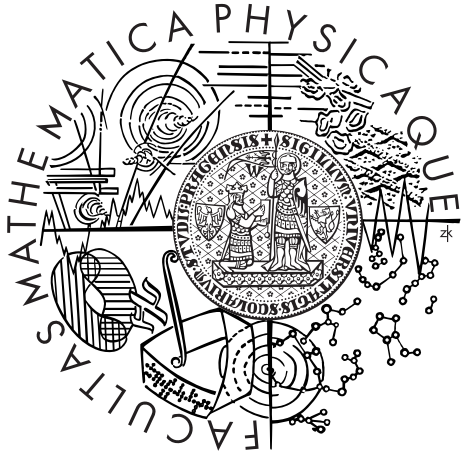
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Martin Chytil

Adaptation of Relational Database Schema

Department of Software Engineering

Supervisor of the master thesis: RNDr. Irena Mlýnková, Ph.D.
Study programme: Informatics
Specialization: Software systems

Prague 2011

I would like to thank to my supervisor RNDr. Irena Mlýnková, Ph.D. for her helpful suggestions, thorough notes, provided related research material and text corrections.
I would also like to thank to Mgr. Martin Nečaský, Ph.D. for his suggestions and comments.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

I hereby declare that I have elaborated this master thesis on my own and listed all used references. I agree with making this thesis publicly available.

In Prague on November 20, 2011 Martin Chytil

Název práce: *Adaptace Schématu Relační Databáze*
Autor: *Martin Chytil*
Katedra: *Katedra Softwarového Inženýrství*
Vedoucí diplomové práce: *RNDr. Irena Mlýnková, Ph.D.*
E-mail vedoucího: *irena.mlynkova@ksi.mff.cuni.cz*

Abstrakt: *V předložené práci studujeme evoluci databázového schématu a její vliv na související oblasti. Práce obsahuje přehled důležitých problémů spojených se změnou úložiště dat a také popisuje existující přístupy k těmto problémům. Detailněji práce rozebírá vliv změn databázového schématu na databázové dotazy, které jsou na příslušném schématu závislé. Přístup představený v této práci ukazuje možnost modelování databázových dotazů společně s modelem databázového schématu. Práce dále popisuje řešení jak upravovat databázové dotazy v závislosti na evoluci příslušného databázového schématu. V neposlední řadě práce obsahuje sadu experimentů ověřujících návrh zvoleného řešení.*

Klíčová slova: *Relační databáze, SQL, mapování, evoluce*

Title: *Adaptation of Relational Database Schema*
Author: *Martin Chytil*
Department: *Department of Software Engineering*
Supervisor: *RNDr. Irena Mlýnková, Ph.D.*
Supervisor's e-mail address: *irena.mlynkova@ksi.mff.cuni.cz*

Abstract: *In the presented work we study evolution of a database schema and its impact on related issues. The work contains a review of important problems related to the change in a respective storage of the data. It describes existing approaches of these problems as well. In detail the work analyzes an impact of database schema changes on database queries, which relate to the particular database schema. The approach presented in this thesis shows a ability to model database queries together with a database schema model. The thesis describes a solution how to adapt database queries related to the evolved database schema. Finally the work contains a number of experiments that verify a proposal of the presented solution.*

Keywords: *Relational database, SQL, mapping, evolution*

Contents

Chapter 1

Introduction

1.1 Motivation

Since most of the current applications are dynamic, sooner or later the structure of the data needs to be changed and so have to be changed also all related issues. We speak about evolution and adaptability of applications. One of the aspects of this problem is adaptation of the respective storage of the data.

The adaptation of the storage covers many related issues:

- Database Schema Evolution
  This issue involves retaining current data and software system functionality despite the database schema changes.

- Database Schema Integration
  This issue involves combining of data sources concepts and knowledge into one schema. It is very common in situations when two database schemas have to be combined together to create a single schema.

- Data Migration
  In relational databases is common situation that data have to be moved from one database management system to another, or the version of database software being used has to be upgraded.

- Adaptation of Respective Queries
  If the database schema has been changed, all database queries have to be reformulated according to the changes. This involves changes of views, stored procuders and functions, or queries used for instance for analytical purposes.

In this thesis we focus on adaptation of SQL queries. The change of the underlaying database schema can cause that SQL queries over this schema may become inconsistent with the new schema, e.g.:

- A database table has a new name in the new schema.

- A table column has a new name in the new schema.

- A database table does not exist in the new schema.

- A table column does not exist in the new schema.
- A new table column, which should be used in SQL query, appeared in the new schema.

All these presented possibilities lead to incorrect SQL queries and so for this reason they should be corrected. Suppose an SQL view *PendingOrders*, which returns all information about the pending orders, including all items of the given order. Now, when the name of the column *itemName* is changed (for instance to *productName*), all SQL queries where this column is used have to be checked by designer and updated respectively.

1.2 Aim of the Thesis

The aim of this thesis is a research on possibilities and limitations of adaptation of a relational database schema. The thesis analyzes the related issues in general and discusses their key problems and solutions. The core of the work is a proposal and implementation of own approach dealing with selected open issue related to database schema evolution. In this thesis we selected *adaptation of SQL queries*. The purpose of the thesis is to:

- design a model of SQL queries usable in *Computer Aided Software Engineering* (CASE) tools
- analyze changes in the database model and their possible impact on related SQL queries
- propose algorithms to propagate changes done in the database model on the query to satisfy valid results

1.3 Organization of the Thesis

The rest of this thesis is organized as follows. Chapter 2 briefly describes the relational data model [19] and selected parts of SQL language [28] related to the aim of this thesis. Chapter 3 introduces a base concept of a model driven architecture [31], which is important to understand roles of the proposed models in this thesis.

Chapter 4 describes related issues of relational database schema adaptation and discusses key problems and solutions. In Chapter 5 are discussed related works which are dealing with a selected database schema adaptation problem.

In Chapter 6 a database model, which is used in this thesis as a model for representing a database schema, is described.

In Chapter 7 we propose a graph-based SQL query model, which is particularly designed for evolution process. The platform-specific visualisation model of the query is introduced as well. A part of this chapter is focused on mapping between SQL query model and the database model. At the end of this chapter we describe an algorithm to generate SQL query from the query model.

Chapter 8 discusses an impact of changes in the database model on the queries in the SQL query model. The biggest part of this chapter is paid to description of algorithms which distribute changes in the query graph.

Chapter 9 presents a prototype implementation of the proposed solution. Results of experiments on the real-world database schema are described as well. Chapter 10 summarizes this thesis and proposes future research directions of this approach.

Chapter 2

Relational Data Model and SQL

This chapter describes the most popular data model of databases - the relational data model [19]. Then we describe selected parts of the SQL language which are related to the aim of this thesis.

2.1 Relational Data Model

The characteristic features of the relational data model are:

- Conceptually simple - the fundamentals are intuitive and very simple.
- Powerful underlaying theory - the relational model is the database model, which is powered by formal mathematics.
- Easy-to-use database language - the Structured Query Language (SQL) [28] is responsible for the success of relational model, though formaly it is not its part; SQL became a de facto standard language for working with relational databases.

