Logical Transaction Handling in Databases

KERSTIN ALQUIST AND CAROLINE SCHMIDT

Department of Computer Science
Lund Institute of Technology
Supervisors: Thomas Raneland, Apptus Technologies AB
and Per Holm, LTH

December 4, 2008
Abstract

Today most database systems use physical locking and logging. In this thesis we investigate the feasibility of predicate locking and what conditions that must be fulfilled for such a system to work. We choose to combine the predicate locking system with a predicate log because the logging system needs to have as fine granularity as the locking system.

The reason predicate locking is an interesting subject to study is that theoretically the granularity of predicate locking can be arbitrarily fine and it solves the phantom problem. Another positive property is that the number of locks is proportional to the number of active transactions and not to the size of the database. Further, there exists no paper that investigates the properties of predicate locking thoroughly.

To find out if it is possible to use predicate locking and predicate logging we made test-implementations of both. We also analysed the problem and did literature studies.

The implementations prove that predicate locking and predicate logging is possible to implement. The implementations and analyses also gave information about what requirements a database system needs to fulfil for it to be compatible with predicate locking and predicate logging. We also found some limitations of the system. One of them is that the fine granularity of the locking system can not always be used. This is because it might cost too much to test dependency between locks.
Acknowledgements

We want to thank our supervisor from Apptus Technologies, Thomas Raneland, for all his help and support throughout the work with this thesis. The time he has spent discussing ideas with us has been really valuable and we are grateful for all his opinions about our work.

We would also like to thank our second supervisor, from Lund University, Per Holm for his help with the thesis. He has been really helpful spending a lot of time reading and commenting on our report.
# Contents

1 Introduction ................................................. 1

2 Background ................................................. 3
   2.1 Predicates ............................................. 3
   2.2 The Database .......................................... 5
   2.3 Logging ................................................ 5
   2.4 Terms .................................................. 6

3 Problem Description ......................................... 9
   3.1 Problem Statement ...................................... 9
   3.2 Problem Discussion .................................... 9
      3.2.1 Locking ........................................... 9
      3.2.2 Logging ........................................... 11

4 Investigation ............................................. 13
   4.1 Locking ................................................. 13
      4.1.1 Conflicting Transactions .......................... 13
      4.1.2 To Lock on a Predicate Expression ............... 18
      4.1.3 The Locking Protocol ............................. 19
      4.1.4 Shared and Exclusive Locks ....................... 19
      4.1.5 Granularity of Predicate Locking ................. 20
      4.1.6 Locking on Extensional Predicate Level .......... 20
      4.1.7 Strategies for Testing Independence .............. 21
      4.1.8 Testing Independence of Actions ................. 21
      4.1.9 Precompilation .................................. 28
   4.2 The Group Operator .................................... 30
   4.3 The Optimizer .......................................... 31
   4.4 The System Types ...................................... 34
   4.5 Logging ................................................ 35
      4.5.1 Logical Redo and Undo ......................... 35
      4.5.2 Different Classes of Add and Remove ............ 37
      4.5.3 How and What to Log ......................... 38

5 Conclusions .................................................. 41
   5.1 Result .................................................. 41
      5.1.1 Locking .......................................... 41
      5.1.2 Logging .......................................... 42
   5.2 Discussion ............................................. 42
      5.2.1 Locking .......................................... 42
5.3 Future Work ............................................................... 43
  5.3.1 Locking ............................................................... 43
  5.3.2 Logging ............................................................... 43

Bibliography ............................................................... 45
Chapter 1

Introduction

Data in a database can be accessed by several users simultaneously. If the users’ accesses to the database are not properly controlled the data in the database will become inconsistent. Say for example that there is just one train ticket left to a certain destination and that two users are trying to buy that one ticket. Without concurrency control they may both succeed in buying the ticket, resulting in problems. The concurrency control is necessary because the task to buy a ticket is not one but a sequence of atomic operations. This is the case with almost all tasks manipulating databases. It is impossible to execute a sequence of atomic operations without temporarily putting the database in an inconsistent state. To solve this problem these atomic operations are grouped into units called transactions. A transaction is treated as if it is a single atomic operation. Either all operations in the transaction are executed or none. Thus, if a transaction starts with a consistent database and runs by itself, after its completion the database will, again, be consistent. The same naturally holds for a serial execution of several transactions. However, the more efficient, concurrent execution of several transactions is desired. To keep the database in a consistent state during the concurrent execution a control mechanism is needed. One such common mechanism to achieve concurrency control is locking. There are two types of locking, physical locking and logical locking. Physical locking locks on records, pages, segments, files, etc. One type of logical locking is predicate locking. It is the logical locking of the exact data a client is manipulating during the transaction. The lock is a predicate describing data. For a definition of predicate in this context see section 2.1. By way of example, suppose a client is manipulating data during a transaction about people owning blue bicycles. A predicate lock will lock all owners of blue bicycles and also new owners added after the lock is taken, whereas a physical lock will lock the page/pages (for example) where the present owners are stored.

The first part of this thesis investigates the viability of predicate locking. Predicate locking has been considered only as a beautiful theoretic model not useful in practical implementations [17, p. 284]. The thesis aims to show that predicate locking is indeed practically useful. One part of this will be to make a test-implementation of predicate locking. The database system in which the implementation is done has some limitations as to how data can be accessed. For details see the section 2.2.

A database system can crash and it must be possible to remove the effects of an uncommitted transaction. To handle these events correctly a log of all the actions of the database must be kept. A locking system requires a log with the same granularity as the locking system, see section 3.2. To get the predicate locking system to work properly a log logging predicates will be implemented. This is the second and smaller part of this thesis.
CHAPTER 1. INTRODUCTION

Chapter 2 presents some background. In chapter 3 the problem is defined and in chapter 4 the problem is investigated and it is described how the problem can be solved. In chapter 5 the results and the conclusions are stated.
Chapter 2

Background

2.1 Predicates

To understand what a predicate is one needs the following concepts:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Literal</td>
<td>A value</td>
<td>John</td>
</tr>
<tr>
<td>Proposition</td>
<td>A statement</td>
<td>&quot;John is a musician&quot;</td>
</tr>
</tbody>
</table>

A predicate, for example $P(x) = "x \text{ is a musician}"$, can be seen as a proposition with a variable. A predicate consists of operators and literals or other predicates. The relevant operators in predicate logic are found in Table 2.1.

<table>
<thead>
<tr>
<th>and</th>
<th>$\land$</th>
<th>conjunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>or</td>
<td>$\lor$</td>
<td>disjunction</td>
</tr>
<tr>
<td>not</td>
<td>$\neg$</td>
<td>negation</td>
</tr>
</tbody>
</table>

Table 2.1. Some operators in predicate logic

There are four operators used in the predicates in the database system used in this thesis. They are found in Table 2.2. 'Group'\(^1\) is introduced to be able to express properties for sets. The predicate logic presented in this thesis is first-order logic. For a definition of first-order logic see [1, p. 101-113].

Predicates can normally only evaluate to true or false. When they are used in the database it is possible to get a result containing several values. The result is the values for which the predicate is true, for example $x = 2$ and $x = 4$.

The predicates are split into two categories: extensional predicates and intensional predicates[5]. The values of the extensional predicates are explicitly stored in the database. The intensional predicates are rules which define how different predicates depend on each other.

Examples of the two are:

- Extensional predicate: $P(x) = "x \text{ is a musician}"$
- Intensional predicate: $Q(x) = P(x) \land S(x)$

\(^1\)Group' groups propositions from a predicate. The group can then be used as an argument to some predicates. That is not part of first-order logic.
Operator | Operation | Description
--- | --- | ---
∨ | or | Logical or.
∧ | and | Logical and.
¬ | not | Logical not.
group | group | Not first-order logic. It is needed to be able to express properties for sets of values.

Table 2.2. The operators in the database system used in this thesis

There are predicates which do not depend on the database, that will always give the same answer for the same input and cannot be updated. One example is $T(x) = x > 2$.

An action is a predicate and a command specifying what the predicate should be used for. The commands used are found in Table 2.3.

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>Evaluation of query</td>
<td>$P(2)$?, $P(x)$?</td>
</tr>
<tr>
<td>¬</td>
<td>Removing a proposition</td>
<td>$\neg P(2)$</td>
</tr>
<tr>
<td>+</td>
<td>Adding a proposition</td>
<td>$+P(2)$</td>
</tr>
</tbody>
</table>

Table 2.3. The commands

All kinds of actions are not allowed for all predicates. Table 2.4 shows the actions which are allowed on extensional predicates and Table 2.5 shows the actions which are allowed on all predicates. The $P$ used in the tables is the $P$ from the example above.

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+P(John)$</td>
<td>Redefines $P$ by adding the proposition &quot;John is a musician&quot;.</td>
<td></td>
</tr>
<tr>
<td>$\neg P(John)$</td>
<td>Redefines $P$ by deleting the proposition &quot;John is a musician&quot;.</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.4. Actions allowed for extensional predicates

Executing $+P(John)$ or $\neg P(John)$ will change the values for which $P$ is true and thereby change the answer to the question $P(x)$?. The values for which $P$ is true has been redefined and so has the predicate $P$.

If for example the propositions $P(John)$ and $P(Paul)$ are true, the content in the database is $P(x) = (x = John) \lor (x = Paul)$. The addition of $P(John)$ would look like $P_N(x) = P_O(x) \lor (x = John)$, where $P_O$ is the old revision of $P$ and $P_N$ is the new revision.

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R(1)$?</td>
<td>Answers the question &quot;Is $R$ true for 1?&quot;.</td>
<td></td>
</tr>
<tr>
<td>$R(x)$?</td>
<td>Answers the question &quot;For which $x$s is $R$ true?&quot;. Returns all the values for which $R$ is true, or false if there is no such value.</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.5. Actions allowed for all predicates

One way of representing predicates will in this thesis be through predicate trees which can look like the one in figure 2.1. The tree represents the predicate $((R(x) \lor (x = 2)) \land (x = 2)) \land \neg((T(y) \land \neg(y = 5)) \land (y = 7))$. The operators $\land$ and $\neg$ are combined to $\land\neg$. 
2.2. THE DATABASE

The database system consists of services, functions and views. There can be many instances of the database system running at the same time. Each one is a separate process and uses its own transaction manager. Everything in an instance shares resources.

An instance can have several services. Each service can be said to have its own database. Services consist of functions and views.

The functions are predefined and are called with different parameters. A function is made from a predicate. It answers for which values the predicate is true. So the database can be queried by predicates.

