the single snapshot case, except that a new modification is merged with the latest stored, unrefreshed entry on the log if such an entry exists. In order to perform this merge correctly, each log entry is given a timestamp reflecting the time when the new section of the log was started. The latest modifications are given the timestamp LogTime. Thus, log en-
tries holding the same timestamp corresponds to modifications recorded between two refresh marks on the sequential log.

After refresh of one snapshot, a new section of the log is started — indicated by incrementing the LogTime to the current refresh time. As for the snapshot just refreshed, entries written to the log with this new timestamp (or timestamps of any succeeding sections) are relevant at its next refresh. A timestamp associated with the snapshot will thus serve the same purpose as the refresh mark in the sequential log.

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 48 shows a log table for this scheme. A timestamp column has been added to the log table. Note that the tuples in the extended condensed log are stored in the order of the base table's primary key, the timestamp and the modification type. The modification type may be coded as commented on in "Chapter 4. Differential Refresh Methods".

The timestamps in the example corresponds to the situation depicted in Figure 47, in which three independent snapshots are defined on the same base table and refreshed at individual refresh times. In addition, we have shown the actual modifications taken place over the given period.

Snapshot $S_1$ is the oldest snapshot in the example given, refreshed at time 2:00 (cf. Figure 47). Therefore, entries in the log refer to changes made sometime after 2:00.

Snapshot $S_2$ is the next oldest snapshot in the example, refreshed at time 3:00. Modifications submitted to the base table sometime after 2:00, but before 3:00, are identified by the timestamp 2:00. Referring to the example, Beirut has been deleted, Cannes updated, and Miami inserted.

Finally, snapshot $S_3$ is the latest refreshed snapshot, refreshed at time 4:00. In between refresh of $S_2$ and $S_3$, Cannes has been updated again, and Crete inserted. These changes have been timestamped 3:00. Following this, a new section of the log is started recording the latest modifications to the base table. These are timestamped 4:00 as shown in the figure. Cannes has been updated once more, and Crete is changed as well.

From the extended condensed log, we can see that Cannes has been recorded three times — timestamped 2:00, 3:00 and 4:00. The reason for this is as follows: To support refresh of $S_1$, relevant modifications are those timestamped 2:00 and later, which is indeed all of the entries. In order to satisfy the refresh on $S_1$, only one recording showing the change of price level from 7 to 6 is necessary. This solution will not satisfy snapshot $S_3$, who would like to see the change in price level from 8 to 6, and finally $S_2$, who would like to see the change from 9 to 6. The compromise is to keep one recording for each time interval not yet seen by all snapshots. This corresponds to the three timestamps in the example given.

As an alternative solution, one could record the modification as it appears to each snapshot individually. This is equivalent to having one condensed log per snapshot table which will increase the overhead instead of utilizing the information already kept in the log.
Associated with each snapshot is a timestamp of its last refresh, see Figure 47. This may either be kept in the system catalog or in the header of the log table. 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 after its last refresh. In other words, log entries having timestamp equal to or newer than the one associated with the snapshot, are relevant for refresh. The refresh create process will merge these entries into one modification which can then be checked to see if it qualifies the snapshot restriction or not.

Again, referring to the condensed log depicted in Figure 48, the relevant entries in the event of refreshing snapshot \( S_1 \), are all of the entries currently kept in the log. A refresh of \( S_2 \), on the other hand, will only involve log entries with the LogTime timestamp (4:00).

As for the refresh of \( S_1 \), the true modification for one base table tuple, is found by taking the before image of its oldest logged entry and the after image of its newest — provided the modifications are of type update. For example, Cannes true change as from \( S_1 \) point of view, is the change in price level from 7 to 6. The merge process can be generalized to include all modification types, simply by applying the merge rules defined in Figure 15 on page 51 repeatedly. The details are given in the pseudo code representation of the algorithm, see Figure 49. The algorithm makes use of additional procedures defined in Figure 50 and Figure 51.

The selection of relevant sections of the condensed log is expressed by the cursor statement in the refresh create algorithm given in Figure 49. Only entries with relevant timestamps are selected. The ORDER SEQUENTIAL clause appended to the cursor emphasizes the point that the log is scanned in the log’s key order. As the primary key of the base table is the most significant part of the composed log key, log entries belonging to the same base table tuple are read in sequence. Furthermore, the older ones will come before the newer ones. For modifications referring to the same base table tuple, the merge rules are applied. When modifications a new tuple is encountered, the merged log entry is checked against the snapshot restriction. This is taken care of by the CheckForShipment procedure defined in Figure 51.

So far, we have not addressed the issue of deleting irrelevant sections of the log after refresh. Once a tuple has been examined for refresh, the condensed log can be compressed by removing entries with irrelevant timestamps. This corresponds to removing a section from the log that has been seen by all snapshots defined on the base table. The section removed may either be the oldest section of the condensed log, or a newer section. In the latter case, its log entries must be merged with the older neighbour section.

Naturally, modifications having a timestamp older than the oldest timestamp of any snapshot defined on the table, can be discarded. This is taken care of by the DELETE statement issued as the last operation in the refresh create algorithm, see Figure 49. MinTimeStamp is the oldest timestamp, after refresh, of the snapshots defined on the base table. An example of this is the refresh of snapshot \( S_1 \) at time 9:00, see Figure 52. After completion, \( S_3 \) will be the oldest snapshot and all entries that have been reflected by \( S_3 \) can be discarded from the log. In other words, entries timestamped before 3:00 can be removed from the log.
RefreshCreate( CondLog, SnapRestrict, LastRefreshTime, DiscardSection, NeighbourTime, MinTimeStamp, Channel );

begin
  LastKey := NULL; (* Primary key of last base table tuple *)
  ModEntry := nil; (* Current merged modification entry *)
  DECLARE C1 CURSOR FOR
    SELECT Entry
    FROM CondLog
    WHERE TimeStamp \geq LastRefreshTime
    ORDER SEQUENTIAL;

  OPEN C1;
  Open( Channel, 'Send' );
  FETCH C1;
  repeat
    begin
      if Entry↑.PKey ≠ LastKey then
        begin
          (* Check shipment of previous merged modification *)
          CheckForShipment( ModEntry );

          (* Start merge for the new base table tuple *)
          LastKey := Entry↑.PKey;
          ModEntry := Entry;
          if DiscardSection then CompressLog( ModEntry );
        end
      else
        begin
          (* True modification found by merging log entries upwards *)
          ModEntry := Merge( Entry, ModEntry );
        end;

      FETCH C1;
    end
  until EndOfScan( C1 );

  (* Shipment of last merged entry, if any *)
  CheckForShipment( ModEntry );
  Close( Channel );
  CLOSE C1;

  (* Remove log entries seen by all snapshots *)
  DELETE FROM CondLog
  WHERE TimeStamp < MinTimeStamp;
end;

Figure 49. Refresh Create for Extended Condensed Log: The log is compressed using procedure CompressLog in Figure 50, if so denoted by the DiscardSection flag. Merged tuples are checked for shipment as defined in Figure 51. The Merge procedure operates according to the merge rules defined in Figure 15 on page 51. Note that entries having timestamp less than the oldest snapshot — MinTimeStamp — are removed from the log. ORDER SEQUENTIAL is not SQL standard.
CompressLog( RemEntry );
begin
  DECLARE C2 CURSOR FOR
  SELECT Entry
  FROM CondLog
  WHERE PKey = LastKey
  AND TimeStamp = NeighbourTime;
  OPEN C2;
  FETCH C2;
  if EndOfScan( C2 ) then
    begin
      (* No neighbour log entry to merge with,
         instead the timestamp is changed to simulate removal *)
      UPDATE CondLog
      SET TimeStamp = NeighbourTime
      WHERE CURRENT OF C1;
    end
  else
    begin
      (* Remove entry of timestamp LastRefreshTime *)
      DELETE FROM CondLog
      WHERE CURRENT OF C1;
      (* Merge the removed entry with its older neighbour, and
         update the CondLog to reflect the result of the merge *)
      ResultEntry := Merge( RemEntry, Entry );
      if ResultEntry = nil
        then
          DELETE FROM CondLog
          WHERE CURRENT OF C2
        else
          UPDATE CondLog
          SET Entry = ResultEntry
          WHERE CURRENT OF C2;
    end;
  CLOSE C2;
end;

Figure 50. Compressing the Condensed Log: The section associated with the
LastRefreshTime of the refreshed snapshot is discarded from the log. Entries of
this timestamp are merged with the older, corresponding log entries of the
neighbour section. Note that the outcome of this merge may be a removal of two
adjacent log entries, cf. the merge rules defined in Figure 15 on page 51. The
procedure is applied as a local procedure within the refresh create process of the
extended condensed log, see Figure 49.

In addition to this, we can remove modifications that have been kept in the condensed
log only to ensure a correct refresh of the snapshot just refreshed. Returning to our
example, Cannes has been logged several times. A refresh of snapshot S₂ at time 9:00
will leave us with snapshots refreshed at 2:00, 3:00 and 9:00. None of the snapshots have
a last refresh time of 4:00, so the recording at time 4:00 is no longer needed and can be
CheckForShipment( Entry );
begin
  if Entry ≠ nil then
  with Entry† do
  begin
    (* Apply the same set of tests as for the basic condensed log refresh create process *)
    end;
    Entry := nil;
  end;
end;

Figure 51. Check for Shipment of Merged Modification: The merged entry is checked for shipment. The same tests as those given for the basic refresh create process of the condensed log applies and therefore not repeated here. For details, see Figure 12 on page 47. The procedure is applied as a local procedure within the refresh create process depict in Figure 49.

merged with its older neighbour as shown in Figure 52. The condensed log after refresh of $S_2$ will therefore only record changes made in the time intervals between 2:00 to 3:00 (timestamped 2:00), and 3:00 to 9:00 (timestamped 3:00). In other words, the old timestamp associated with $S_2$, is removed from the log.

