NTNU

Innovation and Creativity

Non-blocking Creation and Maintenance of Materialized Views

Øystein Aalstad Jonasson

Master of Science in Computer Science

Submission date: June 2006

Supervisor: Svein-Olaf Hvasshovd, IDI

Co-supervisor: Jørgen Løland, IDI

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

Problem Description

Materialized Views have grown to become a staple of modern databases. When a materialized view is created, the referenced parts of the database will be blocked and made unavailable for other transactions untill creation is completed. For highly available systems, such a downtime is intolerable. It is therefore important to develop techniques for creating materialized views in a non-blocking fashion.

Assignment given: 20. January 2006
Supervisor: Svein-Olaf Hvasshovd, IDI

PREFACE

As computers become a staple of everyday life, so does the need to store and retrieve data. As a result, the use of databases for both personal and commercial use has become more widespread.

To evolve the data stored in these databases into information and knowledge, it has to be structured and analyzed. Views are used to store queries on the stored data, and these can be materialized by storing the results to reduce access time. Materializing a view is trivial if other transactions can be blocked from altering the data while the materialized view is created.

Since some systems need to be available at all times, the downtime required to create new materialized views is unacceptable. Some method for performing this without blocking other transactions is clearly needed.

This thesis has been submitted to the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU) as the master thesis for the "Sivilingeniør" (MSc) degree. The work has been carried out for Svein-Olaf Hvasshovd of the Database System Group.

Acknowledgements

Many people have helped me while working on this thesis. I'd like to give special thanks to Jørgen Løland, who has been invaluable for inspiration and guidance through my work.

I'd also like to thank my friends and family for their invaluable support, especially Tor-Erling Bjørstad and Per Christian Henden for proofreading and help in the mysterious world of LaTeX.

ABSTRACT

Materialized views have grown to become a staple of modern databases, as they speed up the processing of complex queries by storing the results. When a materialized view is created, the referenced parts of the database will be blocked and made unavailable for other transactions until creation is completed. For highly available systems, such a downtime is intolerable. It is therefore important to develop techniques for creating materialized views in a non-blocking fashion.

The approach used in this document is to create an assisting table as a bridge between the base table and view, on which only projection of non-identifying attributes is performed. This makes it possibly to distinguish individual records to enable log recovery methods to keep the assisting table consistent with the base table. The assisting table can then be used to create and maintain the view without blocking the base table.

Contents

Contents iii

Figures v

I Introduction vii

1 Introduction 1

2 Concepts 3
- 2.1 Databases . . . 3
- 2.2 Consistency . . . 4

3 State of the Art survey 7
- 3.1 Views . . . 7
- 3.2 Transformations . . . 9

II Method 13

4 Abstract Method 15
- 4.1 Creating the Assisting Table and View . . . 15
- 4.2 Projection and Selection . . . 18
- 4.3 Grouping and Aggregates . . . 19

5 Creation 21
- 5.1 Fuzzy-copy . . . 21
- 5.2 Revised Fuzzy-copy . . . 23
- 5.3 Triggers . . . 24

6 Maintenance 27
- 6.1 Propagation . . . 27
- 6.2 Immediate . . . 30
- 6.3 Deferred . . . 30

III Conclusion **33**

7 Conclusion **35**
7.1 Discussion 35
7.2 ACID 36
7.3 Further Work 36

Bibliography **40**

List of Figures

2.1 Two-phase locking. 4

3.1 Blakeley's consistency example. 8

4.1 The table Employee. 15
4.2 A query projecting the attributes Department and Salary. 16
4.3 An illustration of how data flows through the system and where operations are performed. 18

6.1 The contents for the query "SELECT SUM(SALARY) FROM EMPLOYEE GROUP BY DEPARTMENT". 28
6.2 The contents after employee 3 has been moved from Sales to Management. . . 28
6.3 The contents after a new employee has been inserted. 29
6.4 The contents after employee 3 has been moved from Sales to Management. . . 30

Part I

Introduction

Chapter 1
Introduction

Databases are normally modeled to describe the existing domain to solve the needs that are perceived to exists then. The real world is however not as neat and static as database designers would wish, and both needs and domains change over time. New types of products require different kinds of information, laws change what can and can not be stored, the organization itself can be reorganized, and not least, bad designs happen.

Once it is clear that the current database schema is unfit for its task, it should be changed. Changing the database schema is trivial if the system can be brought down for the duration of the change: Deny any new transactions and let the current transactions complete. When no transactions are running it's possible to either alter the old tables, or where that is not feasible, create new tables and use the old tables to populate them.

This solution works well for systems that can be taken down for the time needed to perform the change. There are however some systems where such a downtime is unacceptable, such as financially sensitive systems, automated measurement systems and medical facilities [GS91, Cri91, Clu] Automated measurement systems typically reads a measuring instrument with fixed time intervals, and writes this raw data to a database. Flow measurement in pipes is an example of this, where individual readings are of little interest. Data is therefore aggregated over intervals to even out spikes in the data source. On-line ordering and banking systems are expected by their customers to be available at all times. Financial systems also use aggregates to help analysts get a clearer picture of the data they contain.

Over the last years, a lot of work has gone into making the inevitable changes to the database schemes as seamlessly as possible, by allowing changes to happen without stopping ongoing transactions [Løl04, Ron00]. Views have not been getting the same focus, which can result in problems when the base table a view is based on is changed in a schema transformation, or a badly designed view is the reason the change is needed.

Data warehouses are a special kind of database that combines the contents of several other databases in a homogeneous way and performs groupings and aggregates to make it easier to analyze. Data warehouses are expensive to create and maintain though, and for analysis within the same database materialized views perform much the same role.

The approach taken in this document to the problem, has been to take already existing non-blocking methods used in database schema transformation and adapt these for use on

materialized views. These methods need to uniquely identify records, which is not possible in normal materialized views. We therefore introduce a bridge between the base table and the MV to which records can be uniquely identified from the logs and base table. This bridge can then identify suitable records in the MV to apply the changes to. It does not need to uniquely identify the records, for reasons which will be clear in chapter 4.

Materialized views are representations of queries, which consist of relational operations. These include amongst others, join, union, difference, projection, selection and grouping. Aggregate functions are possible to perform on groupings. The focus of this document is the relational operations that take only a single table as input, namely selection, projection and grouping/aggregates.