Views are used to update the database when it is online. Predicates can not be added or removed when the database is online. The updates that are allowed are adding and removing propositions to/from the extensional predicates. It is not possible to update propositions. Exactly how the extensional predicates are updated is predefined. Some examples of updates that can be allowed are the following:

- Add/remove one proposition to/from one predicate
- Add/remove two propositions to/from one predicate
- Add/remove many propositions to/from many predicates

This database system follows the closed world assumption, meaning that everything that is not in the database is assumed to be false. Everything that is in the database is assumed, but not said, to be true.

2.3 Logging

To be able to restore the database after a crash and to handle aborts of transactions, a log of the database actions is kept.

There are two types of logging, physical and logical logging. Physical logging is saving the changes of the physical blocks of the disc. For example, how a block looked before and after a change (before and after revisions) can be saved. For more logging methods see [7]. Logical logging is saving the logical changes of the database. It is the actual change, for example an added proposition that is saved, not how this proposition changes the blocks.
CHAPTER 2. BACKGROUND

The log is used to undo and redo actions on the database. Both redo and undo do not have to be used in a recovery algorithm, but it is most common to use both [17, p. 439-443]. They are also used in the algorithm in this thesis. Redo is to redo logged changes of the database and produce a later revision of the database. Redo is used when recovering from a crash. Undo is to undo logged changes of the database and to restore an earlier revision of the database. Undo is used both when recovering from a crash and when transactions abort. During crash handling there are two phases, a redo phase and an undo phase. During the redo phase log records are read and actions redone. Before or in this phase the uncommitted transactions are listed. In the undo phase the actions of these transactions are read from the log and undone. In the resulting database all the committed logged actions have been redone.

Redo and undo will be done on the database without knowledge of if the logged action has actually been executed or not. The knowledge is missing because the log is written before the action is executed. It is done in that order to make sure that undo will be done correctly [17, p. 438-439]. For the redo and the undo to yield the same result no matter if the action has been executed or not the logged expression must be idempotent. An idempotent expression is an expression which yields the same result no matter how many times it is executed. An example of a non-idempotent expression is one that removes the biggest element from a set. Which element that is removed depends on which elements that exist in the set. With different sets different elements will be removed. An idempotent expression is for example one that removes a specific element. The removal of a specific element always removes the same element from the set. The resulting set will be the same no matter how many times the element is removed.

2.4 Terms

This section contains some definitions of terms that are used in the thesis.

The word \textit{changed} is used for when a value of a variable of the database is changed. It is then said that the database is changed.

\textbf{Transaction:} A transaction is a group of operations treated as a single atomic operation. Either all operations in the transaction are executed or none. In this thesis the operations are called actions.

\textbf{Action:} An action is a predicate expression and a command specifying what the predicate should be used for. There are two kinds of actions, updates and queries.

\textbf{Command:} A command defines what a predicate should be used for.

\textbf{Update:} An update changes the content of the database by either adding or removing some propositions.

\textbf{Proposition:} A proposition is a statement.

\textbf{Query:} A query is a question about the content of the database.
2.4. TERMS

**Checkpoint:** A checkpoint log record contains necessary information to restart the system. It specifies which log records that must be read to restore the database to the state it had before a crash.

**DML:** DML is short for data manipulation language.

**Materialized database:** “The materialized database is the state that the DBMS finds at restart after a crash without having applied any log information.” [7]

**Propagation:** The operation who moves previously written data from the temporary memory to the materialized database is called propagation [7].

**Atomic propagation:** Any set of modified pages are propagated as a unit, such that either all or none of the updates becomes part of the materialized database [7].

**Transaction consistent:** The database is transaction consistent if it contains the result of all committed transactions [7].

**Operation consistent:** The database is operation consistent if the result of an operation is completely in the database or not in the database at all [11].
Chapter 3

Problem Description

3.1 Problem Statement

This thesis investigates predicate locking and a transactional logging system that can be used together with a predicate locking system. The investigation is done through implementation and literature studies. The implementations are done in the database system described in section 2.2. The transaction log will log predicates, as the database is queried and updated by predicates. It will be a predicate log. The questions to be answered by the thesis are:

- How can a predicate locking system be implemented?
- Under which circumstances does a predicate locking system perform well and for which kinds of systems are predicate locking suited?
- How can predicates be logged and what are the demands on the system for predicate logging to be feasible?

3.2 Problem Discussion

3.2.1 Locking

Predicate locking has been considered only as a beautiful theoretic model not useful in practical implementations [17, p. 284]. The viability of predicate locking is supported by [16]. In their experiments they found that only a small number of locks must be maintained and that the number of locks is proportional to the number of active transactions and not to the size of the database. The authors of the paper end by recommending a more detailed study to refute or support their speculations. No such study can be found by the authors of this thesis. All there is, is several papers and books [2, 6, 9, 10, 13, 16, 17] mentioning that predicate locking is known not to be useful for such and such reasons referring to other books and articles saying similar things. It seems to be a universally acknowledged truth that has an unknown origin.

A couple of common objections to predicate locking quoted from articles:

1. "The major problem of predicate locking is high execution cost of checking whether two logic expressions conflict. Note that the overhead for checking conflict between two arbitrary logic expressions is known to be NP-complete" [2].
CHAPTER 3. PROBLEM DESCRIPTION

2. It is difficult [10].

3. "In general, it is difficult to discover the predicates" [6, p. 405].

4. "/.../it performs poorly, since it permits a low level of concurrency between concurrent database transactions" [13].

5. Predicate locks are pessimistic in nature. "Pessimism arises because two predicates intersecting in the attribute space but without any tuples in their intersection for the particular database instance will nonetheless prevent two different transactions from simultaneously locking these predicates. This rule protects against phantoms, but reduces the allowable concurrency" [9].

Comments to the above objections:

1. In practice, the standard expression will not be really complicated or involve a really large amount of data. In the case when it is, a defensive decision to lock can be taken without checking the predicates for conflict. Another alternative is to restrict the kinds of expressions allowed by the system. This will reduce the problem with the time complexity.

2. It is unclear exactly what this means. It is therefore hard to comment on this objection. However a supposed difficulty of some sort is not reason enough to not try using and implementing predicate locking.

3. The exact meaning of this is hard to determine - in what way is it hard to discover the predicates? No matter what was meant, it will not present a problem here because the system in which this implementation will be done is based on predicates. Predicates are used to both query and update the database, so the predicates already exist.

4. The level of concurrency depends partly on the granularity of the locks. As predicate locks can be taken with an arbitrarily fine granularity [8] they could potentially give a high level of concurrency.

5. The objection is founded on the assumption that the predicate locking system is designed for a purely relational database. The predicate locking system that will be attempted in this thesis will be for an arbitrary database model, where data can be accessed according to the description in section 2.2. The predicate locks to be implemented correspond to, in relational database terms, locks on fields and not tuples or columns. Using this kind of predicate locks will not give the mentioned problem, even if it is used in a relational database.

There is no paper proving beyond doubt that predicate locking is not useful. In most references made to predicate locking, for example [2, 4, 9, 13, 17], it is supposed that the predicate locking in question is used in a relational database system and the predicates are conditions on attributes. Therefore very little is known about a general predicate locking system that can be used in any kind of database, including in a relational database. It is such a predicate locking system that will be attempted in this thesis.

Almost all the articles like the idea of predicate locking because it solves the phantom problem. The phantom problem is how to lock data that does not exist. Say for example that a transaction asks about the existence of some data. If the data exists it needs to be locked so that no other transaction will delete it before the first transaction has finished. If the data does not exist it still needs to be locked so that no other transaction will add it
3.2. PROBLEM DISCUSSION

before the first transaction finishes. In the second case, data that does not exist has to be
locked. It is this data that is called phantoms [4].

This altogether makes it interesting to study predicate locking.

3.2.2 Logging

The database will be updated and queried by predicates and locks will be taken on predi-
cates. The granularity of a predicate locking system can be arbitrarily fine. The log needs
to have as fine granularity as the locking system.

This fine granularity is needed for aborts of transactions to work correctly. When a
transaction is aborted nothing but the changes the actions of the transaction really made
in the database should be rolled back. Several transactions are allowed to change the same
block when predicate locking is used. When using physical logging, undo will replace whole
blocks with earlier revisions of the block. This will undo actions of other transactions
that have not aborted. Physical logging of blocks will therefore yield incorrect aborts of
transactions.

Logical logging on the other hand can handle arbitrarily fine changes of blocks as it is
the logical change of the database that is logged. Logical logging can therefore be used
together with the predicate locking system.

In the literature logical logging is mentioned as a possibility [7, 11, 12, 14, 18]. What
is meant by logical logging there is either the logging of operators and their arguments [7]
or the logging of operations that can read one or more objects and possibly write multiple
objects [11, 12, 14]. A logical method of logging suggested is the transition logging of DML
statements and parameters [7]. This method is very similar to the logical logging that will
be attempted for this thesis. Instead of DML-statements and parameters, predicates will
be logged.

In [7] it is mentioned that logical transition logging is less expensive than other methods
during normal processing and more expensive for recovery operations. On the other hand
[18] says that physical logging has longer recovery time than logical logging since physical
logging produces a large amount of data. For logical logging to work, atomic propagation
is said to be necessary [7] [15]. That is because logical redo and undo need to start from a
consistent state.

Some other positive things said about logical logging:

• ".../logical logging, which means that not everything that was changed on a page
  needs to be logged explicitly, thereby saving log space." [14]

• "Being able to perform logical undos allows higher levels of concurrency/.../" [14]

• "Logical log operations can greatly reduce the amount of data written to the log, and
  hence reduce the normal execution cost of providing recovery." [11]

No article found says that logical logging does not work or has anything really major to
say against it.
Chapter 4

Investigation

4.1 Locking

A locking system will be constructed to solve the concurrency control problem. Transactions which conflict should not be allowed to execute in parallel. Two transactions conflict when actions of the transactions are dependent. Predicates are used to query and update the database. So in order to develop a predicate locking system, one must know when two predicate expressions are dependent, what kind of lock that must be taken, how the transactions are allowed to lock, etc. This theory is presented in this section.

4.1.1 Conflicting Transactions

A transaction consists of a sequence of actions. There are two types of actions, queries and updates. An action is a predicate expression and a command specifying what the predicate should be used for.

Definition 4.1.1. If $e_a$ is the predicate expression defining action $a$ and $db$ the database then $e_a[db]$ is the result of evaluating $e_a$ in $db$.

Note that $e_a[db]$ is not a traditional function. To evaluate a predicate expression in a database is to take the definitions of all the predicates on which the predicate expression depends from the database. $e_a$ is first simplified by applying the rules from the database, then the extensional predicates on which $e_a$ depends are substituted by their definitions from the database.