The merging scheme, in which the section removed is merged with the older neighbour section, can only be performed if the following condition is fulfilled: No other snapshot, defined on the same base table, can have the same timestamp associated with it, i.e. the snapshot being refreshed "owns" the actual section. We name this merging scheme neighbour merge.

If in our example, a snapshot $S_4$ was defined and refreshed at time 4:00, section removal would not be possible when performing the refresh of $S_2$ at time 9:00. The entries timestamped 4:00 would have to be kept until reflected in a later refresh of $S_4$. Then, if no other snapshot claims this timestamp, the removal of the section can take place.

As it is, the condition for applying neighbour merge can be determined at refresh time — without having to scan the log. By consulting the last refresh time associated with the snapshots defined on the base table, this can be verified and used as input to the refresh create process. In Figure 49, the DiscardSection flag denotes this event.

When a section — other than the end section — is removed, its logged changes must be embedded in the previous neighbour section of the log. One way of seeing this is to say that the previous section is given a time interval now including the section removed, or alternatively, we may pretend that the snapshot refresh never actually took place at the particular time (the time now being removed).

Figure 50 is a pseudo code representation of this neighbour merge scheme. The operation is invoked in parallel with the merge of modifications recorded for one base table tuple. The entry of the removed section is either merged with the corresponding entry in the older neighbour section if such an entry exists, or its timestamp is simply updated to denote the change in section membership. The NeighbourTime applied in the proce-
dure denotes the timestamp of the next older snapshot defined on the base table, i.e. the timestamp of the neighbour section. This value can be found along with the testing of the merge condition, prior to the condensed log scan. The neighbour merge scheme is initiated when a new set of modifications are detected in the algorithm in Figure 49, and only if the DiscardSection flag is set.

In Figure 52, the merge of neighbour sections is demonstrated for two different cases; the refresh at time 9:00 for snapshots S_2 and S_3 respectively. Note that timestamps as of the last refresh time for the snapshot being refreshed, no longer are present in the condensed log. When refreshing S_1 for example, entries have either been merged with a neighbour entry (Cannes), or the timestamp is updated (Crete). Updating the timestamp so as to obtain the change in section membership does not provide any compression of the log at first. By changing the timestamp to the refresh time of the older snapshot, the reduction may be obtained at a later refresh of the snapshot that “owns” this section.

The extended condensed logging method supports independent snapshots as demonstrated by the examples above. A new refresh create algorithm — Figure 49 — has been defined to accomplish the extension. No change of the refresh update algorithm logic is required.

We will now discuss how the extension to the condensed log confirms with the requirement for replicated snapshots and the replica consistency issue for replicated snapshots.

As we have seen in the support of independent snapshots, the timestamp associated with each snapshot plays the same role as the refresh mark in the extended sequential log. Snapshot replicas confirming to mutual consistency can therefore be supported as in the sequential approach by keeping the refresh time with the snapshot frame. In other words, the frame definition corresponds to one independent snapshot.

The looser notion of replica consistency can also be supported using the same technique as suggested for the extended sequential log. To each snapshot replica we tie a timestamp of its last refresh. Each replica may then be viewed as an independent snapshot. In the unconditional mode, we may use the procedure defined in Figure 49 directly. Thus, all log entries recorded after the last refresh time of the replica, will be selected.

In the conditional mode, log entries from the replicas timestamp and up to the timestamp of the most up-to-date sibling replica should be selected. This test can be employed in the refresh create algorithm by adding the upper restriction on timestamps to the cursor statement in Figure 49. The support of this strategy can be explained using the same example as given for the extended sequential log.

As before, we assume that the three snapshots represent identical, replicated snapshots and that the snapshot replica named S_1 has been unavailable since 2:00, and replica S_3 since 3:00. Snapshot replica S_2, on the other hand, was available both at refresh time 3:00 and 4:00. Given that both snapshot S_1 and S_3 become available sometime after 4:00, then both snapshots must consider the log entries from their own last refresh time up to S_2's refresh time. In other words, entries in the log shown in Figure 48 having timestamp 2:00 and 3:00 must be read by S_1, whereas only entries with timestamp 3:00 need to be read by snapshot S_2. After completion, log entries with timestamp older than 4:00 are discarded. All three snapshots will now reflect the base table state as at refresh.
### EXTENDED CONDENSED LOG AFTER REFRESH OF $S_1$

<table>
<thead>
<tr>
<th>Time Stamp</th>
<th>Modif. Type</th>
<th>Resort</th>
<th>Value Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:00</td>
<td>UPD</td>
<td>Cannes</td>
<td>France</td>
<td>8</td>
</tr>
<tr>
<td>3:00</td>
<td>-</td>
<td>Cannes</td>
<td>France</td>
<td>9</td>
</tr>
<tr>
<td>4:00</td>
<td>UPD</td>
<td>Cannes</td>
<td>France</td>
<td>9</td>
</tr>
<tr>
<td>4:00</td>
<td>-</td>
<td>Cannes</td>
<td>France</td>
<td>6</td>
</tr>
<tr>
<td>3:00</td>
<td>INS</td>
<td>Crete</td>
<td>Greece</td>
<td>3</td>
</tr>
<tr>
<td>4:00</td>
<td>UPD</td>
<td>Crete</td>
<td>Greece</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Crete</td>
<td>Greece</td>
<td>4</td>
</tr>
</tbody>
</table>

### EXTENDED CONDENSED LOG AFTER REFRESH OF $S_2$

<table>
<thead>
<tr>
<th>Time Stamp</th>
<th>Modif. Type</th>
<th>Resort</th>
<th>Value Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:00</td>
<td>DEL</td>
<td>Beirut</td>
<td>Lebanon</td>
<td>6</td>
</tr>
<tr>
<td>2:00</td>
<td>UPD</td>
<td>Cannes</td>
<td>France</td>
<td>7</td>
</tr>
<tr>
<td>2:00</td>
<td>-</td>
<td>Cannes</td>
<td>France</td>
<td>8</td>
</tr>
<tr>
<td>3:00</td>
<td>UPD</td>
<td>Cannes</td>
<td>France</td>
<td>8</td>
</tr>
<tr>
<td>3:00</td>
<td>-</td>
<td>Cannes</td>
<td>France</td>
<td>6</td>
</tr>
<tr>
<td>3:00</td>
<td>INS</td>
<td>Crete</td>
<td>Greece</td>
<td>4</td>
</tr>
<tr>
<td>2:00</td>
<td>INS</td>
<td>Miami</td>
<td>United States</td>
<td>7</td>
</tr>
</tbody>
</table>

### EXTENDED CONDENSED LOG AFTER REFRESH OF $S_3$

<table>
<thead>
<tr>
<th>Time Stamp</th>
<th>Modif. Type</th>
<th>Resort</th>
<th>Value Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:00</td>
<td>DEL</td>
<td>Beirut</td>
<td>Lebanon</td>
<td>6</td>
</tr>
<tr>
<td>2:00</td>
<td>UPD</td>
<td>Cannes</td>
<td>France</td>
<td>7</td>
</tr>
<tr>
<td>2:00</td>
<td>-</td>
<td>Cannes</td>
<td>France</td>
<td>9</td>
</tr>
<tr>
<td>4:00</td>
<td>UPD</td>
<td>Cannes</td>
<td>France</td>
<td>9</td>
</tr>
<tr>
<td>4:00</td>
<td>-</td>
<td>Cannes</td>
<td>France</td>
<td>6</td>
</tr>
<tr>
<td>2:00</td>
<td>INS</td>
<td>Crete</td>
<td>Greece</td>
<td>3</td>
</tr>
<tr>
<td>4:00</td>
<td>UPD</td>
<td>Crete</td>
<td>Greece</td>
<td>3</td>
</tr>
<tr>
<td>4:00</td>
<td>-</td>
<td>Crete</td>
<td>Greece</td>
<td>4</td>
</tr>
<tr>
<td>2:00</td>
<td>INS</td>
<td>Miami</td>
<td>United States</td>
<td>7</td>
</tr>
</tbody>
</table>

Figure 52. Merging Entries of Extended Condensed Log: Entries having a timestamp no longer associated with any snapshot are merged with corresponding entries of the older neighbour section on the log, thus compressing the log. The example shows the condensed log after refresh at time 9:00 for three different cases.

time 4:00, and as it is, the log will only contain the latest modifications submitted to the base table (those having timestamp 4:00).
This example also demonstrates that it may pay off to keep the refresh messages in stable storage in the case that some snapshot replicas are unavailable for refresh. This will release several of the log entries and speed up the refresh when a snapshot replica becomes available later on. The disadvantage of this approach is that refresh messages must be kept around for some time — with the possibility that more than one refresh message set will have to be kept (two in the example above). A safer approach may therefore be to base the refresh of a previous unavailable replica, on the content of the extended condensed log.

When comparing the extended condensed log with its basic version in “Chapter 4. Differential Refresh Methods”, the following differences in performance are noticeable: First, the cost of logging of entries can be “shared” by several snapshots. Secondly, this reduction is obtained at the expense of a longer log length compared with a log supporting one snapshot only. This stems from the partitioning of the log into several smaller sections in which the “collision” effect of modifications is decreased. Scanning the extended condensed log at snapshot refresh, is therefore more costly. To compensate for the longer log length, a merge of modification entries is performed. As a result of this, the number of refresh messages can be kept down to the same size as in the single case. The refresh communicating cost and the refresh update cost is therefore the same.