2.1.1 Structure

There exists only one data structure in the relational data model - the *relation*. The idea of a relation is based on a mathematical construct of *relation*. Relations in the relational data model obey a certain restricted set of rules:

- Each relation must have an unique name.
- Each attribute in a relation must have an unique name within the relation.
- Each attribute in a relation defines a set of permitted values of given attribute. This set of permitted values is called the *domain* of given attribute.
- The ordering of attributes in a relation is not significant.
- Each tuple of relation must be unique (from the mathematical definition of relation). In other words, duplicate tuples are not allowed.
- The ordering of tuples of relation is not significant.

- Formally, for a relation r with n attributes $a_1, \ldots, a_n$, each attribute a_k, where $k \in 1 \ldots n$, has a domain D_k, and any given tuple of r is a n-tuple $(v_1, \ldots, v_n)$ such that $v_k \in D_k$. Therefore, any instance of relation r is a subset of the Cartesian product $D_1 \times \cdots \times D_n$.

- There is an assumption, that each domain D_k must have only *atomic values*. Composite structures are not allowed to be used.

- Each domain has one extra allowed member - *null value*, which means an unknown or non-existent value. In practice, this value causes a number of practical issues, so it is intention to avoid the using of the *null value*.

The relational database is a collection of *tables*, where each table corresponds to the term of *relation* in the sense of relational data model.

2.1.2 Keys

The *keys* are values or sets of values used to distinguish one tuple from another one in the same relation.

We distinguish these kind of keys:

- *Superkey*
  The superkey is a non-empty set of attributes, which uniquely identify a tuple in a relation. It can range from one attribute to the entire tuple.
  Formally, if R is a schema of relation, then a subset K of R is a superkey for R if, for a relation r with schema R ($r(R)$) and tuples t_1 and t_2 in r, $t_1 \neq t_2 \rightarrow t_1[K] \neq t_2[K]$.
  In practice, not every superkey can be useful.

- *Candidate key*
  The candidate key corresponds to the superkey, for which no proper subset is a superkey too. Simply it is a minimal superkey.
  Still there can exist more than one candidate key for a relation, consisting of different subset of attributes.

- *Primary key*
  The primary key is a candidate key, that has been marked by a designer. It usually has some meaning for identifying tuples in relation.

- *Foreign key*
  Let us have two schemas of relations R_1 and R_2, which share some subset of attributes. Then K_p is a *foreign key*, if K_p is a primary key of R_2 and $K_p \subseteq R_1$ (attributes of R_1 include the primary key of R_2).
  The relation, which contains the foreign key (r_1), is called the *referencing relation*.
  The relation for which K_p is a primary key (r_2), is called the *referenced relation*.
  In practice, specifying a foreign key rises a *constraint* on the data: $\forall$ tuple $t_1 \in r_1$ there must exist a tuple $t_2 \in r_2$, such that $t_1[K_p] = t_2[K_p]$

2.1.3 Relational Schema Diagram Notation

A relational database schema can be visualized by a *relational schema diagram*. Its notation is similar to ER [24] or UML [35], but the notation is highly focused and specific for the relational data model. In contrast to, for instance, UML class diagram, it is very easy.

The rules for drawing relational schema diagrams are as follows:

- Relations are drawn as boxes with the relation name above the box.
- Attributes of given relation are listed within its box.
- The attributes belonging to the primary key of relation are listed first. There is a line separating primary key from other attributes and the background is gray.
- Foreign key dependencies are drawn as arrows from the referencing relation to the referenced relation.

An example of relational database schema is illustrated in Figure 2.1. The example describes documents in an school information system - relation *document*. Each document was written by a student, relation *student*, and it belongs to some user, relation *user*. Each document contains keywords, relation *keyword*. Assigning of the keyword to the given document is represented by a relation *doc-keyword*.

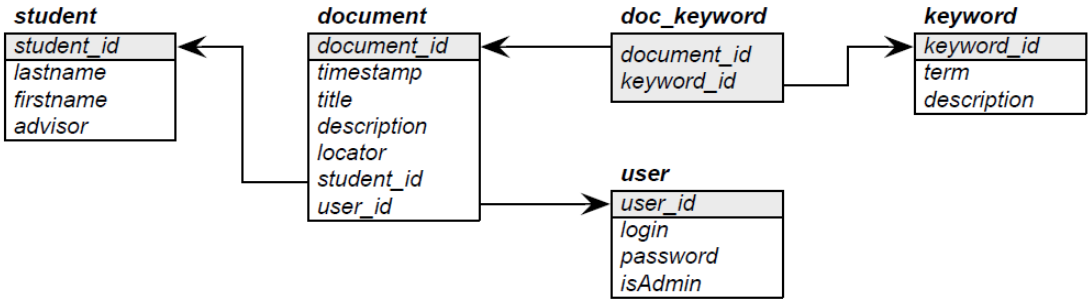


Figure 2.1: A sample of relational schema diagram

2.2 SQL Language

The SQL language became a de facto standard language for relational databases. Therefore every relational database management system (RDBMS) uses SQL as its front end.

The SQL language is, in fact, composed of three parts:

- *Data Definition Language* (DDL)
  This part of the language specifies constructs for defintion of schema, relations, integrity constraints and views.

- *Data Manipulation Language* (DML)
  This part of the language specifies construct for inserting, deleting, updating and querying the data in the relations (tables).

- *Data Control Language* (DCL)
  This part of language is used to control access to data stored in the relations. It specifies operations for which privileges may be granted to (or revoked from) a given database user or role.

2.2.1 The SELECT Statement

In many modern uses of databases, all we really need to do with the database is to select some subset of the records from given table(s), and let some other program manipulate the result. For these operations SQL uses the SELECT statement.

The SELECT statement is consist of six clauses:

- **SELECT**
  This part is called *projection*. It specifies the columns or computations with columns, which will appear in the result set of records corresponding to the given query.
  It can also contain so called *aggregate functions*, which are computations applied on the whole result set of given column. Aggregate functions are especially mathematical functions, like minimum, average, sum, etc.
  The SELECT clause is obligatory in each query.

- **FROM**
  This clause describes data sources for a given query. It can consist of one or more tables, which are linked through Cartesian product or standard JOIN construct of relational algebra [25].
  The clause is obligatory in each query.

- **WHERE**
  This clause is called *restriction*. It specifies search conditions that are applied to the result of the preceding FROM clause. If the record satisfies these conditions, it will appear in the result set.
  The clause is optional.

- **GROUP BY**
  This clause is used together with combination of aggregate functions. The clause partitions the set of records in a table into groups based on the given criteria and computes aggregate function for each group. All records from preceding clauses (FROM claues and WHERE clause if applicable) that agree on a set of grouping attributes are put into a corresponding group. Every group puts one record (tuple) to the output. All the grouping attributes must also appear in the SELECT clause.
  The usage of this clause is linked with occurrence of aggregate functions in the SELECT clause.

