Concurrent Transaction and Query Processing — A Compensation-Based Approach

Øystein Grøvlen

Department of Computer and Information Science
Norwegian University of Science and Technology

August 6, 1998
Abstract

Conventional concurrency control techniques are not well suited for running a mix of simple and complex transactions. In order to achieve transaction-consistency, long-lived transactions will hold locks on accessed data items for a long time. These locks will block updates by other transactions. Thus, introducing ad-hoc queries in a production database system, may significantly increase response times for updating transactions due to lock contention.

Compensation-based query processing has been proposed as a solution to this problem. Using this approach, a query avoids lock contention by reading an inconsistent version of the database. However, the query will compensate for updates made by concurrent transactions before returning the result. In this thesis, several algorithms for compensation-based query processing are presented. The main focus is put on a new compensation-based algorithm that uses undo/no-redo compensation. A query reads an inconsistent version of the database, but all updates made by concurrent transactions are later undone so that the query result only reflects changes made by transactions that committed before the start of the query. Queries process the database internal log to obtain information on concurrent updates. Thus, query execution imposes no extra work on updating transactions.

Undo/no-redo compensation has the advantage that it allows for pipelining the compensation with the scanning of base relations. In addition to increased performance, this reduces the requirement for temporary storage compared to earlier algorithms for compensation-based query processing. Undo/no-redo compensation also allows for efficient pipelining with relational algebra operations, and algorithms for the most used algebra operations are presented in this thesis.

A simulation study of the proposed algorithm shows that response times for query execution is significantly improved compared to the earlier compensation-based algorithms. For systems with spare capacity available for query execution, compensation gives only a small increase in query response times compared to executing queries without consistency requirements. At the same time, the response time effect on concurrent transactions are negligible.

This thesis also presents two algorithms for compensation-based query processing in distributed database systems. The algorithms differ in how the transaction-consistent set of transactions that should be reflected in the query result is computed. One algorithm centralizes this computation to a single node, while the other algorithm exploits the messages of the two-phase commit protocol to distribute the computation to all nodes participating in the query. The distributed computation is the most efficient and scalable, but the centralized computation will require less modifications to an existing database system.
Contents

Abstract iii
Preface ix

I Introduction 1

1 Concurrent Transaction and Query Processing 3

2 Basic Concepts 7
  2.1 DBMS Concepts .............................................. 7
  2.1.1 Data Model .............................................. 7
  2.1.2 Database Operations .................................... 8
  2.2 Transaction Model ........................................... 8
  2.2.1 Concurrency Control .................................... 8
  2.2.2 Recovery Concepts ..................................... 10

3 Related Work 15
  3.1 Degrees of Consistency ..................................... 15
  3.2 Database Snapshots ......................................... 16
  3.3 Transient Versioning ....................................... 17
    3.3.1 Chan et al., 1982 ...................................... 18
    3.3.2 Agrawal and Sengupta, 1989 ............................ 20
    3.3.3 Bober and Carey, 1992 ................................ 21
    3.3.4 Mohan et al., 1992 .................................... 24
    3.3.5 Wu et al., 1993 ........................................ 25
    3.3.6 Transient Versioning in Commercial DBMS ............ 26
    3.3.7 Distributed Transient Versioning ....................... 26
  3.4 Compensation-Based Query Processing ..................... 28
    3.4.1 Disadvantages .......................................... 30
  3.5 Distributed Database Checkpointing ....................... 31
    3.5.1 Pu, 1986 .............................................. 32
    3.5.2 Son and Agrawala, 1989 ............................... 33
    3.5.3 Pilarski and Kameda, 1992 ............................ 34
    3.5.4 Kim and Park, 1994 .................................. 35
  3.6 Concluding Remarks ....................................... 37
## II  Centralized Query Processing

### 4  Log-Oriented Compensation-Based Query Processing  41

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Basic Idea</td>
<td>41</td>
</tr>
<tr>
<td>4.2</td>
<td>Undo/No-Redo Compensation</td>
<td>44</td>
</tr>
<tr>
<td>4.2.1</td>
<td>The Log Processing Threads</td>
<td>45</td>
</tr>
<tr>
<td>4.2.2</td>
<td>The Scan Thread</td>
<td>48</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Synchronizing the Scan Thread and the Forward Log Processing Thread</td>
<td>49</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Example</td>
<td>49</td>
</tr>
<tr>
<td>4.3</td>
<td>Extensions and Optimizations of Undo/No-Redo Compensation</td>
<td>51</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Preserving Scan Order in the Query Output</td>
<td>51</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Space Optimization of the Update-Table</td>
<td>52</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Index-Based Scans</td>
<td>53</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Multi-Query Transactions</td>
<td>56</td>
</tr>
<tr>
<td>4.3.5</td>
<td>Delta Operations</td>
<td>59</td>
</tr>
<tr>
<td>4.3.6</td>
<td>Complete Tuple Logging</td>
<td>60</td>
</tr>
<tr>
<td>4.3.7</td>
<td>Compatibility with Other Concurrency Control Methods than</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Strict 2PL</td>
<td>61</td>
</tr>
<tr>
<td>4.4</td>
<td>Storing the Update-Table on Disk</td>
<td>63</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Hash-Based Partitioning</td>
<td>63</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Range-Based Partitioning</td>
<td>68</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Alternative methods</td>
<td>70</td>
</tr>
<tr>
<td>4.5</td>
<td>Redo-Based Compensation</td>
<td>70</td>
</tr>
<tr>
<td>4.5.1</td>
<td>No-Undo/Redo Compensation</td>
<td>71</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Comparison of Undo/No-Redo and No-Undo/Redo Compensation</td>
<td>74</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Undo/Redo Compensation</td>
<td>75</td>
</tr>
<tr>
<td>4.6</td>
<td>Non-Strict Query Consistency</td>
<td>76</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Determining Transaction Dependencies</td>
<td>77</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Query Processing</td>
<td>80</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Levels of Query Consistency</td>
<td>82</td>
</tr>
<tr>
<td>4.7</td>
<td>Concluding Remarks</td>
<td>82</td>
</tr>
</tbody>
</table>

### 5  Query Execution  85

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Selection Queries</td>
<td>85</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Output Filtering</td>
<td>85</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Input Filtering</td>
<td>86</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Comparison of Output and Input Filtering</td>
<td>89</td>
</tr>
<tr>
<td>5.2</td>
<td>Aggregate Queries</td>
<td>90</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Scalar Aggregates</td>
<td>90</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Aggregate Functions</td>
<td>95</td>
</tr>
<tr>
<td>5.3</td>
<td>Combining Aggregation with Selection Predicates</td>
<td>97</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Selection Predicates for Scalar Aggregates</td>
<td>97</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Selection Predicates for Aggregate Functions</td>
<td>100</td>
</tr>
<tr>
<td>5.4</td>
<td>Join Queries</td>
<td>101</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Input or Output Compensation</td>
<td>101</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Join Algorithms</td>
<td>101</td>
</tr>
</tbody>
</table>
5.4.3 Sharing Memory Buffer Between Hash-Tables and Update-Tables .................................................. 103
5.5 Concluding Remarks .................................................. 104

6 Analytical Performance Model .................................................. 107
6.1 System Load and Capacity .................................................. 107
6.2 Performance Analysis of Producing a Snapshot of a Relation .................. 108
6.3 Discussion .................................................. 114

7 Performance Evaluation of Compensation-Based Query Processing .................................................. 115
7.1 Implementation Issues .................................................. 115
  7.1.1 UNDO Compensation .................................................. 115
  7.1.2 REDO Compensation .................................................. 117
  7.1.3 Two-Phase Locking .................................................. 117
  7.1.4 GO processing .................................................. 118
7.2 Simulation Model .................................................. 118
  7.2.1 System Model .................................................. 118
  7.2.2 Application Model .................................................. 120
7.3 Experiments and Results .................................................. 121
  7.3.1 Update-Intensive Experiment .................................................. 123
  7.3.2 Transaction-Mix Experiment .................................................. 131
  7.3.3 Space Optimization of the Update-Table .................................................. 136
7.4 Discussion .................................................. 137

III Distributed Query Processing .................................................. 141

8 Distributed Model .................................................. 143
  8.1 Transaction Controller .................................................. 144
  8.2 Database Kernel .................................................. 144
  8.3 Distributed Query Execution .................................................. 146

9 Distributed Algorithms .................................................. 147
  9.1 Global Transaction-Consistency in a Distributed Database System .................................................. 147
    9.1.1 Representing Transaction Dependencies .................................................. 149
    9.1.2 The Viewpoint of a Query .................................................. 151
  9.2 Centralized Computation of the Global AFTER set .................................................. 153
    9.2.1 Computation of the Global AFTER Set .................................................. 153
    9.2.2 Determining the Part of the DT Log Needed by the Query Controller .................................................. 154
    9.2.3 The Algorithm .................................................. 156
    9.2.4 Example .................................................. 157
    9.2.5 Proof of Correctness .................................................. 159
    9.2.6 Optimizations .................................................. 161
  9.3 Distributed Computation of the AFTER Set .................................................. 164
    9.3.1 Local Insertions in the AFTER Set .................................................. 165
    9.3.2 The Algorithm .................................................. 168
    9.3.3 Example .................................................. 170
    9.3.4 Proof of Correctness .................................................. 172
9.3.5 Optimizations .............................................. 173
9.4 Implementing WAIT Sets and AFTER Sets .................. 177
9.5 Forms of Query Consistency .................................. 179
9.6 Performance Analysis ........................................ 180
  9.6.1 Analysis of the Centralized Computation ............... 181
  9.6.2 Analysis of the Distributed Computation ............... 183
  9.6.3 Performance Comparison ............................... 184
9.7 Advantages of the Centralized Computation ............... 186
9.8 Related Work ................................................. 187
9.9 Concluding Remarks .......................................... 189

IV Conclusions and Future Work 191

10 Conclusions 193
  10.1 Summary ................................................... 193
  10.2 Main Contributions ....................................... 194

11 Directions for Future Work 195

A System Model for the Database Simulator 207
  A.1 CPU module ............................................... 207
  A.2 Disk Manager .............................................. 207
  A.3 Buffer Manager ........................................... 209
  A.4 Checkpoint Manager ....................................... 211
  A.5 Lock Manager ............................................. 211
  A.6 Log Manager ............................................... 211
  A.7 Transaction Manager ...................................... 212
  A.8 Query Manager ........................................... 213

B Low-Cost Compensation-Based Query Processing 215
Preface

This is a thesis submitted to the Norwegian University of Science and Technology (NTNU) for the doctoral degree “doktor ingeniør” (dr.ing.). The reported work has been carried out at the Department of Computer and Information Science (IDI) at NTNU under the supervision of Professor Arne Halaas. The work has been funded by The Norwegian Research Council.

The work was started in December 1992, and until August 1997, I was employed as a Research Fellow at IDI, NTNU. From November 1995 to December 1996, my position at NTNU was only half-time as I also worked at Telenor R&D. From August to December 1997, I was employed by Telenor R&D before joining ClustRa, a startup company making highly available database systems.

Part of the work presented in this thesis has earlier been presented at the International Workshop on Advanced Transaction Models and Architectures [Gro+96], the 15th British National Conference on Databases (BNCOD) [Gro+97], and the 24th International Conference on Very Large Databases (VLDB’98) [Gro+98].

Acknowledgments

I wish to thank my supervisor Professor Arne Halaas for inviting me to this study, for always being helpful, and for giving me freedom in pursuing this work.

Most of all, I wish to thank Professor Svein-Olaf Hvasshovd and Dr. Øystein Torbjørnsen for all their support during the last years. Their belief in the importance of my work and their encouragements have been very important in giving me confidence in my own work. From the time I approached them, asking for suggestions for a research topic, they have always been ready to discuss my work. Their insightful comments and advice on my work have improved this thesis a lot.

Many thanks also goes to my fellow Dr.Ing. student and former office mate Rune Humborstad. He has shown a great interest in my work, and has acted as a bouncing board for my ideas. His suggestions and feedback during our many discussions have been of invaluable importance to my work.

I also wish to thank Telenor R&D and Oddvar Rismes for employing me on the ClustRa project. This valuable experience gave me a much better understanding on the design and implementation of database management systems. I am also indebted to Telenor R&D for letting me spend some time on writing this thesis during the fall of 1997.

Thanks also to Svein Erik Bratsberg, Gunnar Brataas, Roger Midstraum, Bjørn Munch, and Maitrayi Sabaratnam who have all provided valuable comments to various drafts of this thesis.
Finally, I will express my gratitude to my wife Grethe who have made this ego-trip possible. Her love and support have meant a lot to me. During these years she have had to take much more than her share of responsibility for the daily life of our family. Both she and our three children, Magnhild, Ingeborg, and Hallvard, have had to sacrifice a lot for my doctoral degree. Hopefully, I can repay some of it in the years to come.
Part I

Introduction

This part presents the objective of this thesis, introduces basic concepts, and presents related work.

Chapter 1 The problem to be treated in this thesis is presented, and an outline of the thesis is given.

Chapter 2 Basic concepts related to transaction and query processing in database management systems is introduced.

Chapter 3 Related work on concurrent transaction and query processing is discussed.
Chapter 1

Concurrent Transaction and Query Processing

A current trend in database management is an increased demand for large and complex queries performed on data produced by business-critical real-time transactions. Introducing ad-hoc queries in such on-line transaction processing systems (OLTP) presents several unsolved problems [DeGr92]. One of the problems is that the concurrency control algorithm used in most commercial database systems, two-phase locking (2PL) [Esw+76], is not very efficient for concurrent transaction and query\textsuperscript{1} processing. This is because 2PL requires that queries acquire a \textit{shared} lock on each data item before accessing it, and no locks may be released before the query terminates. Any transaction that tries to update a data item that is locked by a query, will be delayed until the query finishes, waiting for an \textit{exclusive} lock. This will not be acceptable in business-critical OLTP systems with strict requirements to transaction response times.

In order to avoid that ad-hoc queries slow down real-time transactions, many organizations keep a separate copy of the database (e.g., a data warehouse) for query processing. Maintenance of the copy will have to be done during low-activity hours (e.g., each night) in order to avoid lock contention with on-line transactions and with queries [QuWi97]. Hence, a problem with this approach will be data staleness. Updates will not be visible to queries until the copy has been refreshed. In addition, extra resources are required both to store and administer the extra database copy and to perform the refresh operation. For large databases with high transaction rates, the duration of the refresh operation may exceed the available off-peak hours, and due to increased globalization, many organizations have hardly any off-peak hours at all.

The execution of a query is \textit{transaction-consistent} if the execution of the update transactions and the query is serializable [Ber+87]. That is, the execution of the concurrent transactions, as seen by the query, is equivalent to executing the transactions one-by-one. Executing a query using the 2PL protocol guarantees transaction-consistency, since 2PL only allows serializable executions. By compromising on transaction-consistency a query may release a lock at once it has read a

\textsuperscript{1}The term \textit{query} will in this thesis refer to a long-running read-only transaction. The term \textit{transaction} will be reserved for transactions that update the database.
2PL

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>READ $x$</td>
</tr>
<tr>
<td>$T_2$</td>
<td>READ $y$</td>
</tr>
<tr>
<td>$Q$</td>
<td>READ $x$</td>
</tr>
<tr>
<td>$T_2$</td>
<td>WRITE $y$</td>
</tr>
<tr>
<td>$T_2$</td>
<td>WRITE $z$</td>
</tr>
<tr>
<td>$Q$</td>
<td>READ $y$</td>
</tr>
<tr>
<td>$Q$</td>
<td>READ $z$</td>
</tr>
<tr>
<td>$T_1$</td>
<td>WRITE $x$</td>
</tr>
<tr>
<td>$T_1$</td>
<td>WRITE $z$</td>
</tr>
</tbody>
</table>

NON-2PL

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>READ $x$</td>
</tr>
<tr>
<td>$T_2$</td>
<td>READ $y$</td>
</tr>
<tr>
<td>$Q$</td>
<td>READ $z$</td>
</tr>
<tr>
<td>$T_1$</td>
<td>WRITE $x$</td>
</tr>
<tr>
<td>$T_2$</td>
<td>WRITE $z$</td>
</tr>
<tr>
<td>$Q$</td>
<td>READ $z$</td>
</tr>
</tbody>
</table>

\[ T_2 \xrightarrow{y,z} Q \xrightarrow{x,z} T_1 \]

\[ T_2 \xrightarrow{y,z} Q \xrightarrow{x,z} T_1 \]

Figure 1.1 Example of a serializable (left) and a non-serializable (right) execution of a query $Q$ and update transactions $T_1$ and $T_2$.

data item, or it may execute without setting any locks at all. Figure 1.1 shows two executions of a query and two concurrent transactions. The associated graphs show the serialization order of the executions with respect to the data items $x$, $y$, and $z$. In the execution on the left, the query is executed using 2PL. The execution of the query is transaction-consistent, and $Q$ serializes after $T_2$ but before $T_1$. $T_1$ is delayed until $Q$ has finished because $Q$ has acquired a shared lock on $x$. In the example on the right, $Q$ does not set any locks. Thus, $T_1$ is not delayed by $Q$. However, this leads to a non-serializable execution as seen by the cycle in the graph. That is, $Q$ sees some but not all updates made by $T_1$. In other words, transaction-consistency is compromised in order not to delay transactions.

Running queries without setting any locks is often referred to as GO processing [Pir90]. This way, queries can see uncommitted updates that may later be rolled back. Cursor stability locking, on the other hand, only lets a query see committed updates by requiring that a shared lock is held while a data item is actually being read [GrRe93]. Compromising on transaction-consistency will, however, not give satisfying accuracy for all applications.

In this thesis, a novel method for transaction-consistent execution of queries without delaying transactions is proposed. Using this method, queries will never hold locks for which transactions will have to wait. At the same time, the method ensures that all queries get a transaction-consistent view of the database. This is achieved by communicating the updates made by concurrent transactions to the query process. These updates will be undone and/or redone to make sure the final answer is transaction-consistent. In other words, transaction-consistency is achieved by having the query compensate for concurrent updates.

The thesis is divided into four parts. Chapter 2 of the first part, introduces basic concepts related to transaction and query processing in database management systems (DBMS), and Chapter 3 presents related work on concurrent transaction
and query processing.

The second part of the thesis presents methods for compensation-based query processing in a centralized DBMS. Chapter 4 presents several compensation-based algorithms, and the rest of the thesis is focused on the one of these algorithms that is based on undo/no-redo compensation. Chapter 5 discusses how the basic algorithm can be optimized for different types of queries. Chapter 6 presents an analytical performance model for this algorithm, while Chapter 7 presents the results of a simulation experiment evaluating the performance of compensation-based query processing.

The third part of the thesis presents algorithms for undo/no-redo compensation in a distributed DBMS. Chapter 8 presents the model of the distributed DBMS used in the presentation of the distributed algorithms in Chapter 9.

The last part of the thesis summarizes the major conclusions of the thesis in Chapter 10, and directions for future work is indicated in Chapter 11.
Concurrent Transaction and Query Processing
Chapter 2

Basic Concepts

This chapter presents the model of the database management systems (DBMS) used in this thesis. The model describes the assumptions made about the DBMS when presenting the algorithms for compensation-based query processing in later chapters. For presentation purposes, some of these assumptions are more restrictive than strictly necessary, and the query algorithms could in many cases be adopted to DBMSs where these assumptions are not valid.

2.1 DBMS Concepts

A database management system is a collection of hardware and software modules supporting database operations and transaction operations [Ber87]. For the first parts of this thesis, it is assumed that a centralized DBMS is used. That is, all the data items of the DBMS is stored at a single autonomous computer site [ElNa94]. A distributed DBMS is a collection of centralized DBMSs, a data communication network connecting these DBMSs, and a software component controlling the distribution of data items [Hvas96]. A model for a distributed DBMS will be presented in Chapter 8.

2.1.1 Data Model

A data model is a formalism for expressing data types, relationships, data integrity constraints, and operations on data [ElNa94]. In this thesis, it will be assumed that a DBMS implements the relational data model. Examples of other data models are hierarchical, object-oriented, and object-relational.

A database is a finite set of data items. Using a relational data model, the data items are organized into relations. A database will contain a schema for each relation. A relation schema $R^*$ is a finite set of attribute names, $\{A_1, A_2, \ldots, A_n\}$. To each attribute name $A_i$, $1 \leq i \leq n$, is assigned a set $D_i$ which is its domain. A relation $R$ is a finite set of tuples $\{t_1, t_2, \ldots, t_m\}$, mapping from $R^*$ to $D_1 \times D_2 \times \ldots \times D_n$. A primary key identifies a tuple uniquely within a relation. A primary key of relation $R$ is a subset of the attribute names of $R^*$.

In this thesis it is assumed that all DBMS internal tuple identification is based on primary keys. In order to support tuple access on primary key, it is assumed that
a relation is stored as a clustered B+-tree with all tuples located in leaf node blocks of fixed size. It is also assumed that a tuple is contained within a single block.

In many commercial DBMSs, database internal tuple identification is based on a DBMS generated key. For example, tuples may be identified by a block identifier and an index within the block. By using the database internal tuple identifiers where primary keys are mentioned in the algorithms of this thesis, the algorithms will be applicable to DBMSs that rely on internal keys.

2.1.2 Database Operations

Declarative relational operations expressed in database languages usually based on tuple relational calculus ([Maie83]) are mapped by the DBMS, often via relational algebra, to tuple-oriented database operations. Where not otherwise stated, this thesis assumes that a DBMS supports the following tuple-oriented database operations: \textbf{insert}, \textbf{delete}, \textbf{read}, \textbf{write}. In addition, it will be discussed how some algorithms may be extended to handle delta operations (i.e., \textbf{increment} and \textbf{decrement}). The term \textbf{update} operations will be used for database operations that change the content of the database (i.e., \textbf{insert}, \textbf{delete}, \textbf{write}).

2.2 Transaction Model

A \textit{database transaction} is an atomic unit of work that performs database operations on the data items of a database. When a transaction has executed all its database operations, the updates introduced by the transaction can be \textit{committed}, that is, permanently applied to the database. If a failure occurs during the execution of a transaction, the transaction is \textit{aborted}, and any updates that the transaction has made to the database must be \textit{undone} in order to fulfill the atomicity property.

A transaction should possess the following properties [GrRe93, EI94]:

\textbf{Atomicity:} All or none of the updates performed by a transaction will be reflected in the database.

\textbf{Consistency:} The execution of a transaction takes the database from one consistent state to another consistent state.

\textbf{Isolation:} It should appear to each transaction \( T \), that other transactions executed either before \( T \) or after \( T \), but not both.

\textbf{Durability:} Updates of committed transactions should never be lost because of subsequent failures.

2.2.1 Concurrency Control

In a multiuser database system, several transactions will be active simultaneously. If the transactions are executed in an uncontrolled manner several problems can occur. One example is the lost update problem [EI94], where a transaction updates a data item without being aware of the previous update made to this item. To avoid such problems, concurrency control protocols are used to ensure that concurrent execution of transactions have the same effect on the database as some \textit{serial} (i.e., one after another) execution of the transactions. An execution is called \textit{serializable}
### 2.2 Transaction Model

<table>
<thead>
<tr>
<th>SERIALIZABLE</th>
<th>NON-SERIALIZABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$ READ $x$</td>
<td>$T_1$ READ $x$</td>
</tr>
<tr>
<td>$T_2$ READ $x$</td>
<td>$T_2$ READ $x$</td>
</tr>
<tr>
<td>$T_1$ WRITE $x$</td>
<td>$T_3$ READ $x$</td>
</tr>
<tr>
<td>$T_3$ READ $x$</td>
<td>$T_1$ WRITE $x$</td>
</tr>
<tr>
<td>$T_2$ WRITE $y$</td>
<td>$T_2$ WRITE $y$</td>
</tr>
<tr>
<td>$T_1$ WRITE $y$</td>
<td>$T_1$ WRITE $y$</td>
</tr>
<tr>
<td>$T_3$ WRITE $x$</td>
<td>$T_3$ WRITE $x$</td>
</tr>
</tbody>
</table>

![Dependency graphs for two different executions of three transactions.](image)

if it is equivalent to a serial execution. Serializable execution is necessary to satisfy the isolation property of transactions.

According to the serializability theorem [Ber+87], an execution is serializable if its dependency graph is acyclic. A transaction $T$ is said to depend on another transaction $T'$, if $T$ reads or updates a data item previously updated by $T'$, or if $T$ updates a data item previously read by $T'$. Note that reading a data item previously read by $T'$, will not introduce a dependency since any order of executing read operations will give the same result. The dependency graph is a directed graph were the nodes are transactions and the arcs are transaction dependencies. Figure 2.1 shows the dependency graph for two different executions of three transactions $T_1$, $T_2$, and $T_3$. The execution on the left is serializable since the dependency graph is acyclic. The execution on the right is not serializable since the dependency graph is cyclic. The execution on the right illustrates the lost update problem. The update made by transaction $T_1$ to data item $x$ is lost since $T_3$ bases its update of $x$ on an older version of $x$.

### Two-Phase Locking

The most used concurrency control protocol for achieving serializable execution of transactions is the two-phase locking (2PL) protocol. Using this protocol, a transaction will have to acquire a lock on a data item before accessing it. The protocol is called two-phased because a transaction may not acquire new locks after releasing a lock [Ber+87]. Associated with each data item are two types of locks: a shared lock which allows several transactions to access the same item, and an exclusive lock which ensures exclusive access to an item. **Read** operations requires a shared lock while update operations (i.e., **insert**, **delete**, and **write**) requires an exclusive lock. Shared and exclusive locks are incompatible. In other words, a transaction will not get an exclusive lock on an item if another transaction holds a shared lock, or vice versa. In that case, the transaction will have to wait for all transactions holding shared locks on the item to release their locks before it can continue its execution.

A locking model is called **semantically rich** if special lock types for delta opera-
Table 2.1 The compatibility matrix for the semantically rich locking model.

<table>
<thead>
<tr>
<th></th>
<th>Shared</th>
<th>Exclusive</th>
<th>Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared</td>
<td>y</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>Exclusive</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>Delta</td>
<td>n</td>
<td>n</td>
<td>y</td>
</tr>
</tbody>
</table>

...
system failures. A transaction failure occurs when a partially completed transaction is aborted. All operations made by the aborted transactions is then undone in order to fulfill the atomicity property.

When a system failure occurs, data stored in volatile storage will be lost. In order to fulfill the durability property, all updates made by committed transactions prior to the failure will have to be recovered. In addition, transactions that were in progress when the failure occurred, must be aborted.

Buffer Management

In order to support efficient management of the main-memory database buffer, the recovery method should support the STEAL/NO-FORCE properties. The STEAL property implies that uncommitted updates may be written to stable storage in order to make room for other pages. Uncommitted updates reflected in stable storage will have to be undone during recovery. The NO-FORCE property implies that updates will not have to be reflected in stable storage before a transaction commits. Such updates will have to be undone during recovery. In other words, the STEAL/NO-FORCE properties require that undo/redo recovery is used [Ber87].

Logging

In order for the DBMS to be able to recover from transaction and system failures, it is assumed that all updates are recorded in a database internal log. Tuple-oriented logging is assumed [Hvas96]. That is, for each update operation, a log record representing this update is inserted into the log. In addition, commit and abort operations are logged in order to be able to determine which transactions had terminated before a system failure occurred.

All log records will contain the following fields [Hvas96]:

- **Log record identifier**
  A unique identifier, or key, called the log sequence number (LSN) [GrRe93]. In this thesis, LSNs are assumed to be monotonously increasing.

- **Operation identifier**
  Indicates the type of operation represented by this log record. Possible iden-
tifiers are insert, delete, write, commit, and abort. If delta operations are supported, delta will also be a valid identifier.

- **Transaction identifier**
  The identifier of the transaction that created this log record.

- **Previous log record identifier of transaction**
  The LSN of the previous log record for the same transaction. This way, the log records of a transaction are chained together. This transaction log chain will be followed when undoing the effects of a transaction.

A log record for a write operation contains the following fields in addition to those listed above:

- **Tuple identifier**
The relation identifier and the primary key of the tuple that was updated by the write operation.

- **Before-image**
The attribute values of the tuple before the operation was performed. Needed in order to be able to undo the operation.

- **After-image**
The attribute values of the tuple after the operation has been performed. Needed in order to be able to redo the operation.

Log records for insert and delete operations are similar to the above except that they do not contain before-images or after-images, respectively. Log records for delta operations will contain delta images instead (i.e., the operands of the delta operation). The use of before-images and after-images for recovery requires that the concurrency control protocol enforce strict execution of transactions [Ber87]. That is, a tuple may not be updated until the transaction that made the previous update has terminated.

Both partial and complete tuple logging may be used. However, where not otherwise stated, partial tuple logging can be assumed. That is, a log record will only contain before-images and after-images of attributes that are changed by the update operation [Hvas96]. The efficiency of some of the proposed algorithms may be improved by using complete tuple logging.

In order to be able to redo updates of committed transactions and undo updates of aborted transactions after system failures, write ahead logging (WAL) will be used [Gra81]. That is, before a transaction is allowed to commit, all its log records must have been written to stable storage, and before a block is written to stable storage, all log records for the tuples of this block must have been written to stable storage.

### Compensation-Oriented Logging

This thesis assumes that compensation-oriented logging is applied in order to determine which operations have been undone [Moh92a]. That is, a compensation log record (CLR) is created when performing an undo operation. Using this scheme, log records representing updates are often called non-CLRs.
2.2 Transaction Model

It is assumed that each block contains a state identifier that tells which operations are reflected in the block. The state identifier will be the LSN of the log record for the most recent operation performed on a tuple of this block. All operations of log records with a smaller LSN are reflected, while no operations of log records with a larger LSN will be reflected. During recovery, the state identifiers will be used to determine whether update operations, represented by non-CLRs, and undo operations, represented by non-CLRs will have to be redone. State identifiers may also be associated with tuples [Hvas96].

Checkpointing

In order to reduce recovery time and to enable efficient garbage-collection of the redo-information contained in log records, fuzzy checkpointing is applied [GrRe93]. Fuzzy checkpointing guarantees that all updates made before the penultimate checkpoint is reflected in stable storage. This way, no log records created prior to the penultimate checkpoint will ever have to be redone, and the after-image of these log records may be garbage-collected. The before-image of a log record may be garbage-collected when the associated transaction has terminated (committed or aborted).
Chapter 3

Related Work

This chapter presents previous work addressing the problem of running complex queries in database systems without affecting the performance of concurrent transactions. As the rest of this thesis, this chapter concentrates on methods for reducing data contention between queries and transactions. When queries and transactions are to be executed in the same system, resource contention must also be considered [DeGr92]. For a discussion on resource contention topics, the interested reader is referred to [Bro+94, Bro+96].

The first two sections of this chapter briefly discuss two alternative strategies to transaction-consistent execution of queries in a transaction processing system; inconsistent query execution and database snapshots. Then, two approaches to reducing data contention while still guaranteeing transaction-consistency are presented. In Section 3.3, several approaches based on maintaining multiple versions of the data items are discussed. These approaches, often called transient versioning, are based on that queries may see a transaction-consistent snapshot of the database by accessing possibly old versions of data items. In Section 3.4, an approach called compensation-based query processing is presented. Using this method, queries achieve transaction-consistency by compensating for updates by concurrent transactions. While the first sections of this chapter mainly discusses concurrent query and transaction processing in a centralized DBMS, Section 3.5 presents work relevant to distributed query processing. This section discusses how checkpoints are synchronized between the nodes of a distributed DBMS in order to ensure transaction-consistency. Similar approaches may be used for transaction-consistent execution of distributed queries.

3.1 Degrees of Consistency

One approach to avoid data contention, is to compromise on transaction-consistency. For systems using two-phase locking, Gray has identified three different degrees of consistency [Gra+76]. For degree 1 consistency, also called GO processing [Pir+90], queries may be run without setting any locks. It follows that GO processing allows queries to read uncommitted (dirty) data. Degree 2 consistency, which is also called cursor stability, is achieved if queries read only committed data. This can be achieved by acquiring a read lock on each data item before it is read. The lock may be released
before the query reads the next data item. Mohan has shown that by exploiting the state identifiers of data pages, cursor stability can be achieved without actually acquiring locks on all data items [Moh90b].

Another approach to reduced transaction-consistency is \textit{epsilon serializability} (ESR) [PuLe91, Wu+92]. ESR allows dirty reads, but the amount of inconsistency can be bounded and automatically controlled by the system. That is, inconsistent execution of queries is allowed as long as it is not too far from transaction-consistent.

Degree 3 consistency implies transaction-consistent query execution. It has also been called \textit{repeatable reads} since it guarantees that a query will see the same value if it re-reads a data item. Traditionally, degree 3 consistency is achieved by using strict 2PL. As discussed in Chapter 1, long queries will block other transactions for a long time when using 2PL. Salem et al. have proposed an extension to 2PL called altruistic locking in order improve the performance at the presence of long-lived transactions [Sal+94]. Using this approach, a transaction may donate a lock to other transactions when it has finished accessing the corresponding data item. Special locking rules, increasing the overhead of locking, are needed in order to guarantee transaction-consistency. Altruistic locking does not solve the problem of long-lived transactions, but it reduces the likelihood of a transaction being blocked by concurrent long-lived transactions.

Hassan and Weikum have shown how a multi-level transaction model can be used to achieve higher concurrency between transactions and queries [HaWe97]. Multi-level transactions are a variant of nested transactions where the subtransactions at different levels represent executions of operations at different levels of abstraction [WeIK91]. Concurrency can be increased by only holding low-level locks for the duration of subtransactions. Transaction-consistency is achieved by following the conflict definition of high-level operations as given by the semantics of these operations.

### 3.2 Database Snapshots

Traditionally, the problem of proper handling of both transaction and query workloads has required the maintenance of two different databases on two different systems [Pir+90]. The transaction workload is run against the up-to-date database, while the second database is an extracted version of the first and is used for query processing.

Adía and Lindsay introduced the concept of snapshots [AdLi80]. A snapshot represents a transaction-consistent state of the database as of the time the snapshot is taken. Snapshots will be refreshed or made up-to-date periodically. A snapshot may be defined as a general query, and the refresh operation may be obtained by executing the query [Rin87].

The refresh operation may be implemented either as a \textit{full refresh} or a \textit{differential refresh} [Kär87]. When applying full refresh, a new snapshot is created periodically. For differential refresh, the snapshot may be updated by maintaining a log of the modifications to the database. When a refresh occurs, this log will be applied to the snapshot. Lindsay et al. have proposed an alternative to logging that is based on extending each tuple of the base tables with attributes that help identify which tuples need to be refreshed [Lin+80]. An advantage of this approach is that multiple changes to the same tuple will be aggregated and sent as a single modification to
the snapshot database.

A snapshot resembles a materialized view [Bla+86], but there are some differences [Risn87]:

- Materialized views represent the current database state, whereas a snapshot represents the state of the database at the last snapshot refresh.
- Unlike a materialized view, a snapshot may continue to exist after the deletion of the base relations.

In other words, database snapshots are more loosely coupled to the master copy of base relations than materialized views. These differences are fundamental since base relations should be able to evolve independently of any snapshots. Still, much of the work on maintenance of materialized views ([GuMu95]) are also applicable to differential snapshot refresh.

There are several disadvantages of having to maintain a separate database copy for query processing. In addition to the extra resources needed to maintain two databases, disk storage requirements are doubled. Also, the snapshot database will often be out-of-date since the refresh operation can only be performed only during low-activity hours in order not to degrade the performance of the master database. In addition, as \(24 \times \text{hours} \times \% \text{days} \) availability are required by an increasing number of database systems, many systems may not have any low-activity hours.

In addition to avoiding data contention between transactions and queries, database snapshots have the advantage that the architecture, database design, access paths, and resource management of the snapshot database system, can be tailored for the execution of complex queries. This is, in addition to the possibility of integrating data from several sources into a single database, the reason for the success of data warehouses [Inmo96]. Due to its popularity, much research have been performed in the area of data warehousing over the last years. Papers have been presented on how to efficiently maintain the views in a data warehouse [LaGa96, Qua+96, Agr+97, Huyn97], how to ensure consistency of views [Zhu+95, Zhu+96, Zhu+97], how to decide which views to materialize in the data warehouse [ThSe97, Yan+97], and the use of pre-aggregation ([Gra+97, Gup+95, Mum+97]) and bitmap indexes ([O’Qu97, WuBu98]) to speed up query processing.

The work presented in this thesis will not be an alternative to data warehousing when query execution requires the integration of heterogeneous data sources or a database design specially tailored for queries. In other cases, being able to run queries on databases used for transaction processing will reduce the need for a data warehouse. Note also that for the maintenance of some types of materialized views, the modification log sent from the source database may have to be supplemented by queries to the source databases in order to determine the actual changes to the view [Zhu+95]. In other words, data warehousing does not entirely eliminate the need to run queries at the source databases.

### 3.3 Transient Versioning

Transient versioning is a variant of multiversion concurrency control where prior versions of data items are maintained temporarily to increase concurrency [Pir+90]. Queries are allowed to see a slightly outdated but transaction-consistent database.
snapshot. Using transient versioning, the storage and management costs are potentially lower than when extracting entire database snapshots, and many logical snapshots may be maintained simultaneously so that each query can be given a recent view.

In multiversion concurrency control algorithms, each update to a data item produces a new version of the data item. The benefit of multiple versions is that the degree of concurrency can be improved [Ber*87]. Read-only transactions may be executed unhindered by accessing a set of old versions that represent a transaction-consistent database state. In addition, out-of-order read requests can be serviced by reading an older version of a data item. If the necessary prior versions of data items are available, this means that a query will not need to block concurrent update transactions in order to serialize before this transaction. Instead, the query could access prior versions of the data items updated by this transaction.

The first multiversion concurrency control algorithm was presented by Reed [Reed78]. In this scheme, prior versions are kept forever, and read-only transactions are allowed to ask for an arbitrary time-slice. As the concurrency control is based on timestamp ordering [Ber*87], queries may cause transactions to abort. In addition, queries have to update timestamps associated with data items. Thus, read operations are turned into writes. This will normally severely increase the I/O load of a system, and in a distributed database system, the 2PC protocol will also be required for read-only transactions.

The first works on transient versioning were based on the observation that the before-values of updates logged for transaction and system recovery could be used to improve concurrency. Bayer et al. ([Bay*80]) proposed a two-version concurrency control approach based on a system using shadow paging ([Lori77]) to support recovery. A dependency graph is used to determine which versions to read and whether a transaction will have to be rolled back in order to preserve consistency. In this scheme, read-only transactions are never rolled back. Another two-version approach has been proposed by Stearns and Rosenkrantz [StRo81]. Recently, Quass and Widom have presented a two-version algorithm for view maintenance in data warehouses [QuWi97]. Since only two versions of a data item is allowed, a transaction will be blocked if both versions are needed by readers. Because of this, most later approaches allow for several prior versions in order to eliminate data contention.

Below, several approaches for transient versioning are presented. The presentation is limited to approaches where the execution of queries does not severely affect concurrent update transactions. This rules out two-version approaches [Bay*80, StRo81], approaches that turns reads into writes [Reed78, Wel87], and optimistic approaches based on certification [LaWi84]. In all approaches presented below, only read-only transactions are allowed to access prior versions. Updating transactions will always read the latest version of a data item, and a copy-on-write mechanism is used for updates.

3.3.1 Chan et al., 1982

Chan et al. CCA proposed an integrated concurrency control and recovery scheme based on transient versioning [Cha*82]. Updating transactions are executed using strict 2PL with the modification that updates causes the prior version to be written to a version pool. Locking and versioning is employed at page level, and prior
3.3 Transient Versioning

versions are used both to avoid data contention between queries and transactions and in order to roll back transactions in the event of failures. The algorithm was implemented in a commercial system developed at Computer Corporation of America (CCA), and a similar scheme was previously used by Prime’s Database Management System [DuBo82].

Queries do not set or wait for any locks. Instead, a query obtains a transaction-consistent view by reading the latest version of a data item created by a transaction that committed before the start of the query. This way, queries do not block transactions, and vice versa. In order to be able to access correct versions, a list of completed transactions (CTL) is maintained. A query is assigned the current CTL when it is started, and will access the latest versions created by transactions in its CTL. The CTL can be efficiently implemented as a bitmap indexed by transaction ID. A transaction ID associated with the CTL represents the index of the first entry of the bitmap.

The versions are divided into the main segment and the version pool. The main segment contains the current versions of all data items, while the version pool contains prior versions. The version pool is implemented as a circular buffer, and for each data item, versions are chained in reverse chronological order. Thus, a query will find the correct version by traversing the version chain until it finds a version created by a transaction in its CTL.

The organization of the version pool has three attractive properties [BoCa94]:

- In-place updates maintains clustering of current versions.
- The version pool is written sequentially (i.e., similar to log writes).
- Garbage collection of prior versions is relative straightforward.

Figure 3.1 shows how the version pool is organized to support efficient garbage-collection. Version pool entries between reader-first and last contain versions that may have to be read by ongoing queries. Entries between update-first and last
Related Work

contain before-images that may be needed to roll back current transactions. The two pointers, reader-first and update-first are updated when the oldest query or transaction, respectively, terminates. Pages beyond last in the circular buffer can be freely reused until reader-first is reached. If there is no free pages, transactions are either blocked, or the page allocation is allowed to advance beyond reader-first. In the latter case, queries may have to be aborted since versions needed by a query may be missing. Pages in the active update range could never be discarded since they are needed for transaction recovery. The database system will stop accepting new transactions when the number of pages not in the active update range falls below a given threshold.

Prior versions are also used for recovery. To recover from a transaction failure, pages in the version pool are read back into the main segment. A undo/no-redo strategy is used for system recovery. To avoid redo, dirty pages must be forced to disk at commit [Ber*87]. The completed transaction list must be maintained in stable storage in order to be able identify the set of uncompleted transactions that have to be rolled back during system recovery.

A drawback of the version pool is the additional cost of copying pages before they are updated. In addition, the ordinary sequential I/O pattern of a query may become disrupted by random accesses to the version pool. Since sequential garbage-collection is performed, space requirements for the version pool may be very high in order to support long queries.

Transient versioning algorithms like the above have been called multiversion two-phase locking (MV2PL) since transactions execute using strict 2PL. However, any other concurrency control algorithm that only allows rigorous schedules ([Bre*91]) could be used. That is, serialization order must be equivalent to commit order.

3.3.2 Agrawal and Sengupta, 1989

Agrawal and Sengupta have presented a scheme for separating the version control and the concurrency control components of transient versioning [AgSe89]. The proposed multiversion protocol can be integrated with any conflict-based concurrency control method.

Transactions are assigned a transaction number that is a unique number that represents their position in the serialization order. When a transaction creates a new version, this version is stamped with the transaction number of the updater. At startup, a query is assigned a transaction number such that all transactions with a smaller version number have terminated. Queries will read the version with the highest version number less than their own transaction number. No other synchronization between transactions and queries is needed. Hence, queries will never delay or abort transactions.

The main task when integrating this multiversion protocol with a concurrency control method is to decide how to assign transaction numbers to transactions. In [AgSe89] it is shown how this can be done for multiversion timestamp ordering and two-phase locking. For timestamp ordering, transaction number assignment is straightforward since the timestamp assigned to transactions at startup, can also be used as transaction number. For two-phase locking, the serial order of transactions correspond to their lock points.\footnote{A lock point is any moment at which the transaction owns all of its locks.} Hence, a transaction could not be assigned its
3.3 Transient Versioning

transaction number until it has reached its lock point. When strict 2PL is used, the commit point can be used instead of the lock point.

For 2PL, deferred updates are assumed in order to be able to include the transaction number of the updater in each version. Deferred updates will significantly increase commit-time processing, especially if the STEAL buffer management policy is supported. However, deferred updates can be avoided by using a transaction ID assigned at startup instead of the transaction number [BoCa92b]. Then, a separately maintained list can be used to map from transaction ID’s to transaction numbers. This way, the only difference between this method applying strict 2PL and the method by Chan et al., except from recovery aspects, are the use a common transaction list with commit timestamps instead of assigning a CTL to each query.

No description has been given about the granularity of locking, chaining of versions, and garbage-collection of old versions in this method.

3.3.3 Bober and Carey, 1992

Bober and Carey have proposed some refinements to the method by Chan et al. in order to make MV2PL more efficient for transaction processing [BoCa92b]. In another paper, they proposed an alternative transient versioning method called multiversion query locking (MVQL) [BoCa92a]. In a third work, different schemes for indexing of versions was presented [BoCa94]. Below, the main results of these works are presented.

MV2PL Refinements

To allow for greater concurrency, Bober and Carey assumed that tuple-level versioning and locking were used instead of page-level versioning. In addition, recovery was separated from versioning by assuming traditional write-ahead logging. That way, the force requirement for buffer pages at commit was avoided. Instead of maintaining CTL’s, commit timestamps were assigned to transactions and startup timestamps to queries as proposed by Agrawal and Sengupta [AgSe89].

An advantage of tuple level versioning is that it allows for on-page caching. That is, a portion of each data page is used for storing prior versions. This way, the number of accesses to the version pool is reduced. If a page cache is full when a tuple is updated, prior versions in the cache will be garbage-collected if they are not needed by any current query. The garbage-collection of the on-page cache does not have to be done in chronological order.

When writing prior versions to the version pool, either only the oldest entry or all entries in the cache could be written. The write-all policy has the advantage that the number of I/O operations are reduced since it will not be necessary to perform a write operation each time a cache entry is replaced. In addition, versions from the same data page are clustered together in the version pool. On the other hand, the write-one policy will minimize the size of the version pool since prior versions may be garbage-collected before they are written to the version pool.

Bober and Carey also introduced the concept of view sharing, in which several queries are using the same transaction-consistent view. This is achieved by letting new queries reuse the startup timestamp of currently executing queries. View sharing will potentially reduce the number of concurrent versions since a prior version is unnecessary if no current query has a startup timestamp between the version number.
Table 3.1 Forms of query consistency [BoCa92a]

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strict</td>
<td>Each query sees a serial order of transactions that is consistent with their commit order.</td>
</tr>
<tr>
<td>Strong</td>
<td>All queries see the same serial order of transactions. (Queries are serializable with respect to all transactions.)</td>
</tr>
<tr>
<td>Weak</td>
<td>Each individual query sees a serial order of update transactions. (Not necessarily the same as other concurrent queries.)</td>
</tr>
<tr>
<td>Update</td>
<td>Each query serializes with all transactions that it sees. (A transaction that the query sees may be read-write dependent on transactions the query does not see.)</td>
</tr>
</tbody>
</table>

of this version and the next version for the same tuple. Thus, reduction in storage requirements is achieved at the expense of letting queries see a slightly out-of-date version of the database.

Bober and Carey also presented a simulation study that showed significantly improvement in query throughput when using on-page caching. When page caches were large enough to keep version pool accesses to a minimum, the query throughput was up to 50 times higher than when using no page cache. The simulation study also showed that, generally, the write-all policy performed better than the write-one policy.

**Multiversion Query Locking**

MVQL tries to make versioning more affordable by allowing weaker forms of transaction-consistency than MV2PL. This is achieved by allowing queries to read more recent versions of objects. This will improve query performance since:

- Accesses can be made to the page cache instead of the version pool.
- Accesses to the version pool is made cheaper since versions are chained in reverse chronological order.
- Old versions may be garbage-collected earlier.

Bober and Carey defines four forms of query consistency (Table 3.1). In all cases the execution of transactions are guaranteed to be serializable by using strict 2PL. MV2PL provides strict query consistency since a query sees the effects of all transactions that committed before it.

Strong and weak consistency was originally introduced by Garcia-Molina and Wiederhold to evaluate schemes for consistency of read-only transactions in a distributed system [GaWi82]. Strong consistency assures serializable execution of read-only transactions, while weak consistency only guarantees that each individual read-only transaction is serializable with respect to updating transactions. That is, the dependency graph may contain multiple-query cycles, but no single-query cycle. In his thesis, Bober also introduced the concept of consistency groups [Bobe93]. For

---

2The newest prior version could not be garbage-collected until the version number of the current version is known. That is, when the creator of the current version has committed.
3.3 Transient Versioning

![Diagram](image)

**Figure 3.2** A dependency graph satisfying update consistency.

example, queries in a strong consistency group serialize with each other, but not necessarily with queries outside the group.

Update consistency allows single-query cycles in the dependency graph if they can be broken by removing a read-write dependency between two transactions. This way, a query serializes with the set of update transactions that produced the values that are directly or indirectly seen by the query. Thus, update consistency guarantees that the query sees a transaction-consistent database even though it may not serialize with the complete set of update transactions. Figure 3.2 shows the dependency graph for the following example of an execution that is update consistent:

\[
\begin{align*}
& r_1[a] \ r_Q[\ell] \ w_1[\ell] \ w_2[a] \ r_Q[a] \\
\end{align*}
\]

Weak query consistency is not achieved since the dependency graph contains a single-query cycle consisting of \(Q, T_1,\) and \(T_2\). However, this cycle can be broken by removing the read-write dependency edge between \(T_1\) and \(T_2\). In other words, update consistency is achieved since \(Q\) does not see any effects of \(T_1\).

For each query, MVQL divides the set of transactions into two subsets, the BEFORE set, which contains all transactions that serializes before the query, and the AFTER set, which contains all transactions that serialize after the query. The query will see a consistent state by always reading the latest version created by a transaction in the BEFORE set. In order to achieve this, the AFTER set of each query will be represented explicitly. The AFTER set can be implemented as a bitmap similar to the implementation of CTL in the method by Chan et al.

A transaction will be inserted in the AFTER set only when it is necessary to prevent a violation of the desired form of consistency. Bober and Carey have described a set of conditions for inserting transactions into the AFTER set in order to enforce the different forms of consistency. In order to check for these conditions, queries are required to obtain a so-called read-only lock on all objects they read. Read-only locks are non-conflicting and are released by queries when they finish.

Bober and Carey have also performed simulation experiments that show improved performance of MVQL compared to MV2PL [BoCa92a].

**Multiversion Indexing**

Commercial database systems like Oracle and DEC’s Rdb, treat index nodes like data tuples with respect to transient versioning. While this approach enables retrieval of the desired prior version of a data item, it is not compatible with non-2PL.