The refresh create process may be speeded up by introducing an index on the timestamp column of the condensed log table. A similar performance improvement can be obtained if the database system can utilize the composed property of the condensed log table’s key [MIME 85]. Anyway, the performance of the refresh operation on a particular database snapshot depends largely on how many log sections that have to be considered for each tuple (cf. Figure 48). This again depends on the number of snapshots defined on the table, and on how widely the refreshes are scattered.

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 with an older neighbour section will keep this number down. Unfortunately, this will add to the cost of refresh processing.

We will now comment on the other alternative — a condensed log based on linear hashing. In the single snapshot case, the condensed log contains one log entry for every modified base table tuple. Thus, we may easily refer to the one modification entry using the linear hashing access method. In the multiple snapshot case, we need to log several log entries of the same tuple — each belonging to one section of the log. The uniqueness of modification entries may be maintained, but how do we obtain a clustering of the log entries so that entries of the same base table tuple can be read sequentially without imposing extra disk page reads, cf. the refresh create algorithm in Figure 49.

A simple solution to this is to allow for duplicate keys in which the log key now consists of the base table primary key only. When a new modification is entered, we must therefore search amongst the duplicates to verify the insertion, or if a modification to an existing entry applies. As for the refresh create process, log entries belonging to the same base table tuple, may now be found at a reasonable cost.
6.4 Impact on Enhancement to Logging

The enhancement to logging, proposed in "Chapter 4. Differential Refresh Methods", attempts to reduce the number of messages sent to the snapshot table by only sending inserted and updated tuples that are relevant to the snapshot. Deletes are included in the insert and update messages. This is obtained by detecting removed regions in the base table and if the removed region has relevance to the snapshot table.

The enhancement requires a sorted version of the log, which implies a B-tree based condensed log or a sequential log sorted by the sort merge refresh strategy. We will now discuss the extended logging method and its impact on the enhanced refresh algorithm. As a sorted, condensed sequential log provides the same property as a condensed log — even in the extended form — we can limit the discussion to extended condensed logging.

The base table is scanned, as well as the condensed log, in order to fulfill the objective of the enhancement. To detect removed regions, the condensed log is checked. In the single snapshot case, this check can be performed as a predicate check, cf. the exists predicate in Figure 25 on page 66. For each base table tuple satisfying the snapshot restriction, one verifies its relevance to the snapshot by consulting the log. If the tuple was indeed updated or inserted, an entry will be found on the log. If the tuple follows a removed region of relevance to the snapshot, this may be detected by checking the log entries below this tuple and the last one that qualified.

In the multiple snapshot case, a similar verification can take place using the extended condensed log, but not quite as easily. To verify that the qualified tuple is updated or not, one may still use a predicate to check for a log entry, with the actual tuple key, submitted sometime after the snapshots last refresh. The detection of removed regions must be performed as a log scan within the actual tuple key boundaries. In this scan, the oldest before images of each base table tuple — timestamped after the snapshots last refresh — will be searched to see if that before image qualifies the snapshot restriction.

Removing a section of the log that is no longer needed can be done in the same way as described for the basic algorithm. This applies to condensed logging only. Note however, that the condensed log need only record the before images of updates as mentioned in "Chapter 4. Differential Refresh Methods". After all, the after image, at any refresh time, can either be found in the base table (which is scanned anyway), or as the before image of a newer section on the log.

The enhancement can support multiple snapshots and all three levels of replica consistency. This is a property inherited from the log method. Removing after images has no consequence to the support of the replica issue.

6.5 Extended R* Method

The R* approach is described in "Chapter 4. Differential Refresh Methods". The basic refresh algorithm — defined by [LIND 86] — is discussed, and two enhancements are proposed to this algorithm. We will now discuss the applicability of the R* approach to multiple snapshots by addressing the basic and the enhanced versions of the refresh algorithms separately.
**BASE TABLE AFTER REFRESH AT 4:00**

<table>
<thead>
<tr>
<th>Addr</th>
<th>Prev Addr</th>
<th>Time Stamp</th>
<th>Resort</th>
<th>Value Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:</td>
<td>nil</td>
<td>3:00</td>
<td>Brighton</td>
<td>England</td>
<td>6</td>
</tr>
<tr>
<td>5:</td>
<td>3</td>
<td>NULL</td>
<td>Cannes</td>
<td>France</td>
<td>6</td>
</tr>
<tr>
<td>7:</td>
<td>5</td>
<td>0:00</td>
<td>Colombo</td>
<td>Sri Lanka</td>
<td>3</td>
</tr>
<tr>
<td>8:</td>
<td>7</td>
<td>NULL</td>
<td>Crete</td>
<td>Greece</td>
<td>4</td>
</tr>
<tr>
<td>9:</td>
<td>8</td>
<td>0:00</td>
<td>Florence</td>
<td>Italy</td>
<td>5</td>
</tr>
<tr>
<td>10:</td>
<td>9</td>
<td>3:00</td>
<td>Miami</td>
<td>United States</td>
<td>7</td>
</tr>
<tr>
<td>11:</td>
<td>10</td>
<td>0:00</td>
<td>Tenerife</td>
<td>Spain</td>
<td>4</td>
</tr>
</tbody>
</table>

**Figure 53. Base Table used in Multiple Snapshots:** The base table has been used in refreshes at time 2:00, 3:00 and 4:00, cf. Figure 48.

First we will study the implication of supporting several independent snapshots. Then we will address the replica consistency issue of replicated snapshots, and comment on the overall solution. Finally, we discuss how the enhancements, proposed to the R* refresh algorithm, can be maintained in the extended R* approach.

In order to support multiple snapshots in both the sequential and the condensed logging approach, some alterations had to be made to the log. In the sequential log, refresh marks were introduced to separate sections of the log. In the condensed log, a timestamp column was added to the log table to provide for a similar partitioning. As for the R* approach, no logging scheme is used. Instead, modifications are identified in the base table itself by using two additional columns in the table. One of these columns records timestamps to reflect the time when the last modification is done. This timestamp can now be used to support multiple snapshots.

The extensions made to the base table to support single snapshots in the R* approach, will also support multiple snapshots. Independent snapshots can be refreshed using the basic refresh algorithm defined in Figure 20 on page 58 — provided that each snapshot has been assigned a timestamp equal to its last refresh time. The algorithm does, however, suffer from the weakness that superfluous refresh messages will be sent to the snapshot table. This weakness is not increased by the fact that several independent snapshots are defined on the base table. Exactly the same set of refresh messages will be sent as in the single snapshot case.

As for replicated snapshots, the basic refresh algorithm can support the strongest level of replica consistency — **mutual consistency** — in the same way as in the extended condensed logging approach. A timestamp is simply associated with the snapshot frame.

The looser notion of replica consistency — enforced in the case of site failure during refresh — can only be supported partially if we continue to use the base table as the only source for snapshot refresh. By treating each snapshot replica as an independent snapshot with its own timestamp, the basic algorithm may support the *unconditional mode* directly. A refresh of a previously unavailable snapshot replica will in this way reflect the current state of the base table.
However, there is no easy way to extend the refresh algorithm to support the conditional mode of replica consistency. The base table always reflects the latest database state, and information of modifications made since last refresh. In other words, there is no history of tuple images belonging to different refresh times in the base table.

The problem associated with support of conditional mode consistency may be demonstrated using the same example as given for the sequential and condensed log. Figure 53 depicts the base table after refresh time 4:00, corresponding to the changes listed in Figure 48. Note that the timestamps have been shifted one refresh cycle when compared with Figure 48. This is because the \( R^* \) method inserts the timestamps as at the next refresh, instead of the previous refresh timestamp. From Figure 53, we can see that Cannes and Crete have been changed since the latest refresh performed on the base table.

As before, we assume that snapshots \( S_1, S_2 \) and \( S_3 \) are replicated, and that \( S_1 \) has been unavailable since its last refresh at 2:00, and \( S_3 \) since 3:00. At the last refresh of the replicated snapshot at time 4:00, \( S_2 \) was the only one available. Given that the two snapshots become available sometime after 4:00, and before the next collective refresh, we end up with a problem if \( S_1 \) and \( S_3 \) have to reflect the same state as \( S_2 \). The image before the latest update of Cannes and Crete can not be revealed. On the other hand, we may easily refresh the snapshots to reflect the current state of the base table which now contains the new values for Cannes and Crete.

To overcome the problem identified above, refresh messages to a replicated snapshot must be kept in stable storage before being sent. If a replica site is unavailable during refresh processing, the stored messages can then be sent if the sites becomes available before the next collective refresh of the snapshot. The messages can first be discarded when all snapshot replicas have been refreshed. As mentioned during the description of the extended condensed log, we may end up keeping several refresh message sets. In the example above, two sets are needed. The first being the set of messages created at refresh time 3:00. The second created at refresh time 4:00. When \( S_1 \) becomes available, both sets are sent, whereas only the latest set needs to be shipped to \( S_3 \).

This message store and forward technique may be advantageous regardless of the replica consistency requirement. In an implementation, the refresh of replicated snapshots may have to be split into two steps. The first step being a sequential scan of the base table, creating the set of refresh messages that needs to be shipped. This corresponds to the refresh create process as defined previously for the basic refresh algorithm. The second step will then handle the shipment of refresh messages to all of the recipients, i.e. to all replicas of the actual snapshot. If some, or even all, of the replicas are unavailable for refresh, the refresh message set must be kept until reflected by all replicas at a later stage.

