# Detecting synchronization conflicts for horizontally decentralized relational databases

JAN-HENK GERRITSEN

JULY 2012

CHAIR FOR DATABASES (DB)
FACULTY OF ELECTRICAL ENGINEERING, MATHEMATICS AND COMPUTER SCIENCE (EEMCS)
UNIVERSITY OF TWENTE, THE NETHERLANDS

SUPERVISORS
DR. A. WOMBACHER
DR. M.M. FOKKINGA

# ABSTRACT

A horizontally decentralized relational database consists of multiple separate databases that have the same database schema. These separate databases are referred to as local databases. In a horizontally decentralized database the structure of the data is the same at all local databases, only the actual data can be different. The goal of this master thesis was to develop a solution which identifies synchronization conflicts when synchronizing two local databases of a horizontally decentralized relational database. These synchronization conflicts are identified by comparing the operations executed at the local databases and not by comparing the database states.

Two local databases are in a synchronized state if the database states of these two local databases are equal and do not violate any of the integrity constraints defined in the database schema. This master thesis investigated in which ways different combinations of operations could lead to synchronization conflicts based on unequal database states or violations of integrity constraints defined in the database schema. The results of this investigation were formalized in a set of theorems that can be used to identify synchronization conflicts for sequences of operations executed at two local databases.

A conflict detection solution was implemented based on the conflict detection theorems. This conflict detection solution was validated against a set of test scenarios executed at a test environment of the Dation Dashboard application. Dation Dashboard is an information system for driving schools developed by the company Dation. This validation identified some problems with the approach that should be investigated further in future research.

One of the problems identified for the conflict detection solution in its current form are that there are some database schema practices that can lead to the detection of a lot of unnecessary conflicts. Another problem that was identified is that a single action of a user can lead to multiple database operations, and for some of these operations the conflict detection solution can detect conflicts while no conflict is expected from the user's point of view. The principal conclusion of this master thesis is that the conflict detection solution proposed in this master thesis seems promising, but there are several problems that should be addressed before it can be used in a production environment.

# TABLE OF CONTENTS

# 1 INTRODUCTION

This master thesis describes the work done for an external graduation project at Dation. Dation is a company located in Enschede that supplies software for driving schools. Their major application is an administration system called Dation Dashboard. Dation has a mobile application that provides access to Dation Dashboard, however, this mobile application is web-based, so this mobile application does not work if there is no connection to the internet available.

To overcome this problem the mobile application can store a local copy of a subset of the data of Dation Dashboard. Users can now use the mobile application even if there is no connection to the internet available, although this leads to a new problem. Changes made to the local data of the mobile application should eventually be synchronized with Dation Dashboard. Automatic synchronization of the data on the mobile devices and the data of the Dation Dashboard application is the topic of this master thesis. Because Dation's applications store their data in a relational database, the data synchronization researched in this master thesis applies to relational databases. However, the solution presented in this master thesis must work for relational databases in general and not only for the relational database schemas used in the applications of Dation.

As said before, the mobile applications use a local copy of the data of a subset of the data of the Dation Dashboard application. This approach is referred to as a horizontally decentralized relational database. In a decentralized database system the data can be stored at several different physical locations. The relational databases at these locations can have different relational database schemas, however, in a horizontally decentralized database the relational database schema is the same at all locations, which means that the data is organized in the same manner, but the actual data can still differ (Miha, 2008).

A benefit of using a horizontally decentralized relational database for the purpose of discussing synchronization is that the data that must be synchronized is organized in the same manner at all locations. This means that all problems that can occur when merging different relational database schemas do not have to be considered.

For a horizontally decentralized relational database the relational database schema is the same at all locations, but in this master thesis a relaxed definition of a horizontally decentralized relational database is used. The relations that take part in the synchronization must have the same structure in their respective local relational schemas, but it is possible for the relational schemas at the different locations to contain additional relations that are missing from the relational schemas at the other locations. These relations are ignored when attempting to synchronize the data of two locations.

Before starting the research into synchronization of two horizontally decentralized databases the concept of synchronization must be defined. Two locations are in a synchronous state if the data in the relations that take part in the synchronization process is the same at both locations. However, this definition is insufficient. Relational databases have properties, for example integrity constraints, that must not be violated. If one of these properties is violated, the relational database is in an inconsistent

state, which is not allowed. From this follows that synchronization is achieved if the relations that take part in the synchronization contain the same data at both locations, and the local relational databases are in a consistent state.

Because synchronization requires that the local databases are in a consistent state, the properties that can be violated and leave the database in an inconsistent state have to be identified. Then all the ways in which the operations that change the data can violate these properties must be investigated. This must be done for single operations, but also for groups of operations that belong together and have the requirement that all these operations must be performed or none of them at all. These groups of operations are referred to as composite operations in this master thesis.

The knowledge of the ways in which operations and composite operations can violate the properties of a relational database can be used to create an automatic synchronization solution. However, developing such an automatic synchronization solution is too much work to cover in this master thesis and is left as future research.

This master thesis focusses on detecting the conflicts that can occur when synchronizing two local databases. The solution proposed in this master thesis is used in an implementation for detecting conflicts between two local databases with the database schema of the Dation Dashboard application. This implementation acts as the validation step of this master thesis.

## 1.1 BACKGROUND

This section contains a general overview of Dation the company and a high-level description of the Dation Dashboard system.

### 1.1.1 DATION THE COMPANY

Dation is a fast growing young company that is a supplier of administration and service systems for driving schools. Since its formation in 2004 the company has grown out to be market leader with hundreds of satisfied customers throughout the Netherlands. Dation consolidates its position as market leader by continuous innovation and expansion of offered services. Dation is an informal and dynamic company and consists of a small team of enthusiastic young people.

Dation delivers two products, Dation Dashboard and Dation Rijlesplanner[1]. Dation Dashboard is an administration system for driving schools. The next subsection describes Dation Dashboard in more detail, following the short description of Dation Rijlesplanner below.

Dation Rijlesplanner is a web portal that aims at improving communications between a driving school and their students. The main features of Dation Rijlesplanner are for students to:

- plan driving lessons or theory lessons.
- view a course overview (e.g. planned lessons, statistics, number of lessons left).

[1] Rijlesplanner is a Dutch word that means driving lesson planner

- view a financial transactions overview.
- update their personal information.

### 1.1.2 THE DATION DASHBOARD APPLICATION

Dation Dashboard has originally been developed as a relatively simple web-based administration system for driving schools. The main functions of Dation Dashboard are what to expect of any administration system:

- A customer database.
- An agenda for planning students, lessons and instructors.
- Financial administration.

Over the years the system expanded considerately in functionality. Besides the administrative functionality numerous extra services were added:

- A mobile application that provides access to Dation Dashboard for driving instructors in the cars.
- Elaborate access rights for different types of users.
- Quite a few links with other third-party systems.

Dation Dashboard is implemented in PHP running on a web server, and uses a MySQL relational database for data storage.

## 1.2 GOAL STATEMENT

The goal of this graduation project is to develop a conflict detection solution to detect the conflicts that can occur when synchronizing two local databases of a horizontally decentralized relational database. An implementation of this solution will be validated against the database schema of the Dation Dashboard application.

The conflict detection solution must have the following properties:

- It should use the operations executed at the local database to detect conflicts and not compare database states.
- It should identify conflicts that will lead to unequal databases states after synchronization.
- It should identify conflicts that will lead to violations of integrity constraints defined in the database schema after synchronization.

## 1.3 RESEARCH QUESTIONS

The goal of this master thesis is to develop a conflict detection solution which can detect synchronization conflicts between two local databases of a horizontally decentralized relational database, which leads to the following research question:

> *How can synchronization conflicts be detected for the synchronization of two local databases of a horizontally decentralized relational database?*

To answer the main research question the following sub-questions have to be answered:

1. *Which properties of horizontally decentralized relational databases can be violated during synchronization?*
2. *How can atomic operations violate these properties during synchronization?*
3. *How can composite operations violate these properties during synchronization?*
4. *How can synchronization conflicts be detected in an automated way?*

## 1.4 APPROACH AND THESIS STRUCTURE

This section describes the approach taken in this master thesis to answer the research questions. This approach is visualized in Figure 1.1. The remainder of this section explains this approach in more detail.

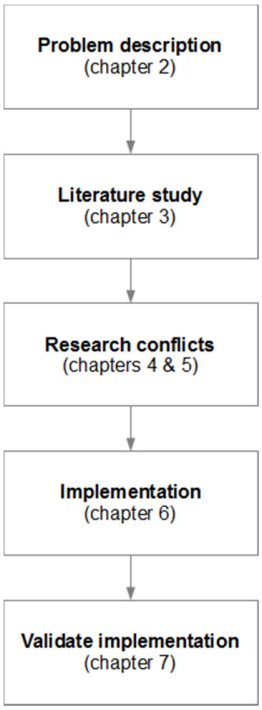


FIGURE 1.1 APPROACH

Chapter 2 starts with a problem description that describes the synchronization scenario for which this master thesis proposes a conflict detection solution. This chapter explains what is meant with a horizontally decentralized relational database and also describes the synchronization approach for which the conflict detection solution proposed in this master thesis should work.

Chapter 3 contains a literature study into the properties of relational databases that can be violated during synchronization. This chapter answers the first research question. It gives an overview of the relational model in section 3.1 and an overview of integrity constraints in section 3.1.1. Section 3.2 gives an overview of relational algebra, because relational algebra is used in other chapters to reason about database operations. Section 3.3 concludes chapter 3 by stating which properties of relational databases will be guaranteed by the proposed conflict detection solution.

Chapter 4 investigates how atomic operations can violate the properties of relational databases that were stated in section 3.3. This chapter answers the second research question. It starts with a definition of a conflict for two atomic operations in section 4.1 and a definition of the atomic operations insert, delete and update in section 4.2. The definition of a conflict consists of two parts. The first part states that there is a conflict if the two atomic operations lead to unequal database states after synchronization and the second part of the conflict definition states that there is a conflict if the two atomic operations lead to an inconsistent database state. Conflicts for atomic operations that are based on unequal database states after synchronization are investigated in section 4.4 and conflicts based on the violation of integrity constraints are investigated in section 4.5. Chapter 4 concludes with section 4.6 which presents three decision graphs that can be used to detect conflicts for two arbitrary atomic operations.

Chapter 5 investigates how composite operations can violate the properties of relational database. This chapter answers the third research question. Section 5.1 defines composite operations and conflicts for composite operations and also investigates how conflicts between composite operations can be identified.

Atomic and composite operations can also be related to each other by the data items they modify. If an atomic or composite operation cannot be synchronized because it conflicts with another operation it is possible that this will also affect related operations. These dependencies between atomic and composite operations are discussed in section 5.2. Chapter 5 concludes with a description of the composite operations conflict graph in section 5.3. The composite operations conflict graph is a graph that can be used to identify conflicts and dependencies for two sequences of composite operations executed at two different local databases. To create a composite operations conflict graph all theorems presented in chapters 4 and 5 are used.

Chapters 6 and 7 will answer the fourth research question. Chapter 6 describes the implementation of the conflict detection solution, and this implementation is used in the validation of the conflict detection solution in chapter 7. The validation of the conflict detection solution analyzes how the implementation of the conflict detection solution behaves in a set of sample scenarios executed against a test environment of the Dation Dashboard application.

# 2 PROBLEM DESCRIPTION

The synchronization problem studied in this master thesis applies to horizontally decentralized databases. A horizontally decentralized database consists of multiple separate databases, referred to as local databases. Each of these local databases can run on different devices on different physical locations, but it is also possible that some of these local databases run on the same device. What matters is that these local databases are separate databases.

In a decentralized database each local database has its own database schema. However, in a horizontally decentralized database each local database has the same database schema as all the other local databases. This means that the data is structured the same at all local databases, but that the actual data can be different at each local database.

In this master thesis a relaxed definition of a horizontally decentralized database is used. The local database schemas can contain relations that are not in the database schemas of the other local databases, but as a consequence the data in these relations cannot be synchronized. Only relations that the local databases have in common can be synchronized.

A decentralized database differs from a distributed database because there is no sharing of data through a communications network. The difference between a decentralized and a distributed database is visualized in Figure 2.1 (Padigela).

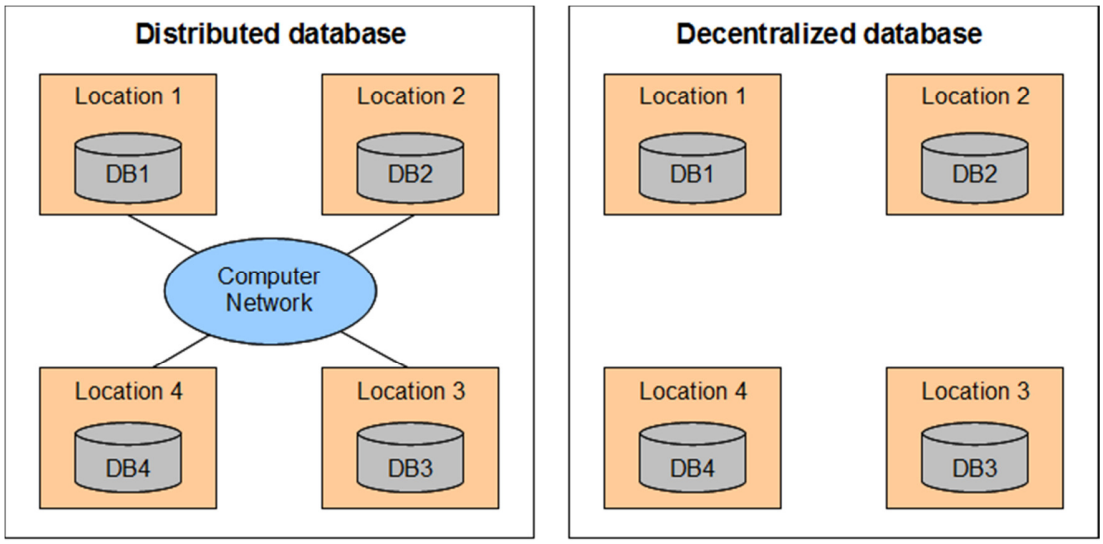


FIGURE 2.1 DISTRIBUTED DATABASE VERSUS DECENTRALIZED DATABASE

In a distributed database all local databases are connected with each other through a communications network and each local database contains its own data, but all the local databases work together as if the data is stored at a single location. When data is requested from a distributed database the local databases in a distributed database work together to deliver the requested data. The data is delivered

from a local database that contains the requested data. In a decentralized database the data must be requested at the local database that contains the data, because the local databases do not work together.

If data from one local database must be made available at another local database in a decentralized database, the data must be synchronized between these two local databases. Detecting conflicts that can occur during synchronization of local databases is the problem researched in this master thesis.

As stated in the previous paragraph, synchronization is performed between two local databases, from here on denoted as DB1 and DB2. It is assumed that these two local databases start out in a synchronous state at the start of the scenario. A database state is a snapshot at a point in time in which all attributes of all the tuples in all the relations in the relational database are assigned a value. Two databases are in a synchronous state if the database states of these two databases are equal.

The synchronization scenario can now be described as follows:

- Both DB1 and DB2 have the synchronous database state S at the start of the scenario, which is denoted as time $T_0$.
- At time $T_1$ the local databases DB1 and DB2 must be synchronized.
- Between $T_0$ and $T_1$ sequences of operations are performed at both local databases.
- At both locations a sequence of operations is executed, denoted as $op_{L,i}$, where L identifies the location and i identifies the position of the operation in the sequence.
- At DB1 a sequence of n operations is executed, at DB2 a sequence of m operations is executed.
- Each operation transforms an existing database state into a new database state. These states are denoted as $S_{L,i}$, where L refers to the location and i refers to the position of the operation in the sequence of operations that lead to this state.

This scenario is visualized in Figure 2.2. At time $T_1$ the database states $S_{1,n}$ and $S_{2,m}$ must be synchronized. The goal of this synchronization is that both local databases end up with the same database state, in which the operations that were performed at both local databases are reflected. If it can be established that a synchronization solution always leads to a synchronous database state, synchronization can always be achieved when the local databases start out in a synchronous state.

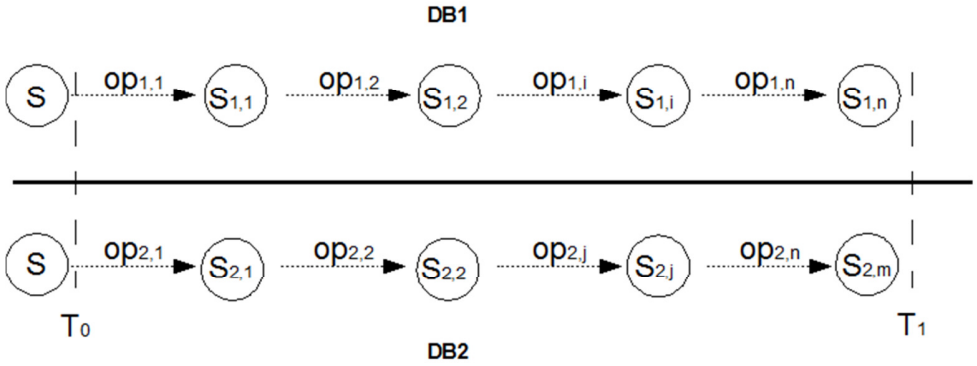


FIGURE 2.2 THE SYNCHRONIZATION SCENARIO

A possible approach for a synchronization solution is to compare all intermediate database states and try to reach a synchronous database state from these comparisons. But keeping track of all intermediate database states is infeasible, especially for large databases. This is why the approach researched in this master thesis focuses on a synchronization solution based on investigating the sequences of operations that are executed at both local databases.

The first problem that must be addressed is whether two atomic operations harmonize or conflict. Two atomic operations harmonize if the effects of these two operations do not interfere with each other. Atomic operations can conflict with each other in two ways: they can lead to unequal database states, or they can leave the database in an inconsistent state. A database is in an inconsistent state if the constraints that are imposed on this database are violated. This means that all the possible database constraints that can be violated must be investigated.

In the synchronization scenario sequences of atomic operations are executed at both local databases. As with harmonization between two atomic operations, it must also be investigated whether these sequences of atomic operations harmonize. Atomic operations can be related to each other through the data items they modify. Applying these related atomic operations in a different order can lead to different results for these data items, which is not acceptable for synchronization.

Besides the relations between atomic operations based on the data items they modify, atomic operations can also be semantically related. Semantically related atomic operations are grouped together in a composite operation. The concept of a composite operation for synchronization is similar to the concept of transactions in database systems. A synchronization solution must also ensure atomicity for composite operations: either all the atomic operations in the composite operation must be synchronized or none of them at all.

Identifying the conflicts that can occur between operations is the first part of the synchronization problem, and this is the part that is covered in this master thesis. The second part of the synchronization problem is how to handle and resolve these conflicts in an automated way. However, this will not be covered in this master thesis.

# 3 PROPERTIES OF RELATIONAL DATABASES

In 1970 E.F. Codd proposed the relational data model and even today most commercial database management systems are based on the relational model (Codd, 1970). So to be able to reason about the properties of relational databases that can be violated during synchronization, the relational model must be investigated. This is done in the first two sections of this chapter. Section 3.1 discusses the relational model and section 3.1.1 discusses integrity constraints in detail. The content of these sections is based on the work of Kifer, Bernstein and Lewis, *Database Systems An Application-Oriented Approach* (Kifer, 2006). The formalizations used in these sections are based on the work of Özsu and Valduriez, *Principles of Distributed Database Systems* (Özsu, 1999). Section 3.2 discusses relational algebra. Relational algebra is used to express the operations that can be performed against a relational database. This chapter concludes with section 3.3 which provides an overview of the properties of relational databases that will be guaranteed by the synchronization solution proposed in this master thesis.

## 3.1 THE RELATIONAL MODEL

The most important concept in the relational model is the concept of a relation. A relation consists of two parts: a schema and an instance. The schema can be regarded as the definition of the relation, and the instance contains data conforming to the definition in the schema. When no confusion can arise, it is common to use the term relation to refer to a relation instance.

A relation schema consists of three parts:

1. The name of the relation, which must be unique within a database.
2. A list of attributes names, and the domains of these attributes.
3. Integrity constraints, which are restrictions on the relational instances of this schema. Integrity constraints are discussed in more detail in section 3.1.1.

A relational schema with the name $RS$ can now be defined as

$$RS\ (A_1:D_1,\ A_2:D_2,\ \ldots,\ A_n:D_n)$$

where each $A_i$ is an attribute name and each $D_i$ is the corresponding domain for attribute $A_i$. The domain is a well-defined set of values, for example the domain of all Integer values or the domain of all Strings.

A relation instance is a table with rows and named columns. The named columns correspond with the attributes of the relation schema. The rows in this table are called tuples. A relation instance $R$ is defined as a set of $n$-tuples $R = \{t_1,\ t_2,\ \ldots,\ t_k\}$. The number $k$ is called the cardinality of a relation, which is the number of tuples in the relation. Each $n$-tuple $t_i \in R$ is an ordered list of values, $t_i = <v_1,\ v_2,\ \ldots,\ v_n>$ such that $v_1 \in D_1,\ v_2 \in D_2,\ \ldots,\ v_n \in D_n$, with each $D_i$ corresponding to the domain of the attribute from the relation schema. The number $n$ is called the arity of the relation, which is the number of attributes of the relation.

A relational database is defined as a finite set of relations. Because relations consist of a relation schema and a relation instance, there are also a database schema and a database instance. As with relations, when no confusion can arise, the term database refers to the database instance.

### 3.1.1 INTEGRITY CONSTRAINTS

An integrity constraint is a statement about all legal instances of a database. To be qualified as a legal database instance a set of relations must satisfy all integrity constraints associated with the database schema.

Two distinctions can be made between the different types of integrity constraints. The first distinction is between intra-relational and inter-relational integrity constraints. An intra-relational integrity constraint is defined for a single relation, while an inter-relational integrity constraint is defined over multiple relations.

The second distinction is between static and dynamic integrity constraints. Static integrity constraints restrict the legal instances of a database, but dynamic integrity constraints restrict the evolution of legal instances. Static integrity constraints are expressed in the database schema, while dynamic integrity constraints are used mainly to express business rules and are implemented at the application level, not at the database level.

Because dynamic integrity constraints are mostly implemented at the application level, this chapter only focuses on static integrity constraints. These can be categorized in the following four groups:

- type constraints
- key constraints
- referential integrity constraints
- semantic constraints

These four types of integrity constraints are covered in the remainder of this section.

### 3.1.2 TYPE CONSTRAINTS

Each value in a tuple in a relation instance must correspond to an attribute of the relation schema. These values must belong to the domain that is associated with that attribute. If this is not the case, there is a type constraint violation. Local databases ensure that the type constraints are not violated, so type constraints can be disregarded when discussing synchronization between two databases.

### 3.1.3 KEY CONSTRAINTS

A superkey `SK` of a relation `R` is a set of attributes of `R` that can be used to uniquely identify tuples in `R`. In other words, there cannot be two distinct tuples with the same values for all the attributes of `SK` in `R`:

```
A set of attributes SK is a superkey of relation R with attributes A, when SK
⊆ A and ¬∃ t1, t2 ∈ R for which t1[SK] = t2[SK]
```

The expressions t1[SK] and t2[SK] are the projections of the attributes of SK on the tuples t1 and t2. Projection is discussed in section 3.2 which covers relational algebra. The expression t1[SK] = t2[SK] compares the values for the attributes of SK of both tuples. This expression is valid if the tuples t1 and t2 have the same values for all corresponding attributes.

A key K of relation R is a superkey of R with the additional property that there does not exist another superkey SK for which SK ⊂ K.

A relation is a set, and sets cannot contain identical tuples, so this means that relations cannot contain identical tuples. From this follows that the set of all attributes of a relation schema is a superkey for that relation, because it can uniquely identify a tuple. This also means that any relational schema has at least one key. When a relational schema has multiple keys, each of these keys is called a candidate key. Exactly one of these candidate keys is designated as the primary key of a relation.

### 3.1.4 REFERENTIAL INTEGRITY CONSTRAINTS

In relational databases tuples in a relation can reference tuples in the same or other relations. When there is a requirement that the referenced tuples must exist, this is a referential integrity constraint. Two kinds of referential integrity constraints exist, foreign key constraints and inclusion dependencies. Foreign keys are discussed first, because the definition of an inclusion dependency is a relaxed form of the definition of a foreign key.

For two relations S and T, with X a set of attributes from S and K a set of attributes from T, and K a key for relation T, the relations S and T satisfy the foreign key constraint S(X) REFERENCES T(K) if for every tuple s ∈ S there exists a tuple t ∈ T that has the same values for the attributes of the key K as the tuple s has for the corresponding attributes in X. Relation S is called the referencing relation, relation T is called the referenced relation, attribute set X is called the referencing attribute set and attribute set K is called the referenced attribute set. Each tuple in S must reference exactly one tuple in T, but not all tuples in T have to be referenced by a tuple in S, and a tuple in T can also be referenced by multiple tuples in S.