---

3 Actually, in order to avoid phantom problems the read-only lock should really have been a key-range lock.
B+ tree concurrency algorithms [BaSc77]. Bober and Carey discusses the following multiversion indexing approaches that differ as to where version selection information is located:

**Chaining:** An index refer the most recent version of a tuple. The prior versions are chained behind the current version in reverse chronological order, as in the scheme presented above.

**Data Page Version Selection:** Differs from chaining in that for each tuple a data page contains a version selection table (VST) that refers all versions of the tuple. A VST contains the timestamps of all versions so that a query can decide which version to access.

**Primary Index Version Selection:** Each entry in the primary index contains a VST with references to all versions the tuple.

**All Index Version Selection:** Also secondary indexes contains a VST with references to all versions with a matching secondary key.

Simulation experiments performed by Bober and Carey showed that a high buffer hit rate on index pages were the most important performance factor [BoCa97]. Thus, in order to minimize the size of indexes, version selection information should not be stored in indexes. Bober and Carey recommended Data Page Version Selection since its I/O cost were somewhat lower than for Chaining for long queries.

### 3.3.4 Mohan et al., 1992

The transient versioning schemes presented so far, with the exception of the concept of view sharing introduced by Bober and Carey [BoCa92b], let each query access a consistent snapshot defined by its arrival time. In other words, there is no restriction on the number of versions maintained for each data item. Mohan et al. have proposed a method that gives control over the number of concurrent versions by dividing the time axis into version periods [Moh+92b]. Each data item has at most one version in each prior version period. In the current version period there may be two versions, the last committed version and the current version.

Transactions execute in the current version period using strict 2PL. At startup, queries will be assigned to the previous current version period (Figure 3.3). That is, a query will only see the effects of transactions that committed before the beginning of the current version period at its time of arrival. As for view sharing, currency is sacrificed for reduction in storage requirement for prior versions.

References to all versions of a data item is stored in a data structure associated with each key located in the primary index. This is equivalent to the Primary Index Version Selection Scheme discussed by Bober and Carey. The number of versions allowed for each data item is limited by the size of this data structure. If the data structure may refer to maximum $N$ versions, $N - 2$ concurrent logical snapshots will be available for queries. Periodically the system may switch from an older to a newer version. However, version $v + 1$ could not be introduced until all queries using version period $v - (N - 2)$ has terminated. Thus, long running queries may delay the start of a new version period, something that may cause new queries to see an obsolete version of the database.
3.3 Transient Versioning

\[ v - 2 \quad v - 1 \quad v \quad \ldots \]

\[ T_1 \quad T_2 \quad T_3 \quad T_4 \]

\[ Q_1 \quad Q_2 \quad Q_3 \quad Q_4 \]

Figure 3.3 Association of transactions and queries to version periods [Moh*92b].

In order to be able to decide which version period a version belongs to, the transaction ID of the creator of each version is stored together with the reference in the primary key index. For each version period, the system will maintain a list of all transactions that were not committed at the end of the version period. A query will access the latest version that is not created by a transaction that is mentioned in the list of the query’s version period. When a new transaction starts, it has to be added to the transaction list of all version periods. If the number of concurrent version periods is high, maintaining these lists may become expensive.

No performance evaluation has been presented for this method.

3.3.5 Wu et al., 1993

Wu et al. have presented another method, called dynamic finite versioning (DFV), for limiting the degree of versioning. In the basic version of DFV, two logical transaction-consistent snapshots are maintained for query processing. New queries entering the system will access the most recent of the two snapshots. When the last query accessing one of the snapshot finishes, a query-snapshot advance is performed. That is, a new snapshot is introduced, and the snapshot lacking active queries are not maintained anymore.

DFV assumes versioning at page level. When an updated page is written back to disk, three physically continuous pages are allocated if only a single physical version exists for the page. The update is written into one of the three newly created pages in order to avoid the cost of copying the old page. In order to preserve sequentiality of the database, the allocation of new pages should be done on the same or on a nearby disk track. This can be achieved by leaving some free space on each track for version management.

The four pages will be used to store the versions of each of the query snapshots in addition to the last committed and the current version of the page. Each page version is tagged with the transaction ID of its creator and a timestamp representing the time the update occurred. When a new snapshot is introduced, the current list of active transactions is copied. A query will normally access the page version that has the largest timestamp smaller than the startup time of its snapshot. However, if this version is tagged by a transaction ID found in the active transaction list of the snapshot, the previous version is accessed.

The DFV algorithm can be generalized to more than two concurrent query snapshots by using more than four versions. Since any snapshot that has no queries accessing it could be released, DFV will reduce storage overhead and query obsoles-
ence compared to the method by Mohan et al. which only allowed for the oldest snapshot to be released. In addition, the DFV algorithm requires no updating of the lists of active transactions.

A disadvantage of DFV is that it assumes page level versioning. This restricts transaction concurrency by requiring page granularity concurrency control. In addition, the required storage per version will be larger than for tuple level versioning. However, it should be straightforward to adapt DFV to tuple level versioning, using on-page caching for prior tuple versions. Then, DFV can be viewed as a more systematic organization of the concept of view sharing introduced by Bober and Carey [BoCa92b].

### 3.3.6 Transient Versioning in Commercial DBMS

Transient versioning has been implemented in a few commercial database systems. As mentioned above, the combined recovery and concurrency control method proposed by Chan et al. has been implemented by CCA [Cha+82], and a similar method was earlier implemented by Prime [DuBo82].

**DEC Rdb**

DEC implemented tuple-level transient versioning in their Rdb DBMS [RaRe91]. Their approach uses versioning only for queries and database backup. Recovery is supported by write-ahead logging. Prior versions are chained together in a version pool, and each version is tagged with the transaction ID of its creator. Versioning may be turned off when no queries are active. In that case, new queries will have to wait until all transactions that were active before versioning were turned back on again, have completed.

**Oracle DBMS**

Oracle has also implemented transient versioning in their DBMS [Bri+97]. Prior versions are stored in a rollback segment which is also used for transaction recovery. Write-ahead logging of redo information for updates to the rollback segment is performed in order to be able to recover the rollback segment at system recovery.

Each transaction is assigned a monotonically increasing system change number (SCN) at commit time. Each page is tagged with the transaction ID of the last transaction that updated it. By consulting a transaction table, it is possible to find the SCN of the page. Queries are assigned the currently highest SCN at startup. When a query requests to read a page, the SCN of the page is checked. If the SCN of the query is smaller than the SCN of the page, the requested version of the page is reconstructed by using prior versions stored in the rollback segment. This reconstructed version, called a clone, resides in buffer until it is replaced by the ordinary buffer replacement activity. During this time, the clone can be reused by the same or other queries.

### 3.3.7 Distributed Transient Versioning

Below, distributed algorithms that have been presented for some of the methods for transient versioning are discussed. All methods assume a two-phase commit
(2PC) protocol to assure atomic commitment of distributed transactions, and extra information is piggybacked on the 2PC messages in order to ensure transaction-consistent execution of distributed queries.

Chan and Gray, 1985

Chan and Gray have proposed a distributed version of the method by Chan et al. [ChGr85]. When a node sends a "prepared to commit" message to the transaction coordinator, it appends its current completed transaction list (CTL). The transaction coordinator assigns as its new CTL, the union of all received CTL's including its own. The new CTL is appended to all "commit" messages. When receiving a "commit" message, a node will include all transaction listed in the received CTL in its own CTL. This ensures that all transactions the committing transaction may depend on at other nodes, are also included in the CTL. Note that the entire CTL needs not be piggybacked to each 2PC message. It is sufficient to piggyback a differential list containing only the transactions added to the CTL since the last message to the recipient.

A query coordinator starts a query by requesting the local copy of the CTL from nodes where the query is going to read data. When the coordinator has received all requested CTL's, it computes the union of the CTL's. The resulting CTL is used by all nodes when determining the versions to be read. This extra coordination phase before starting the query, also makes it possible to guarantee that no versions needed by a query are garbage-collected before the query is finished. Instead of computing the union of the participating CTL's, the query coordinator could decide to use its local CTL as the CTL of the query. Hence, the first round of messages could be avoided. However, the cost will be that all transactions committed before the start of a query will not necessarily be seen by the query. In addition, versions needed by the query may already have been garbage-collected at some nodes.

Executing queries using this distributed method will only guarantee weak query consistency. The main reason is that there are no synchronization between different query coordinators. However, even queries using the same query coordinator may not be serializable since they access different sets of nodes, and since messages may not arrive out-of-order.

Mohan et al., 1992

The distributed version of the method by Mohan et al., is based on assigning the same version period to a transaction at all nodes [Moh+92b]. The transaction coordinator will assign a version number to a transaction when starting the 2PC protocol. All other nodes are informed about the version number of a transaction in the first phase of the 2PC protocol. If the version number of a transaction is smaller than the number of the current version period at a node, this node will vote to abort this transaction. On the other hand, if the version number is higher than that of the current version period, the transaction may not release its write locks until the version period corresponding to its version number is started. This is necessary in order to prevent transactions belonging to earlier version periods from seeing the effects of this transaction.

The advantage of this approach, is that since no information is piggybacked on the commit message, the read-only optimization of the 2PC protocol could be
used [Sam+95]. That is, the second phase of the commit protocol can be avoided for nodes where the transaction has not made any updates. However, this advantage is probably outweighed by the fact that transactions may be aborted or blocked. The probability of abortion could be reduced by keeping track of the current version numbers at nodes participating in the transaction. Then, a version number slightly higher than the maximum could be chosen for the transaction. However, this will increase the likelihood of transactions being blocked.

Since queries will run in the same version period at all nodes, this distributed method guarantee strong query consistency.

Bober, 1993

In his thesis, Bober discussed a method for distributed execution of the MVQL algorithm [Bober93]. A node will piggyback on the prepare message information about all local AFTER set insertions involving the transaction being committed. The transaction coordinator will piggyback this information on the commit messages to all nodes participating in the transaction. This way, a transaction inserted in a query’s AFTER set at one node, will be inserted in the query’s AFTER set at all nodes. The method requires that queries block behind transactions that are prepared-to-commit and have not been inserted in the AFTER set, locally. This is necessary since it will not be known until commit time whether these transaction should be inserted in the AFTER set.

This method does only guarantee weak query consistency for the same reason as described for the method by Chan and Gray.

Oracle DBMS

In the Oracle DBMS, 2PC messages are used to propagate information on the system change numbers (SCN) of transactions [HaBa95]. The transaction coordinator will collect the locally highest SCN’s at prepare time from all participating nodes. The highest of these SCN’s will be assigned to the transaction, and this number will be piggybacked on the commit message to all participants. As described above, queries will be assigned the currently highest SCN at startup, and if the query’s SCN is smaller than the SCN of the last transaction that updated a tuple, a prior version of the page must be reconstructed. For transactions that are prepared, but not yet committed, the query will have to block behind a transaction if the query’s SCN is greater than the prepare time SCN of the transaction. When the commit message is received, the correct SCN of the transaction is established, and the query can decide whether to read the current or the previous version of the tuple.

3.4 Compensation-Based Query Processing

Srinivasan and Carey have proposed a method called compensation-based query processing that allows queries to execute concurrently with update transactions [SrCa92a]. Concurrent updates to data accessed by a query, is communicated to the query process, which compensates for the updates so that the changes are reflected in the result of the query. Unlike transient versioning, only consistent versions
of data relevant to the query need to be represented. This can be viewed as a form of semantic versioning [Bobe93].

The compensation-based approach is based on a method for on-line index construction [SrCa92b]. The query process executes in two phases, the scan phase and the compensation phase (Figure 3.4). In the first phase, queries do a “fuzzy” scan of the base relations, and a set of temporary relations are created. In addition, during the scan phase, transactions that update relations being queried, append a compensation record for each update to an update-list. The update-list entry for a tuple update contains the type of update (insert or delete\textsuperscript{4}), the values of the attributes that are relevant to the query, and an tuple ID.

During the compensation phase, the temporary relations are combined with the changes stored in the update-list in order to provide the query with a transaction consistent view of the relevant data. This is done by sorting both the temporary relations and the update-list on a criteria that includes the tuple-id. Only the last entry (insert or delete) in the update-list needs to be kept for each tuple. The two sorted runs are then merged to resolve inconsistencies using the following rules:

1. If the last entry for a particular tuple in the update-list is an insert, this tuple is included in the output of the merge.

2. If the last entry for a particular tuple in the update-list is a delete, this tuple is omitted from the output of the merge.

3. If the temporary relation contains a particular tuple, and the the update-list has no entry for this tuple, this tuple is included in the output of the merge.

\textsuperscript{4}A modification of a record is entered as a delete followed by an insert.
The above rules ensure that the merged list contains a copy of the relevant information from the relation that is transaction-consistent at the end of the scan phase.

In order to achieve a transaction-consistent result, either dirty reads must be avoided, or the query must wait in the compensation phase for all of the transactions that have updated the update-list to terminate. Dirty reads can be avoided by locking a tuple while it is being read, using the same mechanism as for cursor stability locking [GrRe93].

In order to handle aborting transactions, either update transactions will have to wait until commit time to insert their entries into the update-list, or the the query process will have to analyze the update-list to determine which of the updates are committed and which are aborted. In order to be able to do this, the entries in the update-list must be tagged with their transaction-id, and each update transaction should append a commit or abort record to the update-list when it terminates.

Srinivasan and Carey also presented how to execute aggregation, projection, selection and join queries using this scheme. Selection and projection is straightforward as described above. When doing aggregation, a temporary result is first computed in the scan phase. In the compensation phase, the records of the update-list are applied to the temporary result in order to compute the transaction-consistent answer. In order for this to work, the query process must scan the relation in a pre-specified order, and in addition maintain a cursor that gives the position of the last tuple that it has completed reading. Thus, update transactions can establish whether an update will be automatically read by the query process (i.e. the tuple is ahead of the cursor), and add entries to the update-list only for tuples that are behind the cursor.

When executing join queries, the process of producing a transaction-consistent copy of the base relations was integrated with execution of the join. Srinivasan and Carey described how to execute nested-loops join, sort-merge join, and GRACE hash join using this strategy. (Indexed join is not applicable since a transaction-consistent index will not be available.) During the scan phase the base relations are scanned and prepared for the join which is performed in the compensation phase. That is, in addition to performing the scan, the relations are sorted on the join criteria when using sort-merge join, or partitioned when using GRACE hash join. During the compensation phase, the join is performed while checking the update-list for updates. This is done by building a hash-table containing the last entry for each tuple in the update-list. At the end, tuples that are not present in the temporary relation, but have an insert entry in the update-list are joined with the other relation.

### 3.4.1 Disadvantages

The compensation-based method by Srinivasan and Carey has some disadvantages:

- Transactions are required to look up information on concurrent queries and enter information on updated data items into the update-list. This adds extra load to transactions which should preferably be unaffected by concurrent queries.

- Temporary relations are needed to store the result of the scan phase. The temporary relations will usually have to be stored on disk, and query performance
may be significantly reduced by the extra disk accesses required to write and read these relations. In addition, disk space requirements will increase.

- The two-phased approach prevents efficient pipelining of relational algebra operations. Since the compensation phase can not start until the scan phase is finished, no tuples can be emitted from the query before the entire base relation(s) have been scanned.

- The query process must either execute under cursor stability or wait for the termination of all transactions that have updated the update-list. In other words, transactions may block queries. At the presence of long-running transactions, response times of queries may be significantly increased.

- The update-list will contain the entries for all updates made to a tuple during the scan phase. However, only the entries of the last update to each tuple will be needed in the compensation-phase.

- Consistent handling of aborting transactions requires that either transactions wait until commit time to insert their entries into the update-list, or the query process must eliminate the updates of all but committed transactions from the update-list before applying it to the temporary relations.

- The method is not applicable to transaction-consistent execution of read-only transactions consisting of several queries [Bobe93].

- A semantically rich locking model (i.e., delta locks) are not supported.

- The serialization of transactions are unnecessarily strict, and transactions may only be rolled forward. For many applications, the serial order of updating transactions as seen by a query does not necessarily have to be the same as the commit order of the transactions [Bobe93]. Weaker consistency, while still transaction-consistent, may be exploited to give better performance. Especially, in distributed database systems it may be useful to apply a more flexible scheme.

This thesis will present an alternative algorithm for compensation-based query processing that avoids most of the disadvantages listed above.

## 3.5 Distributed Database Checkpointing

Checkpointing is used in database systems in order to limit the amount of redo recovery required after a crash [H{"a}tte83]. In some database systems checkpointing is performed by creating a transaction-consistent backup copy. That way, a previous transaction-consistent state can quickly be reconstructed during recovery, and such a state may even be recovered in the case of log media failure.

Checkpointing is related to query processing in that a potentially large part of the database is accessed. In both cases, the amount of interference with normal transaction activities should be minimized. This section will discuss related work on transaction-consistent checkpoints in distributed database systems. The necessary synchronization of the nodes involved in distributed checkpointing in order to achieve transaction-consistency, is closely related to distributed query processing. For a more
Related Work

general survey on distributed database checkpointing the reader is referred to Lin et al. [Lin+97].

3.5.1 Pu, 1986

Pu has presented an approach for global consistent reading of entire databases [Pu86]. The algorithm assigns colors, white or black, to data items. Before starting the global-read, all data items are colored white. The global-read will lock each data item, read its content, change its color to black, and unlock it. In other words, white data items have not yet been read by the global-read, while black data items have already been processed.

A transaction that only updates white data items, is called a white transaction, and can execute normally since it will serialize before the global-read. Similarly, black transactions serialize behind the global-read by updating only black data items. If a transaction updates both white and black items, transaction-consistency will not be achieved. In the original algorithm, such transactions, called grey transactions, are aborted.

Pu et al. later proposed some strategies to avoid aborting transactions [Pu+88]. Basically, grey transactions may either be turned white, by undoing effects of black transactions, or turned black, by letting the global-read see all the effects of this transaction. Simulation studies show that the most efficient of these strategies, is a turn-black strategy called save-some [Pu+88]. Using the save-some strategy, grey transactions will copy the before-images of white data items they are going to update to a temporary buffer. The data items will be colored black, and when the global-read accesses a black data item, it will check the buffer for the “white” values. Since it will not be known at the beginning of a transaction whether the transaction will end up as white or grey, all before-images of white transactions should be copied to a private buffer. If the transaction turns grey, the private buffer could be transferred to the global-read buffer.

In a distributed database system, color information is propagated by the messages of the 2PC protocol. Each node piggyback the color of a transaction on its prepare message, and based on the local colors, the transaction coordinator determine the global color of the transaction. The nodes participating in the transaction, are informed about the color by the commit message.

Ammann et al. have shown that the algorithm by Pu does not guarantee transaction-consistency [Amm+95]. The problem is that Pu did only address write-write dependencies; not write-read and read-write dependencies. Write-read dependencies could be accounted for by also letting the colors of data items that are read but not written take part in determining the color of a transaction. By considering write-read dependencies, but not read-write dependencies, update query consistency, as defined by Bober [BoCa92a], will be achieved by Pu’s method.

In order to achieve weak or strong query consistency, read-write dependencies must also be considered. This requires the use of more than two different colors for data items. Ammann et al. suggest to assign a shade, pure or off, to data items in addition to the color. Figure 3.5 shows the transitions for colors and shades of data items. Gray transactions are disallowed as in the original algorithm. Furthermore, white transactions may not write an off-white item or read an off-black item. In both cases, a violation of this rule would serialize a white transaction behind a black trans-
3.5 Distributed Database Checkpointing

![Diagram of transitions for colors and shades of data items](image)

Figure 3.5 Transitions for colors and shades of data items [Amm95].

action. By using the save-some strategy, such white transactions could be turned black, and transaction-consistency achieved without aborting any transactions.

The main disadvantage of the above approach is that transactions are required to copy the before-images before updating just as for transient versioning. In addition, unlike transient versioning, the save-some strategy requires the global-read to use cursor stability locking since the global-read will not have access to the private buffers of transactions. This makes the global-read vulnerable to long-lived transactions. On the other hand, the storage requirements will be less than for transient versioning since a private buffer area is deallocated when the transaction completes. However, extra storage is required for representing color information.

3.5.2 Son and Agrawala, 1989

Son and Agrawala have proposed a method for transaction-consistent checkpointing that is designed for systems using timestamp ordering for concurrency control [SoAg89]. The method divides the transactions into two groups: before checkpoint transactions (BCPT) and after checkpoint transactions (ACPT). During a checkpoint the after-images of ACPT’s are stored in a temporary buffer instead of applied to the database. After all BCPT’s are completed, the contents of the database is copied before the database is merged with the temporary buffer.

Each node has a local clock which is manipulated by the clock rules of Lamport [Lamp78]. That is, if a message is received with a greater timestamp than the current value of the clock, the local clock is adjusted to a slightly higher value than the received timestamp. The checkpointing algorithm exploits the local clocks in order to determine the timestamp of the checkpoint. A checkpoint coordinator broadcasts a *checkpoint-request* message to all nodes. When receiving this message, a node will use the current value of its local clock, which may have been changed by the incoming message, as its temporary checkpoint timestamp. Transactions with a lower timestamp will be BCPT’s while transactions with a higher timestamp will be temporary ACPT’s (T-ACPT). A T-ACPT will be temporarily be regarded as a ACPT, but may later be converted to BCPT.

The local checkpoint timestamps will be reported to checkpoint coordinator which will choose the maximum of these timestamps as the global checkpoint timestamp. This timestamp will be communicated to all nodes and used to determined the true ACPT’s. If a timestamp of a T-ACPT is smaller than the global checkpoint
timestamp, that T-ACPT is converted to a BCPT by applying its updates to the database.

The major disadvantage of this method is the overhead in maintaining the after images for ACPT's. Especially, in order not to block transactions, access by later transactions to these versions must be supported. Adapting this method to systems using strict 2PL, requires deferred updates by all transactions since the timestamp of a transaction could not be determined until commit time. Thus, it will generally not be known at startup whether a transaction may become ACPT or not.

3.5.3 Pilarski and Kameda, 1992

Pilarski and Kameda have proposed a checkpointing approach which is based on assigning colors to data items [PiKa92]. A data item may be either white, grey, or black. Before starting a checkpoint, all data items are white. A grey data item means that the current version of the data item should be checkpointed. If a data item is black, it means that the version of the data item that should be checkpointed has been overwritten.

Transactions will either be white or black. A transaction is originally white, but becomes black if it overwrites a grey data item, reads or overwrites a black data item. Figure 3.6 shows how the colors of data items will change. Operations that do not change the color is not shown. Before a transaction overwrites a grey data item, this data item will be checkpointed.

Note that the color of a transaction will generally not be known until it has performed all its operation. Thus, if a white data item X is updated by a currently white transaction, this transaction may later turn black. In that case, X should also turn black. When using two-phase locking this could be achieved by coloring data items when releasing locks. At that time the color of the transaction will be known. For timestamp ordering, all transactions with a larger timestamp than a predetermined checkpoint timestamp will be initially colored black. The timestamp ordering protocol will ensure that if a white transaction access a black data item, it will be aborted. Thus, the final color of a transaction will be known at startup.

Since a data item will be checkpointed when it turns from grey to black, sequential checkpoint output may not be achieved. This may be a drawback when using
the checkpoint for recovery. Pilarski and Kameda do not discuss how to represent the color information of data items. If it is stored within the data items, reads may become writes. In addition, data items may have been flushed to disk before the color of the updating transaction is known. If represented in main memory, the storage overhead may be substantial for large databases. Pilarski and Kameda have earlier presented another checkpointing approach based on checkpoint number instead of colors [PiKa90]. However, this approach requires even more overhead per data item.

The idea of representing transaction dependencies by color information associated with data items and transactions will also be used in this thesis.

### 3.5.4 Kim and Park, 1994

Kim and Park have proposed a checkpointing approach which maintains two separate copies of the database, the current database and the checkpoint area [KiPa94]. During normal processing, transactions only access the current database. When checkpointing is started, the transactions are divided into a BEFORE set and an AFTER set. Updates by members of the BEFORE set will be reflected in the current database while transactions belonging to the AFTER set will update the checkpoint area.

When all updates by transactions in the BEFORE set are reflected in the current database, the roles of the two storage units are exchanged. The main task of the checkpoint process will be to bring the new current database up-to-date by copying data from the former previous current database (now checkpoint area) to the new current database. During this copying, care must be taken not to overwrite data written by transactions in the AFTER set. This is achieved by maintaining a bitmap indicating which data items are already up-to-date. Before the new current database is up-to-date, transactions may also have to access the (new) checkpoint area.

All transactions that have not yet become prepared-to-commit when checkpointing starts at a node, will be included in the AFTER set of the checkpoint. All prepare messages to a transaction coordinator will contain information on whether checkpointing has been started at this node. If checkpointing has started at any node participating in a transaction, the transaction coordinator will, via the commit message, tell all participants to include this transaction in the AFTER set. Thus, no transaction will be member of the BEFORE set at one node and of the AFTER set at another node. If a node is told by the transaction coordinator to insert a committing transaction in its AFTER set when there is no on-going checkpointing, checkpointing is started immediately. Hence, all transactions which have not yet become prepared-to-commit, will be included in the AFTER set. Since strict 2PL is assumed, this will guarantee that all transactions dependent on the committing transaction will also be members of the AFTER set.

Figure 3.7 shows an example with a distributed transaction $T_1$ executing at both node 1 and node 2 and a local transaction $T_2$ executing at node 1. When $T_2$ becomes prepared-to-commit at node 2, checkpointing has already been started. Hence, the transaction coordinator will tell both node 1 and node 2 to include $T_1$ in the AFTER set. When the commit message for $T_1$ arrives at node 1, checkpointing has not yet been started. By starting checkpointing at this point, $T_2$ will also be included in the AFTER set. If the start of checkpointing had been delayed until the checkpoint
request was received, $T_2$ would not have been included in the AFTER set. That might have violated the consistency of the checkpoint since $T_2$ may possibly be dependent on $T_1$.

Transactions that are active when a new checkpoint is to be generated, may already have updated the current database. Hence, these transactions should be members of the BEFORE set. In order to avoid that these transactions are included in the AFTER set, the checkpointing is delayed until all these transactions have become prepared-to-commit. When checkpointing is to be started, the checkpoint coordinator will broadcast a *pre-coordination* message. When a node receives this message, it will build a list of on-going transactions and wait for all these transactions to become prepared-to-commit before replying to the checkpoint coordinator. When the checkpoint coordinator has received all replies, it starts checkpointing by broadcasting the *checkpoint-request* message. Transactions which are started during this pre-coordination phase, will store their updates in a temporary buffer until it is known whether they belong to the BEFORE set or the AFTER set.

There is a flaw in the algorithm as presented by Kim and Park. Consider a transaction $T$ that is active at the reception of the pre-coordination message at node 1, but starts during the pre-coordination phase at node 2 (Figure 3.8). Node 1 will wait for $T$ to become prepare-to-commit before replying to the checkpoint coordinator, while node 2 will not. This means that the checkpoint coordinator may send, and node 2 possibly receive, the checkpoint request before $T$ has become prepared-to-commit at node 2. If $T$ becomes prepare-to-commit after the checkpoint has started at node 2, the transaction coordinator, upon receiving the prepare message from node 2 indicating that checkpointing has started, will include $T$ in the AFTER set. However, $T$ will be in the BEFORE set at node 1 since its updates have already been applied to the current database. This flaw can be avoided by requiring that all transactions that are active at the start of the pre-coordination phase have terminated, not just become prepare-to-commit, before initiating the checkpoint process.

Kim and Park do not discuss how to handle system crashes during checkpointing when both database areas may be inconsistent. This problem could be solved by making a backup copy of the checkpoint area between checkpoints.

One major drawback of the checkpoint algorithm by Kim and Park is the copying needed to make the new current database up-to-date. Before the current database is up-to-date, response times and performance of transactions will be affected since
transactions may have to access both the current database and the checkpoint area. Kim and Park also assumed page granularity locking. If finer granularity is used, the space requirements for the bitmap indicating which data item needs copying, may become prohibitive for large databases.

One of the algorithms for distributed query processing that is presented in this thesis, will exploit the messages of the 2PC protocol to synchronize query execution between nodes in a similar way as described above.

### 3.6 Concluding Remarks

This chapter has presented previous work on concurrent transaction and query processing. Two different approaches to avoid data contention between transactions and queries while still guaranteeing transaction-consistency have been presented. The transient versioning algorithms presented in Section 3.3 can be divided into two main categories. The approaches suggested in [Cha82, AgSe89, BoCa92b] will always provide a query with the latest committed version of each data item, as of the start of the query. In other words, a potentially unlimited number of versions may be required. The approaches suggested in [Moh92b, Wu93] restricts the number of a concurrent versions at the expense of letting queries see obsolete data.

Compensation-based query processing as proposed in [SrCa92a], will always provide a query with the latest committed versions. Only versions for the relevant portions of the database are materialized for use by queries, and the materialization may be tailored for specific query operations. However, for the relevant portions of the database, a unlimited number of versions may be stored in the update-list, and the same versions may be duplicated in the update-lists of different queries. As discussed in Section 3.4, the proposed algorithm also has quite a few disadvantages with respect to query performance.

A problem with all presented algorithms for transient versioning and compensation-based query processing is that they require extra work by transactions. Versioning is performed by transactions in all approaches.

The rest of this thesis will present an alternative approach to compensation-based query processing that requires much less work by transactions, materializes only the version actually needed by queries, and, at the same time, avoids most of the disadvantages of the compensation-based approach presented in [SrCa92a]. This thesis also presents the first distributed algorithms for compensation-based
query processing. Aspects from the works on distributed database checkpointing presented in this chapter will be applied in these algorithms.
Part II

Centralized Query Processing

This part presents several algorithms for compensation-based query processing and investigates their performance.

Chapter 4 Several compensation-based algorithms is presented. One of these algorithms, which is based on undo/no-redo compensation, is explored in most detail. The rest of the thesis will focus on this algorithm.

Chapter 5 The chapter discusses how the basic algorithm for undo/no-redo compensation may be optimized for different types of relational queries.

Chapter 6 An analytical performance model for undo/no-redo compensation is presented and used to evaluate the performance of the algorithm.

Chapter 7 A simulation experiment evaluating the performance of compensation-based query processing is presented. The performance of undo/no-redo compensation is compared to alternative algorithms.
Chapter 4

Log-Oriented Compensation-Based Query Processing

As stated in Chapter 1, the goal of this work is to run transaction-consistent queries in a database system without affecting the performance of concurrent transactions. Unfortunately, the existing such methods put extra load on updating transactions.

In the compensation-based approach proposed by Srinivasan and Carey, transactions must twice record information about updates they have made [SrCa92a]. In addition to inserting entries in the update-list, traditional logging of updates have to be performed in order to support recovery. This means that if the compensation could be based on the database internal log instead of the update-list, transactions would not have to maintain an update-list. In fact, they would not have to bother about concurrent queries at all. Transactions could just go ahead logging their updates as usual.

The compensation-based query-processing method presented in this chapter, bases its compensation on the database internal log.

4.1 Basic Idea

In order to achieve non-blocking execution of queries with respect to transactions, queries will not set any locks on the tuples they access. This will give the query an inconsistent view of the database. However, the log will be used to bring the result of the scan into a transaction-consistent state. This is done by collecting information about concurrent updates from the log. A transaction-consistent query result is achieved by redoing and undoing operations recorded in the log.

Using this approach, a query process will, in addition to the scanning the base relation(s) of the query, scan the log, extracting relevant information from log records, and perform the necessary undo/redo-operations before emitting a tuple. In other words, a query process will consist of the following main tasks:

Scan The base relation(s) of the query are scanned tuple-by-tuple without setting any read locks. Thus, a query will not have to wait for transactions to commit
before reading a tuple, and transactions, which execute using strict two-phase locking (2PL) [Es96], will not have to wait for the query to finish before updating a tuple. The scan does not necessarily need to be a sequential file scan; other access methods (e.g., an index) may be used.

Log Processing: Concurrently with the scan, transactions that update the relations being queried will have entered their updates into the log. The query process will extract information from these log records. What information is extracted will depend on the type of query.

Compensation: The information extracted from the log is used in order to undo or redo operations of concurrent transactions. The set of all concurrent transactions is divided into two disjoint subsets, the BEFORE set and the AFTER set. When doing this division, it is ensured that no transaction in the AFTER set is dependent on any transaction in the BEFORE set. In order to achieve transaction-consistency, a query result should reflect all updates made by transactions in the BEFORE set and no updates made by transactions in the AFTER set. In other words, transactions in the BEFORE set are rolled forward, while transactions in the AFTER set are rolled back. Thus, the task of the compensation is to redo operations of members of the BEFORE set and to undo operations of members of the AFTER set.

When performing the scan, the equivalent of a latch will be set on each tuple only while it is being read to protect the read operation from other operations. For each tuple encountered in the scan, the data necessary for the execution of the query is extracted. In addition, the tuple identifier and the state identifier of the tuple will be extracted. The tuple identifier will be needed in order to identify the log records of a tuple. For the presentation below, it is assumed that a content-based primary key is used for tuple identification. In order for the primary key to be used, it must be stable. That is, if the primary key is to be changed, it must be logged as a delete-insert pair. Any stable database internal tuple identifier may be used instead of a primary key. The state identifier will be used to determine which operations have been applied to the tuple before it was read. The state identifier of the corresponding block may be used if state identifiers are not maintained for each tuple.

In order to be able to relate log records to the tuples read during the scan, transactions must log their state-changing operations using a record oriented logging policy [Hvas96]. This means that each log record will contain an unique tuple identifier (e.g., a relation identifier and primary key) for the tuple that was affected by the operation.

In order to ensure that the necessary log records are processed, log processing must, in general, start at the first log record of the oldest active transactions at the time the scan starts. Log processing may also have to continue after the scan is finished in order to process log records of concurrent transactions that are still active. However, the division of concurrent transactions into the BEFORE set and the AFTER set may be done in such a way that the processing of log records created before or after the scan is avoided.

\footnote{An explicit latch is not necessary if the reading of a tuple is an atomic operation with respect to other processes of the DBMS. This could be achieved if the DBMS is implemented as a single process using non-preemptive scheduling of its threads.}
4.1 Basic Idea

Based on the state identifier of a tuple, the log sequence numbers (LSN) of log records, and whether transactions that created these log records belong to the AFTER set or the BEFORE set, it is decided which operations need to be compensated for. When rolling a transaction in the AFTER set back, all operations performed by the transaction on a tuple before it was read by the scan are undone. When rolling a transaction in the BEFORE set forward, all operations performed on a tuple after it was read by the scan are redone.

Approaches to compensation-based query processing may be classified into three categories based on which criteria is used when establishing the BEFORE set and the AFTER set:

**Undo/No-Redo** All transactions that are active\(^2\) during the scan are included in the AFTER set. Thus, the query’s view of the database will only include updates by transactions that committed before the start of the query. This means that log records created before the start of the query may have to be processed.

**No-Undo/Redo** All transactions that are active during the scan are included in the BEFORE set. In addition, transactions that are started after the end of the scan, may have to be included in the BEFORE set because other members of the BEFORE set may depend on them. Using this approach, the query’s view will be more up-to-date than for undo/no-redo compensation. However, the query process will have to wait for all transactions that are active during the scan to terminate. The method proposed by Srinivasan and Carey [SrCa92a] uses this approach.

**Undo/Redo** The only restriction on the establishment of the BEFORE set and the AFTER set, is that no transaction in the BEFORE set may depend on any transaction in the AFTER set. One way to ensure this, without having to keep track of transaction dependencies, is to include all transactions that have committed before a certain point in time in the BEFORE set, while all transactions committed after that point are included in the AFTER set. As long as no transaction is active during the entire scan, undo/redo compensation makes it possible to avoid processing log records created before or after the scan. Thus, a more up-to-date view of the database than for undo/no-redo processing is possible without having to wait for active transactions to terminate.

One way of viewing this compensation-based approach is that a base relation of a query is joined with the log (Figure 4.1). The join keys of both operands of the join operation will be the primary key. The join will be a kind of outer join since tuples that have no corresponding log record will be part of the result. Due to the similarity with the join operation, many of the same techniques (e.g., nested loops, sort-merge, hash-based) could be used to combine the relations and their log records. The algorithms presented in this thesis will thus be modified versions of ordinary algorithms for processing joins. Since hash-based algorithms show the best overall performance for joins of large tables [HaRa96], most of the algorithms will be based on hashing.

---

\(^2\) A transaction is regarded as active from its start until it is terminated (i.e., committed or aborted).
Log-Oriented Compensation-Based Query Processing

The main difference from an ordinary join is the temporal aspect. Log records that will be needed for the compensation of a tuple, may not yet exist when a tuple is read by the query. Thus, a two-phased approach where compensation is delayed until the log records are available may be necessary. Note that, this is only necessary if compensation may perform redo operations. For undo operations, all necessary log records are available in the log when the tuple is read. Thus, by using undo/no-redo compensation a two-phased approach can be avoided. As discussed in Section 3.4, this could significantly improve the efficiency of compensation-based query processing. Due to this advantage, most of this thesis will focus on algorithms for undo/no-redo compensation. Algorithms for other approaches will only be briefly discussed.

4.2 Undo/No-Redo Compensation

As discussed above, one of the main advantages of undo/no-redo compensation is that all log records needed for performing the compensation on a tuple are already available when the tuple is read. For each tuple read during the scan, the log records for this tuple can be fetched and the necessary operations undone before the next tuple is read. By interleaving scanning and compensation in such a manner, intermediate storage of the scanned tuples, possibly on disk, is avoided.

This interleaved execution requires direct access to the log records of a tuple. Direct access on primary key to log records is usually not supported by traditional database systems. In order to support such access, the log processing activity retrieves relevant information from log records and stores it in an update-table for later use. This update-table will support direct access (hash-based) on primary key, and, for the first presentation of the algorithm, it will be assumed that it can be entirely stored in main memory. Efficient methods for storing update-tables on disk
4.2 Undo/No-Redo Compensation

![Diagram](image)

**Figure 4.2** Undo/no-redo compensation.

are discussed in Section 4.4. Where not otherwise stated, it will be assumed that each query maintains its own update-table. However, it is possible to let several queries share the same update-table.

When executing queries using undo/no-redo compensation, all transactions that are active during the scan will be members of the query's AFTER set. Hence, relevant information from all log records created by members of the AFTER set must be entered in the update-table.

A query process executing using undo/no-redo compensation consists of one scan thread and two log processing threads (Figure 4.2). Since the compensation will be interleaved with the scan, both operations will be performed by the scan thread. In addition to the log processing thread that processes log records that are created during the scan, a separate thread will be used to process the log records that are created by transactions in the AFTER set before the start of the query.

Below, an algorithm for transaction-consistent execution of queries using undo/no-redo compensation is presented. For the first presentation it is assumed that the state-changing operations performed by transactions are restricted to write, insert, and delete. It will later be explained how delta operations may be handled. Both partial and complete tuple logging may be used, however, where not otherwise stated, partial tuple logging can be assumed.

### 4.2.1 The Log Processing Threads

The log processing threads process all log records\(^3\) produced before the end of the scan by transactions in the AFTER set. When a query starts, its *forward log*
processing thread (FLP) start processing all new log records. In order to avoid reading the log records from disk, FLP should process log records before they are removed from main memory.

The log records produced by members of the AFTER set before the start of the query, must also be processed. This is done by a backward log processing thread (BLP). At the start of the query, BLP will insert into its AFTER set all transactions that are recorded as active. It will then go backwards in the log, processing all log records created by members of this set. This could either be done by a sequential scan, or by following the transaction log chains backward in the log for all members of the AFTER set. Note that BLP may have to fetch old log records from disk. The shaded log records in Figure 4.3 represent the records that will be processed by the log processing threads. The arrows represent the log chains of transactions that are active at the start of the query.

At any time during query execution, the update-table will contain one entry for each tuple that so far has been changed by a transaction in the AFTER set. Each entry will contain one value for each relevant attribute that has been updated. An attribute is relevant if it is either part of the query's projection or is needed for processing the query (e.g., used in the selection predicate). The values used in the entries are the before-images of operations to the tuple.

When a log processing thread processes a log record, it does a hash-based lookup into the update-table using the primary key found in the log record. BLP will enter the before-images of all attributes found in a log record into the update-table, substituting possible previous values. FLP will only enter a before-image into the update-table if no previous value exists for this attribute. BLP and FLP will be executed in parallel. FLP must process log records in chronological order, while the only restriction for BLP is that the log records of each transaction is processed in reversed chronological order. Log records by different transactions may be processed in any order since the use of strict 2PL prohibits that two transactions that are active at the start of the query could have updated the same tuple. When BLP is finished, the update-table will contain the committed values at the start of the query for all attributes that have so far been changed by members of the AFTER set.
4.2 Undo/No-Redo Compensation

Table 4.1 An overview of what the log processing threads enter into the update-table when processing a log record.

<table>
<thead>
<tr>
<th>Current entry type</th>
<th>FLP</th>
<th>BLP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>write</td>
<td>delete</td>
</tr>
<tr>
<td>None</td>
<td>Wall</td>
<td>D:all</td>
</tr>
<tr>
<td>Write</td>
<td>W:new</td>
<td>D:new</td>
</tr>
<tr>
<td>Delete</td>
<td>D:no</td>
<td>D:no</td>
</tr>
<tr>
<td>Insert</td>
<td>I:no</td>
<td>I:no</td>
</tr>
</tbody>
</table>

Transactions in the AFTER set may commit before BLP is finished. In that case, the garbage collection of the undo information (after-image) of their log records must be delayed until BLP has finished.

The entries in the update-table could be of three different types: write, insert, or delete. Write entries contain before-images of relevant attributes that have been changed by members of the AFTER set. For delete entries, the before-images of all relevant attributes of the deleted tuple are included in the entry. Insert entries contain no before-image. How a tuple is handled by the scan thread is determined by the type of its entry in the update-table. A write entry tells the scan thread that this tuple has been updated by transactions in the AFTER set, and the attribute values stored in the update-table should be used instead of those currently recorded in the tuple. An insert entry means that this tuple should be ignored since it has been inserted by a transaction in the AFTER set. Delete entries represent tuples that have been deleted by transactions in the AFTER set. Tuples with delete entries must be included in the query result even if they are not read by the scan thread.

Table 4.1 shows what information the log processing threads enter into the update-table. Their decision is based on what is previously entered in the update-table, the type of the current log record, and whether this log record was found during backward or forward processing. The capital letters show the type of the entry after the log record has been processed (Write, Delete, Insert). The last part shows which before-images are entered into the update-table. all means that the before-images of all relevant attributes present in the log record will be entered into the update-table, possibly substituting old values. new means that only before-images of relevant attributes that are not present in the old entry should be entered. no means that no before-image from this log record should be entered into the update-table. (—” indicates an impossible combination.) Summarizing Table 4.1, the type of an existing entry will not be updated when processing log records for write operations, and the type of a write entry will always be changed when processing log records for delete and insert operations. The type of delete or insert entries will only be changed by BLP.

If a semantically rich locking model is used, rules for entering information into the update-table become somewhat more complex. How log records for delta operations should be handled will be discussed in Section 4.3.5.
Scan-Thread(R)
1 for each tuple t in base relation R do
2    Read t
3    if state-identifier[t] < LSN of oldest log rec. processed by BLP then
4        emit t
5    else
6        Wait for necessary log processing by FLP
7        k ← Update-Table-Entry(primkey[t])
8        if k = NIL then
9            emit t
10       else
11          if type[k] ≠ INSERT then
12              t ← Compensate(t, k)
13              emit t

Figure 4.4 Basic algorithm for the scan thread.

4.2.2 The Scan Thread

The scan thread waits until BLP is finished before starting the scan of the base relation(s) of the query. At that time entries have been made in the update-table for all tuples that have so far been modified by transactions in the AFTER set. How the scan thread is executed depends on the operations of the query. Figure 4.4 shows the basic algorithm for producing a snapshot of a relation. The scan thread scans the entire relation and checks for each tuple the corresponding state identifier. If the state identifier is smaller than the LSN of the oldest log record processed by BLP, the tuple can be emitted as it is since it could not possibly have been changed by members of the AFTER set.

Before checking the update-table, the scan thread must make sure that the necessary log records have been processed by FLP. This synchronization between the scan thread and FLP is further described in Section 4.2.3. When the scan thread has made sure that the necessary information for the current tuple has been entered in the update-table, it does a hash-based lookup in the update-table on the primary key of the current tuple. If an insert entry is found, the tuple is not emitted since it has been inserted by a member of the AFTER set. If an update or delete entry is found, the compensation is performed. That is, the values of each attribute found in the update-table are substituted for the corresponding attribute values of the tuple. If no entry is found in the update-table, the tuple is emitted in the form it was read.

When the entire relation has been read by the scan thread using the algorithm of Figure 4.4, the result will include all tuples of a transaction-consistent snapshot, except possibly some tuples that have been deleted during the scan. These tuples could be emitted at the end of the scan by searching the update-table for delete entries that have not been visited by the scan thread. However, query evaluation algorithms often exploit that the scan sequence is sorted on a combination of attributes. In Section 4.3.1 it will be discussed how the scan order can be preserved in the output of the query.
4.2 Undo/No-Redo Compensation

4.2.3 Synchronizing the Scan Thread and the Forward Log Processing Thread

Before the scan thread performs compensation, all log records of operations that were reflected in the current tuple when it was read, must have been processed by FLP. If not, the scan thread will wait for the FLP thread to process more log records.\(^4\)

One way to ensure that sufficient log has been processed, is to process all new log records before compensation is performed. However, a less eager strategy can be used by taking advantage of the state identifiers included in each tuple/block. If the state identifier of the current tuple/block is smaller than the LSN of the last log record processed by FLP, all operations needed for doing the compensation are already entered into the update-table. Thus, no more log records need to be processed before compensation is performed. Otherwise, the scan thread is suspended until the necessary log records have been processed. In other words, line 6 of the algorithm in Figure 4.4 should be changed to:

\begin{verbatim}
while state-identifier[t] > LSN of last log rec. processed by FLP do
  Wait
\end{verbatim}

Using the synchronization described above instead of processing all new log, the amount of log processing could normally be reduced. However, FLP should still make sure to process all log records before they are removed from main memory. Thus, how much log processing is saved by loose synchronization will depend on how much space is reserved in main memory for log pages.

4.2.4 Example

To illustrate the above algorithm, a relation called EMPLOYEE will be used. Table 4.2 shows the original content of this relation before running any queries.\(^5\) The primary key of the relation is emp_no.

Figure 4.5 shows an example of a query making a snapshot of the EMPLOYEE relation. Time increases increases from top to bottom of the figure. The leftmost column represents the output from the query, and the next column represent the tuples read by the scan thread. A plain arrow, \(\rightarrow\), represents tuples that need no compensation, while \(\overleftarrow{\rightarrow}\) is used for tuples for which compensation is performed. \(\overrightarrow{\rightarrow}\) is used for deleted tuples that are emitted at the end of the scan.

The rightmost column shows the log records processed by FLP. For simplicity, it is assumed that BLP processes no log records. The first three fields of the log records are the log sequence number, the type of operation, and the primary key (emp_no). For insert and delete operations, the rest of the log record lists the rest of the after-image or before-image, respectively. For write operations, the rest of the log record contains, for each attribute that has been updated, (the initial of) the attribute name, the before-image and the after-image. In other words, partial tuple logging is assumed.

---

\(^4\)Note that if before-images of all relevant attributes are available in the update-table, the compensation can be done without synchronizing with FLP. Processing more log records will in this case never change the entry in the update-table.

\(^5\)The characters and firm depicted in this relation are fictitious. Any similarity with actual persons, living or dead, or to actual firms is purely coincidental.
Table 4.2 The EMPLOYEE relation.

<table>
<thead>
<tr>
<th>emp.no</th>
<th>name</th>
<th>salary</th>
<th>dept.no</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Karl</td>
<td>29</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>Tore</td>
<td>28</td>
<td>02</td>
</tr>
<tr>
<td>3</td>
<td>Sophus</td>
<td>32</td>
<td>03</td>
</tr>
<tr>
<td>4</td>
<td>Øystein</td>
<td>35</td>
<td>03</td>
</tr>
<tr>
<td>5</td>
<td>Oddmund</td>
<td>35</td>
<td>03</td>
</tr>
<tr>
<td>6</td>
<td>Heidi</td>
<td>27</td>
<td>02</td>
</tr>
<tr>
<td>7</td>
<td>Gunnar</td>
<td>38</td>
<td>02</td>
</tr>
<tr>
<td>8</td>
<td>Svein E.</td>
<td>33</td>
<td>03</td>
</tr>
<tr>
<td>9</td>
<td>Øystein</td>
<td>35</td>
<td>03</td>
</tr>
<tr>
<td>10</td>
<td>Yurong</td>
<td>31</td>
<td>03</td>
</tr>
<tr>
<td>11</td>
<td>Harald</td>
<td>45</td>
<td>02</td>
</tr>
<tr>
<td>12</td>
<td>Ole</td>
<td>26</td>
<td>03</td>
</tr>
<tr>
<td>13</td>
<td>Bjørn</td>
<td>31</td>
<td>03</td>
</tr>
<tr>
<td>14</td>
<td>Bjørn</td>
<td>31</td>
<td>03</td>
</tr>
</tbody>
</table>

**Figure 4.5** Example of making a transaction-consistent snapshot of the EMPLOYEE table.
The update-table contains one column for each of the attributes of the relation in addition to the entry type. Entries are listed at the time they are made. An entry will overwrite any previous entry with the same emp_no. The example shows that FLP enters the before-images of attributes found in the log records into the update-table only if no previous value exists. For example, when processing log record 10, the before-image of dept_no (01) is not entered in the update-table since the entry for tuple 16, already contains a value (03) for this attribute. Since log records for insert, contain no before-images, the insert entry made for log record 5, consists only of entry type and primary key. No entry is shown for log record 9 since processing it will not change the entry for tuple 17.

Tuples for which the update-table contain no entry at the time they are read by the scan thread (i.e., no entries listed in the update-table above their position in the column of the scan thread), are emitted unchanged. For other tuples compensation is performed. For example, when emitting tuple 2, dept_no is changed from its current value (01) to the value found in the update-table (02). For the same reason, the name and salary of tuple 1 and all attributes of tuple 16 are changed.

Tuple 12 is not emitted since the update-table contains an insert entry. That way, including tuples inserted after the start of the query is avoided. Tuple 7 is emitted at the end, when searching the table for unprocessed delete entries. Tuple 17 was emitted when the the reinserted tuple was read by the scan thread.