Example 4.1.1. $e_a = P(x) \land (x = 2)$, In the database: $P(2)$

\[
x = 2 \Rightarrow e_a = P(2) \land (2 = 2)
\]

From the database: $P(2) = true$ \Rightarrow $e_a[db] = true \land true = true$

Definition 4.1.2. An action, $a$, is independent of another action, $b$, if the evaluation of $a$ yields the same result regardless of whether $b$ has been executed or not.

Definition 4.1.3. Two transactions conflict when their actions are dependent.
Definition 4.1.4. $f_a$ is a function representing how the action $a$ changes the database. If $db$ is the database before the change then $f_a(db)$ is the database after the change.

Note that $a$ will only change the database if it is an update.

Definition 4.1.5. An update changes the database by adding or removing propositions. The extensional predicates of which the proposition defines something are thereby redefined.

Example 4.1.2. $e_a = P(2)$, command: $\neg$

$P(x)$ is redefined: $P_N(x) = P_O(x) \land (x = 2)$,

$N$ for new definition and $O$ for old definition

Definition 4.1.6. The inverse $a^{-1}$ of an action $a$ is, if $a$ is a query, the query itself. If $a$ tries to add some propositions then $a^{-1}$ is the action trying to remove the same propositions. If $a$ tries to remove some propositions then $a^{-1}$ is the action trying to add the same propositions.

Example 4.1.3. Action $a$ is $+P(2)$. Action $a^{-1}$ is $\neg P(2)$

Definition 4.1.7. $e_a^*$ is the predicate expression exactly describing the set of propositions $e_a$ added or removed.

Example 4.1.4. $e_a = P(2)$, command: $\neg$

$e_a^* = P(2)$

Example 4.1.5. In the database: $T(2,1)$ and $T(2,2)$

$e_a = T(2,y)$, command: $\neg$

$e_a^* = T(2,1) \land T(2,2)$

Which propositions $e_a$ removed depends on the content of the database.

Lemma 4.1.1. $b$ is an update.

If the update is an add operation then:

a) $f_b(db) \land \neg f_{b^{-1}}(db) = e_b^*$

If the update is a remove operation then:

b) $f_{b^{-1}}(db) \land \neg f_b(db) = e_b^*$

Proof. a) and b) follows from that remove and add are the only two possible updates and that they are inverse actions.
4.1. LOCKING

Example 4.1.6. \( e_b = P(2) \), command: +
\( e_{b-1} = P(2) \), command: −
\( e_b^* = P(2) \)

Case 1: \( db = \{ P(2), P(3) \} \)
\( f_b(db) = db \)
\( f_{b-1}(db) = P(3) \)
\( \Rightarrow f_b(db) \land \neg f_{b-1}(db) = P(2) = e_b^* \)

Case 2: \( db = \{ P(3) \} \)
\( f_b(db) = \{ P(2), P(3) \} \)
\( f_{b-1}(db) = db \)
\( \Rightarrow f_b(db) \land \neg f_{b-1}(db) = P(2) = e_b^* \)

Lemma 4.1.2. If \( a \) is a query then
a) \( f_a(db) = db \)

If \( a \) is an update then
b) \( f_a(db) = db \Rightarrow f_{a-1}(db) \neq db \)
\( f_a(db) = db \land f_{a-1}(db) \neq db \Rightarrow f_a(f_a(db)) \neq db \)

c) \( f_{a-1}(db) = db \Rightarrow f_a(db) \neq db \)
\( f_{a-1}(db) = db \land f_a(db) \neq db \Rightarrow f_a(f_{a-1}(db)) \neq db \)

d) \( f_{a-1}(f_a(db)) \neq f_a(f_{a-1}(db)) \)

Proof. a) Trivial as a query does not change the database.
b) Follows from that \( a \) and \( a^{-1} \) are inverse update actions and that one of two such action will always change the database and from that \( f_{a-1}(f_a(db)) = f_{a-1}(db) \) and \( f_{a-1}(db) \neq db \).
c) Equivalent to b).
d) From \( f_{a-1}(f_a(db)) = f_{a-1}(db) \) and \( f_{a-1}(db) \neq db \) and \( f_a(db) = f_a(f_{a-1}(db)) \) \( \Box \)

The objective of the predicate locking system is that no conflicting transactions should be allowed to execute in parallel. To achieve this it must be known when two transactions conflict. They conflict when the actions they consist of are dependent. The following problem must be solved:
**Problem.** Decide when transaction $A$ conflicts with transaction $B$, $A \neq B$. $A$ consists of actions $a_i$ and $B$ of actions $b_j$.

**Solution.** For each action $a_i$ in $A$ decide if $a_i$ is independent of all actions $b_j$ in $B$. One of the following holds:

(i) $a_i$ is a query and $b_j$ is an update
(ii) $a_i$ is a query and $b_j$ is a query
(iii) $a_i$ is an update and $b_j$ is an update
(iv) $a_i$ is an update and $b_j$ is a query

For the different cases the following holds for independence:

(i) $a_i$ is independent of $b_j$ if evaluation of $e_{a_i}$ yields the same result before and after $b_j$ has been executed. This holds when $e_{a_i}[db] = e_{a_i}[f_{b_j}(db)]$.
(ii) Queries do not change the content of the database, $f_{a_i}(db) = f_{b_j}(db) = db$, and therefore $a_i$ and $b_j$ are always independent.
(iii) $a_i$ is independent of $b_j$ if evaluation of $e_{a_i}$ yields the same result before and after $b_j$ has been executed. This holds when $e_{a_i}[f_{b_j}(db)] = e_{a_i}[db]$.
(iv) $a_i$ is independent of $b_j$ if evaluation of $e_{b_j}$ yields the same result before and after $a_i$ has been executed. This holds when $e_{b_j}[db] = e_{b_j}[f_{a_i}(db)]$.

In the problem above, one important thing has not been taken into account. The actions are executed and compared in a concurrent system, with the consequence that the exact content of the database is unknown in real time.

**Example 4.1.7.** Transaction $t_1$ takes a lock and executes action $a$: $+P(2)$, transaction $t_2$ tries to take a lock and execute action $b$: $\neg P(2)$.

After $t_1$ has taken its lock, $t_2$ tries to take a lock. At this moment $t_1$ can either have executed $a$ or not have executed $a$. The independence between $a$ and $b$ is investigated in the two cases with the formula $e_{a_i}[f_{b_j}(db)] = e_{a_i}[db]$.

**Case 1:** In database: $\{P(2)\}$ 
\[ f_{b_j}(db) \neq db \Rightarrow e_{a_i}[f_{b_j}(db)] = false, \]
\[ e_{a_i}[db] = true \Rightarrow a \text{ and } b \text{ dependent} \]

**Case 2:** In database: $\{\}$ 
\[ f_{b_j}(db) = db \Rightarrow e_{a_i}[f_{b_j}(db)] = e_{a_i}[db] \Rightarrow a \text{ and } b \text{ independent} \]

In case 1, the result of $a$ is in the database and in case 2 the result of $a$ is not in the database. Depending on whether the result of executing $a$ has been written to the database or not the formula yields different results. This is not acceptable.

The independence of two actions must not depend on whether the result of one of them is written to the database or not. If it does, the unacceptable situation of Example 4.1.7 may arise. The formula to test independence between actions $(e_{a_i}[f_{b_j}(db)] = e_{a_i}[db])$ must be
reformulated.

The problem is that in case 2, \( b \) does not change the content of the database but in case
1 it does. To test the independence between \( b \) and other actions there must be a difference
between versions of the database on which \( b \) has been applied and versions on which \( b \) has
not been applied. In case 2, \( b \) does not change it but when \( f_b(db) = db \) it is known from
Lemma 4.1.2b that \( f_{b^{-1}}(db) \neq db \). Therefore \( f_{b^{-1}} \) is applied instead of \( f_b \). The formula to
be tested will then be \( e_a[db] = e_a[f_{b^{-1}}(db)] \). But then the content of the database will not
be changed in case 1 \( (f_{b^{-1}}(db) = db) \). If instead the formula \( e_a[f_b(db)] = e_a[f_{b^{-1}}(db)] \) is used
the difference between \( f_b(db) \) and \( f_{b^{-1}}(db) \) will be \( e_b \) (Lemma 4.1.1). This is the difference
action \( b \) could make if the content of the database were such that action \( b \) changed it. To
use this difference when evaluation \( e_a \) will make the independence of \( a \) and \( b \) independent
of the variable content of the database at the moment of the evaluation.

**Definition 4.1.8.** The variable content of the database is defined as the content of the
database a user is allowed to change during execution, that is the propositions defining the
extensional predicates.

The independence of the variable content of the database is exactly the required in-
dependence. If Example 4.1.7 was solved again and independence of \( a \) and \( b \) tested by
\( e_a[f_b(db)] = e_a[f_{b^{-1}}(db)] \) the result of case 1 and case 2 would be unambiguous:

**Example 4.1.7*.** The independence between \( a \) and \( b \) is investigated in
the two cases with formula \( e_a[f_b(db)] = e_a[f_{b^{-1}}(db)] \).

Case 1: In database: \( \{ P(2) \} \)

\[
\begin{align*}
f_b(db) & \neq db \Rightarrow e_a[f_b(db)] = false, \\
e_a[f_{b^{-1}}(db)] &= true \Rightarrow a \text{ and } b \text{ dependent}
\end{align*}
\]

Case 2: In database: \( \{ \} \)

\[
\begin{align*}
f_b(db) = db \Rightarrow e_a[f_b(db)] = e_a[db] = false, \\
e_a[f_{b^{-1}}(db)] = true \Rightarrow a \text{ and } b \text{ dependent}
\end{align*}
\]

The answers from the two cases are the same.

From these results the following lemma can be stated:

**Lemma 4.1.3.** The independence of two actions is independent of the variable content of
the database in which the actions are executed, if the independence of the actions are tested
with \( e_a[f_b(db)] = e_a[f_{b^{-1}}(db)] \).

**Definition 4.1.9.** If the independence of two actions does not depend on the variable
content of the database the actions are totally independent.

**Lemma 4.1.4.** If the independence of two actions \( a \) and \( b \) are totally independent then
\( a \) and \( b^{-1} \) are totally independent
\( a^{-1} \) and \( b \) are totally independent
\( a^{-1} \) and \( b^{-1} \) are totally independent

**Proof.** a) and b) When \( a \) and \( b \) are totally independent it follows directly from Definition
4.1.6.

c) consequence of a) and b)
CHAPTER 4. INVESTIGATION

Note that Lemma 4.1.4 does not hold for all independent actions, only for totally independent actions.

Definition 4.1.10. A predicate expression is independent of another predicate expression if changing the truth value of one of the expressions does not affect the truth value of the other expression.