As stated before, the definition of an inclusion dependency is a relaxed form of the definition of a foreign key constraint. An inclusion dependency is defined as a foreign key constraint, only without the requirement that the referenced attribute set is a key in the referenced relation. This means that a tuple in the referencing relation of an inclusion dependency can reference multiple tuples in the referenced relation of the inclusion dependency.

One important aspect regarding referential integrity constraints is the fact that you cannot simply delete tuples from a relation that acts as a referenced relation in some referential integrity constraint, because this could violate these referential integrity constraints. Two approaches exist to maintain referential integrity: when deleting a referenced tuple all referencing tuples must also be deleted, or just do not allow to delete tuples that are still referenced by other tuples. Both approaches are enforced locally by a relational database.

### 3.1.5 SEMANTIC CONSTRAINTS

Key constraints and referential integrity constraints deal with the structure of the data. Semantic constraints do not deal with the structure of the data, but instead reflect business rules or conventions from the particular application domain being modeled by the database. A well-known textbook example is a constraint that states that the number of students registered for a course cannot exceed the number of seats in the classroom that is assigned to the course. Another example is a constraint that states that a zip code belongs to exactly one street and city. Common approaches for specifying semantic constraints for relational databases are the use of functional dependencies and assertions.

A functional dependency FD, expressed as $X \rightarrow Y$, is an intra-relational constraint on two sets of attributes X and Y of a relation schema RS. X is called the determinant attribute set and Y is called the dependent attribute set. A relational instance R of the relational schema RS satisfies the functional dependency FD when for every pair of tuples $t, s \in R$, if t and s agree on all attributes in X, then t and s agree on all attributes in Y. Key constraints are a special case of functional dependencies, because the values for the key attributes in a tuple uniquely define the values of all the other attributes in the tuple.

Assertions are semantic constraints that are expressed as conditional expression over the contents of a database. When this conditional expression evaluates to false, the constraint is violated. Assertions can serve as intra-relational or inter-relational constraints.

## 3.2 RELATIONAL ALGEBRA

This section introduces relational algebra. Relational algebra consists of a set of operators that operate on relations, and relational algebra is derived from set theory (relations are sets). Each operator takes one or two relations as its operands and produces a result relation, which in turn may be an operand to another relational algebra operator. In this master thesis relational algebra is used to express the operations performed against a database.

There are five fundamental relational algebra operators and five other operations that can be defined based on these fundamental operators. The fundamental operations are *selection*, *projection*, *union*, *set difference* and *Cartesian product*. Selection and projection are unary operators and the other three are binary operators. The additional operators that can be defined based on the definitions of the fundamental operators are *intersection*, *θ-join*, *natural join*, *semi-join* and *quotient*.

For binary relational algebra operators the operands should be union compatible. Two relations R and S are union compatible if and only if they are of the same degree and the *i*th attribute of both relations is defined over the same domain.

The remainder of this section discusses the selection, projection, union, set difference and Cartesian product operators. For more details on relational algebra take a look at (Kifer, 2006), (Özsu, 1999) or (Relational Algebra).

#### 3.2.1.1 SELECTION

The selection operator produces a horizontal subset of a given relation. The subset consists of all the tuples that satisfy a formula $F$ in a relation $R$.

*Notation*

$\sigma_F(R)$

$F$ is a formula whose terms are of the form $A\theta c$ or $A\theta A$, where $A$ is an attribute of $R$, $\theta$ is one of the arithmetic comparison operators $<$, $>$, $=$, $\neq$, $\leq$, $\geq$ and $c$ is a constant. The terms can be connected by the logical operators $\wedge$, $\vee$ and $\neg$.

#### 3.2.1.2 PROJECTION

The projection operator produces a vertical subset of a given relation. The subset contains only those attributes of the original relation R over which projection is performed. Thus the degree of the result relation is less than or equal to the degree of the original relation.

*Notation*

$\pi_{A,B}(R)$

$A$ and $B$ are the attributes of relation R that should remain in the resulting relation after the projection is performed.

In this master thesis projection is also used on the tuple level. For a tuple $t$ of a relation $R$ and a subset $X$ of the attributes of relation $R$, the operation $t[X]$ performs the projection of the attributes in $X$ over the tuple $t$. Only the attributes in $X$ remain in the resulting tuple.

#### 3.2.1.3 UNION

The union of two relations $R$ and $S$ is the set of all tuples that are in $R$, or in $S$ or in both. $R$ and $S$ should be union compatible. Union can be used to insert a new tuple into an existing relation, where the new tuple acts as one of the operands of the union operator.

*Notation*

$R \cup S$

#### 3.2.1.4 SET DIFFERENCE

The set difference of two relations $R$ and $S$ is the set of all tuples that are in $R$ but not in $S$. $R$ and $S$ should be union compatible. Set difference can be used to delete a tuple from an existing relation, where the tuple that must be deleted acts as the right-hand operand of the set difference operator.

*Notation*

$R - S$

Set difference is asymmetric, which means that $R - S \neq S - R$.

#### 3.2.1.5 CARTESIAN PRODUCT

The Cartesian product of two relations, relation $R$ with degree $k_1$ and relation $S$ with degree $k_2$, is the set of ($k_1$ + $k_2$)-tuples, where each result tuple is a concatenation of one tuple of $R$ with one tuple of $S$, for all tuples of $R$ and $S$.

*Notation*

$R \times S$

## 3.3 OVERVIEW OF GUARANTEED PROPERTIES

This section provides an overview of the properties of relational databases that will be guaranteed by the synchronization solution proposed in this master thesis. To delimit the work for this master thesis, only the static integrity constraints that deal with the structure of the data in the database will be considered:

- Key constraints
- Functional dependencies
- Foreign key constraints
- Inclusion dependencies

# 4 SYNCHRONIZATION OF ATOMIC OPERATIONS

This chapter investigates the ways in which atomic operations can violate the properties of a relational database during synchronization. The atomic operations investigated are the manipulative operations that change a database: insert, delete and update. This chapter focuses on all combinations of these operations.

The investigation in this chapter focuses on a small part of the scenario described in chapter 2. This is the part shown in figure 4.1. There are two locations taking part in the synchronization, DB1 and DB2, and both these locations start out in the synchronous state S at time $T_0$. At both locations a single atomic operation is performed, $op_{1,1}$ at DB1 and $op_{2,1}$ at DB2. This leads to the new states $S_{1,1}$ at DB1 and $S_{2,1}$ at DB2. By comparing the atomic operations $op_{1,1}$ and $op_{2,1}$ it must be determined whether the states $S_{1,1}$ and $S_{2,1}$ can be merged into a single synchronous state. This is done for all the possible combinations of the atomic operations insert, delete and update. How sequences of atomic operations are handled during synchronization is the topic of chapter 5, but the work in chapter 5 depends on the work in this chapter.

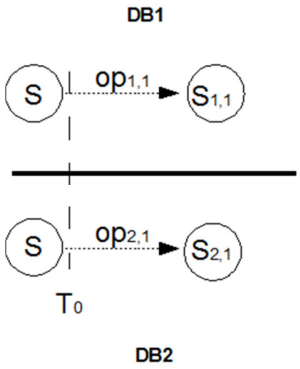


FIGURE 4.1 SYNCHRONIZING TWO ATOMIC OPERATIONS PERFORMED AGAINST THE SAME DATABASE STATE AT TWO DIFFERENT DATABASES.

This chapter starts with the definition of a conflict for two arbitrary atomic operations in section 4.1. This definition is used to determine if a conflict occurs between two arbitrary atomic operations. Section 4.2 presents definitions of the atomic operations insert, delete and update using relational algebra. Section 4.3 presents the example data that is used in the examples throughout this chapter and the next chapters.

The definition of a conflict for two arbitrary atomic operations in section 4.1 identifies two types of conflicts that can occur between two atomic operations. These are conflicts based on unequal database states after synchronization and conflicts based on the violation of integrity constraints of the database schema. Conflicts based on unequal database states after synchronization are investigated in section 4.4 and conflicts based on violations of the integrity constraints of the database schema are investigated in section 4.5.

The integrity constraints that are guaranteed to be preserved by the synchronization solution researched in this master thesis were identified in section 3.3, these are key constraints, functional dependencies, foreign key constraints and inclusion dependencies. These are the integrity constraints for which conflicts are investigated in section 4.5. Section 4.5 is split into two parts, section 4.5.1 investigates conflicts based on violations of intra-relational integrity constraints, which are key constraints and functional dependencies, and section 4.5.2 investigates conflicts based on violations of inter-relational integrity constraints, which are foreign key constraints and inclusion dependencies.

Sections 4.4 and 4.5 present a set of theorems that can be used to identify conflicts between two arbitrary atomic operations. Section 4.6 presents three decision graphs that can be used to determine if there is a conflict between two arbitrary atomic operations. These decision graphs cover all theorems presented in sections 4.4 and 4.5. For two arbitrary atomic operations all three decision graphs must have the outcome that there is no conflict between those two atomic operations to be able to state that there are no conflicts between those two atomic operations.

## 4.1 DEFINITION OF A CONFLICT FOR TWO ATOMIC OPERATIONS

The goal of synchronization is to reach equal and consistent database states between two databases at different locations. So before a definition of a conflict between two atomic operations can be given, definitions of database state and database state equality are needed, as well as a definition of a consistent database state.

#### DEFINITION 4.1.1 DATABASE STATE

A database state is a snapshot at a point in time in which all attributes of all the tuples in all the relations in the database are assigned a value.

#### DEFINITION 4.1.2 DATABASE STATE EQUALITY

Two database states are considered to be equal if all relations contain exactly the same tuples in both databases.

#### DEFINITION 4.1.3 CONSISTENT DATABASE STATE

A consistent database state is a database state for which all integrity constraints are satisfied.

In the scenario of a conflict between two atomic operations investigated in this chapter, both databases taking part in this scenario, $DB1$ and $DB2$, start out with the same database state, denoted as $S$. At each database exactly one atomic operation is executed. These operations are denoted as $op_{1,1}$ for $DB1$ and $op_{2,1}$ for $DB2$. These operations transform the starting database state $S$ at both databases into new database states, which are $op_{1,1}(S)$ for $DB1$ and $op_{2,1}(S)$ for $DB2$. Both operations on their own do not violate any of the integrity constraints of the relational schema, this follows from the fact that both operations were already allowed on the local databases.

To synchronize the two databases, it must be determined if executing the operations $op_{1,1}$ and $op_{2,1}$ at the other databases lead to equal and consistent database states at both databases. To do this, operation $op_{1,1}$ must be applied to $op_{2,1}(S)$ and operation $op_{2,1}$ must be applied to $op_{1,1}(S)$. This leads to the new states $op_{2,1}(op_{1,1}(S))$ and $op_{1,1}(op_{2,1}(S))$. When these states are equal and do not violate any of the integrity constraints of the relational schema, the atomic operations $op_{1,1}$ and $op_{2,1}$

harmonize. When the databases states $op_{2,1}(op_{1,1}(S))$ and $op_{1,1}(op_{2,1}(S))$ are not equal or are not consistent, the two atomic operations conflict with each other. A conflict between two atomic operations can now be defined as follows:

DEFINITION 4.1.4 CONFLICT BETWEEN TWO ATOMIC OPERATIONS

For two arbitrary atomic operations $op_1$ and $op_2$ that do not violate any integrity constraints of the database schema on their own, these two atomic operations conflict during synchronization if for some initial consistent database state S $op_1(op_2(S)) \neq op_2(op_1(S))$, or if $op_1(op_2(S))$ or $op_2(op_1(S))$ violates one or more integrity constraints of the database schema.

This definition of a conflict between two atomic operations consists of two parts. The first part states that there is a conflict between two arbitrary atomic operations $op_1$ and $op_2$ for an initial consistent database state $S$ when the database state $op_1(op_2(S))$ is not equal to the database state $op_2(op_1(S))$. Throughout this master thesis this will be referred to as *a conflict between two atomic operations based on unequal database states after synchronization*.

The second part of definition 4.1.4 states that there is a conflict between two arbitrary atomic operations $op_1$ and $op_2$ for an initial consistent database state $S$ when $op_1(op_2(S))$ or $op_2(op_1(S))$ violates one or more integrity constraints of the database schema. Throughout this master thesis this will be referred to as *a conflict between two atomic operations based on a violation of integrity constraints.*

The approach used in this master thesis to determine whether two atomic operations conflict or not is based on comparing the atomic operations, and not by comparing database states. However, sometimes it will not be possible to determine with certainty whether two atomic operations harmonize or conflict by only comparing the operations themselves. For these cases an additional concept is needed, the concept of a possible conflict:

DEFINITION 4.1.5 POSSIBLE CONFLICT BETWEEN TWO ATOMIC OPERATIONS

For two arbitrary atomic operations $op_1$ and $op_2$ that do not violate any integrity constraints of the database schema on their own, there is a possible conflict during synchronization if it cannot be proved that they do not conflict.

Like the definition of a conflict between two atomic operations, the definition of a possible conflict between two atomic operations can also be based on unequal database states after synchronization or be based on a violation of integrity constraints.

## 4.2 DEFINITION OF ATOMIC OPERATIONS

To be able to reason about conflicts between the atomic operations insert, delete and update during synchronization, definitions of these atomic operations are needed. These operations are expressed in relational algebra. For more details on relational algebra see section 3.2.

Because relational algebra works on sets, the insert, delete and update operations are also set-based. An insert operation adds a set of new tuples to a relation. Delete and update operations remove or

modify a subset of the tuples in a relation that satisfy a condition `C`. This condition `C` is an atomic formula whose terms are of the form `AθC` or `AθA`, where `A` is an attribute of the relation, `θ` is one of the arithmetic comparison operators $<$, $>$, $=$, $\neq$, $\leq$, $\geq$ and `C` is a constant. The terms can be connected by the logical operators $\wedge$, $\vee$ and $\neg$.

However, to keep the discussion of conflicts between two atomic operations in this chapter succinct, the definitions of the atomic operations are restricted to inserting, deleting and updating a single tuple. Atomic operations that insert, delete or modify multiple tuples can be translated into individual operations for each tuple in the set of tuples affected by the operation. To preserve the atomicity of the operation during synchronization, the concept of a composite operation is introduced, which is discussed in chapter 5.

Inserting a new tuple into a relation is the union of this relation and the new tuple:

#### DEFINITION 4.2.1 INSERT OPERATION

For a tuple $t = \langle v_1, v_2, \ldots, v_n \rangle$ and a relation R with relation schema RS $(A_1:D_1, A_2:D_2, \ldots, A_n:D_n)$, with $v_1 \in D_1$, $v_2 \in D_2$, …, $v_n \in D_n$, the insert operation of tuple t into relation R is defined as $R \leftarrow R \cup \{t\}$.

Deleting a tuple from a relation is the set difference of the relation and the tuple to delete. The relation must be placed on the left side of the set difference operator, and the tuple to delete on the right side, because the set difference operator is asymmetric:

#### DEFINITION 4.2.2 DELETE OPERATION

For a tuple $t = \langle v_1, v_2, \ldots, v_n \rangle$ and a relation R with relation schema RS $(A_1:D_1, A_2:D_2, \ldots, A_n:D_n)$, with $t \in R$ and $v_1 \in D_1$, $v_2 \in D_2$, …, $v_n \in D_n$, the delete operation of tuple t from relation R is defined as $R \leftarrow R - \{t\}$.

Updating a tuple in a relation can be expressed as a delete immediately followed by an insert operation. However, the delete immediately followed by an insert must be treated as a single atomic operation to preserve the semantics of the update operation:

#### DEFINITION 4.2.3 UPDATE OPERATION

For a tuple $t = \langle v_1, v_2, \ldots, v_n \rangle$ and a relation R with relation schema RS $(A_1:D_1, A_2:D_2, \ldots, A_n:D_n)$, with $t \in R$ and $v_1 \in D_1$, $v_2 \in D_2$, …, $v_n \in D_n$, the update operation of tuple t to tuple $t' = \langle v'_1, v'_2, \ldots, v'_n \rangle$ with $v'_1 \in D_1$, $v'_2 \in D_2$, …, $v'_n \in D_n$ is defined as $R \leftarrow R - \{t\} \cup \{t'\}$.

## 4.3 EXAMPLE DATA

The example data presented in this section is used throughout this chapter and the chapters that follow to illustrate some of the presented theorems. The example data consists of two relations, `Student` and `Instructor`. The schema of the `Student` relation is:

```
Student (
      id: Integer, instructor: Integer, first_name: String, last_name:
      String, street: String, number: String, city: String,
      zip_code: String, balance: Decimal
)
```

The relation schema of the `Instructor` relation is:

```
Instructor (
      id: Integer, first_name: String, last_name: String
)
```

Table 4-1 and table 4-2 show the example data of the `Student` and `Instructor` relations in their initial synchronized state.

| id | instructor | first_name | last_name | Street | number | city | zip_code | balance |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | Joe | White | Kalverstraat | 5 | Amsterdam | 1071 EX | 50.00 |
| 2 | 1 | Bob | Black | Coolsingel | 19 | Rotterdam | 3021 FF | 95.00 |
| 3 | 1 | Alice | Blue | Vredenburg | 12 | Utrecht | 3525 AK | 35.00 |
| 4 | 2 | George | Yellow | Spuistraat | 1 | Den Haag | 2511 BC | 10.00 |
| 5 | 2 | Carol | Green | Steenstraat | 23 | Arnhem | 6821 DL | 75.00 |

TABLE 4-1 EXAMPLE DATA FOR THE STUDENT RELATION

| id | first_name | last_name |
|---|---|---|
| 1 | Roger | Orange |
| 2 | Nancy | Purple |

TABLE 4-2 EXAMPLE DATA FOR THE INSTRUCTOR RELATION

Both relations have the `id` attribute as their primary key. Besides the primary key, the `Student` relation contains a functional dependency that states that the `zip_code` attribute uniquely determines the `street` and `city` attributes:

$FD_{zip\_code}$ = zip_code → street, city

There is also a foreign key constraint defined for the `Student` relation. This foreign key constraint relates every `Student` tuple to an `Instructor` tuple:

$FK_{instructor}$: Student (instructor) REFERENCES Instructor (id)

## 4.4 CONFLICTS BASED ON UNEQUAL DATABASE STATES AFTER SYNCHRONIZATION

Definition 4.1.4 of a conflict between two atomic operations consists of two parts. As stated in section 4.1, a conflict between two arbitrary atomic operations can either be based on unequal database states after synchronization or be based on a violation of integrity constraints of the database schema. This section investigates conflicts between two arbitrary atomic operations based on unequal database states after synchronization. Section 4.5 investigates conflicts between two arbitrary atomic operations based on violations of integrity constraints of the database schema.

### 4.4.1 UNEQUAL DATABASE STATES AFTER SYNCHRONIZATION

A conflict based on unequal database states after synchronization between two atomic operations $op_1$ and $op_2$, which both operate on an initial database state $S$, occurs if the database state $op_2(op_1(S))$ is

not equal to the database state $op_1(op_2(S))$. These two database states are not equal if they do not contain exactly the same tuples (definition 4.1.1 and definition 4.1.2).

If it can be proved that for two arbitrary atomic operations $op_1$ and $op_2$, which both operate on an initial database state $S$, the database states $op_2(op_1(S))$ and $op_1(op_2(S))$ are equal for all possible operations $op_1$ and $op_2$ and all possible initial database states $S$, it can be concluded that there cannot be a conflict based on unequal database states between two operations of the types of $op_1$ and $op_2$.

Section 4.4.2 investigates all combinations of types of atomic operations. For each combination an attempt is made to prove that $op_2(op_1(S))$ is equal to $op_1(op_2(S))$. This is done by trying to transform $op_2(op_1(S))$ into $op_1(op_2(S))$ using set theory.

Because the atomic delete and update operations have the requirement that the tuple they affect must still exists in the relation they operate on, each proof must have the additional property that this requirement holds in every step of the proof. If this requirement does not hold in every step of the proof, further analysis must be done to see if the operations conflict.

### 4.4.2 IDENTIFIED CONFLICTS BETWEEN ALL TYPE COMBINATIONS OF ATOMIC OPERATIONS

For all the combinations of atomic operations, this section investigates if it is possible to proof the database state $op_2(op_1(S))$ is equal to $op_1(op_2(S))$. As an example, for two insert operations that insert the tuples $s$ and $t$ into a database state $S$, an attempt is made to proof that $S \cup \{s\} \cup \{t\}$ is equal to $S \cup \{t\} \cup \{s\}$. For an insert and a delete operation that insert a tuple $s$ and delete a tuple $t$ from a database state $S$, an attempt is made to proof that $S \cup \{s\} - \{t\}$ is equal to $S - \{t\} \cup \{s\}$.

As stated before, each proof must have the additional property that for each step in the proof the requirement that the tuples affected by an update or delete operation must still exist in the relations they operate on. So for each proof it must also be examined if this requirement holds in each step of the proof. The remainder of this section refers to this requirement as the *existing tuple requirement for delete and update operations*.

If it can be proved that for a combination of atomic operations the database state $op_2(op_1(S))$ is equal to $op_1(op_2(S))$ and the existing tuple requirement for delete and update operations holds in every step of this proof, there can be no conflict between those two types of atomic operations. If a proof exists for equal states exists, but it is not possible to proof that all steps of this proof hold the existing tuple requirement for delete and update operations, the conditions in which this requirement is violated are also the conditions for which these two types of atomic operations can conflict. If no proof for equal database states exists, the atomic operations of these two types conflict.

The existing tuple requirement for delete and update operations only has to be checked in the cases where the conflicts between any combinations of delete and update operations are investigated. In the cases where the conflicts between a delete or an update operation and an insert operation are investigated, the existing tuple requirement for delete and update operations is always preserved, because the insert operation does not remove tuples.

As stated before, the existing tuple requirements for delete and update operations is not preserved if for a delete or an update operation the tuple that is deleted or updated does not exist in the database state the operation operates on. This means that for each set difference operator that occurs in a step of a proof, the database state represented by the left hand side of the set difference operator must still contain the tuple that is at the right hand side of the set difference operator.

If a step in a proof only contains a single set difference operator the existing tuple requirement for delete and update operations holds, because no other operator in this step of the proof can remove a tuple. If a step in a proof contains two set difference operators it is possible that the existing tuple requirement for delete and update operations is violated, because the first set difference operator can remove the tuple that is required for the second set difference operator. For the first set difference operator in these cases the same argument holds as for the steps in the proofs that contain only a single set difference operator.

If a step in a proof contains two set difference operators, it must be checked if the database state represented by the left hand side of the second set difference operator still contains the tuple of the right hand side of the second set difference operator. If this is the case, the existing tuple requirement for delete and update operations holds. If this is not the case, a condition for which it is possible that there is a conflict is identified. Further investigation is necessary to determine whether this condition always leads to a conflict, or if there are certain circumstances under which this condition does not lead to a conflict.

In each of the next subsections a combination of atomic operations is investigated. For each combination of atomic operations a proof is given, and each step of the proofs is checked to see if the existing tuple requirement for delete and updates operations holds. Each subsection concludes with a theorem regarding conflicts based on unequal database states after synchronization for the discussed combination of atomic operations. The discussion in the subsections is based on an initial database state that consists of a single relation $\mathrm{R}$, to keep the text succinct. This discussion translates directly into database states consisting of multiple relations in the cases where only the database state is considered, and not the integrity constraints of the database schema.

The proofs in the upcoming subsections make use of a number of named set theory manipulations (Union is Commutative), (Set Difference with Union), (Set Difference is Right Distributive over Union), that are used to transform one step of a proof into another step of a proof. These set theory manipulations hold for all possible values of the arguments in the manipulation. The set theory manipulations used are:

1. Set union is commutative
   $R \cup S \cup T = R \cup T \cup S$

2. Set difference with union
   $R - S - T = R - (S \cup T)$

3. Set difference is right distributive over union
   $R \cup S - T = (R - T) \cup (S - T)$

#### 4.4.2.1 INSERT/INSERT CONFLICTS

Two insert operations insert tuples `s` and `t` into a relation `R`:

***Given***:
$relation$: $R$,
$insert\ operation\ op_1$: $R \cup \{s\}$,
$insert\ operation\ op_2$: $R \cup \{t\}$

Insert operations add new tuples to a relation:

***With***:
$s, t \notin R$,
$S = \{s\}$,
$T = \{t\}$

For all possible values of `R`, `s` and `t` try to prove that the database state `R ∪ {s} ∪ {t}` is equal to the database state `R ∪ {t} ∪ {s}`:

***Prove***:
$\forall\ R, s, t: R \cup S \cup T = R \cup T \cup S$

***Proof***:

1. $R \cup S \cup T$
2. $R \cup T \cup S$ $\qquad\qquad$ $Set\ union\ is\ commutative$

Because this proof concerns two insert operations there are no set theory manipulations that can violate the semantics of the atomic operations. And since the set theory manipulation holds for all possible values of `R`, `S` and `T`, it can be concluded that there can be no conflicts between two insert operations based on unequal database states after synchronization:

THEOREM 4.4.1
There are no conflicts based on unequal database states after synchronization for two insert operations.