As a foundation for this work, basic concepts are described in chapter 2 while chapter 3 gives a survey of the state-of-the-art. An abstract method for how this non-blocking creation works is given in chapter 4, and some different ways to implement this method is presented in chapter 5. Chapter 6 describes how such a view can be maintained and how changes are propagated from the AT to the MV itself.

Chapter 2
Concepts

This chapter will describe some of the standard concepts in databases. Chapter 2.1 explains the basics of database systems, while chapter 2.2 describes how consistency is maintained.

2.1 Databases

Databases are persistent storages for information. While other storages such as filesystems focus on storing information about entities, databases are designed to also store the relations between entities. Databases are also designed to support multiple users accessing information at the same time, and to interleave so that the actions of different users do not conflict. Access to a database goes through the "Database Management System" (DBMS). The description of a database is called a schema, which describes amongst other things which attributes are stored in tables, what views exists and access rights for users.

The basic storage type of a database is the table. A table consists of a finite set of related records (also called tuples or rows). The contents of a view however, are not stored physically. Instead, a view is defined as the result of a query, and is thus related to the tables. Materialized Views (MVs) are a special type of view, in which the contents are stored. The view is still based on a query, but for a MV the results are stored. If the results are kept consistent with the base table, then the query need not be executed when the view is accessed. Views and MVs are described in further detail in 3.1.

Tables operate with set semantics, which means that duplicate records are not allowed. Views on the other hand, operate with bag semantics, which means that duplicates are allowed. It is possible to force views to operate with set semantics instead.

All operations against the database is organized into transactions. A transaction is a partially ordered sequence of read and write operations [PABG87]. For most databases, such as the ones this document deals with, all transactions must conform to the "ACID" [PABG87, GR93, HGM02] properties, as described below:

A *Atomicity:* "(...) the all-or-nothing execution of transactions." [HGM02] Either all of the components of the transaction are executed and applied to the database, or no changes are made at all.

C *Consistency:* Each transaction must always transform the database from one consistent state to another.

I *Isolation:* “(...) the fact that each transaction must appear to be executed as if no other transaction is executing at the same time.” [HGM02] Transactions should not observe the effects of transactions that have yet to commit.

D *Durability* “(...) the condition that the effect on the database of a transaction must never be lost, once the transaction has completed.” [HGM02]

2.2 CONSISTENCY

There are two main ways of guaranteeing consistency. By preventing conflicts by locking records, and the optimistic approach which resolves conflicts that occurs. The optimistic approach allows any transaction access to any resources, and checks for conflicts at commit time and resolves those conflicts that occurs [J. 81]. Locking is the preferred method for normal use. Whenever a transaction requests a resource, wether it is a record, parts of or an entire table[1], that resource is locked. If another transaction requests the same resource, it is denied. This ensures that only one transaction at a time can modify that resource.

Gray et al. [I. 76] defines four degrees of consistency, from 0 to 3. The degrees differ in how much a transaction allows other transactions to see of the data it accesses during its execution. This is accomplished by locking the accessed records in different ways for different lengths of time. Degree 3 provides complete protection by completely isolating potentially conflicting transactions, while degree 0 only assures that only one transaction is writing a record at any given time. Degree 3 uses 2-phase-locking (2PL) to provide a completely consistent transaction model.

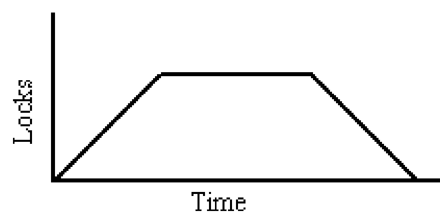

Figure 2.1: Two-phase locking.

Figure 2.1 shows how a transaction using 2PL acquires locks. The transactions sets locks on all the records it access during its execution, and holds those locks until it commits. A

[1] Gray et al. [I. 76] describes various ways to lock resources

transaction can release locks early, but once a lock is released that transaction can not take any other locks, and must commit. If it does abort, any transactions that used records that were locked and released also have to be aborted. This ensures that data written by the transaction is not visible to other transactions until it commits, and data read by the transaction cannot be changed by others until after the transactions is committed.

One way to accomplish durability is to write the effects of all transactions to a separate log. This log can then be used to restore the database to a consistent state in which all committed transactions are reflected. Each entry in the log (called a log record) is usually given a unique number, called Log Sequence Number (LSN), in a monotonically ascending order. By writing the LSN of the last log record that modified a data block, it implies that all log records with a lower LSN are included in that block.

CHAPTER 3
STATE OF THE ART SURVEY

Chapter 3.1 will describe some of the research which has gone into views, especially ways to maintain materialized views.

As a part of highly available systems, a lot of research has also been performed on ways to avoid downtime for database systems [GS91, Cri91, Clu]. This includes both voluntary (such as maintenance and upgrades) and involuntary (such as power outages or natural disasters) downtime. Chapter 3.2 will describe some of the research which has gone into the research on changing the database schema without taking the system down for mmaintenance.

3.1 VIEWS

A views is one class of relation in the SQL schema, but unlike Tables the contents are not stored physically [HGM02]. Instead views are defined as a stored SQL query which references tables in the database schema. When the View is queried, the query it stores is executed and the results from it are passed on as the contents of the view.

For complex queries, the execution can take considerable time. To speed up the response time for queries, materialized views (MV) were created. Materialized Views store the results for the query instead of calculating it each time. MVs are the result of work started in 1980 by Abida and Lindsay [Bru80] on snapshots.

Snapshots store parts of the database from an earlier point in time. This version is itself internally consistent, but outdated, which means relations to other, current, parts of the database may not be consistent. Snapshots started to be used to replicate data in distributed systems, and to answer frequently used queries without executing the query [Bru80]. This only works if the system is allowed to work on data that is out of date. Snapshots were refreshed either by a direct command from the user, or periodically. Abida and Lindsay [Bru80] presents two methods for refreshing the snapshot: completely reevaluating the query, or detecting which changes are not reflected in the snapshot and then apply them. Both of these will be described further.

As the effects of snapshots became appreciated, the concept was expanded upon and eventually evolved into materialized views. Materialized views are divided into two types, depending on when its contents are updated: immediate and deferred updates. These will

be described further. Unlike its predecessor, MVs are not allowed to be outdated when they are queried. Deferred updated MVs are temporarily out of date after the base table has been changed, but must be brought up to date before they can be used.

Base Table

A	B
1	10
2	10
3	20

MV

B
10
20

Figure 3.1: Blakeley's consistency example.