### 4.3 Extensions and Optimizations of Undo/No-Redo Compensation

This section presents several extensions and optimizations to the basic algorithm presented in Section 4.2.

#### 4.3.1 Preserving Scan Order in the Query Output

Query evaluation algorithms often exploit that the scan sequence is sorted on a combination of attributes, the sort key. This will not be possible if deleted tuples are emitted at the end of the scan. In order to be able to preserve the scan order, the scan thread should be informed by the log processing threads about deleted tuples that must be emitted before the current tuple in order to preserve the scan order. This can be achieved if the log processing threads, when processing a log record for a deletion, also insert the sort key of the tuple into a priority queue together with a reference to the corresponding entry in the update-table. For each tuple it reads, the scan thread will check the priority queue for tuples that should be emitted before the current tuple. These tuples can be reconstructed by using the entries stored in the update-table. Note that an entry for the current tuple may be found in the priority queue in case it has been reinserted. In that case, the scan thread should skip the current tuple and only use the entry in the priority queue.

Using this scheme, when log record 3 is processed in the example of Figure 4.5, a reference to the entry for tuple 7 is inserted in the priority queue. After the scan thread has read tuple 8, it checks the priority queue and discovers that it contains an entry for a tuple with a smaller key (7). Hence, the content of the update-table for tuple 7 will be emitted before emitting tuple 8. When checking the priority queue
before emitting tuple 17, it discovers that the priority queue also contains an entry for this tuple. Hence, the version in the update-table is used instead of the tuple that has been read.

The priority queue could also be used when it is not necessary to preserve scan order. Then, tuples in the queue could be emitted at any time during the scan. This way, scanning the update-table for unprocessed delete entries at the end of the scan can be avoided.

When synchronizing the scan thread and FLP, the state identifier of the current block can be used to check whether all possible deletions of tuples with a smaller sort key than the current tuple have been entered in the priority queue. When crossing block boundaries the state identifiers of both blocks must be checked. If a relation is scanned using a secondary index, the state identifiers of the index blocks must be used instead (Ref. Section 4.3.3).

4.3.2 Space Optimization of the Update-Table

In order to minimize memory usage, the update-table should be kept as small as possible. The following optimizations will reduce the size of the update-table:

- Only before-images of relevant attributes are entered into the update-table.

- If the scan order is predefined, FLP should, if possible, check whether the tuple referred in the current log record has already been read by the scan thread. If so, it is not necessary to enter information from this log record into the update-table. For example, FLP may skip log record 7 in Figure 4.5 since tuple 5 has already been read by the scan thread. In order to be able to decide whether a tuple lies behind or ahead of the scan thread, the log record must contain the attributes determining scan order.

- When the scan thread has finished processing a tuple, the entry for this tuple in the update-table can be removed. However, if it is not guaranteed that FLP will be able to decide whether the tuple of a later log record lies behind or ahead of the scan thread, a new entry may be made for this tuple later. In order to avoid this, only the attribute values could be deleted, while the primary key is kept and the entry is marked processed. However, if the probability of another update to the same tuple is low, such entries will not be cost-effective.

For primary key range scans, tuples that will not be read by the scan thread can be safely ignored by the log processing threads. Range scans using a secondary index must be treated differently, since updates to the index keys are possible. Section 4.3.3 explains how such scans should be handled.

An entry may be made in the update-table for a tuple that has already been accessed by the scan thread or lies outside the range of the scan, because the log record did not contain enough information to determine this. Adding information of later log records may show that this entry will never be needed. At that time its attributes may be safely deleted and the entry marked as processed.

Section 4.4 discusses how large update-tables may be (partly) stored on disk. The space optimizations presented above will also be important in that case. Reducing the size of a disk-based update-table will reduce the amount of disk I/O needed to execute the query.
4.3 Extensions and Optimizations of Undo/No-Redo Compensation

![Diagram](image)

**Figure 4.6** An update causes the scan thread to miss an index entry.

### 4.3.3 Index-Based Scans

If a secondary index is used for the scan, an update to an attribute may cause the index entry for this tuple to move. If the index entry of a tuple that has not yet been read by the scan thread is moved to a part of the index that has already been processed by the scan thread or to a position outside the range covered by the scan, the scan thread will overlook this tuple. Figure 4.6 shows an update to the index entry that currently lies ahead of the cursor used by the scan thread. The update causes the index entry for tuple 17 to be moved behind the cursor. Hence, the scan thread will never access tuple 17.

The above problem can be solved by letting the log processing threads treat an update to (part of) the index key as a delete/insert hybrid. That is, a reference to the entry in the update-table is entered in the priority queue, but the type of the entry will be insert. The priority queue will be sorted on index keys. As for delete entries, the scan thread will emit the tuple at its position prior to any updates by transactions in the AFTER set. At the same time, the insert type will instruct the scan thread to ignore the index entry of the tuple at its new position.

If the before-image of the index key show that a tuple has already been emitted, or that it originally did not lie within the range of the scan, the entry in the priority queue will not be made. However, the insert entry in the update-table will be made if the index entry is moved to a position ahead of the scan thread and within the range of the query. This is necessary in order to tell the scan thread to skip this tuple when it later runs into the index entry for this tuple.

As long as partial tuple logging is used, entries for write operations differ from delete entries in that they will not always contain the before-image of all attributes. Thus, the scan thread may not always be able to reconstruct the tuple from the entry in the update-table. Hence, if index order should be maintained in the query result, the scan thread may have to read the current version of the tuple when the tuple is to be emitted in order to find the values of attributes that have not been updated. In most cases, this strategy will not significantly affect query performance, since when using secondary index scans, tuple access is unclustered anyway.

In order to avoid reading a tuple twice, the update-table should be checked for
the tuple of an index entry before the tuple is accessed. If the type of the entry in
the update-table is insert, the tuple access will be omitted. The tuple access has
already been or will be made when the tuple was/is at the head of the priority queue.
An alternative to accessing tuples in the priority queue at emit time is to let the
log processing threads access these tuples when the insert entry is made. This will
increase the size of the update-table since the values of unchanged attributes will
have to be stored until the tuples are emitted.

If entries in the update-table are deleted when they are not needed anymore, it
must be both ensured that the tuple has been emitted, and that the current index
entry lies behind the scan thread before this is done. Hence, an insert entry will
either be deleted when the tuple is emitted or when the index entry of the tuple is
read. When an entry has to be kept beyond the time the tuple is emitted, the before-
images of the entry may be deleted, keeping only type and primary key information.
If an entry is deleted, a new entry may be made for the same tuple if it is later
updated. This will never cause the same tuple to be emitted twice since FLP will
not make an entry in the priority queue for tuples whose previous position were
behind the scan thread.

Example

Figure 4.7 shows an example of using a secondary index on the salary attributes
to access the EMPLOYEE relation. The same concurrent updates as used in the
example of Figure 4.5 is assumed. The symbol $E$ is used to indicate that the tuple
given by the head of the priority queue is accessed in order to find values for the
attributes not found in the update-table. The figure also indicates when entries are
deleted from the update-table.

In Figure 4.7 insert entries are made for all log records that contain updates to
salary and references to these entries are made in the priority queue. Log records
that only contain updates to other attributes results in ordinary write entries (log
records 2 and 6).

When log record 1 is processed an insert entry in the update-table is made for
tuple 1 since its index key has changed, and the before-image of the index key (29)
and a reference to the insert entry is entered in the priority queue. When the index
entry for tuple 1 is later read by the scan thread, the insert entry will prevent the
scan thread from accessing tuple 1. When the scan thread has read the index entry
for tuple 14, it checks the priority queue and discovers that the head entry has a
lower index key. Hence, before looking up tuple 14, the scan thread will look up
tuple 1 to get the attributes missing in the update-table. When tuple 1 has been
emitted, the entry in the update-table is deleted since the current value of the index
key gives that the index entry for tuple 1 lies behind the scan thread.

The scan thread never sees the index entry of tuple 17 as illustrated by Figure 4.6.
However, the entry in the priority queue will ensure that the tuple is read and emitted
according to its original position in the index. When the entry for tuple 5 is at the
head of the priority queue, the current value of the index key (37) gives that the
index entry currently lies ahead of the scan thread. Hence, the entry in the update-
table is not deleted, but before-images are removed. When the scan thread later
reads the index entry, the insert entry will prevent it from looking up the tuple, and
the absence of any before-images indicates that the entry can be deleted.
4.3 Extensions and Optimizations of Undo/No-Redo Compensation

Output Comp. Scan | Update-table | FLP
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(16, Ole, 26, 03) ← (16, Ole, 26)</td>
<td>I 1 Karl 29</td>
<td>1 W 1 1 Karl Carl 29 28</td>
</tr>
<tr>
<td>(7, Heidi, 27, 02) ← (7, Heidi, 27, 02)</td>
<td>W 2 02</td>
<td>2 W 2 2 02 01</td>
</tr>
<tr>
<td>(2, Tore, 28, 02) ← (1, Carl, 28, 01)</td>
<td>2 deleted</td>
<td>3 D 1 Heidi 27 02</td>
</tr>
<tr>
<td>(1, Karl, 29, 01) ← (14, Yurog, 31, 03)</td>
<td>I 17 31</td>
<td>4 W 17 3 21 28</td>
</tr>
<tr>
<td>(14, Yurog, 31, 03) ← (14, Yurog, 31, 03)</td>
<td>1 deleted</td>
<td>5</td>
</tr>
<tr>
<td>(17, Bjorn, 31, 03) ← (4, Sophas, 32, 03)</td>
<td>W 16 Ole</td>
<td>5 W 16 2 Ole Ole 21 23 03 01</td>
</tr>
<tr>
<td>(4, Sophas, 32, 03) ← (4, Sophas, 32, 03)</td>
<td>17 deleted</td>
<td>6 W 16 2 Ole Ole 21 23 03 01</td>
</tr>
<tr>
<td>(9, Svein E., 33, 03) ← (9, Svein E., 33, 03)</td>
<td>I 5 35</td>
<td>7 D 1 Bjorn G. 26 04</td>
</tr>
<tr>
<td>(5, Oystein, 35, 03) ← (6, Oddmund, 35, 03)</td>
<td>6 D 1 Bjorn G. 26 04</td>
<td></td>
</tr>
<tr>
<td>(6, Oddmund, 35, 03) ← (6, Oddmund, 35, 03)</td>
<td>8 D 1 Bjorn F. 28 03</td>
<td></td>
</tr>
<tr>
<td>(11, Oystein, 35, 03) ← (5, Oystein, 35, 03)</td>
<td>5 deleted</td>
<td>9 D 1 Bjorn F. 28 03</td>
</tr>
<tr>
<td>(5, Oystein, 37, 03) ← (8, Gunnar, 38, 02)</td>
<td>16 deleted</td>
<td>10 W 16 2 26 31 4 01 04</td>
</tr>
<tr>
<td>(8, Gunnar, 38, 02) ← (8, Gunnar, 38, 02)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(15, Harald, 45, 02) ← (15, Harald, 45, 02)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.7 Example of accessing the EMPLOYEE table through a secondary index on the salary attribute.

In the example of Figure 4.7, entries are not made for tuple that lies behind the scan thread. Hence, no entry is made for log records 3, 5, 8, and 9. When processing log record 6, FLP will make an entry although tuple 16 has already been processed by the scan thread. FLP is not able to determine this since it does not know the salary value for tuple 16. Later, when processing log record 10, FLP discovers that tuple 16 has already been processed and deletes its entry.

If the secondary index used for the scan is based on multi-attribute index keys, the entire index key may not be available in the log record in case of partial updates to the index key of a tuple. In that case, the key used for the entry in the priority queue must be the smallest possible key for this tuple based on the information in the log record and the current position of the scan thread. When this tuple is later read, because it is at the head of the priority queue, the entire index key is constructed and a new entry based on this key is made in the priority queue. Thus, when it is time for the tuple to be emitted, the entry in the update-table will contain all necessary attributes.

With multi-attribute index keys, the alternative approach of letting the log processing threads access the actual tuple before making an entry in the priority queue.

6 Assuming the scan is performed in ascending index key order.
become more promising. With access to the entire index key, the correct position in
the priority queue could then be determined before the entry is made.

4.3.4 Multi-Query Transactions

When several queries are run in one transaction, all queries must see the same state
of the database. That is, the same AFTER set must be used for all queries. This
could be achieved by using the same update-table for all queries of a transaction.
However, the log processing threads may not know which information is needed in
the update-table until a query is submitted. Thus, if a query is submitted some time
after the start of its transaction, a lot of old log records may have to be processed
in order to bring the update-table up-to-date. Much of this log must presumably be
fetched from disk leading to a significant reduction in query performance.

In order to avoid this, one can pre-declare at the start of a read-only transaction
the relations the queries of the transaction will access. Then, log processing could
be started on all relations at once, and all log records can be processed while they
are still available in main memory. A predeclaration of all relations of a database
may be used as default. Since the entries in the update-table may be needed for
later queries, this approach prohibits the use of the space optimization techniques
presented in Section 4.3.2. In addition, the information entered in the update-table
cannot be tailored to specific queries since the properties of the queries are not
known when the log processing is performed. In other words, the update-table may
become very large for long-lasting read-only transactions.

An alternative to predeclaration is to reduce the work that has to be done during
backward log processing. This can be done by only including in the AFTER set
transactions that are dependent on previous queries of this read-only transaction.
That is, at the start of a new query, a transaction will have to be included in the
AFTER set if:

- It updates data accessed by previous queries, or
- It is dependent on transactions that have made updates that were not reflected
  in the query result of previous queries.

Obviously, if a query accesses the same (part of) a relation as previous queries of
the same transaction, this approach will not be wise. In that case, entries of the
update-tables used for previous queries will have to be reentered by BLP in addition
to entries for later log records. On the other hand, if a query accesses other data
than previous queries, less space will be needed for the update-table if transaction
dependencies are moderate. Note that this approach does not guarantee strict query
consistency [BoCa92a]. That is, a query will not necessarily see a serial order of
transactions that is consistent with their commit order.

Figure 4.8 shows an execution of five transactions, T1 through T5, and a query,
Q. Each edge of the dependency graph is labeled with the operations that generated
the edge. Note that the edge between Q and T1 points in the opposite direction of
the order of the operations. This is a result of compensation-based query processing.
Before Q reads X3, the log record for the update by T1 will have been processed by
FLP and the before-image of this update will have been entered in the update-table.
Thus, Q will not see the effects of T1. Assume that Q is the first query of a read-only
transaction that later starts another query, Q', which reads all Y1. In that case, BLP
4.3 Extensions and Optimizations of Undo/No-Redo Compensation

![Diagram of transaction dependencies]

Figure 4.8 An execution of five transactions and one query and the corresponding dependency graph. (Time progressing from left to right.)

of $Q'$ does only need to process log records for $T_1$, since the other transactions are not dependent on $Q$. BLP will inspect the log records of $T_1$, and enter the before-image of the update to $Y_1$ by $T_1$ into the update-table. Note that if $T_3$ had read $X_3$ instead of $X_2$, $T_3$ would depend on $T_1$, and BLP would also have to process the log records of $T_3$ and $T_4$.

Transaction dependencies need not be explicitly represented in order to be able to only roll back transactions that are dependent on a previous query. It is sufficient to establish which transactions must be serialized after a previous query. This can be done using a two-coloring algorithm. For each concurrent multi-query transaction, both transactions and tuples have a color, which may be either black or white. In addition, colors may be assigned to a subset of a relation as specified by a primary or secondary key value interval. When a read-only transactions starts, the color of all tuples in the database will be white. When a query of this read-only transaction starts, a key interval that covers all tuples that will be scanned by this query is colored black. Later coloring of tuples and transactions will follow the following rules:

1. Transactions that updates tuples within a black interval are black.
2. All tuples that are accessed by black transactions will be colored black.
3. Transactions that have accessed at least one black tuple are black.

When a later query in the same read-only transaction is started, all black transactions will be included in the AFTER set together with all currently active transactions before BLP starts.\footnote{Unlike for single-query transactions, there may exist dependencies between transactions to be processed by BLP. Thus, BLP must make sure to process log records for a single tuple in correct sequence.}

Rule 3 above implies that the color of a transaction cannot be determined until commit time. Thus, the coloring of tuples will be part of commit processing. When releasing a lock, the color of the corresponding tuple will be set to black. To represent
the colors of tuples a bit vector can be used. Each bit of the bit vector represent the
color of one tuple. Each concurrent multi-query transaction will need its own bit
vector. Black transactions should not be removed from the transaction table until
the corresponding multi-query transaction is finished. That way, BLP could find a
reference to the last log record for this transaction. Thus, BLP will not need to look
at all log records in order to find the log records for black transactions. The work
for BLP could be even further reduced if each transaction contained an associated
bit vector representing the relations that have been updated by these transactions.
That way, BLP could ignore transactions that have not updated the base relation(s)
of the query.

The coloring algorithm given by the rules above will regard two update transac-
tions as dependent if they have read the same tuple. This is not necessary in order
to achieve transaction-consistency. It is possible to be able to ignore read→read “de-
pendencies” by using three or four colors for each tuple [Amm+95, PiKa92]. This
will reduce the number of black transactions, at the expense of using more space for
representing the colors of tuples.

The rules above assumes that color information is also maintained for tuples that
are covered by an interval color. This could be avoided if Rule 1 is changed so that
also read operations within the interval colors the transaction black. Then, the color
of a tuple would always be the same as that of the corresponding interval. Otherwise,
a transaction will not turn black by reading a tuple within a black interval. However,
this would introduce read→read “dependencies” between queries and transactions.
Since queries usually reads a lot of tuples, this would substantially increase the
number of dependent transactions. For example, $T_2$ of Figure 4.8 would be regarded
as dependent on $Q$, and $T_2$ and $T_5$ would have to be processed by BLP of later
queries.

Note that it is necessary to maintain color information for all tuples in the sys-
tem, not just tuples accessed by the different queries of the read-only transactions.
Consider the example of Figure 4.8, if $T_3$ is black, the color of $Z_1$ is needed in order
to determine that $T_4$ should also be black. Otherwise, a query that reads $Z_2$ will
see the effects of $T_3$ even if $T_4$ is dependent on earlier queries of the same read-only
transaction.

Instead of representing dependency at tuple granularity, a coarser granularity
(e.g., block) may be used. Coarser granularity of dependencies will result in more
work for BLP since more transactions will be regarded as dependent on previous
queries. The savings are that less space is needed to represent color information of
data items.

Whether it will be cost-effective to apply the coloring algorithm described above
in order to reduce work by BLP, will depend on the types of queries and transactions
that are executed in a database system. If the number of transaction dependencies
are high, many transactions will have to be processed by BLP. In that case, it

---

8In order to reduce space requirements, a bit may represent several tuple by using a hashing
function. The expense is that “false” dependencies are introduced.
9Each bit may represent a single relation or a group of relations.
10Interval colors are needed in order to avoid phantom problems. If only tuples were colored,
insertions into an interval covered by an earlier query, would be reflected in the results of later
queries.
11The granularity may be coarser than the one used for concurrency control (locking). If this is
the case, not all the determined dependencies will be real dependencies.
4.3 Extensions and Optimizations of Undo/No-Redo Compensation

Table 4.3 Processing of delta operation log records.

<table>
<thead>
<tr>
<th></th>
<th>FLP</th>
<th>BLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current entry</td>
<td>update</td>
<td>update</td>
</tr>
<tr>
<td>Before-image</td>
<td>ignore</td>
<td>ignore</td>
</tr>
<tr>
<td>Operand</td>
<td>undo</td>
<td>subst.</td>
</tr>
</tbody>
</table>

will probably be more cost-effective to maintain update-tables for all queries from the start of the multi-query transaction. Also note that FLP has a lower cost for processing log records than BLP since FLP will never need to read log records from disk. In addition, if the read-only transaction is very long, log records for some black transactions may already have been garbage-collected. Further more, the coloring algorithm will require extra work by transactions.

Further studies are needed in order to determine when the use of the coloring algorithm is cost-effective. Also, an investigation of the trade-offs between the granularity of dependency representation, the number of colors used for data items, and space requirements is needed.

4.3.5 Delta Operations

If the locking model supports delta locks combined with delta operations, rules for entering information into the update-table become more complex. Since log records for delta operations do not contain a before-image, the operand of the operation will be entered instead if the update-table contains no previous value for the attribute of the delta operation. When the scan thread later finds this operand in the update-table, the corresponding delta operation is undone on the current tuple as part of the compensation.

An entry in the update-table will contain either a single operand or a single value for each attribute that has been updated by a transaction in the AFTER set. When several log records representing delta operations are performed for the same tuple, the delta operations will be aggregated (e.g., added) into a single operand. When both non-delta and delta operations are performed against a tuple, the attribute values in the update-table will be a result of undoing earlier delta operations on the before-images of the oldest non-delta updates.

If the update-table contains a before-image while doing forward log processing, or the log record contains a before-image while doing backward log processing, the actions of a log processing thread will be similar to when delta operations are not supported (Ref. Table 4.1). In other words, later operations are ignored while earlier operations cause a substitution of the content.

If the update-table contains a delta operation operand and a log record for a write operation is processed by FLP, the delta operations are undone on the before-image, and the resulting value stored in the update-table. Similarly, if a delta log record is processed by BLP, and the update-table contains a before-image, the delta operation will be undone, and the resulting image stored in the update-table. In both cases, an older before-image is restored by undoing delta operations.

Table 4.3 summarizes what the log processing threads will do when processing either (non-delta) updates or delta operations. As described above, the decision is
based on whether the old entry of an attribute contains a before-image or an operand. 
ignore means that the old before-image is kept. subst. means that the before-image of 
the current log record is substituted for the old before-image/operand(s). aggr. 
means that the operands of the current log record and the old entry is aggregated. 
undo means that an older before-image is created by undoing the delta operation 
on the before-image. When deciding the type of an entry, delta log operations will 
be treated as ordinary write operations (Ref. Table 4.1).

The rules of Table 4.3 will not be correct if log records of delta operations that 
are executed after a tuple is read by the scan thread, is processed by FLP before the 
scan thread checks the update-table for this tuple. Then, the resulting tuple may be 
incorrect since operations that were not reflected in the tuple could be undone. This 
will not be a problem when the old entry contains a before-image since the operand 
in that case will be ignored by FLP. Neither is it a problem for non-delta update 
operations since the before-images of unreflected operations will be identical to the 
 corresponding attribute values of the tuple. However, if several delta operations are 
performed to a tuple, it must be ensured that only reflected operations are undone. 
This could be achieved in several ways:

- Each time the scan thread is suspended because more log processing is nec-
  essary, it is rescheduled when FLP has processed the log record given by the 
  state identifier of the current tuple.

- FLP should check the state identifier of the current tuple each time its current 
  log record represent a delta operation to this tuple. If the LSN of the current 
  log record is greater than the state identifier of the current tuple, FLP should 
  be suspended until the scan thread has finished this tuple.

- Operands are not aggregated but stored separately in the update-table together 
  with the LSN of the corresponding log records. The scan thread could then by 
  itself decide which operands were already applied when the current tuple was 
  read.

If the entry in the update-table will be kept for later queries, the scan thread 
may substitute the attribute values of the compensated tuple for the operands of 
the update-table. That way, all later log records for updates to these attributes of 
the tuple could be ignored by FLP.

4.3.6 Complete Tuple Logging

So far, it has been assumed that partial tuple logging is used. If, on the other 
hand, complete tuple logging is used, the before-image of the entire tuple will be 
stored in a log record. This way, the log processing threads will have access to all 
attributes of a tuple, and as will be shown in Chapter 5, this can be exploited to 
optimize query execution. Moreover, FLP will always be able to decide whether the 
tuple of a log record lies behind or ahead of the scan thread when base relations are 
scanned in a predefined order. Thus, the space optimization techniques presented 
in Section 4.3.2, could be fully exploited.

Complete tuple logging also simplifies the maintenance of the update-table. All 
attributes of a tuple will be available in the first log record processed by one of the
log processing threads. Hence, only BLP needs to process more log records for this tuple.

If complete tuple logging is used when running queries based on secondary index scans, an entry in the priority-queue will always contain sufficient information for emitting the tuple. Thus, the scan thread will not have to access the current version of this tuple. In addition, the insert entry in the update-table will cause the scan thread to skip reading the tuple after it has been emitted. Thus, complete tuple logging will reduce the I/O cost for queries based on secondary indexes. With frequent updates to attributes that are part of the index key, this could give significant performance improvements. In fact, as long as the update-table resides in main memory, it will probably be more efficient to check the update-table for every index entry. Then, access to actual tuples will only be necessary for tuples that have not been updated by transactions in the AFTER set.

Using complete tuple logging instead of partial tuple logging, will increase the log volume. This will affect transaction performance since it will increase disk I/O in connection with commit processing and flushing of data pages. The increased log volume will also slow down recovery. Using complete tuple logging to improve performance of compensation-based query processing, has similarities to transient versioning. Also transient versioning requires that transactions write extra data. However, when using transient versioning an extra page will usually have to be accessed for each update in order to store the old version. When using complete logging instead of partial logging most updates will still go to a single log page.

A database system may be designed so that it is possible to switch between partial and complete logging on request. This way complete tuple logging may be turned on when it will lead to significant improvements in query performance. If complete tuple log records are needed for BLP, query processing will have to be delayed until all transactions that were active before complete tuple logging was initiated have terminated. If complete tuple logging is only needed in order to determine whether a tuples lies ahead of or behind the scan, query processing can start immediately.

If undo/no-redo compensation should be able to exploit complete tuple logging also when delta log records are used, the entire before-image of a tuple must also be included in delta log records in addition to the operand(s) of the delta operation. (Normally, delta log records do not contain before-images.) The before-images of delta log records will only be used by compensation-based query processing and not for transaction recovery. If delta log records contain before-images, FLP could handle delta log records as ordinary log records and enter the before-image instead of operands into the update-table. BLP must still enter the operands since the before-images will not reflect delta operations that were later performed by transactions that are not members of the AFTER set.

4.3.7 Compatibility with Other Concurrency Control Methods than Strict 2PL

It has so far been assumed that update transactions are executed using strict 2PL. That is, a transaction will release no locks before it terminates. While strict 2PL is the by far most used concurrency control method, many other methods have been proposed [Ber+87]. In this section, it will be briefly discussed which modifications are needed to run the algorithm for undo/no-redo compensation when using other
transaction schedulers than strict 2PL.

Strict 2PL has also been called rigorous 2PL [Bre+91] since it is more conservative than necessary to enforce strictness. Rigorous transaction schedulers will not allow any conflicting operations by two uncommitted transactions. This means that serialization order of transactions will be equivalent to their commit order. For non-rigorous transaction schedulers, a committed transaction may be dependent on a still active transaction. Thus, when running the algorithm for undo/no-redo compensation where only transactions that are active at the start of a query are included in the AFTER set, members of the BEFORE set may depend on transactions in the AFTER set when non-rigorous concurrency control is used. As long as the execution of transactions is rigorous, the algorithm will be correct regardless of whether the transaction scheduler is based on 2PL, timestamp ordering, serialization graph testing (SGT), or other concurrency control algorithms. Below it will be described how strong query consistency could be achieved for different types of non-rigorous transaction schedulers.\[12]

In order for the undo/no-redo compensation to be correct for non-rigorous schedulers, transactions that have committed before the start of a query must also be included in the AFTER set when they are dependent on transactions that have been included in the AFTER set. For 2PL-based schedulers, this is avoided by including in the AFTER set only transactions that have not released any locks at the start of the query. No transaction could depend on a transaction that have not release any locks. This means that the scan thread should not start until all transactions that has started releasing locks have terminated. Until then, new transactions will be added to the AFTER set when they start, and FLP will only process log records of transactions in the AFTER set.

For scheduler based on timestamp ordering, a transaction may only depend on transactions with a lower timestamp than its own. Thus, all committed transactions that should be included in the AFTER set will have a higher timestamp than the lowest timestamp of a transaction that is still active. In other words, a single timestamp can be used to represent the AFTER set. BLP will process log records for all transactions with a larger timestamp.

If the transaction scheduler is based on SGT, the serialization graph could be directly used to determine which committed transaction are dependent on active transactions. This way, only committed transactions that are actually dependent on active transactions will be included in the AFTER set. Thus, the AFTER set when using SGT will usually be smaller than when using timestamp ordering.

The algorithm for undo/no-redo compensation is based on using before-images of log records for compensation. Logging schemes based on before-images requires strict execution of transactions. That is, uncommitted data will never be read or overwritten. Otherwise, recovery could not be based on before-images since restoring the before-image of an update may result in removing the effects of other updates that should not be undone [Ber+87]. In such cases, logging must be based on logical operations. For such systems, the log processing threads could enter the logical operations into the update-table as described in Section 4.3.5 for delta operations. An alternative to log processing could be to let transactions enter information on their updates into the transaction table. That way, the scheme using before-images could be kept since it is made sure that all dependent transactions will also be

\[12\text{Strict query consistency is not possible when serialization order differ from commit order.}\]
members of the AFTER set.

Since strict schedulers do not allow access to uncommitted data, transactions in the BEFORE set could only be read-write dependent on transactions in the AFTER set. This means that the algorithm for undo/no-redo compensation will guarantee update query consistency [BoCa92a] when transaction execution is strict.

As have been shown above, the proposed algorithm for undo/no-redo compensation is not restricted to systems using strict 2PL. However, the rest of this thesis will assume strict 2PL since this is the most widely used concurrency control method for database systems.

### 4.4 Storing the Update-Table on Disk

The update-table may become too large to be able to keep it entirely in main memory. For long queries in database systems with high update rates, this may occur even when using the space optimization techniques presented in Section 4.3.2. This section presents methods for storing (parts of) the update-table on disk.

The main goal when designing methods for storing the update-table on disk will be to minimize disk I/O. Most importantly, FLP should not have to access disk in order to check whether an entry for the tuple of the current log record already exists. Otherwise, FLP may not be able to keep up with the rate at which log records are produced. In addition, the number of disk accesses required by the scan thread should be kept as low as possible.

Below, two different methods for storing the update-table on disk are presented. The first method uses hashing to partition the entries of the update-table between disk blocks. The second method range-partitions the update-table in order to further reduce the amount of disk I/O. This method requires that all log records contain the information needed by FLP in order to decide in which range the corresponding tuple is located. For both methods it is assumed that the entire update-table could reside in main memory until BLP is finished.

#### 4.4.1 Hash-Based Partitioning

In order to store an update-table on disk, its entries must be partitioned between disk blocks. Since access to the update-table is hash-based, it will be reasonable to base the partitioning on hashing. That way, by hashing on primary key it can be determined which disk block may contain the entry of a tuple. The hashing scheme should allow the number of disk blocks used to vary with the size of the update-table. The hashing scheme used below will be based on linear hashing [Litw80], but other dynamic hashing schemes could have been used instead [Fag79, Lars78, Mart82].

**Linear Hashing**

As other hashing schemes, linear hashing assigns the records of a table to specific buckets by applying a hashing function to the keys of the records. When the table is stored on disk, each bucket usually represent a specific disk block. Consider a hash table originally consisting of \(N\) buckets with addresses \(0, 1, \ldots, N-1\). Linear hashing increases the address space (i.e., the number of buckets) gradually by splitting the buckets in a predetermined order: First bucket 0, then bucket 1, and so on, up to
and including bucket $N - 1$. Splitting bucket $n$ involves moving approximately half of the records to bucket $n + N$. When all $N$ buckets have been split so that the hash table contains $2N$ buckets, the splitting process starts over again from bucket 0. If the size of the hash table decreases, the last created bucket may be removed by moving its records back to the bucket that was used for these records before their current bucket was created.

In order to compute the correct address for records that originally belong to a bucket that has later been split, a dynamic hashing function is used. The hashing function will be based on the following sequence of hashing functions:

$$h_j(K) = g(K) \mod (2^j N),$$

where $g$ is a normal hashing function such that $h_0$ is the hashing function used when the hash table contains the original $N$ buckets. It follows that the address space of $h_j$ is $2^j N$ buckets. Because of the predetermined order of splitting, the dynamic hashing function $h$ could be computed without the use of tables:

$$h(K) = \begin{cases} h_j(K) & \text{if } h_j(K) < M, \\ h_{j-1}(K) & \text{otherwise.} \end{cases}$$

where $M$ is the current number of buckets and $j = \lfloor \log_2 \frac{M}{N} \rfloor$. That is, $j$ represent the number of times bucket 0 has been split.

The predetermined order of splitting implies that a bucket (i.e., a disk block) may become full before it is time to split this bucket. One solution to this problem would be to split all buckets preceding this bucket. However, this would give very low utilization of disk blocks. In the worst case, inserting one extra record would double the number of buckets. Instead, a record that do not fit in the bucket given by $h(K)$, will be stored in another bucket, the overflow bucket for this record. Such a record will be called an overflow record. In this presentation, it is assumed that overflow records is handled through an approach called recursive linear hashing [RaSa84]. However, other strategies for handling overflow records could have been used [Litw80, Mull81, Lars82, RaSa84, Lars85]. More details on recursive linear hashing will be presented below.

Different strategies could also be used as to when to split or remove a bucket. One strategy could be to split a new bucket each time an overflow record is created. However, analysis show that this will give an average bucket load of only about 60% [Litw80]. By requiring that the bucket load should exceed a threshold before a split is performed, average bucket load may be increased to 90% without significantly affecting the cost of a lookup [Litw80]. Even better performance may be achieved by using partial expansions [Lars80]. This technique reduces the variation in number of records per bucket. This improves performance by reducing the number of overflow records. The algorithm presented below could easily be extended to use partial expansions.

Since bucket size is determined by disk block size, the average number of records in a bucket may not be optimal for a hash table residing in main memory. This may be solved by organizing the tuples within a bucket into several linked lists. If the bucket of a record is determined using the hashing function $h_j$, the chain containing the record could be found using the hashing function $h_{j+i}$ when the bucket contains $2^i$ chains. Having two or more chains within a bucket will reduce the work when
4.4 Storing the Update-Table on Disk

![Recursive linear hashing diagram](image_url)

**Figure 4.9** Recursive linear hashing.

splitting a bucket since the records that should remain in the old bucket and those that should be moved to the new bucket will be located in separate chains. The extra expense of such a scheme is the cost of evaluating another hashing functions. However, this cost could be made smaller by choosing $N = 2^n$, where $n$ is an integer. In that case, all the extra work could be achieved by bit shifting [Litw80].

**Recursive Linear Hashing**

The basic idea behind recursive linear hashing is to also use linear hashing for overflow records [RaSa81]. This means that overflow records are entered into a separate hash table implemented by linear hashing. This second hash table may have its own overflow records that will be inserted into a third hash table, and so on. It turns out in practice that rarely more than three levels are needed [RaSa81]. Recursive linear hashing may be organized as an array of files, each file containing the overflow records for the file at the level below.

Figure 4.9 shows an example of a hash-table using recursive linear hashing. To simplify the example, $N = 1$ and $g(K) = K$ in (4.1). Thus, the hashing functions currently in use are $h_0(K) = K \mod 8$ and $h_4(K) = K \mod 16$. The hash-table consists of three levels, and each bucket have room for 3 records. The next bucket to be split at the lowest level is bucket 2. The reason it has not yet been split even if several overflow records exist, is that the average load of buckets at level 1 is not particular high.

When a new record is inserted into the hash table, an attempt is first made to place the record in the first level. If the bucket given by the hashing function is full, an attempt is made to place the record at level 2. This process is continued until a level is found where the record could be placed. In the example above, when record 59 was inserted, both bucket 3 at level 1 and bucket 1 at level 2 were full. Thus, another level was created for record 59.

When the average load of buckets at level $i$ exceeds a predefined threshold, another bucket is added to this level and records are moved to this new bucket from the bucket whose turn it is to be split. In addition, those records at higher levels
that belong to the chain of overflow buckets for the buckets that have been split, is
collected and redistributed at a lower level if possible. This splitting process should
be designed so that only full buckets will ever have overflow records. This way, a
search can be terminated as soon as a non-full bucket is encountered.

In the example of Figure 4.9 bucket 2 is the first to be split. When this occurs,
record 42 will be moved to the bucket 10. Likewise, when bucket 3 is later split
records 27 and 75 will be moved to bucket 11. Then, bucket 1 at level 2 will be
inspected for tuples belonging to buckets 3 or 11 at level 1, and records 19 and 67
are moved to bucket 3. Finally, bucket 0 at level 3 is checked, and record 59 is moved
to bucket 11 at level 1. Since level 3 is now empty, bucket 0 is dropped, and the
hash-table will consist of only two levels. In addition, the average bucket load at
level 2 is probably so low that bucket 2 will be dropped.

Update-Table Based on Recursive Linear Hashing

The update-table will be partitioned into buckets (blocks) using recursive linear
hashing, and the buckets may either be stored in a main-memory buffer or on disk.
A buffer pool of fixed size may be allocated to each query, or the update-tables of
all queries may share a common buffer pool. If a bucket is to be split or read from
disk, a bucket in the buffer may have to be flushed to disk in order to make room
for the new bucket. Efficient buffer replacement strategies will be briefly discussed below.

As mentioned above, FLP should not have to access disk even if part of the
update-table is stored on disk. When applying linear hashing for the implementation
of a disk-based update-table, this is achieved by treating all entries to a bucket that
resides on disk as overflow records even when the bucket is not full.

Recursive linear hashing will be used to store overflow records. The advantage of
recursive linear hashing over associating overflow records with their buckets through
hashing, is that the original bucket need not be updated when an overflow occurs.
Thus, an overflow record may be inserted without fetching the block of its original
bucket from disk.

Since FLP will not check whether an entry already exists for a tuple when the
corresponding bucket resides on disk, the update-table may contain several entries
for the same tuple. Each entry in the update-table will contain a state identifier that
tells the scan thread which log records have been reflected in this entry. Each time
FLP updates an entry in the update-table, it sets the state identifier of the entry to
the LSN of the current log record. This state identifier will inform the scan thread
whether it needs to look for another entry for this tuple in order to compensate for
all updates.

After having read a tuple, the scan thread uses the primary key of the tuple as
input to the hashing function given by (4.2). If the results refer to a bucket that
resides in main memory, this bucket is inspected for an entry for the current tuple.
If the bucket resides on disk, it must first be read into main memory. If the bucket
contains an entry for the current tuple, the state identifiers of the tuple and the entry
are compared. If the state identifier of the entry is the smallest, this tuple has been
updated while the bucket resided on disk. In that case, the corresponding overflow
bucket at the next level must be inspected for an entry for the current tuple. If no
entry is found, or the state identifier of this entry also is smaller than that of the
tuple, the process is repeated at the next level.
4.4 Storing the Update-Table on Disk

Since entries is put into an overflow bucket if the corresponding buckets at lower levels resides on disk, non-full buckets may have entries residing in overflow buckets. Thus, unlike ordinary recursive linear hashing, a search for an entry can not stop at the first level where a non-full bucket is encountered. The entry could still exist in an bucket at a higher level. However, this will probably not have major impact on the performance of lookups in the update-table. For successful searches, the state identifier of the tuple will tell when to stop searching for more entries. The number of unsuccessful searches will usually be low since the state identifier of a tuple in most cases can be used to decide whether an entry may exist. Unsuccessful searches may only occur for tuples that was updated by transactions in the BEFORE set after the first transaction in the AFTER set had started.

When the scan thread accesses an overflow bucket, it will try to move its entries to buckets at the level below if the corresponding bucket reside in main memory and there is available space. If a bucket and its overflow bucket both contain an entry for the same tuple, the two entries will be combined into a single entry.

The use of state identifiers as discussed above, requires that a state identifier is maintained for each tuple. If only blocks have state identifiers, this state identifier will usually be larger than the LSN of the last log record of a tuple. Thus, all lookups will have to access all levels. The number of levels that need to be accessed could be reduced by using a bit vector for each level with one bit per bucket. A bit will be set if the corresponding bucket has an overflow bucket. This scheme could generally be used to reduce the cost of unsuccessful searches.

At a given level, a new bucket will be split when the average load of the buckets at this level exceeds a threshold. The bucket to be split may have to be read from disk. However, FLP does not have to wait for the splitting to finish. It may just treat entries to this bucket as overflow records until the split has been performed. Overflow buckets may have to be fetched from disk in order to transfer overflow records to the new space that has become available as a result of the split. If the average bucket load at a given level goes below a predefined minimum, the last bucket created at this level will be removed after its records have been moved to the corresponding bucket. Note that the deletion of a bucket may also produce overflow records.

In order to avoid that FLP must wait for disk accesses, a background process should make sure to flush buckets to disk so that there will always be buckets that may be replaced without further flushing. This way FLP may create new overflow buckets without delay. The flushing of disk pages will normally not become a bottleneck since both data pages and log records must also be flushed to disk. Thus, disk I/O will prevent the update rates from becoming so high that FLP will insert new entries into the update-table faster than they could be flushed to disk. In addition, FLP will only enter information on updates that are relevant the query.

Several strategies for replacing buckets could be used in order to reduce the total cost of disk I/O:

- Try to keep top level buckets in main memory. This way an extra level is only created when a bucket at the top level is full, and not because this bucket resides on disk. By limiting the number of levels, the cost of a search is kept low.

- Replace the fullest buckets first. This reduces the total number of buckets
since a higher load average is achieved.

- Avoid replacing buckets that will soon be split. This reduces the probability of having to read buckets from disk during a split.

These approaches may conflict. For example, the buckets that will soon be split will probably contain more entries than average. Thus, further studies are needed in order to find the right combination of these strategies. Also, an analysis should be performed in order to find how performance is affected by the strategy of placing entries to buckets residing on disk in overflow buckets.

4.4.2 Range-Based Partitioning

Using linear hashing to store the update-table on disk, requires random disk accesses by the scan thread to find entries in the update-table. If the scan thread could be restricted to sequential disk accesses, the I/O cost of storing the update-table on disk could be significantly reduced. The idea behind range-based partitioning of the update-table is to keep the part of the update-table that will first be accessed in main memory. This method requires that FLP is able to decide the scan order positions for tuples for all log records. In other words, the scan must either be a primary index scan, or a secondary index scan where complete tuple logging is used. The term scan key will be used below for the combination of attributes that determines scan order.

Range-based partitioning will divide the domain of the scan key into $N$ ranges. For each range, a separate buffer will be used to store entries of the update-table. A buffer may consist of several disk blocks. Within each buffer the entries will be organized in a hash table. If dynamic hash tables are used [Lars88], this should not give much overhead compared to a single hash tale for the entire update-table. Figure 4.10 shows an update-table that is divided into 5 ranges, each having its own buffer and hash-table. The figure assumes that the scan key is equivalent to the primary key. Generally, the key used for range-partitioning may be different from the key used for accessing the hash-table.

When no more space is available for the update-table in main memory, the buffer
with the largest scan key values is stored in a file on disk. The memory space allocated for this buffer is then released and can be reused by other buffers. A single block is kept in main memory for later entries to this range. Each time this block goes full, its content is appended to the file of this range. Each time the update-table runs out of memory a new buffer will be flushed to disk.\footnote{Assuming the scan is performed in ascending scan key order.}

When the scan thread has processed all tuples in a range, the buffer for this range will be flushed to disk if the entries will be needed for later queries. Otherwise, the remaining entries for this range are deleted.\footnote{Actually, the flushing should start before the update-table runs out of memory. That way, it can be ensured that there will exist blocks that can be replaced without further flushing. Thus, FLIP need not have to wait for disk flushing in order to get the space needed for inserting a new entry.} Before the scan thread can start processing the new range, all entries in the update-table for this range should be read into main memory.

If the scan thread deletes processed entries, part of the space occupied by entries of the current range could be released before this range is finished. That way, the entries of the next range could be prefetched from disk so that all entries are present in main memory when the scan thread starts working on this range. If entries are not deleted after being accessed by the scan thread, optimal performance is achieved by requiring that no range will occupy more than half the available memory space. That way, the entries of previous range could be written to disk and the entries of the next range could be prefetched while the current range is scanned.

The addressing scheme for the hash tables of each range should be designed so that it will not be necessary to rebuild the hash table when reading the entries from disk. However, the entries appended after the range was first flushed to disk must be inserted into the hash table. For the tuples of some of these entries, an entry may already exist in the update-table. In that case, the entries are combined as described earlier.

When the scan key is different from the primary key, an update may change the range of a tuple. In such cases, the insert/delete hybrid described for secondary index scans, should be inserted in the range determine by the after-image of the update. However, the range of the before-image must be checked for an existing entry for this tuple. If a previous entry exists, the new entry will be a combination of the old entry and the before-image of the log record.

The range-partitioning should be chosen so that the total entries for a range will fit in main memory. Systems that have histogram information available for query optimization could use this when deciding partitioning. However, the number of tuples within a range does not directly determine the size needed for the update-table. It is the distribution of updates during the scan that determine the size. Thus, extra information about access pattern will help. If the entries of the update-table is only needed for a single query, the probability of an entry in the update-table will be highest for tuples with a large scan key. In this case, a non-uniform range-partitioning will be best.

If the size of the update-table for one range is too large to fit in main memory, the range can be split on demand. This can be done by reading the entries of the range from disk and writing the entries of the largest subrange back to disk.

By using the algorithm for range-partitioning as presented above the cost of
storing the update-table on disk could be significantly reduced compared to a hash-based partitioning. However, the method requires that FLP is able to determine which range a log record belongs to. In other words, if a secondary index is used for the scan, either full tuple logging must be used or the update-table must be stored using hash-based partitioning.

4.4.3 Alternative methods

Instead of using a dynamic hashing method like linear hashing, a static hashing scheme could be used. That is, the update-table will be partitioned into a fixed number of buckets. To handle overflow records, a chain of disk blocks could be associated with each bucket. When the block at the head of such a chain is full, a new empty block will be inserted as the head of the chain. In order to avoid that FLP will have to access disk, at least the head of each chain will have to be stored in main memory. In other words, the number of buckets should be smaller than the number of disk blocks that could reside in main memory. This means that the amount of memory available for the update-table must be determined at query startup.

Another alternative method is inspired by the on-page caching suggested for transient versioning [BoCa92b]. This way, a portion of each data block will be reserved for update-table entries referring to tuples of the block. As long as FLP does not fall far behind the head of the log, a block will still reside in main memory when needed by FLP. Compensation is also made easier since a tuple and its entry in the update-table will reside in the same block. This scheme also makes it easy to let several concurrent queries share the same entry. That way, the space requirements could be reduced compared to when each query maintain separate update-tables. Several entries may still exist for a single tuple since different queries may require different versions of a tuple. The entries should be tagged with the query IDs of the queries that may access it. An entry could be garbage-collected when all its queries are finished with the block.

When there is no more space for entries in a data block, entries for this block must be stored in separate storage. Access to such overflow entries could be supported through chaining. Another alternative is to handle overflow entries with the method based on linear hashing as presented above. However, one of the main advantages of the on-page versioning scheme is that the implementation of a separate buffer management strategy for the update-table is not necessary. The main disadvantage with this scheme is that the buffer hit ratio for transactions will decrease since data will be distributed across more disk blocks.

4.5 Redo-Based Compensation

As discussed in Section 4.1, compensation may consist of both undo and redo operations. So far, only algorithms for undo/no-redo compensations has been described. This section will briefly describe alternative strategies that involve redo compensation. First, strategies for redo-only compensation will be discussed. Then, strategies that involve both redo and undo compensation are discussed.
4.5 Redo-Based Compensation

End of scan

\[ \text{start}_1 \ r_1 \ [x] \ \text{start}_2 \ r_2[y] \ \text{commit}_2 \ \ r_1[y] \ w_1[x] \ \text{commit}_1 \]

**Figure 4.11** A transaction in the BEFORE set, \( T_1 \), is dependent on a transaction that is started after the end of the scan.

### 4.5.1 No-Undo/Redo Compensation

When no-undo/redo compensation is used, all transactions that are active during the scan will be included in the query’s BEFORE set. All operations that are not reflected in a tuple when it is read by the scan thread must be redone before the query could emit this tuple. This means that a tuple may not be emitted until all transactions in the BEFORE set have terminated. In other words, the query could not emit any tuple until after the scan has finished.

As shown by Figure 4.11, a transaction that is concurrent with the scan may depend on a transaction that is started after the scan is finished. Thus, some transactions that start after the scan may also have to be added to the BEFORE set. The easiest way to guarantee transaction-consistency, will be to include in the BEFORE set all transactions that commit before any transaction that is concurrent with the scan. That way, it is not necessary to keep track of transaction dependencies. In addition, strict query consistency is achieved since queries will see a serial order of transactions that are consistent with their commit order.

The redo of operations made by transactions in the BEFORE set could be done in two ways. One approach is to apply each log record to the corresponding tuple when the log record is processed. The main problem with this approach is that if the base relation(s) of the query is too large to fit in main-memory, the redo operations will include random accesses to disk. In addition, each disk block data may have to be fetched from disk several times since there may be several updates to the same block.

An alternative approach is to collect relevant information from log records during the scan and perform all the compensation after all transactions in the BEFORE set has terminated. That way, disk I/O could be reduced by performing compensation on tuples in sequential order. Using this scheme, a query will consist of two phases: a scan phase and a compensation phase (Figure 4.12).

During the scan phase, a scan thread will read the tuples of the base relation(s) without setting any locks. Since the tuples of the query result will not be emitted

![Figure 4.12](image-url) The two phases of no-undo/redo compensation.
until after the scan phase is finished, the tuples are stored in temporary relations. Concurrently with the scan, a forward log processing thread (FLP) will process log records while they are still in main memory.\textsuperscript{16} For each log record representing an update to the base relation(s), FLP will extract the after-images of relevant attributes for later use.

When all the transactions in the BEFORE set have terminated, the compensation phase starts. As mentioned earlier, compensation can be viewed as a join of the base relation(s) and the log. As for traditional joins, there are several choices as to how this join could be executed. Srinivasan and Carey use a sort-merge approach [SrCa92a]. A sort-merge approach could also be used for log-based compensation. During the scan phase, FLP will append relevant after-images to an update-list. At the start of the compensation phase, the entries of the update-list is sorted so that the ordering is equivalent to scan order. For example, if a base relation is scanned in primary key order, the entries of the update-list will be sorted on primary key. If a secondary index is used, the entries of the update-list will be sorted on the corresponding index key.\textsuperscript{17}

In the compensation phase, the temporary relation and the sorted update-list can be merged to produce a transaction-consistent query result. If a tuple occurs in both the temporary relation and in the update-list, the last after-image for each attribute in the update-list will be used instead of the values read during the scan. Delete entries in the update-list will cause the corresponding tuple in the temporary relation to be ignored. Insert entries will add tuples to the query result. If a secondary index is used for the scan, updates to the index key should be entered in the update-list as a delete/insert pair. This will cause the corresponding tuple to be removed from its old position in the temporary relation and inserted at its new position.