A two step algorithm of this kind is most useful in a distributed environment that lacks a robust broadcasting mechanism. Then, a shipment to all replicas consists of just as many cooperating write transactions updating the snapshot tables at the snapshot sites. Obviously, one site failure within the refresh transaction is all it takes to abort the entire refresh operation. By splitting the operation into two steps — letting the second stage open one shipment at the time — a site failure will only affect the snapshot replica residing at the “failed” site.
Alternatively, the refresh operation may be performed as one transaction per replica by treating each replica as an independent snapshot. In this case, the base table is scanned once per replica. If there are many replicas defined on a large table, the refresh cost may become unacceptably high. By keeping the refresh messages in stable storage before shipment, only one base table scan will be needed.

The message store and forward technique can also be applied in the extended logging approach to support refresh of replicated snapshots. Unlike the R* method where this is the only alternative to obtain the conditional replica mode, the logging approach can fulfill all requirements using the log only. The message store and forward technique does therefore represent an alternative in the logging methods.

Before we end the discussion on support of multiple snapshots based on the basic R* refresh algorithm, we will briefly relate the refresh of multiple snapshots to the single snapshot case.

The refresh processing overhead per snapshot is not increased when compared with the single snapshot case. The base table is scanned for each snapshot individually in order to extract those tuples that have been modified since the last refresh of that snapshot. The scan is performed exactly as in the single snapshot case. Some cost reduction may be obtained. This relates to the fixup operations on the base table performed as part of the refresh operation. NULL values are replaced by real values causing update of some base table tuples. This cost can now be "shared" amongst the snapshots defined on the base table.

We will now turn to the enhancements proposed to the basic R* refresh algorithm to see if they can be applied in the refresh of multiple snapshots as well. We first discuss Enhancement One before going onto Enhancement Two.

The objective of Enhancement One is to reduce the number of refresh messages caused by unqualified inserts. The enhancement relies on the NULL value settings in the base table. These are used to distinguish inserts from updates so as to avoid that unqualified inserts signal a removed region to the snapshot table.

In the case that several snapshots use the same base table for refresh, these NULL value settings can only be used once. Let us return to the original example defined in Figure 47 and Figure 48, in which the three snapshots are defined as independent snapshots. Figure 53 shows the base table after refresh of the three independent snapshots. As can be seen from the figure, snapshot $S_3$, refreshed at time 3:00, has replaced the NULL timestamp for inserted Miami with the current time. In a successive refresh of $S_2$ at time 4:00, Miami can therefore no longer be identified as an insert. If $S_2$ is defined with the snapshot restriction $PriceLevel < 6$, then Miami will be considered as a delete region causing the shipment of unchanged Tenerife.

As a result of this, the effect of Enhancement One will be reduced when used for multiple snapshot support. Only the latest inserts can be excluded from being misinterpreted. In order to ensure continued effect of the enhancement for each snapshot defined on the base table, each snapshot must request for its private timestamp column in the base table. As the number of snapshots defined on the base table increases, the approach will gradually become less attractive. In other words, the reduction in number of refresh messages will hardly outweigh the increase in table space.
### BASE TABLE BEFORE REFRESH AT 4:00

<table>
<thead>
<tr>
<th>Addr</th>
<th>Prev Addr</th>
<th>Time Stamp</th>
<th>Qual 1 2 3</th>
<th>Resort</th>
<th>Value Country</th>
<th>Price Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:</td>
<td>nil</td>
<td>3:00</td>
<td>YNY</td>
<td>Brighton</td>
<td>England</td>
<td>6</td>
</tr>
<tr>
<td>5:</td>
<td>3</td>
<td>NULL</td>
<td>NNY</td>
<td>Cannes</td>
<td>France</td>
<td>6</td>
</tr>
<tr>
<td>7:</td>
<td>5</td>
<td>0:00</td>
<td>YYN</td>
<td>Colombo</td>
<td>Sri Lanka</td>
<td>3</td>
</tr>
<tr>
<td>8:</td>
<td>7</td>
<td>NULL</td>
<td>NYN</td>
<td>Crete</td>
<td>Greece</td>
<td>4</td>
</tr>
<tr>
<td>9:</td>
<td>8</td>
<td>0:00</td>
<td>YYYY</td>
<td>Florence</td>
<td>Italy</td>
<td>5</td>
</tr>
<tr>
<td>10:</td>
<td>9</td>
<td>3:00</td>
<td>NNY</td>
<td>Miami</td>
<td>United States</td>
<td>7</td>
</tr>
<tr>
<td>11:</td>
<td>10</td>
<td>0:00</td>
<td>YYYY</td>
<td>Tenerife</td>
<td>Spain</td>
<td>4</td>
</tr>
</tbody>
</table>

**Figure 54. Base Table Extended for Enhancement Two:** Three snapshots are defined on the base table: Snapshot $S_1$ with snapshot restriction $PriceLevel < 7$, $S_2$ with $PriceLevel < 6$, and $S_3$ with $PriceLevel > 4$. Each snapshot has their private qualification flag which is set if the tuple’s presence in the snapshot is known. The base table has been used in refresh of $S_1$ at time 2:00, $S_2$ at 4:00, and $S_3$ at 3:00.

The same criticism may be used against Enhancement Two relying on a separate qualification flag for each independent snapshots currently defined on the base table. Unlike timestamps, flags can be packed so the space requirement can be met quite easily. The Qual column introduced by the enhancement, cf. Figure 24 on page 64, may hold qualification flags for several snapshots. Figure 54 illustrates this for a base table having three independent snapshots defined.

Enhancement Two may incorporate the objectives of both enhancements proposed to the $R^*$ refresh algorithm, cf. “Chapter 4. Differential Refresh Methods”. The number of refresh messages sent to the snapshot table — caused by unqualified inserts and updates — can therefore be reduced. For instance, the refresh of $S_2$ at time 4:00 will now disregard the Miami tuple that was inserted at time 3:00, see Figure 54. Thus, Tenerife will not be shipped to the snapshot table.

An enhanced version of the basic $R^*$ algorithm may therefore support independent snapshots and with the same accuracy in selecting modifications for refresh as in the single snapshot case — provided the refresh create process is informed of which qualification flag to use.

The enhanced algorithm shows the same performance for the multiple and single snapshot case. The base table is scanned once for each snapshot that is requested refreshed, and the qualification flags are maintained in the same manner as in the single case. The number of tuples that will have their flags updated is determined by the snapshot restriction, so there is no change in this working load. The effect of “sharing” some of the base table fixup cost which is possible in the basic algorithm, is lost. This is caused by the updating of the qualification flags. In a way, all the fixup cost as seen by each snapshot is inherited by the updating of the qualification flags.

Replicated snapshots can also be supported by the enhancements, but it will become difficult to maintain the effect of the enhancements if the looser notion of replica con-
sistency is enforced. The way this is solved in the basic $R^*$ refresh algorithm — and in the logging approaches — is to treat each replica as an independent snapshot.

In the basic algorithm this is accomplished using the timestamp column provided, and with some loss of accuracy in selecting base table tuples for refresh. In the Enhancement One algorithm, a timestamp per snapshot must be used to ensure continued effect of the enhancement. A timestamp column must therefore be allocated to each replica. Similarly, a qualification field must be allocated to each replica using Enhancement Two. The columns must be allocated to the base table at snapshot definition time. They can not be introduced dynamically whenever needed. One may say that the solution embeds a great deal of redundancy, or robustness.

It is questionable if this robustness is ever needed. The looser notion of replica consistency can be supported less ambitiously by relying on the basic refresh algorithm. Whenever refresh of a previously unavailable replica is needed, the basic refresh algorithm is used to support the unconditional mode. Refresh of this replica will consequently not be quite as efficient. The conditional replica mode can be supported by the message store and forward technique.

6.6 Analysis

In this section, we will analyze the extended version of the refresh methods by comparing their performance impact on replicated snapshot maintenance. The analysis is similar to the one given in “Chapter 5. Refresh Method Analysis”, but some alterations have been made to the cost function to reflect the change in weight of the cost contributing cost factors — cf. Figure 5 on page 30.

The estimates expressing the number of entries/tuples satisfying the snapshot restriction, defined in “Chapter 5. Refresh Method Analysis”, are used in extended cost comparison. Although the cost function has been changed, the cost model still includes both processing and communication cost. Like in “Chapter 5. Refresh Method Analysis”, the cost model disregard cost that may be caused by recovery operations, i.e. we assume that the snapshots are successfully refreshed at each snapshot refresh even in the collective refresh of replicas. The cost model does therefore not address the added issue of refreshing previously unavailable snapshots in the event of site failure.

Replication of snapshots can either be obtained by having several independent snapshots defined on a base table, or by creating replicas of snapshots confirming to a common frame definition. This introduces an added complexity to the cost analysis, as we ideally should address both issues, i.e. for a given base table, there may be several independent snapshots defined, some of which may be snapshot frames having replicas defined.

In the following, we will first address the issue of maintaining replicated snapshots. We will show how the cost function can be changed to reflect collective refresh of snapshot replicas. We define $m$ to be the number of replicas based on one snapshot frame definition.