#### 4.4.2.2 DELETE/DELETE CONFLICTS

Two delete operations delete tuples `s` and `t` from a relation `R`:

***Given***:
$relation$: $R$,
$delete\ operation\ op_1$: $R - \{s\}$,
$delete\ operation\ op_2$: $R - \{t\}$

Delete operations remove existing tuples from a relation:

***With***:
$s, t \in R$,
$S = \{s\}$,
$T = \{t\}$

For all possible values of `R`, `s` and `t` try to prove that the database state `R - {s} - {t}` is equal to the database state `R - {t} - {s}`:

***Prove***:
$\forall\, R, s, t: R - S - T = R - T - S$

***Proof***:

1. $R - S - T$
2. $R - (S \cup T)$ — *Set difference with union*
3. $R - (T \cup S)$ — *Set union is commutative*
4. $R - T - S$ — *Set difference with union*

Because this proof concerns two delete operations it must be checked if the semantics of the delete operations are preserved in each step of the proof. As explained in the introduction of section 4.4.2, for each step in the proof that contains two set difference operators, it must be checked that the database state represented by the left hand side of the second set difference operator still contains the tuple of the right hand side of the second set difference operator.

In this proof the first and fourth step contain two set difference operators. In the first step the database state at the left hand side of the second set difference operator is `R - S` and the right hand side is `T`. In the fourth step the database state at the left hand side of the second set difference operator is `R - T`, and the right hand side is `S`. For both steps the semantics of the delete operation are violated when `S` is equal to `T`, which is possible for two delete operations because the tuples `s` and `t` are both from `R`.

However, in the case for two delete operations this violation of the semantics can be disregarded, because the purpose of delete operations is to remove a tuple from a database state. If both delete operations delete the same tuple, for synchronization purposes it is not relevant which operation actually deletes the tuple, as long as the synchronized database state does not contain the tuple.

Because a proof exists which preserves the semantics of the delete operations, there can be no conflicts between two delete operations based on unequal database states after synchronization:

THEOREM 4.4.2
There are no conflicts based on unequal database states after synchronization for two delete operations.

#### 4.4.2.3 UPDATE/UPDATE CONFLICTS

Two update operations update tuples `s` and `t` from a relation `R` to the tuples `s'` and `t'`:

***Given***:
$relation$: $R$,
$update\ operation\ op_1$: $R - \{s\} \cup \{s'\}$,
$update\ operation\ op_2$: $R - \{t\} \cup \{t'\}$

The relation `R` does contain the tuples to update `s` and `t`, but does not contain the updated tuples `s'` and `t'`:

***With***:
$s, t \in R$,
$s', t' \notin R$,
$S = \{s\}$,
$S' = \{s'\}$,
$T = \{t\}$,
$T' = \{t'\}$

For all possible values of `R`, `s`, `t`, `s'` and `t'` try to prove that the database state `R - {s} ∪ {s'} - {t} ∪ {t'}` is equal to the database state `R - {t} ∪ {t'} - {s} ∪ {s'}`:

***Prove***:
$\forall\ R, s, s', t, t'$: $R - S \cup S' - T \cup T' = R - T \cup T' - S \cup S'$

***Proof***:

1. $R - S \cup S' - T \cup T'$
2. $R \cup S' - S - T \cup T'$ — $s \in R, s' \notin R \Rightarrow S \cap S' = \varnothing$; $S \cap S' = \varnothing \Rightarrow R - S \cup S' = R \cup S' - S$
3. $R \cup S' - S \cup T' - T$ — $t \in R, t' \notin R \Rightarrow T \cap T' = \varnothing$; $T \cap T' = \varnothing \Rightarrow R - T \cup T' = R \cup T' - T$
4. $R \cup S' \cup T' - S - T$ — $s \in R, t' \notin R \Rightarrow S \cap T' = \varnothing$; $S \cap T' = \varnothing \Rightarrow R - S \cup T' = R \cup T' - S$
5. $R \cup S' \cup T' - (S \cup T)$ — $Set\ difference\ with\ union$
6. $R \cup S' \cup T' - (T \cup S)$ — $Set\ union\ is\ commutative$
7. $R \cup S' \cup T' - T - S$ — $Set\ difference\ with\ union$
8. $R \cup T' \cup S' - T - S$ — $Set\ union\ is\ commutative$
9. $R \cup T' - T \cup S' - S$ — $t \in R, s' \notin R \Rightarrow S' \cap T = \varnothing$; $S' \cap T = \varnothing \Rightarrow R \cup S' - T = R - T \cup S'$
10. $R \cup T' - T - S \cup S'$ — $s \in R, s' \notin R \Rightarrow S \cap S' = \varnothing$; $S \cap S' = \varnothing \Rightarrow R \cup S' - S = R - S \cup S'$
11. $R - T \cup T' - S \cup S'$ — $t \in R, t' \notin R \Rightarrow T \cap T' = \varnothing$; $T \cap T' = \varnothing \Rightarrow R \cup T' - T = R - T \cup T'$

Because this proof concerns two update operations it must be checked if the semantics of the update operations are preserved in each step of the proof. As explained in the introduction of section 4.4.2, for each step in the proof that contains two set difference operators, it must be checked that the database state represented by the left hand side of the second set difference operator still contains the tuple of the right hand side of the second set difference operator.

Table 4-3 lists all the steps of the proof that contain two set difference operators. For each of these steps the table shows the database state at the left hand side of the second set difference operator, the right hand side and the condition for which the semantics of the update operations are violated.

| Step | Left hand side of second set difference operator | Right hand side of second set difference operator | Condition for violation of semantics of operations |
|---|---|---|---|
| 1 | $R - S \cup S'$ | $T$ | $S = T$ |
| 2 | $R \cup S' - S$ | $T$ | $S = T$ |
| 3 | $R \cup S' - S \cup T'$ | $T$ | $S = T$ |
| 4 | $R \cup S' \cup T' - S$ | $T$ | $S = T$ |
| 7 | $R \cup S' \cup T' - T$ | $S$ | $S = T$ |
| 8 | $R \cup T' \cup S' - T$ | $S$ | $S = T$ |
| 9 | $R \cup T' - T \cup S'$ | $S$ | $S = T$ |
| 10 | $R \cup T' - T$ | $S$ | $S = T$ |
| 11 | $R - T \cup T'$ | $S$ | $S = T$ |

TABLE 4-3 CONDITIONS WHERE VIOLATIONS OF THE SEMANTICS OF THE UPDATE OPERATIONS OCCUR IN THE PROOF FOR CONFLICTS BASED ON UNEQUAL DATABASE STATES BETWEEN TWO UPDATE OPERATIONS

As can be seen from Table 4-3, the semantics of the update operations can be violated when `S` is equal to `T`. But because the purpose of an update operation is to update the values of some attributes of a tuple, the possibility of merging two updates to the same tuple should be investigated. If it is possible to merge two updates to the same tuple and still reach a valid synchronized database state, the number of conflicts based on unequal database states for two update operations on the same tuple can be limited.

When two update operations update the same tuple, but affect different attributes of this tuple, the updates can be merged. As an example consider the following two update operations:

```
Student ← Student −
      {<1, 1, Joe, White, Kalverstraat, 5, Amsterdam, 1071 EX, 50.00>}
      ∪
      {<1, 1, Joe, White, Kalverstraat, 5, Amsterdam, 1071 EX, 0.00>}

Student ← Student −
      {<1, 1, Joe, White, Kalverstraat, 5, Amsterdam, 1071 EX, 50.00>}
      ∪
      {<1, 1, Joe, White, Leidseplein, 9, Amsterdam, 1017 PR, 50.00>}
```

The first update operation changes the `balance` attribute from `50.00` to `0.00` and the second update operation changes the address related attributes (`street, number, zipcode`) from `Kalverstraat 5 1071 EX` to `Leidseplein 9 1017 PR`. Because these update operations affect different attributes

they can be merged so that the synchronized database state will contain the tuple <1, 1, Joe, White, Leidseplein, 9, Amsterdam, 1017 PR, 0.00>, in which both update operations are reflected.

In the case where two update operations update the same attribute(s) of a tuple and the updated values of one or more of these attributes differ for the resulting tuples, the updates cannot be merged. As an example consider the following two update operations:

```
Student ← Student –
      {<1, 1, Joe, White, Kalverstraat, 5, Amsterdam, 1071 EX, 50.00>}
      ∪
      {<1, 1, Joe, White, Kalverstraat, 5, Amsterdam, 1071 EX, 0.00>}

Student ← Student –
      {<1, 1, Joe, White, Kalverstraat, 5, Amsterdam, 1071 EX, 50.00>}
      ∪
      {<1, 1, Joe, White, Kalverstraat, 5, Amsterdam, 1071 EX, 25.00>}
```

Both update operations update the value of the `balance` attribute, however the first update operation sets an updated value of `0.00` and the second update operation sets an updated value of `25.00`. In this case there is a conflict based on unequal database states after synchronization, because the value of the `balance` attribute for the resulting tuple can either be `0.00` or `25.00`. This leads to the following theorem:

THEOREM 4.4.3

There is a conflict based on unequal database states after synchronization for two update operations upd1 and upd2, if rel(upd1) = rel(upd2) = R, tuple(upd1) = tuple(upd2), t = result(upd1) and s = result(upd2), and there exists an attribute A ∈ attrib(upd1) ⋂ attrib(upd2) for which t[A] != s[A].

Theorem 4.4.3 makes use of the functions `rel(op)`, `tuple(op)`, `result(upd)` and `attrib(upd)`. All functions used in this master thesis are listed in Appendix A. The function `rel(op)` returns the relation affected by an operation `op`. The function `tuple(op)` returns the tuple affected by an operation `op`. For an insert and a delete operation these are the inserted and deleted tuple, for an update operation this is the tuple that is updated. The function `result(upd)` returns the updated tuple for an update operation `upd`. The function `attrib(upd)` returns a set of attributes of a tuple that are updated by an update operation `upd`.

But even in the case where an attribute of a tuple is updated to the same value by two update operations, there still is the possibility of a conflict based on unequal database states after synchronization. This depends on the type of update that is performed on that attribute. Two types of updates are distinguished: relative updates and absolute updates.

A relative update increments or decrements an existing value of an attribute of a tuple. An absolute update sets a new value for an attribute of a tuple, ignoring the existing value of that attribute. An

example of a relative update of the balance attribute is `balance = balance + 10.00`. An example of an absolute update of the same attribute is `balance = 50.00`.

It is easy to see that absolute updates to the same attribute of the same tuple do not conflict when both update operations update the attribute to the same value, because the end result after synchronization will be that the updated tuple has that absolute value. Absolute updates to the same attribute of the same tuple to different values do conflict, this follows from Theorem 4.4.3. The case of relative updates to an attribute of the same tuple is more complicated. When the relative updates to an attribute of the same tuple are relative updates with different values, there is a conflict; this follows from Theorem 4.4.3, because the resulting tuples will have different values for the same attribute. The complication occurs for relative updates that update the attribute with the same value.

The problem with two relative updates of an attribute with the same value is that the synchronization process cannot determine the correct value for the attribute, because there are two possibilities. The first possibility is that the relative update is performed only once, and the second possibility is that the relative update is performed twice. When for example two relative updates both increment the `balance` attribute with `10.00` after synchronization the result should reflect that the balance attribute is incremented with either `10.00` or `20.00`. Because there are two possibilities, the synchronization process cannot determine which result is valid, and thus relative updates of an attribute of the same tuple with the same values must be identified as a conflict.

From the discussion in the previous paragraphs can be concluded that there is a conflict between two update operations that update the same attribute to the same value, when at least one of these updates to this attribute is a relative update. This is formalized in Theorem 4.4.4.

THEOREM 4.4.4

There is a conflict based on unequal database states after synchronization for two update operations upd1 and upd2, if rel(upd1) = rel(upd2),
tuple(upd1) = tuple(upd2), t = result(upd1) and s = result(upd2), and there exists an attribute A ∈ attrib(upd1) ⋂ attrib(upd2) for which t[A] = s[A] and where updType(upd1, A) = RELATIVE or updType(upd2, A) = RELATIVE.

Theorem 4.4.4 introduces the function `updType(upd, A)`. This function returns the type of update that the operation `upd` performs on attribute `A`. The result can be `ABSOLUTE` or `RELATIVE`.

#### 4.4.2.4 INSERT/DELETE CONFLICTS

An insert operation adds a tuple `s` to a relation `R` and a delete operation removes a tuple `t` from a relation `R`:

$\boldsymbol{Given}$:
$relation$: $R$,
$insert\ operation\ op_1$: $R \cup \{s\}$,
$delete\ operation\ op_2$: $R - \{t\}$

The relation R does not contain the inserted tuple s, but does contain the deleted tuple t:

***With***:
$s \notin R$,
$t \in R$,
$S = \{s\}$,
$T = \{t\}$

For all possible values of R, s and t try to prove that the database state R ∪ {s} − {t} is equal to the database state R − {t} ∪ {s}:

***Prove***:
$\forall R, s, t: R \cup S - T = R - T \cup S$

***Proof***:

1. $R \cup S - T$
2. $(R - T) \cup (S - T)$ — *Set difference is right distributive over union*
3. $(R - T) \cup S$ — $s \notin R, t \in R \Rightarrow S \cap T = \varnothing$; $S \cap T = \varnothing \Rightarrow S - T = S$
4. $R - T \cup S$

Because this proof concerns an insert and a delete operation there are no set theory manipulations that can violate the semantics of the atomic operations. And since the set theory manipulations hold for all possible values of R, S and T, it can be concluded that there can be no conflicts between an insert and a delete operation based on unequal database states after synchronization:

THEOREM 4.4.5
There are no conflicts based on unequal database states after synchronization for an insert and a delete operation.

#### 4.4.2.5 INSERT/UPDATE CONFLICTS

An insert operation inserts a tuple s into a relation R and an update operation updates a tuple t in a relation R to the tuple t':

***Given***:
$relation$: $R$,
$insert\ operation\ op_1$: $R \cup \{s\}$,
$update\ operation\ op_2$: $R - \{t\} \cup \{t'\}$

The relation R does not contain the inserted tuple s, it does contain the tuple to update t, but it does not contain the updated tuple t':

$\boldsymbol{With}$:
$s \notin R$,
$t \in R$,
$t' \notin R$,
$S = \{s\}$,
$T = \{t\}$,
$T' = \{t'\}$

For all possible values of `R`, `s`, `t`, and `t'` try to prove that the database state `R ∪ {s} − {t} ∪ {t'}` is equal to the database state `R − {t} ∪ {t'} ∪ {s}`:

$\boldsymbol{Prove}$:
$\forall R, s, t, t' \colon R \cup S - T \cup T' = R - T \cup T' \cup S$

$\boldsymbol{Proof}$:

| | | |
|---|---|---|
| 1. | $R \cup S - T \cup T'$ | |
| 2. | $(R - T) \cup (S - T) \cup T'$ | $Set\ difference\ is\ right\ distributive\ over\ union$ |
| 3. | $(R - T) \cup T' \cup (S - T)$ | $Set\ union\ is\ commutative$ |
| 4. | $(R - T) \cup T' \cup S$ | $s \notin R, t \in R \Rightarrow S \cap T = \varnothing$<br>$S \cap T = \varnothing \Rightarrow S - T = S$ |
| 5. | $R - T \cup T' \cup S$ | |

Because this proof concerns an insert and an update operation there are no set theory manipulations that can violate the semantics of the atomic operations. And since the set theory manipulations hold for all possible values of `R`, `s`, `t` and `t'`, it can be concluded that there can be no conflicts between an insert and an update operation based on unequal database states after synchronization:

THEOREM 4.4.6
There are no conflicts based on unequal database states after synchronization for an insert and an update operation.

#### 4.4.2.6 DELETE / UPDATE CONFLICTS

A delete operation deletes a tuple `s` from a relation `R` and an update operation updates a tuple `t` from relation `R` to a tuple `t'`:

$\boldsymbol{Given}$:
$relation \colon R$,
$delete\ operation\ op_1 \colon R - \{s\}$,
$update\ operation\ op_2 \colon R - \{t\} \cup \{t'\}$

The relation `R` does contain the tuple to delete `s` and the tuple to update `t`, but it does not contain the updated tuple `t'`:

***With***:
$s, t \in R$,
$t' \notin R$,
$S = \{s\}$,
$T = \{t\}$,
$T' = \{t'\}$

For all possible values of `R`, `s`, `t`, and `t'` try to prove that the database state `R - {s} - {t} ∪ {t'}` is equal to the database state `R - {t} ∪ {t'} - {s}`:

***Prove***:
$\forall R, s, t, t' \colon R - S - T \cup T' = R - T \cup T' - S$

***Proof***:

| | | |
|---|---|---|
| 1. | $R - S - T \cup T'$ | |
| 2. | $R - (S \cup T) \cup T'$ | *Set difference with union* |
| 3. | $R - (T \cup S) \cup T'$ | *Set union is commutative* |
| 4. | $R - T - S \cup T'$ | *Set difference with union* |
| 5. | $R - T \cup T' - S$ | $s \in R, t' \notin R \Rightarrow S \cap T' = \varnothing$<br>$S \cap T' = \varnothing \Rightarrow R - S \cup T' = R \cup T' - S$ |

Because this proof concerns a delete and an update operation it must be checked if the semantics of the delete and update operation are preserved in each step of the proof. As explained in the introduction of section 4.4.2, for each step in the proof that contains two set difference operators, it must be checked that the database state represented by the left hand side of the second set difference operator still contains the tuple of the right hand side of the second set difference operator.

In this proof the first, fourth and fifth step contain two set difference operators. In the first step of the proof the left hand side of the second set difference operator is the database state `R - S` and the right hand side is `T`. In the fourth step of the proof the left hand side of the second set difference operator is the database state `R - T` and the right hand side is `S`. In the fifth step of the proof the left hand side of the second set difference operator is the database state `R - T ∪ T'` and the right hand side is `S`. For all these three steps the semantics of the delete and/or update operation are violated when `S` is equal to `T`, which is possible for these operations because the tuples `s` and `t` are both from `R`. Because the purpose of a delete operation is to remove a tuple from a database state and the purpose of an update operation is to update an existing tuple in a database state, it can be concluded that there is a conflict based on unequal database states after synchronization between a delete and an update operation when they affect the same tuple, and there are no conflicts based on unequal database states after synchronization between a delete and an update operation if they affect different tuples:

THEOREM 4.4.7
There is a conflict based on unequal database states after synchronization for a delete operation del and an update operation upd, if rel(del) = rel(upd) and tuple(del) = tuple(upd).

## 4.5 CONFLICTS BASED ON VIOLATIONS OF INTEGRITY CONSTRAINTS

Definition 4.1.4 of a conflict between two atomic operations consists of two parts. A conflict between two arbitrary atomic operations can either be based on unequal database states after synchronization or be based on a violation of integrity constraints of the database schema. Conflicts between two arbitrary atomic operations based on unequal database states after synchronization were investigated in section 4.4. This section investigates conflicts between two arbitrary atomic operations based on violations of integrity constraints of the database schema.

This section distinguishes between two types of integrity constraints, intra-relational integrity constraints and inter-relational integrity constraints. As stated in the introduction of section 3.1.1, an intra-relational integrity constraint is an integrity constraint defined for a single relation, while an inter-relational integrity constraint is defined over multiple relations. Section 4.5.1 investigates conflicts between two arbitrary atomic operations based on violations of intra-relational integrity constraints, while section 4.5.2investigates conflicts between two arbitrary atomic operations based on violations of inter-relational integrity constraints.

### 4.5.1 VIOLATIONS OF INTRA-RELATIONAL INTEGRITY CONSTRAINTS

This section investigates conflicts between two arbitrary atomic operations that can occur due to violations of intra-relational integrity constraints. The intra-relational integrity constraints that are investigated in this section are key constraints and functional dependencies. These are the intra-relational integrity constraints that are identified in chapter 3 and which will be guaranteed by the synchronization solution proposed in this master thesis, as stated in section 3.3.

Because key constraints are a special kind of functional dependency the detection of conflicts based on violations of intra-relational integrity constraints is based on determining violations of functional dependencies. This section starts with the definition of a functional dependency violation, followed by a discussion of functional dependency violations specific to all possible combinations of atomic operations.

#### 4.5.1.1 FUNCTIONAL DEPENDENCY VIOLATION

Functional dependencies are discussed in section 3.1.5. Multiple functional dependencies can exists for a single relation, but each relation has at least one, since each relation has at least one key and a key is a functional dependency. A functional dependency is described in the form `X` → `Y`, with `X` the determinant attribute set and `Y` the dependent attribute set of the functional dependency.

In this master thesis the determinant attribute set of a functional dependency `FD` is identified with the function `det(FD)`, and the dependent attribute set is identified with the function `dep(FD)`. To be able to detect functional dependency violations, a method to compare the determinant and dependent attribute sets of tuples is needed. The projection of an attribute set on a tuple is used for this comparison. This master thesis uses the notation `t[X]` for projection on a tuple, with `t` as the tuple and `X` as the projected attribute(s).

To compare the determinant attribute sets for two tuples `s` and `t`, let `X = det(FD)`. The projections of the determinant attribute set `X` on the tuples `s` and `t` are `s[X]` and `t[X]`. When `s[X] = t[X]` the determinant attribute sets of the two tuples are equal. Comparing depending attribute sets is analogous: let `Y = dep(FD)`, the depending attribute sets of the tuples `s` and `t` are equal when `s[Y] = t[Y]`.

A requirement for these comparisons is that the tuples belong to the same relation, and that the functional dependency `FD` is defined for this relation. For this the `rel()` function is used to determine the relation of an operation, a tuple or functional dependency. The requirement that the compared tuples must belong to the same relation and that the functional dependency must be defined for that relation can now be expressed as `rel(s) = rel(t) = rel(FD)`.

A functional dependency in a relation is violated if there exist two tuples in a relation that have the same values for the attributes in the determinant attribute set of the functional dependency, but have one or more different values for the attributes in the dependent attribute set. A functional dependency violation is defined as:

DEFINITION 4.5.1 FUNCTIONAL DEPENDENCY VIOLATION

For some database state S there is a functional dependency violation if there exist two tuples s, t and a functional dependency FD with X = det(FD), Y = dep(FD), for which rel(s) = rel(t) = rel(FD) and s[X] = t[X] and s[Y] != t[Y].

#### 4.5.1.2 DETECTING FUNCTIONAL DEPENDENCY VIOLATIONS FOR TWO ATOMIC OPERATIONS

When identifying conflicts between two arbitrary atomic operations, it is assumed that these atomic operations do not violate any of the integrity constraints of the database schema on their own, as stated in Definition 4.1.4. This means that there can only be conflicts based on functional dependency violations when the two investigated operations both add tuples to or update tuples of the same relation in such a way that the resulting tuples conflict with each other for a functional dependency defined for that relation.

Because delete operations do not add tuples to a relation or update tuples of a relation, any combination of operations where a delete operation is involved cannot lead to a functional dependency violation. For all combinations of insert and update operations there are conflicts based on functional dependency violations if they both operate on the same relation, and there exists a functional dependency for this relation for which the inserted or updated tuples have the same values for all the attributes in the determinant attribute set of this functional dependency, but have different values for at least one of the attributes in the dependent attribute set of this functional dependency.

#### 4.5.1.3 INSERT/INSERT CONFLICTS

Two insert operations violate a functional dependency if both operations insert a tuple into the same relation for which the functional dependency is defined, and these tuples have the same values for all the attributes of the determinant attribute set of this functional dependency, but have different values for at least one of the attributes of the dependent attribute set of this functional dependency. This follows directly from Definition 4.5.1.

This means that two insert operations conflict which each other based on a functional dependency violation when they both insert a tuple into the same relation, and there exists a functional dependency for this relation for which these tuples have the same values for all the attributes of the dependent attribute set of this functional dependency, but have different values for at least one of the attributes of the dependent attribute set of this functional dependency:

THEOREM 4.5.1
There is a conflict based on a functional dependency violation for two insert operations ins1 and ins2, if rel(ins1) = rel(ins2) = R, t = tuple(ins1), s = tuple(ins2), and there exists a functional dependency FD with rel(FD) = R, X = det(FD), Y = dep(FD) for which t[X] = s[X] and t[Y] != s[Y].

The function `tuple(op)` used in Theorem 4.5.1 returns the inserted tuple if applied to an insert operation. For a delete operation it returns the deleted tuple and for an update operation it returns the tuple that was updated by the update operation.