Instead of the sort-merge approach, a hash-based update-table similar to the one used for undo/no-redo compensation could be used (Figure 4.13). In the scan phase, FLP will enter after-images of relevant updates into the update-table. A new after-image will substitute any previously entered value for the same attribute. Thus, the update-table will contain only one entry for each tuple. In the compensation phase, the temporary relation will be scanned, and for each tuple, the update-table will be checked for any entries. If an entry is found, the tuple is emitted using the after-images in the update-table instead of the attribute values in the temporary relation. If no entry is found, the tuple is emitted as found in the temporary relation. In order to preserve scan order, a priority queue similar to the one used for undo/no-redo compensation could be used for insert entries.

Further studies are needed in order to determine which is the most efficient of the sort-merge and the hash-based approaches. However, based on other comparisons of sorting and hashing, it is likely that hashing will outperform sorting. In addition, an update-table will be smaller than the update-list since there will be maximum one entry for each tuple. Thus, it is more likely that the update-table could be stored entirely in main memory than that the entire update-list could be stored in main memory during sorting. Note also that log records for tuples that have not yet been

\textsuperscript{16}Since no undo compensation will be performed, there will be no need for backward log processing.

\textsuperscript{17}Generally, full tuple logging is needed in order to be able to base the sorting on a secondary key. If partial tuple logging is used, both the temporary relation and the update-list will have to be sorted on primary-key before merging. Then, another sort is required if scan order should be preserved in the output.
4.5 Redo-Based Compensation

![Diagram of scan and compensation phases]

**Figure 4.13** No-undo/redo compensation when using an update-table.

read by the scan thread, may be ignored by FLP since the update will be reflected in the tuple when it is later read by the scan thread.

As mentioned above, FLP continues processing log records until all transactions in the BEFORE set have terminated. All transactions that are active during the scan phase will be members of the BEFORE set. At the end of the scan, a BEFORE set containing all presently active transactions will be created. Log processing will continue until FLP has processed commit or abort log records for all transactions in the BEFORE set.

Transactions that are active when all transactions in the BEFORE set have terminated, should not be reflected in the query result. FLP will record the active transactions at the end of log processing in an AFTER set. Log records of transactions in the AFTER set will not be applied to the temporary relation. Since it will not be known until FLP has completed whether a transaction belongs to the AFTER set, entries based on log records produced by members of the AFTER set must be ignored in the compensation phase. This requires that the transaction ID of the last update is recorded in the update-list/table. If the sort-merge approach is used, entries in the update-list representing updates by transactions in the AFTER set will be ignored.

For the hash-based approach, two after-images will be needed in the update-table for tuples that are updated after the end of the scan; the after-images for the two last updates to the tuple by different transactions. For tuples that are updated for the first time since the start of the query, the before-image of the log record will

---

18Note that as long as compensation log records are used, it does not matter whether a transaction commits or aborts. In both cases the last log record will contain the after-image of the last committed operations.
give the oldest version while the after-image will give the newest version. In the compensation phase, if two versions exist in the update-table, it will be checked whether the transaction that last updated the tuple is a member of the AFTER set. If not, the newest version will be used. Otherwise, the oldest version must be used. In other words, the last committed version will be used.

4.5.2 Comparison of Undo/No-Redo and No-Undo/Redo Compensation

As long as the base relation of a query is too large to fit in main memory, undo/no-redo compensation will usually outperform no-undo/redo compensation. The main reason for this is that no-undo/redo compensation will use additional disk I/O in order to store and retrieve temporary relations. If temporary relations could be entirely stored in main memory, similar performance may be achieved by both approaches. However, no-undo/redo compensation will require more main memory space. Since a temporary relation will be larger than an update-table, this will also be the case if FLP applies the log records to the temporary relations at once.

The are several other advantages of undo/no-redo compensation:

- Since the query can start emitting tuples immediately, it is well suited for pipelining of relational algebra expressions [Grae93]. Using no-undo/redo compensation, no tuples could be emitted until the scan is finished.

- The query will not have to wait for transactions to terminate. Using no-undo/redo compensation, all transactions that are concurrent with the scan will have to terminate before compensation is started.

- The update-table will be smaller because entries can be deleted after they have been accessed by the scan thread. Using no-undo/redo compensation, all entries have to be stored until the compensation phase.

- The update-table need not be checked for all tuples. The state identifier of a tuple may be used to check whether the update-table may contain any entry for this tuple. When using no-undo/redo compensation, the update-table must be checked for all tuples since a tuple contain no information about future updates.

- Transaction-consistent execution of a transaction containing several read-only queries could be achieved by using the same update-table for all queries. Using no-undo/redo compensation, multi-query transactions could not be supported since each query’s view is determined by the end of the scan phase.\(^{19}\)

One advantage of no-undo/redo compensation is that it will give a more up-to-date view of the database. However, to the user the view given by undo/no-redo compensation will represent a more well defined point in time; the time the query started. The view given by no-undo/redo compensation represent the state at the end of the scan phase. That is, some time between the start and the end of the query.

\(^{19}\) Actually, parallel execution of queries could be supported by extending the scan phase until all scans are finished. However, sequential execution of queries is not possible.
4.5 Redo-Based Compensation

Another advantage of no-undo/redo compensation is that backward log processing is avoided. This may reduce the cost of log processing since no log pages will ever have to be fetched from disk. Note however, that while backward log processing is limited by the number of log records that have to be processed, log processing for no-undo/redo compensation is limited by the lifetime of concurrent transactions. For a transactions that involve several rounds of user input, the lifetime may be significantly longer that the time it takes to access its log records. Thus, a single long transaction may significantly increase the response time of queries when using no-undo/redo compensation.

Summing up, the advantages of undo/no-redo compensation clearly outweighs the disadvantages when compared to no-undo/redo compensation. Most of the advantages is a consequence of that a two-phase approach is avoided.

4.5.3 Undo/Redo Compensation

Combining both undo and redo compensation, gives larger freedom to decide which transactions will be reflected in the query result. The simplest way to ensure that the result will be transaction-consistent, is to include in the BEFORE set all transactions that commit before a certain point in time, while all transactions that are active after that point in time are included in the AFTER set. Such an approach also guarantees strict query consistency. Generally, any point in time between the start and the end of the scan could be chosen, but it seems reasonable to choose a point in time that combine the best of undo/no-redo and no-undo/redo compensation, while avoiding the disadvantages of both.

Generally, a query using undo/redo compensation will apply the no-undo/redo approach until a point in the time which will be called the viewpoint of the query. The viewpoint defines the query’s view of the database. From that point on the undo/no-redo approach is applied. In other words, a query will be executed in two phases, the redo phase and the undo phase (Figure 4.14).

During the redo phase, the scan thread will store the tuples it reads in a temporary relation. The log processing thread will process all log records created after the start of the query and enter relevant after-images into the update-table. Since it is not possible to know which transactions will commit before the viewpoint, the log processing thread must maintain two versions for each entry in the update-table as described for the termination of the scan phase of no-undo/redo compensation. This means that when an new entry is created in the update-table, both the before-image and the after-image of the log record will be entered. This way, the last version committed before the viewpoint will be available in the update-table.

All transactions that are active after the viewpoint will be members of the AFTER set. This means that before-images of log records created before the viewpoint
by such transactions must be entered in the update-table. For tuples updated after
the start of the query, the before-images will have been entered as the oldest of the
two versions in the update-table. However, log records for updates performed before
the start of the query by transactions in the AFTER set must also be processed.
There are two ways to process these log records:

- A backward log processing thread (BLP) as used for undo/no-redo compensation
  is started when the query starts. This BLP should only enter the before-
  images and after-images when no entry already exist for a tuple. The scan
  thread does not have to wait for BLP to finish, but the undo phase may not
  start until BLP has finished.

- No backward log processing is performed at the start of the query. If some of
  the active transactions at the start of the query are still active at the viewpoint,
  the log records created by these transactions before the start of the query will
  be processed. The log records must possibly be fetched from disk, but when
  long updating transactions are rare, this strategy will probably have better
  overall performance than the one above.

When starting the undo phase, the scan thread starts reading tuples from the
temporary relation, checking the update-table for entries for each tuple. If an entry
is found, it will use the newest uncommitted version as described above for no-
undo/redo compensation. Concurrently, the log processing thread is still processing
all log, but now before-images of log records are entered into the update-table if no
earlier entry exist for the corresponding tuple. If an entry contains two versions, the
last committed version prior to the viewpoint is kept while the other is deleted.

When the scan thread has finished reading the temporary relation, it will start
reading the part of the base relation that was not read during the first phase. If a
tuple has been changed since the start of the query, the update-table is checked for
an entry for this tuple.\footnote{If some transactions that started before the query were still active at the viewpoint, the update-
table must be checked for all tuples that have been updated after the start of the oldest of these transactions.}

One obvious choice of viewpoint is when the scan is finished. Using this viewpoint,
a query will not have to wait for concurrent transactions to terminate. However,
the other disadvantages of a two-phased approach are still present.

Another choice of viewpoint is inspired by the main disadvantage of undo/no-
redo compensation. In order to avoid backward log processing, the viewpoint is
chosen as the point in time when all transactions that were active at the start of the
query have terminated. At the presence of long-lived transactions this approach
could degenerate to no-undo/redo compensation. However, this could be avoided
by limiting how late the viewpoint could occur. For example, a time limit may specify
how long a query is allowed to execute before starting to emit tuples. In addition,
by requiring that the viewpoint should occur while the entire temporary relation
still fit in main memory, the I/O disadvantage of a two-phased approach is avoided.

4.6 Non-Strict Query Consistency

Bober and Carey [BoCa92a] show how the performance of transient versioning can
be improved by relaxing the level of consistency provided to queries. Section 4.3.4
4.6 Non-Strict Query Consistency

![Diagram of data item colors and transitions](image)

**Figure 4.15** Transitions for colors of data items.

showed how backward log processing could be reduced for multi-query transactions if non-strict query consistency is allowed. Below it will be shown how this approach could be extended to also reduce the amount of forward log processing and, hence, the size of the update-table.

The basic idea is that only operations performed by transactions that are dependent on a query will be included in the AFTER set. In other words, the query will still use undo/no-redo compensation, but undo is limited to the transactions that is necessary to roll back in order to achieve transaction-consistency. Below, it will first be described how it is determined which transactions are dependent on a query. Then, the changes to the scan thread and the log processing threads are described.

### 4.6.1 Determining Transaction Dependencies

To determine which transactions are dependent on a query, the coloring algorithm by Pilarski and Kameda ([PKa92]) discussed in Section 3.5.3 will be used. For the first presentation it will be assumed that a single query is executed at a time. The algorithm colors data items and transactions in order to determine dependencies. A data item may be either white, grey, or black. Before starting a query, all data items are white. A grey data item means that the query should see the current version of the data item. If a data item is black, it means that the version that should be seen by the query has been overwritten. If a query reads a black data item, undo compensation will have to be performed on this data item before it is emitted. Transactions will either be white or black. A transaction is black if it is dependent on the query.

Figure 4.15 shows how the color of data items will change in the algorithm by Pilarski and Kameda. Operations that do not change the color is not shown. A transaction becomes black if it overwrites a grey data item (i.e., read→write dependent on the query or a black transaction), reads a black data item (write→read dependency), or overwrites a black data item (write→write dependency). Note that the color of a transaction will generally not be known until it has performed all its operations. Thus, if a white data item $X$ is updated by a currently white transaction, this transaction may later turn black. In that case, $X$ should also turn black. For the algorithm presented below, it is assumed that data items are tuples, but other
granularities (e.g., blocks) could also be used.

In a system with several concurrent queries, tuples and transactions may have different colors with respect to different queries. The colors will be represented as bit vectors of a fixed size. A specific bit in a color vector may represent the color with respect to a single query or to a group of queries. Queries that are sharing the same bit position form a consistency group. That is, these queries will have the same transaction-consistent view of the database. Hence, multi-query transactions could be achieved by placing all queries of a transaction in the same consistency group.

Representing Color Information

A color vector will be associated with each transaction. A bit in the color vector of a transaction is set to 1 when the transaction turns black with respect to the corresponding consistency group. Initially, all bits of the color vector will be 0. That is, the transaction is white with respect to all current consistency groups.

Associated with each tuple will be two color vectors, a grey vector and a black vector. When a tuple is white with respect to a given consistency group, the corresponding bit of both bit vectors will be 0. When a tuple turns grey or black, the bit in the grey vector or black vector, respectively, will be set to 1.\(^\text{21}\)

An alternative to using two bit vectors could be a single bit vector with two bits for each consistency group. However, the reason for the separate representation of grey and black is that black vectors need not be explicitly represented. Since a tuple will always have the same color as the last transaction that updated it, the color vectors of transactions will also be used as black vectors. The ID of the last transaction that updated a tuple will be stored within the tuple in order to make it possible to find its black vector.

The grey vectors should not be stored within the tuple. The reason is that a tuple will turn grey as a result of a read operation. Thus, if the grey vectors are included in the tuple, read operations would cause data pages to become dirty. This would significantly affect the performance of most database applications. In other words, the grey vectors of each tuple should be stored in main memory.

The grey vectors will only be used for tuples that have been read by black transactions. Tuples that are read by a query will also turn grey, but this information will be represented by a key range read-only lock on the key determining scan order. As opposed to ordinary key range locks, a read-only lock will not prevent updates to tuples within the range. However, any transactions requesting a conflicting tuple lock will turn black. In order to get as few black transactions as possible, only tuples that have already been read will be covered by a query's read-only lock. That is, the range will be incrementally extended as the query proceeds.

Read-only locks were proposed by Bober and Carey to implement their rules for insertions in a query's AFTER set [BoCa92]. However, they did not recognize that key range locks are needed in order to avoid phantom problems. That is, a transaction that inserts a tuple into a range that has already been read by the query, must turn black. That way, all other updates by this transaction will also be invisible to the query.

Achieving the same transaction-consistent view for all queries of a consistency

\(^{21}\)It will not be necessary to reset the bit in the grey vector when a tuple goes from grey to black. If both bits are set, the color will be interpreted as black.
4.6 Non-Strict Query Consistency

If the queries of a consistency group do not need to see the same transaction-
consistent view, read-only locks may be released when the corresponding query has
completed. The queries will then form a strong consistency group [Bobe93]. That
is, the queries will serialize with each other, but not necessarily with queries of
other groups. Early release of read-only locks will require that grey vectors are
also maintained for tuples covered by read-only locks. Otherwise, information on
read→write dependencies between transactions will be lost.

Rules for Color Transitions

The color transitions given by Figure 4.15 will be implemented as follows:

1. When a transaction acquires a write lock for a tuple, the bits of its color vector
   will be set to 1 for consistency groups containing queries that have a read-only
   lock covering this tuple. In other words, a transaction that updates a tuple
   after it has been read by a query will turn black.

2. Each time a transaction updates a tuple, its color vector, the tuple’s grey
   vector and the color vector of the previous transaction that updated this tuple
   is OR’ed together, and the result is assigned to the transaction’s color vector.
   Thus, a transaction will turn black when it updates a black or a grey tuple. In
   addition, an update by a black transaction will cause the tuple to turn black,
   since the transaction’s color vector now also acts as the tuple’s black vector.

3. When a transaction reads a tuple, its color vector and the grey vector of the
   tuple should be OR’ed to become the new grey vector. In other words, the
   tuple should turn grey if the transaction is black. This operation can be delayed
   until the read lock is released since the new grey vector is not needed until the
   next update. Since 2PL is assumed, the final color of transactions will be
   known when locks are released.

The rules above requires that queries do not read uncommitted data. This is
because the color of a transaction can not be determined until it has completed.
In other words, the true color of a currently white tuple could not be determined
until the transaction behind the last update to the tuple has completed. Reading
uncommitted data can be avoided by requiring that the scan thread use cursor
stability locking. However, this could significantly delay the query in the presence of
long-lived transactions. In order to avoid having to wait for transactions to commit,
a turn-black strategy ([Pu^88]) could be used instead. That is, the uncommitted
transaction is colored black so that the query will undo the update and see the
committed version of the tuple.

Since the color vectors are of a fixed size, the number of concurrent consistency
groups will be limited to the size of the color vectors. If no bit position is free when
a query is started, it must be included in an existing consistency group. When all
queries of a consistency group has finished, the corresponding bit position may be reused.

When reusing a bit position, all bits at that position should be set to 0 since tuples and transactions are initially white. However, it will not be necessary to update all vectors each time a new query starts. Instead, the vectors will be updated when they are needed. For color vectors of transactions, the LSN of the latest log record of the transaction could be used in order to determine whether its color vector needs to be recomputed. That is, if a new consistency group has been started after this log record was created, a re-computation may be necessary. A table could be maintained that contained information on which bit positions need to be updated for different ranges of LSN. Each entry could be a bit mask that should be AND’ed to the color vector. Each time a bit position is reused, the entries of this table is recomputed. For the same purpose, a state identifier could be stored with each grey vector indicating whether bit positions have been reused since the last time the grey vector was updated.

To use as little storage as possible, grey vectors should only be kept for tuples that are grey with respect to at least one consistency group. All grey tuples will turn black when they are updated. This means that a grey vector can always be removed after an update. In other words, grey vectors will only be maintained for tuples that have been read, but not yet written, by a black transaction. To access the grey vectors a hash table may be used. The hash table should show good performance with respect to unsuccessful searches since only a small fraction of the tuples in a system will normally be grey. This means that the number of buckets in the hash table should be so large that the probability of having more than one grey vector per bucket is low.

In order to reduce the storage requirements for grey vectors, the same grey vector could be used for several tuples. For example, grey vectors may be stored in a table of fixed size, and a hashing function may be used to map tuples to grey vectors. This will also speed up access to grey vectors since key comparisons will not be required. Using this approach, if a tuple associated with a particular grey vector is read by a black transaction, all tuples associated with this grey vector will be grey for the rest of the lifetime of the consistency group unless they turn black. Since “false” read→write dependencies are introduced, using a fixed number of grey vectors will increase the number of black transactions. In other words, the amount of log processing will increase.

4.6.2 Query Processing

In order to achieve transaction-consistency, queries should not read black tuples. When the scan thread reads a tuple that is black with respect to the query's consistency group, compensation must be performed. A straightforward approach for compensation is to let the scan thread access the necessary log records in order to perform undo-compensation. This requires that the previous state identifier of a tuple is included in each log record. This way, log records for updates to a given tuple are chained together. The scan thread could then start compensation by accessing the log record given by the current state identifier of a tuple. It will then go backwards in the tuple log chain, undoing operations until it reaches a log record for a white transaction.
4.6 Non-Strict Query Consistency

In order to be able to know the color of a tuple and when to stop going backwards in a tuple log chain, the color vectors of transactions must be available after the transactions have terminated. A color vector may be garbage-collected when no active consistency group includes queries that started before the termination of the transaction. If no color vector is found for a transaction, it may be assumed to white with respect to all consistency groups.

Backward log processing by the scan thread will be efficient when log records reside in main memory. However, having to access disk to read log records will reduce the performance of both queries and concurrent transactions. An alternative is to introduce a forward log processing thread (FLP) that will process log records when they are still in main memory. FLP may restrict log processing to black tuples and enter the relevant information into an update-table which will be accessed by the scan thread as described for undo/no-redo compensation.

As discussed above, the first black version of a tuple may not yet have been colored black if the updating transaction has not terminated. Thus, in addition to the log records processed by FLP, another log record may have to be processed for each tuple. When FLP creates a new entry in the update-table, it should include the LSN of the previous update to the tuple if the last update has not yet been committed. This way, the scan thread could then if necessary process this log record before using the entry for compensation. Alternatively, a special thread could be assigned to this task. FLP could insert the LSN of such log records in a list which is later processed by this thread. This way, I/O cost could be further reduced by using an elevator algorithm ([Denn67]) when accessing log records. If also the ID of the transaction that performed the previous update to a tuple is included in log records, the amount of extra log processing could be reduced by only accessing log records created by black transactions.

Tuples that have been deleted by black transactions must also be included in the query result. Deletions can be handled by a priority queue as described for undo/no-redo compensation. Note that when processing log records for a transaction that is still active, references to deleted tuples must always be inserted in the priority queue since the transaction may later turn black. An entry in the priority queue will be ignored if the corresponding deletion was performed by a finally white transaction.

An alternative to using a priority queue is to create a so called tombstone when tuples are deleted. That is, a reference to the deleted tuple is stored in the data page in which the tuple was located before the deletion. A tombstone will contain the key, the LSN of the delete log record, and the ID of the transaction that deleted the tuple. When a query finds a tombstone for a tuple that has been deleted by a black transaction, it will restore the deleted tuple by using the before-image of the log record given by the LSN of the tombstone. If a tuple with the key of a tombstone is inserted, the LSN of the tombstone will be part of the insert log record. Hence, the chaining of the log records of a tuple is maintained. Tombstones may be garbage-collected when it is no longer possible that the deleting transaction will be black with respect to current or future consistency groups. That is, when the transaction both has terminated and is white with respect to all current consistency groups.

An advantage of the tombstone approach is that no forward log processing will be required. In other words, one could, if preferred, base the compensation entirely on backward log processing. Hence, compensation-based query processing could be performed without the need for an update-table.
4.6.3 Levels of Query Consistency

The coloring algorithm presented in this section can support all levels of query consistency as defined by Bober and Carey [BoBe93]. The strict approach of undo/no-reduo compensation as presented in Section 4.2, is equivalent to coloring all current transactions black with respect to a new consistency group. In addition, all transactions that start during the lifetime of the consistency group will be initialized as black. The transient versioning approach used by Oracle [Bri+97] is really a similar approach where commit time is used to determine color of transactions and data items.

As discussed above, queries referring to the same bit position in the color vectors will form a strong consistency group. To achieve strong consistency for all queries, a timestamp indicating their startup time will be assigned to each consistency group. If a transaction turns black with respect to a consistency group it should also turn black with respect to older consistency groups. Hence, serialization order of queries will be given by the timestamps of the consistency groups.

The coloring algorithm as described by Pilarski and Kameda guarantees weak query consistency since all three types of dependencies are taken care of. Thus, there could be no single-query cycle in the dependency graph. Running queries using weak consistency will give better performance than strong consistency. This is because access to tuples read by queries of one consistency group will never make a transaction black with respect to other consistency groups. Hence, the number of black transactions will be lower.

Maintaining grey vectors are not needed when only guaranteeing update query consistency. Without grey vectors the coloring algorithm will ignore read→write dependencies between transactions. Hence, single-query cycles may appear in the dependency graph. Using update consistency will improve performance by reducing the number of grey tuples. In addition, transactions will not have to update grey vectors, and main memory requirements of the coloring algorithm could be significantly reduced.

Further studies are needed in order to analyze how much could be saved by relaxing the level of consistency provided to queries.

4.7 Concluding Remarks

In this chapter, a novel algorithm for compensation-based query processing has been presented. By using undo/no-reduo compensation, many of the disadvantages of the earlier algorithm by Srinivasan and Carey [SrCa92a] will be avoided:

- Queries do not have to wait for transactions to terminate. In other words, a single long-lived transaction will not block query processing.

- Efficient pipelining of relational algebra operations is possible since a query can start emitting tuples as soon as backward log processing is finished.

- No special treatment is needed for aborting transactions. Log processing will ensure that the before-images of updates by concurrent transactions are entered in the update-table regardless of the outcome of the transactions.
4.7 Concluding Remarks

- Temporary storage of base relations is avoided since the query could be executed in a single phase.

- Transaction-consistent execution of read-only transactions involving several queries is possible by using the same update-table for all queries.

- Delta locks/operations could be supported by storing operands instead of before-images in the update-table.

The presented algorithm bases its compensation on information collected from the database internal log. That way, less work is put on transaction than when using transient versioning or the compensation-based approach by Srinivasan and Carey. All these approaches require that transactions make an extra copy of data for use for query processing. The approach by Srinivasan and Carey also requires that transactions looks up information on concurrent queries.

In the original transient versioning algorithms, prior versions are stored in a separate version pool [Cha82]. This will potentially reduce query performance since the data clustering is disrupted with respect to queries. In addition, several disk accesses may be needed in order to locate the correct version of a data item. To reduce this problem, on-page caching of prior version has been proposed [BoCa92b]. However, on-page caching will decrease the buffer hit ratio for transactions since the current versions will be distributed over more pages.

Unlike transient versioning, compensation-based query processing will only materialize versions that are actually needed by queries in the system. This means that there will be no overhead when there is no active queries. On the other hand, transient versioning will never maintain more than one instance of a prior version while update-tables for concurrent queries may each contain a copy of the same version of a tuple. However, this can also be achieved with compensation-based query processing by letting a single forward log processing thread and a single update-table serve multiple concurrent queries. The update-table may then contain several versions of the same tuple, and each version will be tagged with the query IDs of the queries that should access this particular version.

Compensation-based query processing also has the advantage of being able to tailor the content of the update-table to specific queries. In addition, if the update-table is (partly) stored on disk, its organization could be optimized on I/O cost given the access pattern of the query. This is not possible for the general version pool used in transient versioning.

This chapter has also briefly discussed algorithms for no-undo/redo compensation and undo/redo compensation. No-undo/redo compensation could be an alternative if the entire base relation fit in main memory. Then, concurrent operations could be redone on this copy, and temporary storage on disk is avoided. Otherwise, the two-phased approach will significantly increase disk I/O compared to undo/no-redo compensation. Still, no-undo/redo compensation will be less suited for pipelining of relational algebra operations and have most of the other disadvantages listed for the approach by Srinivasan and Carey in Section 3.4.

The proposed algorithm for undo/redo compensation tries to avoid backward log processing by rolling forward transactions that are active at the start of the query. The duration of the redo activity should be limited in order not to gain the disadvantages of a two-phased approach. Thus, at the presence of long-lived transactions, backward log processing may not be totally avoided. However, in most
cases the number of transactions that is necessary to roll back will be significantly reduced.

Finally, an algorithm was proposed which tries to reduce the necessary amount of log processing by only including in a query's AFTER set, transactions that are dependent on the query. A coloring algorithm is used to determine the dependencies. This approach will not guarantee strict query consistency [Bob93], but for most applications this is not necessary. One of the main advantages of this approach will be reduced space requirements for the update-table. In fact, at the expense of having to read more log records from disk, the algorithm makes it possible to perform compensation-based query processing without the use of an update-table.

The rest of this thesis will concentrate on the algorithm for undo/no-redo compensation. First, it is shown how different types of queries can be efficiently executed using this approach. Then, its performance is evaluated both analytically and through simulations. Finally the algorithm will be extended to query execution in distributed database systems.

---

22 For distributed systems, the notion strict query consistency is meaningless since there will normally not be any globally defined commit order of transactions.
Chapter 5

Query Execution

In the previous chapter it was described how a transaction-consistent snapshot could be obtained using undo/redo compensation-based query processing. Any query could be executed by first obtaining a transaction-consistent copy of the base relations and then run the query on the copy. However, this simplistic two-phased approach may result in a potentially large overhead with respect to storage, disk accesses and other resources. This chapter discusses how queries can be executed more efficiently by merging query execution with undo/redo compensation.

Below it is described how the relational algebra operations selection, aggregation, and join could be implemented. Some of the methods presented below, may require that the log processing threads have access to certain attributes of the tuple (in addition to the primary key). This can not be guaranteed when using partial tuple logging. In order for these methods to work, complete tuple logging must be used. It will be explicitly stated where complete tuple logging is required.

5.1 Selection Queries

A selection query can be viewed as a filter on a relation. The filtering may either be done on the inputs to the scan thread and the log processing threads or on the output from the compensation (Figure 5.1).

5.1.1 Output Filtering

Output filtering is just a selection performed on the transaction-consistent output of the compensation. That is, the operations of the scan thread and the log processing threads are the same as when producing a snapshot, but after the compensation has been performed on a tuple, the scan thread will evaluate the selection predicate, and only emit those tuples that satisfy the predicate.

Assuming the following query over the EMPLOYEE relation of Table 4.2:

**Query 1.**  SELECT name FROM employee WHERE salary < 30

The log processing threads will for each log record that has updated the EMPLOYEE relation make an (empno, name, salary) entry in the update-table. The scan thread will extract the same attributes from each tuple it reads, and if necessary
check the update-table and compensate for any updates to these attributes. After the compensation has been performed, the selection predicate is evaluated, and only the name attribute of tuples with a salary below 30 are emitted.

In addition to performing a selection, Query 1 also performs a projection. Attributes that are neither part of the projection nor the selection predicate are not entered in the update-table. This implies that the log processing threads need not process log records that only contain updates to other attributes. For Query 1 this means that log records for updates that only changes dept_no may be skipped.

Figure 5.2 shows the execution of Query 1 when using output filtering, assuming the same concurrent updates as used in the example of Figure 4.5. An extra column has been added showing the tuples after compensation but before performing output filtering. The figure assumes that a priority queue is used to achieve sorted output and that entries are not made in the update-table for tuples that have already been read by the scan thread. Note also that FLP does not make an entry in the update-table when log record 2 is processed since no relevant attributes have been updated.

### 5.1.2 Input Filtering

When performing input filtering, the scan thread will only check the update-table for tuples that satisfy the selection predicate. The log processing threads will evaluate the predicate on the before-images of log records. If the before-image evaluates to true, a delete entry is entered into the update-table. If it evaluates to false, an insert entry is entered into the update-table signalizing that this tuple should not be part of the output. (Remember that tuples with insert entries are ignored by the scan thread.) If an entry for a tuple already exists in the the update-table, only BLP, and not FLP, will perform a new evaluation.

For each tuple that passed input filtering, the scan thread performs the compensation as described in Section 4.2, adding tuples with delete entries and ignoring tuples
5.1 Selection Queries

<table>
<thead>
<tr>
<th>Out Filt.</th>
<th>Comp.</th>
<th>Scan</th>
<th>Update-table</th>
<th>FLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Karl) ← (Karl,29) ∈</td>
<td>(1, Karl, 28,01)</td>
<td></td>
<td>W 1 Karl 29</td>
<td>1 W 1 in Karl, Karl ∈ 29,28</td>
</tr>
<tr>
<td>(Tore) ← (Tore,28) ∈</td>
<td>(2, Tore, 28,01)</td>
<td></td>
<td>2 W 2 2 2 20</td>
<td></td>
</tr>
<tr>
<td>← (Sophus,32) ∈</td>
<td>(4, Sophus, 32,03)</td>
<td>D 7 Heidi 27</td>
<td>3 D 7 Heidi, 27 2 2 02</td>
<td></td>
</tr>
<tr>
<td>← (Oystein,35) ∈</td>
<td>(5, Oystein, 35,03)</td>
<td>W 17 31</td>
<td>4 W 17 a 31 28</td>
<td></td>
</tr>
<tr>
<td>← (Oddmund,35) ∈</td>
<td>(6, Oddmund, 35,03)</td>
<td></td>
<td>I 12</td>
<td>5 I 12 Bjorn G, 26 04</td>
</tr>
<tr>
<td>(Heidi) ← (Heidi,27) ∈</td>
<td>(8, Gunnar, 38,02)</td>
<td>W 16 Ole</td>
<td>6 W 16 a Ole, Ole-H, 0 3 0</td>
<td></td>
</tr>
<tr>
<td>← (Svein,E,) ∈</td>
<td>(9, Svein E, 33,03)</td>
<td></td>
<td>D 17 Bjorn 31</td>
<td>7 W 5 a 38 37</td>
</tr>
<tr>
<td>← (Oystein,35) ∈</td>
<td>(11, Oystein, 35,03)</td>
<td></td>
<td>8 D 17 Bjorn, 28 03</td>
<td></td>
</tr>
<tr>
<td>← (Yurong,31) ∈</td>
<td>(14, Yurong, 31,03)</td>
<td></td>
<td>9 I 17 Bjorn P, 26 03</td>
<td></td>
</tr>
<tr>
<td>← (Harald,45) ∈</td>
<td>(15, Harald, 45,02)</td>
<td></td>
<td>W 16 Ole 26</td>
<td>10 W 16 a 26 31 4 01 04</td>
</tr>
<tr>
<td>← (Ole,26) ∈</td>
<td>(16, Ole-H, 31,04)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>← (Bjorn,31) ∈</td>
<td>(17, Bjorn P, 25,03)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.2 Example of an execution of Query 1 using output filtering.

with insert entries. Hence, tuples that have been changed so that they no longer satisfy the selection predicate, are still included in the query result. Tuples that have been changed so they now evaluate to true, but in the transaction-consistent view should evaluate to false, will be eliminated from the query result because an insert entry is found in the update-table.

Input filtering generally requires the use of complete tuple logging. In order for the log processing threads to perform the filtering, it will need access to all attributes needed for evaluation of the selection predicate. This is not guaranteed when using partial tuple logging.

Figure 5.3 shows the execution of Query 1 when using input filtering. Since full tuple logging is required, log records contain before-images of all attributes plus after-images of changed attributes. This means that only the first relevant log record needs to be processed for each tuple. Hence, FLP will not process log records 8→10. Since the salary attribute is only needed for evaluating the selection predicate, it can be omitted from the update-table when the evaluation is performed by FLP.

As shown by Figure 5.3, the scan thread ignores tuples with a salary above 30, while FLP enters a delete entry when the salary is less than 30 and an insert entry when the salary is above 30. A delete entry in the priority queue cause tuple 16 to be included in the output regardless of it not being included by the scan thread. An insert entry for tuple 17 cause it to be omitted from the query output since its
Query Execution

<table>
<thead>
<tr>
<th>Out Comp.</th>
<th>Fit.</th>
<th>Scan</th>
<th>Update-table</th>
<th>FLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Karl) ( \text{Fl}_1 ) (1.Karl) ( \leftarrow ) (1.Karl,28,01)</td>
<td>D 1 Karl</td>
<td>W 1 Karl 29 b1</td>
<td>Karl 28</td>
<td></td>
</tr>
<tr>
<td>(Tore) ( \text{Fl}_2 ) (2.Tore) ( \leftarrow ) (2.Tore,28,01)</td>
<td>D 7 Heidi</td>
<td>W 4 Bjørn 31 b3</td>
<td>Bjørn 26</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Heidi) ( \text{Fl}_3 ) (8.Gunnar,38,02)</td>
<td>I 17</td>
<td>W 4 Bjørn 31 b3</td>
<td>Bjørn 26</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Ole) ( \text{Fl}_4 ) (16.Ole-H,31,04)</td>
<td>D 16 Ole</td>
<td>W 6 Ole 26 b3</td>
<td>Ole-H 41</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 5.3** Example of an execution of Query 1 using input filtering.

salary was originally above 30.

Creating a delete entry for all log records with a before-image that evaluates to true, will result in a very large priority queue for queries with high selectivity factors. Since the cost of inserting and removing entries from a large priority queue will be higher than for a hash-table, this will also affect query performance.\(^1\)

In order to reduce the number of delete entries, the log processing threads may only enter a delete entry when the after-image of the log-record evaluates to false. For log records with after-images that satisfy the selection predicate, a delete entry is not necessary since the scan thread will include the corresponding tuples anyway. Hence, a write entry may be used instead to communicate updates to relevant attributes. Table 5.1 gives the entry types used in the update-table for this strategy. Using this strategy, log record 1 of Figure 5.3 will cause a write entry since the update does not change the outcome of evaluating the selection predicate. Note that this strategy requires that FLP re-evaluates the after-images for log records of tuples for which a write entry already exists.

The log processing threads may enter a write entry and skip evaluation of the selection predicate if it can be assured that the update could not possibly alter the

\(^1\)If output order is not an issue, the priority queue will not be needed. Instead, the log processing threads could emit tuples satisfying the selection predicate immediately. In that case, an insert entry must be made in order to prevent the scan thread from emitting the same tuple later.
5.1 Selection Queries

Table 5.1 Entry types in the update-table for input filtering.

<table>
<thead>
<tr>
<th>Before-image</th>
<th>After-image</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

outcome of the evaluation. One example is updates where none of the attributes used in the selection predicate have been changed. If another update is made to the same tuple later, both the before-image and the after-image may have to be evaluated. In Figure 5.3, a write entry could be made for log record 6 since the salary is not updated. This entry would be changed to a delete entry when processing log record 10. The update to the salary attribute of this log record alters the outcome of predicate evaluation.

An alternative when using partial tuple logging could be to combine input and output filtering. This way, the log processing threads will filter as much as possible, leaving the rest until after the compensation. In that case, delete entries would also have to be made for all log records that the log processing threads were not able to fully evaluate. These entries should be marked in order to cause a re-evaluation when processed by the scan thread. Depending on the selection predicate it may be possible to only re-evaluate parts of the predicate. However, combining input and output filtering will further increase the size of the priority queue.

5.1.3 Comparison of Output and Input Filtering

Compared to output filtering, input filtering tries to reduce query processing costs by reducing the number of tuples that needs to be compensated for. This is achieved at the expense of having both the scan thread and the log processing threads evaluate the selection predicate.

In order to compare the cost of input and output filtering, the parameters listed in Table 5.2 will be used. Using these parameters the cost of compensation and predicate evaluation for output filtering will be

\[
C_{\text{output}} = nC_e + uC_z.
\]  

(5.1)

Assuming uniform access pattern for transaction, the corresponding cost for input

Table 5.2 Parameters used for comparing input and output filtering.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n)</td>
<td>Number of tuples in base relation</td>
</tr>
<tr>
<td>(s)</td>
<td>Selectivity factor of the query</td>
</tr>
<tr>
<td>(u)</td>
<td>Number of tuples updated during the query</td>
</tr>
<tr>
<td>(C_e)</td>
<td>Cost of evaluating the selection predicate</td>
</tr>
<tr>
<td>(C_z)</td>
<td>Cost of performing compensation</td>
</tr>
</tbody>
</table>
filtering will be

\[ C_w = (n + u)C_e + s u C_e. \] (5.2)

In order for input filtering to be cost-effective, we have that

\[
\begin{align*}
C_w &< C_{st} \\
(n + u)C_e + s u C_e &< nC_e + u C_e \\
u C_e &< u C_e - s u C_e \\
C_e &< (1 - s)C_e
\end{align*}
\]

In other words, input filtering should only be used for queries with low selectivity factors where the cost of evaluating the selection predicate is smaller than the cost of performing the compensation. Also note that the since the number of delete entries will be larger for input filtering, the size of the priority queue and the cost of maintaining it will be larger. When also taking into account that input filtering requires full tuple logging, the conclusion will be that for most queries output filtering should be preferred.

## 5.2 Aggregate Queries

There are two basic types of aggregates, scalar aggregates and aggregate functions. While scalar aggregates compute a single value, aggregate functions return a set of values, each for a different subgroup of the data. In SQL, aggregate functions are achieved by using the **GROUP BY** clause.

Below, algorithms for compensation-based execution of aggregate queries are presented. Some of the algorithms requires that FLP knows whether a tuple lies behind or ahead of the scan thread. This requires that all log records that are processed by FLP contain the attributes determining scan order. This will be true in the following situations:

- Scan sequence is in primary key order.
- A single attribute determine scan order, and FLP need only process log records with updates to this attribute.
- Complete tuple logging is used.

### 5.2.1 Scalar Aggregates

Aggregate operations can be divided into two types:

- Operations that for each set compute a single value in which the values of all members of the set are reflected (e.g. **COUNT, SUM, AVG**).
- Operations that return one of the atomic values of the set (e.g. **MIN, MAX**).

For both types of operations it is possible to use an approach similar to the output filter approach for selection queries. Using this approach, aggregation will be pipelined with compensation. That is, after compensation has been performed,
the scan thread will apply the tuple to the aggregated result. However, the amount of compensation can be significantly reduced by using input aggregation.

The input aggregation approach will be different for the two groups of aggregate operations. Below, algorithms for performing the aggregate operations for the two different groups are presented.

**SUM-like Aggregate Operations**

For aggregate operations of the first group, the following query will be used to described the algorithm:

**Query 2.** \( \text{SELECT SUM(salary) FROM employee} \)

When executing Query 2, the scan thread will proceed as normal, computing the sum of the tuples it sees. The log processing threads will compensate for concurrent updates by subtracting the delta of each update from the current sum (Figure 5.4). FLP should only do this for updates that are reflected in the tuple when read by the scan thread. This is achieved by checking the cursor of the scan thread in order to determine whether the tuple of the current log record lies ahead of or behind the scan thread. A log record for a tuple that lies behind the scan thread should be ignored since the tuple was read before the update occurred.\(^2\) Since the log processing threads directly updates the aggregate result, a transaction-consistent result will have been computed when the scan thread is finished. Note that no update-table is required for the execution of Query 2.

Let \( \sigma \) be the variable used for storing the aggregated sum of the salary attribute during the execution of Query 2. At the start of the query, \( \sigma \) will be initialized to zero. During query execution, the scan thread will update \( \sigma \) by adding the value of the salary attribute for each tuple it reads. Concurrently, the log processing threads will also update \( \sigma \) in order to undo concurrent updates to the salary attribute. Let \( \alpha \) and \( \beta \) be the after and before-images, respectively, of the salary attribute of a log

\(^2\)Note that it is assumed that the scan thread and FLP is synchronized as described in Section 4.2.3. That is, before the value of a tuple is added to the current sum, it is made sure that all log records for this tuple have been processed.
record. Then, the log processing threads will undo the effects of the corresponding update by performing the following computation:

\[ \sigma \leftarrow \sigma - \alpha + \beta \]  \hfill (5.3)

In other words, the before-image will be reflected in the aggregate result instead of the after-image. The computation in (5.3) is only applied for log records for tuples that lie ahead of the scan thread.

Figure 5.5 shows an example of an execution of Query 2. The leftmost column shows the current value of the sum (\(\sigma\)). The sum is updated each time a new tuple is read. For log records that contain updates to the salary attribute (1, 4, and 10), the computation in (5.3) is performed. One exception is log record 7 since tuple 5 has already been read by the scan thread. For delete and insert log records the salary is added or subtracted, respectively, to the current sum. This way, the salary of tuple 7 is included in the sum even if it is not read by the scan thread, and tuple 12 is not reflected in the sum even if its salary is added to the current sum by the scan thread. Note that an update is compensated for by FLP before the update is reflected in the sum.

To compute the COUNT operation, the scan thread will increment the counter for each tuple it reads. The log processing threads will increment or decrement the counter when processing delete or insert entries, respectively. If NULL values are present, write log records must also be taken into account when computing the count of a specific attribute. The AVG operation will be computed by running a query
computing both a transaction-consistent sum and count. Other statistical functions (e.g., \textsc{Variance}) could be computed in a similar way.

\textbf{MIN-like Aggregate Operations}

The following query will be used to describe the algorithm for aggregate operations that select one of values for the given attribute:

\textbf{Query 3. SELECT MIN(salary) FROM employee}

The basic idea for the algorithm is to let the scan thread compute the minimum of the salary values that are not changed during the scan while the log processing thread computes the minimum of salary values that are changed by concurrent transactions (Figure 5.6).

When executing Query 3, the scan thread will scan the relation updating the aggregate result with the minimum salary value found so far. Since the scan thread should only consider tuples not changed by concurrent transactions, it will ignore tuples for which an entry exist in the update-table. Note that the update-table need only be checked for tuples with a salary value that is lower than the current minimum.

The log processing threads will, in addition to inserting entries in the update-table for updates to the \textsc{salary} attribute, compute the minimum over the before-images. The entries in the update-table will serve two purposes. In addition to making the scan thread ignore values written by concurrent transactions, it tells FLP which tuples have been updated before. Log records for such tuples should be ignored since their before-images will not be part of the transaction-consistent view.

Note that until BLP is finished, FLP could not know by checking the update-table whether the current log record represents the first update to the tuple. One way to solve this problem is to delay the start of FLP until BLP has finished. A problem with this approach is that this may prohibit FLP from processing log records while they reside in main memory. An alternative approach is to not start updating the aggregate variable until BLP is finished. Until then, BLP and FLP will also include
before-images of the salary attribute in the entries in the update-table. When BLP
is finished, the scan thread and FLP may start updating the aggregate variable as
described above. In addition, the update-table will be scanned, and the current
minimum updated if a lower salary value is found.

Figure 5.7 shows an example of an execution of Query 3. The leftmost column
shows the current minimum values during the scan. When reading tuples 1, 2, 12,
and 17, the salary value is lower than the current minimum. Before updating the
current minimum the update-table is checked. For tuples 1, 12, and 17, an entry is
found, and the current minimum is not updated. Log records 1 and 10 also updates
the current minimum since the value of their before-image is lower than the current
minimum.

Note that no entry is made when processing log record 3. This is because the
scan thread will not see tuple 7 unless a new tuple with the same primary key is
inserted later. At that time an insert entry will be made for the tuple. Similarly,
the entry for tuple 17 could have been deleted when the tuple is deleted (log record
8). A new entry would then be made when the tuple is re-inserted (log record 9).
Entries in the update-table are in Figure 5.7 only shown for log records that contain
updates to the salary attribute of tuples that lie ahead of the scan thread. However,
it is not necessary that FLP is able to decide whether a tuple lies ahead of the scan
thread. If not, a tuple may be processed by both the scan thread and FLP, but the
result will not change.

<table>
<thead>
<tr>
<th>MIN</th>
<th>Comp.</th>
<th>Scan</th>
<th>Update-table</th>
<th>FLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>←</td>
<td></td>
<td></td>
<td>1 W 1 a Karl Karl s 29 28</td>
</tr>
<tr>
<td>28</td>
<td>←</td>
<td>(1, Carl, 28, 01)</td>
<td>I 1</td>
<td>2 W 2 a 02 01</td>
</tr>
<tr>
<td>27</td>
<td>←</td>
<td>(2, Tore, 28, 01)</td>
<td>(4, Sophus, 32, 03)</td>
<td>3 D 7 a Heidi 27 22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(5, Oystein, 35, 03)</td>
<td>(6, Oddmund, 35, 03)</td>
<td>I 17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4 W 17 s 31 28</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(8, Gunnar, 38, 02)</td>
<td>I 12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(9, Svein E., 33, 03)</td>
<td>(11, Oystein, 35, 03)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5 W 12 Bjorn G. 26 04</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6 W 16 a Ole Ole-H. 4 03 03</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(12, Bjorn G., 26, 04)</td>
<td>7 W 5 a 35 07</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(14, Yurong, 31, 03)</td>
<td>(15, Harald, 45, 02)</td>
<td>8 D 17 a Bjorn 28 03</td>
</tr>
<tr>
<td></td>
<td>←</td>
<td>(16, Ole-H., 31, 04)</td>
<td>(17, Bjorn P., 25, 03)</td>
<td>I 16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10 W 19 a 26 31 01 04</td>
</tr>
</tbody>
</table>

**Figure 5.7** Example of an execution of Query 3.
5.2 Aggregate Queries

If the minimum or maximum values of more than one attribute are computed by the same query, each entry in the update-table will have flags, signaling which attributes have been changed. This way, the scan thread could determine which attributes it should handle and which should be left to the log processing threads. The flags are necessary if partial tuple logging is used since the log processing threads will not necessarily have access to all attributes of an updated tuple. If complete tuple logging is used, the scan thread may ignore all updated tuples since the log processing threads will have access to all attributes of these tuples.

5.2.2 Aggregate Functions

Aggregate functions compute aggregates for subgroups of the tuples in a relation based on certain attribute values, as specified by the BY-list of the query. The following query is an example of an aggregate function:

**Query 4.**

\[
\text{SELECT dept\_no, SUM(salary) FROM employee GROUP BY dept\_no}
\]

Query 4 will compute the total salary expenses per department. There are two basic algorithms that are commonly used to compute aggregate functions [Eps79]:

1. The first technique maintains a temporary relation having one tuple for each subgroup. When a tuple is read from the base relation, the tuple of the corresponding subgroup, as defined by the BY-list is updated. When the base relation has been read, the temporary relation will contain the result of the aggregation.

2. An alternative strategy is to first project the base relation on the needed attributes and then sort the results on the BY-list. The final merge of the sort can easily produce the aggregate values since all of the entries are clustered on their BY-list values.

A hash-based version of the first method is likely to be most efficient, regardless of the number of subgroups [Brat84]. The second method will be preferred when the base relation is already sorted on the BY-list. Below, compensation-based algorithms for both methods are presented. Both algorithms require that the log processing threads have access to the attributes of the BY-list. This can only be guaranteed if complete tuple logging is used.

**Hash-based Aggregation**

The compensation-based execution of the hash-based method will not be very different from the execution of scalar aggregates. The main difference is that an update may change which subgroup a tuple belongs to. This means that for Query 4 a log record that updates the salary attribute, causes the log processing thread to subtract the after-image of the salary from the aggregate variable for subgroup given by the after-image, and add the value of the before-image to aggregate variable of the subgroup of the before-image. Hence, if the log record also contains an update to the dept\_no attribute, two different aggregate variables will be affected. If dept\_no
and not salary is updated, the current salary value is subtracted from the new subgroup and added to the old subgroup. In both cases the log processing threads may need access to attributes that have not been updated. In other words, complete tuple logging is needed.

Figure 5.8 shows an example of an execution of Query 4. Complete tuple logging is assumed. The left-most column gives the current sums for departments 01, 02, 03, and 04, respectively. One difference from the scalar example of Figure 5.4 is that not only log records for updates to salary must be processed but also those containing updates to dept_no. For example, since log record 2 records a change in dept_no of tuple 2 from 02 to 01, the salary value for tuple 2 will be subtracted from the aggregate variable for department 01 and added to the aggregate variable for department 02.

In the example of Figure 5.8, department 04 should not be part out the query result since the transaction-consistent view contain no tuples belonging to this department. In order to know whether tuples with a sum of 0 should be emitted from the query, the number of elements belonging to each subgroup need also be counted.

For MIN-like operators, the scan thread will, if there exist an entry in the updateable for a possible new minimum candidate for a subgroup, ignore it as for scalar aggregation. Both the scan thread and the log processing threads will update the temporary relation, and the minimum values for each subgroup at the end of the scan will be the minimum of the values found by the two threads.