- **HAVING**
  This clause is sometimes called *second restriction*. It is used to elimination

of groups from the preceding GROUP BY claues. It eliminates groups that do not satisfy a given search condition.
The clause can appear only with combination with previous the GROUP BY clause, but it is optional.

- **ORDER BY**
  This clause is used to specify the order of records in the result set. There can be used a column from the SELECT clause or any expression.
  The clause is optional.

All the following examples of SQL queries are related to the database schema illustrated in Figure 2.1.

The first example illustrates simple usage of standard SELECT statement.
The query returns *firstname* and *lastname* of all students in the table *student*.

```
SELECT
      firstname
    , lastname
FROM
      student
```

The next example illustrates usage of the clauses WHERE and ORDER BY. The query returns *title*, *timestamp* and author's *firstname* and *lastname* of those *documents*, whose author's *firstname* is 'John'. The result has to be ordered by author's *lastname*.

```
SELECT
        d.title
      , d.timestamp
      , s.firstname
      , s.lastname
FROM
      document d
     JOIN student s ON (s.student_id = d.student_id)
WHERE
      s.firstname = 'John'
ORDER BY
      s.lastname
```

The last example illustrates usage of the clauses GROUP BY and HAVING. The query returns *firstname*, *lastname* and number of documents he or she has written of those *students*, who have already written more than one document.

```
SELECT
      s.firstname
    , s.lastname
    , COUNT(*) as number_of_documents
FROM
    student s
    JOIN document d ON (s.student_id = d.student_id)
GROUP BY
```

```
        s.firstname
      , s.lastname
HAVING
    COUNT(*) > 1
```

2.2.2 Views

Views in SQL provide virtual relations which contain data spread across different tables. They are used because:

- they simplify query formulations,
- they hide the real database schema and hide inappropriate data,
- they provide a logical data indepence,
- they do not need to be stored as permanent tables (but they can - we speak about *materialized views*).

View Statement

The view can be defined simply by syntax:
CREATE VIEW VIEW-NAME AS <SELECT STATEMENT>
This creates a View with a name <VIEW-NAME> with a structure and data defined by the result of the <SELECT STATEMENT>.

Chapter 3

Model Driven Architecture

In this chapter we briefly introduce *model driven architecture*, an approach to system development.

3.1 Model Driven Architecture

The *Model drive architecture* (MDA), [31] is an approach to system development, which increases the power of models in that work. It is *model-driven*, because it provides means for using models to understanding, design, construction, deployment, operation, maintenance and modification.
MDA deals with the idea of separating the specification of the operation of a system from details of the way that system uses the capabilities of its platform.
MDA provides an approach for the following actions:

- Specifying a system independently of the platform that supports.
- Specifying platforms.
- Choosing a particular platform for the system.
- Transforming the system specification into one for a particular platform.

3.1.1 Computation Independent Model

A *computation independent model* (CIM) is a view of a system from the computation independent viewpoint. It focuses on the environment of the system, and the requirements for the system. The details of the structure and processing of the system are hidden or as yet undetermined.

3.1.2 Platform Independent Model

A *platform independent model* (PIM) is a view of a system from the platform independent viewpoint. It focuses on the operation of a system while hiding the details necessary for a particular platform. PIM represents part of the complete specification that does not change from one platform to another.
For modeling of PIMs, there is often used a general purpose modeling langugage, like UML. It is also possible to use a language specific to the area in which the system will be used.

3.1.3 Platform Specific Model

A *platform specific model* (PSM) is a view of a system from the platform specific viewpoint. It combines the platform independent viewpoint with an additional focus on the detail of the use of a specific platform by a system.

3.1.4 Model Transformation

A *model transformation* is the process of converting one model to another model of the same system.
From the perspective of MDA, the most interesting is model transformation from PIM to PSM. This model transformation can be described as follows:

1. The PIM is completed with special mapping marks, which defines general mapping rules.
2. There is chosen specific platform.
3. Model transformations corresponding to the mapping marks are executed according to the chosen specific platform.

Figure 3.1 illustrates a schema of the Billing system according to MDA layers. The CIM layer represents a general policy of the given company about the billing process. The PIM layer represents an analysis model of the billing system. The PSM layer contains many various models, which are particularly designated for specific purpose, e.g. database model, web server model, etc.

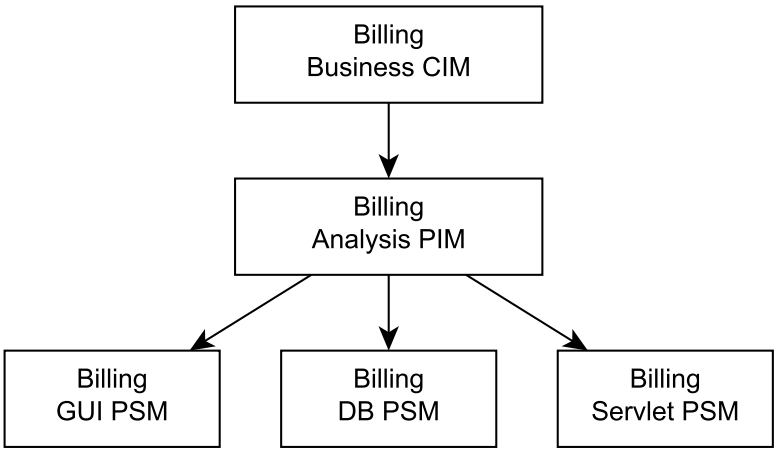


Figure 3.1: An example of the usage of MDA in system development.

Chapter 4

Database Schema Adaptation

This chapter contains an introduction into database schema adaptation issues. From the global perspective the main issues of database schema adaptation are described.

Database schema adaptation is a complex operation which starts mainly with change of database schema and involves change of each subsystem (in general) related to given database schema.
Each discussed issue has a smaller or greater relation to the database schema adaptation, but it is necessary to consider each of them.

4.1 Database Schema Integration

Database schema integration issue involves combining of integrating data sources concepts and knowledge into an integrated view that isolates users from the system organization. Constructing an integrated view of data sources can be difficult because they will store different types of data, in varying formats, with different meanings, and will reference it using different names. In addition, constructing an integrated view requires resolving different mechanisms for storing data (structural conflicts), for referencing data (naming conflicts), and for attributing meaning to the data (semantic conflicts).

Types of schema integration

There are two major types of schema integration:

- **View Integration**
  This type is used during the design of a new database when user requirements may be different for each user group. The process of view integration merges different viewpoints into a single data model.

- **Database Integration**
  This type is used when two or more databases have to be combined to create a single schema. We call this schema a global schema.

Problems of schema integration

Paper [6] introduces basic problems of database schema integration. These problems can be summarized into 4 main categories:

- **Different Perspectives**
  The problem is occuring when two database schemas are designed by different designers using different user requirements. The resulting schemas will often present contrasting views of the same data.
  For example, the relationship between *employee* and *project* in one database is represented as a relationship between *employee*, *department* and *project* in another database.