Next, we address the issue of refreshing several independent snapshots. We discuss how the cost function, applied to the snapshot replicas, can be used for estimating the cor-
Extended Log Method Summary

<table>
<thead>
<tr>
<th>Log Method</th>
<th>Implementation</th>
<th>Overall Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential Log</td>
<td>Basic Refresh</td>
<td>( C_{SL} )</td>
</tr>
<tr>
<td></td>
<td>Sort Merge Refresh</td>
<td>( C_{SM} )</td>
</tr>
<tr>
<td>Condensed Log</td>
<td>B-tree</td>
<td>( C_{CB} )</td>
</tr>
<tr>
<td></td>
<td>Linear Hashing</td>
<td>( C_{CH} )</td>
</tr>
<tr>
<td>Enhanced Log</td>
<td>Sort Merge</td>
<td>( C_{ES} )</td>
</tr>
<tr>
<td></td>
<td>B-tree</td>
<td>( C_{EB} )</td>
</tr>
</tbody>
</table>

Definition of Symbols and Formulas

\[
C_{SL} = M_{SL} + \text{CEIL}(M/b)t_4 + m(M_{t_4} + M_{t_4}) \\
C_{SM} = M_{SM} + \text{CEIL}(M/b)t_4 + \text{CEIL}(M_{t_4})t_{SM} + m(L_{t_4} + \text{MIN}(L_{t_4}, \text{CEIL}(sN/bf))2t_4) \\
C_{CB} = L_{TB} + (M-L)L_{TB} + \text{CEIL}(L/bf)t_4 + m(L_{t_4} + \text{MIN}(L_{t_4}, \text{CEIL}(sN/bf))2t_4) \\
C_{CH} = L_{TH} + (M-L)L_{TH} + \text{CEIL}(L/bf)t_4 + m(L_{t_4} + L_{t_4}2t_4) \\
C_{ES} = C_{SM} + \text{CEIL}((N + (p-q)M)/bf)t_4 - m(L_{TB} - E_{c})t_4 \\
C_{EB} = C_{CB} + \text{CEIL}((N + (p-q)M)/bf)t_4 - m(L_{TB} - E_{c})t_4 \\
m = "\text{number of snapshot replicas}" \\
\]

Figure 55. Cost Estimates for Replicated Snapshots — Logging Methods: The log lengths and the number of entries that qualify for the snapshot are given in Figure 28 on page 80. Definition of unit costs, etc., are given in Figure 37 on page 96.

responding cost impact when refreshing \( m \) number of independent snapshots based on the same base table.

Finally, we discuss the results from the cost comparison and emphasize on the factors that may influence on the results. We will also comment on the effect of neglecting the processing cost partly or totally. We end the analysis by comparing differential to a full refresh strategy.

6.6.1 Replicated Snapshots

As we assume that all sites holding a snapshot replica are available at refresh, all \( m \) replicas confirming to a snapshot frame are refreshed collectively. The snapshot frame defines the content of the snapshot replicas. As described in “Chapter 3. Implementing Snapshots”, the refresh manager is responsible for extracting data relevant to the frame, and for distributing refresh messages to each snapshot replica site.

Replicated snapshots have a common modification set. By referring to the cost factors illustrated on Figure 5 on page 30, the following observations are made:
Extended R* Method Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Overall Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Algorithm</td>
<td>$C_{RB}$</td>
</tr>
<tr>
<td>Enhancement One</td>
<td>$C_{R1}$</td>
</tr>
<tr>
<td>Enhancement Two</td>
<td>$C_{R2}$</td>
</tr>
</tbody>
</table>

**Definition of Symbols and Formulas**

\[
C_{RB} = C_{RC} + m(R_B t_b + \text{MIN}(L_p, \text{CEIL}(sN/bf))2t_d)
\]
\[
C_{R1} = C_{RC} + m(R_1 t_b + \text{MIN}(L_p, \text{CEIL}(sN/bf))2t_d)
\]
\[
C_{R2} = C_{RC} + m(R_2 t_b + \text{MIN}(L_p, \text{CEIL}(sN/bf))2t_d)
\]
\[
C_{RC} = \text{CEIL}((N + (p-q)M)/bf)t_d + \text{MIN}(R_F, \text{CEIL}((N + (p-q)M)/bf))t_d
\]
\[
m = \text{"number of snapshot replicas"}
\]

**Figure 56. Cost Estimates for Replicated Snapshots — R* Methods:** The number of entries that qualify for the snapshot are given in Figure 32 on page 87. Definition of unit costs, etc., are given in Figure 38 on page 99.

First, the **identify modification** cost applies to logging of entries in the logging approach, and to marking of base table tuples in the R* approach. This cost is independent of the number of replicas (as the snapshot frame corresponds to one single snapshot).

Secondly, the **refresh create** process introduces the cost of reading the modification set. This cost is also independent of the number of replicas.

Thirdly, **refresh messages** must be sent to each snapshot replica table. The total shipping cost is therefore $m$ times higher than in the single snapshot case.

Finally, the **refresh update** cost applies to each snapshot replica table, and the total cost is thus $m$ times the cost of a single snapshot.

The cost of communicating refresh messages and updating the snapshot tables are therefore more significant to the overall cost of snapshot maintenance than in the single snapshot case. Indeed, the weight of the last two cost factors are determined by the number of replicas $m$.

This is formalized in Figure 55 — giving the cost estimates for replicated snapshots using extended logging, and Figure 56 — giving the corresponding estimates using the extended R* approach. Note that the cost of logging and refresh create processing are the same as in the single snapshot case.
6.6.2 Independent Snapshots

The overall cost of maintaining \( m \) independent snapshots can be estimated in two different ways: One is to estimate the overall cost impact per snapshot. Another is to estimate the overall cost of refreshing the batch of \( m \) snapshots over a long common refresh cycle.

In the first case, the cost of refreshing the one snapshot is basically equal to the cost of refreshing a single snapshot which is discussed in detail in “Chapter 5. Refresh Method Analysis”. Note that refreshing cost does not include the cost of logging, i.e. the first term of the cost estimates given for the logging approach in Figure 37 on page 96 is not included. The cost of logging may be viewed as “shared” between the \( m \) snapshots defined on the base table. The overall cost impact of independent snapshots — in an extended logging scheme — can be calculated per snapshot by reducing the weight of the logging cost. In general, the logging cost can be divided equally between the \( m \) independent snapshots, i.e. the weight of this cost factor is \( 1/m \).

The alternative method is to estimate the overall cost of maintaining the \( m \) snapshots over a long common refresh cycle. The length of this cycle is determined by the snapshot least frequently refreshed, i.e. \( M \) is the modification set size applied in refresh of this snapshot. The problem associated with this method is that we have to specify the selectivity and refresh frequency of each snapshot in the batch. In the previous estimation method — where the cost impact is calculated on the snapshot individually — this is avoided.

The overall cost impact, in the case of \( m \) independent snapshots, is therefore estimated per snapshot. Also, this enables a comparison of refreshing a single snapshot with refresh of a snapshot when multiples are defined.

Above, we stated that the refreshing cost of independent snapshots is basically the same as in the single snapshot case. This is an approximation. In some cases, the cost may be less. Yet in others, the cost may be higher. The inaccuracy, if any, introduced in each approach is therefore discussed in the following.

The cost of refreshing a snapshot based on a sequential log is independent of the number of snapshots currently defined on the base table. The number of log entries — reflecting the changes made to the base table since the last refresh of the snapshot — is the same in the extended sequential log. Therefore, exactly the same number of log entries are scanned as in the single case. This applies to both refresh strategies based on sequential logging. We assume that the sort merge strategy sorts the relevant portion of the log at each refresh. Some reduction in cost may be obtained if we keep the sorted image in the extended scheme, see the discussion earlier in this chapter.

As for the condensed logging approach, the number of log entries scanned at refresh may be higher than in the single snapshot case. In the extended condensed log, there may be several entries reflecting changes to the same base table tuple — depending on the update activity and the refresh frequency of the other \((m-1)\) snapshots. As a result, the “collision” effect is decreased, and in the worst case, we may end up scanning the same number of entries as in the sequential approach. The merging of neighbour sections scheme will however try to maintain the condensed property of the log, but does
## Extended Log Method Summary

<table>
<thead>
<tr>
<th>Log Method</th>
<th>Implementation</th>
<th>Overall Cost</th>
<th>$C_{SL}$</th>
<th>$C_{SM}$</th>
<th>$C_{CB}$</th>
<th>$C_{CH}$</th>
<th>$C_{ES}$</th>
<th>$C_{EB}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential Log</td>
<td>Basic Refresh</td>
<td></td>
<td>$C_{SL}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sort Merge Refresh</td>
<td></td>
<td></td>
<td>$C_{SM}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Condensed Log</td>
<td>B-tree</td>
<td></td>
<td></td>
<td></td>
<td>$C_{CB}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Linear Hashing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$C_{CH}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enhanced Log</td>
<td>Sort Merge</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$C_{ES}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B-tree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$C_{EB}$</td>
</tr>
</tbody>
</table>

### Definition of Symbols and Formulas

\[
C_{SL} = (l/m)M_{SL} + \text{CEIL}(M/b)t_4 + M_t t_s + M_t 2t_d
\]
\[
C_{SM} = (l/m)M_{SM} + \text{CEIL}(M/b)t_4 + \text{CEIL}(M/b)t_{SM} + L_t t_s + \text{MIN}(L_t, \text{CEIL}(sN/bf)) 2t_d
\]
\[
C_{CB} = (l/m)M_{IB} + (M-L)_{IB} + \text{CEIL}(L/bf)t_4 + L_t t_s + \text{MIN}(L_t, \text{CEIL}(sN/bf)) 2t_d
\]
\[
C_{CH} = (l/m)M_{IH} + (M-L)_{IH} + \text{CEIL}(L/bf)t_4 + L_t t_s + L_t 2t_d
\]
\[
C_{ES} = C_{SM} + \text{CEIL}((N + (p-q)M)/bf)t_4 - (L_t - E_s)t_s
\]
\[
C_{EB} = C_{CB} + \text{CEIL}((N + (p-q)M)/bf)t_4 - (L_t - E_s)t_s
\]

$m$ = "number of independent snapshots"

---

**Figure 57. Cost Estimates for Independent Snapshots — Logging Methods:** The log lengths and the number of entries that qualify for the snapshot are given in Figure 28 on page 80. Definition of unit costs, etc., are given in Figure 37 on page 96.

not come free of charge. Thus, by assuming the same cost as in the single case, we introduce some inaccuracy. The cost is generally underestimated.