<table>
<thead>
<tr>
<th>SUM</th>
<th>Scan</th>
<th>FLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,0,0,0)</td>
<td></td>
<td>1 W</td>
</tr>
<tr>
<td>(-27,28,0,0)</td>
<td>(1,Carl,28,01)</td>
<td>2 W</td>
</tr>
<tr>
<td>(29,28,0,0)</td>
<td>(2,Tore,28,01)</td>
<td>3 D</td>
</tr>
<tr>
<td>(29,55,0,0)</td>
<td>(4,Sophus,32,03)</td>
<td>4 W</td>
</tr>
<tr>
<td>(29,55,67,0)</td>
<td>(5,Oystein,35,03)</td>
<td>5 D</td>
</tr>
<tr>
<td>(29,55,70,0)</td>
<td>(6,Oddmund,35,03)</td>
<td>6 W</td>
</tr>
<tr>
<td>(29,55,105,0)</td>
<td>(8,Gunnar,38,02)</td>
<td>7 W</td>
</tr>
<tr>
<td>(3,93,159,26)</td>
<td>(9,Stein E.,33,03)</td>
<td>8 D</td>
</tr>
<tr>
<td>(3,93,192,26)</td>
<td>(11,Oystein,35,03)</td>
<td>9</td>
</tr>
<tr>
<td>(3,93,227,26)</td>
<td>(12,Bjorn G.,26,04)</td>
<td>10</td>
</tr>
<tr>
<td>(3,93,202,26)</td>
<td>(14,Yuang,31,03)</td>
<td>11</td>
</tr>
<tr>
<td>(3,93,223,0)</td>
<td>(15,Harald,45,02)</td>
<td>12</td>
</tr>
<tr>
<td>(3,138,231,0)</td>
<td>(16,Ole-H.,31,04)</td>
<td>13</td>
</tr>
<tr>
<td>(3,138,233,0)</td>
<td>(17,Bjorn P.,25,03)</td>
<td>14</td>
</tr>
<tr>
<td>(26,138,258,0)</td>
<td>(18,Oddmund,35,03)</td>
<td>15</td>
</tr>
</tbody>
</table>

Figure 5.8 Example of an execution of Query 4.
5.3 Combining Aggregation with Selection Predicates

Sort-Based Aggregation

The algorithm for the sort-based method is a slight modification of the method proposed by Srinivasan and Carey [SrCa92a]. For this method, an update-table is needed both for \texttt{SUM}-like and \texttt{MIN}-like operators. Two entries in the update-table may be needed for each tuple. In addition to a delete entry based on the before-image of the earliest log record updating the aggregated attribute, the log processing threads will maintain an insert entry based on the after-image of the last log record produced until the tuple is read by the scan thread. This is done by substituting an entry based on the after-image of the current log record for the previous entry if the tuple lies ahead of the scan thread. When the scan is finished, both the scanned relation and the update-table are sorted on the combination of \texttt{BY}-list and primary key. This sorting makes it efficient to calculate the aggregates during the final merge. The compensation is done by removing tuples from subgroups that have an insert entry in the update-table, while tuples with delete entries in the update-table are inserted into the subgroup before the aggregate value of the subgroup is computed.

5.3 Combining Aggregation with Selection Predicates

This section will discuss queries that contain both aggregate operation and selection predicates. In addition to selection predicates on the input to scalar aggregate and aggregate functions, a selection predicate may be used for the temporary relation produced by aggregate functions.

5.3.1 Selection Predicates for Scalar Aggregates

Consider the following query:

\textbf{Query 5.} \hspace{1em} SELECT \texttt{MIN(salary)} FROM employee WHERE \texttt{dept_no} = 03

If partial tuple logging is used, the log processing threads may not be able to decide whether an update to \texttt{salary} will affect the current minimum since it may not have access to \texttt{dept_no}. The solution to this problem is to let a log processing thread insert a write entry in the update-table for log records that do not contain sufficient information. If additional updates are later made to the same tuple, the log processing threads may at a later stage have enough information to process the tuple. In that case, an insert entry will substitute the write entry.

As described above, before the scan thread can update the current minimum, the update-table must be checked if the state identifier indicates that the tuple may have been changed by transactions in the \texttt{AFTER} set. The difference for aggregate queries with a \texttt{WHERE}-clause is that the update-table will also have to be checked in cases where the current tuple does not contain a lower value than the current minimum. The tuple may earlier have contained a lower value, but when processing the log record for the update, the log processing thread may not have been able to evaluate the selection predicate. Hence, if the state identifier indicates that an update has been made to a tuple, the scan thread will have to check the update-table unless the updates needed to bring the tuple from a candidate for a new minimum to its current state would give the log processing threads enough information to handle
this tuple. Using Query 5 as an example, if the salary of a tuple is above the current minimum and the selection predicate is not satisfied, both salary and dept_no would have to be updated to make this tuple a candidate for a new minimum. In that case, the log processing threads would discover this. Hence, the tuple can be ignored by the scan thread.

Figure 5.9 shows an example of an execution of Query 5. The leftmost column shows the current minimum and the neighboring column shows a \( F \) for tuples for which the update-table is checked. The update-table is not checked for tuple 16 since in order for it to have been a candidate for a new minimum, both salary and dept_no must have been updated. In that case, the tuple will have caused the log processing thread to update the current minimum.

Since dept_no is not included in log record 1, a write entry is entered in the update-table. When tuple 1 is later processed, it is discovered that the selection predicate is not satisfied, and the current minimum is not updated. For log records 2 and 3 no entry is made since salary is not changed and the updates could not possibly alter the evaluation of the selection predicate. If the after-image of dept_no in log record 2 had been 03, an insert entry would have been made in order to tell the scan thread to ignore tuple 2.

Log record 4 results in a write entry in the update-table since the before-image of salary is lower than the current minimum. Otherwise, an insert entry would have been made. When processing log record 6, dept_no is entered in the update-
table to tell the scan thread that tuple 16 originally satisfied the selection predicate. When log record 10 later show that the original value of salary was lower is lower than the current minimum, the current minimum is updated, and an insert entry is substituted for the write entry.

If full tuple logging is used, Query 5 could be executed much like input filtering for selection queries. The main difference is that no priority queue is needed. This is because for aggregation queries the processing order of tuples will not matter.

For sum-like operators, an update-table will only be needed if partial tuple logging is used. As for min-like operators, the update-table will in that case be used to tell the scan thread whether a tuple has already been handled by a log processing thread. If complete tuple logging, the update-table will not be needed. In that case the scan thread will apply all tuples that satisfy the selection predicate to the aggregate variable(s). The log processing threads will evaluate the selection predicate for both the before-image and the after-image of a log record. In the case of the sum operation, if the before-image satisfies the selection predicate, its value will be added to the aggregate variable. If the after-image satisfies the selection predicate, its value will be subtracted from the aggregate variable.

Multi-Expression Selection Predicates

If the selection predicate involves more than a single attribute, it will be more complicated for the scan thread to decide whether it needs to check the update-table for the current tuple. Consider a selection predicate of the following type:

\[ p_1 \text{ AND } p_2 \text{ AND } p_3 \text{ AND} \ldots \text{ AND } p_n, \]

assuming that all sub-predicates \( p_1, p_2, p_3, \ldots, p_n \) are single-attribute relational or equality expressions (i.e., \( \text{attribute} \ <\text{rel-op}> \text{rel-op} \ <\text{constant}> \)). A log processing thread should not update the aggregate variable unless this selection predicate evaluates to true. For the above expression, this requires that the log processing thread must be able to evaluate all sub-predicates. As discussed above, the scan thread could only skip checking the update-table for the current tuple when it is sure that if the selection predicate originally evaluated to true, the log records for this tuple would make the log processing threads able to determine this. This means that for the above selection predicate, the scan thread could only skip a tuple when all sub-predicates are false. Otherwise, a tuple that originally satisfied the selection predicate could have had updates to a subset of the attributes used in the selection predicate. In that case, the log processing threads would not have enough information to evaluate the selection predicate.

On the other hand, consider a selection predicate of the following type:

\[ p_1 \text{ OR } p_2 \text{ OR } p_3 \text{ OR} \ldots \text{ OR } p_n, \]

again assuming that all sub-predicates \( p_1, p_2, p_3, \ldots, p_n \) are relational or equality expressions. If a tuple that is read by the scan thread satisfies this selection predicate, the update-table will have to be checked since it may contain updates to the attribute under aggregation. Hence, just like above, checking the update-table may only be skipped when none of the sub-predicates are satisfied. If the tuple originally satisfied the selection predicate, there will be log records containing enough information for the log processing threads to determine this.
Generalizing the above conclusions to queries where the selection predicate is any logical expression of single-attribute relational and equality expressions, gives that the scan thread need not check the update-table if all the relational and equality expressions evaluates to false for the current tuple.

For complex selection predicates the above strategy will probably not be cost-effective. Just as for input filtering, it will cost too much to have both the scan thread and the log processing threads evaluate complex selection predicates. The possible savings by avoiding lookups in the update-table will also be reduced since the chance of all sub-predicates being false will decrease when increasing the number of sub-predicates. In addition, no savings could be made at the presence of expressions like \texttt{<attribute1> <rel-op> <attribute2>} since an update to only one of the attributes will not give a log processing thread enough information to evaluate this expression.

### 5.3.2 Selection Predicates for Aggregate Functions

Consider the following query:

\textbf{Query 6.} \hspace{1em} \texttt{SELECT dept\_no, COUNT(*) FROM employee}
\texttt{WHERE salary < 35}
\texttt{GROUP BY dept\_no}
\texttt{HAVING COUNT(*) > 2}

Query 6 contains selection predicates both on the input and the output of the aggregate function. As discussed earlier, complete tuple logging will be needed for aggregate functions if the log processing threads should take part in the aggregation. Assuming complete tuple logging, the WHERE-clause will be evaluated by both the scan thread and the log processing threads.

For Query 6 the scan thread will increment the count for the relevant department each time it finds a tuple satisfying the WHERE-clause. The log processing threads will increment the counter given by the dept\_no of the before-image each time the before-image satisfies the WHERE-clause. In addition, to compensate for increments made by the scan thread, the counter given by the after-image will be decremented if the after-image satisfies the WHERE-clause. When the scan thread has read the entire relation the resulting relation of tuples (\texttt{dept\_no, count}) will be scanned, and the tuples satisfying the HAVING clause will be emitted.

While no update-table is needed to execute Query 6, an update-table will be needed for queries with \texttt{MIN}-like aggregate operations. The log processing threads will make insert entries in the update-table for tuples that have been updated. In addition, if the before-image satisfies the WHERE-clause, the aggregate variable of the subgroup determined by the before-image may be updated. The scan thread will never update an aggregate variable unless the WHERE-clause is satisfied and no entry exists for the current tuple in the update-table.

If partial tuple logging is used, output filtering must first be applied to find the tuples that satisfy the WHERE-clause. Then, aggregation may be performed on the transaction-consistent relation produced by the selection query. Note that filtering and aggregation may be pipelined. Figure 5.10 illustrates the pipelined execution of Query 6 when using output filtering for the WHERE-clause. Evaluation of the HAVING-clause will have to wait until the aggregation is finished. All the other steps are executed in parallel. Hence, the only temporary storage needed, in addition to the update-table, will be the result of the aggregate function.
5.4 Join Queries

When a query has more than one base relation, the same transaction-consistent state must be used for all relations. That is, the same set of transactions must be reflected in each relation. This is automatically achieved if FLP starts its processing at the same point in the log and BLP use the same AFTER set for all relations.

5.4.1 Input or Output Compensation

As for selection queries, compensation could be done either on the inputs or on the output of the join. If the compensation is done on the inputs, each relation is processed in the same way as described for selection queries. In other words, transaction-consistent output of the different selections queries are joined to produce the result of the join query. The selection queries, the compensation, and the join can be pipelined as discussed above for aggregation queries.

Doing the compensation on the output of the join is generally more complex. It will in many cases not be possible to determine the effect of an update on the result of a join without having access to the input relations. For example, if the join key of a tuple is updated, the tuples matching the new join key will have to be located. Compensation on the join output has similarities to maintenance of materialized views [GuMu95], and will not be treated any further here. Such a strategy could only be cost-effective when the number of tuples in the join result is much smaller than the total number of tuples in the base relations. In that case, the reduced amount of compensation may justify accesses to the base relations.

5.4.2 Join Algorithms

Several algorithms for performing join queries have been proposed [MiEi92]. Most of them could directly be used for joining the transaction-consistent output of selection queries. Below, it will be described how the most popular join algorithms can be combined with compensation-based query processing.
Nested-Loops Join

When joining two relations using nested-loops join, for each tuple of one of the relations (called the inner relation), all tuples of the other relation (called the inner relation) is checked to see if the join condition is satisfied. A more efficient variant is the nested-block join [Haa+97]. In this case, the inner relation is partitioned into chunks that will fit in main-memory. Then, the outer relation is scanned once for each chunk. For equi-joins, efficient matching can be supported by building a hash-table for each chunk. Nested-block joins using a hash-table will be assumed below.

For the inner relation, compensation will be performed on each tuple of a chunk before it is inserted in the hash-table. For the outer relation, compensation is performed for each tuple before a lookup is made in the hash-table. If the inner relation do not fit in main-memory, the outer relation will have to be read in several passes. In order to avoid performing the compensation for the outer relation more than once, the transaction-consistent version could be stored on disk during the first pass. This will require extra temporary storage. However, the space requirements for the update-table will be lower since FLP could stop processing log records for the outer relation when the first pass has completed.

The priority queue for deleted tuples will normally not be needed for nested-loops join. Unless only one pass is needed for the outer the relation, the output order will not be predictable. For the inner relation, the log processing threads could insert deleted tuples into the hash-table. For the outer relation, the log processing threads could perform the join for deleted tuples by doing a lookup in the hash-table. When the hash-table for the next chunk is being built, deleted tuples of the outer relation will have to be temporarily stored until all tuples of the chunk has been entered in the hash-table.

A variant of nested-loops join called indexed nested-loops join may generally not be used for compensation-based query processing. Indexed nested-loops join exploits an index on the join-key of the inner relation to find tuples matching the current tuple of the outer relation. However, since the index reflect the current state of the database, deleted tuples and tuples for which the join-key has been updated, may not be located through the index. One exception is when the join-key of the inner relation is its primary key. In that case, there will be no updates to the join-key, and deleted tuples could be found by checking the update-table.

Sort-Merge Join

Sort-merge join is executed by first sorting both relations on their join-key. Then, the relations are joined by scanning the two relation in parallel. Relations that do not fit in main-memory will have to be sorted in several passes. That is, the relation is partitioned into chunks which can be sorted in main-memory. These sorted chunks are then merged to produce a sorted relation.

The compensation-based version of sort-merge join will perform the compensation on each tuple the first time it is read into main-memory. This way, the sorted chunks will represent a transaction-consistent view of the relation. If a relation is already sorted on its join-key, compensation will have to be performed during the merge. This means that delete entries in the update-table must be inserted in a priority queue in order to preserve the scan order. Updates to the join-key must be
treated like an delete/insert hybrid as discussed in Section 4.3.3 for secondary index scans. This way, the use of a priority queue will ensure that tuples that have had updates to their join key will be relocated to its original position.

Hash Join

Hash-based join methods create a hash-table of the inner relation. This hash-table will be probed with the tuples of the outer relation. If the inner relation does not fit in main-memory, the relations will be partitioned using a hashing function.

For simple hash-partitioned join, the relations will be scanned for tuples belonging to one specific partition at a time. A hash-table is first made of the partition of the inner relation before the outer relation is scanned and the hash-table probed with tuples of the corresponding partition. Tuples not belonging to the current partition will be written to a temporary relation on disk. These relations will be read in the next iteration, looking for tuple of another partition. Using this join method, compensation may be performed on all tuples during the first scan of each relation. This way, the temporary relations will be transaction-consistent. Hence, log processing may end when the scan of the base relations has completed.

Grace hash join is executed in two phases. In the first phase, the partition phase, each relation is partitioned using a hashing function, and the partition written back to disk. The number of partitions are chosen so that each partition of the inner relation will fit in main-memory. In the second phase, the matching phase, hash-tables are built for one partition of the inner relation at a time. Each hash-table is probed with the tuple of the corresponding partition of the outer relation before the next hash-table is built. Compensation will be performed during the partition phase. Thus, the partitions used in the matching phase will be transaction consistent. Hybrid hash join operates as Grace hash join except that the hash-table of the first partition is built and probed during the partition phase. Compensation will still done on input relations. Hence, the hash-table will be transaction-consistent.

5.4.3 Sharing Memory Buffer Between Hash-Tables and Update-Table

For join algorithms that use an in-memory hash-table to find matching tuples, the buffer area used for the tuples of the inner relation could be shared with the update-table (Figure 5.11).

When a tuple of the inner relation is read by the scan thread, the update-table will be checked for an entry for this tuple. If an entry is found, compensation is performed, and the resulting tuple will be written in the buffer space earlier occupied by the tuple’s entry in the update-table. The chains for the corresponding buckets of the update-table and the hash-table join will be updated so that the entry is removed from the bucket of the update-table and the tuple inserted in the bucket of the hash-table.

The above strategy means that if the inner relation fit in main-memory, no extra space will be needed for the update-table of the inner relation. If the inner relation has to be partitioned, its update-table will require some extra space since the update-table will contain entries for all tuples of the relation, not just the tuples of the partition under processing. If log records always will contain the join-key, the update-table could also be partitioned. That is, entries for tuples of later partitions
Figure 5.11 Using the same buffer for the update-table as for the hash-table of the inner relation.

may be stored on disk. When a new partition is to be processed, the entries for this partition will be read from disk.

5.5 Concluding Remarks

This chapter has presented efficient compensation-based algorithms for the execution of relational algebra operations. Pipelining query execution and compensation, will have significant performance gains compared to obtaining a transaction-consistent snapshot for queries to run against.

For selection queries, compensation may be performed either on the input to both the scan thread and the log processing threads, or on the output from the compensation. The idea behind input filtering is that the amount of compensation may be reduced since compensation will only be performed for tuples that satisfy the selection predicate. However, input filtering requires complete tuple logging. A performance analysis also showed that input filtering can only be cost-effective for queries with a low selectivity factor and low predicate evaluation costs.

For aggregation, the savings by doing aggregation on the inputs will be larger. For operations like SUM, COUNT or AVG, an update-table will not be needed at all. For operations like MIN or MAX, compensation need only be performed before updating the aggregate variables. For aggregate functions full tuple logging is needed in order to be able to do aggregation on inputs. When using partial tuple logging, aggregation
must be done on the output from the compensation.

When executing join queries, compensation will be performed on input relations. In other words, transaction-consistent output of selection queries will be joined.
Query Execution
Chapter 6

Analytical Performance Model

In this chapter, a simple analytical performance model for the undo/no-redo compensation algorithm is presented. The analysis will not give an exact prediction of the performance characteristics of the algorithm. However, it will be used to show that the algorithm can be executed efficiently in an OLTP system by comparing the query execution times to that of GO processing.1

6.1 System Load and Capacity

Compared to running queries using GO processing, all extra work added by the algorithm presented in Section 4.2 is CPU-based as long as the update-table is stored in main-memory. Because of this, the performance analysis will be mainly based on the CPU load of an OLTP system. A prerequisite for on-line query processing is that the OLTP system has free capacity that can be used for running queries.

Let $L$ denote the processing load of a system. This load is an abstraction of the CPU load, and the capacity of the system is set to 1.0. The overall load, $L$, of a system is the sum of the load contributed by each of the various tasks that must be served concurrently. For on-line query processing, the load will be divided into four main components:

$L_{\text{OLTP}}$: Load caused by normal OLTP. This is assumed to be constant during the execution of the query.

$L_{\text{Q}}$: Query processing load when running a query without setting any locks and without compensating for concurrent updates (GO processing).

$L_{\text{LP}}$: Load caused by the log processing thread.

$L_{\text{S}}$: Load caused by the compensation part of the scan thread.

---

1GO processing means that the query reads the base relations without acquiring or waiting for locks.
At any time in a system running OLTP transactions and queries concurrently, we have that:

\[ L = L_{\text{OLTP}} + L_q + L_{\text{op}} + L_{\text{c}} \leq 1.0. \tag{6.1} \]

The performance of a query will also be limited by the disk capacity (bandwidth) of the system. Let \( D \) denote the total disk load in the system, and let the capacity be set to \( d \), the number of disks in the system. The disk load consists of the bandwidth used by the OLTP transactions, \( D_{\text{OLTP}} \), and the bandwidth used by the query, \( D_q \), and

\[ D = D_{\text{OLTP}} + D_q \leq d \tag{6.2} \]

must hold at any time.

6.2 Performance Analysis of Producing a Snapshot of a Relation

Let the database consist of a single relation with \( n \) tuples. Concurrently with OLTP transactions, a single query makes a transaction-consistent copy of the relation. To simplify the analysis it is assumed that no OLTP transactions are active at the start of the query (no backward log processing), and that complete tuple logging is used. It is also assumed that the update-table will be stored in the temporary buffer used by the query process. Thus, the buffer hit ratio for OLTP-transactions will not be affected by the size of the update-table. Table 6.1 lists the parameters used for the performance analysis. The settings used for these parameters are based on measurements made on a single-node version of the ClustRa DBMS [Hva+95].

The main components of the OLTP load are executing transactions and their database operations, fetching blocks from disk into the database buffer, logging of operations to disk, and writing dirty pages to disk during checkpoint. Let \( \alpha_r = n_r \alpha_t \) and \( \alpha_w = n_w \alpha_t \) be the access rate of read and write operations, respectively, to the database. Then the following equation approximates the OLTP load of the system:

\[
L_{\text{OLTP}} = \alpha_r W_t + \alpha_r (W_{tr} + (1 - h) W_{sw}) + \alpha_w \left( W_{rw} + (1 - h) W_{sw} + W_{\log} + \frac{W_{sw}}{n_{\log}} \right) + n_{bw} W_{sw} t_{checkpoint}, \tag{6.3}
\]

where \( n_{\log} \) is the average number of log records in a log page, and \( n_{bw} \) is the average number of pages updated during one checkpoint interval. It is assumed that group commit is used, and that transactions are delayed a maximum of \( t_{checkpoint} \) seconds. Assuming that \( n_p / 2 \) log records fit in one page, this gives \( n_{\log} = \min(n_p/2, \alpha_w t_{checkpoint}) \).

Assuming a uniform access pattern among \( n \) data objects, the probability that a specific data object has been updated after \( x \) write operations will be \( 1 - (1 - 1/n)^x \).

Thus, the probability that a page is written to disk during a checkpoint interval

\footnotesize{\textsuperscript{2}The settings for the different CPU operations is given in number of instructions. The actual values used for the constants will be the given number of instructions divided by the CPU rate, \( \frac{\text{CPU}}{\text{CPU}} \).}
### Table 6.1 Parameters used in the performance analysis.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W_t$</td>
<td>Cost of starting and committing a transaction</td>
<td>80000 instr.</td>
</tr>
<tr>
<td>$W_{rr}$</td>
<td>Cost of performing a read operation to a random tuple in the buffer</td>
<td>35000 instr.</td>
</tr>
<tr>
<td>$W_{rw}$</td>
<td>Cost of performing a write operation to a random tuple in the buffer (not including logging)</td>
<td>40000 instr.</td>
</tr>
<tr>
<td>$W_s$</td>
<td>Cost of sequential access to one tuple</td>
<td>8000 instr.</td>
</tr>
<tr>
<td>$W_{io}$</td>
<td>Cost of transferring a page between memory and disk</td>
<td>50000 instr.</td>
</tr>
<tr>
<td>$W_{log}$</td>
<td>Cost of creating a log record</td>
<td>7000 instr.</td>
</tr>
<tr>
<td>$W_{log}$</td>
<td>Cost of processing a log record</td>
<td>1400 instr.</td>
</tr>
<tr>
<td>$W_{hash}$</td>
<td>Cost for a hash lookup into the update-table</td>
<td>1000 instr.</td>
</tr>
<tr>
<td>$W_{int}$</td>
<td>Cost of inserting an entry into the update table</td>
<td>7500 instr.</td>
</tr>
<tr>
<td>$B_r$</td>
<td>Disk bandwidth used for random access to a disk block</td>
<td>0.008</td>
</tr>
<tr>
<td>$B_s$</td>
<td>Disk bandwidth used for sequential access to a disk block</td>
<td>0.0005</td>
</tr>
<tr>
<td>$r_{CPU}$</td>
<td>Instruction rate of CPU</td>
<td>300 MIPS</td>
</tr>
<tr>
<td>$n_r$</td>
<td>Number of read operations in each transaction</td>
<td>2</td>
</tr>
<tr>
<td>$n_w$</td>
<td>Number of write operations in each transaction</td>
<td>4</td>
</tr>
<tr>
<td>$n_p$</td>
<td>Number of tuples in one page</td>
<td>30</td>
</tr>
<tr>
<td>$h$</td>
<td>Database buffer hit ratio</td>
<td>0.8</td>
</tr>
<tr>
<td>$t_{ckpt}$</td>
<td>Time interval between checkpoints</td>
<td>5 s</td>
</tr>
<tr>
<td>$t_{grpt}$</td>
<td>Time interval between group commits</td>
<td>0.5 s</td>
</tr>
<tr>
<td>$d$</td>
<td>Number of disks in the system</td>
<td>4</td>
</tr>
<tr>
<td>$n$</td>
<td>Number of tuples in the database</td>
<td>$10^6$</td>
</tr>
<tr>
<td>$\alpha_t$</td>
<td>Arrival rate of transactions</td>
<td>250 per sec.</td>
</tr>
</tbody>
</table>

is, assuming uniform distribution of access, $1 - (1 - 1/(n/n_p))^{n_{log}}$. Assuming a 20/80 access pattern\(^3\) gives:

$$n_{lmw} = \frac{0.2n}{n_p} \left( 1 - \left( 1 - \frac{n_p}{0.2n} \right)^{0.8n_{log}} \right) + \frac{0.8n}{n_p} \left( 1 - \left( 1 - \frac{n_p}{0.8n} \right)^{0.2n_{log}} \right).$$

The disk bandwidth used by OLTP transactions corresponds to the I/O part of (6.3). It will be assumed that the log, indexes, and temporary data are placed on separate disks that have sufficient bandwidth for supporting both the OLTP transactions and the query. Thus, only the disks used for storing the relation will be modeled:

$$D_{oltp} = \alpha_r (1 - h) B_r + \alpha_w (1 - h) B_r + \frac{n_{lmw} B_s}{t_{ckpt}},$$  \hspace{1cm} (6.4)

assuming that the checkpoint algorithm writes the dirty pages in physical order.

The work done by the log processing thread has three main components: The work associated with reading a log record from the log and extracting the primary key ($W_{log}$); the work associated with hashing into the update-table to see if there is an entry for this primary key ($W_{hash}$); and the work associated with extracting the relevant information from the log record and inserting it into the update-table ($W_{int}$).

\(^3\) 80\% of the accesses goes to 20\% of the pages, and vice versa.
Since complete tuple logging is used, the last operation will only be performed if an entry is not found in the update-table. Let \( f(t) \) be the probability that an entry for a given tuple already exists in the update-table at time \( t \) after the start of the query. Assuming a 20/80 tuple access pattern gives:

\[
f(t) = 0.2 \left( 1 - \left( 1 - \frac{1}{0.2n} \right)^{0.8\alpha_q t} \right) + 0.8 \left( 1 - \left( 1 - \frac{1}{0.8n} \right)^{0.2\alpha_q t} \right),
\]

The load caused by the log processing thread can now be modeled as:

\[
L_p(t) = \alpha_q(W_{\log} + W_{\text{hash}} + (1 - f(t))W_{\alpha_q}). \tag{6.5}
\]

When executing the query, it is assumed that the entire relation is read from disk and the output written to a temporary relation on disk. The load caused by the query if executed using GO processing will be:

\[
L_q(t) = \alpha_q(t) \left( \frac{W_{\alpha_q}}{n_p} + W_{\alpha_q} + \frac{W_{\alpha_q}}{n_p} \right), \tag{6.6}
\]

where \( \alpha_q(t) \) is the rate at which the query is processed at time \( t \). The definition of \( \alpha_q(t) \) implies:

\[
\int_0^{\tau_q} \alpha_q(t) dt = n, \tag{6.7}
\]

where \( \tau_q \) is the time it takes to process the query.

The extra work performed by the scan thread when doing compensation is to check the update-table for an entry for the current tuple. By checking the state identifier of the tuple, this operation can be avoided for tuples that have not been changed since the start of the query. In other words:

\[
L_c(t) = \alpha_q(t)f(t)W_{\text{hash}}. \tag{6.8}
\]

The disk bandwidth used by the query will be:

\[
D_q(t) = \frac{\alpha_q(t)B_{\alpha_q}}{n_p}. \tag{6.9}
\]

The rate at which the query is processed, \( \alpha_q(t) \), is limited by both the CPU capacity and the disk capacity (bandwidth). The maximum processing rate at any time \( t \) will be the maximum value which can be chosen for \( \alpha_q(t) \) such that both (6.1) and (6.2) hold.\(^4\) Having computed \( \alpha_q(t) \), (6.7) can be solved for \( \tau_q \) in order to compute the query response time.\(^5\)

Figure 6.1 shows the CPU load when executing the query with the parameter settings given in Table 6.1. The extra work required by the compensation-based algorithm uses about 3.3% of the available CPU time. Most of this overhead is caused by the log processing. The load used for the compensation \( (L_c(t)) \) never exceeds 0.003 for this query. The query uses 126.7 seconds to process the 1 million tuples. This is 6.7% longer than for GO processing, 10.4% of the tuples were entered in the update-table at the end of the query.

\(^4\) This assumes that the query will read all disks in parallel. If the query reads one disk at the time then \( D_{\alpha_q}/d + D_q \leq 1 \).

\(^5\) In order to be able to solve the integral equation a polynomial Pade approximation [GeGi96] was computed for \( \alpha_q(t) \). The approximation was only used for values of \( t \) for which the difference to \( \alpha_q \) was less than 1%.
6.2 Performance Analysis of Producing a Snapshot of a Relation

![Performance Analysis Graph](image)

**Figure 6.1** Cumulative CPU load for compensation-based query processing.

As shown by Figure 6.2, query performance decreases with increasing OLTP load. The relative difference between the execution times for GO processing and compensation-based processing also increases as long as the query-processing is CPU-bound (Figure 6.3). With no OLTP load, the compensation gives no overhead since the scan thread will only check the update-table for tuples that have been updated by concurrent OLTP transactions. At \( L_{olf} = 0.50 \) the overhead has increased to 7.5%. When increasing the OLTP load to the point when GO processing becomes disk-bound, only a little extra increase in the load will also make compensation-based query processing disk-bound. This can be seen from the sharp drops in the performance ratios in Figure 6.3. When both approaches are disk-bound, the performance of the compensation-based algorithm will be equal to that of GO processing.

The size of the update-table will also increase when increasing the OLTP load (Figure 6.4). Increasing the load means both increased update rates and longer execution times, both of which leads to an increase in the total number of concurrent updates. For low OLTP loads, the number of entries in the update-table is relatively small. However, at \( L_{olf} = 0.65 \), 20% of the tuples will have entries in the update-table. Even when using the space optimization techniques presented in Section 4.3.2, the size of the update-table may require too much resources at very high loads.

When increasing the size of the base relation, the relative overhead in execution time compared to GO processing slightly increases (Figure 6.5). This is most likely because the OLTP load of the system increases due to more work associated with checkpoints. Note that Figure 6.5 assumes that the buffer hit ratio remain constant. In other words, the database buffer is scaled with the size of the database. In
Figure 6.2 Query execution times when varying OLTP load.

Figure 6.3 Relative query performance compared to GO processing when varying the number of disks in the system.
6.2 Performance Analysis of Producing a Snapshot of a Relation

![Graph](image)

**Figure 6.4** Relative size of the update-table when varying OLTP load.

![Graph](image)

**Figure 6.5** Relative query performance compared to GO processing when varying the size of the base relation. (Logarithmic scale on the horizontal axis.)
addition, the model does not take into account that the disk bandwidth used from random access \( (B_r) \) will probably increase when storing a larger data volume on the same number of disks.

6.3 Discussion

This performance analysis shows that compensation-based query processing can be efficiently performed in OLTP systems with low to medium system utilization by OLTP transactions. At such loads, the overhead to achieve transaction-consistency is low. The overhead compared to GO processing increases more than linearly with increasing OLTP load. At high system utilization, the overhead may become large. On the other hand, it is not surprising that introducing query processing in a highly utilized OLTP system will require extra system capacity.

The simple analytical model above assumes that the base relation is the only relation in the database. If this is not the case, the log processing thread does not have to process log records of relations that are not accessed by the query. In that case, both the compensation overhead and the size of the update-table will be smaller. On the other hand, if partial tuple logging is used, more work has to be done by the log processing thread. However, the size of the update-table will probably decrease as not all relevant attributes of a tuple will usually have been updated.

The model above assumes that the throughput of transactions are not affected by concurrent queries. Experience shows that this is a fair assumption as long as there is some spare capacity in the system. This capacity-based model can not be used to analyze how response times of transactions are affected by concurrent queries. For that purpose a queuing model or simulations are needed. This topic will be further investigated in the next chapter which presents simulation experiments on compensation-based query processing.
Chapter 7

Performance Evaluation of Compensation-Based Query Processing

This chapter presents an evaluation of the performance of compensation-based query processing. The evaluation is based on simulation experiments. The simulation model was implemented in the C++-based CSIM18 simulation language [Schw96, Mes]. The performance of undo/no-redo compensation was compared to the performance of other query algorithms. In the first section of this chapter, the query algorithms used in the simulation are presented and the details of their implementation are discussed. Then, the simulation model is described before presenting experiments and results.

7.1 Implementation Issues

In this section, implementation choices concerning the simulated query algorithms are discussed. In addition to the compensation-based algorithms presented in this thesis (UNDO compensation) and by Srinivasan and Carey (REDO compensation), two-phase locking, and GO processing was simulated.

Since the context of this work is to be able to do query processing without significantly reducing the performance of transactions, the queries were, where not otherwise stated, given less priority than transactions by the CPU scheduler. Threads executing as part of the query were only scheduled for the CPU if no other types of threads were requesting the CPU.

7.1.1 UNDO Compensation

UNDO compensation is implemented according to the presentation in Chapter 4.2. The update-table is assumed to be stored as a hash table in a temporary main memory buffer separate from the database buffer.\(^1\) Thus, the size of the update-

\(^1\)It is common for a DBMS to reserve separate main memory storage for query operations (e.g., join, sort). When a query is executed, a main memory area will be assigned to the query for the
Log

|-----|--------|--------|--------|--------|--------|--------|--------|--------|--------|

Transaction table

<table>
<thead>
<tr>
<th>Tid</th>
<th>State</th>
<th>LastLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Committed</td>
<td>44</td>
</tr>
<tr>
<td>11</td>
<td>Active</td>
<td>45</td>
</tr>
<tr>
<td>12</td>
<td>Committed</td>
<td>48</td>
</tr>
<tr>
<td>13</td>
<td>Active</td>
<td>48</td>
</tr>
<tr>
<td>14</td>
<td>Active</td>
<td>47</td>
</tr>
<tr>
<td>15</td>
<td>Active</td>
<td>—</td>
</tr>
</tbody>
</table>

Backward Log Processing:

<table>
<thead>
<tr>
<th>Actions</th>
<th>List of LSNs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>45 46 47</td>
</tr>
<tr>
<td>Process LSN 47</td>
<td>45 46</td>
</tr>
<tr>
<td>Process LSN 46</td>
<td>45 43</td>
</tr>
<tr>
<td>Process LSN 45</td>
<td>40 43</td>
</tr>
<tr>
<td>Process LSN 43</td>
<td>40</td>
</tr>
<tr>
<td>Process LSN 40</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.1 The implementation of backward log processing. Example of how the list of LSNs is used to implement the AFTER set.

table will not affect the hit ratio of the database buffer. Since the simulation model assumes complete tuple logging, information is never entered into the update-table more than once for each tuple during forward log processing. Where not otherwise stated, the space optimization techniques presented in Section 4.3.2 are not applied. In other words, information is entered for all log records representing updates of tuples of the base relation, and no entry is deleted from the update-table before the query is finished.

The AFTER set is implemented as a list of LSNs. At the start of the query, the transaction table is inspected, and the LSNs of the last log record produced by each active or aborting transaction are entered into the list. This means that active transactions that have not yet performed any operations, will not have any entry in the list that represents the AFTER set. When doing backward log processing, the maximum LSN is removed from the list, the corresponding log record is accessed, and the relevant information is entered into the update-table. If this LSN does not represent the first log record of the transaction, the LSN of the previous log record for the same transaction is then entered into the list. This process is repeated until there are no more entries in the list.

Figure 7.1 shows an example of how this list is maintained during backward log processing. The transaction table shown represents the state of transactions at the start of the query. For each transaction, the LSN of its last log record is shown in addition to its ID and state. For each log record, the ID of the transaction that created the log record and a pointer to the previous log record by this transaction are shown. Finally, the figure shows the current list of LSNs after processing each log record. At any time, the list contain one LSN for each transaction of the AFTER set for which all its log records has not yet been processed.

A priority queue is used to make sure that the scan order is preserved for deleted tuples. When a log record for a delete operation of a tuple that has not yet been read by the scan thread is processed, a pointer to the entry in the update-table...
7.1 Implementation Issues

for this tuple is entered together with the primary key\(^2\) of the deleted tuple. The priority queue is implemented as a partially ordered binary tree ([Aho\textsuperscript{88}]) ordered on primary key.

The scan thread will check the primary key of the entry at the head of the priority before emitting a tuple. If this key is smaller than the primary key of the current tuple, the entry is removed from the primary queue, and the deleted tuple is emitted using the information found in the update-table. In order to be sure that all log records representing relevant deleted tuples have been processed, the scan thread waits until the log processing thread has processed the log record with LSN equal to the state identifier of the current page.

The implementation of UNDO compensation requires that log records are processed before they are removed from the log buffer. If more than half the log buffer has not yet been processed, the log processing thread is given transaction priority. When all log records have been processed, the priority of the log processing thread is reduced to its previous value. If the transaction load is so high that log records are created at a higher rate than the log processing rate at increased priority, the query will be aborted when the first log page containing unprocessed log records is removed from main memory. Only backward log processing may fetch log records from disk.

7.1.2 REDO Compensation

REDO compensation is implemented as described in [SrCa92a] except that the update-list is implemented as a hash table instead of as a sequential list. This way, sorting the update-list between the scan phase and the compensation phase is not necessary. The hash-table will contain maximum one entry per tuple.

In order to avoid reading uncommitted data, a read lock is acquired before a tuple is read. When the query has finished processing a tuple, the lock is released before the next tuple is locked. Information on updates made by a transaction is entered into the update-table at commit time. The only extra CPU cost modeled for transactions is the cost associated with inserting an entry into the update-list. Neither the cost to access information on concurrent queries nor the cost to keep a list of all updates of transactions until commit time are modeled. Hence, the experiments will show somewhat better performance for transactions when performing REDO compensation than what could be expected from an implementation of the algorithm.

7.1.3 Two-Phase Locking

All tuple accesses by queries are covered by primary-key range locks. Before accessing a new page, a query extends its key range lock to cover all tuples of the page. When the query is finished, the range lock is released. Note that the simulation model assumes that transactions are aborted if they have not finished within a specific time limit. Thus, long queries will cause many transactions to be aborted while waiting for locks.

\(^2\)All scans are done in primary key order.
7.1.4 GO processing

When queries are executed using GO processing, no locks are waited for or set by the query, and all tuples are emitted in the form they are read. In other words, the query result will not be transaction-consistent. The performance of GO processing will be used as a reference to how much performance must be sacrificed in order to achieve transaction-consistency.

7.2 Simulation Model

The simulation model presented below is divided into two main parts, the system model and the application model. The first part models the behavior of the DBMS and its resources, while the latter part models the database and transaction and query workloads.

7.2.1 System Model

Below, the components of the system model for the database simulator are briefly described. A more detailed description of the system model can be found in Appendix A.

The system model encapsulates the logical and physical resources of a DBMS and its underlying operating system and hardware. The system model consists of a single CPU, a single disk manager which administers several disks, a buffer manager, a checkpoint manager, a lock manager, and a log manager. In addition, the system has several transaction managers and query managers, each executing a single transaction or query, respectively, at a time.

CPU module

The CPU module models the behavior of the CPU scheduler. The scheduler is priority-based and non-preemptive. It is assumed that the DBMS is run as a collection of light-weight threads in a single process. The CPU scheduler will assign the CPU to the requesting thread with the highest priority. In case of ties in priorities, the scheduler uses a first-come, first-served (FCFS) policy. Threads executing queries are given lower priority than threads executing transactions.

Disk Manager

The disk manager will receive requests from the buffer manager to transfer a certain number of blocks starting with a given block ID between the disk and the database buffer. Based on the block ID, the disk manager will direct each request to the corresponding disk. Asynchronous I/O towards raw disk devices is assumed. Disk requests are scheduled using a FCFS policy.

The service time for a disk request is computed as the sum of the seek time, settle time, rotational latency, and transfer time in addition to a disk controller overhead. The model of the seek time is based on [RuWi91] and is dependent on the seek distance \(d\), the number of cylinders in between the cylinders of the previous and the current request. For small seek distances, the seek time is proportional to the square root of the distance, while for larger seeks, the seek time is linearly dependent...
on the seek distance. The rotational latency is chosen uniformly over the range 0 to
\textit{DiskMaxLatency}. The settings of the disk parameters is based on the data sheet for
the Seagate Cheetah 4LP disks and on measurements presented in [Wor+95].

**Buffer Manager**

The buffer manager handles the database buffer using an LRU replacement policy.
The database buffer is shared among data pages and index pages. A data page can
be accessed either through the buffer hash table based on its block ID, or the page
holding a particular tuple can be found using a B-tree index. For simplicity, it is
assumed that the B-tree has a fixed size, and that index pages are never updated.

After a transaction has requested a page in the buffer, that particular page is
guaranteed to reside in the buffer until the transaction has performed the next
operation to a tuple on this page. The buffer manager also supports requests to
read and write several physical adjacent pages in a single disk request. To avoid
that pages from large sequential scans fill the entire buffer, such pages are inserted
at the front of the LRU list.

**Checkpoint Manager**

All flushing of dirty pages in the database buffer to disk is normally done by the
checkpoint manager. The checkpoint manager is implemented as a separate thread
which is activated when the number of dirty pages gets high. Before writing the
dirty pages to disk, the references to these pages are sorted in order to achieve
sequential access to the disks. All disks are written to in parallel.

**Lock Manager**

The lock manager implements key range locking in addition to ordinary tuple-level
locking. Transactions are executed using strict 2PL. Read and write locks are automati-
cally set when a transaction performs read or update operations, respectively.
Read locks are upgraded if a transactions updates or deletes a tuple it has previ-
ously read. No deadlock detection is implemented. Deadlocks are resolved by the
transaction managers which abort transactions that have not terminated within a
given time limit.

**Log Manager**

The log manager maintains a buffer of the most recent log pages. For each update
performed by a transaction, the log manager inserts log records into the current
log page. Complete tuple redo/undo logging is used, and it is assumed that each
log record will occupy twice as much space as the tuple that was updated. When a
transaction is aborted, compensation log records (CLR) are created for all operations
that are undone. If a log record that is no longer residing in the log buffer is needed
either during undo processing or during backward log processing, the corresponding
log page will be read from disk. Log records are flushed to disk either upon request
by the checkpoint manager or when transactions are committed.
Transaction Manager

The number of transaction managers in the system is given by the multiprogramming level (MPL) of the system. Each transaction manager executes a single transaction at a time. Once its current transaction has terminated, the transaction manager immediately starts a new transaction. If the execution time of a transaction exceeds a given time limit, it is aborted since it has probably become part of a deadlock.

In order to reduce the work associated with flushing the necessary log records to disk, a group commit policy is used. A separate group commit thread will, when starting a group commit, commit all transactions that are ready to commit at that point in time. As long as there are transactions that are ready to commit, the group commit thread will start a new group commit as soon as it has completed the previous one. This is similar to the method used by the Oracle DBMS [Ora95]. If no transaction is waiting to commit, the group commit thread will wait until the next transaction has finished its operations before starting a new commit.

Query Manager

Each query manager runs a single query at a time. When one query is finished a new query is immediately started. The execution of a query involves at least three separate threads. In addition to the scan thread, a query will have a read and a write thread. The read thread reads one batch of pages at a time from disk. When the scan thread starts processing such a batch, the read thread will request a new batch from the buffer manager. The query result is stored in a temporary buffer, and the write thread writes batches of pages from the temporary buffer to disk. If UNDO compensation is used, the query will also include a forward log processing thread. Backward log processing is performed by the scan thread. In order to minimize the effects on transaction processing, a scan thread or a log processing thread will only process a single tuple or log record, respectively, each time it is scheduled for the CPU.

The main parameters of the system model are presented in Table 7.1. The values used for the CPU cost parameters are based on measurements done on the ClustRa DBMS [Hva+95] and is presented in Table 7.2.

7.2.2 Application Model

The model of the database is based on the TPC-B specification [Tra94]. The database consist of four relations as shown in Table 7.3. In the experiments, the database size is fixed and not scaled to the transaction load as required by the TPC-B specification. In addition, in order to be able to support a higher number of concurrent transactions without scaling the database, the number of accounts per branch was set to 2000 and not 100000 as required by the specification. For each relation, the fraction of the maximum number of tuples that exists at the start of a simulation can be specified. Each relation is stored as a clustered B+-tree indexed on primary key. NumDisks disks is used to store the Account relation. One disk is used for each of the other relations, and three additional disks are used for indexes, temporary tables, and the log.

The workload of the system is modeled as a fixed set of terminals, each either requesting the execution of transactions or queries. Each terminal only submits one
7.3 Experiments and Results

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPURate</td>
<td>Instruction rate of CPU</td>
</tr>
<tr>
<td>BufSize</td>
<td>Number of pages in the database buffer</td>
</tr>
<tr>
<td>DiskBlockSize</td>
<td>Disk block size</td>
</tr>
<tr>
<td>DiskMaxLatency</td>
<td>Maximum rotational delay</td>
</tr>
<tr>
<td>DiskTransfer</td>
<td>Disk transfer rate</td>
</tr>
<tr>
<td>MaxSeqIO</td>
<td>Maximum number of pages in a sequential read/write</td>
</tr>
<tr>
<td>LogBufSize</td>
<td>Number of log pages in log buffer</td>
</tr>
<tr>
<td>TransMaxTime</td>
<td>Execution time limit for transactions</td>
</tr>
<tr>
<td>SchedCPU</td>
<td>Overhead for thread switch</td>
</tr>
<tr>
<td>BufMissCPU</td>
<td>Cost to find a page in the database buffer</td>
</tr>
<tr>
<td>BuffMissCPU</td>
<td>Cost to handle a miss in the database buffer</td>
</tr>
<tr>
<td>DiskXferCPU</td>
<td>Cost to set up an asynchronous disk request</td>
</tr>
<tr>
<td>DiskPollCPU</td>
<td>Cost to poll for a finished disk request</td>
</tr>
<tr>
<td>LockReqCPU</td>
<td>Cost to request a lock</td>
</tr>
<tr>
<td>LockRelCPU</td>
<td>Cost to release a lock</td>
</tr>
<tr>
<td>LogCrRecCPU</td>
<td>Cost to create a log record</td>
</tr>
<tr>
<td>TransInitCPU</td>
<td>Cost to initiate a transaction</td>
</tr>
<tr>
<td>TransCommitCPU</td>
<td>Cost to commit a transaction</td>
</tr>
<tr>
<td>TransAbortCPU</td>
<td>Cost to abort a transaction</td>
</tr>
<tr>
<td>ReadCPU</td>
<td>Cost to execute a read operation</td>
</tr>
<tr>
<td>UpdateCPU</td>
<td>Cost to execute a write operation</td>
</tr>
<tr>
<td>DeleteCPU</td>
<td>Cost to execute a delete operation</td>
</tr>
<tr>
<td>InsertCPU</td>
<td>Cost to execute an insert operation</td>
</tr>
<tr>
<td>TupProcCPU</td>
<td>Cost for the scan thread to process a single tuple</td>
</tr>
<tr>
<td>TupEmiCPU</td>
<td>Cost for the scan thread to emit a single tuple</td>
</tr>
<tr>
<td>LogProcCPU</td>
<td>Cost for the log processing thread to process a log record</td>
</tr>
<tr>
<td>UpTableCPU</td>
<td>Cost to insert an entry in the update-table</td>
</tr>
<tr>
<td>UpTabOutCPU</td>
<td>Cost to do a lookup in the update-table</td>
</tr>
</tbody>
</table>

job at a time, and a new job is submitted at once its previous job has terminated.

Queries are executed by scanning a given fraction, a specified primary key range, of the Account relation in primary key order. The selectivity factor of the query determines the size of the key range. Two types of transactions are used in the experiments, TPC-B transactions and transactions that use a random function to determine the number and types of operations. The two types of transactions are described in more detail when presenting the experiments.

The list of parameters for the application model is given in Table 7.4.

### 7.3 Experiments and Results

Below, the results of the simulation experiments are presented. The main performance metrics of the experiments are query response time and transaction through-
Table 7.2 Parameter settings.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Setting</th>
<th>Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPURate</td>
<td>300 MIPS</td>
<td>LockRelCPU</td>
<td>300 instructions</td>
</tr>
<tr>
<td>BufSize</td>
<td>4096 pages</td>
<td>LogCreRecCPU</td>
<td>7000 instructions</td>
</tr>
<tr>
<td>DiskBlockSize</td>
<td>4 kBytes</td>
<td>TransInitCPU</td>
<td>35000 instructions</td>
</tr>
<tr>
<td>DiskMaxLatency</td>
<td>6.0 msec</td>
<td>TransCommitCPU</td>
<td>40000 instructions</td>
</tr>
<tr>
<td>DiskTransfer</td>
<td>13.7 MB/sec</td>
<td>TransAbortCPU</td>
<td>40000 instructions</td>
</tr>
<tr>
<td>MaxSeqIO</td>
<td>32 pages</td>
<td>ReadCPU</td>
<td>8000 instructions</td>
</tr>
<tr>
<td>LogBufSize</td>
<td>40 pages</td>
<td>UpdateCPU</td>
<td>15000 instructions</td>
</tr>
<tr>
<td>TransMaxTime</td>
<td>5 secs</td>
<td>DeleteCPU</td>
<td>6000 instructions</td>
</tr>
<tr>
<td>SchedCPU</td>
<td>180 instructions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BufMissCPU</td>
<td>1500 instructions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DiskXferCPU</td>
<td>60000 instructions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DiskPollCPU</td>
<td>50000 instructions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LockReqCPU</td>
<td>1500 instructions</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2000 instructions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.3 Relations in the TPC-B database.