- **Equivalent Concepts**
  Different databases are dealing with the same concept in a different way.
  There are two situations that must be dealt with:

  1. *Different concepts are modelled in the same way.* For instance, in a university database staff and students are both represented by the entity person even though they are different concepts.
  2. *The same concepts are modelled in different way.* For instance, a given property is designed as an entity in one database, but as an attribute in another database.

- **Incompatible Designs**
  Two database designs might be incompatible because there are different constraints placed on the data or mistakes were made in the initial design. For example, relationship between two entities is represented as one-to-many in one database and many-to-many in another one.
  During schema integration these different viewpoints must be unified.

- **Resolving conflicts**
  During the integration of independant schemas different conflicts may appear.

  - **Name conflicts**
    Name conflicts may cause a problem, because information is going to be duplicated in the integrated database. It is important to identify correct data items in each schema that should be represented using different structures in the integrated schema or that actually represents the same concept.

    Synonyms - two similar concepts occur with different names.
    Homonyms - two different concepts occur with the same name.

  - **Structural conflicts**
    Structural conflicts occur when the actual method of representing the same concept in different databases is different or incompatible.
    There exist three cases:

    * *Identical Concepts*
      The same concept in different databases is represented in the same

way.
These entities can be merged directly.

* *Compatible Concepts*
The same concept in different databases is represented in compatible way.
These entities can be merged through some given conversion way.

* *Incompatible Concepts*
The same concept in different databases is represented using different structures. It is often difficult to merge them directly.
Incompatible designs have to be often resolved by analysing the data and adapting one or more of the schemas. Also there can be constructed a new, common representation.

Strategies of schema integration

There exist two main strategies for applying database schema integration:

- All-in-one
The strategy is based on merging all schemas into a single large schema (called the *global schema*).
This approach is going to be difficult when the schemas are large or when there is a large number of schemas.

- Stages
This strategy is also called *gradual schema integration*. At first there are integrated some schemas, then there are gradually integrated remaining schemas to the result schema from the step before.
This approach is going to be more appropriate when the schemas are complex or when there are a large number of schemas.

4.2 Database Schema Versioning and Schema Evolution

Schema versioning is one of related areas dealing with the same general problem - using multiple heterogeneous schemas for various database related tasks. Together with its weaker companion, schema evolution, it deals with the necessity to retain current data and software system functionality despite the changing database structure.

Paper [4] gives a good definition of terms of schema modification, evolution and versioning to understand the difference in their usage.

- **Schema Modification**
Schema Modification means that a database system allows for changes to the schema definition of a populated database.

- **Schema Evolution**
Schema Evolution means that a database system facilitates the modification of the database schema without loss of existing data.

- **Schema Versioning**
  Schema Versioning means that a database system allows the accessing of all data, both retrospectively and prospectively, through user definable version interfaces.
  Schema versioning can be divided into two categories by distinguishing between retrieval and update activity:

  - *Partial Schema Versioning*
    Partial Schema Versioning is a weaker concept of versioning. It means that a database system allows the viewing of all data, both retrospectively and prospectively, through user definable version interfaces. Data stored under any historical schema may be updated only through the current or active schema.
  - *Full Schema Versioning*
    Full Schema Versioning means that a database system allows the viewing and update of all data, both retrospectively and prospectively, through user definable version interfaces. It means that data stored under any historical schema may be updated through any other schema, not only the current or active schema.

The considerable difference between evolution and versioning is the capability for users to identify stable points in the definition for later reference. Schema evolution does not require the ability to version data except when each changed schema can be considered as a new version. Also it does not require that the database system provides a view-mechanism using past schema definitions.

Schema evolution does not imply full historical support for the schema. It only requires the ability to change the schema definition without loss of data. In contrast, schema versioning (even in its simplest form), requires that a history of changes has to be maintained to enable the retention of past schema definitions.

Schema changes will not necessarily result in creating a new version. More often schema changes will be grained finer than the definable versions.

Domain/type evolution

Domain/type evolution is the simplest type of database schema evolution. It involves changes of attributes' domain, which results in change of attributes' type in given *database management system* (DBMS). It could seem, that there is not any problem with such evolution. But the research in [5] shows the importance of capturing the semantics of a domain and the identification of that semantic content within the metadata.

Domain evolution is a great example of the difference between schema evolution and schema versioning. In schema evolution process, existing instances must be converted to the new format and thus existing applications become incompatible. Schema versioning supports program compatibility by leaving the existing definition in place.

Relation/class evolution

Relation and class evolution involve attribute and relation/class definition, redefinition, deletion and class modification. Suggestions from the research indicate that modification of the database schema to accommodate changes at the relational or class design (and above) can be achieved in a number of ways. For instance, within the relational model a set of atomic operations is proposed which result in a consistent and, as far as possible, reversible database structure, [7].

4.3 Data Migration

Data migration problem is closely related to schema adaptation issue.
Data migration is in general the process of transfering data between two computer systems. In relational databases it often includes moving from one DBMS to another, or upgrading the version of database software being used.
The latter case is less likely to require a physical data migration, but this can happen with major upgrades. A physical transformation process in this case could be required since the underlying data format can change significantly.
The data migration process can affect behaviour in the applications layer, depending on whether the data manipulation language or protocol has changed – but current applications are written to be independent on the database technology, so usually it is only necessary to make data migration test in database layer.

The data migration project plan usually involves these 7 phases, which relate to the overall project:

- Strategy
- Analysis
- Design
- Build
- Testing / Implementation
- Revise
- Maintain

Strategy

The strategy phase is often the easiest part of the project planning process. The focus of the overall project is determined in this phase. The data migration projects do not operate independently. Rather they are a part of other development efforts. The project managers are clearly focused on determining the requirements that the new system must satisfy, and pay little or no attention to the data migration that must occur. It is common to review a project plan for a new system and discover that data migration is merely listed as a single task item.

Analysis

The analysis phase of data migration project should be scheduled to occur concurrently with the analysis phase of the core project. The aim of the analysis phase in data migration projects is to identify the data sources that must be transported into the new system. An important part of the analysis phase involves becoming acquainted with the actual data you plan to migrate. At this point of the project, there is no idea if the data is even of high enough quality to consider migrating. It can be helpful to obtain reports that can provide row and column counts, and other statistics regarding your source data. This kind of information gives us a rough idea of just how to migrate data. You can find that the cost of migration is prohibitive relatively to the quantity of data that needs to be moved.

Design

The design phase involves going through the list of data elements from each and every source data structure, and deciding whether to migrate each one. The design phase is not intended to identify the transformation rules by which historical data will be pushed into the new system. Rather, it is basically the act of making a checklist of the legacy data elements that must be migrated.

Build

The generation of the new data structures and their creation within the database is the task of this phase.

Testing / Implementation

Testing and implementation are often combined into one phase, because these two phases are practically inseparable. The first step is to execute the mapping. Testing breaks down into two areas: logical errors and physical errors.