As an example of the application of Theorem 4.5.1 consider the following two insert operations into the example `Student` relation described in section 4.3:

```
Student (
      id: Integer, instructor: Integer, first_name: String,
      last_name: String, street: String, number: String,
      city: String, zip_code: String, balance: Decimal
)

s = <6, 2, Frank, Red, Grote Markt, 33, Groningen, 9731 CG, 25.00>
t = <6, 2, Hank, Gray, Graafseweg, 8, Nijmegen, 6541 LB, 10.00>
Student ← Student ∪ {s}
Student ← Student ∪ {t}
```

The functional dependencies for the Student relation are:

```
FD_PK:  id       → instructor, first_name, last_name, street,
                   number, city, zip_code, balance

FD_ZIP: zip_code → street, city
```

The projections of the determinant and dependent attribute sets for the $FD_{PK}$ functional dependency on the tuples `s` and `t` are:

```
X = det(FD_PK) = <id>
Y = dep(FD_PK) = <instructor, first_name, last_name, street, number,
                  city, zip_code, balance>
s[X] = <6>
t[X] = <6>
s[Y] = <2, Frank, Red, Grote Markt, 33, Groningen, 9731 CG, 25.00>
t[Y] = <2, Hank, Gray, Graafseweg, 8, Nijmegen, 6541 LB, 10.00>
```

The projections of the determinant and dependent attribute sets for the $FD_{ZIP}$ functional dependency on the tuples s and t are:

```
X = det(FD_ZIP) = <zip_code>
Y = dep(FD_ZIP) = <street, city>
s[X] = <9731 CG>
t[X] = <6541 LB>
s[Y] = <Grote Markt, Groningen>
t[Y] = <Graafseweg, Nijmegen>
```

Because the values for the attributes of the determinant attribute set of $FD_{PK}$ are the same for the tuples s and t, but the values for the attributes of the dependent attribute set of $FD_{PK}$ are different for the tuples s and t, Theorem 4.5.1 states that there is a conflict based on a functional dependency violation between these two insert operations, based on the violation of $FD_{PK}$. Functional dependency $FD_{ZIP}$ is not violated by these two insert operations, since the values of the attributes of the determinant attribute set differ for the tuples s and t.

#### 4.5.1.4 DELETE/DELETE CONFLICTS

A functional dependency violation can only occur if both operations affect the same relation and both operations insert or update tuples, as stated in section 4.5.1.2. Because delete operations remove tuples from a relation, there cannot be conflicts based on functional dependency violations for two delete operations:

THEOREM 4.5.2

There are no conflicts based on a functional dependency violation for two delete operations.

#### 4.5.1.5 UPDATE/UPDATE CONFLICTS

A functional dependency violation can occur for two update operations when they both affect the same relation, this follows from the discussion in section 4.5.1.2. However, two different cases can be distinguished for conflicts based on functional dependency violations for two update operations. The first case occurs when the two update operations update different tuples from the same relation. This case is similar to the discussion of conflicts based on functional dependency violations for two insert operations. The second case occurs when the two update operations update the same tuple.

Two update operations that update different tuples violate a functional dependency if both operations update a tuple of the same relation for which a functional dependency is defined, and the updated tuples have the same values for all the attributes of the determinant attribute set of this functional dependency, but have different values for at least one of the attributes of the dependent attribute set of this functional dependency. This follows directly from Definition 4.5.1.

This means that two update operations that update different tuples of the same relation conflict with each other based on a functional dependency violation if there exists a functional dependency for this relation for which the updated tuples have the same values for all the attributes of the dependent attribute set of this functional dependency, but have different values for at least one of the attributes of the dependent attribute set of this functional dependency:

THEOREM 4.5.3

There is a conflict based on a functional dependency violation for two update operations upd1 and upd2, if rel(upd1) = rel(upd2) = R, tuple(upd1) != tuple(upd2), t = result(upd1), s = result(upd2) and there exists a functional dependency FD with rel(FD) = R, X = det(FD), Y = dep(FD) for which t[X] = s[X] and t[Y] != s[Y].

Theorem 4.5.3 makes use of the function `result(upd)`, which returns the updated tuple for an update operation and is only valid for update operations.

For the case where two update operations update the same tuple of the same relation it must be investigated whether these two different updates can be merged during synchronization. There is the possibility of conflict based on unequal database states after synchronization, which is discussed in section 4.4.2.3. However, a merge of the two update operations can also lead to the violation of a functional dependency if attributes of a determining or dependent attribute set of some functional dependency are updated.

When identifying conflicts for updates to two different tuples it is assumed that both update operations do not violate any of the integrity constraints of the database on their own. But when merging two updates to the same tuple, this assumption does not hold any longer. However, to determine if there actually is a functional dependency violation the resulting merged tuple must be compared to all other tuples in the relation the tuple belongs to. But the other tuples of the relation are not known during synchronization, so it will not be possible to determine with certainty if a functional dependency violation occurs when merging two update operations that update the same tuple of the same relation.

The solution chosen in this master thesis is to identify a possible conflict based on the violation of a functional dependency for two update operations that update the same tuple of a relation, if one of the update operations updates an attribute of the determinant or dependent attribute set of a functional dependency that is defined for the relation, because it is possible that the tuple that is the result of merging these two update operations violates a functional dependency defined for this relation:

THEOREM 4.5.4

There is a possible conflict based on a functional dependency violation between two update operations upd1 and upd2, if rel(upd1) = rel(upd2) = R, tuple(upd1) = tuple(upd2), and there exists a functional dependency FD with rel(FD) = R for which there exists an attribute A ∈ R where A ∈ attrib(upd1) ∪ attrib(upd2) and also A ∈ det(FD) ∪ dep(FD).

Theorem 4.5.4 uses the function `attrib(upd)`, which returns the set of attributes of the tuple that were updated by an update operation `upd`. This function is only valid for update operations. Two update operations `upd1` and `upd2` do not update the same attributes if for these two update operations the statement `attrib(upd1) ∩ attrib(upd2) = ∅` is true.

It must be noted that the approach used in Theorem 4.5.4 is a simplified approach which can be optimized. If for example both update operations update a single attribute of a tuple that is also in the determinant or dependent attribute set of a functional dependency defined for the relation to which the updated tuple belongs, and both update operations perform an absolute update to the same value for this attribute, there is no conflict based on a functional dependency violation after merging these tuples.

As an example of the application of Theorem 4.5.4 consider the following two update operations to the same tuple of the example `Student` relation of section 4.3:

```
Student (
      id: Integer, instructor: Integer, first_name: String,
      last_name: String, street: String, number: String,
      city: String, zip_code: String, balance: Decimal
)

FD_PK:  id       → instructor, first_name, last_name, street,
                   number, city, zip_code, balance

FD_ZIP: zip_code → street, city

upd1: Student ← Student −
      {<1, 1, Joe, White, Kalverstraat, 5, Amsterdam, 1071 EX, 50.00>}
      ∪
      {<1, 1, Joe, White, Kalverstraat, 5, Amsterdam, 1071 EZ, 0.00>}

upd2: Student ← Student −
      {<1, 1, Joe, White, Kalverstraat, 5, Amsterdam, 1071 EX, 50.00>}
      ∪
      {<1, 1, Joe, White, Leidseplein, 10, Amsterdam, 1071 EX, 0.00>}
```

To determine if these two update operations have a possible conflict due to a violation of the functional dependency $FD_{ZIP}$ it must be checked if there is an attribute that exists in both `attrib(upd1) ∪ attrib(upd2)` and `det(FD) ∪ dep(FD)`:

```
attrib(upd1) = {zip_code}
attrib(upd2) = {street, number}
attrib(upd1) ∪ attrib(upd2) = {zip_code, street, number}

det(FDZIP) = {zip_code}
dep(FDZIP) = {street, city}
det(FD) ∪ dep(FD) = {zip_code, street, city}
```

Because the attributes `zip_code` and `street` are in both `attrib(upd1) ∪ attrib(upd2)` and `det(FD) ∪ dep(FD)`, Theorem 4.5.4 states that the update operations `upd1` and `upd2` have a possible conflict with each other based on the possible violation of the functional dependency $FD_{ZIP}$.

#### 4.5.1.6 INSERT/DELETE CONFLICTS

A functional dependency violation can only occur if both operations affect the same relation and both operations insert or update tuples, as stated in section 4.5.1.2. Because for this combination only the insert operation inserts a tuple into a relation, there cannot be conflicts based on functional dependency violations between an insert and a delete operation:

THEOREM 4.5.5

There are no conflicts based on a functional dependency violation for an insert and a delete operation.

#### 4.5.1.7 INSERT/UPDATE CONFLICTS

The discussion for conflicts based on a functional dependency violation between an insert and an update operation is similar to the discussion for conflicts based on a functional dependency violation for two insert operations, as described in section 4.5.1.3. The difference is that for an insert operation and an update operation the values of the attributes for the determinant and dependent attribute sets of functional dependencies must be compared between the inserted and the tuple that is the result of the update operation.

An insert and an update operation violate a functional dependency if both operations affect the same relation for which a functional dependency is defined, and the inserted and updated tuples have the same values for all the attributes of the determinant attribute set of this functional dependency, but have different values for at least one of the attributes of the dependent attribute set of this functional dependency. This follows directly from Definition 4.5.1.

This means that an insert and an update operation conflict which each other based on a functional dependency violation if the update operation updates a tuple in the relation in which the insert operation inserts a tuple, and there exists a functional dependency for this relation for which the inserted and updated tuples have the same values for all the attributes of the dependent attribute set of this functional dependency, but have different values for at least one of the attributes of the dependent attribute set of this functional dependency:

THEOREM 4.5.6

There is a conflict based on a functional dependency violation for an insert operation ins and an update operation upd, if rel(ins) = rel(upd) = R, t = tuple(ins), s = result(upd), and there exists a functional dependency FD with rel(FD) = R,  X = det(FD), Y = dep(FD) for which t[X] = s[X] and t[Y] != s[Y].

#### 4.5.1.8 DELETE/UPDATE CONFLICTS

A functional dependency violation can only occur when both operations affect the same relation and both operations insert or update tuples, as stated in section 4.5.1.2. Because only the update operation updates a tuple of a relation, there cannot be conflicts based on functional dependency violations between a delete and an update operation:

THEOREM 4.5.7

There are no conflicts based on a functional dependency violation for a delete and an update operation.

### 4.5.2 VIOLATIONS OF INTER-RELATIONAL INTEGRITY CONSTRAINTS

This section investigates conflicts between two arbitrary atomic operations that can occur due to violations of inter-relational integrity constraints. The inter-relational integrity constraints that are investigated in this section are foreign key constraints and inclusion dependencies. These are the inter-relational integrity constraints that are identified in chapter 3 and which will be guaranteed by the synchronization solution proposed in this master thesis, as stated in section 3.3.

This section starts with the definition of a referential integrity constraint violation, which is followed by two sections that investigate which conflicts based on violations of foreign key constraints and inclusion dependencies can occur for two arbitrary atomic operations.

#### 4.5.2.1 DEFINITION OF REFERENTIAL INTEGRITY CONSTRAINT VIOLATION

As stated in section 3.1.1, for every tuple in a referencing relation of a referential integrity constraint there must exist a referenced tuple in the referenced relation of that referential integrity constraint. So a violation of a referential integrity constraint occurs if there exists a tuple in the referencing relation for which there does not exist a referenced tuple in the referenced relation.

In this master thesis the function `referencingRel(RIC)` returns the referencing relation of a referential integrity constraint `RIC` and the function `referencedRel(RIC)` returns the referenced relation of `RIC`. The function `referencingAttrib(RIC)` returns the referencing attribute set of `RIC` and the function `referencedAttrib(RIC)` returns the referenced attribute set of `RIC`. A referential integrity constraint violation can now be defined as:

DEFINITION 4.5.2 REFERENTIAL INTEGRITY CONSTRAINT VIOLATION

For some database state S there is a referential integrity constraint violation if there exists a referential integrity constraint RIC with R = referencingRel(RIC), S = referencedRel(RIC), X = referencingAttrib(RIC), Y = referencedAttrib(RIC) and a tuple $t \in R$ for which there does not exists a tuple $s \in S$ where t[X] = s[Y].

#### 4.5.2.2 FOREIGN KEY CONSTRAINT CONFLICTS

When investigating conflicts based on foreign key constraint violations, atomic operations can affect the referenced or referencing relation of a foreign key constraint. Because Definition 4.1.4 states that atomic operations on their own do not violate any integrity constraints of the database schema, conclusions can be drawn based on the type of relation affected by an atomic operation, whether it is the referenced or referencing relation of a foreign key constraint.

For a delete operation that deletes a tuple from a referenced relation of a foreign key constraint, it can be concluded that there are no tuples in the referencing relation of that foreign key constraint that referenced the deleted tuple. This statement is true because for a foreign key constraint the referenced attribute set is a key of the referenced relation, which means there can be only one tuple with a specific set of values for the referenced attribute set in the referenced relation.

The same reasoning holds for an update operation that updates attributes of the referenced attribute set of a tuple in a referenced relation of a foreign key constraint, there are no tuples in the referencing relation of that foreign key constraint that referenced the tuple that was updated.

An insert operation that inserts a tuple into a referenced relation of a foreign key constraint cannot lead to a violation of this foreign key constraint, because foreign key constraints do not have the requirement that all tuples in the referenced relation of a foreign key constraint must be referenced by a tuple in the referencing relation of that foreign key constraint.

For an insert operation that inserts a tuple into a referencing relation of a foreign key constraint, it can be concluded that there exists a tuple in the referenced relation of that foreign key constraint that is referenced by the inserted tuple.

The same reasoning holds for an update operation that updates attributes of the referencing attribute set of a tuple in a referencing relation of a foreign key constraint, there exists a tuple in the referenced relation of that foreign key constraint that is referenced by the updated tuple.

A delete operation that deletes a tuple from a referencing relation of a foreign key constraint cannot lead to a violation of this foreign key constraint, because foreign key constraints do not have the requirement that all tuples in the referenced relation of a foreign key constraint must be referenced by a tuple from the referencing relation of this foreign key constraint.

For a tuple inserted into a referencing relation of a foreign key constraint or a tuple in a referencing relation of a foreign key constraint for which attributes of the referencing attribute set it is known that a referenced tuple exists in the referenced relation of the foreign key constraint. If this referenced tuple is deleted by another operation or the attributes in the referenced attribute set of this tuple are updated by another operation, the foreign key constraint is violated according to Definition 4.5.2, because there does not exist a referenced tuple in the referenced relation of the foreign key constraint anymore. This reasoning holds because the referenced attribute set of a foreign key constraint is a key of the referenced relation, which means that tuples in a referencing relation of a foreign key constraint can reference exactly one tuple in the referenced relation of this foreign key constraint.

From the discussion in the previous paragraphs it can be concluded that there are four cases in which a foreign key constraint can be violated by two atomic operations:

1. If an insert operation inserts a tuple into the referencing relation of a foreign key constraint, and a delete operation deletes the referenced tuple of this inserted tuple from the referenced relation of this foreign key constraint:

   THEOREM 4.5.8
   There is a conflict based on a foreign key constraint violation for an insert operation ins and a delete operation del if there exists a foreign key constraint FK with referencingRel(FK) = rel(ins), referencedRel(FK) = rel(del), X = referencingAttrib(FK), Y = referencedAttrib(FK), t = tuple(ins), s = tuple(del), for which t[X] = s[Y].

2. If an update operation updates attributes of the referencing attribute set of a tuple in the referencing relation of a foreign key constraint, and a delete operation deletes the referenced tuple of this updated tuple from the referenced relation of this foreign key constraint:

   THEOREM 4.5.9
   There is a conflict based on a foreign key constraint violation for an update operation upd and a delete operation del if there exists a foreign key constraint FK with referencingRel(FK) = rel(upd), referencedRel(FK) = rel(del), X = referencingAttrib(FK), Y = referencedAttrib(FK), t = result(upd), s = tuple(del), for which t[X] = s[Y].

3. If an insert operation inserts a tuple into the referencing relation of a foreign key constraint, and an update operation updates attributes of the referenced attribute set of the referenced tuple of this inserted tuple in the referenced relation of this foreign key constraint:

   THEOREM 4.5.10
   There is a conflict based on a foreign key constraint violation for an insert operation ins and an update operation upd if there exists a foreign key constraint FK with referencingRel(FK) = rel(ins), referencedRel(FK) = rel(upd), X = referencingAttrib(FK), Y = referencedAttrib(FK), t = tuple(ins), s = tuple(upd), s' = result(upd), for which t[X] = s[Y] and s[Y] != s'[Y].

4. If an update operation updates attributes of the referencing attribute set of a tuple in the referencing relation of a foreign key constraint, and another update operation updates attributes of the referenced attribute set of the referenced tuple of this updated tuple in the referenced relation of this foreign key constraint:

   THEOREM 4.5.11
   There is a conflict based on a foreign key constraint violation for two update operations upd1 and upd2, if there exists a foreign key constraint FK with X = referencingAttrib(FK), Y = referencedAttrib(FK), for which

   a)
   referencingRel(FK) = rel(upd1), referencedRel(FK) = rel(upd2), t = result(upd1), s = tuple(upd2), s' = result(upd2) for which t[X] = s[Y] and s[Y] != s'[Y].

   or

   b)
   referencingRel(FK) = rel(upd2), referencedRel(FK) = rel(upd1), t = result(upd2), s = tuple(upd1), s' = result(upd1) for which t[X] = s[Y] and s[Y] != s'[Y].

This section is concluded with an example of the application of Theorem 4.5.8 for an insert and a delete operation. This example uses the example data from section 4.3, the `Student` and `Instructor` relations and the $FK_{instructor}$ foreign key constraint:

```
Student (
      id: Integer, instructor: Integer, first_name: String,
      last_name: String, street: String, number: String,
      city: String, zip_code: String, balance: Decimal
)

Instructor (
      id: Integer, first_name: String, last_name: String
)

FK_instructor: Student (instructor) REFERENCES Instructor (id)
```

The insert and the delete operations for this example are:

```
ins:  Student ← Student ∪
        {<6, 2, Hank, Gray, Graafseweg, 8, Nijmegen, 6541 LB, 10.00>}

del:  Instructor ← Instructor −
        {<2, Nancy, Purple>}
```

The insert operation inserts a tuple into the `Student` relation, which is the referencing relation of the $FK_{instructor}$ foreign key constraint, and the delete operation deletes a tuple from the `Instructor` relation, which is the referenced relation of $FK_{instructor}$. According to Theorem 4.5.8 these operations conflict when the values of the referencing attribute set of the inserted tuple and the values for the referenced attribute set of the deleted tuple are equal.

The referencing attribute set of the `Student` relation consists of the `instructor` attribute, and the value of the `instructor` attribute of the inserted tuple is `2`. The referenced attribute set of the `Instructor` relation is the `id` attribute, and the value for the `id` attribute of the deleted tuple is also `2`. So according to Theorem 4.5.8 these insert and delete operation conflict with each other based on a violation of the $FK_{instructor}$ foreign key constraint.

#### 4.5.2.3 INCLUSION DEPENDENCY CONFLICTS

The difference between an inclusion dependency and a foreign key constraint is that the referenced attribute set of an inclusion dependency does not have to be a key of the referenced relation. The consequence of this is that there can be multiple tuples in the referenced relation with the same values for the referenced attribute set.

The conflicts for foreign key constraints, identified in section 4.5.2.2, occur when a delete operation deletes a tuple from a referenced relation of a foreign key constraint or an update operation updates attributes of the referenced attribute set of a tuple of a referenced relation of a foreign key constraint, and inserted or updated tuples of the referencing relation of this foreign key constraint rely on the existence of these deleted or updated tuples.

Because the referenced attribute set of a foreign key constraint is a key of the referenced relation, there is exactly one referenced tuple for each tuple in the referencing relation of a foreign key constraint. If this referenced tuple in the referenced relation of a foreign key constraint on which an inserted or updated tuple relies is deleted, or the attributes of the referenced attribute set of this tuple are updated, there is no other tuple in the referenced relation that also is a referenced tuple for the inserted or deleted tuple, and the foreign key constraint is violated. However, for inclusion dependencies there can be multiple tuples in the referenced relation with the same values for the attributes of the referenced attribute set of an inclusion dependency, so deleting or updating the referenced attribute set of a referenced tuple of an inclusion dependency does not necessarily mean there are no other tuples left in the referenced relation of this inclusion dependency with the same values for the referenced attribute set as the deleted or updated tuple. But because the database state

is not known during synchronization, it cannot be determine with certainty whether this is the case or not.

From the discussion in the previous paragraph can be concluded that for inclusion dependencies the same conflicts can occur as for foreign key constraints, but because the referenced attribute set of an inclusion dependency is not a key of the referenced relation of this inclusion dependency and the database state is unknown during synchronization, it cannot be determined with certainty if a conflict actually occurred. This means that the theorems that identify conflicts based on foreign key constraint violations, theorems Theorem 4.5.8 to Theorem 4.5.11, must be adapted for inclusion dependency violations to identify possible conflicts. This leads to the following theorems for possible conflicts based on inclusion dependency violations:

THEOREM 4.5.12
There is a possible conflict based on an inclusion dependency violation for an insert operation ins and a delete operation del, if there exists an inclusion dependency IC with referencingRel(IC) = rel(ins), referencedRel(IC) = rel(del), X = referencingAttrib(IC), Y = referencedAttrib(IC), t = tuple(ins), s = tuple(del), for which t[X] = s[Y].

THEOREM 4.5.13
There is a possible conflict based on an inclusion dependency violation for an update operation upd and a delete operation del, if there exists an inclusion dependency IC with referencingRel(IC) = rel(upd), referencedRel(IC) = rel(del), X = referencingAttrib(IC), Y = referencedAttrib(IC), t = result(upd), s = tuple(del), for which t[X] = s[Y].

THEOREM 4.5.14
There is a possible conflict based on an inclusion dependency violation for an insert operation ins and an update operation upd, if there exists an inclusion dependency IC with referencingRel(IC) = rel(ins), referencedRel(IC) = rel(upd), X = referencingAttrib(IC), Y = referencedAttrib(IC), t = tuple(ins), s = tuple(upd), s' = result(upd), for which t[X] = s[Y] and s[Y] != s'[Y].

THEOREM 4.5.15
There is a possible conflict based on an inclusion dependency violation for two update operations upd1 and upd2, if there exists an inclusion dependency IC with X = referencingAttrib(IC), Y = referencedAttrib(IC), for which

a)
referencingRel(IC) = rel(upd1), referencedRel(IC) = rel(upd2), t = result(upd1), s = tuple(upd2), s' = result(upd2) for which t[X] = s[Y] and s[Y] != s'[Y].

or

b)
referencingRel(IC) = rel(upd2), referencedRel(IC) = rel(upd1), t = result(upd2), s = tuple(upd1), s' = result(upd1) for which t[X] = s[Y] and s[Y] != s'[Y].

As explained in section 4.5.2.2, for a delete operation that deletes a tuple from a referenced relation of a foreign key constraint or an update operation that updates attributes of the referenced attribute set of a tuple in the referenced relation of a foreign key constraint, it is known that there do not exist tuples in the referencing relation of this foreign key constraint that reference the deleted or updated tuple, because the delete or update operation was allowed on the local database and because the referenced attribute set is a key of the referenced relation.

However, for an inclusion dependency the referenced attribute set does not have to be a key of the referenced relation, which means that the referenced relation can contain more than one tuple with the same values for the referenced attribute set. This means that a delete operation that deletes a tuple from a referenced relation of an inclusion dependency or an update operation that updates attributes of the referenced attribute set of an inclusion dependency was allowed at the local database for at least one of the following two reasons:

1. The referencing relation does not contain tuples that reference the deleted or updated tuple.

2. The referenced relation contains at least one other tuple with the same values for the attributes of the referenced attribute set of this inclusion dependency as the deleted or updated tuple.

The first case is covered by Theorem 4.5.12 to Theorem 4.5.15, but the second case is not addressed yet.

If it is the case that other tuples exists in the referenced relation with the same values for the referenced attribute set as the deleted or updated tuple, there still can be tuples in the referencing relation of the inclusion dependency that reference these other tuples. If another operation would delete or update the attributes of the referenced attributes of the last one of these other tuples, and there still exists tuples in the referencing relation that reference this last tuple, there would be a conflict.

But during synchronization it cannot be determined with certainty if there are still tuples in the referencing relation that reference this last tuple because the database state is unknown, and it is also not possible to determine with certainty whether a tuple is the last tuple with a specific set of values for the referenced attribute set of an inclusion dependency.

The solution is to identify the cases in which any combination of delete and update operations affect tuples from a referenced relation of an inclusion dependency with the same values for the attributes of the referenced attribute set of the inclusion dependency as a possible conflict. For combinations with an update operation, the update operation must update at least one of the attributes of the referenced attribute set of the inclusion dependency. This leads to the following theorems:

THEOREM 4.5.16
There is a possible conflict based on an inclusion dependency violation for two delete operations del1, del2, if rel(del1) = rel(del2) = R, t = tuple(del1), s = tuple(del2), and there exists an inclusion dependency IC with referencedRel(IC) = R, X = referencedAttrib(IC), for which t[X] = s[X].

THEOREM 4.5.17
There is a possible conflict based on an inclusion dependency violation for a delete operation del and an update operation upd, if rel(del) = rel(upd) = R, t = tuple(del), s = tuple(upd), and there exists an inclusion dependency IC with referencedRel(IC) = R, X = referencedAttrib(IC), for which X ⋂ attrib(upd) != ∅ and t[X] = s[X].