<table>
<thead>
<tr>
<th>Relation Name</th>
<th># Tuples (Max)</th>
<th>Tuples/Page</th>
<th>Pages</th>
<th>Index Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account</td>
<td>1000000</td>
<td>30</td>
<td>33334</td>
<td>3</td>
</tr>
<tr>
<td>Branch</td>
<td>500</td>
<td>30</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>Teller</td>
<td>5000</td>
<td>30</td>
<td>167</td>
<td>2</td>
</tr>
<tr>
<td>History</td>
<td>100000</td>
<td>60</td>
<td>1667</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 7.4 Application model parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumDisks</td>
<td>Number of disks used for the Account relation</td>
</tr>
<tr>
<td>MPL</td>
<td>Number of terminals submitting transactions</td>
</tr>
<tr>
<td>QMPL</td>
<td>Number of terminals submitting queries</td>
</tr>
<tr>
<td>AvgOp</td>
<td>Average number of operations per transaction</td>
</tr>
<tr>
<td>Sel</td>
<td>Selectivity factor for queries</td>
</tr>
<tr>
<td>KeysPP</td>
<td>Number of keys per index page</td>
</tr>
<tr>
<td>Fill&lt;sub&gt;rel&lt;/sub&gt;</td>
<td>Fraction of max. tuples existing in relation rel at startup</td>
</tr>
</tbody>
</table>
7.3 Experiments and Results

BEGIN WORK;
  UPDATE accounts
    SET Abalance = Abalance + :delta
    WHERE AccountId = :Aid;
  SELECT Abalance INTO :Abalance
    FROM accounts
    WHERE AccountId = :Aid;
  UPDATE tellers
    SET Tbalance = Tbalance + :delta
    WHERE TellerId = :Tid;
  UPDATE branches
    SET Bbalance = Bbalance + :delta
    WHERE BranchId = :Bid;
  INSERT INTO history(Tid, Bid, Aid, delta, time)
    VALUES (:Tid, :Bid, :Aid, :delta, CURRENT);
  COMMIT WORK;

Figure 7.2 TPC-B transaction profile.

In addition, the storage needed to store the update-table and the time used for backward log processing are studied.

Based on these metrics the UNDO compensation-based algorithm is compared with the following query algorithms: Two-phase locking, GO processing, and the REDO compensation-based algorithm presented by Srinivasan and Carey. Two different types of transactions have been used in the experiments. In the first experiment, TPC-B transactions are used while in the second experiment, the number and types of operations transaction are randomly determined. A third experiment was made in order to investigate the effects of the space optimization techniques to the update-table. In all experiments, queries make a copy of a given fraction of the Account table.

For all measurements, statistical validity was ensured by verifying that the 90% confidence interval were sufficiently tight. This was done by repeating the experiments at least until the size of the confidence intervals were, if not otherwise stated, within 1% of the mean. The confidence intervals were computed using the replication method for steady-state simulations [Ban96].

7.3.1 Update-Intensive Experiment

The transactions used in this experiment are similar to the transactions used in the Transaction Processing Performance Council (TPC) Benchmark B and models a deposit to or withdrawal from a bank account. All transactions executes the code shown in Figure 7.2 against a database consisting of the relations given in Table 7.3. A uniform random function is used to decide TellerId. In 85% of the cases BranchId

\footnotetext{2}{Since for each simulation run, transactions are generated by a constant number of terminals, transaction throughput is really a measure of average transaction response time. The only difference is that throughput is affected by the abort rate of transactions.}
Table 7.5 Application model parameters for the update-intensive experiment.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumDisks</td>
<td>8</td>
</tr>
<tr>
<td>MPL</td>
<td>1, 3, 6, 10, 15, 21, 28</td>
</tr>
<tr>
<td>QMPL</td>
<td>1</td>
</tr>
<tr>
<td>AvgOp</td>
<td>5 (Constant)</td>
</tr>
<tr>
<td>Sel</td>
<td>2%, 5%, 10%, 25%, 50%, 100%</td>
</tr>
<tr>
<td>KeysPP</td>
<td>150</td>
</tr>
<tr>
<td>Fill{Account, Branch, Teller}</td>
<td>100%</td>
</tr>
<tr>
<td>Fill_{History}</td>
<td>0%</td>
</tr>
</tbody>
</table>

is given by TellerId. In the other cases BranchId is chosen uniformly among the other branches, AccountId is chosen uniformly among the accounts belonging to the given branch. All branches contain the same number of tellers and the same number of accounts. A sequence number is used as primary key in the History relation. Thus, the tuples will be inserted in physical order. As given in Table 7.3, there is a maximum on the number of tuples in the History relation. When this number is exceeded, the sequence number generator is reset, and previous tuples are overwritten when new tuples are inserted.

In this experiment, no transaction is aborted except when using two-phase locking (2PL) for queries. Using 2PL, aborts happen because transaction execution times sometimes will exceed the maximum limit. The application parameters for this experiment are shown in Table 7.5. The experiment was run over a range of multiprogramming levels (MPL) and query selectivities (Sel). The utilization of the CPU and the disks holding the Account relation when running TPC-B transactions is shown in Figure 7.3.

**Query Response Time**

Figure 7.4 compares the query response time for the different query algorithms used in the experiment. Queries could not be run using UNDO compensation at MPL = 28 because log records were created at a faster rate than the processing rate of the log processing thread. Executing the query using two-phase locking (2PL) gives the lowest response times. However, as will be shown below, using 2PL also results in a significant lower transaction throughput. In fact, the reason for the good query performance is that the system utilization by transactions is reduced due to lock contention.

The experiment shows that UNDO compensation only gives a slight increase in response-times compared to GO processing. This increase represents the work associated with processing the log, maintaining the update-table, and compensating for concurrent updates. REDO compensation give more than twice as high query response times as UNDO compensation. This is as expected since the two-phased approach requires intermediate storage of data on disk. Figure 7.5 shows that the

---

4By inadvertence, this is a slight deviation from the TPC-B specification which specify that in BranchId is always determined by TellerId, while AccountId in 85% of the cases should belong to the branch of the chosen teller.
7.3 Experiments and Results

Figure 7.3 System utilization when running TPC-B transactions.

Figure 7.4 Query response times for various MPL. (Set = 10%)
relative query response times of all algorithms except for two-phase locking are independent of the selectivity of the query. Two-phase locking shows better performance with increasing selectivity. This is because increasing the selectivity also increases the number of transactions that are blocked by the query.

Figure 7.6 compares the performance of UNDO compensation and GO processing. At low system utilization the compensation overhead is small. At higher utilization, the overhead increases up to about 30% when MPL is 21 (about 80% CPU utilization by transactions). When MPL was increased to 28 (about 87% CPU utilization), all queries were aborted because the log processing thread was not given enough CPU time to process all log records before they were removed from the log buffer. The other two algorithms REDO compensation and 2PL is not shown in Figure 7.6 since their response time ratios lie outside the range covered by the graph.

Note that the starvation of the log processing thread does not imply that UNDO compensation is only practical for a small number of concurrent transactions. The limiting factor for log processing is CPU time. As long as the CPU has spare capacity for log processing, there will be no limitation on the number of concurrent transactions, and the overhead to achieve transaction-consistency will be low when using UNDO compensation.

Transaction Throughput

How the execution of queries affects the throughput of committed transactions is shown in Figure 7.7. The reduction in throughput caused by concurrent execution
7.3 Experiments and Results

Figure 7.6 Relative performance of UNDO compensation compared to GO processing. (Sel = 10%)

Figure 7.7 Transaction throughput when running queries. (Sel = 10%)
of queries is small except when two-phase locking is used. A closer inspection (Figure 7.8) shows that the transaction throughput for UNDO compensation is slightly higher than for REDO compensation and GO processing. This is mainly a consequence of the non-preemptive CPU scheduling. Since the log processing thread holds the CPU for a shorter time period than the scan thread, the average time transactions have to wait for the CPU is reduced by introducing the log processing thread. Another explanation is that the UNDO algorithm, since it uses longer time to execute a query, will have less frequent disk accesses than GO processing. Hence, when using UNDO compensation, transactions will not as often have to wait for disk accesses made by queries.

Figure 7.8 also shows that the relative reduction in transaction throughput is lowest for small values of MPL. The reason is that the probability that a transaction waits for a query thread, and not another transaction, to release the CPU will be greatest when the number of concurrent transactions are low. Since transactions have higher priority with respect to CPU scheduling, a query thread will only be scheduled when no other transaction has requested the CPU. In other words, queries will more often block transactions for small MPL values.

The extra work imposed on transactions by REDO compensation did not significantly increase transaction response times. The main reason is that in the experiment response times were dominated by disk access time. In addition, during the compensation phase queries only access disks holding temporary tables. Thus, transactions do only have to wait for disk requests made by queries during the scan phase. For applications where transactions seldom need to access disk, the extra work required by REDO processing may become more significant, especially for
Figure 7.9 Distribution of transaction response times, \(MPL = 10, Sel = 10\%\)

queries that require more complex processing of concurrent updates. Also note that
the transaction throughput of REDO processing will be actually somewhat lower
since not all the cost of maintaining the update-list has been modeled.

As long as the response times of queries is dominated by disk access time, the
extra work by transactions required by REDO compensation will not give a signiﬁcant
increase in transaction response times. In addition, during the compensation
phase a query will only access the disk holding temporary tables. Thus, transactions
will only have to wait for disk requests made by the query in the scan phase. For
applications where the transactions very seldom need to access disk, the extra work
required by REDO processing may be signiﬁcant, especially for queries that require
more complex processing of concurrent updates.

When using two-phase locking the throughput also depends on the selectivity of
the query since a longer query will delay transactions for a longer time. In addition,
increased selectivity means a larger number of lock conﬂicts. Anyhow, the execution
of queries using 2PL results in a signiﬁcant reduction in transaction throughput for
all selectivities.

A look at the distribution of response times for transactions (Figure 7.9) when
running queries using UNDO compensation, shows that the delay caused by query
execution does not signiﬁcantly change the distribution of transactions response
times. For \(MPL = 10\) and \(Sel = 10\%\), the response times of transactions in the 90th
percentile increased with only 3\%. In other words, by using UNDO compensation,
transaction-consistent query execution can be achieved without any signiﬁcant effect
on concurrent transaction processing.
Performance Evaluation of Compensation-Based Query Processing

Figure 7.10 Final size of the update-table.

Size of Update-Table

In order to be able to apply UNDO compensation, the main memory requirements for the update-table should not be too large. The storage needed is dependent on both the multiprogramming level (MPL) and the length of the query. Figure 7.10 shows that for a given MPL, the maximum size of the update-table is proportional to the selectivity (length) of the query. As long as MPL is low, only a small fraction of the tuples have entries in the update-table. When MPL is increased, the storage needed for the update-table increases both due to increased update rates and longer query execution times.

This experiment shows that for short queries at low to medium update rates, storing the update-table in main-memory will not impose a problem. For long queries at high update rates, the update-table may become prohibitively large. However, the size of the update-table can be significantly reduced by using the space optimization techniques of Section 4.3.2. This will be further investigated in a later experiment.

Time Used for Backward Log Processing

The average number of transactions that were included in backward log processing (BLP) in this experiment never exceeded 1, and the time used for BLP seldom exceeded 1 ms. The reason for this is the profile of TPC-B transactions. Most transactions will only have to fetch one page from disk, the page for the Account relation. Since the Account relation is accessed by the first operation, most of the active TPC-B transactions will not have performed any operations and need not be included in BLP.
7.3 Experiments and Results

Table 7.6 Operation mix of transactions.

<table>
<thead>
<tr>
<th>Type</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>40%</td>
</tr>
<tr>
<td>Update</td>
<td>30%</td>
</tr>
<tr>
<td>Insert</td>
<td>15%</td>
</tr>
<tr>
<td>Delete</td>
<td>15%</td>
</tr>
</tbody>
</table>

7.3.2 Transaction-Mix Experiment

While TPC-B transactions reflect a simple update-intensive transaction typically of an OLTP application, an OLTP system is often characterized by multiple transaction types of varying complexities. While in the previous experiment, all transactions had the same sequence of operations and the same length, in this experiment, transactions of various lengths and operations are used. The number of operations in a transaction are Poisson distributed, and the mix of operations used is shown in Table 7.6. For each operation the relation is chosen uniformly among the four relations of the database shown in Table 7.3, and within each relation the tuple to be accessed is chosen uniformly. Hence, the relations will have different tuple access rates. At startup, all relations contain 75% of its maximum as specified by Table 7.3. Since the frequency of insert and delete operations are equal, this fill degree will be maintained throughout the simulation.

The application parameters for the experiment is shown in Table 7.7. The utilization of the CPU and the disks holding the Account and the History relations when running the transactions of this experiment is shown in Figure 7.11. Note that the system is disk bound for high MPL values due to contention for the single disk holding the History relation.

When running a transaction, the type of each operation is determined just before it is to be executed based on whether the tuple to operate on exists in the database at that time. At that point, another transaction may hold an exclusive lock on this tuple. When this lock is released, the tuple may have been deleted or inserted by this other transaction. A transaction will abort if it discovers that it is about to read, update or delete a non-existing tuple, or insert an already existing tuple. If a deadlock between to transactions occur, the oldest transactions will be aborted.

Table 7.7 Application model parameters for the transaction-mix experiment.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumDisks</td>
<td>8</td>
</tr>
<tr>
<td>MPL</td>
<td>1, 3, 6, 10, 15, 21, 28</td>
</tr>
<tr>
<td>QMPL</td>
<td>1</td>
</tr>
<tr>
<td>AvgOp</td>
<td>1 + 4 (Poisson)</td>
</tr>
<tr>
<td>Sel</td>
<td>2%, 5%, 10%, 25%, 50%, 100%</td>
</tr>
<tr>
<td>KeysPP</td>
<td>150</td>
</tr>
<tr>
<td>Fill{Account, Branch, Teller, History}</td>
<td>75%</td>
</tr>
</tbody>
</table>
Figure 7.11 System utilization when running the transaction-mix experiment.

by its transaction manager when its execution time exceeds TransMaxTime. The measured total abort rate varies from none at $MPL = 1$ to 0.1% at $MPL = 28$.

Query Response Time

Figure 7.12 shows the query performance of UNDO processing and GO processing, and Figure 7.13 shows the relative performance of the two algorithms. UNDO compensation has an overhead of 7% when maximum CPU utilization (60%) is reached. Comparing the query performance of this experiment with the previous experiment (Figure 7.14), shows that the overhead for UNDO compensation for a given CPU utilization is highest in the first experiment. This is because TPC-B transactions have a higher frequency of modifying operations (update, insert, and delete) than the transactions used in the second experiment. In other words, more log processing and compensation per operation are needed for running queries concurrently with TPC-B transactions.

This experiment shows that the overhead associated with UNDO compensation is dependent on the type of transaction workload. Since also the transactions used in this experiment are pretty update-intensive, even less overhead should be expected for compensation-based query processing in most transaction processing systems.

Transaction Throughput

The reduction of transaction throughput when running queries (Figure 7.15) is not very different from the previous experiments except that the difference between
7.3 Experiments and Results

Figure 7.12 Query response times for various MPL, (Sel = 10%)
Figure 7.14 Comparing the overhead of UNDO compensation in the two experiments. (Sel = 10%)

Figure 7.15 Reduction in transaction throughput when running queries. (Sel = 10%)
UND0 compensation and GO processing is smaller. As above, the distribution of transaction response times is not significantly changed by the concurrent execution of UNDO compensation-based queries. In other words, the effects on transaction throughput by UNDO compensation are insignificant in both experiments.

**Size of Update-Table**

The maximum size of the update-table for this experiment is shown in Figure 7.16. As discussed above, the storage needed for the update-table is dependent on the update rate and the length of the query. Since the fraction of update operations is smaller than in the previous experiment, the number of entries in the update-table will be smaller for a given query length. In addition, since transaction execution is disk bounded, transactions will never exceed 60% CPU utilization. In other words, the maximum possible execution time of a specific query will be smaller than in the TPC-B experiment where much less CPU time may be available to queries. Hence, the maximum number of entries that could accumulate in the update-table is lower in this experiment.

**Time Used for Backward Log Processing**

The execution time of backward log processing is shown in Figure 7.17. As expected, the time is dependent on the multiprogramming level (MPL) since this directly determines the size of the AFTER set. The reason for the more than linear growth in execution time with increasing MPL, is that an increasing update rate will increase
the possibility of having to access log pages that are no longer in main memory. By using a larger buffer for log pages, less pages will have to be fetched from disk during backward log processing. Accessing log records on disk, may potentially slow down transactions. However, no significant effects on transaction throughput was observed in this experiment.\footnote{Note that the required precision of the confidence interval was not established for the execution time. The reason is that the execution time varied a lot depending on whether it was necessary to fetch log pages from disk or not.}

While this experiment shows a more than linear growth in execution time with increasing MPL, the execution times are always insignificant compared to the total execution times of the queries.

### 7.3.3 Space Optimization of the Update-Table

Another experiment using TPC-B transactions was run in order to evaluate the space optimization technique for the update-table (Ref. Section 4.3.2).

Assuming uniform tuple access patterns and constant scan rates, avoiding insertions of entries for already processed tuples and deleting entries after processing tuples, should theoretically reduce the space requirements to one fourth of the non-optimized case. It can easily be shown that the update-table will reach its maximum size halfway through the scan. At that time, half of the entries have not yet been created, and the update-table will only contain entries for that half of the tuples that have not yet been read. In other words, the size of the update-table at halfway through the scan will be one fourth of the size at the end of the scan when not apply-
7.4 Discussion

Figure 7.18 Non-optimized and optimized (OPT) space requirements for the update-table. (Logarithmic scale on the vertical axis.)

ing any space optimization. In addition, entries need only be made for tuples within the selectivity range of the query. In other words, optimization should reduce the space requirements to $Sel/4$ of the non-optimized case, where $Sel$ is the selectivity factor of the query.

As shown by Figure 7.18, the simulation experiment supports this theoretical analysis. Note, however, that in order to apply the optimizations, the log processing thread need to be able to determine whether a tuple lies behind or ahead of the scan. Thus, for partial tuple logging, the non-optimized and optimized results in Figure 7.18 can be viewed as the upper and lower limits, respectively, for the space requirements of the update-table. Note also that a skewed access pattern will in most cases lead to reduced storage requirements.

The optimizations also lead to increased query performance due to reduced work for the log processing thread. For long-running queries at high MPL, the query response times were reduced by 10% (Figure 7.19). Transaction throughput was not significantly different from the non-optimized case.

7.4 Discussion

The simulation experiments show that it is possible to efficiently perform compensation-based query processing without significantly affecting the performance of transactions. As expected, UNDO compensation outperforms REDO compensation with respect to query response times and 2PL with respect to transaction throughput.
The extra work required to achieve transaction-consistency only slightly increases the response times for queries in a system with low to medium CPU utilization by transactions. The maximum overhead compared to GO processing was 30% when running a query concurrently with TPC-B transactions. For less update-intensive transactions (i.e., more read operations), the overhead to achieve transaction-consistency will be smaller. At very high CPU utilization by transactions, the log processing thread was not able to keep up with the production of new log records.

The starvation of the log processing thread can be avoided if it processes more than a single log record each time it is scheduled. The expense is longer transaction response times.

The increase in transaction response times by introducing queries is negligible when threads executing queries are giving lower priority than other threads. The experiments also showed that the extra work required by transactions when doing REDO compensation is not significant as long as transactions need to read pages from disk. For transactions that do not need to access disk, this extra work may possibly affect transaction response times, especially for queries that require more complex processing of concurrent updates.

The experiments show that the critical part for the execution of UNDO compensation is the size of the update-table. For short queries or low to medium update rates, the size will not represent a problem. For long queries and very high update rates, the update-table may become too large to be kept entirely in main memory even when using the space optimization techniques. For such queries, the methods presented in Section 4.4 for storing the update-table on disk should be considered.
Further studies are needed in order to determine how much a disk-based update-table will affect query performance.

The time needed to perform backward log processing (BLP) was negligible compared to the entire execution time of the query in all experiments. The BLP time is generally dependent on the length and number of concurrent transactions. For most queries the BLP time will be a small fraction of the execution time even for much longer transactions than those used in this performance study.

All the simulation experiments have only run one query at a time ($Q_{MPL} = 1$). If several queries are executed concurrently, query response times will of course be higher. However, as long as transactions are given higher priority than query threads, transaction throughput should not be significantly affected.

Many other performance aspects of compensation-based query processing should be studied. Simulation experiments should be performed to evaluate the algorithms in the context of a main-memory DBMS. The performance aspects of partial tuple logging should also be studied: How much is transaction throughput increased due to less log volume compared to full tuple logging, and how much extra work is required by queries? The use of secondary index scans should also be evaluated in order to determine how the update frequency of transactions affect the size of the priority queue. Finally, the algorithms for selection, aggregation, and join presented in Chapter 5 should be evaluated.
Performance Evaluation of Compensation-Based Query Processing
Part III

Distributed Query Processing

This part presents algorithms for undo/no-redo compensation in a distributed DBMS.

Chapter 8 The model of a distributed DBMS as used in this thesis is presented.

Chapter 9 Two algorithms for establishing transaction-consistent execution of compensation-based queries are presented. The algorithms differ in whether a central coordinator is used to establish transaction-consistency. The performance of the two algorithms is discussed and compared.
Chapter 8

Distributed Model

This chapter describes a model of a distributed database system. This model will be used in the next chapter for the presentation of distributed algorithms for compensation-based query processing.

The distributed hardware model is based on a shared-nothing architecture [Ston86]. Each node has exclusive access to its own disk units and memory modules, and the nodes are connected through a communication network.

On each node the database system consists of two processes: a transaction controller and a database kernel (Figure 8.1). The transaction controller receives requests from clients to execute transactions. It coordinates the transaction execution and forwards the operations to the kernels that contain the relevant data (function shipping). One main advantage of such a model is that it gives a clean division between the administration of transactions and the administration of data. In ad-

![Figure 8.1 Distributed database system with shared-nothing architecture. Each of the three nodes has a transaction controller (TCON) and a database kernel (KERN).]
dition, having both a transaction controller and a kernel at each node gives better possibilities for load balancing. Requests from clients could be routed to transactions controllers at nodes with a low kernel activity.

For each transaction a separate dialog is established between the transaction controller and each of the kernels that participate in executing the transaction. A dialog uses a connection-oriented protocol ([Sta88]) which is initiated by the transaction controller. Reliable communication is achieved by resenting a message if its acknowledgment is not received within a specified time interval. The protocol ensures that the transaction controller may not send a new message to a kernel before it has received acknowledgment on the previous message of the dialog. Thus, it is guaranteed that the message sequence will be kept within each dialog.

8.1 Transaction Controller

Clients that want to run transactions, connect to one of the transaction controllers of the system. The clients will supply the user code or specify the precompiled procedures to be executed. The transaction controller will compile the user code into an internal code format. For each operation to be executed, the transaction controller will, based on the data dictionary, determine which kernels contain the data needed by the operation. The transaction controller will then send a request to the kernels to perform the operation. The kernels that are involved in executing a transaction, are called the participants of the transaction.

When executing a distributed transaction, it must be ensured that all participants in the transaction reach consistent states. That is, either the transaction controller and all participants commit the transaction or they all abort it (atomic commitment). It is here assumed that when all operations of a transaction have been executed the transaction controller starts a two-phase commit (2PC) protocol [Ber87] to ensure atomic commitment. The two phases of the 2PC is the voting phase and the decision phase. In the voting phase the controller sends a vote-req message to all the participants. When a participant receives a vote-req message, it responds by sending a yes message if it can guarantee that it can commit the transaction if the controller decides to do so. Otherwise, it sends a no message and aborts the transaction. In the decision phase the transaction controller collects the votes from all participants. If all participants voted yes, it decides to commit the transaction and sends a commit message to all participants. Otherwise, it decides to abort the transaction and sends abort messages to all participants. Each participant then either commits or aborts the transaction based on the message it receives. Figure 8.2 illustrates the 2PC protocol.

Several optimizations to the basic 2PC protocol, described above, is possible [Sam95]. Unless otherwise stated, it is assumed that basic 2PC is used. However, the algorithms presented below will also work for most optimizations of 2PC.

8.2 Database Kernel

The database kernel has the main database storage manager capabilities, like concurrency control, logging, buffer management, and access methods. It is assumed

---

1 Small capital letters are used to indicate messages.
that strict two-phase locking [Ber+87] is used to provide strict serializable execution of transactions.

A database relation can be distributed to several kernels by horizontal fragmentation. For simplicity, it is assumed that the fragments are not replicated.

At each kernel a transaction will go through a set of states (Figure 8.2). The state of a transaction is recorded in the transaction table of the kernel. A transaction is said to be *active* at a kernel from the time the kernel receives the first message for this transaction until it receives the *vote-req* message. If the kernel votes Yes, the transaction becomes *uncertain*. The transaction will be uncertain until the kernel receives either a *commit* or *abort* message. Then the transaction will become *committed* or *aborting*, respectively. Aborting transactions will enter the *aborted* state when all the local effects of the transaction have been undone. Similarly, if the kernel votes No, the transaction will be in the aborting state until it is aborted. The kernels will log the state changes in a *distributed transaction log* [Ber+87] in order to be able to recover from failures.
8.3 Distributed Query Execution

Clients send their requests for running a query to one of the transaction controllers. In the following, a transaction controller executing a query is called a query controller. The query controller will based on the data dictionary determine which kernels should take part in executing the query. It will then send a QUERY-START message to the kernels executing the query. This message includes the internal code for executing the query. The time a kernel receives a QUERY-START message will be defined as the starting point of the query at this kernel. The kernel will then execute the supplied code, and return the result to the query controller which may perform more operations before sending the query result to the client.

In the following, it is described how compensation-based query processing can be performed in a distributed system.
Chapter 9

Distributed Algorithms

Two distributed algorithms for doing compensation-based query processing are described in this chapter. The algorithms synchronize the kernels of the system in order to get a globally transaction-consistent query result. The operations of each kernel is basically the same as in the centralized version. Before describing the algorithms, the requirements for achieving global transaction-consistency in a distributed database system is discussed.

9.1 Global Transaction-Consistency in a Distributed Database System

When executing a query, the set of all transactions in which a kernel is participating, is divided into two disjoint subsets, the BEFORE set and the AFTER set. The query will see all the effects of transactions in the BEFORE set and none of the effects of transactions in the AFTER set. Before starting the scan thread, the kernel must do backward log processing on log records produced before the starting point of the query, inserting entries in the update-table for log records created by members of the AFTER set. In addition, forward log processing is performed on all new log records until the scan thread has finished. As in the centralized version, only the subset of the AFTER set needed during backward log processing will have to be explicitly represented.

In order to get a transaction-consistent result, the database states seen by the query at each kernel must together define a globally transaction-consistent state. In other words, no transaction should be in a query's BEFORE set at one kernel and in the AFTER set of the same query at another kernel. In addition, no transaction dependent on any of the transactions in the local AFTER sets should be in the BEFORE set at any kernel. Hence, if a transaction is included in the local AFTER set of one kernel, this transaction and all transactions depending on it, must be included in the local AFTER sets of all other kernels participating in the transactions.

Let the global AFTER set of a query be the union of all the query's local AFTER sets. Since forward log processing is performed on all log produced after the starting point, none of the effects of transactions that perform update operations after the starting point at one kernel, must be visible to the query. Thus, all transactions that are active after the starting point at at least one kernel will be included in the global
AFTER set. In addition, all transactions that are directly or transitively dependent on these transactions must be included in the global AFTER set.

In addition to transactions that are active at the starting point, transactions that are or later will be aborting may perform operations after the starting point. Hence, these transactions must be included in the local AFTER set of the kernel. However, aborting transactions do not necessarily have to be included in the local AFTER set of all participating kernels. This is because if a transaction has terminated before the starting point at a kernel, effects of this transaction are no longer visible at this kernel. This means that this transaction can be safely ignored during backward log processing at this kernel. In other words, the global AFTER set must contain all transaction that are active after the starting point at any kernel. In addition, transactions that are aborting at the starting point of a kernel should be added to the AFTER set at this kernel before starting backward log processing.

Figure 9.1 shows an example with local dependency graphs for four kernels. Black nodes are used to represent transactions that are active after the starting point. Nodes for transactions that are black in at least one local dependency graph, will be black in the global dependency graph. Grey nodes represent transactions that are dependent on transactions represented by black nodes. In other words, the global AFTER set consists of all transactions having either black or grey nodes.

In the example of Figure 9.1, transactions $T_3$, $T_6$, and $T_7$ are active after the starting point at least one kernel. Since they may have logged operations that will be processed by the forward log processing, these transactions must be included in the AFTER set. In addition, $T_4$ must be included in the AFTER set since it is dependent on $T_3$ (at kernel 1). Since $T_5$ is dependent on $T_4$ (at kernel 2), $T_5$ must also be added to the AFTER set.

Note that transaction dependencies at kernels that do not participate in the
query, must also be considered. If the query in Figure 9.1 will not execute at kernel 2, the fact that \( T_5 \) depends on \( T_4 \) must still be considered in order to achieve a transaction-consistent execution of the query. Otherwise, \( T_5 \) would not be included in the global AFTER set. For simplicity, it will be assumed where not otherwise stated that all kernels of the system participate in the query. It will later be discussed how to efficiently handle kernels that do not participate in the query.

The computation of the part of the AFTER set needed for backward log processing could either be distributed to each kernel, or it could be centralized to the query controller. Below two methods for establishing transaction-consistent AFTER sets in a distributed system are presented. In the first method, the centralized method, a global AFTER set is computed by the query controller of the query while the second method distributes the computation of the AFTER sets to each kernel. The centralized method lets a single coordinator collect information on transaction dependencies from all kernels. Using this information, the global AFTER set could be computed by building the equivalent of a global dependency graph. This AFTER set will be distributed to the kernels participating in the query and used for backward log processing.

The distributed method uses piggybacking of information on the messages of the 2PC protocol both to synchronize the distributed computation of AFTER sets and to inform other kernels about local AFTER set insertions. In other words, the transaction controllers will distribute the necessary dependency information to all the participants of a transaction.

Before describing the two algorithms, it will be discussed how the necessary transaction dependencies could be efficiently represented and maintained.

### 9.1.1 Representing Transaction Dependencies

As discussed above, knowledge of transaction dependencies are essential in order to be able to establish a global AFTER set that guarantees transaction-consistent execution of a query. In other words, each kernel must maintain a representation of local transaction dependencies. This could be done by maintaining the actual dependency graph as done by concurrency control protocols based on serialization graph testing (SGT) [Ber+87]. However, as long as SGT is not used for concurrency control, this would require extra space and resources. In addition to storing the actual graph, each kernel must keep track of which transactions have made access to which tuples. For each new operation performed, the kernel would have to update the dependency graph by checking for possible conflicts with operations performed earlier by other transaction. Hence, extra work would have to be done by transactions.

Another approach to determine transaction dependencies, could be to reconstruct the dependency graph by analyzing the log. However, in order to be able to find all dependencies, this requires that also read operations are logged.

While the two approaches discussed above requires extra resources, the approach used below to establish transaction dependencies will be based on information that is already maintained by the kernels. The penalty to be paid is that a superset of the actual dependencies will be established. Thus, the AFTER set may be larger than needed, resulting in more work to be done during backward log processing.

The basis for finding possible dependencies is the use of strict 2PL for concurrency control. Using strict 2PL means that a transaction holds all its locks until it
commits.\(^1\) Thus, a transaction \(T\) that wants to perform an operation that conflicts with an operation already performed by another transaction \(T'\), will have to wait until \(T'\) commits and releases its locks. In other words, a transaction \(T\) could only be dependent on another transaction \(T'\) if there exists a kernel where \(T\) is active after \(T'\) has committed. Such an extended definition of dependency will be used here:

**Definition 1.** A transaction \(T\) is dependent on another transaction \(T'\) \((T' \rightarrow T)\) iff there exists a kernel where \(T\) is active after \(T'\) has committed.

The new definition of dependency can be viewed as defining dependency at a coarser granularity. While the original definition defines dependency at tuple granularity (i.e., the order of (conflicting) access to a tuple), the extended definition defines dependency at node granularity (i.e., the order of access to a node). Other dependency granularities are also possible (e.g., file, relation, or block). Using a coarser dependency granularity may increase the size of the AFTER set. However, the cost of maintaining the dependencies decreases.

To be able to use the extended definition of dependency, it must be ensured that the extension will not introduce cycles in the dependency graph. To show this the notion of transitive dependency is introduced:

**Definition 2.** A transaction \(T_k\) is transitively dependent on another transaction \(T_1\) \((T_1 \rightarrow^* T_k)\) iff the dependencies \(T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_k\) exists in the global dependency graph.

That is, in order to have cycles in the dependency graph, both the transitive dependencies \(T_1 \rightarrow^* T_k\) and \(T_k \rightarrow^* T_1\) must exist. To show that such a cycle is not possible, the following fact about the 2PC protocol will be used:

**Lemma 1.** Let \(t_x(U_i)\) and \(t_x(C_i)\) be the time, with respect to a global clock, that transaction \(T_x\) turns uncertain and committed, respectively, at kernel \(x\). Then the 2PC protocol ensures that for all participating kernels \(x\) and \(y\), \(t_x(U_i) < t_y(C_i)\).

**Proof.** The transaction controller requires that all participating kernels have replied to the VOTE-REQ message (i.e., become uncertain) before COMMIT messages are sent. Thus, no kernel could commit a transaction before all the other participating kernels have turned uncertain.

Using Lemma 1 it can be shown that the following property holds for transitive dependent transactions:

**Lemma 2.** Let there be two transactions \(T_1\) and \(T_k\) such that \(T_1 \rightarrow^* T_k\). Then there exist two kernels \(x\) and \(y\) such that \(t_x(C_i) < t_y(U_k)\).

**Proof.** Assume without loss of generality that the transitive dependency \(T_1 \rightarrow^* T_k\) is caused by the existence of the following dependencies: \(T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_k\). That

\(^1\)This is the most common definition of strict 2PL [Ber+87]. Breitbart et al. call this variant for rigorous 2PL [Bre+91] since in order to assure strictness, only write locks need to be held until the transaction terminates. The requirement for the algorithms presented here is that only rigorous transaction schedules are permitted. That is, no access conflicts between uncommitted transactions are allowed.
is, for each dependency \( T_i \rightarrow T_{i+1} \), there exists a kernel \( x \) where \( T_{i+1} \) is active after \( T_i \) commits (i.e., \( t_x(C_i) < t_x(U_{i+1}) \)). From Lemma 1 it follows that for any kernel \( z \), \( t_x(U_i) < t_x(C_i) < t_x(U_{i+1}) \). Similarly, for any kernel \( z' \) there exists a kernel \( x' \) such that \( t_x(C_{i+1}) < t_x(U_{i+1}) < t_x(U_{i+1}) \). Thus, there exists two kernels \( x \) and \( x' \) such that \( t_x(C_i) < t_x(U_{i+1}) < t_x(C_{i+1}) < t_x(U_{i+2}) \). This gives that there exists kernels \( x, x', \) and \( x'' \) such that \( t_x(C_i) < t_x(U_2) < t_x(C_2) < t_x(U_3) < \cdots < t_x(C_{k-1}) < t_x(U_k) \).

Lemma 2 gives that if there is a cycle in the dependency graph then there must exist kernels \( x, x', y, \) and \( y' \) such that both \( t_x(C_i) < t_y(U_k) \) and \( t_{x'}(C_k) < t_{y'}(U_1) \) hold. Applying Lemma 1 to the first requirement gives that for all kernels \( z \) and \( z' \), \( t_x(U_i) < t_x(C_i) \). Since this contradicts the second requirement there could not be any cycle in the dependency graph.

If applying dependency at node granularity, the information needed to find transaction dependencies, are which transactions are active after the commitment of a transaction. In other words it is necessary to keep track of when transactions are active, uncertain and committed at each kernel. Recall from Section 8 that this information is logged in the distributed transaction log (DT log). In other words, each kernel can establish the local transaction dependencies by processing the DT log.

### 9.1.2 The Viewpoint of a Query

At each kernel, the point in time which comes first of the starting point and the first commitment of a transaction in the global AFTER set occurs, will be called the viewpoint of the query. Using node granularity dependencies, all transactions that are active after the earliest commitment of a transaction in the global AFTER set, will be regarded as dependent on a this transaction, and these transactions must also be included in the AFTER set. Thus, the query should see none of the effects of transactions that are active after the viewpoint. On the other hand, all of the effects of transactions committed before the query's viewpoint should be visible to the query. Effects of a transaction that is uncertain at the viewpoint, will be visible to the query only if the transaction is not included in the global AFTER set. That is, the transaction is active after the viewpoint at other kernels.

Figure 9.2 shows an example of how the transaction-consistent viewpoints of a query is determined. For each of the four kernels of the system, the local sequence in which transactions becomes uncertain, \( U_i \), and committed, \( C_i \), is shown. The solid lines marked SP, represent the starting points of one particular query. The dashed lines marked VP, represent transaction-consistent viewpoints for the same query. Per definition, the viewpoints at each kernel are given by the global AFTER set if transactions committed before the local starting point are members of the global AFTER set.

In Figure 9.2, \( T_1 \), \( T_3 \), and \( T_6 \) belong to the query's global AFTER set because they are active after the starting point at least one kernel. \( T_5 \) is included in the AFTER set because it at Kernel 1 may be dependent on \( T_6 \) which is in the AFTER set. Because of that, \( T_7 \) must also be included in the AFTER set since it may be dependent on \( T_5 \) at Kernel 3. \( T_3 \) and \( T_4 \) will be in the query's BEFORE set since they are neither active after a starting point nor after the commitment of any of the transactions in the AFTER set. In other words, \( T_3 \) and \( T_4 \) could not possibly
be dependent on any of the transactions in the AFTER set. As defined above, the viewpoints of the query at each kernel will be before the earliest commitment of a transaction in the AFTER set except at kernel 0 where no transactions in the AFTER set committed before the starting point.

The following properties of viewpoints will be used later:

**Property 1.** Let $t_y(SP)$ be the starting point of a query at any kernel $y$. That is, a kernel $y$ receives the QUERY-START message for this query at time $t_y(SP)$ with respect to a global clock. Consider a kernel $x$ such that $t_x(SP) < t_y(SP)$ for all kernels $y \neq x$. Then the query's viewpoint at kernel $x$ will be equal to its starting point.

*Proof.* Consider a transaction $T_2$ that commits at $x$ before the local starting point (i.e., $t_x(C_2) < t_x(SP)$), and a transaction $T_1$ on which $T_2$ transitively depends. Combining $T_1 \xrightarrow{x} T_2$ with Lemma 1 and Lemma 2 and the fact that kernel $x$ has the earliest starting point gives that there exist kernels $z'$ and $z''$ such that

$$t_z(U_1) < t_{z'}(C_1) < t_{z''}(U_2) < t_z(C_2) < t_z(SP) < t_y(SP),$$

for all kernels $z$ and $y \neq x$. Since no transaction that $T_2$ may transitively depend on could have been active after the starting point at any kernel, $T_2$ will not be included in the AFTER set. In other words, no transaction that are committed before the starting point at kernel $x$ will be included in the AFTER set. Thus, at kernel $x$ the viewpoint will not be earlier than the starting point.

**Property 2.** Let $t_y(VP)$ be the time of the earliest commit at kernel $y$ of a transaction in the global AFTER set (i.e., the viewpoint at kernel $y$). Consider the kernel $x$ such that $t_x(SP) < t_y(SP)$ for all kernels $y \neq x$ (i.e., the kernel with the earliest starting point). Then, $t_x(SP) < t_y(VP)$ for all $y \neq x$.

![Figure 9.2 Example of viewpoints (VP) and starting points (SP) for a query in a system with four kernels.](image-url)
9.2 Centralized Computation of the Global AFTER set

Proof. Consider a transaction \( T_1 \) that at some kernel \( z \) is committed before any other commitment of a transaction in the AFTER set. In other words, the commitment of \( T_1 \) defines the viewpoint at kernel \( z \). If kernel \( z \) should have the earliest viewpoint, then \( t_x(C_1) < t_x(SP) \) where kernel \( x \) is the kernel with the earliest starting point. From Lemma 1 it follows that \( t_y(U_1) < t_x(SP) \) at all kernels \( y \). From Lemma 2 it follows that if \( T_1 \) is transitively dependent on a transaction \( T_2 \), then there exists a kernel \( z' \) such that \( t_{z'}(C_2) < t_y(U_1) < t_x(SP) \) for all kernels \( y \). Again, it follows from Lemma 1 that \( t_{y'}(U_2) < t_{z'}(C_2) < t_x(SP) \) for all kernels \( y' \). Thus, transaction \( T_1 \) could not be member of the AFTER set since it could neither be active after any starting point nor transitively dependent on transactions that are active after a starting point. Hence, all transactions in the AFTER set commits at all participating kernels after the first kernel has received the QUERY-START message.

Summing up, there will always be at least one kernel where the viewpoint is equal to the starting point (Property 1), and the kernel with the earliest starting point will always have the earliest viewpoint (Property 2).

9.2 Centralized Computation of the Global AFTER set

In a centralized computation of the AFTER set, each kernel informs the query controller about the transactions that are active after its starting point and about local transaction dependencies. Based on this information, the query controller will be able to compute a global AFTER set. This AFTER set is then distributed to all kernels that participate in the query and used for backward log processing.

The distributed transaction logs (DT log) maintained by each kernel will contain all the information needed by the query controller to compute the global AFTER set. The only requirement is that the kernel has the reception of the QUERY-START message in the DT log. In that case, the query controller can by looking at the DT log, find which transactions have been active after the starting point. In addition, node granularity dependencies can be found by looking at which transactions are active after the commitment of a transaction.

Below, it is first described how the query controller computes the global AFTER set. It is assumed that each kernel has sent a sufficient part of its DT log to the query controller. It will later be shown how a kernel can determine what part of the DT log is needed by the query controller.

9.2.1 Computation of the Global AFTER Set

When the query controller has received the DT log from all participants, it can start computing a globally consistent AFTER set for the query.

All transactions that are active after the starting point of the query at a kernel, will be included in the AFTER set. Hence, all transactions that are either active at a starting point or starts after a starting point, are included in the AFTER set. In addition, any transactions that may be dependent on transactions in the AFTER set must be included in the AFTER set. Using node granularity dependency, all transactions that are active at any kernel after the commit of a transaction in the AFTER set will also be included in the AFTER set.
The computation of the global AFTER set will be an iterative process. First, all transactions that are active after the starting point at any kernel are included in the AFTER set. Then all transactions transitively dependent on any of the members of the AFTER set at any kernel are added. This last step will be repeated until no new transactions are added to the AFTER set. This is because the newly added transactions may have dependent transactions that are not already members of the AFTER set. In other words, in iteration $i$ of the computation, transactions dependent on transactions included in iteration $i - 1$ is added to the AFTER set. The computation terminates when no more transactions are included in the AFTER set.

The number of iterations is limited by the number of kernels in the system. Property 1 gives that there will be at least one kernel where the starting point is equal to the viewpoint. At this kernel, no transaction that is committed before the starting point will be added to the AFTER set. Thus, all transactions in which this kernel participate and which will be members of the global AFTER set, are included in the first iteration. In other words, at most $n - 1$ kernels will contain local dependencies that may result in the inclusion of new transactions in the second iteration. For each iteration, let each kernel have a temporary viewpoint as defined by the current AFTER set. That is, the temporary viewpoint is given by the time of the earliest commitment of a transaction in the current AFTER set. Consider the kernel that have the earliest temporary viewpoint of the $n - 1$ kernels that take part in the second iteration. By substituting temporary viewpoint for starting point\footnote{This substitution can be done since the temporary viewpoints have the same properties as starting points. All transactions active after a temporary viewpoint will be added in the next iteration of the computation, and all other transactions dependent on such transactions will be added in later iterations.}, it follows from Property 1 that the temporary viewpoint at this kernel will be equal to its final viewpoint. In addition, Property 2 gives that none of the viewpoints at the remaining $n - 2$ kernels will precede this viewpoint in time. Thus, no more transactions active at this kernel will be added to the AFTER set in the third or later iterations. In other words, the number of kernels that may have dependencies that leads to new inclusions in the AFTER set is reduced by at least one for each iteration. Hence, the number of iterations will be limited by the number of kernels.

### 9.2.2 Determining the Part of the DT Log Needed by the Query Controller

In order to be able to determine which parts of the DT logs should be sent to the query controller, it must be determined which transactions the query controller need to know about. The DT log is used for two purposes: to find which transactions are active after the starting points, and to find transaction dependencies.

1. Since the AFTER set is only needed for backward log processing, information on transactions that are not active before the starting point at any kernel are not needed for computing the AFTER set. Hence, a kernel could send its DT log to the query controller when it is no longer possible that a transaction that have been active before the starting point at another kernel may become active (is started) at this kernel. Note that if such a transaction is active at the starting point of one kernel, the query controller will include it in the AFTER
set regardless of the contents of the DT logs of other kernels. Thus, a kernel only has to wait until no transaction that became uncertain before the starting point at another kernel could become active at this kernel.

It follows from Lemma 1 that transactions that have committed before the starting point at one kernel, could only be active after the starting point at another kernel if this other kernel received the QUERY-START message earlier. In other words, no information will be found about such transactions in the DT logs produced after all kernels have received the QUERY-START message. For transactions that are uncertain at a starting point, how far one would have to wait will depend on the implementation of the 2PC protocol. If the basic 2PC protocol as described in Chapter 8 is used, a transaction that is uncertain at one kernel could not later be started at another kernel. Thus, for all transactions that are active before the starting point at one kernel, the query controller will need no information about what happens after all kernels have received the QUERY-START message.

2. The query controller only needs information on transaction dependencies involving transactions that should be included in the global AFTER set. Since only transactions that are active after a viewpoint will be dependent on transactions in the AFTER set, no information is needed on transactions that are not active after a viewpoint. Thus, information on transactions that are not active at any kernel after the earliest starting point is not needed when computing the AFTER set. (Property 2 gives that no viewpoint represent an earlier point in time than the earliest starting point.) This means that it is not necessary to send DT log produced before the first kernel received the QUERY-START message.

Summing up, all kernels must provide the query controller with the part of the DT log produced between when the first kernel received the QUERY-START message and when all kernels have received the QUERY-START message. The first requirement is satisfied if the query controller makes sure that before it sends the QUERY-START messages, all kernels have marked the start of the part of the DT log needed for processing this query. The second requirement is satisfied if the query controller informs the kernels when it has received acknowledgment on the QUERY-START message from all kernels.

Note that there may be transactions that have started before the query and are still active when the kernels send their DT logs. These transactions may not be mentioned in the DT logs received by the query controller. Thus, a kernel must when logging the reception of the QUERY-START message in the DT log, also include the list of currently active transactions.

It is in this presentation assumed that the startup of transactions are recorded in the DT log, and that the query controller uses this information to determine which transaction is active after the starting point. If the startup of transactions is not recorded in the DT log, each kernel must also send the current list of active transactions when sending its DT log to the query controller.

If using a 2PC protocol that allows the transaction controller to request new operations from one kernel after a VOTE-REQ message has been sent, a transaction that is uncertain at the starting point of one kernel may later become active at the same or some other kernel. Such a transaction must be included in the AFTER set,
and the kernels should not send their DT log until it is guaranteed that no transaction that become active in the future have been uncertain before the starting point at any kernel. In general, this could only be guaranteed when all such transactions have terminated. In other words, the DT log can be sent when both all kernels have received the QUERY-START message, and all transactions that were uncertain at any starting point have terminated. If the kernels do not acknowledge the QUERY-START message until all transactions that locally are uncertain at the starting point has terminated, both these requirements are satisfied when the query controller has received acknowledgments from all kernels.

9.2.3 The Algorithm

Below the entire algorithm for establishing a globally transaction consistent AFTER set is presented. The message sequence diagram in Figure 9.3 gives an overview of the message protocol used by the algorithm. The only assumption about the version of the 2PC protocol used is that Lemma 1 holds.
9.2 Centralized Computation of the Global AFTER set

The transaction controller starts a query by sending QUERY-PREPARE messages to all kernels. When receiving the QUERY-PREPARE message, the kernel marks the current end of the DT log as the start of the part of the DT log it will later send to the query controller for this query. The kernel then sends an ACK message back to the controller. When the controller has received ACK messages from all the kernels, all kernels have set the necessary mark in their DT log, and the query controller can now send QUERY-START messages to all kernels. When a kernel receives the QUERY-START message, it records in the DT log that the query has started together with the transaction IDs of all currently active transactions. If the kernel is going to participate in the execution of the query, it will start forward log processing. It also records a WAIT set containing all currently uncertain transactions. When all transactions in the WAIT set have terminated, the kernel sends a QUERY-STARTED message to the controller.

When the controller has received the QUERY-STARTED messages from all kernels, the necessary DT log has been produced by all kernels, and the query controller sends HISTORY-REQUEST messages to all kernels. When a kernel receives this message, it sends the DT log produced after receiving the QUERY-PREPARE message to the controller. As described above, the controller will then compute a globally consistent AFTER set by inspecting the DT logs received.

The computed AFTER set will be distributed to all kernels that are participating in the query. Upon receiving this AFTER set, the kernels will start backward log processing using the AFTER set. When backward log processing is finished, the kernel can start scanning the base relation(s) of the query.

As discussed in Section 9.2.2, the kernels do not have to wait for uncertain transactions to terminate if basic 2PL is used. In that case, the QUERY-STARTED message could be sent immediately after receiving the QUERY-START message. Anyhow, before backward log processing is started, all transactions in the WAIT set must have terminated. This is because if a transaction in the WAIT set aborts, it must be included in the local AFTER set.