Blakely et al. [BLT86] presents a problem that occurs when the bag semantics of views is reduced to sets, as illustrated in figure 3.1: The MV is defined as the set of all Bs in the base table. If a transaction deletes (1,10) from the base table, the correct action for keeping the MV consistent is deleting (10) from the MV. However, if a transaction deletes (2,20) from the base table, the record (20) should not be deleted from the MV, since another record has a B value of 20. Based on the work of Shmueli et al. [Alo84], Blakeley et al. suggest using a counter to keep track of how many times that entry would be present in the view if it was using bag semantics. This method is later expanded by Gupta et al. [GKM92, Gea93] to work on other relational operators, and presents a new algorithm for use one recursive MVs, Called "Delete and Re-derive". This method is not discussed further, as it is of little interest for this document.

Datawarehouses are a special type of database which is closely related to snapshots and materialized views [Rou97]. A datawarehouse is a copy of the contents of up to several different databases, in which it contents have been standardized and organized to answer OLAP queries. It's contents are out of date, and is typically refreshed from the live databases every night.

Regardless of when a MV is refreshed, the same two methods Abida and Lindsay presented to refresh snapshots [Bru80] are used. The first method, rebuilding the entire view, has not received much attention, since it is expensive to use [Gea93, TG95]. As an illustration of just how expensive, the creation method presented in chapter 5 can be considered one implementation of this method.

The second method, detecting which changes have occurred since the last refresh, and applying these, have gotten much more attention. Both Qian and Wiederhold [QW91] and Griffin et al. [TG95, GLT97] use algebra for computing immediate updates.

Qian and Wiederhold "present an efficient algorithm for the incremental re-computation of active relational expressions based on finite differencing techniques. (...) the algorithm derives, by update propagation, the minimal incremental relational expressions that need re-computation." [QW91] They present an algorithm that create both an insert set and a delete set from updates on a Select-Project-Join (SPJ) view. These sets, when applied to the MV, will bring the MV up to date. Griffin et al. [TG95] expands on this to include bag semantics, and in a later article [GLT97] proves the original algorithm is not minimal and presents an improved algorithm.

Colby et al. [CGL+96] identifies the problem they call "the state bug", which occurs when immediate methods are used for deferred updates without modification. The authors extend the work of Qian et al. [QW91] to overcome this problem. This algorithm imposes very little penalty on ordinary transactions, as the only work needed is to write log records. Their method uses the log and two differential tables (one for updates and one for deletes) to transform the MV from a state consistent with a previous state to a more recent state.

Since not all updates to the base tables will affect the MV, it's important for good performance to detect which updates can be ignored by the view maintenance [BCL89]. An update which does not alter the MV is called irrelevant. Shmueli and Itai [Alo84] present a solution to detect irrelevant updates by use of a special tree datastructure. This tree contains which entries in the MV are derived records in the table. When a given record is updated, any entries in the MV which are derived from it must be recalculated. The paper makes no guarantees that the number of propagated updates are minimal.

Blakeley et al. [BLT86] presents another method for detecting irregular updates to a SPJ view by evaluating updates against the defining query. This filters out updates that are outside the scope of the query, and therefore can't possible affect it. Not all updates that doesn't modify the MV are filtered out this way though. Another article by Blakeley et al. [BCL89] examines the problem in more detail, this time using algebra instead of a algorithm based approach.

3.2 TRANSFORMATIONS

The work on schema transformations can be broken into two main methods. Ronström's method for soft transformation which uses triggers, and Løland's method for equijoin transformation which is based on fuzzy-copy.

3.2.1 Trigger based transformation

Ronström [Ron00] presents how a complex schema transformation can be performed as a soft schema change. Ronström defines a soft schema change as one in which old transactions continue executing on the old table while new transactions are started on the new schema.

The general method for the transformation is to create the new table description, and set up a number of triggers that ensures actions on one table is propagated to the other. The triggers also ensure that the proper transformation takes place. When the triggers are set up, the transformation process scans the old table and copies all records to the new table, without setting any locks. Because any updates to the tuples that happens during the scan are propagated by the triggers, this will result in a copy that is consistent with the original. When the copying is complete, the system waits until all transaction that were running on the old table have completed by either committing or aborting. At this point, all transactions are running on the new table, so the triggers and the old table can be removed.

3.2.2 Fuzzy Copy

Fuzzy copy is a technique for generating a copy of a table that will be eventually consistent with the original, without setting blocking locks. This is accomplished by making an inconsistent copy without locking any records, and using log records to propagate the effects of transactions, making the copy consistent.

At the start of the copy, a checkpoint is created in the log. This forces all buffers to be written to disk, and writes a checkpoint to the log which includes the ID of all transactions that are currently running. This reduces the amount of work which is needed to redo from a log. All operations before the checkpoint are included in the buffers that were written, so these records do not need to be redone. However, any transactions that were running when the checkpoint was taken might have to be undone, so the records before the checkpoint are still interesting for undo work. Checkpoints are explained in many books, Molina et al. [HGM02] being one of these.

Hvasshovd et al. [Odd91] present a mechanism for create a non-blocking image in the HypRa database. A checkpoint is created and marked as "begin-fuzzy-production" and the interesting parts of the table are copied without setting any locks. Only latches are used to prevent conflicts. When the copy is completed, an "end-fuzzy-production" mark is written to the log. By applying the log records between the two marks, redoing all transactions that commit and undoing the others, the copy will be made consistent with the original table at the time of the second checkpoint.

3.2.3 Equijoin

Løland [Løl04] presents how fuzzy-copy can be used to perform equijoin transformation on database schemes. It creates a new table, performs the equijoin into this table and then switches control from the old table to the new.

A fuzzy checkpoint is written to the log at the start, after the new table is created. The involved tables are then scanned and the equijoin operation is performed with the result inserted into the new table. When the scan is completed, a new checkpoint is written and the log between the checkpoints is processed and applied to the new table. If one of the join attributes is changed, the algorithm looks in the original tables to find a new, if any, join.

During this log scan, new log records are most likely created. If a lot of updates have occurred, a new checkpoint is created and the log is again applied between these two points. This is repeated until few enough log records are created, and the algorithm for switching transactions over to the new table is performed. A pause in transaction activity is enforced while the final log scan is performed and transactions are redirected to the new table.