Example 4.1.8. Suppose
\[ P(x) = (x = 3) \lor (x = 4) \lor (x = 5) \], then
\[ e_a = P(x) \land (x = 5) = \text{true} \text{ for } x = 5, \]
\[ e_b = P(x) \land (x = 4) = \text{false} \text{ for } x = 4. \]

Will the truth values of \( e_a \) and \( e_b \) affect each other?
The truth value of \( e_a \) will change if \( x = 5 \) is removed from \( P(x) \) and the truth value of \( e_b \) will change if \( x = 4 \) is removed from \( P(x) \).
Answer: No, \( e_a \) and \( e_b \) are independent.

Example 4.1.9. Suppose
\[ P(x) = (x = 3) \lor (x = 5) \], then
\[ e_a = P(x) \land (x = 5) = \text{true} \text{ for } x = 5, \]
\[ e_b = P(x) \land (\neg x = 5) = \text{false} \text{ for } x = 5. \]

Will the truth value of \( e_a \) and \( e_b \) affect each other?
The truth values of \( e_a \) and \( e_b \) change if \( x = 5 \) is removed from \( P(x) \).
Answer: Yes, \( e_a \) and \( e_b \) are dependent.

Lemma 4.1.5. If an action, \( a \), is totally independent of another action, \( b \), the predicate expression defining \( a \) is independent of the predicate expression defining \( b \).

Proof. If the predicate expressions truth values are dependent then the actions must also be dependent as an action is a predicate expression and a command. Therefore the predicate expressions must be independent if the actions are totally independent. \(\square\)

4.1.2 To Lock on a Predicate Expression

An update adds or removes propositions to the database. By adding and removing these propositions the extensional predicates are redefined. To query the database is to evaluate the predicate expression in the database.

A transaction locks the predicate expression it has evaluated in the database or added or removed from the database. It locks the expression because it is it that should be evaluated to test if it yields the same result before and after another action has been executed. By testing this, the truth value of the predicate expression is protected and the transaction holding it will only read consistent data. So a lock is a saved predicate expression and it protects its truth value and thereby the data affecting the evaluation of it is indirectly locked.
4.1. LOCKING

4.1.3 The Locking Protocol

A particular sequence of actions in a set of transactions is called a schedule. A schedule which gives each transaction a consistent view of the database is called a consistent schedule. To make certain that only consistent schedules are executed, a protocol for how the transactions are allowed to lock and unlock their predicate expressions is needed.

Definition 4.1.11. Strong Strict Two Phase Locking (SS2PL)

Locks are handled by a transaction in two distinct, consecutive phases. The first phase is during the actions execution and the second is after the transaction has committed or aborted.

Phase 1: Locks are acquired and no locks are released
Phase 2: All locks are released

The SS2PL protocol guarantees that only consistent schedules will be executed [17, p. 144]. The SS2PL protocol does not prevent deadlock. The SS2PL protocol is used in the implementation for this thesis. A transaction locks its predicate expressions in the order it is using them and none of the predicate expressions are unlocked until after the transaction has either aborted or committed. To solve the deadlock problem one alternative is to define a limit for how long a transaction is allowed to wait for a lock held by another transaction. If the waiting exceeds the limit the transaction is rolled back. Another alternative is to set up rules for how transactions can be allowed to preempt each other. The alternative to set a time limit was chosen for the implementations made in this thesis. This choice was made because it is the simplest alternative to implement and the focus of this thesis is not on preemption strategies.

4.1.4 Shared and Exclusive Locks

The database can be queried or updated. A query only reads from the database whereas an update writes (or overwrites) data to the database. As reading data does not change the database several transactions can be allowed to read the same data, but only one transaction at a time should be allowed to write/overwrite the same data. To regulate these two different behaviours two types of locks are introduced: shared and exclusive locks. An exclusive lock will assure exclusive access to the data the predicate expression protects. Such a lock is
CHAPTER 4. INVESTIGATION

needed if the transaction is manipulating the database by redefining extensional predicates, in other word if it is adding or removing data from the database [17, p. 131]. A shared lock can be held by several transactions simultaneously. All these transactions are allowed to read the protected data. Such a lock is sufficient if the transaction is only querying the database [17, p. 131].

4.1.5 Granularity of Predicate Locking

Predicate locking can have arbitrarily fine granularity. Actions that are logically independent will be independent in a predicate locking system. Predicate locking has the positive properties of both "tuple" and "column" locking from a relational database context. Consider for example a query asking how many people that are employed at a company and an update changing the salary of an employee of the company. These two actions are independent because the number of employees and their salaries are independent. This is more or less the equivalence of column locking in a physical locking system. But this system also supports the equivalence, more or less, of tuple locking. To change the salaries of two different employees are also two independent actions in a predicate locking system.

4.1.6 Locking on Extensional Predicate Level

The database is changed \((f_k(db))\) by redefinitions of extensional predicates. The extensional predicates are redefined by the addition or removal of propositions. The result of an evaluation of a predicate expression will only be different, \(e_a[f_k(db)] \neq e_a[f_k^{-1}(db)]\) if the evaluation depends on the changed variables, that is the evaluated predicate expression and the predicate expression that changed the database depend on the same extensional predicate. The following theorem is constructed from the above reasoning:

**Theorem 4.1.1.** Two actions are independent if the predicate expressions defining them do not depend on the same extensional predicate.

**Proof.** Follows directly from the reasoning above and Lemma 4.1.5. \(\Box\)

Note that the opposite does not hold. If two predicate expressions depend on the same extensional predicate, the actions they define need not be dependent.

In a first implementation, the predicate expressions defining the actions are locked and the protocol and lock types, which have been described, are used. When a transaction wants to execute a new action it is tested if the predicate expression, which defines the action, depends on the same extensional predicates as the taken locks do. If it does not, the action can be executed and the lock on the predicate expression is taken. The independence of actions is tested through dependence of extensional predicates because this is an easier test to make than to test \(e_a[f_k(db)] = e_a[f_k^{-1}(db)]\). The granularity of the locks in this system will be coarse, a lock locks whole extensional predicates. This is more or less equivalent to locking on table level in a physical locking system. An extensional predicate in a predicate locking system is more or less the equivalence of a table in a physical locking system.

The time complexity of locking on extensional predicate level can be reduced to \(O(n)\) where \(n\) is the number of nodes in the locked predicate expression. \(O(n)\) comes from traversing the predicate expression to find the extensional predicates it depends on. A hash map can be used to store the locked extensional predicates and then the time to check if a certain extensional predicate is locked is on average \(O(1)\). If the predicate expression contains \(t\) extensional predicates then the cost to test them all is \(O(t)\). As \(n > t\) the total time complexity is \(O(n)\).
4.1. LOCKING

4.1.7 Strategies for Testing Independence

**Definition 4.1.12.** An action conflicts with a lock if the result of evaluating the locked predicate expression yields different results depending on whether the action has been executed or not.

A transaction, $t_1$, wants to execute an action. Both other transactions and $t_1$ may at that time hold locks on predicate expressions. Before $t_1$ can execute it must test that the action does not conflict with any lock held by other transactions. One of the following two strategies can be used to do this.

**Definition 4.1.13. Strategy Check Others’ Locks**
A transaction checks whether other transactions hold conflicting locks before taking a lock.

**Definition 4.1.14. Strategy Check Own Locks First**
A transaction first checks if it holds the lock or locks equivalent to the lock it wants to take. If it has, it need not take a new lock. If it has not, it checks whether other transactions hold conflicting locks before taking the lock.

It is known from Definition 4.1.15 that two predicates are equal if they for every input value returns the same truth value. This can be tested according to Theorem 4.1.2. This is how the equality of two locks a transaction owns is tested.

The drawback of strategy “Check Others’ Locks” is that every time a transaction uses a predicate expression it has already locked, it will not realize it already holds the lock, but instead test if any other transaction holds a conflicting lock. The other transactions will never hold such a lock, as the transaction itself holds the lock, and therefore the conflict testing is unnecessary. The result will be that the transaction holds as many locks on the same predicate expression as the number of times the transaction has executed the action which is defined by the predicate expression. Holding multiple locks will however only be a problem (space demanding and time consuming) in a system where long transactions are prone to execute the same actions many times. Consider for example if an active transaction has used an action 10000 times, then there exists 10000 locks on that action and all other transactions wanting a lock must test their actions 10000 times against the same lock. However the situation described seems quite unlikely. Adding or removing the same propositions multiple times does not make sense. It is hard to imagine the application. Strategy ”Check Own Locks First” does not have the drawback of multiple locks. But if the transactions are of a type that uses many different locks, it will be time consuming to test if it already has the lock before testing if any other transaction has it. Strategy ”Check Own Locks First” works better when the number of transactions is large because the transactions part of the locks is then probably a small proportion of the total number of locks. It therefore takes little more time testing the own locks before testing conflict with other locks and a lot of time is saved if the transaction owns the lock. In the implementation made for this thesis strategy ”Check Others’ Locks” was chosen.

4.1.8 Testing Independence of Actions

Predicate expressions are either true or false. $e_a$ is a predicate expression. Such an expression can sometimes have a truth value that is independent of the variable part of the database it is evaluated in.

**Lemma 4.1.6.** A predicate expression evaluated in the database can sometimes have a truth value that is independent of the variable content of the database.
Proof. Follows by Example 4.1.10 below. □

Note that the predicate expression can still depend on rules defined for the database.

**Example 4.1.10.** $e_a = (P(x) \land (x = 3) \land \neg(x = 3)) = false$, for all $x \Rightarrow
\ e_a(db) = false, \text{ for all } x$

For this expression it is unnecessary to collect data from the database to evaluate it - its value is independent of the variable content of the database. The following expression, on the other hand, needs data from the database before the truth value of it is known.

**Example 4.1.11.** $e_a = (P(x) \land (x = 3))$ has an unknown truth value. All propositions defining $P(x)$ must be collected from the database to evaluate $e_a(db)$.

$e_b$ is the predicate expression that changed the database to $f_b(db)$. $e_b$ may also be independent of the database.

**Example 4.1.12.** $e_b = P(3) \land P(2)$, command: +

Predicate expressions which change the database can also depend on the database. The following expression is dependent on the database.

**Example 4.1.13.** $e_b = R(2, y)$, command: ¬

When $e_a^*$ is added to or removed from the database one or several extensional predicates are redefined. If $e_b$ is independent of the database then it is known how the extensional predicates are redefined.

**Example 4.1.14.** $e_b = P(3) \land P(2)$, command: +

$P_N(x) = P_O(x) \lor (x = 2) \lor (x = 3)$,

$N = \text{new } db, \; O = \text{old } db$

The new definition of $P(x)$ is the old one with the two new propositions added.