9.2.4 Example

Figure 9.4 shows an example of the parts of the DT log sent from four different kernels to the query controller. The DT log shows the state changes for transactions at each kernel between the receptions of the QUERY-PREPARE and the HISTORY-REQUEST messages of a query Q. The reception of the QUERY-START message and the sending of the QUERY-STARTED message are shown in bold face. The DT log record for the reception of the QUERY-START message gives the starting point of the query and includes the list of active transaction at that point. When kernel 0 receives the QUERY-START message it records the currently active transactions, in this case only T1. Note that T1 becomes active at kernel 0 before the reception of the QUERY-PREPARE message, and it is also active when the HISTORY-REQUEST message is received. If it is not logged as active at the starting point, the query controller would not discover that T1 should be included in the AFTER set. The currently uncertain transactions, T2 and T9, are inserted in the WAIT set. When they commit, they are removed from the WAIT set. When T2 commits, the WAIT set is empty and

---

2 The sending of the QUERY-STARTED message does not have to be logged in the DT log. It is just shown here for illustration purposes.
Figure 9.4 Example of four DT logs received by the query controller for computation of the AFTER set.
the \texttt{QUERY-STARTED} message is sent. At kernel 2 there is no uncertain transaction at the starting point. Thus, the \texttt{QUERY-STARTED} message can be sent immediately.

When the query controller receives the DT logs, it can find all transaction active after a starting point by adding all transactions started after the starting point to the list of active transaction at the starting point. In the example of Figure 9.4, the following transactions are active after the starting point:

- Kernel 0 : \{T_1, T_8, T_{11}, T_{14}, T_{15}\}
- Kernel 1 : \{T_1, T_2, T_6, T_{10}, T_{11}, T_{12}, T_{15}\}
- Kernel 2 : \{T_2, T_6, T_8, T_{11}, T_{12}, T_{14}, T_{16}\}
- Kernel 3 : \{T_{14}, T_{15}\}

To compute the global \texttt{AFTER} set, the query controller first finds the union of the sets above. In this case, the union of the four sets is:

\[ \{T_1, T_2, T_6, T_8, T_{10}, T_{11}, T_{12}, T_{14}, T_{15}, T_{16}\}. \]

Then the query controller checks if any of the transactions in this set commit before the starting point at any kernel. The only occurrence is at kernel 0 where \(T_6\) commits before the starting point. Here, \(T_6\) is active after the commitment of \(T_6\) and should be inserted in the \texttt{AFTER} set. Then the query controller checks whether the newly inserted transaction \(T_6\) commits before the starting point at any kernel. This occurs at kernel 3 where \(T_{13}\) is active after the commitment of \(T_9\). Thus, \(T_{13}\) will also be inserted in the \texttt{AFTER} set. \(T_{13}\) does not commit before the starting point at any kernel so the computation of the global \texttt{AFTER} set is finished. The \texttt{AFTER} set

\[ \{T_1, T_2, T_6, T_8, T_{10}, T_{11}, T_{12}, T_{13}, T_{14}, T_{15}, T_{16}\} \]

is sent back to the kernels by the \texttt{AFTER-SET} message.

In this example, the viewpoint of the query are equal to the starting point at kernel 1 and kernel 2. At kernel 0, the viewpoint is before the commitment of \(T_6\), while at kernel 3, the viewpoint is before the commitment of \(T_9\).

### 9.2.5 Proof of Correctness

In order to prove that the algorithm is correct it must be shown that all queries will have a transaction-consistent view of the database. That is, there should be no cycles in the dependency graph consisting of the specific query and all update transactions.\(^4\) As illustrated by Figure 9.5, this is satisfied if:

1. No transaction is in the query's \texttt{BEFORE} set at one kernel and in the \texttt{AFTER} set at another kernel.
2. No transaction in the \texttt{BEFORE} set is dependent on a transaction in the \texttt{AFTER} set.

It is first assumed that the query controller has access to the entire DT logs of all kernels. It will later be proven that the parts of the DT logs sent to the query controller contains sufficient information.

\(^4\)Note that if several queries are included in the dependency graph, cycles may occur.
Figure 9.5 If the BEFORE set and the AFTER set are disjoint sets, cycles could only occur in the dependency graph if a transaction in the BEFORE set is dependent on a transaction in the AFTER set.

Proof of Requirement 1. Due to forward log processing, all transactions that are active after the starting point of a kernel will be member of the AFTER set at this kernel. The AFTER set computed by the query controller contains all transactions that are active after the starting point at at least one kernel. Since this AFTER set is used by all kernels for backward log processing, no transaction that is in the AFTER set could be in the BEFORE set at any kernel. Hence, the BEFORE set and the AFTER set of a query will be disjoint sets.

Proof of Requirement 2. As discussed in Section 9.1.1, when using strict 2PL a dependency $T \rightarrow T'$ could only occur if there exists a kernel where $T'$ is active after $T$ has committed. The query controller ensures that all transactions that are active after the commitment of a transaction in the AFTER set, also are included in the AFTER set. Thus, all transactions that could possibly be dependent on a transaction in the AFTER set will also be members of the AFTER set.

In order to prove that the algorithm is correct it must also be shown that the part of the DT logs supplied by the kernels contain sufficient information for the query controller to determine the AFTER set.

The only way the two requirements for transaction-consistency may be violated is if one of the following cases occur:

1. Requirement 1 will be violated if a transaction $T$ that has become uncertain before the starting point at a kernel $x$, becomes active at another kernel $y$ after kernel $y$ has sent the DT-LOG message. $T$ will be a member of the AFTER set at $y$ due to forward log processing. However, it will not be included in the global AFTER set since the query controller will not be aware of that $T$ has been active after a starting point. Hence, $T$ will be a member of the BEFORE set at kernel $x$.

2. Requirement 2 will be violated if a transaction $T$ that is member of the global AFTER set has committed at kernel $y$ before the QUERY-PREPARE message is
received. In other words, the existence of \( T \) at kernel \( y \) will not be reflected by the part of the DT log sent to the query controller. Thus, the controller will not be aware of that transactions that kernel \( y \) participates in may depend on \( T \). Hence, another transaction \( T' \) that is dependent of \( T \) at kernel \( y \) may not be included in the AFTER set.

Case 1 is avoided since the kernel waits until all transactions in the WAIT set have terminated before it sends the QUERY-STARTED message. Thus, when the query controller sends the HISTORY-REQUEST messages, none of the transactions that become uncertain before the starting point is still active.

It will be shown by induction that case 2 is not possible. It is first shown that it could not occur for transactions that are added to the AFTER set in the first iteration of the computation. (That is, transactions that are active after the starting point of at least one kernel.) Then, assuming that the case could not occur for transactions added to the AFTER set in the first \( k \) iterations, it is shown that it could not occur for transactions added in iteration \( k+1 \):

**Proof. (Basis).** Let \( T_1 \) be a transaction that is active after the starting point at kernel \( x \) (i.e., \( t_x(\text{SP}) < t_x(U_1) \)). Hence, Lemma 1 gives that \( t_x(\text{SP}) < t_y(C_1) \), for all kernels \( y \). Since the query controller, before sending the QUERY-START messages, ensures that all kernels have marked the start of the relevant part of the DT log, the commitment of \( T_1 \) could not lie ahead of this mark at any kernel.

**Induction Step.** Consider a transaction \( T_2 \) that is added to the AFTER set in iteration \( k \). The induction step hypothesis is that there exist a kernel \( x \) such that \( t_x(\text{SP}) < t_y(C_2) \) for all kernels \( y \). Assume that there exist a kernel \( x' \) such that \( t_x'(C_2) < t_x'(U_3) \) for some transaction \( T_3 \) not already added to the AFTER set. In other words, \( T_3 \) will be added to the AFTER set in iteration \( k+1 \). The induction hypothesis gives that \( t_x(\text{SP}) < t_x'(C_2) \), for all kernels \( z \). Combining this with Lemma 1 gives that \( t_x(\text{SP}) < t_x'(C_3) \) for all kernels \( z \).

Hence, it is shown by induction that case 2 could not occur for any transaction in the AFTER set.

9.2.6 Optimizations

Compacting the DT log

The message volume could be reduced by compacting the DT log. The query controller does not need all the information contained in the DT log. What the query controller needs to know is which transactions are active after the starting point. In addition, in order to determine dependencies it needs to know which other transactions have been active after a transaction has committed. The latter information is only needed for transactions that are not active after the starting point.

The information on what transactions are active after the commitment of another transaction can be represented by sending a sequence of transaction IDs in commit order. To each transaction ID a dependency number is assigned. This number represent the number of transactions in the sequence that had committed when this transaction became uncertain. Using this scheme, the query controller only needs to send a single number to each kernel to inform about the AFTER set. All transactions that were assigned a dependency number equal to or larger than this number will be part of the AFTER set.
Kernel 0 | Kernel 1 | Kernel 2 | Kernel 3
---|---|---|---
$T_3 : 0$ | $T_4 : 0$ | $T_5 : 0$ | $T_6 : 0$
$T_4 : 0$ | $T_3 : 0$ | $T_3 : 0$ | $T_3 : 0$
$T_7 : 2$ | $T_1$ | $T_4 : 1$ | $T_5 : 2$
$T_6 : 2$ | $T_2$ | $T_2$ | $T_{13} : 3$
$T_9 : 4$ | $T_6$ | $T_6$ | $T_{14}$
$T_2 : 2$ | $T_8$ | $T_8$ | $T_{15}$
$T_{11}$ | $T_{10}$ | $T_{12}$ | 
$T_{11}$ | $T_{11}$ | $T_{11}$ | 
$T_8$ | $T_{12}$ | $T_{14}$ | 
$T_{14}$ | $T_{15}$ | $T_{16}$ | 
$T_{15}$ | | | 

Figure 9.6 Compacted version of the DT logs shown in Figure 9.4.

Figure 9.6 shows the compacted version of the DT log in Figure 9.4. Above the solid line, the transactions that become uncertain before the starting point are listed in commit order together with their dependency number. Below the solid line, the transactions that are active after the starting point are listed.

To assign dependency number each kernel can use a counter that counts the number of committed transactions. A transaction is assigned a dependency number equal to the current value of the counter when it becomes uncertain. In Figure 9.6, a separate counter is used for each query. When $T_2$ becomes uncertain at kernel 0, two transactions have committed ($T_3$ and $T_4$). Hence, it is assigned the dependency number of 2. However, the same counter could be used for all queries by using the current value at the reception of the QUERY-PREPARE message as an offset.

When the query controller is computing the global AFTER set, it first includes all transactions that are listed as active after a starting point. Then, it checks the list of transactions committing before the starting point. It finds $T_6$ as the fourth member of the list from kernel 0. Thus, all transactions with a dependency number greater than or equal to 4 (i.e., $T_6$), should be inserted in the AFTER set. At kernel 3, $T_6$ is third in the list, and all transactions with a dependency number greater or equal to 3 (i.e., $T_{13}$), are inserted in the AFTER set. To inform the kernels about the AFTER set, the query controller only sends the number in the commit list for the first transaction that is member of the AFTER set. Hence, 4 is sent to kernel 0 and 3 is sent to kernel 3. All transactions with a higher dependency number than this number will be added to the local AFTER set before backward log processing. No number is sent to kernel 1 and kernel 2. This means that their viewpoint is equal to their starting point, and their local AFTER set need only contain transactions that were active at their starting point.
9.2 Centralized Computation of the Global AFTER set

Sending QUERY-START messages only to participating kernels

Above it was assumed that the QUERY-START messages was sent to all kernels. However, it is not necessary to send QUERY-START messages to kernels that will not participate in the query. The reason for requiring that all transactions that are active after the starting point are included in the AFTER set, is that log records produced by these transactions will be processed during forward log processing. However, a query will not perform any log processing at non-participating kernels. Thus, at these kernels, it is not necessary to determine any starting point. However, information about transaction dependencies at these kernels is still needed. Thus, non-participating kernels still have to send the DT log produced between the reception of the QUERY-PREPARE and HISTORY-REQUEST messages.

Sending only new information to the query controller

If several queries are run concurrently, their needed parts of the DT log may overlap. In this case, the message volume could be further decreased by only sending the information that has not earlier been sent to the query controller.

This requires that the kernels keep track of which part of the DT log has earlier been sent. This could be done by having a pointer for each query controller. The pointer should point to the last DT log record sent to the corresponding controller. If the next part of the DT log that should be sent to a query controller, overlaps with the previous part, this pointer determines the start of the new information.

Note that in order for the query controller to be able to use the compacted version of the DT log, the calculated dependency numbers should be independent of which query sends the DT log. In other words, either a common counter should be used to determine the dependency numbers, or the dependency number of a transaction should be relative to its position in the commit sequence.

The query controller must be able to determine when DT log records may be garbage collected. Note that a shipment of DT log will only be needed by queries for which the QUERY-PREPARE message has already been sent. Thus, when the global AFTER set is computed for all these queries, DT log records could be deleted.

The distributed model as described in Section 8 does not guarantee that messages of different queries arrive at a query controller in the same order they were sent from the kernel. Thus, if only new information is sent from the kernel, the query controller may have to wait for messages belonging to other queries before being able to compute the global AFTER set.

Using a finer granularity of dependency

Other methods for determining the dependencies between transactions could be used. The method described above uses node granularity dependencies, i.e., all transactions that are active at a kernel after a transaction \( T_1 \) has committed is regarded as depending on \( T_1 \). Using a finer granularity, only transactions that access a common file, block, or data object would introduce a dependency. The dependencies could be maintained using a dependency graph similar to the graphs used in serialization-graph testing [Ber+87].

By determining the dependencies at a finer granularity, instead of sending parts of the DT log, a dependency graph needs to be sent from the kernel to the query.
controller. In addition, a list of the transactions that are active after the starting point must be sent. The dependency graph will give a smaller number of dependencies which means that the computation time and the size of the AFTER set will be smaller. (Less transactions that are not active at any kernel after the starting point will have to be included in the AFTER set.) In other words, less work needs to be done during backward log processing. The penalty is the extra work and space needed to maintain the dependencies. In addition, the query's view of the database will not be defined by a point in time.

To establish transaction dependencies, the read and write sets of a transaction can be established at commit time when the locks of the transaction is released. In addition, the corresponding sets of earlier committed transactions are checked for conflicts, and the dependency graph is updated. As noted earlier, transaction dependencies are only needed for transactions that become uncertain before the starting point. The algorithm ensures that all such transactions have committed before the HISTORY-REQUEST message is received. Thus, all needed transaction dependencies will be established in time.

In order to be able to use finer granularity of dependency, the two requirements for transaction-consistency presented in Section 9.2.5 must still be satisfied. The BEFORE and AFTER set of the query will still be disjoint since the same AFTER set is used at all kernels, and the algorithm still guarantees that no transaction that become active after a kernel has reported its active transactions to the query controller, could have become uncertain before the starting point at some other kernel. In addition, it must be made sure that the necessary transaction dependencies are shipped to the query controller. That is, the query controller must be aware of all transaction dependencies $T \rightarrow T'$ where $T$ is active after the starting point at a kernel and $T'$ is not active after any starting point. Note that the algorithm ensures that no transaction that is active after the starting point at a kernel, or any transaction (transitively) dependent on such a transaction, could have committed before the receipt of the QUERY-PREPARE message at any kernel. Thus, a transaction dependency $T \rightarrow T'$ could only be needed by the query controller if $T$ commits after the receipt of the QUERY-PREPARE message. This means that it is sufficient to maintain transaction dependencies from the receipt of the QUERY-PREPARE message until the WAIT set becomes empty. In other words, as long as no query is in this phase of query execution, maintaining the dependency graph at commit time will not be necessary.

## 9.3 Distributed Computation of the AFTER Set

There are three main drawbacks of the algorithm for centralized computation of the AFTER set. First, several round of message passing between the query controller and the kernels are needed before the participants can start backward log processing. Secondly, the shipments of parts of the DT log from all kernels to the query controller will increase the message volume of the system. Finally, a centralized computation is less scalable since a single node, the query controller, will have to be informed about all transactions in the system. All these drawbacks could be avoided if each kernel could be able to compute its own local AFTER set.

The main idea of the algorithm for distributed computation of the AFTER set is that each kernel will inform all other kernels participating in a transaction, via
the transaction controller, when it includes a transaction in its local AFTER set. If
the participants could receive this information before they commit the transaction,
they would not need to go backward in the log to find dependent transactions. All
transactions that may be dependent on this transaction will still be active. In other
words, no transaction that is neither active after the local starting point nor active
after the commitment of a transaction that has been added to the AFTER set at
another kernel, needs to be added to the local AFTER set. Thus, just as in the
centralized computation, the viewpoint of the query will be whatever comes first of
the starting point and the commitment of a transaction that have been added to the
query's AFTER set at another kernel.

The information from a kernel about the inclusion of a transaction in the AF-
TER set, will have to be sent to the other participants of the transaction via the
transaction controller of this transaction. This could be done by separate messages.
However, by piggybacking the information on the messages of the 2PC protocol, it
can be assured that the information will arrive at the other participants before the
transaction commits. If a transaction is added to the AFTER set at one kernel,
this information will be piggybacked on the YES message for this transaction. The
transaction controller will distribute this information to the other participating ker-
nels by piggybacking it on the COMMIT messages of the transaction. If a kernel,
when receiving the COMMIT message for such a transaction, has not yet received a
QUERY-START message for the query, the viewpoint of this query will be established.
Each kernel could then compute a local AFTER set based on which transactions are
active after the viewpoint of the query. Other kernels will be informed about inclusions
in the local AFTER set of a kernel via the 2PC messages. The global AFTER set
will be the union of all local AFTER sets.

Figure 9.7 shows an example of using the 2PC messages to synchronize the view-
points of a query. Transaction $T$ will be included in AFTER set of $Q$ at Kernel 0
since it is active when the QUERY-START message arrives. This fact is piggybacked
on the YES message to the query controller, and propagated to Kernel 1 via the
COMMIT message. When receiving the COMMIT message, Kernel 0 will establish the
viewpoint for $Q$ since it has not received its QUERY-START message. This way, $T$
and all transactions dependent on $T$ will be included in the local AFTER set.

A similar algorithm to establish AFTER sets for multiversion query locking has
been briefly described by Bober [Bobe93]. Bober uses the AFTER sets to decide
which versions of a tuple is too new for a query, while in the algorithm described
below, the AFTER sets are used to determine which transactions should be rolled
back during backward log processing. Thus, the AFTER set does not need to include
all transactions that are concurrent with a query. It is sufficient to include those
transactions that may be encountered during backward log processing. Messages of
the 2PC-protocol is also used to synchronize nodes in the epoch algorithms presented
in [Gar+90].

### 9.3.1 Local Insertions in the AFTER Set

The decision to insert a transaction in a query's AFTER set will be based on the
local state of the transaction at the viewpoint and the starting point of a query. As
described in Section 8, in a distributed system applying the 2PC protocol to achieve
atomic commitment, a transaction can, at a participating kernel, be in one of the
following states: active, uncertain, committed, aborting or aborted. If a transaction is active at the starting point of the query, or becomes active later, the transaction must be included in the AFTER set as described above. In addition, no transaction that is committed before the viewpoint of the query at one kernel, should be included in the AFTER set. Thus, in order to be able to establish a globally consistent AFTER set, the viewpoints of a query must be synchronized so that a transaction that is active at the starting point at one kernel, can not have committed before the viewpoint at another kernel. By piggybacking information about the starting points of a query on the messages of the 2PC protocol, viewpoints will be established at all kernels before this could happen.

If a transaction is uncertain at the viewpoint at one kernel, no more database operations will be performed at this kernel unless the transaction controller decides to abort the transaction.\footnote{In some variants of the 2PC protocol, an uncertain transaction may later become active. The transaction should then be included in the AFTER set when this happens.} Otherwise, there is no local reason to include the uncertain transaction in the AFTER set. However, the transaction may be either be active after the viewpoint at another kernel, or it may committed before the viewpoint at another kernel. (The piggybacking ensures that both cases can not occur at the same time.) In the first case, the transaction should be included in the AFTER set of all kernels participating in this transaction, while in the second case it should not be included in the AFTER set. Thus, if a transaction is uncertain, the kernel could not on its own decide whether to include the transaction in the AFTER set or not. The decision must be based on the state of the transaction at the viewpoint of other kernels. This information may be piggybacked on the COMMIT message of the transaction. Thus, the AFTER set will not be established until the transactions
that were uncertain at the viewpoint has terminated.

If the transaction is aborting at the starting point, the kernel will include it in its AFTER set since undo operations may occur. This inclusion does not have to be shared with other kernels. If the transaction is active, uncertain or aborting at the starting point of another kernel, this kernel will unilaterally include it in its own AFTER set. If the transaction is aborted, it need not be included in the AFTER set since its operations are not longer reflected in the base relations of the query.

Figure 9.8 shows an example of the execution of some transactions in a system with three kernels. A query is run concurrently with the transactions, and a solid horizontal line gives the starting point of the query at each kernel. At kernel 0 no viewpoint has been established when the QUERY-START message arrives. Thus, the viewpoint of the query at kernel 0 will be equal to the starting point. Transactions \( T_1 \) and \( T_4 \) will be added to the local AFTER set because they are active and aborting, respectively, after the starting point. \( T_3 \) is uncertain at the starting point, and kernel 0 will have to wait until it is terminated to get to know whether it was included in the AFTER set at other kernels. Since \( T_3 \) is not active after the viewpoint at any kernel, kernel 0 will not receive any request from the transaction controller to include \( T_3 \) in its AFTER set. Thus, when kernel 0 receives the commit message for \( T_3 \), it could start backward log processing using \{\( T_1, T_4 \)\} as the AFTER set.

Kernel 1 will by a 2PC message be informed about the inclusion of \( T_1 \) in the AFTER set at kernel 0. Thus, a viewpoint for the query must be established at kernel 1 before the commitment of \( T_1 \), and \( T_1 \) must be included in the AFTER set. \( T_2 \), which is active after the viewpoint, will also be added to the AFTER set, while the decision for \( T_3 \) and \( T_4 \), which are uncertain at the starting point,
will be postponed until their termination. Since $T_4$ is finished aborting before the starting point, it need not be included in the AFTER set. At kernel 1, backward log processing could not start when the QUERY-START message arrives since it will not yet know whether $T_3$ should be included in the AFTER set. When $T_3$ commits, backward log processing can be started using $\{T_1, T_2\}$ as the AFTER set.

At kernel 2, the viewpoint is established before the commitment of $T_2$, which were included in the AFTER set at kernel 1. At the viewpoint it will not be known whether to include $T_4$ in the AFTER set. However, at the starting point $T_4$ is aborting and must be included in the AFTER set. Then, backward log processing could be started using $\{T_2, T_4\}$ as the AFTER set.

### 9.3.2 The Algorithm

When a transaction controller receives a request to run a query, it sends a QUERY-START message to all the kernels that contain data that will be read by the query. When a kernel receives a QUERY-START message, it records the query as being active and enters all currently active or aborting transactions into this query's AFTER set. Transactions that are currently uncertain are entered into a WAIT set for this query if a viewpoint has not already been established. Then, forward log processing is started for the base relations of the query.

When a kernel as part of the first phase of the 2PC protocol receives a VOTE-REQ message, it inserts the transaction in the AFTER sets of all currently active\(^6\) queries for which it has not yet received the QUERY-START message. If the kernel votes Yes, it piggybacks the query IDs of all active queries on the YES message which is sent to the transaction controller. No information is piggybacked on a NO message since each kernel can unilaterally decide whether to insert aborting transactions in their AFTER sets.

If a transaction controller in the second phase of the 2PC decides to commit, it computes the union of the query IDs piggybacked on the received YES messages. This set represents the queries that have added this transaction to their AFTER set at least one kernel, and the transaction should thus be included in the AFTER sets of these queries at all participants in the transaction. The controller piggybacks this set of query IDs on the COMMIT messages sent to the participants of the transaction.\(^7\) As for NO messages, no query IDs need to be piggybacked when ABORT messages are sent.

When a kernel receives a COMMIT message, it checks the piggybacked query IDs. If the transaction is a member of some of the WAIT sets of these queries, the transaction is moved from these WAIT sets to the corresponding AFTER sets. Then, the transaction is removed from the WAIT set of all other queries since it is now known that the transaction has not been added to the AFTER sets of these queries at any of the participating kernels. If a previously unknown query ID is piggybacked on the COMMIT message, the viewpoint of the query should be established. This is done by recording the query as active and by entering all uncertain transactions into

---

\(^6\)A query is active at a kernel from the first time the kernel gets to know about it until it has been notified by the query controller that query processing has ended at all kernels.

\(^7\)A query ID does really only have to be sent to participants that did not include it in their YES message.
9.3 Distributed Computation of the AFTER Set

Kernel receives QUERY-START for Q:

1. \( \text{started}[Q] \leftarrow \text{TRUE} \)
2. \( \text{AFTER}[Q] \leftarrow \{T \mid T \text{ is active or aborting} \} \)
3. if \( Q \notin \text{ACTIVE} \) then
4. \( \text{ACTIVE} \leftarrow \text{ACTIVE} \cup \{Q\} \)
5. \( \text{WAIT}[Q] \leftarrow \{T \mid T \text{ is uncertain} \} \)
6. if \( \text{WAIT}[Q] = \emptyset \) then
7. Start backward log processing
8. Start forward log processing

Kernel sends YES for T:

1. for each \( Q \in \text{ACTIVE} \) do
2. Piggyback \( Q \) on message

Kernel receives COMMIT for T:

1. for each piggybacked \( Q \) do
2. if \( T \in \text{WAIT}[Q] \) then
3. \( \text{AFTER}[Q] \leftarrow \text{AFTER}[Q] \cup \{T\} \)
4. if \( Q \notin \text{ACTIVE} \) then
5. \( \text{ACTIVE} \leftarrow \text{ACTIVE} \cup \{Q\} \)
6. \( \text{WAIT}[Q] \leftarrow \{T \mid T \text{ is uncertain} \} \)
7. for each \( Q \in \text{ACTIVE} \) do
8. if \( T \in \text{WAIT}[Q] \) then
9. \( \text{WAIT}[Q] \leftarrow \text{WAIT} - \{T\} \)
10. if \( \text{WAIT}[Q] = \emptyset \) and \( \text{started}[Q] \) then
11. Start backward log processing

Transaction controller receives YES for T:

1. for each piggybacked \( Q \) do
2. \( \text{BEFORE}[T] \leftarrow \text{BEFORE}[T] \cup \{Q\} \)

Transaction controller sends COMMIT for T:

1. for each \( Q \in \text{BEFORE}[T] \) do
2. Piggyback \( Q \) on message

Figure 9.9 Distributed algorithm for computing the AFTER set.

the query's WAIT set. The kernel will then start maintaining the AFTER and WAIT sets for this query as described above. Note that at this point the kernel does not know whether the query will ever read any data on this node. However, these actions are necessary in order to make sure that transaction dependencies at this kernel are reflected in the AFTER sets of other kernels. Also, it is necessary to establish the viewpoint in case a QUERY-START message for the query later arrives.

If an ABORT message is received for a transaction, all non-empty WAIT sets are checked for this transaction, and where found, it is moved to the corresponding AFTER set if the QUERY-START message for the corresponding query has already been received. Otherwise, the transaction is just removed from the WAIT set. The transaction will be inserted in the AFTER set at the starting point if it has not finished aborting by then.

When both the WAIT set has become empty, and the START-QUERY message has been received, the query will start backward log processing from the starting point using the computed AFTER set. From this point on the kernel does not need to maintain its local AFTER set. However, it must still piggyback the query ID on the YES messages since other kernels may not yet have emptied their WAIT sets. In Section 9.3.5 a protocol for determining when the kernels can stop piggybacking the query ID is described. When backward log processing is finished, the scan thread of the query can start reading data at this node.

The algorithm is summarized in Figure 9.9. All sets are assumed to be initially empty and \( \text{started} \) is initially false for all queries. A query is removed from the ACTIVE set when piggybacking for this query is no longer necessary.

---

8This must be done before the kernel commits the transaction in order for the transaction to be correctly included in the AFTER set of the query.
9.3.3 Example

Figure 9.10 shows a message sequence diagram of four transactions and three queries running in a system with two nodes. To simplify the example, it is assumed that no transaction failures occur. The messages produced by the transactions are represented as dashed lines. For each transaction only the initial startup message and the messages of the 2PC protocol are shown. Within the square brackets of the YES and COMMIT messages, the query IDs piggybacked on these messages are listed. The QUERY-START messages issued by the query controllers are represented as solid lines. It is assumed that all the queries are active for the rest of the time frame covered by the diagram. The bidirectional arrows at the margin indicate the time interval when the kernel waits for the WAIT set of the given query to become empty.

When kernel 0 receives the QUERY-START message for \( Q_1 \), transactions \( T_1, T_3 \), and \( T_4 \) are active, and these transactions will be inserted in the AFTER set of \( Q_1 \). Since no transaction is uncertain, the WAIT set is empty, and backward log processing can be started immediately. During backward log processing the log records produced by the three transactions in the AFTER set are processed and relevant information is stored in the update-table. When kernel 0 later receives the VOTE-REQ message for \( T_3 \), \( Q_1 \) is the only active query, and the query ID of \( Q_1 \) is piggybacked on the YES message. At kernel 1, \( Q_2 \) is the only active query when the VOTE-REQ message is received, and the query ID of \( Q_2 \) is piggybacked on the YES message. Note that since backward log processing has been started for the corresponding queries, the kernels have stopped maintaining their local AFTER sets when the VOTE-REQ messages for \( T_3 \) are received.

When the transaction controller at node 0 has received the YES messages from both participants in \( T_3 \), it computes the union of the piggybacked query: \( \{Q_1, Q_2\} \). This is the global AFTER set which should be piggybacked on the COMMIT messages. However, it is not necessary to piggyback a query ID on a message to a kernel that has already inserted the transaction in the query’s AFTER set. Thus, only \( Q_2 \) is piggybacked on the COMMIT message sent to kernel 0, and only \( Q_1 \) is piggybacked on the message sent to kernel 1.

When kernel 0 receives the COMMIT message for \( T_3 \), it finds the query ID for \( Q_2 \) in it. At this point in time, kernel 0 has not yet received the QUERY-START message for \( Q_2 \). It will then create a WAIT set for \( Q_2 \) consisting of \( T_3 \), the only uncertain transaction at that point. \( T_3 \) will immediately be moved to the AFTER set when processing the COMMIT message for \( T_3 \), and the WAIT set will be now be empty. Note that \( Q_2 \) from this point on is regarded as active at kernel 0. Thus, when the kernel later receives a VOTE-REQ message for \( T_4 \), both the query IDs of \( Q_1 \) and \( Q_2 \) are piggybacked on the YES message, and \( T_3 \) is inserted in the AFTER set of \( Q_2 \). (Kernel 1 has stopped maintaining the local AFTER set for \( Q_1 \).) When kernel 0 receives the QUERY-START message for \( Q_2 \), it will insert the currently active transactions, \( T_1 \) and \( T_2 \), into the AFTER set and, since the WAIT set of \( Q_2 \) is empty, immediately start backward log processing for \( Q_2 \) using the computed AFTER set which includes all four transactions.

When kernel 1 receives the QUERY-START message for \( Q_1 \), the active transactions \( (T_1, T_2, \text{ and } T_3) \) are inserted in the query’s AFTER set. In addition, \( T_3 \) is inserted in the WAIT set of \( Q_1 \) since it is uncertain at the starting point. Hence, kernel 1 can not start backward log processing for \( Q_1 \), until it receives the COMMIT message.
Figure 9.10 Message sequence diagram illustrating the distributed computation of the AFTER set.
for $T_3$. When the commit message for $T_3$ arrives, the query ID of $Q_1$ is piggybacked on it. Thus, $T_3$ is moved from the WAIT set to the AFTER set of $Q_1$. Now the WAIT set is empty, and backward log processing is started using an AFTER set consisting of all four transactions. Note that $T_2$ was not a member of the AFTER set used for backward log processing of $Q_1$ at kernel 0. The execution of $Q_1$ will still be transaction-consistent since all log records produced by $T_2$ at kernel 0 will be processed during forward log processing.

When the query-start message for $Q_3$ arrives at kernel 1, $T_2$ is active and inserted in the AFTER set while $T_1$ is uncertain and therefore added to the WAIT set of $Q_3$. When the commit message for $T_1$ arrives, no query ID is piggybacked on it. However, there is one non-empty WAIT set, that of $Q_3$, that must be checked for $T_1$. When $T_1$ is removed from the WAIT set, and backward log processing is started. When kernel 0 receives the query-start message for $Q_3$, there are two uncertain transactions, $T_1$ and $T_4$. Kernel 0 will wait for the commit messages of both transactions before starting backward log processing. Since none of the commit messages have piggybacked query IDs, none of the transactions are added to the AFTER set. As on kernel 1, the only transaction in the AFTER set of $Q_3$ is $T_2$ which is the only transaction still active at the starting point.

9.3.4 Proof of Correctness

As discussed in Section 9.2.5, the following two requirements must be satisfied in order to give a query a transaction-consistent view of the database:

1. The query's BEFORE set and AFTER set must be disjoint.

2. No transaction in the BEFORE set is dependent on a transaction in the AFTER set.

To prove that these two requirements hold, it will first be assumed that no transaction aborts. Later it will be shown that these requirements also hold if transactions aborts.

Proof of Requirement 1. If a transaction is inserted in the AFTER set at one kernel, the query ID will be distributed to all other participating kernels through the YES and COMMIT messages if the kernel receives the VOTE-REQ while the query is still active. All kernels that receive the COMMIT message with the piggybacked query ID, will include the transaction in their AFTER set if the kernel has not yet received the query-start message for this query, or if the transaction is a member of the WAIT set of the query. Otherwise, the transaction was either active at the starting point and added to the AFTER set then, or started after the starting point. Transactions that it is started after the starting point will always be in the AFTER set since all its log records will be processed by forward log processing. In other words, no members of the AFTER set at one kernel could be members of the BEFORE set at another kernel if they become uncertain while the query is active.

If the query receives the VOTE-REQ message for a transaction $T_1$ when the query is not active anymore, $T_1$ must have been active after the starting point at all the other participants (i.e., $t_x(SP) < t_x(U_1)$ for all kernels $x$ participating in $T_1$). If $T_1$ was uncertain at the starting point at one kernel, the WAIT set at this kernel would still not be empty, and the query would still be active. If $T_1$ should have
committed before the starting point at one kernel (i.e., there exists a kernel \( y \) such that \( t_y(C_1) < t_y(SP) \)), then Lemma 1 gives that when using the 2PC protocol, \( t_x(U_1) < t_y(SP) \) for all kernels \( x \) participating in \( T_1 \). Since the query is active at all kernels until query processing has stopped at all kernels, the query will still be active at all participating kernels when the \textsc{vote-req} message is received. Thus, \( T_1 \) could not be uncertain or committed at the starting point of any kernel. As shown above, a transaction that is active after the starting point of a kernel will always be included in the \textsc{after} set at this kernel. Thus, a transaction that become uncertain at a kernel when a query is not active anymore, will be member of the \textsc{after} set of all participating kernels.

\textbf{Proof of Requirement 2.} As discussed in Section 9.1.1, when using strict 2PL a dependency \( T_1 \rightarrow T_2 \) could only occur if there exists a kernel \( x \) such that \( t_x(C_1) < t_x(U_2) \). If \( T_1 \) is a member of the \textsc{after} set at kernel \( x \), then \( t_x(VP) < t_x(C_1) \) since either a viewpoint will have been establish before the commitment of \( T_1 \) because \( T_1 \) was added to the \textsc{after} set of a another kernel, or \( T_1 \) was added to the \textsc{after} set because it was active after the the starting point at kernel \( x \). Thus, \( t_x(VP) < t_x(U_2) \) and \( T_2 \) will be added to the \textsc{after} set when receiving the \textsc{vote-req} message if \( t_x(U_2) < t_x(SP) \). If \( t_x(SP) < t_x(U_2) \), \( T_2 \) is active after the starting point, and it follows from the proof of Requirement 1 that \( T_2 \) will be member of the \textsc{after} set. Hence all transactions that are dependent on a transaction in the \textsc{after} set will also be members of the \textsc{after} set.

If a transaction aborts, the insertion in the \textsc{after} set is not distributed to the other participants. Thus, an aborted transaction will be in the \textsc{before} set at the participants where it has terminated before the starting point, and in the \textsc{after} set at the other participants. This will still give a transaction-consistent query result since no effects of an aborted transaction is visible in the database, which implies that no transaction can be dependent on an aborted transaction.

### 9.3.5 Optimizations

\textbf{Reducing the amount of piggybacked query IDs}

As described above, each kernel will piggyback a query ID on all \textsc{yes} messages sent while the query is active. However, an explicit representation of the \textsc{after} set is only needed for backward log processing, and a kernel does not need to be informed about inclusions in the \textsc{after} set anymore when its \textsc{wait} set has become empty. Thus, the kernels should inform the query controller when their \textsc{wait} set has become empty by sending a \textsc{query-started} message. When the query controller has received \textsc{query-started} messages from all participants, it sends \textsc{query-initialized} messages to all kernels. When a kernel receives this message, it stops maintaining the \textsc{after} set for the query. That is, the query ID is not added to \textsc{yes} messages anymore. Note that the kernels that does not participate in the query must keep information about the existence of the query for some time after receiving the \textsc{query-initialized} message so that it does not start maintaining another \textsc{after} set if it later receives a \textsc{commit} message on which the query ID of this query is piggybacked.

In Figure 9.11, \textsc{query-started} and \textsc{query-initialized} message have been added to the example presented in Figure 9.10. Since both kernel 0 and kernel
Figure 9.11 QUERY-STARTED and QUERY-INITIALIZED messages added to the example of Figure 9.10.
Reducing the time before a kernel can start backward log processing

If there is a long period between when a kernel receives the VOTE-REQ message and the COMMIT or ABORT message, queries may be blocked for a long time. To avoid this, a kernel can send a ROLLBACK-REQ message to the transaction controller in order to request that the transaction is added to the AFTER set of a particular query by all participating kernels. Upon receiving a ROLLBACK-REQ message, the transaction controller, if it has not already sent COMMIT/ABORT messages, sends a ROLLBACK message, including the particular query ID, for this transaction to all the participants.\(^9\) A kernel that receives such a message will move the transaction from the WAIT set to the AFTER set of the query.

This optimization will not compromise transaction-consistency since all participating kernels will be informed about the inclusion of a transaction in the AFTER set, and all transactions dependent on such a transaction will also be included in the AFTER set since viewpoints will be established before the commitment of the transaction at all kernels.

In Figure 9.12, which shows the last part of the example from Figure 9.10, kernel 0 sends a ROLLBACK-REQ message to the transaction controller of \(T_1\), requesting the insertion of \(T_1\) in the AFTER set of \(Q_3\). When kernel 0 receives the corresponding ROLLBACK messages, it moves \(T_1\) from the WAIT set to the AFTER set. Now, kernel 0 does not have to wait for the COMMIT message of \(T_1\) before starting backward log processing. The penalty for this, is that more work must be done during backward log processing since \(T_1\) would not otherwise have been inserted in the AFTER set.

Kernel 1 receives the ROLLBACK message after the COMMIT message for \(T_1\). This shows the importance of piggybacking the query ID on the COMMIT message even if a rollback decision has been made. Otherwise, kernel 1 would not have processed log records for \(T_1\) during backward log processing.

In general, the kernels do not have to wait until sending the YES messages to inform the transaction controllers about insertions in the AFTER sets. This information could be piggybacked on earlier messages, e.g., ACK messages, sent to the transaction controller. Similarly, a controller could use the first message it sends to a kernel to inform about insertions in the AFTER set.

Representing dependencies at finer granularity in order to reduce the size of the AFTER set

In Section 4.3.4 it was shown how representing dependencies on finer granularity may reduce the size of the AFTER set for later queries in a multi-query transaction in a centralized database system. In a distributed database system, it may be useful to

\(^9\)The controller must still piggyback the query ID on the COMMIT messages in case a kernel receives the COMMIT message before the ROLLBACK message. (Message sequence is not guaranteed between transactions.)
Figure 9.12 Example of using the ROLLBACK-REQ message to request the insertion of an uncertain transaction in the AFTER set.

use a variant of the same coloring algorithm also for single-query transactions. In the distributed algorithm it may take some time from the establishment of a viewpoint for a query to the start of backward log processing. The algorithm described above inserts all transactions that are active at the kernel after the viewpoint in the AFTER set. In other words, node is the granularity of transaction dependencies. Some of these transactions may terminate before the starting point. If these transactions are not dependent on any other transaction in the AFTER set, they could be removed from the AFTER set without compromising the consistency of the query. Other transactions that are active after the starting point, does not access data in the query’s domain. These transactions could be removed from the AFTER set if they are not dependent on any other transaction in the AFTER set.

To color a transaction black is equivalent to inserting it in the AFTER set. Thus, a transaction will have one color status for each concurrent query. The color of a transaction with respect to a given query is propagated to the other kernels by piggybacking the query ID on the YES message of the transaction. All data items are originally white and will turn black if accessed by a black transaction. In addition, all data items belonging to a query’s domain\(^{10}\), will turn black at the starting point

---

\(^{10}\)The domain of a query is the part of the database for which the log processing thread will maintain an updateable [e.g., the base relations of the query].
of the query. For single-query transactions, an update transaction will be black if it has accessed a data item that has turned black before the transaction turned uncertain. Thus, all transactions that access a data item in the query’s domain and are active after the starting point, will be black.

When a kernel is going to send a YES message for a transaction, it must check the color information of each data item locked by the transaction. The query IDs for all queries for which at least one of the data items is black should be piggybacked on the message, and the color for all items with respect to this query should be set to black. Note that when a kernel is doing backward log processing, it will have to insert all active transactions at the starting point into the local AFTER set since the color of active transactions is not yet known. However, some of these transactions may never be colored black because they do not access any data item in the query’s domain, and the insertion of these transactions will never be propagated to other kernels.

This coloring algorithm reduces the size of the AFTER set in two ways:

- Instead of adding all transactions that are active after the viewpoint to the AFTER set, transactions terminating before the starting point will not be included in the AFTER set if they are not dependent on a transaction that was inserted in the AFTER set at another kernel.

- Instead of adding all transactions that are active after the starting point to the AFTER set, transactions that do not access the query’s domain are not added to the AFTER set unless they are dependent on another transaction in the AFTER set.

It is doubtful that the reduction in the size of the AFTER set will justify the extra work needed by update transactions to maintain color information and the cost of storing this information for all data items. For single-query transactions in a system with small communication delays, the time between the viewpoint and the starting point will be short, and few transactions will terminate during this interval. The use of the other two optimizations presented above will also reduce the benefit of a smaller AFTER set. In any case, it will probably never be cost-efficient to maintain color information at tuple or block granularity, especially since the query’s domain usually will be expressed at a coarser granularity.

### 9.4 Implementing WAIT Sets and AFTER Sets

The implementation of WAIT sets and AFTER sets must support efficient insert, lookup, and deletion. It is especially important to minimize the work required by update transactions to maintain these sets. To support efficient access by transactions, two bitmaps, one for each of the two types of sets, can be associated with each transaction. Each bit in the bitmap corresponds to a specific query. These two bitmaps will be called the wait bitmap and the after bitmap. The bit for a query will be set if the transaction is a member of the corresponding set for this query. In order to detect when a set becomes empty, a counter for each set can be associated with each query. The counter is incremented if a transaction is added to a set and decremented when a transaction is removed from the set.
When a query is adding transactions to its WAIT set, it will scan the transaction table for transactions in the uncertain state. For each transaction found, it will set the bit associated with this query in the wait bitmap of the transaction. In addition, the counter associated with the WAIT set for this query is incremented. When a transaction terminates, all counters associated with queries for which the WAIT bit is set, is decremented. Then, all bits of the transaction's wait bitmap is reset. In other words, the terminated transaction is removed from all WAIT sets. When a counter has reached zero, the WAIT set of the corresponding query will be empty and the kernel will start backward log processing if it has received a QUERY-START message for this query. Note that the update of counters need not be done by transactions themselves. The transactions may just copy their wait bitmap to a temporary buffer when they commit. The bitmaps could later be processed by queries which will update the counters.

For the distributed computation, the implementation of the AFTER set could be similar to the implementation of the WAIT set. At the starting point of a query, the bit corresponding to this query will be set in the AFTER bitmap of all active transactions. In addition, when a transaction commits, the bit of all queries for which the query ID is piggybacked on the COMMIT message, is set. When sending the YES message for a transaction, query IDs corresponding to the bits that is set in its AFTER bitmap will be piggybacked on the message. Before doing backward log processing, the query will scan the transaction table looking for transactions that have set their corresponding bit in the AFTER bitmap. For each transaction, the LSN of the last log record produced by the transaction is added to a list of LSNs representing log records that should be processed during backward log processing. In addition, the bit in the AFTER bitmap is reset. During backward log processing the list of LSNs is maintained as described in Section 7.1.1. Note that a transaction should not be removed from the transaction table until no bits in its AFTER bitmap is set.

The bitmaps may be indexed by sequence numbers locally assigned to queries. A bitmap only need to represent the WAIT sets and AFTER sets that are computed during the lifetime of the transaction. Thus, the first entry of the bitmap of a transaction could be indexed by the sequence number of the oldest query not finished computing its AFTER set when the transaction is started. Note that this start index may be changed during the execution of the transaction. Thus, when a query is scanning the transaction table, inserting active and uncertain transactions in its WAIT set and AFTER set, respectively, it may update the start index of a bitmap to the currently oldest query not finished computing its AFTER set in order to make room for more queries. In addition, the bitmap must be shifted to the left to reflect the change in start index.

For the centralized computation, no bitmap is needed to implement the AFTER set. However, the LSN of the last log record produced before the starting point by members of the AFTER set must be available when the AFTER set is received from the query controller. The LSNs for the active transactions at the starting point can be added at that time. In addition, the LSNs of the last records of transactions that are uncertain or committed at the starting point must be recorded.

\[11\] The AFTER bitmap of the transaction could be directly piggybacked to the YES messages if queries were assigned the same sequence number at all nodes. However, coordinating the assignment of sequence numbers will affect query performance.
9.5 Forms of Query Consistency

Garcia-Molina and Wiederhold introduced the concepts of strong and weak consistency for read-only transactions in a distributed system [GaWi82]. Strong consistency assures serializable execution of read-only transactions, while weak consistency only guarantees that each individual read-only transaction is serializable with respect to updating transactions. Bober and Carey added two other forms of consistency; strict and update consistency [BoCa92a]. Strict consistency, which requires that a query's view is consistent with the commit order of transactions, is not applicable to distributed database system where commit order of transaction may differ between nodes.

The two algorithms presented above will both guarantee weak query consistency, but not strong query consistency. In order to achieve strong consistency, serializability of queries must be enforced. For both algorithms this can be achieved by using the same query coordinator for all queries. When this query coordinator has sent the QUERY-START messages for a query, it should not send QUERY-START messages for other queries until it has received all QUERY-STARTED messages for the first query. This way, all starting points of the first query will come before any starting points of later queries, and serialization order of queries will be well defined. If strong consistency is only required for a subset of the queries, queries may form strong consistency groups [BoCa92a]. All queries of a strong consistency group will use the same query coordinator, while queries outside the group may use any query coordinator.

As mentioned in Chapter 3, for update query consistency, read-write dependencies between transactions may be ignored. As discussed in Section 4.3.7, strict execution of transactions is still guaranteed if read locks are released between the lock point and the commit point. This can be exploited in distributed database systems by releasing read locks when the VOTE-REQ message is received. It follows that commit messages will not be needed for kernels that have only performed read operations [Sam+95]. This will mean that a transaction \( T_1 \) which commits before another transaction \( T_2 \) may be read-write dependent on \( T_2 \). Such dependencies will not be recognized by the definition of dependency as presented in Definition 1 and used in both algorithms presented in this chapter. Hence, only update consistency will be guaranteed.

By changing the definition of dependency to also take into account all read-write dependencies when read locks are released before commit time, weak query consistency can still be achieved. The new definition assumes that for each transaction the last time it obtained a lock and the first time it released a lock are recorded. Let \( t_e(O_1) \) and \( t_e(R_1) \) be the times a transaction \( T_1 \) obtains its last lock and releases its first, respectively. Then, the following definition of dependency may be used:

**Definition 3.** A transaction \( T_2 \) is dependent on another transaction \( T_1 \) (\( T_1 \to T_2 \)) iff there exists a kernel \( x \) such that \( t_e(R_1) < t_e(O_2) \).

As long as locking is two-phased, circular dependencies may not occur when
Table 9.1 Parameters used in performance comparison.

<table>
<thead>
<tr>
<th>Name</th>
<th>Explanation</th>
<th>Typical Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_S$</td>
<td>Total number of nodes in the system</td>
<td>$2\ldots 32$</td>
</tr>
<tr>
<td>$n_Q$</td>
<td>Number of kernels participating in the query</td>
<td>$n_S/2$</td>
</tr>
<tr>
<td>$\mu_T$</td>
<td>Average number of kernels participating in a transaction</td>
<td>$3$</td>
</tr>
<tr>
<td>$t_m$</td>
<td>Processing time used to send or receive a message.</td>
<td>$0.12$ ms</td>
</tr>
<tr>
<td>$t_{WAIT}$</td>
<td>Average time to empty the WAIT set</td>
<td></td>
</tr>
<tr>
<td>$S_{Header}$</td>
<td>Size of a message header</td>
<td>$16$ bytes</td>
</tr>
<tr>
<td>$S_{ID}$</td>
<td>Size of a transaction (or query) ID</td>
<td>$8$ bytes</td>
</tr>
<tr>
<td>$S_{tart}$</td>
<td>Average size of the body of the QUERY-START message</td>
<td>$256$ bytes</td>
</tr>
<tr>
<td>$f_T$</td>
<td>Number of transactions executed per second in the system</td>
<td>$0\ldots 1000$</td>
</tr>
<tr>
<td>$\tau$</td>
<td>Average lifetime of a transaction</td>
<td>$30$ ms</td>
</tr>
</tbody>
</table>

using Definition 3. Note that actual clock times do not have to be recorded for lock operations. For example, assuming monotonically increasing LSNs, the currently latest allocated LSN may be used instead of clock time.