Part II

Method

CHAPTER 4
ABSTRACT METHOD

This chapter will explain an abstract method for creating and maintaining materialized views in a non-blocking way. This method focuses on projection and selection based views, as well as views which perform grouping and aggregation. Joins are not considered, but the method can be extended to include these operations. Chapter 4.1 describes how the AT and MV van be created. Chapter 4.2 describes how projection and selection is performed with this method and chapter 4.3 how grouping and aggregation is performed.

Chapter 5 will detail some ways to implement the non-blocking creation of materialized views, while chapter 6 will detail ways to maintain the view.

Figure 4.1 shows a table that will be used as a recurring example.

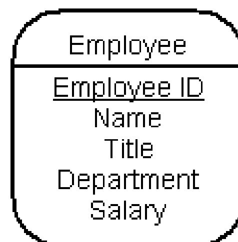

Figure 4.1: The table Employee.

4.1 CREATING THE ASSISTING TABLE AND VIEW

Creation of a materialized view normally involves a scan of the base tables which the view referenced. This scan collects the information necessary for building the view, and to avoid inconsistencies this scan needs to be blocking.

One way to make the creation non-blocking, is to use existing non-blocking methods to make a consistent copy of the base table. This copy wil be used as an assisting table (AT) and can be used to run the blocking MV creation on, without blocking the base table. The AT itself can be seen as a MV itself, on which only projection (without removing idenitifying attributes) has been performed.

In order to create a copy, individual records have to be distinguishable, as well as different versions of the same record. Identifying attributes can be projected away from the MV, and are not enough to distinguish between different versions of the same record. To be able to tell if a stored record reflects the changes of an operation, Log Sequence Number (LSN) can be used. If the LSN stored with a record is the same or higher then the LSN given to the operation, then that operation is reflected in the record. The version of a record with the highest LSN is the most recent.

By storing both LSN and all of the records identifying attributes, it is possible to distinguish uniquely between records and different versions of the same record. This allows us to map both log records and records from the base table into the AT. Records from the AT can not be uniquely mapped into the MV, but that is not necessary. As long as the AT is treated as the source of all that is in the MV, then records that are indistinguishable in the MV can be considered to be equal seen from the AT. This means that records in the AT can be traced back to log records and base table, but records in the MV can not be traced anywhere.

The AT needs not be a complete copy, since vertical (projection) and horizontal (selection) cuts from the view declaration can be performed now. However, identifying attributes can not be stripped away, since they are needed to uniquely identify records in the AT. This will be discussed further in chapter 4.2.

Let's consider a query that projects the attributes Department and Salary (as shown in figure 4.2, using bag semantics.

EmployeeID	Name	Title	Department	Salary
1	Alice	Manager	Sales	35 000
2	Bob	Clerk	Sales	18 000
3	Charles	Clerk	Sales	18 000

a) Contents of Base Table

Department	Salary
Sales	35 000
Sales	18 000
Sales	18 000

b) Contents of MV

Figure 4.2: A query projecting the attributes Department and Salary.

During creation, it's important that records can be distinguished, so that the same person does not appear twice in the AT. As long as only those that are entered into the AT are entered into the MV, they need not be distinguishable in the MV. If there are two persons with the same salary in the same department, and one of them changes department, then that person needs to be removed from the AT. However, since the records in the MV are indistinguishable, either of them can be removed. If EmployeeID 2 is removed from the table in figure 4.2a), either of the lower two records in 4.2b) can be removed to keep the MV consistent.

After the view is created from the AT, the AT and view needs to be refreshed with the updates from transactions that have committed during the scan of the AT. Exactly how this is done varies with the implementation.

The process can be divided into two separate parts; the algorithm for creating the AT, and the algorithm which propagates changes from the AT to the MV. These are split into two separate algorithms since the creation algorithm can then be an existing method implemented without much modification. The propagation algorithm can be reused in maintenance of the view.

The method for creating the AT will be responsible for making changes to the AT in such a way that when it is complete, the AT is a consistent copy of the relevant parts. On each change to the AT, it will call the propagation algorithm with the details of the change, in the order described below.

The creation algorithm can quickly be described as:

1. Find a record.
2. Perform selection.
3. Complete set of attributes and LSN is passed to Update Algorithm.
4. Perform projection, but keep identifying attributes.
5. Resulting set of attributes and LSN is inserted into AT.

The propagation algorithm will be called each time either the creation or maintenance algorithm changes the AT. As each record is inserted into the AT, it is first checked if it is an relevant update [BLT86, BCL89]. Chapter 4.2 details this process further. If the record is irrelevant for the view, it is discarded. Relevant records are first applied to the view, as described below, before it is applied to the AT. All records in the AT store all the identifying attributes, all attributes used by the view and the latest Log Sequence Number (LSN) to affect that record. This ensures that all updates up to and including the LSN stored are reflected in both the view and the AT, for that given record. Thus, if the stored LSN for all records in the AT is the latest LSN for that record, the view and AT are both up to date. No guarantees are given if the stored LSN is lower and there are no current updates being processed on the AT. This data can be used to limit the work needed to recreate the MV after a database crash.

The propagation algorithm can quickly be described as:

1. record is checked for relevance, and discarded if irrelevant.
2. Copy algorithm reads old values from AT.
3. If old LSN is same as new, update is discarded.

4. MV is modified as based off new value from Copy Algorithm and Old Value from AT.

During maintenance of the view, the selected maintenance algorithm is responsible for propagating changes on the base table to the AT, and executing the Propagation Algorithm. Maintenance will be discussed fighter in chapter 6.

Figure 4.3 shows how data flows through the system, and how the different operations are performed at different stages. Data from the log and base table are projected and form the contents of the AT. Attribute based selection is then performed on the data before it is grouped and aggregated and stored in the MV. This is discussed in chapter 6.1. Finally, selection based on aggregated values is performed on the contents of the MV.