Physical errors are usually syntactic so they can be easily identified and resolved. Physical errors have nothing to do with the quality of the mapping effort. Rather, this level of testing is dealing with semantics of the scripting language used in the transformation effort.

Implementation is the area of identifying and resolving logical errors. The best test of data mapping is providing the populated target data structures to the users that assisted in the analysis and design of the core system. These users will begin to identify scores of other historical data elements that must be migrated that were not apparent to them during the analysis/design sessions.

Revise

In the revise phase a clean-up is managed. All data model modifications, transformation rule adjustment, and script modification are basically combined to form for the revise phase.
In this phase the maintaining of logical and physical data model is managed.

CASE tools can be used to maintain the link between the logical and physical models. It is highly sensible to select a CASE tool providing an API to the meta-data, because it will be needed most certainly.

Maintain

In the maintenance phase all of the mappings are validated and successfully implemented in a series of scripts that have been thoroughly tested. The maintenance phase differs depending upon whether the migration is to an On-line Transaction Processing (OLTP) [9] or an On-line Analytical Processing (OLAP) [10]) system. The aim of migration to an OLTP system is to successfully migrate the legacy data into the new system, rendering the migration scripts which become useless when the migration has been accomplished.
In the migration to an OLAP system, the new system will be reloading most likely in timely intervals. As new information is recorded in the OLTP system, it is necessary to transfer it to the OLAP system. Script performance is a highly critical issue in OLAP migrations, while OLTP migrations pay no attention (or little) to script performance since they will only be run once.

4.4 Adaptation of the Database Queries

Adaptation of the database query issue is highly related to database schema adaptation issue, but, in addition, it is also very close to the *database schema evolution* issue, mentioned before.
Database query adaptation is the process of reformulation of database query according to the way the underlaying database schema has been changed.
Each DBMS usually contains these queries related to schema change:

- Views
- Constraints
- Stored procedures

The aim of the research on this issue is the possibility of reaction for a database schema change and automated reformulation of the given queries.

Chase and Backchase

Chase and backchase is the original algorithm for enumerating the reformulations of a query under constraints. The algorithm was developed by [11]. But a research showed, that for its properites (especially correctness and completeness) it could be simply generalized and might be used in other areas, like XML [12], [13] or in complex database schema evolution framework [14].

The algorithm proceeds in two phases:

- **Chase**
  Let us have a rewriting (extending) rule D. If query Q contains the left-hand-side of D, then the right-hand-side of D is added to Q as a conjunct.

This does not change answers of *Q'* — if Q satisfies left-hand-side of *D'*, it also satisfies right-hand-side of *D'*.

Such query extension is repeated until Q cannot be extended any further. We call the largest query obtained at this point a *universal plan*, U.

- **Backchase**
  At this phase, the system removes from U every atomic formula (conjunct) that can be obtained back by a chase. This step does not change the answer, either.
  Atomic formulas of *U'* are repeatedly removed, until no atomic formula can be dropped any further. So we obtain another equivalent query *Q'*.

Simple example:

- *Let us have a query Q*:
  SELECT DISTINCT STRUCT (E: t.TMember)
  FROM depts d, d.DProjs pn, Teams t, Payroll p
  WHERE pn = t.TProj and d.DName = 'Security' and p.PDept = d.DName and p.Empl = t.TMember

- *By the chase phase it is extended by views V_1 and V_2 and constraints to universal plan, which is equivalent to default query Q*:
  SELECT DISTINCT STRUCT (E: t.TMember)
  FROM depts d, d.DProjs pn, Teams t, Payroll p, V_1 v_1, V_2 v_2
  WHERE pn = t.TProj and d.DName = 'Security' and p.PDept = d.DName and p.Empl = t.TMember
  and v_1.D = d.DName and v_1.P = pn and v_1.E = p.Empl
  and v_2.D = d.PDept and v_2.E = t.TMember and v_2.P = t.TProj

- *By the backchase phase it is induced by maximal reduction of constraints obtainable by chase phase to another equivavalent query*:
  SELECT DISTINCT STRUCT (E: t.TMember)
  FROM V_1 v_1, Teams t
  WHERE v_1.D = 'Security' and v_1.E = t.TMember and v_1.P = t.TProj

Chapter 5

Related Work

In this chapter papers related to evolution of database schema are discussed. The chapter contains analysis of 4 works dealing with a selected database schema adaptation problem. Each analysis introduces the given problem, the solution described in the work and possibilities of the solution.
The last section of this chapter gives a summary comparison of all discussed works and briefly introduces main issue of this work.

5.1 Database Schema Integration Process

Paper [1] describes an approach for integration of complex database schemas. The main aims of the paper are:

- To design a database schema through the approach of a gradual integration of external schemas.
- To suggest a new conceptual modeling design able to be used for conceptual database schema design instead of ER data model.
- To develop a CASE tool which provides complete support for schema integration process.

5.1.1 Phases of the Database Schema Design Process

Design of a complex database schema is based on a gradual integration of external schemas. An *external schema* is a complex structure that formally defines a user view on a database schema (at the conceptual level). The authors divide the whole process of the database schema design into 5 phases:

1. Identifying groups of similar end-user business tasks
   In the first phase it is necessary to design separate external schema for each group of similar end-user business tasks. Each program that supports a user request is based on an associated external schema.
2. Conceptual modeling
   In this phase each external schema is integrated into a common conceptual database schema. The set of the subschemas represents an outcome of this

phase.
The authors use *form type concept* (see Section 5.1.2) for conceptual database schema design in contrast to mainly used ER data model or UML class diagrams.

3. Transformation into relational data model
   Each resulting subschema is translated into relation data model.

4. Generating a database schema
   A potential database schema is created by integration of relation subschemas. This operation can be done by applying the synthesis algorithm [3].

5. Consolidation
   The process of independent design of external schemas may lead to collisions in expressing the real system constraints. The last phase provides a mechanisms for detection of such constraint collisions and their resolving.

5.1.2 Form Type Concept

Form type concept is an approach for conceptual database schema design. The authors assume that the form type concept may be used for conceptual database schema design instead of ER data model or UML class diagrams. The main idea of this one approach comes from the late 80's (see [2]).
The concept is based on the fact that users communicate with an information system through application forms. So the designer's work is to specify screen forms of transaction programs.
The main reason for using form type is the fact that the concept is more familiar to end-users' perception of information system, than the concepts of entity and relationship types in ER data model. In addition, form type is a concept that is formal enough to precisely express all the rules signifcant for structuring future database schema.

Example

Figure 5.2 shows the structure of a form type *F*, which generalizes an application form of Operation plans in Figure 5.1. The form type consists of one component type *OP*, graphically represented by a rectangle. Underlined attributes indicate component type keys.

Operating Plans

Identification	From Date	Until Date

Figure 5.1: An example of a screen form.