Actions $a$ and $b$ can only be dependent if they depend on the same extensional predicate (Theorem 4.1.1). If the evaluation of $e_a$ depends on the database, then the definitions of the extensional predicates on which $e_a$ depends will be collected from the database. Say $e_a$ depends on the extensional predicate $P(x)$ and $b$ redefines $P(x)$. To evaluate $e_a, P(x)$ will be read from the database - but by knowing action $b$, the redefinition of $P(x)$ is known without reading from the database ($e_b$ is independent of the database). The redefinition of $P(x)$ can be added directly into $e_a$.  

22
4.1. LOCKING

Example 4.1.15. \( e_a = P(x) \land (x = 3) \)

From example 4.1.14: \( P_N(x) = P_O(x) \lor (x = 2) \lor (x = 3) \)

Add \( P_N(x) \) into \( e_a \Rightarrow e_a^{f_b} = (P_O(x) \lor (x = 2) \lor (x = 3)) \land (x = 3) \)

Lemma 4.1.7. When \( e_b \) is independent of \( db \) and redefines an extensional predicate on which \( e_a \) depends, the redefinition can be added to \( e_a \). This new expression is named \( e_a^b \) and \( e_a^b = e_a \)

\[ e_f^b(a) = (P_O(x) \lor (x = 2) \lor (x = 3)) \land (x = 3) \]

Proof. To write \( e_b \) to the database and afterwards read the redefinition/redefinitions is equivalent to adding the redefinition/redefinitions directly into the \( e_a \) and read the old value of the redefined predicate expression from the database if needed. If necessary rules defined for the database may be applied before the redefinitions are added.

\( e_a^b \) is a predicate expression and can have a value that is independent of the database.

The continuation of Example 4.1.15:

Example 4.1.15 continued.

\[ e_b = (P_O(x) \lor (x = 2) \lor (x = 3)) \land (x = 3) = (P_O(x) \land (x = 3)) \lor ((x = 2) \land (x = 3)) \lor ((x = 3) \land (x = 3)) = (P_O(x) \land (x = 3)) \lor (x = 3) = true, \text{ for } x = 3 \text{ and false for all other values} \]

\[ e_a^{f_b-1}(db) = (P_O(x) \land \neg((x = 2) \lor (x = 3))) \land (x = 3) = (P_O(x) \land ((\neg(x = 2)) \land \neg(x = 3))) \land (x = 3) = false, \text{ for all } x \]

Equality of predicates is defined in the following definition.

Definition 4.1.15. Two predicates are equal if they, for every input, both return the same truth value.

Theorem 4.1.2. The equality of two predicate expressions, \( e_1(x) \) and \( e_2(x) \), can be tested through the expression

\[ (e_1(x) \land \neg e_2(x)) \lor (e_2(x) \land \neg e_1(x)) \]

If the expression is constant false the expressions are equal.

\( e_1(x) = e_2(x) \Rightarrow (e_1(x) \land \neg e_2(x)) \lor (e_2(x) \land \neg e_1(x)) = false, \text{ for all } x \)

<table>
<thead>
<tr>
<th>( e_1 )</th>
<th>( e_2 )</th>
<th>( e_1 \land \neg e_2 )</th>
<th>( e_2 \land \neg e_1 )</th>
<th>( (e_1 \land \neg e_2) \lor (e_2 \land \neg e_1) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Proof. Only in the first and last row, where \( e_1 \) and \( e_2 \) have the same values, does the expression evaluate to false. \( \square \)
From the theorem follows:

**Corollary 4.1.1.** \((c \land \lnot d) \lor (d \land \lnot c) = \text{false} \iff (c \land \lnot d) = \text{false and } (d \land \lnot c) = \text{false}\

Therefore \(e^L_a[db] = e^{L^h-1}_a[db]\) when \(e^L_a[db] \land \lnot e^{L^h-1}_a[db] = \text{false}, \) for all \(x\) and \(e^{L^h-1}_a[db] \land \lnot e^L_a[db] = \text{false}, \) for all \(x.\)

Both \(e^L_a\) and \(e^{L^h-1}_a\) may be independent of the database. As \(e^L_a\) and \(e^{L^h-1}_a\) are predicate expressions, they evaluate either to true or to false for certain \(x\) in the database. As they are bound together with a predicate operator they can be combined into one compound expression that can be evaluated in the database.

**Lemma 4.1.8.** If \(e_1\) and \(e_2\) are defined then
\[
\begin{align*}
e_1[db] \land \lnot e_2[db] & \iff (e_1 \land \lnot e_2)[db] \\
e_2[db] \land \lnot e_1[db] & \iff (e_2 \land \lnot e_1)[db]
\end{align*}
\]

**Proof.** The evaluation is substituting parts of the predicate expression with their definitions from the database. If this substitution is done before or after the predicate expressions have been connected with \(\land\lnot\) makes no difference, as the database from which the connected expressions takes the definitions is the same for both expressions.

The reason the predicate expressions are combined into a bigger predicate expressions before the expressions are evaluated in the database is that the expressions might be simplified in such a way that they have constant truth values. That is, the whole expressions are independent of the database.

**Theorem 4.1.3.** When \(e_1\) is independent of the database and \(e^L_a\) and \(e^{L^h-1}_a\) are defined then if \(e_1[db][f_a(db)]\) is equal to \(e_1[f_a(db)]\) can be tested by checking if \((e^{L^h-1}_a \land \lnot e^L_a)[db]\) and \((e^{L^h-1}_a \land \lnot e^{L^h-1}_a)[db]\) are both false.

**Proof.** Follows from Lemma 4.1.7 and Lemma 4.1.8

**A Special Case**

There is a special dependence between actions that must be detected, namely when two actions try to redefine the same extensional predicate with the same proposition. Why is explained in the example below. The dependence is detected correctly by a test according to Theorem 4.1.3. Suppose transaction \(t_1\) redefines \(P\) by adding the proposition \(P(2)\) and takes a lock. When \(t_2\) also tries to redefine \(P\) by adding \(P(2)\) it should be blocked. Why is this a conflict at all? Both transactions want to add the proposition \(P(2)\) to \(P.\) When \(t_1\) locks \(P(2)\) it takes an exclusive lock \(L_1\) on the answer to \(P(2)\). If \(t_2\) was allowed to take a lock on \(P(2)\) a problematic situation could arise. \(t_2\) will only be allowed to take a shared lock as it cannot be allowed to change the answer to the question \(P(2)\), which \(t_1\) locked. If \(t_2\) takes a shared lock, \(t_1\) will, if strategy "Check Other’s Locks" for comparing locks is used, the next time it wants to use \(P(2)\), only be allowed to take a new shared lock despite already having an exclusive lock on the same truth value. If instead strategy "Check Own Locks First" was used and \(t_2\) was allowed to take a lock on \(P(2)\) \(t\) it would be even more problematic, as \(t_1\) then would be allowed to change the answer to \(P(2)\) and \(t_2\) would read inconsistent data. Conclusion:

**Definition 4.1.16.** Two transactions should not be allowed to make exactly the same redefinition of an extensional predicate.
4.1. LOCKING

Now an example, illustrated in pictures 4.1 - 4.8, which shows that Theorem 4.1.3 detects the conflict correctly.

**Example 4.1.16.** Transaction $t_1$ has executed action $a$. $e_a$ redefines the extensional predicate $P(x)$ through removing proposition $P(3)$ from the definition. $e_a$ is illustrated in Figure 4.1. Transaction $t_2$ wants to execute action $b$. $e_b$ redefines the extensional predicate $P(x)$ through removing proposition $P(3)$ from the definition. The redefinition $b$ made of $P(x)$ is found in Figure 4.2. The redefinition $b^{-1}$ made of $P(x)$ is found in Figure 4.3. Figure 4.4 illustrates how the two new revisions of $P(x)$ are added to $e_a$. In Figure 4.5 $e_a^b$, the predicate constructed by adding the revision from Figure 4.2 to $e_a$. In Figure 4.6 $e_a^{b^{-1}}$, the predicate constructed by adding the revision from Figure 4.3 to $e_a$. In Figure 4.7 $e_a^{b^{-1}} \land \neg e_a^b$ is shown and in Figure 4.8 $e_a^{b^{-1}} \land \neg e_a^{b^{-1}}$ is shown. The truth values of these two predicate expressions should be compared. If they both are false, they are according to Theorem 4.1.3 independent. As they are not both false for all values they are dependent and to the detect the conflict defined in Definition 4.1.16 correctly they should be.

---

![Figure 4.1](image1.png)

**Figure 4.1.** The locked predicate expression, $e_a$, defining action $a$.

![Figure 4.2](image2.png)

**Figure 4.2.** The update defining action $b$ has redefined extensional predicate $P(x)$. $P_N(x) = P_O(x) \land \neg(x = 3)$, $N$ for new revision and $O$ for the old one.

![Figure 4.3](image3.png)

**Figure 4.3.** The update defining action $b^{-1}$ has redefined extensional predicate $P(x)$. $P_N(x) = P_O(x) \lor (x = 3)$, $N$ for new revision and $O$ for the old one.
CHAPTER 4. INVESTIGATION

Figure 4.4. The two new revisions of $P(x)$ are added to $e_a$

Figure 4.5. $e_a^{f_{k-1}}$, the new definition of $P(x), P_N(x)$, has been added to $e_a$.

Figure 4.6. $e_a^{f_k}$, the new definition of $P(x), P_N(x)$, has been added to $e_a$.

Is Definition 4.1.16 obeyed when Theorem 4.1.1 is used to test the independence? As both transactions causing the conflict defined in Definition 4.1.16 manipulates the same extensional predicate, they are dependent according to Theorem 4.1.1.
4.1. LOCKING

Algorithm 1

The algorithm used in the implementation, for testing independence between an action $a$ and an action $b$ (an update), is the following:

1. independent = false;
2. finished = false;
3. if(eb not dependent on database){
   4. if(ea constant){
      5. independent = true;
      6. finished = true;
   7. }else if(ea and eb do not depend on the same extensional predicate){
      8. independent = true;
      9. finished = true;
   10. }else{
      11. construct the revisions of the extensional predicates action b and b-1 defined;
      12. define eafb and eafb-1 by adding the revisions of the extensional predicates to ea;
      13. construct (eafb-1 AND-NOT eafb) and (eafb AND-NOT eafb-1);
      14. simplify the expressions and determine their truth value;
CHAPTER 4. INVESTIGATION

if(both are false){
    independent = true;
    finished = true;
}else if(both are true){
    independent = false;
    finished = true;
}
}
if(not finished){
    estimate the cost of evaluating the expressions;
    if(not too expensive){
        lock parts of the database used for the evaluation;
        evaluate the expressions in the database;
        independent = result of evaluation;
    }else{
        independent = false;
    }
}

Comments on the algorithm:
Row 3: If $e_b$ is dependent on the database, it is not known how $e_b$ redefines the extensional predicates.
Row 4: If $e_a$ has a constant truth value it will not be affected by action $b$.
Row 7: Independence according to Theorem 4.1.1.