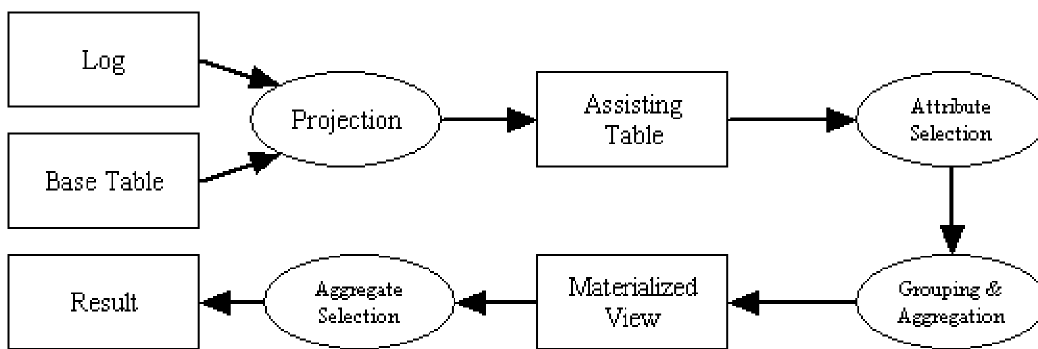

Figure 4.3: An illustration of how data flows through the system and where operations are performed.

If grouping or projection removes the identifying attributes, a hidden count will be used to keep track of how many entities in the base table that entry represents. This is needed to maintain consistency when set semantics makes it records indistinguishable, and is an implementation of the counting method presented in chapter 3.1

4.2 Projection and Selection

When the AT is created, the attributes that needs to be stored is calculated based on the projection of the query. Identifying attributes are kept even if they are projected out, as well as Log Sequence Number (LSN). The resulting set of attributes will be stored as the description of the AT, which will be used as a filter on all other incoming records to the AT.

Selection is slightly more troublesome then projection. There are two distinct and different selections that can be done, selection based on aggregate values and selection based on attribute values.

Selection based on aggregate values can for obvious reasons not be done before the aggregate is calculated. A query which selects all departments with more then 5 employees is an example of this. Therefore all records that are used to calculate aggregates must be taken into account and entered into the AT and MV, and the selection needs to be performed on the contents of the MV.

Selections based on attribute values can be done on the records before they are entered into the AT, which in most cases will reduce the storage space needed for the AT and MV by a significant amount. Updates can make a record that was irrelevant before relevant again, so all updates on values which are selected away needs to be watched closely. Since the old version of the record is not stored in the AT, it needs to be fetched from the base table.

The alternative is to keep all the records in the AT, and do selection when they are entered into the MV. This will reduce the workload when updates make irrelevant records relevant, but it will also lead to more work to be done when searching the AT. Depending on how often updates on selection attributes occur, it is probably less work for the system to do selection before entering records into the AT.

Some queries do selection on both an aggregate value and an attribute in the record, like a query that selects all departments with 5 employees that earn more then a given amount. If all employees that does not fulfill the second selection criteria are not entered into the AT like described above, then the MV can be built on the data in the AT without doing any selection.

4.3 Grouping and Aggregates

How each of the Aggregate functions work are described in detail below. To avoid locking and increase concurrency, incremental operations can be used to modify the stored aggregate values in the MV. These operations are not idempotent, and are thus not safe to redo. Since it is already impossible to distinguish between different versions in the MV, this limit makes no difference from idempotent operations.

Count For each grouping that the count applies to, a counter is maintained. On all inserts into the AT, the counter is raised by 1, and on all deletes from the AT it is lowered by 1. Changes to records are ignored.

Example: A MV stores how many work in each department. The AT will then contain all employees and the attributes for Employee ID and Department, as well as LSN. The MV will contain one entry for each Department, and a counter for how many Employee IDs in the AT have that Department. If an employee changes departments in the AT, the counter for his old department is decreased and the counter for his new department is increased.

Sum For each grouping that the sum applies to, a counter is maintained. On all inserts into the AT, the counter is raised by the corresponding value in the inserted record. On a delete from the AT, the counter is lowered by the corresponding value in the deleted record. When a record in the AT is changed, the value of the counter is changed in a corresponding way.

Average For averages, 2 counters for sum and count are maintained as described above, and average is computed when needed as sum divided by count.

Min and Max When a record with a value lower then the current min or higher then the current max for a grouping is inserted, the old value is overwritten with the value of the inserted record. If a delete or update operation affects a record with the lowest or highest value, a scan of the AT is conducted to determine the new value. However, if the update operation changes it to another value which is lower then the current minimum or higher then the current maximum, the new value can be used without performing a table scan.

CHAPTER 5 CREATION

This chapter will describe two different ways of making a copy of the base table in a non-blocking way. These are responsible for making sure all the records of the base table is inserted into the AT, upon which the algorithm for updating the MV will be activated, as described in chapter 4.

The first method is based on the Basic Log Propagation in Jørgen Løland's Equijoin Schema Transformation [Løl04] which itself is based on fuzzy-copy, and is detailed in chapter 5.1. The other method is based on Mikael Ronström's On-line Schema Update [Ron00], and is detailed in chapter 5.3. Both of these methods only deal in creating the AT, the process for propagating changes from AT to MV will be detailed in chapter 6, since it is related to view-maintenance.

Before the AT can be populated, it needs to be created. The table description for the AT will contain all attributes referenced in the SELECT or GROUP BY clauses of the query, any attributes in the primary key for the referenced table and an attribute for the LSN. After a table with this schema has been created, it has to be populated.

5.1 FUZZY-COPY

The fuzzy copy algorithm can quickly be described as follows:

1 Create checkpoint

2 Redo committed transactions from last checkpoint

3 Create checkpoint

4 Undo between last checkpoints

5 If more log records, repeat from 2

6 Start view maintenance

After the AT has been created, a checkpoint is created. This includes forcing all buffers belonging to the referenced table to disc before a checkpoint marker with the IDs of all running transactions is written in the log.

The first part of populating the AT is to perform a non-blocking table scan of the base table and copying these records to the AT. A table scan reads every single record in the table. This will create a inconsistent copy since no locks are set during the scan. This inconsistent copy will form the base of the new table, and will be made consistent again by the following log scan.

By performing a redo/undo scan of the log since the checkpoint, all operations that were concluded during the scan will be reflected in the AT, making it consistent. The scan starts by finding the checkpoint that was written previously, and starts reading the log records from that point in sequence. Only log records for transactions that commit will be applied to the AT. Each log record is processed as follows:

Insert If a record with the ID specified in the log record already exists in the AT, the log record is ignored. Otherwise a new record is added to the AT with the attributes listed in the log record.

Update If the attribute the update applies to is not present in the AT's description, the log record is irrelevant and is ignored. If the attribute is relevant, the LSN of the log record is checked against the LSN stored in the AT. If the LSN of the log record is higher, that version is newer and the AT is changed to use the value stored in the log.

Delete If a record with the ID specified in the log record exists in the AT, that entry is deleted. If no records in the AT match the ID it has already been deleted.