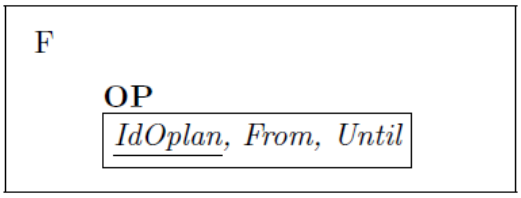


Figure 5.2: An example of a form type.

5.1.3 IIS*Case

IIS*Case (Integrated Information Systems*Case) is a resulting software developed by the authors of the paper, that supports an approach for gradual integration of external schemas. It is based on the form type concept mentioned before.
IIS*Case is designed to provide complete support for developing complex database schemas with regard to the number of concepts used, and to give an intelligent support in the course of the whole schema integration process. IIS*Case supports:

- Conceptual modeling of external schemas.
- Automated design of the so-called relational database subschemas in the 3rd normal form [23].
- Automated integration of relational database schema from designed subschemas.
- Detecting and resolving the constraint collisions between a database schema and a set of subschemas.

5.1.4 Discussion

The paper presents an original approach to automatic integration of database schemas. The approach is based on the form type data model. From the designer's point of view, the form type data model offers a simple way for defning the initial set of attributes and constraints. For the designers, the database design of even complex information systems became easier because the process of modeling is raised to the level of a user without an advanced knowledge of the database design.
The paper also suggests some methods, how to detect and finally resolve collisions between independly designed database schemas. The success of approach is guaranteed by many results of theoretical research.
As the big advantage of the paper can be found that presented approach is supported by own CASE tool - IIS*Case.

5.2 QuickMig

Paper [15] describes a semi-automatic approach to determining semantic correspondences between schema elements for data migration applications.

5.2.1 The QuickMig Migration Process

The QuickMig approach divides a process of migration into 5 phases:

1. Answering a Questionnaire
2. Injection of Sample Instances
3. Schema and Instance Import
4. Matcher Execution
5. Review

Answering a Questionnaire

The purpose of the first phase is to collect as much information as possible about the source system. This data will be used to automatically reduce the complexity of the target schemas. This also helps with reduction of the complexity of the matching process.
The first phase has to be performed manually by a person with some knowledge of the capabilities of the source system.

Injection of Sample Instances

In this phase instances existing in the target system are manually created in the source system. These samples are used by the instance-based matching algorithms in order to determine correspondences between the source and the target schemas. Using sample data this knowledge can be exploited by the matching algorithms. By injecting sample data into the source system the matching algorithms do not only have arbitrary instances available but one dedicated instance which maps exactly to a specific instance of the target schema.

Schema and Instance Import

In the third phase of migration process the source schemas as well as correspoding sample instances are imported into the QuickMig system.

Matcher Execution

In this phase schema matching algorithms are executed. These automatically determine a mapping proposal using different matching algorithms. The resulting mapping proposal includes similarities between elements of the source and target schemas as well as a proposal for the *mapping categories*.

Review

In the final phase a developer reviews and corrects the mapping proposal. When the mapping proposal is acceptable, real mapping code is generated and the mapping is stored.

5.2.2 Mapping Categories

Each returned correspondence from the matching phase (phase 4 of migration process) is associated with some *mapping category*. These mapping categories are used to create parts of the necessary mapping expressions. At least it can be used to provide a mapping expression template that can be easilly completed by a developer.

QuickMig distinguishes the following 11 mapping categories:

- CreateInstance
- KeyMapping
- InternalId
- LookUp
- Move
- ValueMapping
- Code2Text
- DefaultValue
- Split
- Concatenate
- Complex

5.2.3 The QuickMig Architecture Overview

The QuickMig is based on COMA++ [16]. QuickMig extends COMA++ by implementing three new matching algorithms:

- **Equality Matcher**
  It is the simplest matcher. It tries to identify equal instance values in the source and target schema.
- **SplitConcat Matcher**
  This matcher checks the instance data for splitting or concatenation relationships.
- **Ontology-Based Matcher**
  This matcher exploits background knowledge provided in a domain ontology in addition to instance data in order to identify corresponding schema elements.

Each matching algorithm returns a list of correspondences between the source and the target schema and also the associated mapping category. The results of all algorithms are combined in order to create the resulting mapping.

5.2.4 Discussion

This paper presents QuickMig, a system for the semi-automatic creation of schema mappings in data migration projects. The system is based on 5-phase migration process. The most important phase is the phase of matching execution, where the mapping proposal is created and each found correspondence is associated with some mapping category.
The proposal approach was experimentally evaluated using real SAP [36] schemas. In these experiments QuickMig achieved very good results. QuickMig is planned to be integrated into SAP data migration tools in future.

5.3 The PRISM Workbench

Paper [14] describes an advanced approach to support database schema evolution in traditional information systems. In such projects the frequency of database schema changes has increased while tolerance for downtimes has nearly disappeared. The main aim of the paper is to create a tool to manage and automate:

- predicting and evaluating the effects of the proposed schema changes,
- rewriting queries and applications to use the new schema, and
- migrating the database.

5.3.1 SMO Language

The whole evolution process is described by *Schema Modification Operators* (SMO). Each SMO represents an atomic operation performed on both schema and data. All SMOs together represents the *SMO language*.
Basically, PRISM provides for database administrator the simple SMO language to express schema changes in the process of designing evolution steps. There exist the following SMOs:

- Create Table $R(\overline{A})$
  - $\rightarrow R(\overline{A})$
- Drop Table R
  $R(\overline{A}) \rightarrow$ -
- Rename Table T into R
  $R(\overline{A}) \rightarrow T(\overline{A})$
- Copy Table R into T
  $R_{V_i}(\overline{A}) \rightarrow R_{V_{i+1}}(\overline{A}), T(\overline{A})$
- Merge Tables R, S into T
  $R(\overline{A}), S(\overline{A}) \rightarrow T(\overline{A})$
- Partition Table R into S (satisfies the condition) and T
  $R(\overline{A}) \rightarrow S(\overline{A}), T(\overline{A})$

- Decompose Table R into $S(\overline{A}, \overline{B})$, $T(\overline{A}, \overline{C})$
  $R(\overline{A}, \overline{B}, \overline{C}) \rightarrow S(\overline{A}, \overline{B}), T(\overline{A}, \overline{C})$
- Join Table R, S into T according to condition
  $R(\overline{A}, \overline{B}), S(\overline{A}, \overline{C}) \rightarrow T(\overline{A}, \overline{B}, \overline{C})$
- Add Column C into table R
  $R(\overline{A}) \rightarrow R(\overline{A}, C)$
- Drop Column C from R
  $R(\overline{A}, C) \rightarrow R(\overline{A})$
- Rename Column B in R to C
  $R_{V_i}(\overline{A}, B) \rightarrow R_{V_{i+1}}(\overline{A}, C)$
- NOP (no operation)
  - $\rightarrow$ -

Marking

- Right arrow ($\rightarrow$) means the change of the left side to the right side.
- $\overline{A}$ - schema or subschema of the table, i.e. all columns or subset of columns of a given table.
- $R(\overline{A})$ - table R with the schema $\overline{A}$.
- $R(\overline{A}, \overline{B})$ - table R with the schema containing two disjoint subschemas $\overline{A}$ and $\overline{B}$.
- $R(\overline{A}, C)$ - table R with the subschema $\overline{A}$ and column C, which is not a part of the subschema $\overline{A}$.
- $R_{V_i}(\overline{A})$ - table R has the schema $\overline{A}$ in the database schema version V_i.