All transactions holding locks may be accessing parts of the database. To execute a predicate expression to get to know if two actions are dependent one must make sure that none of the active transactions manipulate the data the predicate expressions depend on. For this an extra level of locks are needed. These locks could lock on extensional predicate level. Double locking systems will yield correct results when investigate dependence but the time complexity for the systems will get much worse because of the extra lock level.

The time complexity of the algorithm is probably too high for it to be practically useful. It is probably a better idea to lock on extensional predicate level instead of using the full algorithm.

4.1.9 Precompilation

If the type of queries and updates are known in advance the predicate expressions can be precompiled. Precompilation is used to lower the time complexity of independence testing.

$n$ - The number of nodes in a predicate expression
$n_1$ - The number of nodes in the precompiled expression
$q$ - The number of possible queries
$u$ - The number of possible updates
$k$ - some constant

$a$ and $b$ are types of actions which can have different arguments.

Precompile all queries combined with the updates ($q * u$) and every update combined with every other update ($\sum_{i=1}^{u} (u - i)$).
4.1. LOCKING

1. Construct \((e_a^{f_{a-1}} \land \neg e_a^{f_{a}})\) and \((e_a^{f_{a}} \land \neg e_a^{f_{a-1}})\)  \(O(1)\)

2. Optimize the constructed expressions  \(O(k^n)\)

Total construction time:  \(O(u \times (q + u))\)
Total optimization time:  \(O((u \times (q + u)) \times k^n)\)

The construction and optimization is performed before execution so the exponential time complexity is not a problem during execution.

Note that if \(a\) and \(b\) does not depend on the same extensional predicate, \(e_a^{f_{a}} = e_a^{f_{a-1}}\) and the optimized expression will be false independent of the values of the arguments of \(a\) and \(b\).

\(t\) - The number of extensional predicates a predicate expression depends on.
\(t \leq n^2 + 1\) but normally \(t \ll n\).

\(s_1\) - The number of shared locks at the time.
\(s_2\) - The number of exclusive locks at the time.

The time complexity below is the one for testing one expression. To test if a new query conflicts with any of the locks, adds factor \(s_2\) to the time complexity and to test if a new update conflicts with any of the locks adds a factor \((s_1 + s_2)\) to the time complexity. Exception: to take the defensive decision to lock the database.

3. Use the precompiled expression to test the independence of the predicate expressions it represents. There are some alternatives:
   a) Defensive decision - lock the database  \(O(1)\)
   b) Test if the predicate expressions depend on the same extensional predicate - if so lock the database  \(O(t)\)
   c) Check if the precompiled expression is constant true or false  \(O(n_1)\)
   d) Execute the expression in real time  \(O(p^n)\) \(^1\).

---

**Example 4.1.17. A precompiled expression**

\(((P(x) \land \neg (x = a)) \land (x = b)) \land \neg((P(x) \lor (x = a)) \land (x = b)) \Rightarrow \neg equal(a,b)\)

If for example \(a = 2\) and \(b = 3\) the expression will be \(\neg equal(2,3) = true \Rightarrow\) the actions are independent.

**Algorithm 2 - with precompilation**

All queries and updates allowed by the system are precompiled. The algorithm used in the implementation for testing independence between an action between an action \(a\) and an action \(b\) (an update), is the following.

\(^1\)When two extensional predicates are combined the combination yielding the worst time complexity is a Cartesian product. The time complexity of a Cartesian product is \(O(p^2)\), where \(p\) is the number of propositions defining the extensional predicate. Consider a predicate expression with \(n\) nodes, depending on extensional predicates that are defined by \(p\) propositions. The worst case time to process that expression is then \(O(p^{2 \log(n) - 1})\).
CHAPTER 4. INVESTIGATION

```plaintext
1 independent = false;
2 finished = false;
3 if(eb not dependent on database){
4     if(ea constant){
5         independent = true;
6         finished = true;
7     }else{
8         investigate the precompiled expressions truth value;
9         if(both are false){
10            independent = true;
11            finished = true;
12         }else if(both are true){
13            independent = false;
14            finished = true;
15     }
16 }
17 }
18 if(not finished){
19     estimate the cost of evaluating the expressions;
20     if(not too expensive){
21         lock parts of the database used for the evaluation;
22         evaluate the expressions in the database;
23         independent = result of evaluation;
24     }else{
25         independent = false;
26 }
27}
```

Comments on the algorithm:
Row 3: If $e_b$ is dependent on the database the revisions it produce of extensional predicates are unknown.
Row 4: if $e_a$ has a constant truth value it will not be affected by action $b$.

The difference between the two algorithms is that rows 7 to 14 in algorithm 1 is exchanged for rows 7 and 8 in algorithm 2. In the second algorithm, no expressions will be built and/or simplified. The truth values of the precompiled expressions will be investigated and due to the precompilation more expression will have known truth values.

4.2 The Group Operator

The group operator does not belong to first order logic, it is a second order logic operator. The operator is used to create groups of values. These groups can be used as parameters to predicate expressions. For example all bike owners can be made a group of and the group can be the parameter to a predicate expression choosing the owner with the biggest bike. With this operator more complex operations, such as finding the largest value in a set of propositions or counting the number of propositions defining a certain extensional predicate, can be expressed. These complex actions makes is possible to use the fine granularity of predicate locking more fully. Consider the example from section 4.1.5. There the group operator is used to count the number of employees and this action is independent of the action changing the salary of an employee.
4.3. THE OPTIMIZER

The group operator also has a negative property. The result of applying a group operator is always dependent on the variable content of the database. For example which element that is the largest one depends on which elements the database contains. Therefore all predicate expressions using group are dependent on the content of the database. Such a predicate expression can only be precompiled into expressions that are still dependent on the database. The rare exception is if there are two group operators take out each other. The consequence is that no truth values will ever be known in advance about predicate expressions containing group. The only way of knowing if an arbitrary predicate expression conflicts with an expression containing group is to execute the compound predicate expression in the database. Such an execution is time consuming in itself and requires a double locking system. Therefore it might be better never to execute it but lock the extensional predicates the predicate expression depends on.

Example 4.2.1. The expression $T(x) \land \max (\text{group}(x), \max \_v)$, where $T(x)$ is an extensional predicate, is locked on extensional predicate level by locking $T(x)$.

4.3 The Optimizer

A predicate expression with group can be precompiled, but the precompiled expression will still be dependent on the variable content of the database. This makes it less interesting to be able to precompile these predicate expressions. How this is done is therefore not investigated in this thesis. The predicate expressions most interesting to precompile is the ones that can be simplified into expressions independent of the variable content of the database. To investigate exactly which predicate expressions this can be done for and which rules the optimizer must know to be able to simplify those lies outside the scope of this thesis. The methods the optimizer uses to precompile is it applies rules, does structure analysis, and recognizes base cases (true, false, =, ≠, <, >). Examples like the following one can of course be simplified.
Example 4.3.1. From the database: $S(x) = U(x) \land (x < 2)$

$e_a = T(x) \land S(x)$

$e_b = T(2)$ command: +

$T_N(x) = T_O(x) \lor (x = 2) \Rightarrow$

$e_{a,b}^N = (T_O(x) \lor (x = 2)) \land S(x) =$

$(T_O(x) \lor (x = 2)) \land U(x) \land (x < 2) = \text{true} \text{ for } x = a$

when $a < 2$ and $T_O(a) = \text{true}$ and $U(a) = \text{true}$

$T_N(x) = T_O(x) \land \lnot(x = 2) \Rightarrow$

$e_{a,b}^{N,-1} = (T_O(x) \land \lnot(x = 2)) \land S(x) =$

$(T_O(x) \land \lnot(x = 2)) \land U(x) \land (x < 2) = \text{true} \text{ for } x = a$

when $a < 2$ and $T_O(a) = \text{true}$ and $U(a) = \text{true}$

By applying the rule for $S(x)$ the optimizer can simplify the predicate expressions and find that they are true and false for the same values of $x$, independent of the content of the database.

Two predicate expressions dependent on different extensional predicates are always independent (Theorem 4.1.1). The optimizer must have a rule to find such independence. Such a rule is easy to construct, the optimizer must only test that two predicate expressions have some predicate expression in common.

Below it will be shown that for a large number of simple cases the optimizer only needs to know and apply one rule to construct an expression independent of the variable content of the database. These simple cases include all queries about existence of an arbitrary group of propositions combined with updates of classes c1-c5 and c8-c10, see section 4.5.2, and all combinations of predicate expressions from the mentioned update classes.

When $e_{a,b}^N$ and $e_{a,b}^{N,-1}$ are constructed the old definitions of extensional predicates are substituted with new definitions. The old definition will in the most common cases be connected, by an and-operator, to some conditions. These conditions will if $T(x_1,x_2,\ldots,x_n)$ is the extensional predicate and $e_a$ is one of the simple cases from above be of the form:


gm \quad ((x_1 = a_{11}) \land (x_2 = a_{12}) \land \ldots \land (x_n = a_{1n})) \lor ((x_1 = a_{21}) \land (x_2 = a_{22}) \land \ldots \land (x_n = a_{2n})) \lor \ldots \lor ((x_1 = a_{m1}) \land (x_2 = a_{m2}) \land \ldots \land (x_n = a_{mn}))

All $a_{ij}$ need not exist for all $i, i = 1, 2, \ldots, m$, and $j, j = 1, 2, \ldots, n$.

The new revisions will have the general formulas:

$$
(T(x_1,x_2,\ldots,x_n) \lor ((x_1 = b_{11}) \land (x_2 = b_{12}) \land \ldots \land (x_j = b_{1n})) \lor ((x_1 = b_{21}) \land (x_2 = b_{22}) \land \ldots \land (x_j = b_{2n})) \lor \ldots \lor ((x_1 = b_{mn1}) \land (x_2 = b_{mn2}) \land \ldots \land (x_n = b_{mnn})))
$$

$$(T(x_1,x_2,\ldots,x_n) \land \lnot((x_1 = b_{11}) \land (x_2 = b_{12}) \land \ldots \land (x_j = b_{1n})) \lor ((x_1 = b_{21}) \land (x_2 = b_{22}) \land \ldots \land (x_j = b_{2n})) \lor \ldots \lor ((x_1 = b_{mn1}) \land (x_2 = b_{mn2}) \land \ldots \land (x_n = b_{mnn})))$$

All $b_{ij}$ need not exist for all $i, i = 1, 2, \ldots, m$, and $j, j = 1, 2, \ldots, n$. 
4.3. THE OPTIMIZER

Now some examples that lead to the general rules that define when the two expressions with the definitions added are equal.