In the R* approach, the approximation is reasonable, but will generally overestimate the cost of the basic R* refresh algorithm as explained below.

In the R* approach, the base table is scanned at refresh time, and NULL values are replaced with real values. The number of fixups may be reduced as the number of independent snapshots increases — especially for large modification sets. This reduction can not be estimated easily. We can not simply divide the cost equally between the $m$ snapshots, as the number of modifications is partially a function of the "collision" effect. This means, as in the condensed log, that the number of marked modifications, in the worst case, remains the same.

When applying the enhancements to the R* algorithm, the net effect is the same as in the single snapshot case — in Enhancement One a separate timestamp is used, in Enhancement Two a separate qualification column is used. Maintaining these columns introduce the same cost impact as in the single snapshot case — and cover the cost of base table fixups.
In view of this, we may state that the approximation is fair to the three approaches. As the refreshing cost is considered the same as in the single snapshot case, only the logging cost will be different when estimating the overall cost impact of multiple, independent snapshots. This has been reflected in the cost estimates for the logging approaches given in Figure 57. Note that the weight of the logging cost factor is \( \frac{1}{m} \). Since the cost of identifying modifications is disregarded in the \( R^* \) approach, the corresponding cost estimates for the \( R^* \) approach are the same as given in Figure 38 on page 99.

6.6.3 Analytic Results

A cost comparison — identical to the one described in "Chapter 5. Refresh Method Analysis" — has been carried out for the extended refresh approach. The objective of this comparison is to study the applicability of each refresh method by relating the cost impact to the one introduced by a basic sequential refresh strategy, i.e. results are given as trade off in cost between each alternative and the basic refresh of sequential logging. The cost trade off is measured in terms of disk and message I/O units \( (e_i = e_d) \).

The same parameter values as applied in "Chapter 5. Refresh Method Analysis" are used. These are:

1. The number of base table tuples prior to modification; \( N = 100,000 \).
2. The maximum number of entries (or tuples) per data page; \( b = 10 \).
3. In a B*-tree, the maximum number of index entries per index page; \( b_i = 146 \), and the load factor; \( f = 0.83 \). These figures are taken from [MIMES 85].
4. In linear hashing, the following figures are used, taken from [LITW 80]; \( f = 0.61 \), \( e_{th} = 3.14e_d \), \( e_{th} = (1.07 + 1)e_d \).
5. In the sort merge refresh strategy, the merge ratio is \( k = 10 \).

We have analyzed the refresh methods, including alternative implementations, under various base table modification conditions and for different snapshot selectivities.

In the analysis of the extended approach — we have addressed replicated and independent snapshots separately. First, we present the analytic results for replicated snapshots, and we begin by commenting on the less prosperous alternatives.

Figure 58 shows the performance impact of condensed logging versus sequential logging when the number of replicas \( m \) vary between 1 and 5. Curves for the restrictive snapshot and small \( m \) are omitted from the figure as they distort the picture and show very poor performance results anyway. The following observations can be made when comparing the two alternatives for condensed logging.

Linear hashing introduces a smaller cost impact for single snapshots — cf. the analytic results presented in Figure 40 on page 102 — whereas the B-tree implementation shows significant improvements when the number of replicas is increased. Still, for very restrictive snapshots \( s < 0.25 \), the approach will not pay off even for several replicas.
Figure 58. **Static Base Table (Condensed Logging):** The curves give the cost trade off in
I/O units as % of base table size for some number of replicas and selectivity —
s = 1.0 solid lines, s = 0.25 dashed lines. The section of a curve above the reference
line means that condensed logging wins, whereas negative values means that the
basic sequential logging wins.

Figure 58 displays curves for a static base table. A similar trend is found when applying
the condensed log to incremental and dynamic base tables. Further, the enhancement
to logging does not improve the approach to any extent. Some reduction in cost can
be obtained for larger modification sets, but will not change the applicability of the ap-
proach versus sequential logging.
Figure 59. **Static Base Table (Sequential Logging and R*):** The curves give the cost trade off in I/O units as % of base table size for some number of replicas and selectivity — $s = 1.0$ solid lines, $s = 0.05$ dashed lines. The section of a curve above the reference line means that sort merge or R* Enhancement Two wins, whereas negative values means that the basic sequential logging wins.

The results indicate that the condensed logging approach introduce a large cost for restrictive snapshots whereas the method performs well for full-copy snapshots when the number of replicas exceed 2. We may therefore conclude that the applicability of the condensed logging approach is limited.
We are therefore left with the sequential and R* approach. To limit the discussion, we only present the results using the Enhancement Two refresh algorithm. Figure 59, Figure 60, and Figure 61 show the performance impact for a static, incremental, and dynamic base table respectively. The following observations can be made.
Figure 61. Dynamic Base Table ($p = 0.2$, $q = 0.1$, $r = 0.7$): The curves give the cost trade off in I/O units as % of base table size for some number of replicas and selectivity — $s = 1.0$ solid lines, $s = 0.05$ dashed lines. The section of a curve above the reference line means that sort merge or R* Enhancement Two wins, whereas negative values means that the basic sequential logging wins.

In general, the enhanced R* algorithm is a loser for very small modification sets. This is mainly due to the overhead involved in scanning the base table. On the other hand, large saving may be obtained when using the R* approach for modification sets exceeding approximately 10 percent of the base table size.
The sort merge refresh strategy does in general pay off — when compared with the basic refresh strategy — for very restrictive snapshots. The larger the number of replicas, the larger the savings. For less restrictive snapshots, the sort merge pays off when the modification set size exceeds 10 to 16 percent of the base table size — depending on the number of replicas involved.

Both the sort merge refresh strategy and the R* approach provide large savings when the number of replicas increases. This is mainly due to the reduction in number of messages sent to the snapshot tables when compared with the basic sequential refresh algorithm. The reduction in snapshot table update cost does also favour these strategies.

Another interesting observation — from the sort merge point of view — is that the sort merge performs nearly as good as R* for less restrictive snapshots.

We now turn to the case of independent snapshots. We recall from the discussion of the cost expressions, that i) the approximation will in general underestimate the refreshing cost of condensed logging, ii) the assumption holds for sequential logging and, iii) the assumption holds for the R* Enhancement Two algorithm.

Figure 62 shows the performance impact per independent snapshot for an extended condensed logging scheme. Curves are given for a full-copy snapshot and for a snapshot with selectivity $s = 0.25$. Curves for very restrictive snapshots show very poor performance results and are omitted from the figure.

When comparing the two alternatives for condensed logging with sequential logging, we see that the B-tree version provides larger savings than the linear hash alternative. The reduced weight of the logging cost does therefore favour the B-tree alternative most. Another way of seeing this, is that the cost of updating the snapshot table becomes more significant to the overall cost, which disfavours the linear hashing alternative. Although the extended condensed logging approach provides significant savings for less restrictive snapshots, the applicability of the approach is limited. For very restrictive snapshots ($s < 0.25$), the approach is a clear looser — compared with sequential logging — even for large modification sets and number of snapshots. Adding the fact that the results are generally underestimated, restricts the applicability even further.

This indicates that sequential logging is to be preferred to condensed logging. Besides, the sequential approach supports two refresh strategies that complement each other. Up to a certain modification set size, the basic refresh strategy shows the best performance. From there on, it becomes profitable to apply the sort merge strategy. This break even point may be determined at refresh invocation time — and we may chose strategy accordingly. The logging cost is obviously the same for each strategy. The difference in overall cost for the two does therefore depend on the refreshing cost only, and is independent of the number of snapshots defined on a base table. Thus, the cost trade off between basic and sort merge refresh is the same as shown for single snapshots in "Chapter 5. Refresh Method Analysis". Results are displayed in Figure 39 on page 101, Figure 42 on page 104, and Figure 44 on page 106.

In the analysis provided for the sort merge refresh strategy, we have assumed sorting of all modifications in the extended sequential log that satisfy the snapshot restriction.
(a) Condensed Log Based on B-tree.

(b) Condensed Log Based on Linear Hashing.

Figure 62. Static Base Table (Independent Snapshots, Condensed Logging): The curves give the cost trade off in I/O units as % of base table size for some number of independent snapshots and selectivity — $s = 1.0$ solid lines, $s = 0.25$ dashed lines. The section of a curve above the reference line means that condensed logging wins, whereas negative values means that the basic sequential logging wins.

Alternatively, we may use the intermediate, sorted log for future refresh of other snapshots, cf. the discussion earlier in this chapter.
Figure 63. Static Base Table (Independent Snapshots, R* approach): The curves give the cost trade off in I/O units as % of base table size for some number of independent snapshots and selectivity — $s = 1.0$ solid lines, $s = 0.05$ dashed lines. The section of a curve above the reference line means that the R* Enhancement Two wins, whereas negative values means that the basic sequential logging wins.

Figure 64. Incremental Base Table (Independent Snapshots, $p = 0.1$, $r = 0.9$): The curves give the cost trade off in I/O units as % of base table size for some number of independent snapshots and selectivity — $s = 1.0$ solid lines, $s = 0.05$ dashed lines. The section of a curve above the reference line means that the R* Enhancement Two wins, whereas negative values means that the basic sequential logging wins.
Figure 65. Dynamic Base Table (Independent Snapshots, $p = 0.2$, $q = 0.1$, $r = 0.7$): The curves give the cost trade off in I/O units as % of base table size for some number of independent snapshots and selectivity — $s = 1.0$ solid lines, $s = 0.05$ dashed lines. The section of a curve above the reference line means that the R* Enhancement Two wins, whereas negative values means that the basic sequential logging wins.