Adjusting the algorithm for the centralized computation of the AFTER set to this definition of dependency is straightforward. Instead of sending the DT log, $t_{\text{e}}(O_i)$ and $t_{\text{r}}(R_i)$ will be included in the DT-LOG message for each transaction $T_i$ that is active after the reception of the QUERY-PREPARE message at kernel $x$. In addition, the LSN of the last log record created by $T_i$ ($LastUpdate, LSN_{x,i}$) is included. The maximum allocated LSN at the starting point ($Start, LSN_x$) will also be included in the message. When computing the AFTER set, all transactions $T_i$ for which $Start, LSN_x < LastUpdate, LSN_{x,i}$, for some kernel $x$, will first be added to the AFTER set. Then, all transactions that according to Definition 3 are dependent on a transaction in the AFTER set are also included in the AFTER set. This operation is repeated until no more transactions are inserted in the AFTER set.

### 9.6 Performance Analysis

To analyze and compare the performance of the distributed and centralized computation of the AFTER set, the following metrics will be used:

$N =$ Number of (extra) messages needed to execute a query.

$T =$ Average time from the query controller starts the query until the kernels can start backward log processing.

$S =$ Total size of the extra information sent in order to execute the query.

To simplify the analysis it is assumed that the time used to send a message is independent of the extra messages introduced by the two algorithms. In other words, the increase in the number of messages and the total size of messages does not influence the time used to send a message between two nodes. In other words, the analysis will give a lower bound on the time needed to compute the AFTER set. It is also assumed that the inter-node message latency is ignorable compared to the processing time associated with sending and receiving messages. This is normally the case in local networks [Gray88]. The parameters defined in table 9.1 will be used
Table 9.2 Extra parameters for the centralized computation of the AFTER set.

<table>
<thead>
<tr>
<th>Name</th>
<th>Explanation</th>
<th>Typical Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>Average number of transactions in the DT-LOG message</td>
<td></td>
</tr>
<tr>
<td>$t_{\text{AFTER}}$</td>
<td>Time used by the query controller to compute the AFTER set</td>
<td></td>
</tr>
<tr>
<td>$S_{\text{log}}$</td>
<td>Average size of each entry of the DT-LOG message</td>
<td>12 bytes</td>
</tr>
<tr>
<td>$S_{\text{AFTER}}$</td>
<td>Average size of the body of AFTER-SET message</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

when giving equations for these metrics. The typical values listed for the parameters are based on measurements of the ClustRa DBMS [Hva95]. These values will be used when comparing the performance of the two algorithms.

9.6.1 Analysis of the Centralized Computation

For the analysis of the centralized computation of the AFTER set, it is assumed that QUERY-START messages are only sent to kernels that are going to participate in the query. It is also assumed that the compact representation of the DT log is used. When using the compact representation of the DT-log, each entry in the DT log will contain a transaction ID and a sequence number indicating the time this transaction turned uncertain. As described in Section 9.3.5, only the sequence number is needed in order to inform the kernels about the computed AFTER set. Table 9.2 gives some extra parameters needed for the analysis of the centralized computation.

Based on the message sequence diagram of Figure 9.3, and the fact that the query controller only needs to communicate with non-participating kernels in the first and the third round of the protocol, it follows that the number of messages needed to establish a transaction-consistent AFTER set can be expressed as:

$$N_c = 4n_S + 3n_Q.$$  \hfill (9.1)

To model the time used to establish the AFTER set, it is assumed that the time between when the query controller starts sending messages to $n$ kernels and when it has received replies from all $n$ kernels can be modeled as $(n+3)t_m + t_k$, where $t_k$ is time between reception of the message at the kernel and the sending of the reply. The controller will use $nt_m$ time units to send all messages, and each kernel will use another $2t_m + t_k$ to process the incoming message and send its reply. Thus, the query controller will have received the reply from the last kernel $nt_m + 2t_m + t_k + t_m = (n+3)t_m + t_k$ time units after it started sending the messages. The time used to execute the entire centralized computation is illustrated in Figure 9.13. Since the time used for processing messages is assumed to be constant, the query controller will have processed the replies from all other kernels before receiving the reply from the last kernel.

The average time before backward log processing is started at the kernels will be the time until the query controller has computed the AFTER set plus $(n_Q/2+1)t_m$, which is the average time before a kernel has processed the AFTER-SET message:

$$T_c = (n_S + 3)t_m + (n_Q + 3)t_m + t_{\text{WAIT}} + (n_S + 3)t_m + t_{\text{AFTER}} + (n_Q/2 + 1)t_m$$

$$= (2n_S + 3n_Q/2 + 10)t_m + t_{\text{WAIT}} + t_{\text{AFTER}},$$  \hfill (9.2)

where $t_{\text{WAIT}}$ is the time spent waiting for the WAIT set to become empty, and $t_{\text{AFTER}}$ is the time used by the query controller to compute the AFTER set. The
average time a transaction is uncertain will be \( (n_T + 2)t_m \) since the transaction controller will send COMMIT messages to \( n_T \) kernels. Since there will generally be several members of the WAIT set at a kernel, \( t_{\text{WAIT}} \) will probably be close to this maximum time:

\[
t_{\text{WAIT}} = (n_T + 2)t_m.
\]

(9.3)

In other words:

\[
T_c = (2n_s + 3n_q/2 + 10)t_m + (n_T + 2)t_m + t_{\text{AFTER}}
\]

\[
= (2n_s + 3n_q/2 + n_T + 12)t_m + t_{\text{AFTER}}.
\]

(9.4)

This execution time could be significantly reduced if the network supports multicast message routing [Hwan93]. Then, the processing cost of sending the same message to many nodes could be halved [Gray88].

The formulas for the size of the different messages of the centralized computation are listed in Table 9.3. The size of the DT-LOG message will depend on the number of transactions that are active between the receptions of the QUERY-PREPARE and HISTORY-REQUEST messages. Figure 9.13 gives that this time interval will be

\[
t_\alpha = (n_s + n_q + 6)t_m + t_{\text{WAIT}} = (n_s + n_q + n_T + 8)t_m.
\]

(9.5)

The average number of transactions in a DT-LOG message will be:

\[
\alpha = \frac{n_T}{n_s} f_T (t_\alpha + \tau).
\]

(9.6)
Table 9.3 The number and size of messages used in the centralized computation.

<table>
<thead>
<tr>
<th>Message</th>
<th>Number</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUERY-PREPARE</td>
<td>ns</td>
<td>$S_{\text{header}} + S_{\text{TID}}$</td>
</tr>
<tr>
<td>ACK</td>
<td>ns</td>
<td>$S_{\text{header}} + S_{\text{TID}}$</td>
</tr>
<tr>
<td>QUERY-START</td>
<td>nq</td>
<td>$S_{\text{header}} + S_{\text{TID}} + S_{\text{start}}$</td>
</tr>
<tr>
<td>QUERY-STARTED</td>
<td>nq</td>
<td>$S_{\text{header}} + S_{\text{TID}}$</td>
</tr>
<tr>
<td>HISTORY-REQUEST</td>
<td>ns</td>
<td>$S_{\text{header}} + S_{\text{TID}}$</td>
</tr>
<tr>
<td>DT-LOG</td>
<td>ns</td>
<td>$S_{\text{header}} + S_{\text{TID}} + \alpha S_{\text{log}}$</td>
</tr>
<tr>
<td>AFTER-SET</td>
<td>nq</td>
<td>$S_{\text{header}} + S_{\text{TID}} + S_{\text{AFTER}}$</td>
</tr>
</tbody>
</table>

The total size of the messages used for the centralized computation of the AFTER set can be found by adding together the sizes of each message type as given by Table 9.3:

$$ S_c = ns(S_{\text{header}} + S_{\text{TID}}) + ns(S_{\text{header}} + S_{\text{TID}}) + nq(S_{\text{header}} + S_{\text{TID}} + S_{\text{start}}) + $$

$$ nq(S_{\text{header}} + S_{\text{TID}}) + ns(S_{\text{header}} + S_{\text{TID}}) + nq(S_{\text{header}} + S_{\text{TID}} + \alpha S_{\text{log}}) + $$

$$ nq(S_{\text{header}} + S_{\text{TID}} + S_{\text{AFTER}}) $$

$$ = (4ns + 3nq)(S_{\text{header}} + S_{\text{TID}}) + nq(S_{\text{start}} + S_{\text{AFTER}}) + nT_f(t_a + \tau)S(\theta_a) $$

### 9.6.2 Analysis of the Distributed Computation

For the analysis of the distributed computation of the AFTER set, it is assumed that the extra QUERY-STARTED and QUERY-INITIALIZED messages are used to end the piggybacking of the query ID to the 2PC messages. While the query controller sends QUERY-START messages only to participating kernels, QUERY-INITIALIZED messages must be sent to all kernels of the system. The total number of extra messages will be:

$$ N_d = 2n_q + n_s. \quad (9.8) $$

The time before a kernel can start backward log processing will consist of the time to receive the QUERY-START message plus the time spent waiting for the WAIT set to become empty. The average time before a kernel receives the QUERY-START message is $(n_q/2 + 1)t_m$ since the query controller sends the message to $n_q$ kernels.

$$ T_d = (n_q/2 + 1)t_m + t_{\text{WAIT}} = (n_q/2 + n_T + 3)t_m. \quad (9.9) $$

The total size of the piggybacked query IDs is dependent on the time between the receptions of the QUERY-START and the QUERY-INITIALIZED messages. For the kernels participating in the query, the average for this time period can be modeled as

$$ T_{\text{init}} = (n_q/2 + 1)t_m + t_{\text{WAIT}} + (n_s/2 + 3)t_m $$

$$ = (ns/2 + n_q/2 + n_T + 6)t_m. \quad (9.10) $$

For other kernels this time period will be smaller since they will start piggybacking when they are informed about the query from a kernel that has earlier received the
QUERY-START message. However, in order to simplify the analysis, it is assumed that $T_{av}$ is the average time spent piggybacking at all kernels.

The total number of YES messages on which a given query ID will be piggybacked will be $f_T \cdot n_T \cdot T_{init}$, and approximately the same number of COMMIT messages will be involved. The total extra size of the messages of the distributed computation of the AFTER set will be the sum of the size of the extra messages and the size of the extra information added to the 2PC messages. Assuming that the QUERY-STARTED and the QUERY-INITIALIZED messages will just contain a header and a query ID ($S_{header} + S_{TID}$), the total extra size of the messages for a query will be:

$$S_d = n_q(S_{header} + S_{TID} + S_{start}) + n_q(S_{header} + S_{TID}) +$$
$$n_s(S_{header} + S_{TID}) + 2f_T \cdot n_T \cdot T_{init} \cdot S_{TID}$$

$$= (n_s + 2n_q)(S_{header} + S_{TID}) + n_q S_{start} +$$
$$f_T \cdot n_T \cdot t_m(n_s + n_q + 2n_T + 12)S_{TID}$$

(9.11)

### 9.6.3 Performance Comparison

It is evident from the above that the distributed computation performs better both with respect to time and number of extra messages. Comparing (9.1) to (9.8) and (9.2) to (9.9) gives that the distributed computation will perform better with respect to these two metrics for all possible parameter values. A closer inspection of the message volume also shows that the distributed method will always have the lowest message volume. Since a transaction ID will be included in each entry in the DT-LOG message, $S_{log}$ will be greater than $S_{TID}$. It follows from (9.7) and (9.11) that in order for $S_c$ to be less than $S_d$, $\tau$, the average lifetime of a transaction, must be substantial less than $(n_T + 1)f_m$. However, the latter is less time than it takes to execute the 2PC protocol. Hence, the extra message volume caused by the distributed computation will be less than the message volume of the centralized computation for all possible parameter values. In addition, the centralized computation will benefit the most from networks supporting multicasings.

Figures 9.14, 9.15, and 9.16 illustrate the performance of the two methods for computing the AFTER set using the typical parameter values given in Tables 9.1 and 9.2.\(^\text{12}\)

The centralized computation needs about three times as many messages and uses about six times as long time as the distributed computation. Increasing the transaction throughput will also give a larger increase in message volume with a centralized computation. Even if the execution time is much longer for the centralized computation, the choice of algorithm will have negligible effects on the total execution time of most queries.

The centralized computation will also be more vulnerable to message delays since the duration of each phase is determined by the longest round trip. Thus, if a message is lost, the start of backward log processing will be delayed at all kernels in the centralized computation. In the distributed computation only the query processing at the recipient of the message will be affected. Similarly, if a kernel has to wait long before the WAIT set is empty, this will affect the start of query

\(^{12}\text{Note that the time used by the query controller to compute the AFTER set, } t_{\text{AFTER}}, \text{ are not included in Figure 9.15.}\)
9.6 Performance Analysis

![Figure 9.14](image1.png)

**Figure 9.14** Number of extra messages used to compute a transaction-consistent AFTER set.

![Figure 9.15](image2.png)

**Figure 9.15** Average time used to compute a transaction-consistent AFTER set.
processing at all kernels only in the centralized computation. In addition, in the distributed computation it is possible to reduce the time spent waiting by requesting the transaction controller to include an uncertain transaction in the \textsc{Wait} set.

\section{Advantages of the Centralized Computation}

The performance analysis of the two algorithms shows that the distributed computation has the best performance with respect to all three performance metrics. However, the centralized approach still has some advantages:

\begin{itemize}
\item A more modular implementation since changes to the 2PC-protocol is not required. The centralized algorithm could be implemented without modifying the modules concerned with transaction processing. All information needed by the algorithm can be found by processing the log.
\item Less work required by transactions. The distributed algorithm requires that the transactions piggyback query IDs and update the \textsc{After} sets based on piggybacked query IDs.
\item Less message volume by transactions. In a highly utilized network, better performance of transactions could be achieve if messages belonging to queries were given lower priority. Hence, by piggybacking the IDs of active queries, not all the message volume generated by query processing could be given low priority.
\end{itemize}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure.png}
\caption{Extra message volume created in order to compute a transaction-consistent \textsc{After} set.}
\end{figure}


- If basic 2PC is used, a query will not have to wait for uncertain transactions to terminate.

- Supports weak consistency for the read only optimization of the 2PC protocol [Sam+95]. The distributed computation will only give update consistency.

- Does not depend on rigorous transaction execution [Bre+91]. By determining dependencies using the last update LSN and uncertain LSN as described in Section 9.5, non-rigorous transaction execution can be supported. For timestamp ordering, the timestamps of the transactions could be used directly to determine possible dependencies. (It would appear that serialization graph testing (SGT) could also be well suited since dependencies are explicitly represented in the serialization graph. However, distributed SGT is not very efficient [Ber+87].)

The centralized computation of AFTER sets has been used for the implementation of on-line parallel backup in the ClustRa DBMS. The main reason for choosing this algorithm over the distributed algorithm was that it did not require any changes to the 2PC protocol. The ClustRa DBMS supports on-line software upgrades, and changes to existing messages should be avoided in order to support message communication between nodes running different versions of the database software.

When implementing backup in ClustRa, the algorithm was divided into two parts. The first part is executed during backup when data and log is dumped to disk. The second part is executed during restore in order to get a transaction-consistent database. During backup, the the three first rounds of the message protocol shown in Figure 9.3 will be executed when all data has been flushed to disk. When a kernel receives any of the three messages, it will create a special log record. The first and the third log record will mark the beginning and end of the part of the log that is needed for computing the AFTER set, while the second log record will represent the equivalent of the starting point. During restore a DT-LOG message will be created for each node by scanning the log. These messages are sent to the restore controller which will compute the global AFTER set. This AFTER set is then used to determine which log records should be redone during restore.

### 9.8 Related Work

The technique of piggybacking information on the messages of the 2PC protocol has been suggested in several works. As described in Section 3.3.7, variants of this technique have been used in distributed algorithms for transient versioning.

In the algorithm by Chan and Gray, the list of currently committed transactions (CTL) is assigned to a query at startup [ChGr85]. A query will only read versions created by transactions in the CTL. In order to ensure that no transactions in the CTL is dependent on a transaction that is not in the CTL, the 2PC messages is used to inform all participating kernels about which transactions have committed before this particular transaction. This is achieved by letting the kernels piggyback on the YES message all transactions that have committed since the previous message was sent to the same transaction controller. The transaction controller will compute the union of all committed transactions and piggyback this list on the COMMIT
messages. Each kernel will add the transactions in this list to its CTL in addition to the committing transaction.

This algorithms does not require that queries wait for the termination of uncertain transactions. The cost is that not all transactions that committed before the start of a query will necessarily be seen by the query. However, this situation can be avoided by an extra coordination phase where the query coordinator collects the current CTL from all participating kernels, and assigns the union of these CTLs to the query. The algorithm by Chan and Gray requires that piggybacking is performed at all times in order to be able to assign a consistent CTL when a new query arrives. When using the distributed algorithm presented in this thesis, piggybacking need only be performed during query startup. In addition, the increase in message volume will be larger for the algorithm by Chan and Gray since it is transactions that are piggybacked and not queries. In most systems, the number of transactions will be much larger than the number of queries.

Bober proposes to use the YES message to inform the transaction controller about local AFTER set insertions involving the update transaction being committed [Boe93]. Similarly to the algorithm presented in this thesis, the transaction controller will propagate these insertions to all kernels participating in this transaction. Bober's algorithm requires that each time a query wants to read a data item that was last updated by a transaction that is currently uncertain, it must wait until the transaction terminates in order to know which version to read. Hence, a query may have to wait for uncertain transactions several times. In the algorithm presented in this thesis, a query will only have to wait for uncertain transactions once. Bober's algorithm also requires that piggybacking is performed during the entire lifetime of the query.

The transient versioning algorithm implemented in the Oracle DBMS may be seen as a variant of the algorithm by Chan and Gray where a so called system change number (SCN) is piggybacked instead of the CTL [HaBa95]. A SCN is assigned to each transaction and represents the serialization order of the transaction. Each kernel will piggyback on the YES message a number that is higher than any earlier assigned SCN, and the transaction controller will choose the greatest of the received numbers as the SCN of the transaction and piggyback this number on the COMMIT messages. Also this algorithm requires that queries block behind uncertain transactions.

Garcia-Molina et al. uses the piggybacking technique in the single-mark epoch algorithm for group commit at backup nodes of a l-safe replicated database system [Gar+90]. Transactions are divided into so called epochs and are applied to backup nodes when the backup nodes is informed that a new epoch has started. One node is designated as epoch master and will periodically start a new epoch by logging the end of the previous epoch and broadcasting END-EPOCH messages to all other primary nodes. When the primary nodes receive such a message, they will log the end of the previous epoch. In order to synchronize the start of a new epoch, the current epoch number will be piggybacked to all YES and COMMIT messages. If a transaction controller or a kernel receives a message with a higher epoch number than its current epoch, it will act as if it had received a END-EPOCH message. This is similar to how the start of a query is synchronized in the algorithm presented in this thesis.

When backup nodes receives log from primary nodes, log records are not pro-
cessed immediately. When a log record indicating the end of an epoch is received, it notifies a backup master and waits for this master to broadcast that all backup nodes have received the end-epoch log record. Then, the log is scanned to find which transaction committed during this epoch. The log records for these transactions will be applied to the backup database. If a transaction was uncertain at the primary node when the epoch ended, the backup node will ask the transaction controller whether the commit decision was made before the epoch ended. If yes, the log records of this transaction will be applied to the database. Otherwise, the transaction is left pending until the end of the next epoch.

A similar piggybacking technique is also used in the distributed checkpointing algorithm proposed by Kim and Park [KiPa94]. Via the YES message, a kernel will inform the transaction controller whether checkpointing is currently in progress. If a transaction becomes uncertain after the start of checkpointing at at least one kernel, the transaction controller will tell all participating kernels via the COMMIT messages that this transaction should not be reflected in the checkpoint. If a kernel receives this information when no checkpointing is in progress, checkpointing will be started.

While several variants of the algorithm for distributed computation of the AFTER set have been presented in the literature, no algorithm similar to the algorithm for centralized computation is known to the author.

9.9 Concluding Remarks

In this chapter two algorithms for compensation-based query processing in an distributed database system with shared-nothing architecture have been presented and compared. Both algorithms achieve a transaction-consistent result by synchronizing the computation of the AFTER set used for backward log processing. This way, each kernel can execute the query using the algorithm for a centralized database system presented in Chapter 4.

The performance analysis of the two algorithms show that the distributed computation of the AFTER performs better than the centralized computation both with respect to the total number and size of messages and the time needed to to perform the computation. Especially, the execution time is much lower for the distributed computation since fewer rounds of message passing between the query controller and the participants are needed to establish a transaction-consistent AFTER set. Also, the execution time for the distributed computation scales better with increasing number of nodes in the system.

Even if the distributed computation gives best performance on all performance metrics, there are still a few arguments for choosing the algorithm for centralized computation of the AFTER set. Most important, its implementation does not require any changes to existing parts of a system. All information about concurrent transaction needed by the algorithm could be found by reading the transaction log. This also means that the performance of concurrent transactions will be less affected by the centralized computation of the AFTER set.

The performance analysis of the execution time for the two algorithms also show that, regardless of the algorithm chosen, the time needed to compute the AFTER set will be insignificant compared to the total execution time of most queries.
Distributed Algorithms
Part IV

Conclusions and Future Work

This last part concludes the thesis and indicates some directions for future work.

Chapter 10 The major conclusions of the thesis are summarized. In addition, a short list containing the major contributions of the thesis is given.

Chapter 11 Directions for future work is indicated.
Chapter 10

Conclusions

10.1 Summary

This thesis has presented several novel algorithms aimed at preventing data contention for concurrent query and transaction processing. In order to avoid that queries delay transactions, queries will not set any locks on the data items they read. However, query results will still be transaction-consistent since queries will compensate for concurrent updates.

Compared to transient versioning algorithms and an earlier compensation-based approach, the algorithms presented in this thesis impose less work on concurrent transactions. The transient versioning approaches require transactions to copy the current version of a data item before updating it, while the earlier algorithm for compensation-based query processing require that transactions add information on their updates to the data structures of concurrent queries. The approach used in this thesis is to let queries obtain information about concurrent updates by processing the database internal log.

This thesis has mainly focused on an algorithm based on undo/no-redo compensation. That is, any update operations of concurrent transactions observed by the query will be undone. A major advantage of this approach is that a two-phased approach is avoided. That is, after reading a tuple, a query may perform the compensation and emit the resulting tuple at once. Other compensation-based approaches require that the entire scan of the base relation(s) are finished before compensation is performed. Pipelining scanning and compensation, also makes it possible to pipeline the scan with relational algebra operations, and compensation-based algorithms for the most used algebra operations have been presented in this thesis. Another advantage of undo/no-redo compensation is that queries will not have to wait for transactions to terminate. Hence, queries will be less vulnerable to long-lived transactions.

The performance of the undo/no-redo algorithm has been evaluated both through analytical modeling and simulations. The evaluation shows that query response times are significantly improved compared to earlier compensation-based algorithms. Compared to non-consistent query processing, only a small overhead is required to achieve transaction-consistency. The simulation experiments also showed that, by giving higher scheduling priority to transaction than to queries, compensation-based
query processing can be performed without significantly affecting the response times of transactions. The performance evaluation also showed that the space requirements for the update-table used to store information about concurrent updates may exceed the available main memory for longer queries in systems with very high transaction workload. However, this thesis has also presented methods that makes it possible to store the update-table on disk.

Another algorithm for undo/no-redo compensation has also been presented. This algorithm reduces the amount of compensation by establishing a transaction-consistent state that does not necessarily reflect a specific point in time. Based on dependency information maintained in the form of colors on data items and transactions, only transactions that are dependent on a query will be rolled back. One promising property of this algorithm is that it eliminates the need for an update-table. Log records will instead be accessed on demand when reading tuples that have been updated by dependent transactions. This strategy will be best suited for DBMSs with large log buffers so that only a few log records will have to be fetched from disk. However, it is also possible to combine this algorithm with sequential log processing. The reduced amount of compensation will then lead to a smaller update-table than in the original algorithm for undo/no-redo compensation.

This thesis has also presented two different distributed algorithms for undo/no-redo compensation. Both algorithms synchronize the start of a query so that queries processing at each node may proceed as in the centralized case. The synchronization is performed by requiring that the same set of transactions are reflected in a query at all nodes. The two algorithms differ in whether the computation of this set is centralized to a single node or distributed among all nodes. A simple performance analysis showed that the distributed computation is more efficient and scalable. However, it requires changes in the implementation of the 2PC protocol of the DBMS. Hence, the centralized computation will provide a more modular implementation. The centralized computation will also be applicable to other concurrency control methods than strict 2PL.

10.2 Main Contributions

The main contributions of this thesis are:

1. A classification of algorithms for compensation-based query processing.
2. A method for using the database internal log to find information about updates by concurrent transactions.
3. A novel query processing algorithm based on undo/no-redo compensation which is more efficient than earlier methods.
5. Two distributed algorithms for compensation-based query processing.
6. The first, to the author's knowledge, performance evaluation of compensation-based query processing.
7. An analytical performance model for undo/no-redo compensation.
Chapter 11

Directions for Future Work

While this thesis has presented the first performance evaluation of compensation-based query processing, further performance experiments are needed in order to fully understand the performance aspects of the algorithms presented in this thesis. This will be necessary in order to make the correct design decisions when implementing the algorithms in a DBMS.

The presented simulation experiments should be extended to compare compensation-based query processing to different algorithms for transient versioning. This way, information could be obtained as to which transaction and query workloads are best suited for compensation-based query processing and which for transient versioning.

The simulation experiments should also be extended in order to investigate several other aspects of compensation-based query processing:

- How the execution of several concurrent queries will affect transaction processing.
- Query and transaction performance in a main-memory DBMS.
- Disk-bounded query processing.
- The use of partial tuple logging. Effects on query response times, and improvements in transaction performance due to reduced log volume.
- Secondary index scans. How will increased usage of the priority queue for delete entries affect query performance?
- How difference log buffer sizes will affect BLP performance, and the possibility of FLP starvation.
- Performance evaluation of the algorithms for selection, aggregation, and join.
- Performance comparison of input and output filtering. (The cost of predicate evaluation versus the cost of compensation.)

In order to be able to evaluate some of the above aspects through simulations, more measurements of an implemented DBMS will be needed.
Further studies are also needed in order to evaluate how much query performance is reduced by the proposed methods for storing the update-table on disk. Especially, simulation experiments are needed in order to determine how the modifications to recursive linear hashing will affect its performance. In that context, different buffer management strategies should be explored. The alternative methods discussed in Section 4.4.3 should also be further explored.

The undo/redo algorithm based on maintaining coloring information, is well suited for implementation in the ClustRa DBMS ([Ha95, Br96]) where log records are not stored on disk but in replicated in main-memory on sites with independent failure modes. Since log records are stored in main-memory, relying entirely on backward log processing should not pose a performance problem. Generally, studies are needed as to how much the amount of compensation is typically reduced by this algorithm. It should also be investigated how large this reduction must be in order to entirely rely on backward log processing without reducing query performance. Further investigations are also needed to determine the most efficient method for representing the grey color vectors of tuples.

The distributed algorithm based on centralized computation of an AFTER set has already been implemented in the on-line backup utility of the ClustRa DBMS. By implementing the algorithm based on distributed computation in the same system, the performance of these to algorithms can be compared. Important aspects to investigate are the scalability of the two algorithms and how they affect the performance of concurrent transactions.

Modifications to the presented algorithms are probably needed in order to adopt them to other system architectures than the one assumed in this thesis. Many of the commercial DBMSs today support the object-relational model, and modifications may be needed in order to support the ways queries access tuples in such systems. Modifications of the algorithms are also needed for implementation in systems based on physiological logging [GrRe93]. Especially, the use of relocation markers for data items that have been moved to another block will have to be handled.

The query optimizer of a DBMS will have to be modified to also consider compensation-based query processing. Especially, it must determine when to use 2PL and when to use compensation for read-only transactions. For very long transactions, the optimizer may consider making a transaction-consistent snapshot to use by the transactions. The optimal use of indexes may also be changed by compensation-based query processing since the space optimizations of the update-table are not applicable to secondary index scans when using partial tuple logging. The optimizer will also need to decide whether to do selection and aggregation before or after compensation (input or output filtering), and whether to switch from partial to complete tuple logging for the execution of a query. A simplified version of the analytical model presented in this thesis could be used when for query optimizations.

In order to support concurrent transaction and query processing, a DBMS must also handle resource contention between transactions and queries. This involves process scheduling, the allocation different types of main-memory buffers, the distribution of data on disks, and disk access scheduling. This thesis has assumed that transaction are always given higher priority than queries, but support for a more general scheme that let users indicate their priorities should be considered.
Bibliography


198

Bibliography


Bibliography


Bibliography


Bibliography


Bibliography


Bibliography
Appendix A

System Model for the Database Simulator

This chapter describes the system model for the simulator used in this thesis to evaluate the performance of compensation-based query model. The model was implemented in the C++-based CSIM18 simulation language [Schw96, Mes]. Below the components of the system model for the database simulator are described. The many parameters of the system model are presented in Tables A.1 and A.2. The values used for the CPU cost is based on measurements done on the ClustRa DBMS [Hva95], and is presented in Table A.4.

The system model encapsulates the logical and physical resources of a DBMS and its underlying operating system and hardware. The system model consists of a single CPU, a single disk manager which administers several disks, a buffer manager, a checkpoint manager, a lock manager, and a log manager. In addition, the system will have several transaction managers and query managers, each executing a single transaction or query, respectively, at the time.

A.1 CPU module

The CPU module models the behavior of the CPU scheduler. The scheduler is priority-based and non-preemptive. The CPU scheduler will assign the CPU to the requesting thread with the highest priority. In case of ties in priorities, the scheduler uses a first-come, first-served (FCFS) policy. Only two different priorities are used in the simulator.

It is assumed that the DBMS is run as a collection of light weight threads in a single process. The threads executing the queries are given the lowest priority while all other threads are given the highest priority. The CPU cost of the CPU scheduler is modeled by charging SchedCPU instructions each time a thread releases the CPU.

A.2 Disk Manager

The disk manager will receive requests from the buffer manager to transfer a certain number of blocks starting with a given block ID between the disk and the database.
Table A.1 Parameters for the system model.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumDisks</td>
<td>Number of disks</td>
</tr>
<tr>
<td>DiskBlockSize</td>
<td>Disk block size</td>
</tr>
<tr>
<td>DiskTrackSize</td>
<td>Number of disk blocks per track</td>
</tr>
<tr>
<td>DiskCylSize</td>
<td>Number of tracks per cylinder</td>
</tr>
<tr>
<td>DiskControllerOverh</td>
<td>Disk controller overhead per request</td>
</tr>
<tr>
<td>DiskSpeedupFactor</td>
<td>Factor relating seek time to short seek distances</td>
</tr>
<tr>
<td>DiskSpeedupBound</td>
<td>Upper limit for short seek distances</td>
</tr>
<tr>
<td>DiskCoastBase</td>
<td>Basis for computing seek times for long seek distances</td>
</tr>
<tr>
<td>DiskCoastFactor</td>
<td>Factor relating seek time to long seek distances</td>
</tr>
<tr>
<td>DiskReadSettle</td>
<td>Settle time for read operations</td>
</tr>
<tr>
<td>DiskWriteSettle</td>
<td>Settle time for write operations</td>
</tr>
<tr>
<td>DiskMaxLatency</td>
<td>Maximum rotational delay</td>
</tr>
<tr>
<td>DiskTransfer</td>
<td>Disk transfer rate</td>
</tr>
<tr>
<td>DiskTrackSwitchRead</td>
<td>Time to reposition when switching track during read operations</td>
</tr>
<tr>
<td>DiskTrackSwitchWrite</td>
<td>Time to reposition when switching track during write operations</td>
</tr>
<tr>
<td>BufSize</td>
<td>Number of pages in the database buffer</td>
</tr>
<tr>
<td>MaxSeqLO</td>
<td>Maximum number of pages in a sequential read/write</td>
</tr>
<tr>
<td>KeysFP</td>
<td>Number of index keys which fit in one index page</td>
</tr>
<tr>
<td>TransMaxTime</td>
<td>Maximum execution time for a transaction before it is aborted</td>
</tr>
<tr>
<td>LogBufSize</td>
<td>Number of log pages which are kept in memory</td>
</tr>
<tr>
<td>LogTermSize</td>
<td>Size of commit and abort log records</td>
</tr>
<tr>
<td>TmpBufSize</td>
<td>Size of the temporary buffer</td>
</tr>
</tbody>
</table>

A system will have NumDisks disks, and based on the block ID, the disk
manager will direct each request to the corresponding disk. It is assumed that
asynchronous I/O is used, and that all disks are used as raw devices. In other
words, no buffering is done by the operating system. To schedule an asynchronous
disk request costs DiskXferCPU instructions, while DiskPollCPU instructions is
used to poll for finished disk requests.

Disk requests are scheduled using a FCFS policy. The disk parameters given in
Table A.3 is based on the data sheet for the Seagate Cheetah 4LP disks and on
measurements presented in [Wor+95]. Each disk block is of DiskBlockSize and there
is DiskTrackSize blocks in each track and DiskCylSize tracks per cylinder.

The service time for a disk request is computed as the sum of the seek time, settle
time, rotational latency, and transfer time in addition to a disk controller overhead
(DiskControllerOverh). The seek time is dependent on the seek distance \(d\), the number of
cylinders between the cylinders of the previous and the current request. The seek time
is composed of a speedup time and a coast time as described in [RuWi94] and
computed from the following formula:

\[
\text{DiskSeekTime}(d) = \begin{cases} 
\text{DiskSpeedupFactor} \cdot \sqrt{d} & \text{if } d < \text{DiskSpeedupBound} , \\
\text{DiskCoastBase} + \text{DiskCoastFactor} \cdot d & \text{if } d \geq \text{DiskSpeedupBound} ,
\end{cases}
\]

(A.1)

where \(d\) is the seek disk distance.

The settle time is represented by the the parameters DiskReadSettle and Disk-
WriteSettle for read and write requests, respectively. The rotational latency is chosen
Table A.2 CPU cost parameters for the system model.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SchdCPU</td>
<td>Overhead for thread switch</td>
</tr>
<tr>
<td>DiskXferCPU</td>
<td>Cost to set up an asynchronous disk request</td>
</tr>
<tr>
<td>DiskPollCPU</td>
<td>Cost to poll for a finished disk request</td>
</tr>
<tr>
<td>BufCPU</td>
<td>Cost to find a page in the database buffer</td>
</tr>
<tr>
<td>BufMissCPU</td>
<td>Cost to handle a miss in the database buffer</td>
</tr>
<tr>
<td>BtreeSearchFac</td>
<td>Factor for the cost of searching an index page</td>
</tr>
<tr>
<td>CkptStartCPU</td>
<td>Cost to scan the database buffer during checkpoint (per page)</td>
</tr>
<tr>
<td>CkptCpyCPU</td>
<td>Cost to copy a page</td>
</tr>
<tr>
<td>LockReqCPU</td>
<td>Cost to request a lock</td>
</tr>
<tr>
<td>LockRelCPU</td>
<td>Cost to release a lock</td>
</tr>
<tr>
<td>LogCrtRecCPU</td>
<td>Cost to create a log record</td>
</tr>
<tr>
<td>LogCrtCLRCPU</td>
<td>Cost to create a compensation log record</td>
</tr>
<tr>
<td>LogCrtTermCPU</td>
<td>Cost to create a commit or abort log record</td>
</tr>
<tr>
<td>LogCrtTransCPU</td>
<td>Cost to create entry in checkpoint log record (per active trans.)</td>
</tr>
<tr>
<td>TransInitCPU</td>
<td>Cost to initiate a transaction</td>
</tr>
<tr>
<td>TransCommitCPU</td>
<td>Cost to commit a transaction</td>
</tr>
<tr>
<td>TransAbortCPU</td>
<td>Cost to abort a transaction</td>
</tr>
<tr>
<td>ReadCPU</td>
<td>Cost to execute a read operation</td>
</tr>
<tr>
<td>UpdateCPU</td>
<td>Cost to execute a write operation</td>
</tr>
<tr>
<td>DeleteCPU</td>
<td>Cost to execute a delete operation</td>
</tr>
<tr>
<td>InsertCPU</td>
<td>Cost to execute an insert operation</td>
</tr>
<tr>
<td>TopProcCPU</td>
<td>Cost for the scan thread to process a single tuple</td>
</tr>
<tr>
<td>TopCpyCPU</td>
<td>Cost for the scan thread to emit a single tuple</td>
</tr>
<tr>
<td>LogProcCPU</td>
<td>Cost for the log processing thread to process a log record</td>
</tr>
<tr>
<td>UpTblInsCPU</td>
<td>Cost to insert an entry in the update-table</td>
</tr>
<tr>
<td>UpTblOutCPU</td>
<td>Cost to do a lookup in the update-table</td>
</tr>
<tr>
<td>PriQInFac</td>
<td>Factor for the cost to insert an entry in the priority queue</td>
</tr>
<tr>
<td>PriQOutFac</td>
<td>Factor for the cost to extract an entry from the priority queue</td>
</tr>
<tr>
<td>AfterFac</td>
<td>Factor for the cost to find next log record from the AFTER set</td>
</tr>
</tbody>
</table>

uniformly over the range 0 to DiskMaxLatency. The transfer rate is given by the parameter DiskTransfer with the addition of DiskTrackSwitchRead or DiskTrackSwitchWrite for each time during a read or write operation, respectively, a track boundary is crossed.

A.3 Buffer Manager

The buffer manager manages a database buffer consisting of BufSize pages using an LRU replacement policy. The database buffer is shared among data pages and index pages. The pages in the database buffer can be accessed either through the buffer hash table based on its block ID, or given the relation and the primary key of a particular tuple, the page holding the tuple can be found using a B-tree. The CPU cost of finding a page in the hash table is BufCPU instructions. If the page is not present in the buffer, BufMissCPU instructions are used to insert it into the buffer.

Associated with each table in the database is a clustered B-tree index. Each index page can have a maximum of KeysPP keys. For simplicity, it is assumed that the B-tree has a fixed size, and that index pages are never updated. The CPU cost
of searching an index page is $\ln(\text{KeysPP}) \cdot \text{BtreeSearchFac}$ instructions.

After a transaction has requested a page in the buffer, that particular page is guaranteed to reside in the buffer until the transaction has performed the next operation on a tuple on this page. This is implemented by associating a counter with each buffer page. The counter is incremented each time the page is requested and decremented each time an operation is performed on it. Only pages with a counter equal to zero will be replaced by the buffer manager.

The buffer manager supports requests to read and write up to $\text{MaxSeqIO}$ pages in a single disk request. To avoid that pages from large sequential scans fill the entire buffer, pages read by sequential scans are inserted at the front of the LRU list. Thus, these pages will be replaced at once the above mentioned counter has been decremented.

For each relation it will be defined how many tuples fit in one page. The system does not represent the actual data of each tuple, but for each tuple a state identifier

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Setting</th>
<th>Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPURate</td>
<td>300 MIPS</td>
<td>DiskWriteSettle</td>
<td>2.08 msec</td>
</tr>
<tr>
<td>DiskBlockSize</td>
<td>4 kBytes</td>
<td>DiskMaxLatency</td>
<td>6.0 msec</td>
</tr>
<tr>
<td>DiskTracks</td>
<td>21 tracks</td>
<td>DiskTransfer</td>
<td>13.7 MBytes/sec</td>
</tr>
<tr>
<td>DiskCoilSize</td>
<td>0.7 msec</td>
<td>DiskTrackSwitchRead</td>
<td>0.78 msec</td>
</tr>
<tr>
<td>DiskControlOver</td>
<td>0.12 msec</td>
<td>DiskTrackSwitchWrite</td>
<td>1.04 msec</td>
</tr>
<tr>
<td>DiskSpeedupFactor</td>
<td>0.74</td>
<td>BufSize</td>
<td>4096 pages</td>
</tr>
<tr>
<td>DiskSpeedupBound</td>
<td>784</td>
<td>MaxSeqIO</td>
<td>32 pages</td>
</tr>
<tr>
<td>DiskCoilBasel</td>
<td>1.677 mcecs</td>
<td>KeysPP</td>
<td>150</td>
</tr>
<tr>
<td>DiskCoilFactor</td>
<td>0.002 mcecs</td>
<td>LogBufSize</td>
<td>40 pages</td>
</tr>
<tr>
<td>DiskReadSettle</td>
<td>1.56 mcecs</td>
<td>LogTermSize</td>
<td>20 Bytes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Setting</th>
<th>Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>SchedCPU</td>
<td>150 instructions</td>
<td>TransInitCPU</td>
<td>350000 instructions</td>
</tr>
<tr>
<td>DiskXferCPU</td>
<td>50000 instructions</td>
<td>TransCommitCPU</td>
<td>400000 instructions</td>
</tr>
<tr>
<td>DiskPollCPU</td>
<td>1500 instructions</td>
<td>TransAbortCPU</td>
<td>400000 instructions</td>
</tr>
<tr>
<td>BuffCPU</td>
<td>1500 instructions</td>
<td>ReadCPU</td>
<td>80000 instructions</td>
</tr>
<tr>
<td>BuffMissCPU</td>
<td>6000 instructions</td>
<td>UpdateCPU</td>
<td>150000 instructions</td>
</tr>
<tr>
<td>BtreeSearchFac</td>
<td>1000 instructions</td>
<td>DeleteCPU</td>
<td>60000 instructions</td>
</tr>
<tr>
<td>CkptScanCPU</td>
<td>15 instructions</td>
<td>InsertCPU</td>
<td>120000 instructions</td>
</tr>
<tr>
<td>CkptSortFac</td>
<td>250 instructions</td>
<td>TupProcCPU</td>
<td>80000 instructions</td>
</tr>
<tr>
<td>BuffCopyCPU</td>
<td>160000 instructions</td>
<td>TupCommitCPU</td>
<td>75000 instructions</td>
</tr>
<tr>
<td>LockReqCPU</td>
<td>2000 instructions</td>
<td>LogProcCPU</td>
<td>14000 instructions</td>
</tr>
<tr>
<td>LockRelCPU</td>
<td>300 instructions</td>
<td>UpTabInCPU</td>
<td>75000 instructions</td>
</tr>
<tr>
<td>LogCrtRecCPU</td>
<td>7000 instructions</td>
<td>UpTabOutCPU</td>
<td>10000 instructions</td>
</tr>
<tr>
<td>LogCrtCLRCPU</td>
<td>7000 instructions</td>
<td>PriQInFac</td>
<td>25 instructions</td>
</tr>
<tr>
<td>LogCrtTermCPU</td>
<td>3500 instructions</td>
<td>PriQOutFac</td>
<td>215 instructions</td>
</tr>
<tr>
<td>LogCkptTransCPU</td>
<td>20 instructions</td>
<td>AfterFac</td>
<td>10 instructions</td>
</tr>
</tbody>
</table>
A.4 Checkpoint Manager

All flushing of dirty pages in the database buffer to disk will normally be done by the checkpoint manager. The checkpoint manager is implemented as a separate thread which is activated when the number of dirty pages gets high. More specifically, if first page in the LRU is dirty the buffer manager triggers the checkpoint manager. In other words, a new checkpoint is started when all pages that were part of the previous checkpoint has been either accessed or swapped out.

The checkpoint manager scans all pages in the database buffer and all presently dirty pages are included in the checkpoint. Before writing the pages to disk it must first write the checkpoint log record and all log records of operations to the pages included in the checkpoint to disk. It then sorts the pages on block ID in order to achieve sequential access to the disks. All disks are written to in parallel.

In order avoid that the checkpoint manager occupies the CPU for too long, the checkpoint manager will release the CPU after it has inspected or sorted 10 pages, and then immediately request the CPU. The CPU cost of inspecting the buffer is \(\text{CkptScanCPU}\) per page, while the cost of sorting the pages in physical order is \(\log(n) \cdot \text{CkptSortFac}\) per page, where \(n\) is the number of pages to be sorted.

In order for the checkpoint manager to avoid flushing the log to disk more than once, the buffer pages will be locked for updates from the time they are included in the checkpoint until they have been written to disk. However, a copy-on-write mechanism is used in order to avoid that update transactions are blocked during the checkpoint. Transactions that want to update a locked page will copy the contents of the page to a new buffer page and update this new version. The CPU cost to copy a page is \(\text{BufCopyCPU}\) instructions. If a page has been copied during a checkpoint, the old version of the buffer page could be replaced by another page when the write lock is released.

A.5 Lock Manager

The lock manager implements tuple-level locking only. A lock request costs \(\text{LockReqCPU}\) instructions while the release of a lock costs \(\text{LockRelCPU}\) instructions. Read and write locks are automatically set when a transaction performs a read or write operation, respectively. Read locks are upgraded if a transactions updates or deletes a tuple it has previously read. No deadlock detection is implemented. Deadlocks are resolved by the transaction managers which abort transactions that have not terminated within \(\text{TransMarTime}\) seconds.

A.6 Log Manager

The log manager maintains a buffer of the \(\text{LogBufSize}\) most recent log pages. For each update performed by a transaction, the log manager inserts log records into
the current log page. Since full tuple logging is assumed, each log record will occupy twice as much space as the tuple that was updated. In addition, a commit or abort log record is created when a transaction terminates. Such records is assumed to occupy $\log\text{TermSize}$ of a page. During checkpoint a checkpoint log record is created. Such a record is assumed to occupy $(t + 1) \cdot \log\text{TermSize}$ of a page where $t$ is the number of active transactions at the start of the checkpoint. When a transaction is aborted, compensation log records (CLR) is created for all operations that are undone. A CLR is assumed to be the same size as the log record it is compensating for. When a log page is full, a new log page is started, removing the oldest log page from the buffer. If a log record that is no longer residing in the log buffer is needed either during undo processing or during backward log processing, the corresponding log page will be read from disk.

To create a log record for an update operation is assumed to cost $\log\text{CrecRecCPU}$ instructions, while creating a CLR is assumed to cost $\log\text{CrecCLRCPu}$ instructions. The cost of creating an abort or commit log record is $\log\text{CrecTermCPU}$, while the cost of creating a checkpoint log record is $\log\text{Crec TermCPU} + t \cdot \log\text{CkptTransCPU}$, where $t$ is the number of active transactions.

Log records are flushed to disk either upon request by the checkpoint manager or when transactions are committed. With each request to flush log records, a LSN is specified which gives the newest log record that needs to be flushed.

### A.7 Transaction Manager

The number of transaction managers in the system is given by the multiprogramming level (MPL) of the system. Each transaction manager executes a single transaction at a time. Once its current transaction has terminated, the transaction manager immediately starts a new transaction. If the execution time of a transaction exceeds a time limit of $\text{TransMaxTime}$ seconds, it is aborted since it has probably become part of a deadlock.

The transaction manager models the reception of a request from a client to run a transaction as the CPU cost of $\text{TransInitCPU}$ instructions. The execution of the transaction consists of a sequence of operations. Each operation represents a read, update, insertion, or deletion of a single tuple identified by relation and primary key. For each operation, the transaction first requests a lock on the tuple of the appropriate type. It then asks the buffer manager to return a pointer to the page in which the tuple is located. The buffer manager will find this page by navigating through the B-tree associated with the relation. When the requested page resides in the database buffer, the transaction will, if it is not a read operation, first log the operation before performing it. The CPU costs of the performing a operation is $\text{ReadCPU}$, $\text{UpdateCPU}$, $\text{DeleteCPU}$, and $\text{InsertCPU}$ for read, update, delete, and insert operations, respectively.

When all operations of a transaction have been executed, the transaction is ready to commit. In order to reduce the work associated with flushing the necessary log records to disk, a group commit policy is used. A separate group commit thread will, when starting a group commit, commit all transactions that are ready to commit at that point in time. For each transaction that will be committed a commit log record is created. All these log records is then flushed to disk before the locks of the transaction is released. The CPU cost of committing a transaction is $\text{TransCommitCPU}$
A.8 Query Manager

which also is supposed to include the CPU cost of sending a reply to the client.

As long as there are transactions that are ready to commit, the group commit thread will start a new group commit as soon as it has completed the previous one. This is similar to the method used in the Oracle DBMS [Ora95]. If no transaction is waiting to commit, the group commit thread will wait until the next transaction has finished its operations before starting a new commit.

Since non-preemptive CPU scheduling is used, a transaction will be granted the CPU until it decides to release it. A transaction will always release the CPU after requesting a lock and after finishing each operation. In addition, the CPU is released if a requested page does not reside in the database buffer.

A.8 Query Manager

The query manager run a single query at the time. When one query is finished a new query is immediately started. The execution of a query involves four separate threads. In addition to the scan thread and the log processing thread, a query will have a read and a write thread. The read thread reads batches of MarSeqIO pages at the time from the disk. When the scan thread starts processing such a batch, the read thread will request a new batch from the buffer manager. The query result is temporarily stored in a temporary buffer of TmpBufSize. The write thread writes batches of MarSeqIO pages from the temporary buffer to disk.

The scan thread uses TupProcCPU instructions to process each tuple and TupEmitCPU instructions if the tuple is emitted to the temporary table. For each deleted tuple that is extracted from the priority queue, log(n) · PriOutFac instructions is charged while UpTabOutCPU instructions is used each time the update-table is checked for entries.

The log processing thread uses LogProcCPU instructions to process one log record. If the log record belongs to the base relation of the query, the update-table is checked using UpTabOutCPU instructions. The CPU cost of inserting a new entry in the update-table is UpTabInCPU instructions. Inserting pointers to delete entries in the priority queue costs log(n) · PriInFac instructions, where n is the size of the priority queue.

The AFTER set used by backward log processing is implemented as a list of LSNs, one for each member of the AFTER set. The LSNs represent the newest log record not yet processed for each transaction. When a log record is processed, the LSN of the previous log record of the transaction log chain is substituted for the LSN of the current log record. When doing backward log processing, the highest LSN is selected from the AFTER set, the corresponding log record is processed, and a previous LSN of the transaction is inserted in the AFTER set. For each log record that is processed, an extra n · AfterFac instructions, where n is the size of the AFTER set, is charged for each lookup in the AFTER set.

System Model for the Database Simulator
Appendix B

Low-Cost Compensation-Based Query Processing

This appendix contains a paper to be presented at the 24th International Conference on Very Large Databases (VLDB) in New York, August, 1998 [Gro98].
Low-Cost Compensation-Based Query Processing