Example 4.3.2. When do the following expressions,

1. \((T(x) \lor (x = b)) \land (x = a)\) and
2. \((T(x) \land \neg(x = b)) \land (x = a)\) have the same truth values?

1. true when \(x = a\) if \(T(a) = true\) or if \(a = b\)
2. true when \(x = a\) and \(a \neq b\) if \(T(a) = true\)

\(\Rightarrow\) the same truth value when \(a \neq b\)

Example 4.3.3. When do the following expressions,

1. \((T(x) \lor (x = b_1) \lor (x = b_2)) \land ((x = a_1) \lor (x = a_2))\) and
2. \((T(x) \land \neg((x = b_1) \lor (x = b_2))) \land ((x = a_1) \lor (x = a_2))\), have the same truth values?

1. true when \(x = a_i\) for some \(i, i = 1, 2\) if \(T(a_i) = true\) or if \(a_i = b_j\) for some \(i, j, j = 1, 2\)
2. true when \(x = a_i\) for some \(i, i = 1, 2\) and \(a_i \neq b_j\) for all \(j, j = 1, 2\) if \(T(a_i) = true\)

\(\Rightarrow\) the same truth value when \(a_i \neq b_j\) for all \(i, j\)

Example 4.3.4. When do the following expressions,

1. \((T(x_1, x_2) \lor ((x_1 = b_{11}) \land (x_2 = b_{12}))) \land ((x_1 = a_{11}) \land (x_2 = a_{12}))\) and
2. \((T(x_1, x_2) \land \neg((x_1 = b_{11}) \land (x_2 = b_{12}))) \land ((x_1 = a_{11}) \land (x_2 = a_{12}))\) have the same truth values?

1. true when \(x_1 = a_{11}\) and \(x_2 = a_{12}\) if \(T(a_{11}, a_{12}) = true\) or if \(a_{11} = b_{11}\) and \(a_{12} = b_{12}\)
2. true when \(x_1 = a_{11}\) and \(x_2 = a_{12}\) if \(b_{11} \neq a_{11}\) and/or \(b_{12} \neq a_{12}\) if \(T(a_{11}, a_{12}) = true\)

\(\Rightarrow\) the same truth value when \(b_{j1} \neq a_{j1}\) and/or \(b_{j2} \neq a_{j2}\)
Example 4.3.5. When do the following expressions,

1. \((T(x_1, x_2) \lor (((x_1 = b_{11}) \land (x_2 = b_{12})) \lor ((x_1 = b_{21}) \land (x_2 = b_{22})))) \land ((x_1 = a_{11}) \land (x_2 = a_{12})) \lor ((x_1 = a_{21}) \land (x_2 = a_{22}))\) and

2. \((T(x_1, x_2) \land \neg (((x_1 = b_{11}) \land (x_2 = b_{12})) \lor ((x_1 = b_{21}) \land (x_2 = b_{22})))) \land

\(((x_1 = a_{11}) \land (x_2 = a_{12})) \lor ((x_1 = a_{21}) \land (x_2 = a_{22}))\) have the same truth values?

1. true when \(x_1 = a_{k1}\) and \(x_2 = a_{k2}\) for some \(k, k = 1, 2\) if \(T(a_{k1}, a_{k2}) = true\)
   or if \(a_{k1} = b_{11}\) and \(a_{k2} = b_{22}\) for some \(k\) and \(i, k = 1, 2\) and \(i = 1, 2\)

2. true when \(x_1 = a_{k1}\) and \(x_2 = a_{k2}\) for some \(k, k = 1, 2\) and if for all \(i\)
   \(a_{kj} \neq b_{ij}\) for some \(j\) if \(T(a_{k1}, a_{k2}) = true\).

\(\Rightarrow\) the same truth value when for every \(k, k = 1, 2\) and for every \(i\), there is a \(j\)
for which \(a_{kj} \neq b_{ij}, j = 1, 2\)

From the above examples a generalized formula is constructed.

\((T(x_1, x_2, \ldots, x_n) \lor (((x_1 = b_{11}) \land (x_2 = b_{12}) \land \ldots \land (x_j = b_{1n})) \lor ((x_1 = b_{21}) \land (x_2 = b_{22}) \land \ldots \land (x_j = b_{2n}))) \lor \ldots \lor ((x_1 = b_{m1}) \land (x_2 = b_{m2}) \land \ldots \land (x_j = b_{mn}))) \land ((x_1 = a_{11}) \land (x_2 = a_{12}) \land \ldots \land (x_j = a_{1n})) \lor ((x_1 = a_{21}) \land (x_2 = a_{22}) \land \ldots \land (x_j = a_{2n}))) \lor \ldots \lor ((x_1 = a_{m1}) \land (x_2 = a_{m2}) \land \ldots \land (x_j = a_{mn})))\)

\((**)*\) and (***) have the same truth values when for every \(k, k = 1, 2, \ldots, n\) and \(i, i = 1, 2, \ldots, n\), there is some \(j, j = 1, 2, \ldots, m\) for which if \(a_{kj}\) and \(b_{ij}\) exist \(a_{kj} \neq b_{ij}\). All \(a_{kj}\)
and \(b_{ij}\) need not exist in a realistic example but the above holds anyway.

The optimizer needs to recognize expression of type (***) and (****) and then know
what conditions must hold for the truth values of the expressions to be equal. From these
conditions a simple expression for when the actions are independent can be constructed. If
\(m = n = 1\) for example the expression is: \(\neg eq(a, b)\). The rule above can also be generalized
to include conditions and manipulations of several predicate expressions at once.

4.4 The System Types

The time of lock testing increases with the number of active transactions, the number
of possible actions and most important with the complexity of the actions. Many active
transactions result in that each transaction only owns a small proportion of the total number
of locks. For a transaction to take a new lock it must test for conflict with many locks. The
larger the number of possible actions is the larger the number of locks that can be owned
at one time is. The complexity of the actions increase the time it takes to test a conflict
between two locks.
4.5. LOGGING

The kind of system least compatible with predicate locking has many not predefined complex queries and updates and the number of transactions is large and the database small.

A system with few updates and many queries works well with predicate locking. The number of updates is small so to lock a new query only a few conflict tests are required. To lock a new update many tests are required, but as update actions are rare this is acceptable.

A system with few queries and many updates also works well with predicate locking if the updates are of some of the classes c1-c5 and c8-c10. This could for example be a search system. The expression constructed to test conflicts can be optimized according to the rule specified in section 4.3 and will for specific values be true or false. It may take more time to check if the updates conflict with the shared locks but as they are few it does not matter. Predicate locking is also good for systems with many simple queries and updates.

4.5 Logging

4.5.1 Logical Redo and Undo

As mentioned in section 3.2.2, a log that has arbitrarily fine granularity is needed to match the locking system. A logical log has this property. It will therefore be investigated when and how logical logging can be used.

Can both undo and redo be done logically? Logical manipulations of the database only has the expected effect if the database is in a consistent state, either operation or transaction consistent. In a system with atomic propagation the materialized database will always be consistent. In such a system both logical redo and undo can always be used. If atomic propagation is not used, the state of the materialized database after a crash is unknown. Logical redo can only be used under such circumstances if the checkpoint the database is started from is a consistent and complete copy of the database. The checkpoint, not the materialized database, is loaded into main memory as a starting-point for the redo process. Nothing saying that such checkpoints exist has been found, so normally logical redo can not be used in systems that do not use atomic propagation. Why atomic propagation is necessary will be explained in the next part of this text.

When the system is running, main memory is for better performance used as much as possible and data is only written to disc at intervals. If the system crashes, the data in main memory is lost. After the crash, in a system not using atomic propagation, there can be partly written blocks that are corrupt on the disc. The following example using figure 4.9 explains what happens after a crash.
Example 4.5.1. The corrupt data is represented by the half, black dot in block C in "Disc". The version of the database on the disc is loaded to main memory, see "After restart". To recover the database the log records written after the checkpoint, "Log" in the picture, are read. When using physical logging, redo and undo replaces blocks with later/earlier revisions of the blocks so the corrupt blocks are not a problem. The blocks A, B, C and E in "After restart", are updated with the revisions logged in "Log". The corrupt block C is replaced by a non-corrupt version of the same block. In the undo phase the uncommitted changes, the grey dots in the picture, are undone. "Main memory" is a description of the database before the crash. The recovered version after the crash will look the same, except that no uncommitted changes will exist. The database is now transactional consistent.

Logical logging does not log revisions of blocks, it logs logical expressions corresponding to changes. A corrupt block can cause corruptness of the data structure which structures the content of the database. A logical expression inserted in a corrupt data structure will not restore the database. The corruption can not be corrected by the logic expression. This
makes it difficult to use logical logging with redo in a non-atomic propagation system.

Logical undo can be combined with physical redo to handle a crash for a non-atomic propagation system, if the redo phase leaves the database in a consistent state. The redo phase will always do this if all logged actions are redone. If the data structures are stable, that is they do not become corrupt during normal execution, the state of the database is always operational consistent when abort is done.

### 4.5.2 Different Classes of Add and Remove

The predicates used for adding and removing can be divided into different classes. They are found in Table 4.1 and Table 4.2.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Always Idempotent</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Remove one proposition from one extensional predicate.</td>
<td>yes</td>
<td>(T(1,2,3,4))</td>
</tr>
<tr>
<td>c2</td>
<td>Remove one or many propositions from one extensional predicate.</td>
<td>yes</td>
<td>((T(1,2,3,4) \lor T(2,3,4,5)))</td>
</tr>
<tr>
<td>c3</td>
<td>Remove one or many propositions from several extensional predicates.</td>
<td>yes</td>
<td>((S(1,2) \lor T(2,3,4,6) \lor S(0,9)))</td>
</tr>
<tr>
<td>c4</td>
<td>Remove k propositions from one extensional predicate.</td>
<td>yes</td>
<td>a) ((T(w,x,y,z) \land (w = 1)))</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>b) (T(1,x,y,z))</td>
</tr>
<tr>
<td>c5</td>
<td>Class c4 used on many extensional predicates.</td>
<td>yes</td>
<td>((T(1,x,y,z) \lor S(2,y)))</td>
</tr>
<tr>
<td>c6</td>
<td>Predicates that depend on one or several other predicates</td>
<td>no</td>
<td>(T(w,x,y,z) \land \max_{\text{group}(x)}, \max_v))</td>
</tr>
<tr>
<td>c7</td>
<td>Ambiguous expressions.</td>
<td>-</td>
<td>a) ((T(1,x,y,z) \land T(2,x,y,z)))</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>b) ((P(w) \land G(x)))</td>
</tr>
</tbody>
</table>

Table 4.1. The remove classes

Some of the classes are subclasses of other classes. This means that if a superclass is supported, its subclasses are also supported. The relationships between the classes are:

- \(c1 \subset c2 \subset c3 \subset c5 \subset c6\)
- \(c2 \subset c4 \subset c5\)
- \(c8 \subset c9 \subset c10 \subset c11\)
### Table 4.2. The add classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Always Idempotent</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>c8</td>
<td>Add one proposition to one extensional predicate.</td>
<td>yes</td>
<td>$T(1, 2, 3, 4)$</td>
</tr>
<tr>
<td>c9</td>
<td>Add one or many propositions to one extensional predicate.</td>
<td>yes</td>
<td>$(T(1, 2, 3, 4) \land T(2, 3, 4, 5))$</td>
</tr>
<tr>
<td>c10</td>
<td>Add one or many propositions to several extensional predicates.</td>
<td>yes</td>
<td>$(S(1, 2) \land T(2, 3, 4, 6) \land S(0, 9))$</td>
</tr>
<tr>
<td>c11</td>
<td>Predicates that depend on one or several other predicates.</td>
<td>no</td>
<td>$S(x) = P(x) \land T(x)$, $P$ and $T$ extensional predicates. $S$ depends on $P$ and $T$.</td>
</tr>
<tr>
<td>c12</td>
<td>Ambiguous expressions.</td>
<td>-</td>
<td>a) $(R(1) \lor P(1))$ b) $T(1, x, y, z)$</td>
</tr>
</tbody>
</table>

Which of these classes should be supported by the system? Class c7 and class c12 should not be supported because the meanings of these expressions are unclear. They are not supported for logical reasons. Which of the other classes that should be supported depends on how simple and fast one wants the log to be and how complex expressions the system supports.