Figure 63, Figure 64, and Figure 65 show the performance impact of independent snapshots using the R* approach versus sequential logging. Figures are taken for the Enhancement Two algorithm as this algorithm provides a higher accuracy than the other R* algorithms, cf. "Chapter 5. Refresh Method Analysis". In the R* approach, the overall cost impact is independent of the number of snapshots defined on the base table. However, when we relate the cost to sequential logging — in which a cost reduction is obtained as the number of independent snapshots increases — we see that the applicability of the R* approach is reduced. In particular, this applies for the restrictive snapshots. Another way of seeing this is that the applicability of sequential logging increases as a function of the number of snapshots. Larger modification sets are therefore required to justify the use of the R* approach. For less restrictive snapshots, the R* method still performs well, but the sort merge may provide nearly as good results for large modification sets.

The analytic results shown are based on several parameter values. A discussion of their significance to the overall result is given in "Chapter 5. Refresh Method Analysis". We found that a reduction in number of disk page I/O is more noticeable in the cost balance than an increase/decrease in communication unit cost. In other words, by reducing the processing cost, the difference in communication overhead imposed by the refresh methods becomes noticeable. If we ignore the processing cost completely, like in [LIND 86], the methods providing for the highest accuracy will perform best. For details, see the discussion in "Chapter 5. Refresh Method Analysis".

The analysis has so far focused on differential refresh strategies. We will now comment on the applicability of differential versus a full refresh strategy. Like in "Chapter 5.
Figure 66. **Differential vs. Full Refresh (Static Base Table):** The curves give the cost trade off in I/O units as % of base table size for some number of replicas and selectivity — $s = 1.0$ solid lines, $s = 0.05$ dashed lines. The section of a curve above the reference line means that differential refresh is profitable, whereas negative values means that a full refresh wins.

Refresh Method Analysis”, the logging cost is excluded in this scenario, as the choice is between applying a full strategy to a differential one at refresh invocation time.

Figure 66 shows the trade off in the choice between (a) sort merge versus full refresh, and (b) R* Enhancement Two versus full refresh. We see that the sort merge strategy
is profitable for a larger range than the R* method. A full refresh will perform better than a differential sort merge strategy when the number of modifications approaches the number of tuples in the base table. For very restrictive snapshots, this point lies around 72 percent of the base table size. For full-copy snapshots, this is around 69 percent for independent snapshots, and may be well above 100 percent for replicated snapshots (outside the value range of the figure) due to the "collision" effect of modifications. Even though the total number of changes exceeds the number of tuples in the base table, the merging of modification entries — that takes place in the sort merge refresh strategy — will reduce the number of refresh messages.

In the R* approach, differential refresh of independent snapshots does not pay off if the snapshot is very restrictive — unless the modification set size is below 5 percent of the base table size. In a collective refresh of snapshot replicas, a differential refresh will pay off for a larger range as the number of replicas increases. In the case of full-copy snapshots, the R* approach is, in general, profitable. The reason for this being a reduction in the number of refresh messages sent to the snapshot tables when compared with a full refresh. This is also the conclusion in [LIND 86].

In the analysis we have studied replicated and independent snapshots separately. In cases where some of the independent snapshots are replicated, we obtain a larger "sharing" of the logging cost. Whereas the R* approach remains the same, the collective refresh using logging is reduced (per independent snapshot). Overall, this will favour the logging scheme to R* — see Figure 63, Figure 64, and Figure 65.

In view of this, and the other analytic results presented above, the following conclusion can be made. The extension to the logging approach can be done more cost effectively than in the R* approach when measured per snapshot. Still, the R* approach is a good alternative provided the snapshots are not very restrictive. In the latter case, one can resort to a full refresh strategy. In other words, the full refresh strategy will complement the differential refresh strategy of R*. Out of the two logging methods, the sequential logging approach performs significantly better — and the two refresh methods are complementary. Like in the R* approach, one can resort to a full refresh strategy. This is only needed when the modification set becomes very large.

6.7 Summary and Concluding Remarks

We have described how the differential approach to snapshot maintenance can be extended to fulfill the requirement for snapshot replication posed in "Chapter I. Distributed Databases and Snapshots". If we disregard the replica consistency issue, this is basically accomplished by using one common technique: Associated with each snapshot we keep a recording of its last refresh.

In the sequential logging approach, a refresh mark — referring to the actual position in the sequential log — is used. In the condensed log, a timestamp is used to distinguish new modifications from older ones. In the R* approach, a timestamp is already provided. At the next refresh of the snapshot, only modifications submitted since the last refresh will have to be considered. In this way, a snapshot refresh may proceed more or less as in the single snapshot case. At the same time, the extended approach integrates modifications relevant to a number of snapshots.
The integration of the replication requirement can be done with minor modification to
the sequential logging scheme. Only a refresh mark is introduced — enabling a section-
ing of the sequential log. As for the condensed logging scheme, a timestamp column has
been added to the log table, and the logic of the refresh create algorithm has been ex-
tended quite extensively. In the R* approach, no change is needed as long as we use the
basic refresh algorithm. In the enhanced version of the refresh algorithm, separate
qualification columns are needed if continued effect of the enhancement is wished for.

Collective refresh of replicated snapshots rises the issue of which consistency level to
enforce in the event of unavailable replica sites at refresh time. This issue is named
replica consistency. If mutual consistency is enforced, then the replicas are always re-
freshed collectively. This can be fulfilled by handling the frame as one snapshot — a
technique that can be applied in both the logging and the R* approach.

The unconditional replica mode can also be supported easily. Each snapshot replica is
simply treated as an independent snapshot. The conditional replica mode can only be
directly supported in the logging approaches. In the R* approach, there is no notion
of tuple images of different refresh time. Therefore, a store and forward technique must
be employed to support this mode. This technique can be applied in the logging ap-
proach as a complementary solution.

A full refresh strategy — in which the snapshot table is reproduced from the base table
— supports replication of snapshots in the same way as the R* approach. The replica
consistency issue is thus only partially supported.

We have earlier — in “Chapter 3. Implementing Snapshots”, and “Chapter 4. Differential
Refresh Methods” — addressed the implication of supporting more complex snap-
shots, i.e. snapshots defined as a join of several base tables. The R* approach can only
be used in differential refresh of snapshots based on one base table. The logging ap-
proach, on the other hand, can be used in differential refresh of more complex snapshots.
Modifications from several base tables can be combined into modifications that reflect
changes relevant to a join-based snapshot. In the extended logging approach this means
extracting data from more than one log table.

In the condensed log, the selection is ruled by the timestamp value of the log entries.
The timestamp of the snapshot’s last refresh can therefore be used as selection criteria
in the relevant log tables.

In the sequential log, the refresh mark is local to each log table. A translation, from the
snapshot’s last refresh timestamp to refresh marks in the relevant log tables, is needed.
A table similar to the one given in Figure 47 can be used to assist this translation. The
table maps refresh marks to timestamps, and can either be kept in the system catalog,
or in the header of the sequential log to avoid the catalog lookup. In this way, both
logging approaches can be used in differential refresh of more complex snapshots.
Chapter 7. Conclusion

In “Chapter 1. Distributed Databases and Snapshots”, we discussed the main issues that may affect the overall performance of a distributed database. One of the tactics used to improve performance is data replication. By storing copies of data at the sites where the data is frequently used, the need for costly, remote data access, is decreased. At the same time, this provides what is called locality of data.

We also commented on the problems related to replication of data. The expected improvement in performance may, in practice, be hard to achieve due to the added cost of maintaining the replicated data. In some situations, even the availability to data may be temporarily lost.

A database snapshot mechanism represents a cost effective substitute for replicated data in a distributed database, where an up-to-date version of data is not of paramount importance, or where data as of a particular time is required. The content of a database snapshot can periodically be 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(s) involved in the snapshot are detected. The last mentioned issue forms the motivation for the research in this thesis.

We ended “Chapter 1. Distributed Databases and Snapshots” by posing three requirements to a database snapshot mechanism. We will now summarize the work in this thesis by discussing what we have done to fulfill these requirements, and we will see how the differential refresh approach addresses these issues.

“Chapter 2. Snapshot Definition and Semantics” proposes a series of statements for definition and manipulation of database snapshots in a distributed database environment. These statements are summarized in Figure 3 on page 22. The statements allow for snapshots based on one or several base tables, and for replication of snapshots — either defined as several independent snapshots with an individual refresh cycle, or as snapshot replicas based on a common snapshot frame which are refreshed collectively. Snapshot replicas introduce another problem that must be solved in the event of site failure at refresh invocation time. Three levels of replica consistency can be enforced, and may be specified in the snapshot frame definition.

In “Chapter 3. Implementing Snapshots”, we introduced a spectrum of refresh methods that can assist the refresh of database snapshots. A differential refresh method and a full refresh method represents both ends of this spectrum. Differential refresh will update a snapshot in an incremental manner. In cases where this strategy can not be applied, a full refresh strategy must be used. This is the case if the snapshot is defined as a complex query, and a differential refresh strategy does not hold sufficient information to update
the snapshot table incrementally. Methods for supporting complex queries require a
recording of modifications made to each base table involved in the snapshot definition
[BLAK 86, HANS 87].

The site failure problem is also addressed in "Chapter 3. Implementing Snapshots", but
from an operational point of view. In the event of a site failure during the refresh op-
eration, data availability may be temporarily degraded. This can be avoided using a
presumed abort protocol for committing a refresh transaction, and is useful if data can
be selected for refresh in local read transactions.