5.3.2 Evolution Process

The process of evolution is divided into 5 phases:

1. Evolutin design
2. Inverse generation
3. Validation and query support
4. Materialization and performance
5. Deployment

Evolutin Design

The first phase of evolution process can be further divided into the sequence of 6 steps:

- The database administrator expresses by the SMO language the desired atomic changes to be applied to the input schema.
- The system virtually applies the SMO sequence to input schema and creates the candidate output schema.
- The system verifies whether the desired evolution is resistant against loss of data during executing the evolution. If it is, we call the evolution as *information preserving*.
- Each SMO in sequence is analyzed for redundancy. For instance, *Copy table* SMO generates redundancies. A database administrator has to decide whether such redundancy is intended or not.
- The system translates the desired SMO sequence into a logical mapping between schema versions. This mapping is expressed by so-called Disjunctive Embedded Dependencies (DEDs).
- Query chase engine rewrites the queries expressed over the old schema into equivalent queries expressed over the new schema. Alternatively the SMO sequence is translated into SQL Views corresponding to the mapping between versions to support queries over the data stored in the basic tables.

The whole phase can be iterated until the candidate schema is satisfactory and the final schema is obtained.

Inverse Generation

In this phase the system, on the basis of the forward SMO sequence, computes the candidate inverse sequence. Then is checks, whether the inverse SMO sequence is information preserving. If both forward and inverse sequences are information preserving, the schema evolution is guaranteed to be completely reversible.

Validation and Query support

In this phase the system translates the inverse SMO sequence into a logical mapping using DEDs. As in the last step of the first phase, queries expressed over the new schema are rewritten into equivalent queries expressed over the old schema. Also corresponding SQL views are generated. According to results of this phase, the database administrator can repeat phases 1 to 3 to improve the evolution.

Materialization and Performance

This phase can be further divided into 4 steps:

- The system translates both the forward and inverse SMO sequences into SQL data migration scripts.

- On the basis of the previous step the system materializes new schema and supporting queries in the old schema by views or query rewriting.
- Rewritten queries are tested against the materialized new schema for absolute performance testing.
- Old queries are tested natively against the old schema. The results are compared with the results of the previous step.

The database administrator can improve performance by modifying the schema layout, for instance by modifying indexes in new schema.

Deployment

In the last phase, the old schema is dropped and old queries are supported by SQL views or by query rewriting. The whole evolution process is recorded into an enhanced *information schema*, to allow schema history analysis. It is possible to preform a *late rollback* by generating an inverse data migration script from inverse SMO sequence.

5.3.3 Data Migration And Query Support

The logical mapping between versions is expressed by DEDs. Each SMO in an input SMO sequence is converted into DED, so each DED represents a given mapping rule between the old schema and the new schema. Each SMO produces both forward and backward mapping. Forward mapping expresses how to migrate data from the source (old) schema version to the target (new) schema version.

Using the generated DEDs, the queries are rewritten by engine using the *chase and backchase* algorithm [11].

SMO to SQL

SMOs are tailored to assist data migration tasks, therefore many SMOs combine actions on both schema and data. In phase 4 of evolution process, each SMO is translated into corresponding SQL data migration code.

PRISM also supports expressing of the mapping between versions in terms of SQL views. This often happens in the data integration area. Views are usually used to enable what-if analysis (forward views), or to support old schema versions (backward views). Each SMO can be translated into a corresponding set of SQL views.

5.3.4 Discussion

This paper presents PRISM workbench, a system to support database schema evolution. The whole evolution process is based on 5-phase evolution process, including manual design of desired evolution by SMO language and automated work of PRISM tools to perform corresponding schema evolution, generate data migration scripts and rewrite queries into equivalent one expressed on the new

database schema.
The system was tested on Wikipedia database schema. Its 170+ schema versions provided good testing environment for validating PRISM tools and ability to support legacy query rewriting.

The PRISM deserves further research, especially in the field of optimization of performance or query rewriting, but it is clear that the PRISM takes a big step toward needs of database administrators to have methods and tools to manage and automate the whole process of database schema evolution.

5.4 Adaptive Query Formulation

Papers [17] and [18] describe a graph-based aproach to database schema evolution.

5.4.1 Graph-based Model

The authors propose a graph-based model that uniformly covers relational tables, views, database constraints and SQL queries. Formally, the given database schema is represented as directed graph $G = (V, E)$, where V are the nodes of the graph representing the entities of the model, and E are the edges representing the relationships between these entities.
There are the following essential components:

- **Relations**
  The relation in the database schema is represented as a directed graph, which includes:
  - *Relation node*, representing relational schema.
  - *Attribute node*, one for each attribute.
  - *Relationship edges* directing from relation node to attribute nodes, indicating belonging of an attribute to the relation.
- **Conditions**
  The conditions refer to selection conditions (of queries and views) and constraints (of a database schema). A *condition node* represents a given condition. The node is tagged with appropriate operator and it is connected to the *operand nodes*. Composite conditions are simply constructed by tagging the condition node with a Boolean operator and the appropriate edges to the conditions composing the given composite condition.
- **Queries**
  The graph representation of the query includes a *query node* and *attribute nodes* corresponding to the query projection. In order to determine relationship between the query and relations, the query is divided into these essential parts:
  - *Select part*
    This part of the query maps appropriate attributes of the involved relations to the attributes of the query projection through edges of

type *map-select* directing from the query attributes toward the relation attributes.

- *From part*
  This part of a query is considered as the relationship between the query and involved relations. The relations involved in this part are combined with the query node through edges of type *from-relationship* directing from the query node towards the relation nodes.
- *Where and Having parts*
  These clauses are assumed to be in *conjunctive normal form*(CNF) [30]. There exist two edge types *where-relationship* and *having-relationship* directing from a query node towards an operator node at the highest level of the conjunction.
- *Group and Order By parts*
  For this part there are two special nodes: *group-by node* (GB) to capture the set of attributes acting as the aggregators and *aggregate function node*. There are the following types of edges: *group-by* directing from query node to GB node, and from GB node to each aggregator, and *map-select* directing from each aggregated attribute to *aggregate function node* and directing from *aggregate function node* to appropriate relation attribute.
  Order-by clause is performed similarly.

- **Views**
  Views are considered either as queries or in case of materialized views as relations.

5.4.2 Evolution Policies

In the context of the proposed graph-based model, changes in the database schema are events, which transform specific parts of the graph and eventually affect other dependent graph constructs, which recursively may raise new changes, which have an impact on other graph constructs.
To handle schema evolution, the graph constructs have to be annotated with policies that allow the designer to specify the behavior of a given construct whenever change events occur. This combination of event and policy triggers the execution of the appropriate action - blocks the event or reshapes the graph respectively.