When there are no more log records to process, another fuzzy checkpoint is created. The log is now scanned in reverse until it encounters the first checkpoint, and all records relating to transactions that were not committed at the time the last will be undone in the AT.

This undo scan needs to be done because locks are not propagated from the base table to the AT. To maintain consistency in the AT only transactions that have been committed can be propagated to the AT. This ensures that other transactions doesn't read dirty[1] data. If only transactions that commit are applied to the AT, the only transactions that need to be undone are those that were running when the copy process started. Unless the log scan takes very little time, or transactions take very long time, it's unlikely that any undo work needs to be done.

During an eventual undo scan, more log records might have been created, necessitating another round of scans. This scan can't start at the last checkpoint however, as records for transactions that were running at the point the checkpoint was created have yet to be applied. The scan must therefore start at the point the first of these transactions start. This process must be repeated until there are no log records left after the undo scan is complete.

[1] Dirty data is data that has been written by a transaction that has yet to commit.

Once the log scans are completed the AT and MV is consistent. Further updates to the MV are now the responsibility of the view maintenance algorithm, which will be described in chapter 6. The AT can be discarded once maintenance is started, as long as a maintenance method that does not rely on the AT is used.

5.2 REVISED FUZZY-COPY

If the view maintenance algorithm is idempotent[2] (using LSN is one way of ensuring the method is idempotent), the propagation of the log can be replaced with starting the maintenance algorithm early. By starting the view maintenance algorithm just before the non-blocking copying, all transactions running during the copying will have their actions properly propagated to the AT.

Idempotency is needed to ensure that changes are applied only once. If a record is changed just as it is copied, the copy algorithm can read the new, altered value and inserts int the AT. Then the maintenance algorithm applies the change to the AT, and if the operation is not idempotent it will be applied twice.

The maintenance and copy algorithms must also handle some special cases:

Update If the record the update applies to is not in the AT, the update is ignored. The only way for the updated record to not be in the AT, is that it has not been copied yet. When it is copied, that update has already been applied to the record, and as such will also be applied to the AT.

Insert Since inserted records will be caught by the maintenance algorithm, the copy algorithm can get an error when it tries to copy a record that already exists in the AT. That record can be safely ignored, as the insert and any subsequent updates will be in the AT, courtesy of the maintenance algorithm.

Delete If the copy algorithm is thorough and copies every single record, then deletion will not be affected. However, if a transaction deletes a record but is later aborted, that record can be missed by the copy algorithm. The maintenance algorithm will catch the undo operation, and use the log to recreate the record.

Incomplete Transactions: However, transactions that were running during the copying and did not commit, has to be undone. This can be done by taking note of which transactions are running when the copying starts and remove those that are committed. Transactions that start during the scan does not need to be undone, since the maintenance algorithm has these under

[2]A function is idempotent if it yields the same result if you apply it to the result

control. In most cases, all transactions that were running during the start of the copying have already committed by the time it finishes. If one of the transactions that was running during the start aborts, the maintenance algorithm must make sure to remove the changes that were applied before the copying started, and not just that have happened while the maintenance algorithm was running.

5.3 TRIGGERS

The trigger based copy algorithm can quickly be described as follows:

1 Create checkpoint

2 Create triggers

3 Run unblocking scan

4 Disallow new transactions and wait until all transactions complete

5 Delete triggers

6 Start view maintenance

7 Allow new transactions

The trigger based method creates triggers on the original table to catch operations on records there. Exactly how the triggers are created varies with database system, but triggers for the following must be created:

Insert: On insert into old table, the trigger inserts the new record into the AT.

Update: On updates to the old table, the trigger checks the AT to see if that record has been copied. If the ID for the changed record exists in the AT, it has been copied, and the update is propagated. Otherwise the update is ignored, since it will be included when the record is copied.

Delete: When a record is deleted from the old table, the trigger checks the AT to see if that record has already been copied. If the ID for the deleted record exists in the AT, the record is deleted from the AT. If it hasn't been copied yet, it will not be copied since it has been deleted from the source table, and thus nothing needs to be done.

After these triggers have been created, a non-blocking scan of the original table is performed to copy all the records over to the AT. The triggers will make sure that changes made to records

that have been copied are properly propagated to the AT. If one of the transactions that change the source table aborts, those changes will be undone on the source table. This will be caught by the triggers and propagated to the AT and MV.

In Ronström's original paper [Ron00], it's the new table that will be in use after the copy process is complete, and not the old table. For the purpose of creating the AT however, transactions will continue to use the old table. This complicates the handover process, which originally is to just let transactions complete. When the copying is done, the effects of transactions that didn't commit needs to be undone in the AT. This can be done by not letting new transactions start while already running transactions complete. When no transactions are running, the triggers are deleted and normal view maintenance is started, before transactions are allowed to start again. If a view maintenance algorithm that doesn't use the AT is used, the AT can also be deleted now.

This hand-over imposes a pause that is depending on how long transactions takes to complete. Since transactions can in some cases take long time to complete, this pause can be intolerable for highly available systems. For most systems however, the pause will be short enough to be tolerable.

CHAPTER 6
MAINTENANCE

Since the AT can technically be considered a MV, existing methods for maintaining it can be used. This chapter will present two maintenance methods that take advantage of the fact that entries in the AT can be uniquely identified, something normal maintenance methods can't assume. Chapter 6.2 will describe a method for immediate updates, while chapter 6.3 describes a method for deferred updates. Chapter 6.1 will detail how changes are propagated from the AT to the MV.

It is also possible to use both normal view maintenance methods and the methods described in this chapter directly on the MV and ignore the AT. Once the initial creation is completed, the AT only serves as a record of which updates have been applied to the MV. This is mainly useful for limiting the work done to recreate the MV in case of a database failure.

6.1 PROPAGATION

When data has been transferred to the AT, the only operation that has been performed on them is projection. Before this data is propagated to the MV, selection based on attribute values needs to be performed, and any grouping and aggregate functions must be performed.

Since database systems already have triggers for reacting to when a table is altered, it is natural to use these to propagate the data from the AT to the MV. By tailoring the triggers to each specific case, they can perform attribute based selection as well as grouping and aggregates while propagating data.

To maintain isolation and consistency, when a read-locked entry in the MV is updated, the trigger forces a write lock and aborts the transactions that held the read lock. This prevents those transactions from using a value that is out of date.