THEOREM 4.5.18
There is a possible conflict based on an inclusion dependency violation for two update operations upd1, upd2, if rel(upd1) = rel(upd2) = R, t = tuple(upd1), s = tuple(upd2), and there exists an inclusion dependency IC with referencedRel(IC) = R, X = referencedAttrib(IC), for which X ⋂ attrib(upd1) != ∅, X ⋂ attrib(upd2) != ∅ and t[X] = s[X].

The example data from section 4.3 is not suited to demonstrate a conflict based on one of the previous three theorems, since it does not contain an inclusion dependency. For this reason the example inclusion dependency from chapter three of the book "*Database Systems An Application-Oriented Approach*" is used (Kifer, 2006). This example contains two relations. The `Teaching` relation models which professors are assigned to which courses and in which semester. The `Transcript` relation models which students signed up for a course in a semester. The relation schemas of both relations are:

```
Teaching (
      ProfId: Integer, CrsCode: String, Semester: String
)

Transcript (
      StudId: Integer, CrsCode: String, Semester: String, Grade: String
)
```

The key of the `Teaching` relation is the `<CrsCode, Semester>` attribute pair and the key of the `Transcript` relation is the `<StudId, CrsCode, Semester>` attribute triple. The inclusion dependency between these two relations must enforce that professors do not teach empty classes:

```
Teaching (CrsCode, Semester) REFERENCES Transcript (CrsCode, Semester)
```

Because the referenced attribute set of this inclusion dependency, `<CrsCode, Semester>`, is not a key of the `Transcipt` relation, this is an inclusion dependency instead of a foreign key constraint. Now consider the following data for the `Teaching` and `Transcript` relations:

| ProfId | CrsCode | Semester |
|---|---|---|
| 2001 | EE101 | F2008 |
| 2002 | CS101 | S2009 |

TABLE 4-4 EXAMPLE DATA FOR THE TEACHING RELATION

| StudId | CrsCode | Semester | Grade |
|---|---|---|---|
| 1001 | EE101 | F2008 | A |
| 1002 | EE101 | F2008 | C |
| 1001 | CS101 | S2009 | B |
| 1003 | CS101 | S2009 | B |

TABLE 4-5 EXAMPLE DATA FOR THE TRANSCRIPT RELATION

Also consider the following two delete operations, each performed against a local database:

```
Transcript → Transcript – {<1001, EE101, F2008, A>}
Transcript → Transcript – {<1002, EE101, F2008, C>}
```

The deleted tuples of both operations have the same values for the `<CrsCode, Semester>` referenced attribute set of the inclusion dependency, and according to Theorem 4.5.16 there is a possible conflict when two tuples with the same values for the attributes of the referenced attribute set of an inclusion dependency are deleted from a referenced relation of this inclusion dependency.

In this specific example there actually is a conflict, because the deletion of those two tuples will leave the <2001, EE101, F2008> tuple of the Teaching relation without a referenced tuple in the Transcript relation, because there are no tuples with the values <EE101, F2008> for the referenced attribute set left. But to determine this, the database state is used, and this is not available during synchronization.

## 4.6 DECISION GRAPHS FOR CONFLICTS BETWEEN TWO ATOMIC OPERATIONS

The previous sections of this chapter identified the conflicts and possible conflicts that can occur when synchronizing two arbitrary atomic operations. This section concludes the chapter by presenting three decision graphs that can be used to determine whether two arbitrary atomic operations conflict or not. These decision graphs are based on the theorems presented in the previous sections of this chapter.

Definition 4.1.4 identified two types of conflicts that can occur between two arbitrary atomic operations, conflicts based on unequal database states after synchronization, which were investigated in section 4.4, and conflicts based on violations of integrity constraints, which were investigated in section 4.5. Section 4.5 exists of two parts, section 4.5.1 investigated violations of intra-relational integrity constraints and section 4.5.2 investigated violations of inter-relational integrity constraints.

Each of the three decision graphs in this section covers one of these types of conflicts that can occur between two arbitrary atomic operations. Figure 4.2 covers conflicts based on unequal database states after synchronization, Figure 4.3 covers conflicts based on violations of intra-relational integrity constraints and Figure 4.4 covers conflicts based on violations of inter-relational integrity constraints.

The decision graphs take two arbitrary atomic operations as input. The constraint language used throughout this chapter is used in the decision nodes of the graphs. An overview of the constraint language can be found in Appendix A.

The decision graphs contain two types of decision nodes:

- Binary decision nodes for yes/no decisions based on
    - the application of a theorem presented in the section of this chapter covered by the decision graph
    - an equality statement to check preconditions of theorems presented in the section of this chapter covered by the decision graph, for example whether the two atomic operations affect the same relation or not
- Ternary decision nodes for checking the type of an operation

Because only binary and ternary decision nodes are used in the graphs it is straightforward to check that the decision graphs are complete.

The end nodes of the decision graphs correspond with the following three statements:

- there is a conflict for the two atomic operations based on the type of conflict covered by the decision graph

- there is a possible conflict for the two atomic operations based on the type of conflict covered by the decision graph
- there is no conflict for the two atomic operations based on the type of conflict covered by the decision graph

In all the decision graphs the end nodes that state that there are no conflicts can only be reached if all conflict and possible conflict end nodes are ruled out, and the end nodes for possible conflicts can only be reached if all conflict end nodes are ruled out. This ensures that conflicts will always be identified before possible conflicts.

In each decision graph the theorems covered by that decision graph are indicated, either in the caption of a decision node or as a label attached to an arc leading to an end node. This makes it straightforward to check that all theorems are indeed covered by the decision graph. If multiple paths can lead to the application of a theorem, the theorem is indicated once for each possible path.

Because each decision graph in this section covers a certain type of conflict that can occur between two arbitrary atomic operations, it is necessary to use all three decision graphs to determine if there is a conflict between two arbitrary atomic operations. Based on the results of the decision graphs for two arbitrary atomic operations a statement can be made about whether these two arbitrary atomic operations conflict with each other or not.

The following theorem can be used to determine if two arbitrary atomic operations conflict with each other by examining the results of the three decision graphs:

THEOREM 4.6.1

To determine for two arbitrary atomic operations $op_1$ and $op_2$ if there is a conflict, a possible conflict or no conflict, the results of the decision graphs of figures Figure 4.2, Figure 4.3 and Figure 4.4 with $op_1$ and $op_2$ as input should be examined.

Based on the results of the decision graphs for $op_1$ and $op_2$, one of three statements can be made regarding conflicts between $op_1$ and $op_2$:

1. If one or more of the decision graphs identify a conflict between $op_1$ and $op_2$, there is a conflict between $op_1$ and $op_2$.
2. If none of the decision graphs identify a conflict between $op_1$ and $op_2$, but one or more of the decision graphs identify a possible conflict between $op_1$ and $op_2$, there is a possible conflict between $op_1$ and $op_2$.
3. If none of the decision graphs identify a conflict or a possible conflict between $op_1$ and $op_2$, $op_1$ and $op_2$ harmonize.

Based on the assumption that the reasoning presented in the sections 4.4, 4.5.1 and 4.5.2 identifies all conflicts and possible conflicts that can occur between two arbitrary atomic operations, and the fact that the decision graphs are complete and cover all theorems presented in those sections, it can be concluded that Theorem 4.6.1 makes a correct statement regarding conflicts between two arbitrary atomic operations.

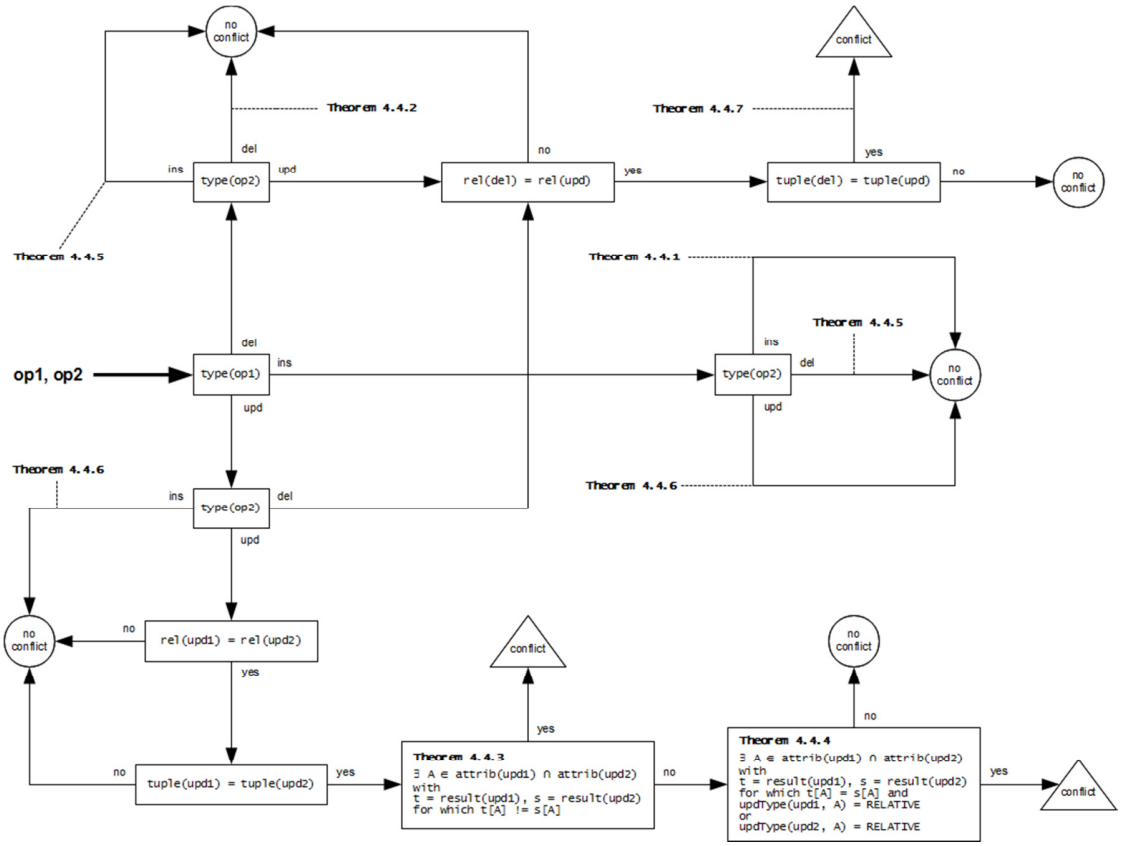


FIGURE 4.2 DECISION GRAPH FOR IDENTIFYING CONFLICTS BETWEEN TWO ARBITRARY ATOMIC OPERATIONS BASED ON UNEQUAL DATABASE STATES AFTER SYNCHRONIZATION

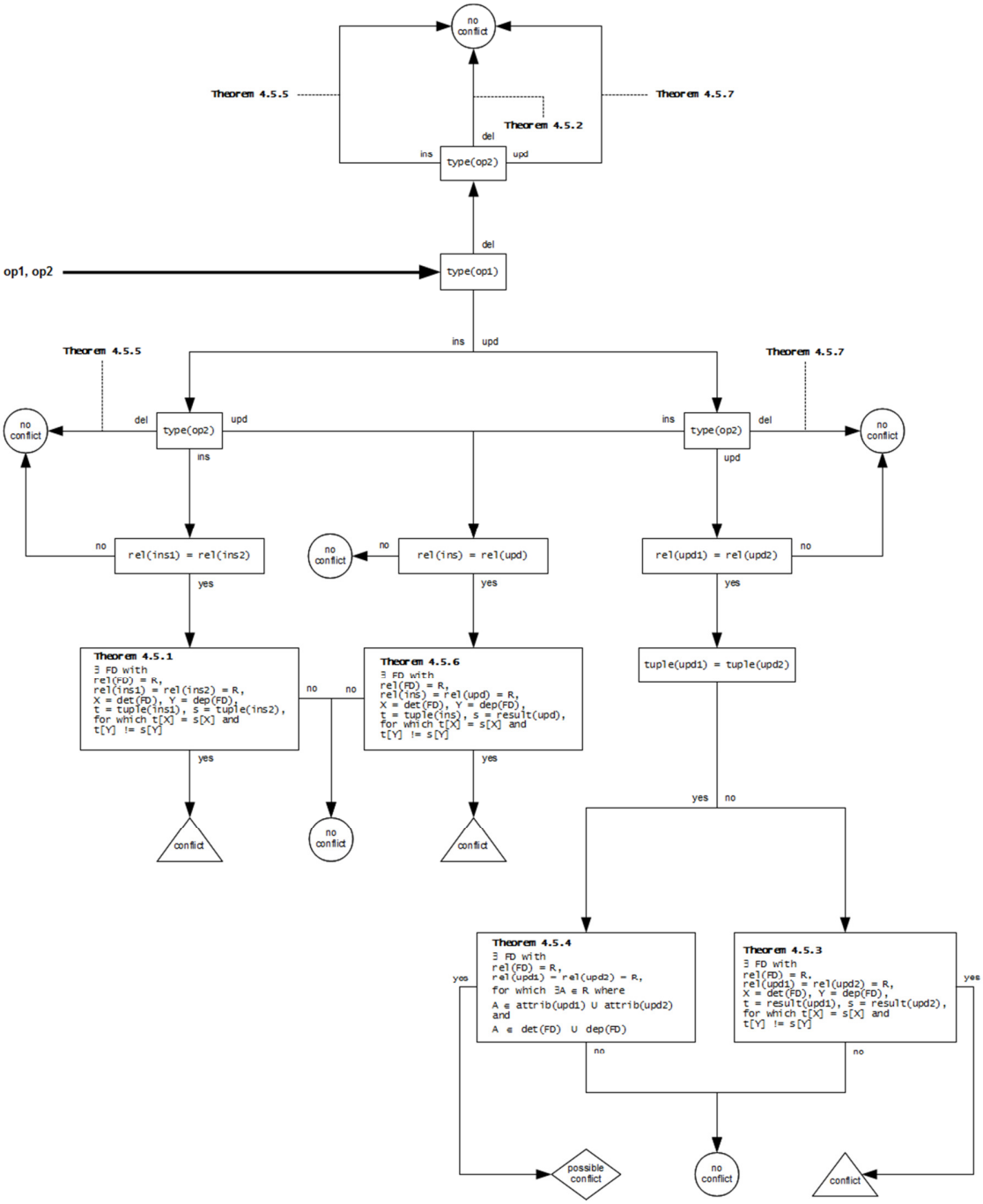


FIGURE 4.3 DECISION GRAPH FOR IDENTIFYING CONFLICTS BETWEEN TWO ARBITRARY ATOMIC OPERATIONS BASED ON VIOLATIONS OF INTRA-RELATIONAL INTEGRITY CONSTRAINTS

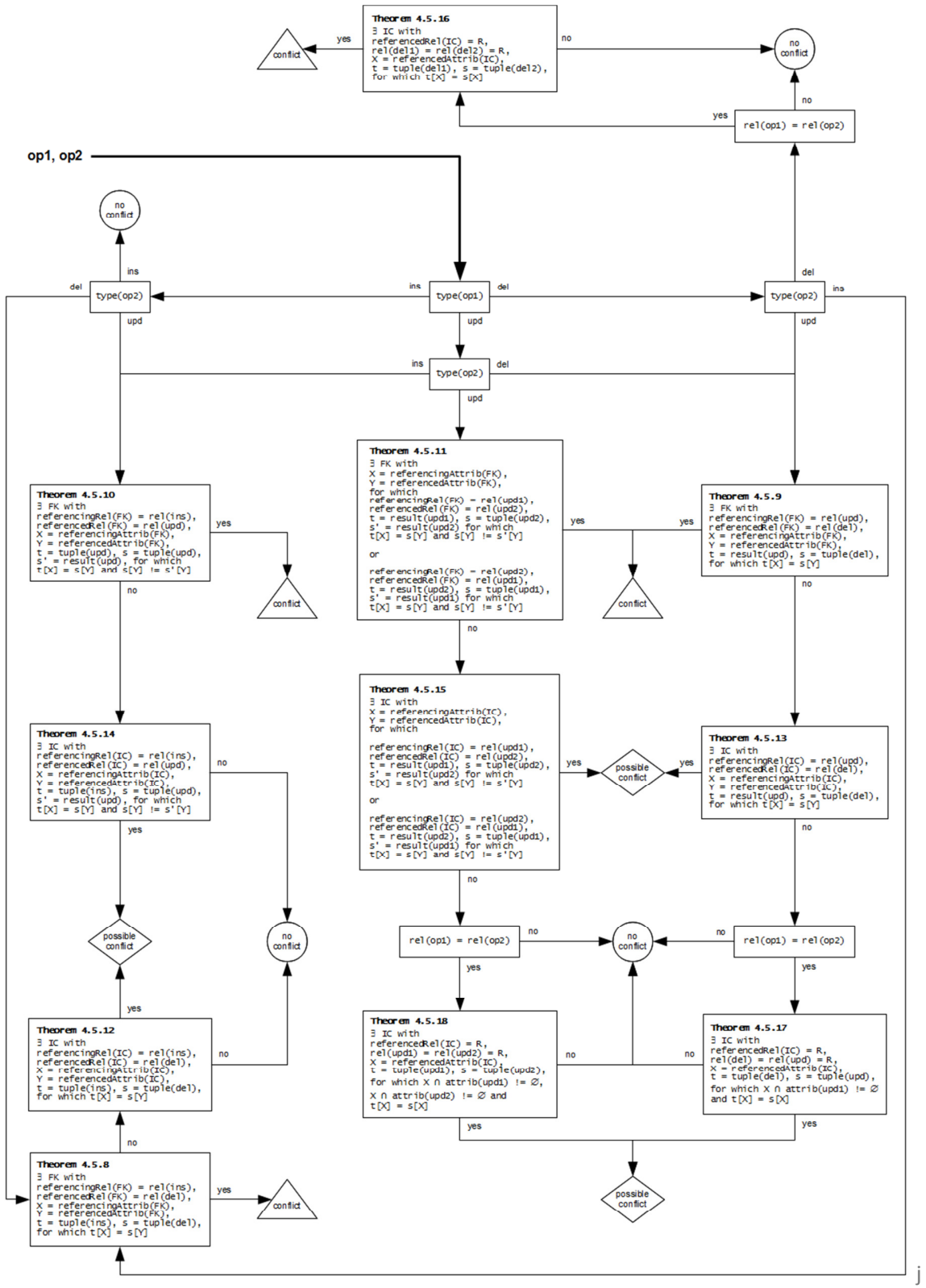


FIGURE 4.4 DECISION GRAPH FOR IDENTIFYING CONFLICTS BETWEEN TWO ARBITRARY ATOMIC OPERATIONS BASED ON VIOLATIONS OF INTER-RELATIONAL INTEGRITY CONSTRAINTS

# 5 SYNCHRONIZATION OF COMPOSITE OPERATIONS

This chapter introduces the concept of composite operations. A composite operation is a sequence of atomic operations that were executed at the same local database and that should be grouped together and treated as a single unit of work. There are two main reasons for introducing composite operations.

The first reason to introduce composite operations is that the previous chapter discussed conflicts for atomic operations that affect a single tuple. But a single database query can affect multiple tuples, and a synchronization solution must treat this query as a single unit of work. So a query that affects multiple tuples can be written out to the tuple level, and the atomic operations that are the result of this should be grouped together in a composite operation.

As an example consider the following query executed against the example data of section 4.3:

```
UPDATE Student SET instructor = 1 WHERE instructor = 2
```

Because there are two tuples in the `Student` relation with a value of `2` for the `instructor` attribute, this query affects two tuples. Writing this out to the tuple level and grouping the atomic operations in a composite operation would lead to the following composite operation:

```
(
Student ← Student –
      {<4, 2, George, Yellow, Spuistraat, 1, Den Haag, 2511 BC, 10.00>}
      ∪
      {<4, 1, George, Yellow, Spuistraat, 1, Den Haag, 2511 BC, 10.00>}
,
Student ← Student –
      {<5, 2, Carol, Green, Steenstraat, 23, Arnhem, 6821 DL, 75.00>}
      ∪
      {<5, 1, Carol, Green, Steenstraat, 23, Arnhem, 6821 DL, 75.00>}
)
```

The other reason to introduce composite operations is to be able to group together multiple database operations that semantically belong together and should also be treated as a single unit of work; this is similar to the transaction concept for database systems. Either all atomic operations in a composite operation should be synchronized or none of them at all.

Composite operations will be the unit of work a synchronization solution will work with. This means that conflicts between composite operations must also be addressed. Section 5.1 presents a definition of composite operations and also discusses how to identify conflicts between composite operations.

Section 5.2 introduces the concept of related atomic and composite operations. Atomic operations can have a dependency on other atomic operations if they affect tuples that are the result of another atomic operation that was executed earlier at the same local database. If a conflict is identified for an atomic or composite operation, this atomic operation cannot be synchronized, and the same should be true for all

atomic or composite operations that have a dependency on this conflicting operation, because otherwise the dependency will be violated.

This chapter concludes with section 5.3, which introduces the composite operations conflict graph. This is a graph that displays the conflicting and related composite operations for two sequences of composite operations executed at different local databases. The composite operations conflict graph can be used by a synchronization solution as a tool to identify conflicting and related composite operations.

## 5.1 COMPOSITE OPERATIONS

Section 5.1.1 starts with the definition of a composite operation and the definition of the atomicity property for composite operations. This definition of a composite operation is used in section 5.1.2 to define conflicts and possible conflicts between two arbitrary composite operations. This section concludes with section 5.1.3 which presents an algorithm that can be used to determine if two arbitrary composite operations conflict with each other.

### 5.1.1 DEFINITION OF A COMPOSITE OPERATION

A composite operation is a sequence of atomic operations executed at a local database that should be treated as a single unit of work. The order of the atomic operations in this sequence is the order in which the atomic operations were executed at the local database.

DEFINITION 5.1.1 COMPOSITE OPERATION

A composite operation is a sequence of n atomic operations ($op_{L,1}$, $op_{L,2}$, …, $op_{L,n}$), with $n > 0$, which were executed at the same local database L. The order of the atomic operations in the composite operation is the order in which the atomic operations were executed at the local database.

Composite operations must be treated as a single unit of work. This is a property of composite operations that must be ensured by a synchronization solution. This property is similar to the atomicity property of transactions in a database system. Kifer, Bernstein and Lewis (Kifer, 2006) define atomicity for database transactions as:

> *The database system must ensure that the transaction either runs to completion or, if it does not complete, has no effect at all (as if it had never been started).*

Adapting this definition of atomicity for database transactions to atomicity for composite operations leads to the following definition of atomicity for composite operations:

DEFINITION 5.1.2 ATOMICITY PROPERTY FOR COMPOSITE OPERATIONS

A synchronization solution must ensure that either all atomic operations of a composite operation are synchronized or none of them at all.

### 5.1.2 DEFINITION OF A CONFLICT FOR TWO COMPOSITE OPERATIONS

Definition 5.1.1 defines a composite operation as a sequence of atomic operations that were executed at the same local database. Because the composite operations were already allowed at the local database it is known that there are no conflicts between atomic operations belonging to the same

composite operation. However, it is still possible that there are conflicts between two atomic operations belonging to different composite operations.

If two atomic operations conflict, they cannot be synchronized. This also means that the composite operations to which these two atomic operations belong cannot be synchronized, because of the atomicity property of Definition 5.1.2. This means a conflict occurs between two composite operations if there is at least one atomic operation of a composite operation that conflicts with an atomic operation of the other composite operation:

DEFINITION 5.1.3 CONFLICT BETWEEN TWO COMPOSITE OPERATIONS

There is a conflict between two arbitrary composite operations $C_1$ and $C_2$, with $C_1 = (op_{1,1}, op_{1,2}, \ldots, op_{1,m}\}$ and $C_2 = (op_{2,1}, op_{2,2}, \ldots, op_{2,n})$, if there exists a pair of atomic operations $op_{1,i} \in C_1$ with $1 \leq i \leq m$ and $op_{2,j} \in C_2$, with $1 \leq j \leq n$, for which there is a conflict.

Besides conflicts between two arbitrary atomic operations, chapter 4 also identified possible conflicts between two arbitrary atomic operations. This means that it is also possible that there exists a pair of atomic operations between two composite operations for which there is a possible conflict, which means that it is also possible that there is a possible conflict between two composite operations. But cases where there are both pairs of conflicting and possibly conflicting atomic operations between two composite operations, these two composite operations should be identified as conflicting composite operations.

DEFINITION 5.1.4 POSSIBLE CONFLICT BETWEEN TWO COMPOSITE OPERATIONS

There is a possible conflict between two arbitrary composite operations $C_1$ and $C_2$, with $C_1 = (op_{1,1}, op_{1,2}, \ldots, op_{1,m}\}$ and $C_2 = (op_{2,1}, op_{2,2}, \ldots, op_{2,n})$, if there is no conflict between $C_1$ and $C_2$ and there exists a pair of atomic operations $op_{1,i} \in C_1$ with $1 \leq i \leq m$ and $op_{2,j} \in C_2$, with $1 \leq j \leq n$, for which there is a possible conflict.