"Chapter 3. Implementing Snapshots" also emphasizes the performance aspect of a
snapshot mechanism, and the four main factors contributing to the overall cost of
snapshot maintenance are identified, cf. Figure 5 on page 30. The figure also sketches
the major steps in support of a differential refresh strategy.

A differential refresh strategy bases its operation on additional information stored in the
database. This information is entered when the base table(s) involved in a snapshot are
subject to modification. This process — named identify modification — introduces some
overhead to the normal update operations on base tables. At refresh time, the refresh
create process extracts the changes that are relevant to the snapshot, refresh messages
are shipped to the snapshot site, and the refresh update process submits the changes to
the snapshot table.

"Chapter 4. Differential Refresh Methods" describes a series of differential refresh
methods having in common that modifications made to the base table since last refresh,
are recorded on a tuple basis. Two main approaches are discussed. The logging ap-
proach — in which modifications are recorded in a separate log, and the R* approach —
in which modifications are marked off directly in the base table for those tuples changed.
In the R* approach [LIND 86], the base table is extended with two columns to record
the changes made since the last refresh. In the logging approach, a sequential and a
condensed logging scheme is proposed.

The main advantage of the logging approach is that it allows the refresh create process
to operate on a domain separated from the base table. Data availability can therefore
be ensured for the base tables during snapshot refresh. Since refresh of snapshots in the
R* approach must be terminated by a full 2-phase-commitment, the base table may be
locked for some time in case of site failure.

The main advantage of the R* approach is the minimal overhead imposed on the normal
base table operations. The method does not degrade the performance of base table op-
erations. For some of the logging alternatives, the logging of modifications represents
a rather significant overhead. This is especially true for the condensed logging method.
In the sequential log, this overhead is not so dominating.

Another disadvantage of the R* approach is that the method can not support differential
refresh of complex snapshot restrictions. Delete modifications are never recorded in the
R* approach, and consequently, the method is ruled out when it comes to supporting
join based snapshots. The logging approach is not constrained by this. Here, all mod-
ifications are recorded — either all of them, or in a condensed form.
Each approach is discussed in detail in "Chapter 4. Differential Refresh Methods", and algorithms for the refresh create and update process are defined. The discussion is kept within the scope of one snapshot per base table, i.e. single snapshots. Two complementary refresh strategies are proposed for the sequential log. Two alternatives for realization of condensed logging are suggested, and the basic refresh algorithm of R* is discussed. We also proposed two enhancements to the R* refresh algorithm, and an enhancement to the logging scheme. The refresh methods differ in their accuracy of selecting entries for updating the snapshot, and the working load on the refresh create and update processes vary.

As for the sequential logging methods, the sort merge refresh strategy is much more selective than the basic refresh strategy. The condensed logging scheme provides for the same accuracy as the sort merge of sequential log. The number of refresh messages may be reduced even further by eliminating the shipment of deleted messages. This is the main objective of the R* approach, and the enhancement to logging — where deletes are sent as part of insert and update messages. The two enhancements proposed for R* refresh cause a smaller set of refresh messages than the basic refresh algorithm.

As for the working load, the refresh update process does in general benefit from having the refresh message received in key order sequence — as in the R* approach, the sorted condensed log, and the sort merge refresh of sequential log. The working load on refresh create is, in the logging approach, proportional to the number of log entries. In the R* approach, the base table size determines this cost.

In "Chapter 5. Refresh Method Analysis", an analytic model of the differential refresh approach is established. Expressions estimating the number of log entries satisfying the snapshot restriction are derived, cf. Figure 28 on page 80. Similar expressions are found for the R* method, cf. Figure 32 on page 87. These are used in estimating the overall cost of snapshot maintenance. The cost function includes both processing and communication cost, mainly by addressing each of the four cost factors identified in Figure 5 on page 30. Figure 37 on page 96, and Figure 38 on page 99 summarize the cost estimates. A cost comparison of the differential refresh methods is performed, and the results are compared with those of a full refresh strategy.

"Chapter 6. Extending Refresh for Snapshot Replication" describes how the differential approach to snapshot maintenance can be extended to fulfill the requirement for snapshot replication. This is basically accomplished by keeping a timestamp of the last refresh of each snapshot. The integration of the replication requirement can be done with minor modification to the sequential logging scheme. Only a refresh mark is introduced — enabling a sectioning of the sequential log. As for the condensed logging scheme, a timestamp column has been added to the log table, and the logic of the refresh create algorithm has been extended quite extensively. In the R* approach, no change is needed as long as we use the basic refresh algorithm. In the enhanced version of the refresh algorithm, separate qualification columns are needed if continued effect of the enhancement is wished for.

Refresh of snapshots may proceed more or less as in the single snapshot case. In the condensed log, the selection is ruled by the timestamp value of the log entries, and is in this sense similar to the R* approach. In the sequential log, the selection of relevant sections of the log are ruled by a refresh mark, which is local to each log table. A translation from refresh timestamp to refresh mark is therefore needed.
The replica consistency issue is discussed in detail in "Chapter 6. Extending Refresh for Snapshot Replication". If mutual consistency is enforced, snapshot replicas are always refreshed collectively, and coordinated by the snapshot frame. The unconditional replica mode can easily be supported by treating each snapshot replica as an independent snapshot. The conditional replica mode can only be directly supported in the logging approach. In the R* approach, there is no notion of tuple images of different refresh time, and a store and forward technique must be employed to support this mode. This technique can be applied in the logging approach as a complementary solution. A full refresh strategy will support replication of snapshots in the same way as the R* approach.

The chapter ends by defining an extended cost model that accounts for the replication of snapshots. When compared with the estimates given in "Chapter 5. Refresh Method Analysis", the weight of the logging cost is reduced when calculating the overall cost per independent snapshot, cf. Figure 57 on page 138, and the weight of shipment and refresh update cost are increased when calculating the overall cost of collective refresh, cf. Figure 55 on page 135, and Figure 56 on page 136.

The cost analysis provided in "Chapter 5. Refresh Method Analysis", and "Chapter 6. Extending Refresh for Snapshot Replication", show that the sequential logging approach performs best for restrictive snapshots, and for small modification sets in general. The extension to support snapshot replication can also be done more cost effectively in the sequential approach, which strengthens its position in comparison to the other alternatives. Unlike the other approaches, the sequential log supports two complementary refresh strategies, and the applicability of each alternative can be dynamically decided. In cases where the modification set is very large, one can still resort to a full refresh strategy.

The cost analysis further shows that the R* approach is a good alternative to logging when more than approximately 10 percent of the base table tuples are modified. However, its applicability is reduced, when compared with sequential logging, as the number of snapshots defined on the base table increases. The R* Enhancement Two algorithm, proposed in this thesis, shows some improvement when compared with the basic R* algorithm. The improvements are most significant for medium range selectivities, and when a medium to smaller ratio of the base table is modified.

A condensed logging approach can only be justified in cases where the modification set is large and the snapshots are less restrictive. However, if we ignore the processing cost, then this approach becomes interesting when combined with the enhancement to logging. This enhancement can also be applied to the sort merge refresh strategy of sequential logging, providing identical results, as the cost function now only addresses communication cost. The R* Enhancement Two algorithm performs nearly as well, except for cases where a large percentage of the modifications are deletes.

Although R* shows good performance and introduces virtually no overhead on normal base table operations, the method can not support join based snapshots. This is a major disadvantage with the approach, considering the large savings that can be obtained by applying an incremental refresh method for these snapshots.
In the logging approach, the log can be used to compose refresh messages relevant to e.g. join-based snapshots. The cost reductions that can be obtained, compared with the cost of using a full refresh strategy, has not been addressed in this thesis. However, the cost model provides the basis for such an analysis. Due to the multiplicity effect of a join, one may assume that the applicability of differential refresh is strengthened versus a full refresh strategy.

The last issue ought to be looking into when implementing a full snapshot mechanism. An implementation of database snapshots is planned for the MIMER* prototype. It will be interesting to see the snapshot mechanism implemented and in operation, and to study the sequential logging approach and its two refresh strategies more closely. A full implementation of snapshots, providing support for join based snapshots, will embed the algorithms proposed in [BLAK 86] and [HANS 87].

The research described in this thesis provides a platform to future research within the field of distributed databases and data replication. The snapshot mechanism provides users with a read-only copy of some portion of the database, and is useful to applications where an up-to-date consistency is not required. In cases where data is extracted for retrieval and update, the mechanism can not be used.

If the snapshot table is presented to the users as an ordinary table, update to the table could be allowed — but will only cause local updating. These may be lost in a future differential refresh of the snapshot, and will always be lost if a full refresh has been issued.

The snapshot mechanism can be extended to support this feature if the requirement is relevant. We may name this a local changeable snapshot, in which the content is basically extracted from a consistent state of the database, and where local changes to the snapshot table are allowed. In order to maintain the local updates at a later stage, a two-way differential refresh strategy may be used. First, a refresh the “master copy” of the snapshot is issued to reflect the current database state. Then, we may add the updates submitted to the local snapshot table, if it is possible to merge them with the “master copy”. This can also be performed as a differential refresh operation if the local changes are recorded in a log for the snapshot table.

Another extension to the mechanism may be a global changeable snapshot, in which updates to the snapshot table are reflected in the base table(s) immediately or as soon as possible. The changes may thus be reflected in other snapshots at a later refresh. The update activity on a given snapshot may be restricted to some portion of the base table(s). If no-overlapping update restrictions are specified, then the mechanism will enable a horizontal partitioning of base table(s).

These issues ought to be investigated in future research. The implication of supporting user-defined events for triggering snapshot refresh should also be addressed in future work.
Bibliography


Bibliography 159