The table Employee from figure 4.1 will be used as an example again to illustrate how changes are propagated from the AT to MV. The MV will be based on the query "SELECT SUM(SALARY) FROM EMPLOYEE GROUP BY DEPARTMENT", as shown in figure 6.1.

Table:

EmployeeID	Name	Title	Department	Salary
1	Alice	Manager	Sales	35 000
2	Bob	Clerk	Sales	18 000
3	Charles	Clerk	Sales	18 000

AT:

EmployeeID	Department	Salary
1	Sales	35 000
2	Sales	18 000
3	Sales	18 000

MV:

Department	Sum(Salary)
Sales	71 000

Figure 6.1: The contents for the query "SELECT SUM(SALARY) FROM EMPLOYEE GROUP BY DEPARTMENT".

Update: The simplest way to handle an update, is to run the insert trigger with the altered values, then fetch the old data from the AT and run the delete trigger on that data. This will ensure that when an update changes the group a record belongs to, it will be removed from it's old group and added to the new. Alternatively it is possible to customize the trigger to only alter the value when grouping is not changed by the update, and alter both new and old aggregate values when group does change.

Table:

EmployeeID	Name	Title	Department	Salary
1	Alice	Manager	Sales	35 000
2	Bob	Clerk	Sales	18 000
3	Charles	Clerk	Management	18 000

AT:

EmployeeID	Department	Salary
1	Sales	35 000
2	Sales	18 000
3	Management	18 000

MV:

Department	Sum(Salary)
Sales	53 000
Management	18 000

Figure 6.2: The contents after employee 3 has been moved from Sales to Management.

Figure 6.2 shows how the contents have changed after a transaction has changed the department of employee 3 from Sales to Management in the base table. The maintenance

algorithm makes sure the same change is applied to the AT, which triggers the update to the MV. The trigger sees that the grouping atttribute's value was changed, and thus subtract the records salary from the old group and adds it to the new.

Insert: When a new record is inserted into the AT, the trigger reads its attribute values and performs selection. If the record does not fit the selection criteria, nothing more is done. Attributes are then used to determine the correct group (if grouping is done in the query), and any aggregate values are modified as described in chapter 4.3.

Table:

EmployeeID	Name	Title	Department	Salary
1	Alice	Manager	Sales	35 000
2	Bob	Clerk	Sales	18 000
3	Charles	Clerk	Management	18 000
4	David	Manager	Management	35 000

AT:

EmployeeID	Department	Salary
1	Sales	35 000
2	Sales	18 000
3	Management	18 000
4	Management	35 000

MV:

Department	Sum(Salary)
Sales	53 000
Management	53 000

Figure 6.3: The contents after a new employee has been inserted.

Figure 6.3 shows how the contents have changed after a transaction has added a new employee as a manager of the management department. The maintenance algorithm updates the AT, and the trigger increases the sum counter for the Management group accordingly.

Delete: When a record is deleted from the AT, the trigger reads the old values from the AT and checks if it would fit the selection criteria. If the record fits, then it was in the MV and needs to be removed.

Figure 6.4 shows how the contents have changed after a transaction has removed all employees in the sales department. The maintenance algorithm ensures the records are deleted from the AT, and the trigger notices the last member of the group is removed and deletes the entire group.

Table:

EmployeeID	Name	Title	Department	Salary
3	Charles	Clerk	Management	35 000
4	David	Manager	Management	18 000

AT:

EmployeeID	Department	Salary
3	Management	18 000
4	Management	18 000

MV:

Department	Sum(Salary)
Management	53

Figure 6.4: The contents after employee 3 has been moved from Sales to Management.

6.2 IMMEDIATE

Immediate updates are triggered when a transaction commits. The responsibility for doing this usually lies with the transaction, but the database system can also assume responsibility. For each running transaction the system can keep a pointer to the first log record for that transaction. When the transaction commits, the system does a redo-scan of the log from that point, applying the changes for that transaction to the AT. The log records for all other transactions are ignored.

By storing a pointer for the first log record, the system knows where the transaction starts. It is possible to avoid the log scan completely by moving more log records to memory. If the system stores a list of all log records for each transaction memory, it only needs to apply these to keep the view updated. The cost of this is more memory usage, but for systems with relatively few transactions that change records, or transactions that are not long lived, it is possible to dedicate enough memory to storing all the records for active transactions.

6.3 DEFERRED

Deferred updates are usually performed when the view is accessed, but maintenance can also be activated by way of an "update view X" command. When triggered, all changes to the base table done by committed transactions since the last update must be applied to the AT.

Writing a checkpoint after the last maintenance was performed makes it possible to continue the maintenance from the correct position. The maintenance can not start at the checkpoint however, but needs to scan back to the start of the first transaction that was running when the checkpoint was written. Since only the effects of committed transactions are transferred to the AT, the log records between transaction start and checkpoint will not have been transferred in the last maintenance.

This scan can be made faster by storing a pointer to the last checkpoint and to the first log record for each transaction, as was mentioned in chapter 6.2.

Once the starting point of the redo-scan has been determined, log records are read sequentially from that point. Records are grouped together by transaction. When a commit record for a transaction is found in the log, the records for that transaction are applied to the AT. If a transaction aborts, or is not committed by the time the scan is complete, those records are discarded. When the scan reaches the end of the log record, a new checkpoint is created to indicate where the next maintenance will start.

1 Find last checkpoint.

2 Find first record for running transactions.

3 Redo from first relevant record.

4 Write new checkpoint.

Similar to immediate maintenance, it is possible to keep all log records since the last update in memory. Depending on how often maintenance is triggered, this can potentially use much more memory than immediate maintenance. To solve this, when the stored log approaches it's defined memory limit, maintenance is performed and the memory is cleared.

Part III

Conclusion

CHAPTER 7
CONCLUSION

Chapter 7.1 discusses which of the methods presented in chapter 5 are most likely to perform well. No definitive conclusions can be drawn however, as no tests are performed due to lack of time to implement the methods. Chapter 7.2 investigates whether the method described conforms to the the ACID principle described in chapter 2. Some suggestions for further work is also presented in chapter 7.3.

7.1 DISCUSSION

This document set out to describe ways to create materialized views in a non-blocking fashion. Avoiding blocks on the database is vital for systems that need to be highly available. An abstract framework for how to accomplish this was presented in chapter 4, which introduces the new concept of an assisting table (AT).