### 5.1.3 IDENTIFYING CONFLICTS BETWEEN TWO COMPOSITE OPERATIONS

Developing a method to identify conflicts and possible conflicts between two arbitrary composite operations is a straightforward procedure using Definition 5.1.3 and Definition 5.1.4. All possible pairs of atomic operations between the two composite operations should be checked for conflicts or possible conflicts, and Theorem 4.6.1 can be used for this. Four possible outcomes can be distinguished:

1)
   a) There is at least one pair of atomic operations between the two composite operations for which there is a conflict,
   b) and there also is at least one pair of atomic operations between the two composite operations for which there is a possible conflict.

2)
   a) There is at least one pair of atomic operations between the two composite operations for which there is a conflict,
   b) and there does not exist a pair of atomic operations between the two composite operations for which there is a possible conflict.

3)
   a) There does not exist a pair of atomic operations between the two composite operations for which there is a conflict,
   b) and there is at least one pair of atomic operations between the two composite operations for which there is a possible conflict.
4)
   a) There does not exist a pair of atomic operations between the two composite operations for which there is a conflict,
   b) and there also does not exist a pair of atomic operations between the two composite operations for which there is a possible conflict.

If the first or second case occurs, there is a conflict between the two composite operations according to Definition 5.1.3. If the third case occurs there is a possible conflict between the two composite operations according to Definition 5.1.4. If the fourth case occurs, the two composite operations harmonize.

Algorithm 5.1.1 can be used to determine if there is a conflict, a possible conflict or no conflict between two arbitrary composite operations.

ALGORITHM 5.1.1

For two arbitrary composite operations $C_1$ and $C_2$, with $C_1 = (op_{1,1}, op_{1,2}, \ldots, op_{1,m})$ and $C_2 = (op_{2,1}, op_{2,2}, \ldots, op_{2,n})$, the following algorithm can be used to determine if there is a conflict, a possible conflict or no conflict between these two composite operations:

```
result = NO CONFLICT;

for 1 ≤ i ≤ m:
   for 1 ≤ j ≤ n:
      if (conflict(op1,i, op2,j)):
            return CONFLICT;
         else if (possible_conflict(op1,i, op2,j)):
         result = POSSIBLE CONFLICT;

return result;
```

Algorithm 5.1.1 introduces two new functions, `conflict(op, op)` and `possible_conflict(op, op)`. The function `conflict(op, op)` returns true if Theorem 4.6.1 identifies a conflict for the two atomic operations passed as arguments to this function, and returns false otherwise. The function `possible_conflict(op, op)` returns true if Theorem 4.6.1 identifies a possible conflict for the two atomic operations passed as arguments to this function, and returns false otherwise.

The remainder of this section contains two examples of the application of Algorithm 5.1.1 to detect conflicts between two composite operations. The first example applies Algorithm 5.1.1 to two composite operations $C_1$ and $C_2$ for which there is no conflict. The atomic operations of composite operation $C_1$ in this example are:

```
op1,1:
    Student ← Student −
      {<5, 2, Carol, Green, Steenstraat, 23, Arnhem, 6821 DL, 75.00>}
op1,2:
    Student ← Student ∪
      {<7, 2, Hank, Gray, Graafseweg, 8, Nijmegen, 6541 LB, 10.00>}
```

The atomic operations of composite operation $C_2$ in this example are:

```
op2,1:
    Student ← Student −
      {<1, 1, Joe, White, Kalverstraat, 5, Amsterdam, 1071 EX, 50.00>}
      ∪
      {<1, 1, Joe, White, Kalverstraat, 5, Amsterdam, 1071 EX, 0.00>}

op2,2:
    Student ← Student ∪
      {<6, 2, Frank, Red, Grote Markt, 33, Groningen, 9731 CG, 25.00>}
```

Table 5-1 shows all the iterations of the Algorithm 5.1.1 if applied to the composite operations $C_1$ and $C_2$. For each of the iterations the results of the functions `conflict(op1,i, op2,j)` and `possible_conflict(op1,i, op2,j)` and the value of the `result` variable are shown.

| i | j | $op_{1,i}$ | $op_{2,j}$ | conflict($op_{1,i}$, $op_{2,j}$) | possible_conflict($op_{1,i}$, $op_{2,j}$) | result |
|---|---|---|---|---|---|---|
| 1 | 1 | $op_{1,1}$ | $op_{2,1}$ | false | false | NO CONFLICT |
| 1 | 2 | $op_{1,1}$ | $op_{2,2}$ | false | false | NO CONFLICT |
| 2 | 1 | $op_{1,2}$ | $op_{2,1}$ | false | false | NO CONFLICT |
| 2 | 2 | $op_{1,2}$ | $op_{2,2}$ | false | false | NO CONFLICT |

TABLE 5-1 THE ITERATIONS OF ALGORITHM 5.1.1 WHEN APPLIED TO THE TWO NON-CONFLICTING EXAMPLE COMPOSITE OPERATIONS $C_1$ AND $C_2$

Because there are no pairs of atomic operations between the composite operations $C_1$ and $C_2$ for which there is a conflict or possible conflict, Algorithm 5.1.1 determines that there is no conflict between the composite operations $C_1$ and $C_2$.

The next example applies Algorithm 5.1.1 to two other composite operations $C_1$ and $C_2$ for which there is a conflict. The atomic operations of composite operation $C_1$ in this example are:

```
op1,1:
    Student ← Student ∪
      {<6, 2, Frank, Red, Grote Markt, 33, Groningen, 9731 CG, 25.00>}
op1,2:
    Student ← Student ∪
      {<7, 2, Hank, Gray, Graafseweg, 8, Nijmegen, 6541 LB, 10.00>}
```

The atomic operations of $C_2$ are:

```
op2,1:
    Student ← Student −
      {<5, 2, Carol, Green, Steenstraat, 23, Arnhem, 6821 DL, 75.00>}
op2,2:
    Student ← Student ∪
      {<6, 2, Frank, Red, Grote Markt, 33, Groningen, 9731 CG, 10.00>}
```

Table 5-2 shows all the iterations of Algorithm 5.1.1 if applied to the composite operations $C_1$ and $C_2$.

| i | j | $op_{1,i}$ | $op_{2,j}$ | conflict($op_{1,i}$, $op_{2,j}$) | possible_conflict($op_{1,i}$, $op_{2,j}$) | result |
|---|---|---|---|---|---|---|
| 1 | 1 | $op_{1,1}$ | $op_{2,1}$ | false | false | NO CONFLICT |
| 1 | 2 | $op_{1,1}$ | $op_{2,2}$ | true | false | CONFLICT |

TABLE 5-2 THE ITERATIONS OF ALGORITHM 5.1.1 WHEN APPLIED TO THE TWO CONFLICTING EXAMPLE COMPOSITE OPERATIONS $C_1$ AND $C_2$

In the second iteration of the algorithm the function conflict($op_{1,i}$, $op_{2,j}$) detects a conflict between the atomic operations $op_{1,1}$ and $op_{2,2}$ based on a functional dependency violation, identified by Theorem 4.5.1. Because a pair of conflicting atomic operations is found between the composite operations $C_1$ and $C_2$, there is a conflict between these two composite operations according to Definition 5.1.3, and Algorithm 5.1.1 terminates with the result that there is a conflict between the composite operations $C_1$ and $C_2$.

## 5.2 DEPENDENCIES BETWEEN COMPOSITE OPERATIONS

Definition 4.2.2 of the atomic delete operation states that the tuple deleted by a delete operation must exist in the relation the delete operation is executed against. Definition 4.2.3 of the atomic update operation states that the tuple updated by an update operation must exist in the relation the update operation is executed against. For the atomic insert operation there is not such a requirement. From these requirements can be concluded that the atomic operations delete and update have a dependency on the tuples they affect, because these operation are not valid if they operate on a database state in which the tuples they affect do not exist.

Section 5.2.1 investigates the dependencies between atomic operations and introduces the concept of related atomic operations. Section 5.2.2 investigates how the concept of related atomic operations relates to composite operations.

### 5.2.1 RELATED ATOMIC OPERATIONS

This section starts by defining a dependency between two atomic operations and related atomic operations. These definitions will be used to present several theorems that will cover the possible dependencies between two arbitrary atomic operations. This section concludes with a decision graph based on the presented theorems which can be used to determine if two arbitrary atomic operations executed at the same local database are related.

As stated in the introduction of section 5.2 the atomic operations delete and update have a dependency on the tuples they affect, because the definitions of these atomic operations have the requirement that the tuple they affect must exist in the relation they operate on. This dependency of the atomic operations delete and update on the tuples they affect can be translated into a dependency on other atomic operations that were executed earlier at the same local database. This is best explained with an example.

Consider the following two atomic operations, an update and delete operation performed against the example `Student` relation of section 4.3, where the update operation was performed before the delete operation:

```
Student ← Student –
      {<5, 2, Carol, Green, Steenstraat, 23, Arnhem, 6821 DL, 75.00} ∪
      {<5, 2, Carol, Green, Steenstraat, 23, Arnhem, 6821 DL, 50.00>}


Student ← Student –
      {<5, 2, Carol, Green, Steenstraat, 23, Arnhem, 6821 DL, 50.00>}
```

In this example the delete operation deletes the tuple that is the result of the update operation, so the dependency of the delete operation in this example on the tuple it deletes can also be stated as a dependency on the update operation. In the same way the update operation in this example could have a dependency on another atomic operation performed earlier at the same local database. If a dependency exists between two atomic operations, these two atomic operations are said to be related.

DEFINITION 5.2.1 DEPENDENCY BETWEEN ATOMIC OPERATIONS

An atomic operation $op_1$ has a dependency on another atomic operation $op_2$ if $op_1$ is executed after $op_2$ at the same local database and $op_1$ affects the tuple that is the result of $op_2$.

DEFINITION 5.2.2 RELATED ATOMIC OPERATIONS

Two arbitrary atomic operations $op_1$ and $op_2$ are related if $op_1$ has a dependency on $op_2$ or if $op_2$ has a dependency on $op_1$.

From Definition 5.2.1 and the definitions of the atomic operations follows that the atomic operations delete and update can have a dependency on other atomic operations executed earlier at the same local database because they have the requirement that the tuple they affect exists in the database state they operate on. The atomic insert operation cannot have a dependency on another atomic operation performed earlier at the same local database, because there is not such a requirement for insert operations.

Dependencies for the atomic delete and update operations are based on the tuples they affect, which means that delete and update operations can only have a dependency on an insert or an update operation and cannot have a dependency on a delete operation, because delete operations remove the tuples they affect from the database state they operate on. This means that there are four types of dependencies between atomic operations that can be distinguished:

- A delete operation that has a dependency on an insert operation that was executed earlier at the same local database.
- A delete operation that has a dependency on an update operation that was executed earlier at the same local database.
- An update operation that has a dependency on an insert operation that was executed earlier at the same local database.
- An update operation that has a dependency on another update operation that was executed earlier at the same local database.

A delete operation has a dependency on an insert operation that was executed at the same local database if the insert operation was executed before the delete operation, and the delete operation deletes the tuple that was inserted by the insert operation:

THEOREM 5.2.1
A delete operation del has a dependency on an insert operation ins if del and ins were both executed at the same local database, ins was executed before del, rel(del) = rel(ins) and tuple(del) = tuple(ins).

A delete operation has a dependency on an update operation that was executed at the same local database if the update operation was executed before the delete operation, and the delete operation deletes the tuple that was the result of the update operation:

THEOREM 5.2.2
A delete operation del has a dependency on an update operation upd if del and upd were both executed at the same local database, upd was executed before del, rel(del) = rel(upd) and tuple(del) = result(upd).

An update operation has a dependency on an insert operation that was executed at the same local database if the insert operation was executed before the update operation, and the update operation updates the tuple that was inserted by the insert operation:

THEOREM 5.2.3
An update operation upd has a dependency on an insert operation ins if upd and ins were both executed at the same local database, ins was executed before upd, rel(upd) = rel(ins) and and tuple(upd) = tuple(ins).

An update operation has a dependency on another update operation that was executed at the same local database if the other update operation was executed before the update operation, and it updates the tuple that was the result of the other update operation:

THEOREM 5.2.4
An update operation upd1 has a dependency on another update operation upd2 if upd1 and upd2 were executed at the same local database, upd2 was executed before upd1, rel(upd1) = rel(upd2) and and tuple(upd1) = result(upd2).

Definition 5.2.2 states that two atomic operations are related if one of these atomic operations has a dependency on the other atomic operation. Figure 5.1 presents a decision graph to determine if two arbitrary atomic operations are related. This decision graph is based on the theorems for dependencies between two atomic operations presented in this section. The decision graph assumes that the atomic operations that are provided as input to the graph, $op_1$ and $op_2$, were executed at the same local

database, and that $op_1$ was executed before $op_2$. The decision graph determines that the two atomic operations are related if a dependency is found between the two atomic operations provided as input to the graph, and determines that the two atomic operations are not related if no dependency is found between the two atomic operations provided as input to the graph.

Based on the assumption that the reasoning in this section identifies all possible relations between two arbitrary atomic operations, and the fact that the decision graph of Figure 5.1 is complete and covers all theorems presented in this section, it can be concluded that this decision graph can be used to determine if two arbitrary atomic operations are related.

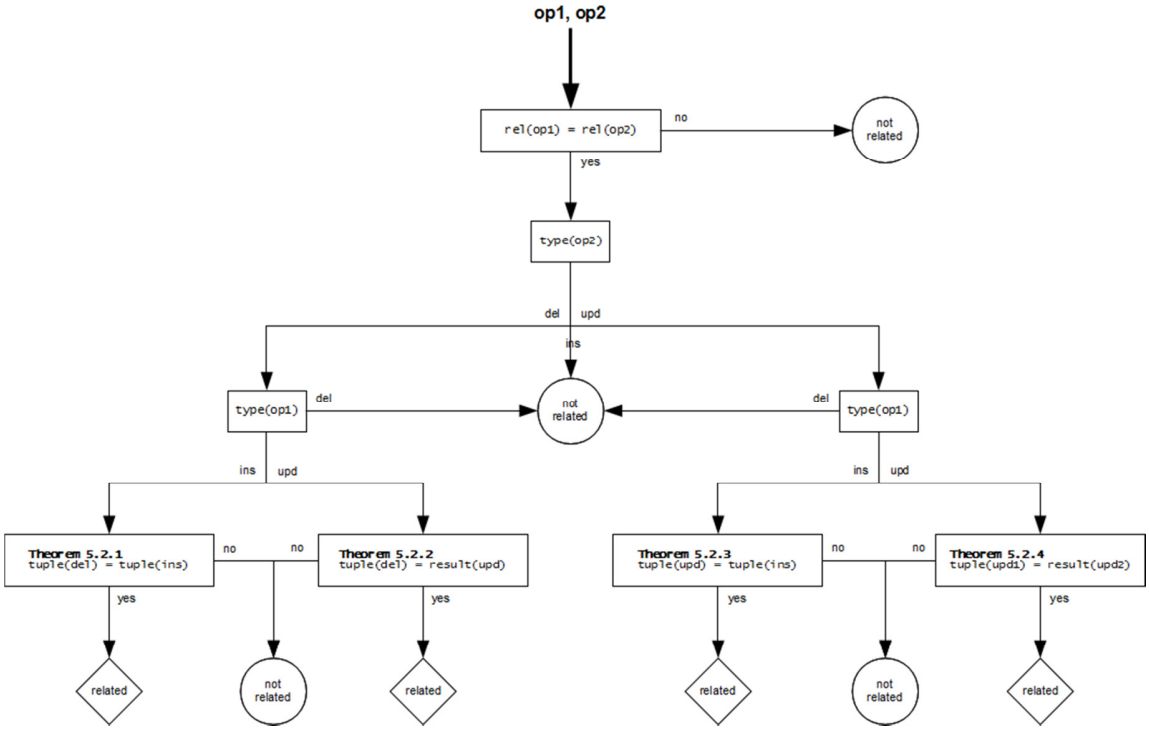


FIGURE 5.1 DECISION GRAPH TO DETERMINE IF TWO ATOMIC OPERATIONS ARE RELATED. THIS GRAPH ASSUMES THAT THE OPERATIONS $OP_1$ AND $OP_2$ WERE EXECUTED AT THE SAME LOCAL DATABASE, AND THAT $OP_1$ WAS EXECUTED BEFORE $OP_2$.

### 5.2.2 RELATED COMPOSITE OPERATIONS

Definition 5.1.3 states that there is a conflict for two arbitrary composite operations if there exists a pair of conflicting atomic operations between the two composite operations. This is based on the atomicity property of composite operations defined in Definition 5.1.2. The same reasoning holds for related composite operations. Two arbitrary composite operations that were executed at the same local database are related if there exists a pair of related atomic operations between these two composite operations.

#### DEFINITION 5.2.3 RELATED COMPOSITE OPERATIONS

Two arbitrary composite operations $C_1$ and $C_2$, with $C_1 = (op_{1,1}, op_{1,2}, \ldots, op_{1,m}\}$ and $C_2 = (op_{2,1}, op_{2,2}, \ldots, op_{2,n})$, are related if there exists a pair of atomic operations $op_{1,i} \in C_1$ with $1 \leq i \leq m$ and $op_{2,j} \in C_2$, with $1 \leq j \leq n$ that are related.

Algorithm 5.2.1 can be used to determine if two arbitrary composite operations that were executed at the same local database are related. This algorithm checks all pairs of atomic operations between two composite operations that were executed at the same local database to see if they are related. If a related pair of atomic operations is found the two composite operations are related, and if no pair of related atomic operations is found the two composite operations are not related.

ALGORITHM 5.2.1

For two arbitrary composite operations $C_1$ and $C_2$ executed at the same local database, with $C_1 = (op_{1,1}, op_{1,2}, \ldots, op_{1,m}\}$, $C_2 = (op_{2,1}, op_{2,2}, \ldots, op_{2,n})$ and $C_1$ executed before $C_2$, the following algorithm can be used to determine if these two composite operations are related:

```
for 1 ≤ i ≤ m:
  for 1 ≤ j ≤ n:
    if (related(op1,i, op2,j)):
      return RELATED;

return NOT_RELATED;
```

Algorithm 5.2.1 introduces the function `related(op, op)`. This function returns true if the decision graph presented in Figure 5.1 in section 5.2.1 determines that the two atomic operations passed as arguments to the function are related, and returns false if they are not related. Like the decision graph in Figure 5.1, the function `related(op, op)` also assumes that the two operations passed as arguments to the function were performed at the same local database, and that the atomic operation passed as the first argument to the function was executed before the atomic operation passed as the second argument to the function.

The remainder of this section contains two examples of the application of Algorithm 5.2.1 to determine if two composite operations are related. The first example applies Algorithm 5.2.1 to two composite operations $C_1$ and $C_2$ that are not related. Both composite operations were executed at the same local database and $C_1$ was executed before $C_2$. The atomic operations of composite operation $C_1$ are:

```
op1,1:
    Student ← Student −
      {<5, 2, Carol, Green, Steenstraat, 23, Arnhem, 6821 DL, 75.00>}
op1,2:
    Student ← Student ∪
      {<7, 2, Hank, Gray, Graafseweg, 8, Nijmegen, 6541 LB, 10.00>}
```

The atomic operations of composite operation $C_2$ are:

```
op2,1:
    Student ← Student −
      {<1, 1, Joe, White, Kalverstraat, 5, Amsterdam, 1071 EX, 50.00>}
      ∪
      {<1, 1, Joe, White, Kalverstraat, 5, Amsterdam, 1071 EX, 0.00>}
```

```
op2,2:
    Student ← Student ∪
      {<6, 2, Frank, Red, Grote Markt, 33, Groningen, 9731 CG, 25.00>}
```

Table 5-3 shows all the iterations of Algorithm 5.2.1 if applied to the composite operations $C_1$ and $C_2$. For each of the iterations the result of the function related($op_{1,i}$, $op_{2,j}$) and the result are shown.

| i | j | $op_{1,i}$ | $op_{2,j}$ | related($op_{1,i}$, $op_{2,j}$) | result |
|---|---|---|---|---|---|
| 1 | 1 | $op_{1,1}$ | $op_{2,1}$ | false | NOT RELATED |
| 1 | 2 | $op_{1,1}$ | $op_{2,2}$ | false | NOT RELATED |
| 2 | 1 | $op_{1,2}$ | $op_{2,1}$ | false | NOT RELATED |
| 2 | 2 | $op_{1,2}$ | $op_{2,2}$ | false | NOT RELATED |

TABLE 5-3 THE ITERATIONS OF ALGORITHM 5.2.1 WHEN APPLIED TO TWO COMPOSITE OPERATIONS $C_1$ AND $C_2$ THAT ARE NOT RELATED

Because there are no pairs of related atomic operations between the composite operations $C_1$ and $C_2$, Algorithm 5.2.1 determines that the composite operations $C_1$ and $C_2$ are not related.

The next example applies Algorithm 5.2.1 to two other composite operations $C_1$ and $C_2$ that are related. Composite operation $C_1$ in this example exists of a single atomic operation:

```
op1,1:
    Student ← Student −
      {<5, 2, Carol, Green, Steenstraat, 23, Arnhem, 6821 DL, 75.00}
      ∪
      {<5, 2, Carol, Green, Steenstraat, 23, Arnhem, 6821 DL, 50.00>}
```

The atomic operations of composite operation $C_2$ are:

```
op2,1:
    Student ← Student ∪
      {<6, 2, Frank, Red, Grote Markt, 33, Groningen, 9731 CG, 25.00>}

op2,2:
    Student ← Student −
      {<5, 2, Carol, Green, Steenstraat, 23, Arnhem, 6821 DL, 50.00>}
```

Table 5-4 shows all the iterations of Algorithm 5.2.1 if applied to the composite operations $C_1$ and $C_2$.

| i | j | $op_{1,i}$ | $op_{2,j}$ | related($op_{1,i}$, $op_{2,j}$) | result |
|---|---|---|---|---|---|
| 1 | 1 | $op_{1,1}$ | $op_{2,1}$ | false | NOT RELATED |
| 1 | 2 | $op_{1,1}$ | $op_{2,2}$ | true | RELATED |

TABLE 5-4 THE ITERATIONS OF ALGORITHM 5.2.1 WHEN APPLIED TO TWO COMPOSITE OPERATIONS $C_1$ AND $C_2$ THAT ARE RELATED

In the second iteration of the algorithm the function related($op_{1,i}$, $op_{2,j}$) determines that the atomic operations $op_{1,1}$ and $op_{2,2}$ are related based on Theorem 5.2.2, because $op_{2,2}$ deletes the tuple that is the result of the update operation $op_{1,1}$. Because a pair of related atomic operations is found

between the composite operations $C_1$ and $C_2$, Algorithm 5.2.1 terminates with the result that the two composite operations $C_1$ and $C_2$ are related.

## 5.3 COMPOSITE OPERATIONS CONFLICT GRAPH

This section presents the composite operations conflict graph, which is a graph that displays conflicting, possibly conflicting and related composite operations that exist for two sequences of composite operations executed at two different local databases. The composite operations conflict graph can be used as a tool by a synchronization solution to identify which composite operations are related or conflict with each other.

The visual representation of a composite operations conflict graph is divided in two sides, with each side representing a local database. The vertices in the composite operations conflict graph represent the composite operations, and at each side of the composite operations conflict graph the vertices are placed from top to bottom, in the order in which the composite operations were executed at the local databases. Arcs between the vertices indicate conflicting, possible conflicting and related composite operations. For an example composite operations conflict graph see Figure 5.4 at the end of this section. The remainder of this section presents the definition of a composite operations conflict graph, Definition 5.1.3, and works through an example of how to create a composite operations conflict graph for two sequences of composite operations executed at two different local databases.

A composite operations conflict graph is a mixed graph (Types of graphs), which is a graph that contains both directed and undirected edges, with undirected edges representing conflicting and possible conflicting composite operations and directed edges representing related composite operations. The composite operations conflict graph is defined as:

#### DEFINITION 5.3.1 COMPOSITE OPERATIONS CONFLICT GRAPH

A composite operations conflict graph is defined over sequences of composite operations that were executed at two local databases DB1 and DB2, $(C_{1,1}, C_{1,2}, \ldots, C_{1,m})$ executed at DB1 and $(C_{2,1}, C_{2,2}, \ldots, C_{2,n})$ executed at DB2.