In the test-implementation, only the classes c1-c5 and c8-c10 are supported.

### 4.5.3 How and What to Log

The logged logical expression should describe the logical change of the database. How is this done? In a relational database context the DML statements and parameters can be logged [7]. The recovery is then achieved by executing some of the previously executed DML statements. DML is not used in a predicate oriented database. Instead predicates describing actions or changes should be saved.

Executing an update can either change or not change the database. Only actions actually changing the database need to be saved in the log. The overhead might be larger if also actions not changing the database are saved. To do undo correctly it must be known exactly how the update changes the database. If the predicate is not idempotent, class c6 for example, it must also be known exactly how the database is changed. How is this done correctly and most efficiently? To find out if the update changed the database or not, the database is queried about the update. For example if the update is $+T(2)$, the query about the update is the question “is $T$ true for $x = 2$?”. The answer to a query like this is none, one or several propositions. If the result shows that the change which would be the result from the update is already in the database, the action is not logged. For example, if the query in the example above gives the result that $T$ is true for $x = 2$ then it is known that the update will not affect the database and the action is not logged. If the result instead is that $T$ is false for $x = 2$, there will be a change to the database and the difference the updates makes should be logged. Depending on which class the predicate belongs to, the difference has to be expressed in different ways. If the predicate belongs to class c1 or c8 a
4.5. LOGGING

A predicate and a command is enough to describe the difference, for example, \(+T(2)\).

If an update updates many predicates, maybe only some of them will actually change the database. If that is the case, there are two options: to find out which propositions that will change the database and log only these, or to log the command and the answer to the query and the predicate and find out before doing redo/undo what actually changed in the database.

If a predicate expression is non-idempotent, other information than the predicate expression must be logged. The result of executing such a predicate depends on the set it is executed on. The difference the expression makes is the answer to the query about the predicate. It is the difference and a command that should be logged.

Therefore there are two kinds of logging information that can be supported by the log.

1. A command and one or several propositions describing the change of the database.
2. A command, a predicate and the answer to a query.

All classes can be logged using method 1. For method 1 the saved facts are for class c2-c6, the answer to the query about the change. It is only when it is more efficient to use more space and not investigate the answer that method 2 is used or when the original predicate is more efficient to process than the propositions. The logging method 1 is for class c1 and c8 equivalent to the logical logging described in [7]. The only difference is that a proposition and a command is logged instead of values and DML commands. The fact that propositions and commands are logged makes it easy to find the inverse actions. Therefore it is easy to use the log records for both logical redo and undo. For method 2, the predicate and command is used for redo and the propositions and the opposite command is used for undo.

Querying the database about an update takes time \(T_1\). Executing the same update takes time \(T_2\). The database is queried about every update to get correct log information. If the update executes the same predicate expression as the query was about, the total time to log and execute a query will be \(T_1 + T_2 + k*n\), where \(n\) is the number of propositions and \(k\) some constant. \(k*n\) is the time it takes to log \(n\) propositions. The query about the update gives information about how the predicate expression changes the database. If the original predicate expression is complex the information can be used to construct a simpler predicate expression that is faster to process than the original one. By executing the simpler expression, time is saved. The simpler expression is a predicate expression describing a set of propositions. The increase in time is therefore proportional to the number of propositions affected by the update. The maximum increase in execution time due to the logging is \(n*(k+t), n*t < T_2\), where \(t\) the time it takes to process one proposition. So even for very complex predicates that are expensive to execute the time increase can be relatively small.
Chapter 5

Conclusions

5.1 Result

5.1.1 Locking

A predicate locking system can be implemented. The basic criterion for predicate locking is that it must be possible to create predicates representing queries and updates in the system. The granularity of the predicate locking system can be arbitrarily fine. There must exist a log that can log as fine changes of the database as the locking system locks. Such a log exists, see section logging below.

The SS2PL protocol can be used in a predicate locking system, see section 4.1.3. A theorem for how total independence between actions can be investigated has been found (Theorem 4.1.3).

There are two strategies, "Check Others’ Locks” and "Check Own Locks First”, to investigate if a new action does not conflict with existing locks, see section 4.1.7. If the same transaction uses the same predicate expression repeatedly and has few other locks strategy "Check Own Locks First” is better. If transactions seldom use the same predicate expressions several times strategy "Check Other’s Lock”s is better. Strategy "Check Own Locks First” is also better when the number of transactions is large.

Whether two arbitrary predicate expressions are dependent or not is an NP-complete problem [2]. Precompilation makes it possible to solve parts of the NP-complete problem in advance and reduce the time complexity of solving the remaining problem in real time, see section 4.1.9. To be able to precompile the kinds of queries and updates allowed by a system must be known in advance and an optimizer must be able to simplify some predicate expressions. The optimizer can simplify many simple predicate expressions if it knows the rule described in section 4.3.

Testing locks on extensional predicate level has time complexity $O(n)$, see section 4.1.6.

Two algorithms one with precompilation and one without have been developed see section 4.1.8 and 4.1.9. The one without precompilation has a high time complexity. The time complexity for the algorithm using precompilation depends on what sort of expressions the optimizer can simplify. If the optimizer is not perfect the complexity also varies with the complexity of the predicates and how exact the conflict test needs to be. A conflict can be exactly tested but the time complexity of it might be high. The test can be done in constant or linear time if the exactness of it is sacrificed.

The cost of conflict testing depends on the number of taken locks and not on the size of the database. This was considered to be a good characteristic by [16].

The time of lock testing also increases with the number of active transactions, the number
of possible actions and most important with the complexity of the actions. Predicate locking works well with systems with simple updates and queries or few updates and many queries. It is also good for systems with few queries and many updates, if the updates are of some of the classes c1-c5 and c8-c10. A system with many complex and not predefined actions, many transactions and a small database is the least compatible with predicate locking.

The system can handle all sorts of actions normal to a database. It must be considered as usable until the opposite has been proved.

5.1.2 Logging

It is possible to have a logical logging system. The logging system in this thesis has as fine granularity as the locking system.

The log implemented in this thesis can not only be used with the predicate locking system described. It would be possible to use with other systems needing such a log.

Both undo and redo can be done logically. For the logical operations to work, the database must be in a consistent state. Consistency is achieved through complete, consistent checkpoints or atomic propagation. One can choose to have physical redo together with logical undo.

To make sure that the expressions used to redo and undo are idempotent, it must be known how the updates change the database.

The classes of add and remove (see section 4.5.2) can be logged in two different ways. They are:

1. Log a command and one or several propositions describing the change of the database.

2. Log a command, a predicate and the answer to a query.

Which classes that should be logged with which method is found in section 4.5.2. Not all classes are supported by a system. Some classes can not be supported and other classes might not be supported because they are considered too expensive to log, see section 4.5.2

When propositions are logged, predicates have to be created from the propositions.

The execution cost of the logging is proportional to the number of propositions the update affects.

The thesis shows that predicate logging is possible to use under certain conditions.

5.2 Discussion

5.2.1 Locking

The performance of the predicate locking system is intimately tied to how good the optimizer is. A predicate locking system is developed gradually as the optimizer gets better and better. Even if the optimizer can not simplify an expression it does not stop working because it can then either execute the expression to get the exact answer to a greater cost or take the defensive decision to suppose a conflict and lock.

Logical locking in a logical system makes it possible to analyse and optimize the system as one unit. The logical locks are of the same type as the expressions being executed. As the same optimizer is used by the database system the whole system will be improved as the optimizer is improved. Once the system is implemented, the only thing that needs to be worked on is the optimizer.
5.3 Future Work

5.3.1 Locking

Among the most important things left for future works is to investigate the exact space and time demands for a predicate locking system. All aspects of such an implementation ought to be compared to physical locking methods.

At the moment, updates of existing propositions are not supported by the predicate language. One way to update could be to query about the existence of the proposition that should be changed and then remove it. After it has been removed a new proposition should be added. A predicate expression used to add and remove propositions can depend on different actions than a predicate expression used to change a proposition. As the actions are equivalent, their dependencies should also be equivalent. A method for solving this problem would be interesting to investigate.

Another interesting aspect left for the future is to investigate how second order logic can be supported by a predicate oriented database and how second order logic will affect the predicate locking system. Is it possible to implement it?

What rules an optimizer needs to support to simplify arbitrary predicate expressions is an interesting subject to study. It would also be valuable to know in which cases a double locking system could be practically useful and what expressions that are not too expensive to execute in such a system.

Another idea is to study which operators an optimal predicate language for databases should consist of.

Yet another idea to investigate is perhaps if it would be useful to create indexes for the predicate locking system. During precompilation it can be investigated which data that needs to be accessed during execution from the database.

5.3.2 Logging

A possibility with this logical log is that it can be used for multiversion. In a multiversion system, a transaction can ask for an old revision of the database. That would be easy to do with this log, it is just to apply the correct undo or redo actions until the wanted revision is reached.

Using multiversion can be done without having data structures. The current revision would then be the first, empty one and then the new ones would be acquired through redoing from the log. For this to work, all log records need to be stored.

It would be interesting to investigate the time and space costs more thoroughly and find out how good the logical logging is compared to physical logging.
Bibliography