This table can be created using existing non-blocking methods for copying tables, and can be used as the basis to run the blocking scan on. For the abstract method, this AT is only used for the creation of the MV and can be discarded afterwards. Two of the implementations (the trigger-based and the fuzzy-copy) described in chapter 5 use this approach and can discard the AT during hand-over. The revised fuzzy-copy method needs the AT to ensure that the maintenance algorithm doesn't propagate changes that are already reflected in the view. This is accomplished by using the Log Serial Number stored in the AT to enable idempotent propagation. Both of the maintenance algorithms described in chapter 6 can be used in conjunction with the revised fuzzy-copy method.

The new maintenance algorithms described in chapter 6 are very similar to each other. Which is best suited depends very much on the usage pattern of the database. If updates are frequent, and access to the view happens rarely, the deferred update will be more effective as maintenance is needed less often. Similarly, if updates are rare and access to the view is frequent, immediate maintenance will be performed less often.

All three of the methods described in chapter 5 accomplish the primary objective of the document, since neither of them causes the base table to be blocked a considerable amount of time. Both the fuzzy-copy and trigger based methods have some handover issues though. The

fuzzy-copy method will only encounter this problem if a transaction that was running at the start of the copy is still running when the copy completes. Under normal circumstances this is unlikely to occur, and this is not really an issue. The trigger based method needs to block the base table while hand-over is taking place, and is therefore inferior to the two fuzzy-copy based methods.

The improved fuzzy-copy method is not very different from the normal, and if used with the immediate maintenance method described in 6.2 is virtually identical in function, except that the AT can not be discarded at the end of creation. The author believes that this is probably the best method to use.

7.2 ACID

As mentioned in chapter 2, transactions should conform to the ACID principle.

As long as only completed transactions are reflected in the AT, it will not see partial effects of transaction so atomicity and isolation are satisfied. Consistency likewise holds for transactions that only access the MV. Consistency for transactions that access both MV and table is maintained by the trigger forcing a read lock on the modified parts of the MV, aborting any transactions that are using inconsistent data.

Durability is trivially satisfied, as transactions never modify the MV directly. Therefore, the effects are stored permanently by the tables. Therefore the MV can always be recomputed based on the original tables in the event of a database failure. If it is not discard after creation, the AT can be used to reduce the amount of work needed to recreate the MV in event of a database failure. The AT stores the last LSN that is applied to each record in the AT, and this update is guaranteed to be represented in both the AT and the MV. Therefore the MV does not need to be recalculated if the LSN for all records in the AT is the same as the most recent log for every committed transaction.

If any record has any other LSN stored as last update, no guarantees are given about which updates are reflected for that record. Any records with a higher LSN can be applied, so it can't be assumed that these are not reflected in the MV. That record and any parts of the MV it might influence must therefore be recomputed. Since updates might have changed which parts it influence, the log must also be checked to determine which parts needs to be recomputed.

7.3 Further Work

This document leaves room for more work in this area. One angle is altering existing views, which can be accomplished with some adaptation. The methods described here can be used to create a new view, but some sort of mechanism for switching from the old to the new view

needs to be implemented. Related to this is altering the view when the table it is based on is altered.

Implementation and benchmarking of the various methods described is a natural approach to determine how useful the methods described in this document really are. Finally, expanding the work to include other relational operators which take more than one table as input. This should be possible by altering the copy process described in 5 and tailoring the propagation from AT to MV to fit.

Bibliography

[Alo84] Oded Shmueli Alon Itai. Maintenance of views. In *ACM SIGMOD international conference on Management of data*, pages 240–255. ACM Press, 1984.

[BCL89] José A. Blakeley, Neil Coburn, and Per-Ake Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems*, 14(3):369–400, 1989.

[BLT86] José A. Blakeley, Per-Ake Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In *SIGMOD Conference*, pages 61–71, 1986.

[Bru80] Michel E. Abida Bruce G. Lindsay. Database snapshots. In *Proc Proc. 6th Int. Conf. Very Large Databases*, pages 86–91. IEEE N.Y., october 1980.

[CGL+96] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. Algorithms for deferred view maintenance. pages 469–480, 1996.

[Clu] Clustra. Clustra database: Technical overview.

[Cri91] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

[Gea93] Ashish Gupta and et al. Maintaining views incrementally (extended abstract). In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 157–166. ACM Press, 1993.

[GKM92] Ashish Gupta, Dinesh Katiyar, and Inderpal Singh Mumick. Counting solutions to the view maintenance problem. In *Workshop on Deductive Databases, JICSLP*, pages 185–194, 1992.

[GLT97] Timothy Griffin, Leonid Libkin, and Howard Trickey. An improved algorithm for the incremental recomputation of active relational expressions. *Knowledge and Data Engineering*, 9(3):508–511, 1997.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc, 1993.

[GS91] Jim Gray and Daniel P. Siewiorek. High-availability computer systems. *IEEE Computer*, 24(9):39–48, 1991.

[HGM02] Jennifer Widom Hector Garcia-Molina, Jeffrey D. Ullman. *Database Systems, The Complete Book*. Prentice Hall, 2002.

[I. 76] Jim Gray R. A. Lorie G. R. Putzolu I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *IFIP Working Conference on Modelling in Data Base Management Systems*, 1976.

[J. 81] H. T. Kung J. T. Robinson. On optimistic methods of concurrency control. In *ACM Trans. on Database Systems*, pages 213–226, 1981.

[Løl04] Jørgen Løland. Equijoin schema transformations, 2004.

[Odd91] Svein-Olaf Hvasshovd Tore Sæter Øystein Torbjørnsen Petter Moe Oddvar Risnes. A continuously available and highly scalable transaction server: Design experience from the hypra project., 1991.

[PABG87] Vassos Hadzilacos Philip A. Bernstein and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Weslay Publishing Company, first edition, 1987.

[QW91] Xiaolei Qian and Gio Wiederhold. Incremental recomputation of active relational expressions. *Knowledge and Data Engineering*, 3(3):337–341, 1991.

[Ron00] Mikael Ronström. On-line schema update for a telecom database. In *16th International Conference on Data Engineering*, 2000.

[Rou97] Nick Roussopoulos. Materialized views and data warehouses. In *Knowledge Representation Meets Databases*, pages 12.1–12.6, 1997.

[TG95] Leonid Libkin Timothy Griffin. Incremental maintenance of views with duplicates. In *SIGMOD Conf.*, 1995.