A composite operations conflict graph G is an ordered quintuple $G = (V_1, V_2, E_C, E_{PC}, R)$ where

> $V_1$ is a sequence of vertices $(C_{1,1}, C_{1,2}, \ldots, C_{1,m})$ representing the composite operations that were executed at local database DB1, in the order in which they were executed at local database DB1
>
> $V_2$ is a sequence of vertices $(C_{2,1}, C_{2,2}, \ldots, C_{2,n})$ representing the composite operations that were executed at local database DB2, in the order in which they were executed at local database DB2
>
> $E_C$ is a set of undirected edges, which are unordered pairs of vertices $(C_{1,i}, C_{2,j})$ with $C_{1,i} \in V_1$ and $C_{2,j} \in V_2$, representing conflicting composite operations
>
> $E_{PC}$ is a set of undirected edges, which are unordered pairs of vertices $(C_{1,i}, C_{2,j})$ with $C_{1,i} \in V_1$ and $C_{2,j} \in V_2$, representing possibly conflicting composite operations
>
> R is a set of directed edges, which are ordered pairs of vertices $(C_{L,i}, C_{L,j})$ with $C_{L,i}$ and $C_{L,j} \in V_L$ and $i < j$, representing related composite operations

The example in this section that shows how to create a composite operations conflict graph uses the example data of the example `Student` and `Instructor` relations of section 4.3:

| id | instructor | first_name | last_name | street | number | City | zip_code | balance |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | Joe | White | Kalverstraat | 5 | Amsterdam | 1071 EX | 50.00 |
| 2 | 1 | Bob | Black | Coolsingel | 19 | Rotterdam | 3021 FF | 95.00 |
| 3 | 1 | Alice | Blue | Vredenburg | 12 | Utrecht | 3525 AK | 35.00 |
| 4 | 2 | George | Yellow | Spuistraat | 1 | Den Haag | 2511 BC | 10.00 |
| 5 | 2 | Carol | Green | Steenstraat | 23 | Arnhem | 6821 DL | 75.00 |

TABLE 5-5 EXAMPLE DATA FOR THE STUDENT RELATION

| id | first_name | last_name |
|---|---|---|
| 1 | Roger | Orange |
| 2 | Nancy | Purple |

TABLE 5-6 EXAMPLE DATA FOR THE INSTRUCTOR RELATION

This example shows how to create a composite operations conflict graph for two local databases `DB1` and `DB2` that start out with the example data of Table 5-5 and Table 5-6 as their initial database state. At each local database a sequence of composite operations was executed, Table 5-7 shows the composite operations executed at local database `DB1` and Table 5-8 shows the composite operations executed at local database `DB2`.

| ID | Atomic operations |
|---|---|
| $C_{1,1}$ | `Instructor ← Instructor ∪ {<3, Peter, Brown>}`<br><br>`Student ← Student ∪`<br>`{<6, 3, Frank, Red, Grote Markt, 33, Groningen, 9731 CG, 25.00>}`<br><br>`Student ← Student ∪`<br>`{<7, 3, Hank, Gray, Graafseweg, 8, Nijmegen, 6541 LB, 10.00>}` |
| $C_{1,2}$ | `Student ← Student −`<br>`{<4, 2, George, Yellow, Spuistraat, 1, Den Haag, 2511 BC, 10.00>}`<br><br>`Student ← Student −`<br>`{<5, 2, Carol, Green, Steenstraat, 23, Arnhem, 6821 DL, 75.00>}`<br><br>`Instructor ← Instructor − {<2, Nancy, Purple>}` |
| $C_{1,3}$ | `Student ← Student −`<br>`{<7, 3, Hank, Gray, Graafseweg, 8, Nijmegen, 6541 LB, 10.00>}`<br>`∪`<br>`{<7, 3, Hank, Gray, Graafseweg, 8, Nijmegen, 6541 LB, 25.00>}` |

TABLE 5-7 SEQUENCE OF COMPOSITE OPERATIONS EXECUTED AT LOCAL DATABASE DB1

| ID | Atomic operations |
|---|---|
| $C_{2,1}$ | Student ← Student − {<4, 2, George, Yellow, Spuistraat, 1, Den Haag, 2511 BC, 10.00>} ∪ {<4, 2, George, Yellow, Spuistraat, 1, Den Haag, 2511 BC, 50.00>} |
| $C_{2,2}$ | Student ← Student − {<3, 1, Alice, Blue, Vredenburg, 12, Utrecht, 3525 AK, 35.00>} ∪ {<3, 1, Alice, Blue, Vredenburg, 12, Utrecht, 3525 AK, 20.00>} |
| $C_{2,3}$ | Student ← Student − {<4, 2, George, Yellow, Spuistraat, 1, Den Haag, 2511 BC, 50.00>} ∪ {<4, 2, George, Yellow, Spuistraat, 1, Den Haag, 2511 BC, 25.00>} |

TABLE 5-8 SEQUENCE OF COMPOSITE OPERATIONS EXECUTED AT LOCAL DATABASE DB2

From Definition 5.3.1 follows that the process of creating a composite operations conflict graph consists of the following three steps:

1. Add vertices to the graph for all composite operations
2. Add arcs for all conflicting and possible conflicting composite operations
3. Add arcs for all related composite operations

For the first step the sequences of vertices $V_1$ and $V_2$ of Definition 5.3.1 must be determined. This is straightforward, since these are the sequences of composite operations that were executed at the local databases DB1 and DB2, so $V_1 = (C_{1,1}, C_{1,2}, C_{1,3})$ and $V_2 = (C_{2,1}, C_{2,2}, C_{2,3})$. The visual representation of the intermediate composite operations conflict graph after this first step looks like Figure 5.2, with $V_1$ at the left side and $V_2$ at the right side:

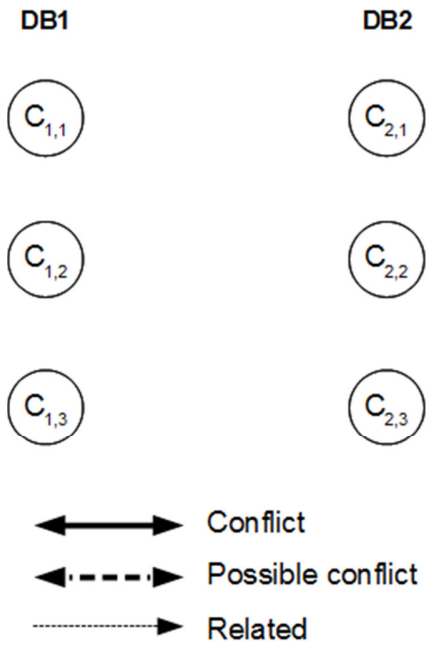


FIGURE 5.2 INTERMEDIATE COMPOSITE OPERATIONS CONFLICT GRAPH AFTER ADDING THE SETS OF VERTICES $V_1$ AND $V_2$ REPRESENTING THE COMPOSITE OPERATIONS

For the second step of creating the composite operations conflict graph the sets of undirected edges $E_C$ and $E_{PC}$ that represent conflicting and possible conflicting composite operations must be determined. Algorithm 5.1.1 can be used to identify conflicting and possible conflicting composite operations. For each combination of composite operations between the two local databases Algorithm 5.1.1 must be applied to see if there is a conflict or possible conflict for that combination of composite operations. The following algorithm can be used to determine $E_C$ and $E_{PC}$ when $V_1$ and $V_2$ are available:

ALGORITHM 5.3.1

$V_1 = (C_{1,1}, C_{1,2}, \ldots, C_{1,m});$
$V_2 = (C_{2,1}, C_{2,2}, \ldots, C_{2,n});$
$E_C = \{\};$
$E_{PC} = \{\};$

for $1 \leq i \leq size(V_1)$:
  for $1 \leq j \leq size(V_2)$:
    if ($conflict(C_{1,i}, C_{2,j})$):
      $E_C = E_C \cup (C_{1,i}, C_{2,j})$;
        else if ($possible\_conflict(C_{1,i}, C_{2,j})$):
      $E_{PC} = E_{PC} \cup (C_{1,i}, C_{2,j})$;

Algorithm 5.3.1 uses the functions $conflict(C_1, C_2)$ and $possible\_conflict(C_1, C_2)$. The function $conflict(C_1, C_2)$ returns true if Algorithm 5.1.1 identifies a conflict between the composite operations $C_1$ and $C_2$, and the function $possible\_conflict(C_1, C_2)$ returns true if Algorithm 5.1.1 identifies a possible conflict between the composite operations $C_1$ and $C_2$.

For the composite operations used in the example of this section only the composite operations $C_{1,2}$ and $C_{2,1}$ conflict, and there are no possible conflicts. The composite operations $C_{1,2}$ and $C_{2,1}$ conflict with each other because the atomic delete operation

```
Student ← Student −
      {<4, 2, George, Yellow, Spuistraat, 1, Den Haag, 2511 BC, 10.00>}
```

of composite operation $C_{1,2}$ conflicts with the atomic update operation

```
Student ← Student −
      {<4, 2, George, Yellow, Spuistraat, 1, Den Haag, 2511 BC, 10.00>}
      ∪
      {<4, 2, George, Yellow, Spuistraat, 1, Den Haag, 2511 BC, 50.00>}
```

of composite operation $C_{2,1}$. According to Theorem 4.4.7 these two atomic operations conflict based on unequal database states after synchronization because the delete operation deletes the tuple that is updated by the update operation.

The result of applying Algorithm 5.3.1 to the composite operations in $V_1$ and $V_2$ is that $E_C = \{(C_{1,2}, C_{2,1})\}$ and $E_{PC} = \{\}$. The visual representation of the intermediate composite operations conflict graph after this second step looks like Figure 5.3:

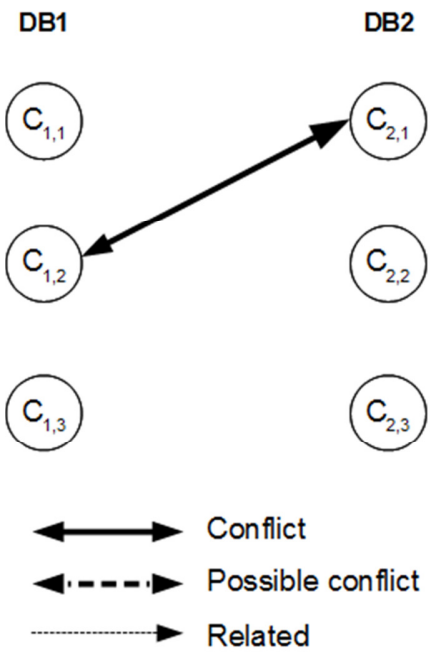


FIGURE 5.3 INTERMEDIATE COMPOSITE OPERATIONS CONFLICT GRAPH AFTER ADDING THE SETS OF UNDIRECTED EDGES $E_C$ AND $E_{PC}$ REPRESENTING THE CONFLICTING AND POSSIBLY CONFLICTING COMPOSITE OPERATIONS

For the third step of creating the composite operations conflict graph the set of directed edges $R$ that represent related composite operations must be determined. Algorithm 5.2.1 can be used to identify related composite operations. For each composite operation Algorithm 5.2.1 must be applied to all combinations of that composite operation and all other composite operations executed earlier at the same local database to determine the related composite operations. The following algorithm can be used to determine $R$ when $V_1$ and $V_2$ are available.

ALGORITHM 5.3.2

$V_1 = (C_{1,1}, C_{1,2}, \ldots, C_{1,m});$
$V_2 = (C_{2,1}, C_{2,2}, \ldots, C_{2,n});$
R = {};

# Skip first composite operation of $V_1$
for $1 < i \leq size(V_1)$:
  for $1 \leq j \leq i - 1$:
    if (related($C_{1,i}$, $C_{1,j}$)):
      $R = R \cup (C_{1,i}, C_{1,j})$;

# Skip first composite operation of $V_2$
for $1 < i \leq size(V_2)$:
  for $1 \leq j \leq i - 1$:
    if (related($C_{2,i}$, $C_{2,j}$)):
      $R = R \cup (C_{2,i}, C_{2,j})$;

Algorithm 5.3.2 uses the function $related(C_1, C_2)$ which returns true if Algorithm 5.2.1determines that the composite operations $C_1$ and $C_2$ are related.

For the composite operations of the example there are two pairs of related composite operations, one at each local database. At local database DB1 the composite operations $C_{1,1}$ and $C_{1,3}$ are related, and at local database DB2 the composite operations $C_{2,1}$ and $C_{2,3}$ are related.

The composite operations $C_{1,1}$ and $C_{1,3}$ are related because the tuple inserted by the atomic insert operation

```
Student ← Student
      ∪ {<7, 3, Hank, Gray, Graafseweg, 8, Nijmegen, 6541 LB, 10.00>}
```

of composite operation $C_{1,1}$ is updated by the atomic update operation

```
Student ← Student
      − {<7, 3, Hank, Gray, Graafseweg, 8, Nijmegen, 6541 LB, 10.00>}
      ∪
      {<7, 3, Hank, Gray, Graafseweg, 8, Nijmegen, 6541 LB, 25.00>}
```

of composite operation $C_{1,3}$, so according to Theorem 5.2.3 the update operation has a dependency on the insert operation. And when an atomic operation of one composite operation has a dependency on an atomic operation belonging to another composite operation executed at the same local database, these two composite operations are related according to Algorithm 5.2.1.

The composite operations $C_{2,1}$ and $C_{2,3}$ are related because the tuple that is the result of the atomic update operation

```
Student ← Student
      − {<4, 2, George, Yellow, Spuistraat, 1, Den Haag, 2511 BC, 10.00>}
      ∪
      {<4, 2, George, Yellow, Spuistraat, 1, Den Haag, 2511 BC, 50.00>}
```

of composite operation $C_{2,1}$ is updated by the atomic update operation

```
Student ← Student
      − {<4, 2, George, Yellow, Spuistraat, 1, Den Haag, 2511 BC, 50.00>}
      ∪
      {<4, 2, George, Yellow, Spuistraat, 1, Den Haag, 2511 BC, 25.00>}
```

of composite operation $C_{2,3}$, so according to Theorem 5.2.4 the update operation of composite operation $C_{2,3}$ has a dependency on the update operation of composite operation $C_{2,1}$. Because there exists a pair of related atomic operations between the composite operations $C_{2,1}$ and $C_{2,3}$, the composite operations $C_{2,1}$ and $C_{2,3}$ are related according to Algorithm 5.2.1.

The result of applying Algorithm 5.3.2 to the composite operations in $V_1$ and $V_2$ is that $R = \{(C_{1,1}, C_{1,3}), (C_{2,1}, C_{2,3})\}$. After this third and final step of creating the composite operations graph the final composite operations conflict graph looks like Figure 5.4:

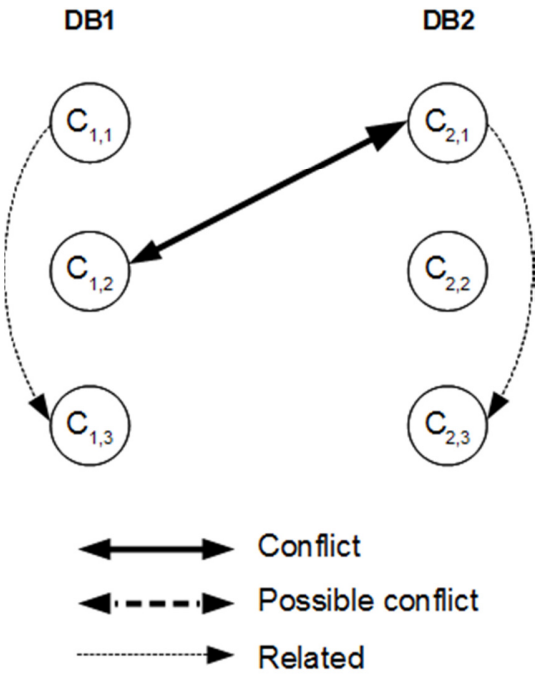


FIGURE 5.4 FINAL COMPOSITE OPERATIONS CONFLICT GRAPH AFTER ADDING THE SET OF DIRECTED EDGES R REPRESENTING THE RELATED COMPOSITE OPERATIONS

This composite operations conflict graph could be used by a synchronization solution to determine which composite operations can be synchronized and which composite operations conflict with each other. In the case of the example in this section the composite operations $C_{1,1}$, $C_{1,3}$ and $C_{2,2}$ can be synchronized, and the composite operations $C_{1,2}$, $C_{2,1}$ and $C_{2,3}$ cannot be synchronized. Composite operations $C_{1,2}$ and $C_{2,1}$ cannot be synchronized because they conflict with each other. Composite operation $C_{2,3}$ cannot be synchronized because it is related to composite operation $C_{2,1}$, which conflicts with composite operation $C_{1,2}$. Note that composite operations $C_{1,1}$ and $C_{1,3}$ are related, but because they are not related to a composite operation that conflicts with another composite operation they can be synchronized.

# 6 CONFLICT DETECTION IMPLEMENTATION

This chapter contains a high level description of the code that implements the conflict detection solution described in chapters 4 and 5. This code is used to validate the conflict detection solution against test data in the next chapter.

Dation uses PHP as the scripting language for their products, so the code described in this chapter is also written in PHP (PHP: Hypertext Preprocessor). During development of the code PHP 5.3.8 was used. But because the code is written using the object oriented programming paradigm (Object-Oriented Programming), it will be straightforward to port the code to another programming language that supports object oriented programming.

The code was developed using the test-driven development process (Test-driven Development), which means that all functionality of the code is covered by unit tests. These unit tests increase the confidence that the developed code is correct. The unit tests can also be used to check if the code still works properly after changes to the code, for example for optimizing the performance of the conflict detection solution.

All code described in this chapter implements the minimal functionality needed for the validation of the conflict detection solution against test data in the next chapter. As such, no guarantees can be made that the code will work properly in other environments.

The conflict detection implementation described in this chapter consists of the three components. The first component is the Database Schema Analyzer. The Database Schema Analyzer creates a model of a database schema, and this model will be used to detect conflicts. The second component is the Operation Logger. The Operation Logger implements the functionality to log the operations that are executed at local databases. The third component is the Conflict Detector. The Conflict Detector uses a database schema model of the Database Schema Analyzer to detect conflicting and related operations for sequences of operations logged by the Operation Logger component. The result of conflict detection by the Conflict Detector component is a composite operations conflict graph.

Section 6.1 describes the implementation of the Database Schema Analyzer component, section 6.2 describes the implementation of the Operation Logger component and section 6.3 describes the implementation of the Conflict Detector component.

## 6.1 DATABASE SCHEMA ANALYZER

The Database Schema Analyzer component creates a model of an existing database schema. The Database Schema Analyzer identifies all relations and integrity constraints of a database schema.

Because Dation uses MySQL databases (MySQL) for their products, the implementation of the Database Schema Analyzer can only be used to analyze a MySQL database schema. However, the model generated by the Database Schema Analyzer component is database agnostic, so it will be straightforward to add support for other database systems to the Database Schema Analyzer component.

The implementation of the Database Schema Analyzer components only identifies the integrity constraints that are guaranteed by the solution proposed in this master thesis. These integrity constraints were stated in section 3.3 and are:

- Key constraints
- Functional dependencies
- Foreign key constraints
- Inclusion dependencies

The database schema constraints that can be defined in a MySQL database schema are (MySQL Table constraints):

- Primary key
- Unique index
- Foreign keys
- Check constraints

The Database Schema Analyzer implementation identifies key constraints and functional dependencies as functional dependencies, because a key constraint is a special type of functional dependency. MySQL uses primary keys and unique indexes to implement key constraints and functional dependencies in a database schema, so the Database Schema Analyzer implementation identifies MySQL primary keys and unique indexes as functional dependencies.

The Database Schema Analyzer implementation also identifies foreign key constraints. However, MySQL only supports foreign key constraints for tables that use the InnoDB storage engine (MySQL Foreign key constraints). For tables with storage engines other than InnoDB, MySQL parses the foreign key syntax in the database schema definition, but does not use or store it.

Inclusion dependencies can be specified using check constraints (Check constraint), however, check constraints are not supported by MySQL. MySQL parses the check constraint syntax in the database schema definition, but ignores it (MySQL Create table syntax). This means that the Database Schema Analyzer implementation does not identify inclusion dependencies, because these cannot be implemented in a MySQL database.

This section concludes with an overview of the integrity constraints supported by MySQL in Table 6-1 and that can be identified by the Database Schema Analyzer component.

| Type of integrity constraint | MySQL equivalent |
|---|---|
| Key constraint | Primary key<br>Unique index |
| Functional dependency | Primary key<br>Unique index |
| Foreign key constraint | Foreign key constraint |
| Inclusion dependency | Not supported |

TABLE 6-1 INTEGRITY CONSTRAINTS SUPPORTED BY MYSQL, WHICH CAN BE IDENTIFIED BY THE DATABASE SCHEMA ANALYZER COMPONENT

## 6.2 OPERATION LOGGER

The Operation Logger component implements the functionality to log the operation executed at local databases. The implementation described in this section represents each query executed at a local database as a composite operation that contains all the atomic operations that were the result of executing a query.

The implementation of the Operation Logger uses triggers to log the atomic operations executed at a local database (Database Triggers). Triggers are procedural code that automatically executes for certain events on a database table. The implementation of the Operation Logger uses triggers that execute after inserting, deleting and updating a tuple to log the atomic operations executed at a local database.

The reasoning behind the decision to use triggers to log the operations that are executed at a local database is the fact that a trigger-based approach lets you log the atomic operations without making modifications to the application code. Furthermore, triggers also let you log operations at the tuple level, which is a requirement for the solution proposed in this master thesis.

The MySQL general query log is used by the implementation of the Operation Logger to log the actual queries executed at a local database (MySQL General query log). MySQL provides the functionality to change the output destination of the general query log to a database table. From this database table the Operation Logger can retrieve the actual queries that were executed against a local database.

Because triggers are supported since MySQL 5.0.2 (MySQL Triggers) and support for the table-based general log query log is available in MySQL 5.1.6 or later (MySQL Log output destinations) the implementation of the Operation Logger requires MySQL 5.1.6 or later.

Before using this implementation of the Operation Logger in a production environment the impact of the trigger-based approach on the performance of the database should be analyzed. There will definitely be a performance impact from this trigger-based approach, because the triggers are executed for every inserted, deleted and updated tuple for each table for which the operations are logged. For high-throughput applications it is possible that this performance impact is not acceptable.

## 6.3 CONFLICT DETECTOR

The Conflict Detector component implements the decision graphs and algorithms presented in chapters 4 and 5. The Conflict Detector creates a composite operations conflict graph for two sequences of operations executed at different local databases. The Operation Logger component of section 6.3 is used to retrieve the sequences of operations executed at the local databases. The Conflict Detector uses the model of the database schema created by the Database Schema Analyzer component of section 6.1 to identify conflicting and related operations.

# 7 VALIDATION

This chapter validates the conflict detection solution proposed in this master thesis by testing the code developed in chapter 6 for some scenarios based on the Dation Dashboard application. These scenarios will only cover some of the theorems presented in chapters 4 and 5, because developing scenarios that cover all the theorems is too much work for a master thesis. Furthermore, the database schema of the Dation Dashboard application does not contain foreign keys or inclusion dependencies, so only conflicts based on unequal database states after synchronization and functional dependency violations can be detected. This means that the validation in this chapter is not complete, but it will give some insight in the feasibility of the approach.

Section 7.1 starts with a description of the approach taken for the validation. Section 0 describes the scenarios and the result of conflict detection for each scenario. Section 7.3 summarizes these results.

## 7.1 APPROACH

For the validation of the conflict detection solution proposed in this master thesis Dation provides a test environment of the Dation Dashboard application. This test environment already contains some data, so a database dump of this test data is used as the initial synchronized database state for the validation scenarios.

Section 7.2 describes the validation scenarios in detail. Each of these scenarios describes two actions that are performed in the Dation Dashboard application. The database operations that are the result of performing these actions against the Dation Dashboard application are captured with the Operation Logger component of section 6.2. To simulate execution at two different local databases the initial synchronized database state is reset between each action that is executed in the Dation Dashboard application by restoring the database dump mentioned in the previous paragraph[2]. Conflicts can be detected between the two sequences of operations captured with the Operation Logger with the Conflict Detector component described in section 6.3.

Because the Dation Dashboard application uses a MySQL 5.0.32 database the Operation Logger developed in chapter 6 cannot be used directly with the Dation Dashboard database, because the Operation Logger needs MySQL 5.1.6 or later. Because of this problem the operations that are the result of performing the actions in the Dation Dashboard application are replayed at an exact copy of the database at a local development machine by using the Toad for MySQL freeware program (Toad for MySQL). The database of this local development machine supports the Operation Logger component.

## 7.2 SCENARIOS

This section describes the scenarios used to validate the conflict detection solution proposed in this master thesis. Each scenario consists of two actions performed against Dation Dashboard

[2] The scenario in section 7.2.1 actually identified a problem which could be overcome by slightly adapting the approach described in section 7.1. This adapted approach was used for all other scenarios.

applications with different local databases. All scenarios are based on the agenda functionality of the Dation Dashboard application.

The Dation Dashboard agenda can be used to plan lessons for driving instructors. Each instructor has his own agenda which consists of blocks of time for which the instructor is available. Driving lessons can be planned by making a new appointment for a block. The scenarios in this section are based on adding new appointments for lessons and deleting or updating planned lessons.

Each scenario is described in its own subsection. For each scenario the expected result and the actual result of conflict detection are stated. If the expected result and the actual result of conflict detection do not match a discussion of why this is the case follows.

### 7.2.1 TWO APPOINTMENTS FOR DIFFERENT INSTRUCTORS

This scenario creates two appointments for different instructors. The expected result of conflict detection is that there are no conflicts and possible conflicts. However, after running the scenario six conflicts and no possible conflict were detected.

All six conflicts were conflicts based on Theorem 4.5.1 which identifies conflicts based on a functional dependency violation for two atomic insert operations that affect the same relation. All conflicts occurred because Dation uses an auto-incrementing integer field as a primary key for most of the relations in their database schema (MySQL auto-increment). When inserting a new tuple into a relation with an auto-incrementing primary key, MySQL will automatically assign sequential integer values as the primary key for these new tuples. Because of the assignment of sequential primary keys at local databases, duplicate primary keys are created for new tuples at the different local databases, and the conflict detection solution will identify conflicts for different tuples with the same value for the primary key.

From this follows that the conflict detection solution proposed in this master thesis will detect a lot of unnecessary conflicts for database schemas that use auto-incrementing primary keys for the relations. A simple solution for this problem would be to use different sets of primary keys for the different local databases.

Because almost all relations in the database of the Dation Dashboard application use auto-incrementing primary keys there will be a lot of unnecessary conflicts due to duplicate primary keys. Because of this the approach for the validation scenarios described in section 7.1 is slightly adapted for all other validation scenarios. After performing the first action of the scenario the database dump is not restored, but the freeware application Toad for MySQL is used to undo all operations that were executed as a result of the performed action (Toad for MySQL). This way MySQL remembers the last used auto-increment value of the first action and uses different values for the second action. This approach simulates different sets of auto-incrementing primary keys for the different local databases, which prevents all the unnecessary conflicts for duplicate primary keys. This adapted approach is used for all other scenarios in this chapter.

After running this scenario again with the approach described in the previous paragraph there were no conflicts and possible conflicts detected which matches the expected result for this scenario.

### 7.2.2 TWO DIFFERENT APPOINTMENTS FOR THE SAME INSTRUCTOR FOR DIFFERENT BLOCKS

This scenario creates two appointments for the same instructor for two different blocks. The details of this appointment are different, for example the student and the reserved vehicle for the lesson. The expected result of conflict detection is that there are no conflicts and possible conflicts. After running this scenario there were no conflicts and possible conflicts detected, which means the conflict detection solution behaves as expected for this scenario.

### 7.2.3 TWO EQUAL APPOINTMENTS FOR THE SAME INSTRUCTOR FOR DIFFERENT BLOCKS

This scenario creates two appointments for the same instructor for two different blocks. The details of this appointment are equal, for example the same student and reserved vehicle for the lesson. The expected result of conflict detection is that there are no conflicts and possible conflicts. However, after running this scenario three conflicts were detected.

All three conflicts were conflicts based on Theorem 4.4.3 which identifies conflicts based on unequal database states after synchronization for two atomic update operations that update at least one attribute of the same tuple to different values.

These conflicts occur because the Dation Dashboard application does not only add a row to the table in the database which contains all the appointments, but also updates rows in other tables after adding a new appointment. As an example, the Dation Dashboard application automatically updates the balance of a Student after planning a new lesson as well as billing details.

This means there are no conflicts detected for the operations that add the new appointments, but there are conflicts for the additional operations executed by the Dation Dashboard application after adding a new appointment. The detection of these conflicts by the conflict detection solution is correct, but not what was expected before executing this scenario, because the additional operations executed by the Dation Dashboard application were not taken into account.

### 7.2.4 TWO DIFFERENT APPOINTMENTS FOR THE SAME INSTRUCTOR FOR SAME BLOCK

This scenario creates two appointments for the same instructor for the same block, where the details of the appointments differ. This scenario is similar to scenario 7.2.2, only this time the appointments are made for the same block. The expected result of conflict detection is that there is a conflict because it should not be possible to create two appointments at the same time. However, after running this scenario no conflicts and possible conflicts were detected.

After some investigation it became clear that this occurred because the Dation Dashboard application enforces the constraint that there cannot be two appointments at the same time at the application level and not with a database constraint. This means that the conflict detection solution cannot detect conflicts for two appointments at the same time, because the conflict detection solution can only detect conflicts for constraints defined in the database schema.

### 7.2.5 DELETE DIFFERENT APPOINTMENTS

This scenario deletes two different appointments. The expected result of conflict detection is that there are no conflicts and possible conflicts. After running this scenario there were no conflicts and

possible conflicts detected, which means the conflict detection solution behaves as expected for this scenario.

### 7.2.6 DELETE THE SAME APPOINTMENT

This scenario deletes the same appointment twice. The expected result of conflict detection is that there are no conflicts and possible conflicts. After running this scenario there were no conflicts and possible conflicts detected, which means the conflict detection solution behaves as expected for this scenario.

### 7.2.7 UPDATE AND DELETE FOR DIFFERENT APPOINTMENTS

This scenario edits an appointment by changing the date and time of the appointment and deletes another appointment. The expected result of conflict detection is that there are no conflicts and possible conflicts. After running this scenario there were no conflicts and possible conflicts detected, which means the conflict detection solution behaves as expected for this scenario.

### 7.2.8 UPDATE AND DELETE FOR THE SAME APPOINTMENT

This scenario edits an appointment by changing the date and time of the appointment while the other action deletes this appointment. The expected result of conflict detection is that there are conflicts. After running this scenario three conflicts were detected. All three conflicts were conflicts based on Theorem 4.4.7 which identifies conflicts based on unequal database states after synchronization in the cases where an atomic update operation updates a tuple that is deleted by an atomic delete operation. This means that the conflict detection solution behaves as expected for this solution.

### 7.2.9 INSERT A NEW APPOINTMENT AND DELETE AN EXISTING APPOINTMENT

This scenario inserts a new appointment while the other action deletes an existing appointment. The expected result of conflict detection is that there are no conflicts and possible conflicts. After running this scenario there were no conflicts and possible conflicts detected, which means the conflict detection solution behaves as expected in this scenario.

### 7.2.10 UPDATE TWO DIFFERENT APPOINTMENTS

This scenario updates two different appointments. The expected result of conflict detection is that there are no conflicts and possible conflicts. After running this scenario there were no conflicts and possible conflicts detected, which means the conflict detection solution behaves as expected in this scenario.

### 7.2.11 UPDATE THE SAME APPOINTMENT WITH DIFFERENT VALUES

This scenario updates the date and time of the same appointment to different values. The expected result of conflict detection is that there are conflicts. After running this scenario three conflicts were detected. All three conflicts were conflicts based on Theorem 4.4.3 which identifies conflicts based on unequal database states after synchronization for two atomic update operations that update at least one attribute of the same tuple to different values. This means that the conflict detection solution behaves as expected for this solution.

### 7.2.12 UPDATE THE SAME APPOINTMENT WITH THE SAME VALUES

In this scenario both actions update the date and time of the same appointment to the same values. The expected result of conflict detection is that there are no conflicts and possible conflicts. However, after running the scenario one conflict and two possible conflicts were detected.

The conflict was a conflict based on Theorem 4.4.3 which identifies conflicts based on unequal database states after synchronization for two atomic update operations that update at least one attribute of the same tuple to different values. In this case there were two updates to the same tuple that set a timestamp that records the time the tuple was last updated. Because the values of the timestamps were different a conflict was detected.

The two possible conflicts were possible conflicts based on Theorem 4.5.4 which identifies possible conflicts for two atomic update operations that update the same tuple where at least one of the update operations updates an attribute that is part of the referencing or referenced attribute set of a functional dependency. However, this theorem will always detect conflicts if there are two updates to the same tuple, because each relation has a primary key and all attributes of the relation are in either the determining or dependent attribute set of the primary key.

### 7.2.13 INSERT A NEW APPOINTMENT AND UPDATE AN EXISTING APPOINTMENT

This scenario inserts a new appointment and updates an existing appointment. The expected result of conflict detection is that there are no conflicts and possible conflicts. After running this scenario there were no conflicts and possible conflicts detected, which means the conflict detection solution behaves as expected in this scenario.

## 7.3 SUMMARY

This section contains a summary of the results of executing the scenarios in the previous section. It summarizes the problems that occurred with the conflict detection solution in its current form. These problems occurred for the scenarios where the expected result of conflict detection did not match with the actual result of conflict detection after running the scenario.

- Scenario 7.2.1 identified a problem with the usage of auto-incrementing primary keys for the relations in the database schema. This will result in duplicate primary keys for tuples inserted into the same relation at different local databases. A simple solution would be to let the local databases use different sets of primary keys.

- Scenario 7.2.3 identified a problem where a single action in the Dation Dashboard application leads to multiple queries, where not only the relation of the manipulated item in the Dation Dashboard application is affected, but also other relations with data that is related to the manipulated item. No conflicts were expected, but there were conflicts for the operations that were executed against the related relations. These types of conflicts were not considered in this master thesis, but should be addressed in follow-up research.

- Scenario 7.2.4 identified a problem where a constraint that is enforced at the application level cannot be detected by the conflict detection solution, because it can only detect conflicts for constraints defined in the database schema.

- Scenario 7.2.12 identified a problem for two updates to the same tuple where timestamps were set to record when the affected tuple was last updated. From this can be deduced that for relations where last updated timestamps are recorded there will be a conflict if there are updates to the same tuples of such a relation, even if all other values are equal.

- Scenario 7.2.12 identified a problem with Theorem 4.5.4 which identifies possible conflicts for two atomic update operations that update the same tuple where at least one of the update operations updates an attribute that is part of the determining or dependent attribute set of a functional dependency. Because each relation has a primary key and all attributes of the relation are in either the determining or dependent attribute set of the primary key, this theorem will always detect a possible conflict if there are two updates to the same tuple.

# 8 CONCLUSIONS AND FUTURE WORK

This chapter will reflect on the work done in this master thesis in section 8.1 and present directions for future research in section 8.2.

## 8.1 CONCLUSIONS

This section starts with answering the research questions from section 0, which is followed by an answer to the main research question based on the results of validation in chapter 7.

1. *Which properties of horizontally decentralized relational databases can be violated during synchronization?*

To address this question a literature study into the properties of relational database was done. The findings of this literature study are in chapter 3.

A synchronization solution should not only make sure the database states are equal after synchronization, but should also ensure that no integrity constraints of the database are violated in the synchronized database state. Section 3.1.1 investigated the types of integrity constraints that can be defined for a relational database.

Two types of integrity constraints were identified, static integrity constraints and dynamic integrity constraints. Static integrity constraints restrict the legal instances of a database, while dynamic integrity constraints restrict the evolution of legal instances of a database. Static integrity constraints are expressed in the database schema, while dynamic integrity constraints are used mainly to express business rules and are implemented at the application level. The work in this master thesis was limited to static integrity constraints. Section 3.3 stated the integrity constraints that are guaranteed by the conflict detection solution proposed in this master thesis, and these are key constraints, functional dependencies, foreign key constraints and inclusion dependencies.

2. *How can atomic operations violate these properties during synchronization?*

This research question is addressed in chapter 4. A conflict between two atomic operations is defined in Definition 4.1.4. This definition consists of two parts. The first part of this definition states that the synchronized database states should be equal. Conflicts based on unequal database states after synchronization were investigated in section 4.4. This section presents several theorems that can be used to identify conflicts based on unequal database states for all combinations of atomic operations.

The second part of the atomic operations conflict definition states that there is a conflict if the synchronized database state violates any of the integrity constraints of the database. Section 4.5 investigates conflicts based on the violation of integrity constraints. The integrity constraints covered in this section are the integrity constraints listed in section 3.3: key constraints, functional dependencies, foreign key constraints and inclusion dependencies. Conflicts between two atomic operations based on violations of integrity constraints can be detected by using the theorems presented in section 4.5.

Section 4.6 concludes chapter 4 by presenting three decision graphs that can be used to identify conflicts between two arbitrary atomic operations. The three decision graphs detect conflicts for unequal database states after synchronization, conflict based on the violation of intra-relational integrity constraints and conflicts based on the violation of inter-relational integrity constraints. However, all three decision graphs must be checked to see if there is a conflict between two arbitrary atomic operations, because the three decision graphs cover different types of conflicts.

3. *How can composite operations violate these properties during synchronization?*

This research question is addressed in chapter 5. In section 5.1 a composite operation is defined as a sequence of atomic operations with the atomicity property. There is a conflict between two arbitrary composites operations if there exists a pair of atomic operations between the two composite operations for which there is a conflict. To determine conflicts between pairs of atomic operations the decision graphs of section 4.6 can be used.

Section 5.3 introduces the composite operations conflict graph. This graph can be used to display the conflicting composite operations between two sequences of composite operations. The composite operations conflict graph also identifies related composite operations.

Related composite operations were discussed in section 5.2, which discussed dependencies between atomic and composite operations. This section states that there can be dependencies between atomic and composite operations based on the data items they affect. If there is a dependency between two operations these two operations are related. If an operation cannot be synchronized this could also have an impact on related operations. Section 5.2 only presents theorems to identify related atomic and composite operations, but this should be used in future research into the actual synchronization of two local databases after the conflicts are detected.

4. *How can synchronization conflicts be detected in an automated way?*

The theorems to detect conflicts presented in chapters 4 and 5 can be used to identify synchronization conflicts. Chapter 6 describes the implementation of the conflict detection solution that was developed based on the theorems presented in chapters 4 and 5. This implementation consists of three components.

The first component is the Database Schema Analyzer which analyzes the constraints that are defined in a database schema. The second component is the Operation Logger which can be used to log the operations that are executed at a local database. The third component is the Conflict Detector which detects conflicts between two sequences of operations executed at two local databases logged with the Operation Logger. The Conflict Detector component implements the theorems presented in chapters 4 and 5. The result of conflict detection by the Conflict Detector component is a composite operations conflict graph.

In chapter 7 the implementation of the conflict detection solution was validated against a set of scenarios executed against a development environment of the Dation Dashboard application. The results of this validation are the basis for the answer to the main research question:

*How can synchronization conflicts be detected for the synchronization of two local databases of a horizontally decentralized relational database?*

The conflict detection solution proposed in this master thesis takes the approach of comparing the operations executed at local database to detect conflicts. The theorems presented in chapters 4 and 5 can be used to detect conflicts, but the validation done in chapter 7 identified several problems with the conflict detection solution in its current form. These problems should be addressed in future research to develop a conflict detection solution that will be usable in production environments.

One of the problems encountered during validation is that there are some database schema practices for which the conflict detection solution will detect a lot of unnecessary conflicts. Scenario 7.2.1 identified a problem where conflicts are detected for duplicate primary keys for tuples inserted into a relation that uses auto-incrementing primary keys. Scenario 7.2.12 identified a problem where the usage of last updated timestamps for tuples leads to conflicts for two updates to the same tuple. The conflict detection solution proposed in this master thesis will perform better for database schemas that do not use these practices.

Another problem found during validation is the fact that a single action at the application level can lead to multiple queries against a local database as was the case in scenario 7.2.3. While no conflicts were expected for the relation that was manipulated with the application level action there were conflicts for operations that affect related relations. Operations that affect related relations were not considered in this master thesis, but this is something that should be covered in future research.

Scenario 7.2.4 identified a problem where conflicts based on constraints that are not defined in the database schema cannot be detected. This was a deliberate choice for the work in this master thesis, but it will not always be possible to define all business rules as constraints in the database schema, so future research should investigate possible approaches to detect conflicts for constraints that are enforced at the application level.

Overall the conflict detection solution proposed in this master thesis seems promising, but there are several problems that should be addressed before it can be used in a production environment.

## 8.2 FUTURE WORK

The work in this master thesis is only the initial research for developing a synchronization solution that synchronizes two local databases by investigating the operations that were executed at both local databases. There is a lot to be done before there will be a synchronization solution suitable for a production environment. The work that still must be done can be divided in two parts. The first part consists of improvement and analysis of the conflict detection solution described in this master thesis, which is discussed in section 8.2.1. The second part consists of actually synchronizing two local databases after the conflicts are detected with the solution described in this master thesis, which is discussed in section 8.2.1

### 8.2.1 CONFLICT DETECTION IMPROVEMENTS

This master thesis focusses on the correctness of the presented conflict detection solution. However, a correct solution with poor performance will not be usable in in a production environment. So analyzing the performance of the proposed solution under different circumstances is one direction for future research.

The implementation of the conflict detection solution described in chapter 6 and used for the validation in chapter 7 is a direct translation of the theorems, algorithms and decision graphs presented in chapters 4 and 5 to code. If these theorems, algorithms and decision graphs can be optimized by further research, these optimizations can also be used to optimize the performance of the implementation of the conflict detection solution.

As an example of an optimization consider the three decision graphs of section 4.6 which are used to detect conflicts for two arbitrary atomic operations. A possible optimization could be to try to merge these three decision graphs into a single decision graph that could be used to detect conflicts between two arbitrary atomic operations.

The current implementation of the Operation Logger component described in section 6.2 uses database triggers (Database Triggers). But there can be situations where this trigger-based approach is not usable, for example when there already exists triggers in the database which are not compatible with the triggers for logging the operations. Developing another solution to log the operations that are executed at a local database could be a direction for future research. Another direction for future research could be to analyze the performance impact on databases of the trigger-based solution.

The conflict detection solution presented in this master thesis can only detect conflicts based on violations of integrity constraints defined in the database schema. However, it is possible that applications use constraints that are defined at the application level and not in the database schema. Adding support for these constraints to the conflict detection solution presented in this master thesis could be another direction for future research.

### 8.2.2 SYNCHRONIZATION

The work in this master thesis describes how to detect conflicting operations between two local databases. However, detecting conflicts is only the first step for a synchronization solution. The second step is to handle the detected conflicts in an automated way so a synchronized database state can be reached. How to actually do this should be the follow-up research to this master thesis. The remainder of this section lists some aspects that should be a part of this research.

To reach a synchronous database state at two local databases the detected conflicts should be resolved. Is it possible to develop a fully automated conflict-resolution solution, or is some form of user input necessary? A fully automated conflict-resolution solution could make use of a set of rules to resolve conflicts, while a conflict-resolution solution that uses user input can let the user decide how to resolve conflicts. A hybrid approach is also possible, where a set of rules is defined to resolve a set of conflicts, while user input is needed to resolve the other conflicts.

A simple approach for a fully automated conflict-resolution solution could be to declare one local database as a master and the other local database as a slave, and always resolve the conflicts so that the operations of the master local database are preserved. But such an approach will not be suitable for all situations, so other approaches should also be investigated.

After deciding how to resolve the detected conflicts these decisions should be applied to the local databases so they end up with the same synchronous database state. How to implement this synchronization of the local databases is the last part of the research for a fully functional

synchronization solution. The remainder of this section describes a possible approach for synchronization that can be researched, but other possible approaches should also be investigated.

For each detected conflict between two operations the conflict-resolution solution should specify the operation that should be retained and which operation should be discarded. All operations that should be retained can be synchronized in the same way as the non-conflicting operations. A possible approach for synchronizing two local databases would be to apply all the non-conflicting and resolved operations of one local database at the other local database, while undoing all operations that were discarded by the conflict-resolution solution.

# 9 BIBLIOGRAPHY


*Check constraint*. (n.d.). Retrieved 06 05, 2012, from http://en.wikipedia.org/wiki/Check_constraint

Codd, E. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 377-387.

*Database Triggers*. (n.d.). Retrieved 06 06, 2012, from Wikipedia: http://en.wikipedia.org/wiki/Database_trigger

Kifer, M. B. (2006). *Database systems: an application-oriented approach.* Boston: Pearson / Addison Wesley.

Miha. (2008, November 23). *Decentralized database*. Retrieved May 5, 2011, from Webeks.net: http://www.webeks.net/software-architecture/decentralized-database.html

*MySQL*. (n.d.). Retrieved 06 05, 2012, from http://www.mysql.com/

*MySQL auto-increment*. (n.d.). Retrieved 07 02, 2012, from http://dev.mysql.com/doc/refman/5.5/en/example-auto-increment.html

*MySQL Create table syntax*. (n.d.). Retrieved 06 05, 2012, from http://dev.mysql.com/doc/refman/5.1/en/create-table.html

*MySQL Foreign key constraints*. (n.d.). Retrieved 06 05, 2012, from http://dev.mysql.com/doc/refman/5.1/en/ansi-diff-foreign-keys.html

*MySQL General query log*. (n.d.). Retrieved 06 06, 2012, from http://dev.mysql.com/doc/refman/5.1/en/query-log.html

*MySQL Log output destinations*. (n.d.). Retrieved 06 06, 2012, from http://dev.mysql.com/doc/refman/5.1/en/log-destinations.html

*MySQL Table constraints*. (n.d.). Retrieved 06 05, 2012, from http://dev.mysql.com/doc/refman/5.1/en/table-constraints-table.html

*MySQL Triggers*. (n.d.). Retrieved 06 06, 2012, from http://dev.mysql.com/doc/refman/5.0/en/triggers.html

*Object-Oriented Programming*. (n.d.). Retrieved 06 05, 2012, from http://en.wikipedia.org/wiki/Object-oriented_programming

Özsu, M. a. (1999). *Principles of distributed database systems.* Upper Saddle River, NJ: Prentice Hall International.

Padigela, M. (n.d.). *Decentralized database*. Retrieved May 5, 2011, from Mahipalreddy.com: http://www.mahipalreddy.com/dbdesign/dbarticle1.htm

*PHP: Hypertext Preprocessor*. (n.d.). Retrieved 06 05, 2012, from http://www.php.net/

*Relational Algebra*. (n.d.). Retrieved May 5, 2011, from Wikipedia: http://en.wikipedia.org/wiki/Relational_algebra

*Set Difference is Right Distributive over Union*. (n.d.). Retrieved May 5, 2011, from Proofwiki: http://www.proofwiki.org/wiki/Set_Difference_is_Right_Distributive_over_Union

*Set Difference with Union*. (n.d.). Retrieved May 5, 2011, from Proofwiki: http://www.proofwiki.org/wiki/Set_Difference_with_Union

*Test-driven Development*. (n.d.). Retrieved 06 05, 2012, from http://en.wikipedia.org/wiki/Test-driven_development

*Toad for MySQL*. (n.d.). Retrieved 07 02, 2012, from http://www.quest.com/toad-for-mysql/

*Types of graphs*. (n.d.). Retrieved May 5, 2011, from Wikipedia: http://en.wikipedia.org/wiki/Graph_(mathematics)#Types_of_graphs

*Union is Commutative*. (n.d.). Retrieved May 5, 2011, from Proofwiki: http://www.proofwiki.org/wiki/Union_is_Commutative

# APPENDIX A CONSTRAINT LANGUAGE

This appendix contains an overview of the constraint language used in this master thesis. The constraint language is a set of literals, expressions and functions used throughout this master thesis. This appendix is split into two sections. Section A.1 lists all the used literals and expressions and section A.2 lists all the used functions.

## A.1 LITERALS AND EXPRESSIONS

**`S`**
: a database state

**`R`**
: a relation

**`op`**
: an atomic operation

**`op(S)`**
: apply an atomic operation `op` to a database state `S`

**`ins`**
: an atomic insert operation

**`del`**
: an atomic delete operation

**`upd`**
: an atomic update operation

**`C`**
: a composite operation

**`s, t`**
: a tuple

**`A, B`**
: a single attribute

**`X, Y`**
: multiple attributes

**`t[X]`**
: the projection of attributes `X` on tuple `t`

**`FD`**
: a functional dependency

**RIC**
: a referential integrity constraint

**FK**
: a foreign key constraint

**IC**
: an inclusion dependency

## A.2 FUNCTIONS

**type(op)**
: returns the type of an operation `op`, which can be `ins`, `del` or `upd`

**rel(op)**
: returns the relation of an operation `op`

**rel(t)**
: returns the relation of a tuple `t`

**rel(FD)**
: returns the relation of a functional dependency `FD`

**tuple(op)**
: returns the affected tuple of an operation op, for a delete operation this is the deleted tuple, for an insert operation this is the inserted tuple and for an update operation this is the existing tuple that was updated

**result(upd)**
: returns the tuple with updated values for an update operation upd

**attrib(upd)**
: returns the set of attributes that are affected by an update operation upd

**updType(upd, A) )**
: returns the type of update that is performed against an attribute `A` by update operation `upd`, which can be ABSOLUTE or RELATIVE

**det(FD)**
: returns the determinant attribute set of a functional dependency `FD`

**dep(FD)**
: returns the depending attribute set of a functional dependency `FD`

**referencingRel(RIC)**
: returns the referencing relation of a referential integrity constraint `RIC`

**referencedRel(RIC)**
: returns the referenced relation of a referential integrity constraint `RIC`

**`referencingAttrib(RIC)`**
: returns the referencing attribute set of a referential integrity constraint `RIC`

**`referencedAttrib(RIC)`**
: returns the referenced attribute set of a referential integrity constraint `RIC`

**`conflict(op, op)`**
: returns true if there is a conflict for two atomic operations or composite operations passed as arguments to the function

**`possible_conflict(op, op)`**
: returns true if there is a possible conflict for two atomic operations or composite operations passed as arguments to the function

**`related(op, op)`**
: returns true if the two atomic operations or composite operations that are passed as arguments to the function are related

# APPENDIX B DEFINITIONS AND THEOREMS

This appendix contains all definitions, theorems and algorithms presented in this master thesis.

## B.1 DEFINITIONS

## B.2 THEOREMS



## B.3 ALGORITHMS