Possible events are defined as Cartesian product of a set of hypothetical actions (addition, deletion, modification) and set of graph constructs, which are subject of evolution (relations, views, attributes and conditions).
There exist three kinds of policies, which can be used with a given event:

- *Propagate* the change, i.e. the graph has to be reshaped to adjust new semantics.
- *Block* the change to retain old semantics and the event has to be blocked.
- *Prompt* the user to interactively decide what will eventually happen.

5.4.3 SQL Extensions

The authors propose an extension of a database system catalog with extra information regarding evolution purposes. Each assertion is considered as a tuple (*event, policy*).
These assertions extend SQL syntax both in DDL statements as well as in SQL queries. The general syntax is:

$$ON < event > THEN < policy >$$

An *event* refers to evolution events in the database schema (delete, add, modify, rename) and a construct type (node, relation, query, view, attribute, condition, PK, FK, NNC, UC). The *policy* can take the values mentioned in previous section (propagate, block, prompt).

5.4.4 Discussion

The papers present a graph-based approach to performing database schema evolution. They focus on propagating potential changes of the database system to all the affected parts of the system.
There was introduced a complex graph-based model, which covers the whole database systems, including such elements like queries or views. Also there was introduced an extension to the SQL language specifically tailored for the management of evolution.
The applicability and efficiency of this approach has been tested in real-world evolution scenario extracted from an application of the Greek public sector. The main goal of the test was to minimize the human effort required for defining and setting the evolution metadata by using the proposed language extension.

5.5 Comparision of the Related Works

All presented works study a problem of database schema adaptation, but each of them focuses on a separate part of this complex issue.

The first paper [1] describes an approach to the process of integration of complex database schemas. It is based on the form-type concept, which is used for conceptual database schema design in contrast to mainly used ER data model or UML class diagrams.

The second paper [15] describes a semi-automatic approach to determining semantic correspondences between schema elements for data migration applications. It focuses on the matching process where a mapping proposal is created and each found correspondence is associated with some mapping category. These are then used to create parts of the mapping expressions.

The third and the fourth works study the same problem - database evolution, but each from the different point of view.
The third work [14] focuses on the support of database schema evolution in traditional information systems. It introduces the SMO language used to design desired evolution changes.

The last work [18] is based on graph model of the whole database system, including views, queries or constraints. Such graph is annotated by policies to

specify behavior of given database system construct whenever change events occur.

As the result of the third work there is a sequence of traditional SQL statements, in contrast to the last work, where the evolution is performed in place and all affected elements are automatically adjusted into new version.

In this thesis we will focus on the problem of database queries adaptation. We will focus especially on the following problems which were not solved in these papers:

- On modeling of database queries in a CASE tool together with the database schema model.
- On the relation between database schema and database queries.
- On the propagation of changes between database schema and database queries.

Chapter 6

Database Model

In this chapter we describe a *Database Model*, which will be used in this thesis as a model for representing a database schema.

6.1 PIM Layer

According to MDA approach, PIM layer defines essential concepts comprising the system and relationships between these concepts. Thanks to interconnections between PIM and PSM, it is possible to propagate changes done in PIM layer to corresponding constructs in PSM layer.
For the purpose of this thesis, it is not important what kind of model is used in PIM diagram. Figure 6.1 shows an example of such PIM diagram based on UML class diagram. The example illustrates a draft of a schema of orders in e-shop. The diagram is consist of:

- *Classes* - visualized as yellow rectangles. Each class has a nonempty name. For instance, in Figure 6.1 *Customer* or *Order* are classes.
- *Attribute* - visualized as a child item of any class. Each attribute has a nonempty name and belongs to exactly one class. For instance, *firstname*, *lastname*, *email* and *phone* are attributes of the class *Customer*.
- *Associations* - visualized as a line connecting two classes. Each association has a name and cardinalities used with the particular part of the association. For instance, association *contains* means that the order contains ordered items. The cardinality setttings means that each ordered item belongs to exactly one order and each order contains at least one ordered item.

6.2 PSM Database Model

The PSM database model is based on the relational database model. Such database model is used as platform-specific according to MDA approach.
In this thesis, the database model has still another role. It is used as PIM model for a query model described in Section 7.2. The idea of using the database model this way is simple. From the perspective of arbitrary query language (not only SQL), the database model creates a basic concept of a database schema. Each

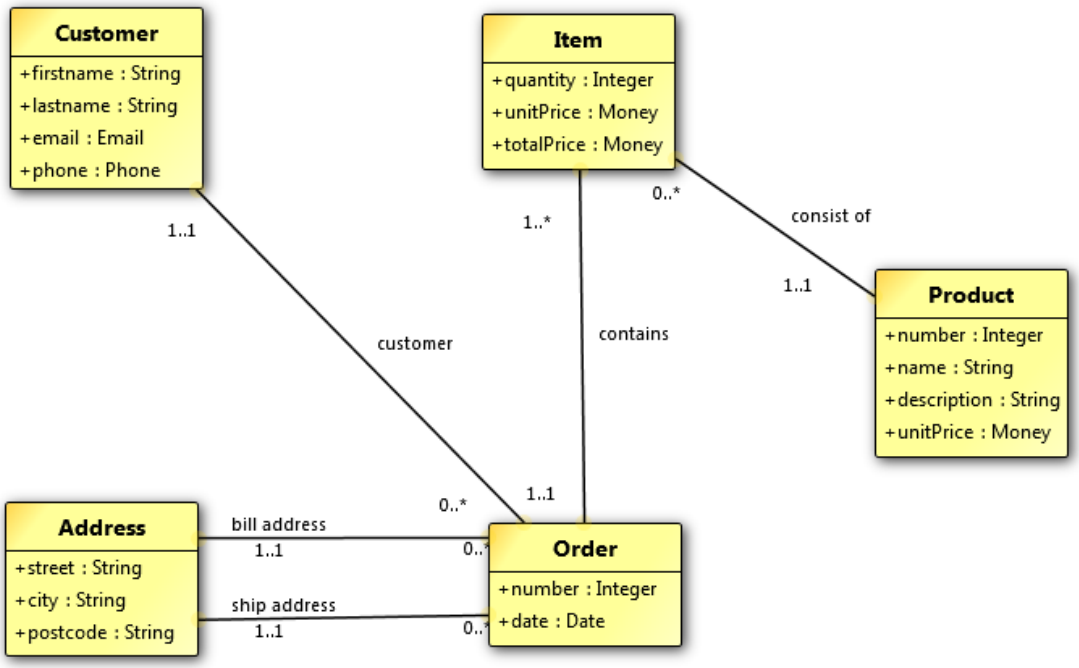


Figure 6.1: An example of PIM diagram.

query language then creates own platform-specific point of view on the database model.

The PSM database model was implemented in the DaemonX framework [32] as a modeling plug-in of PSM database schema. Figure 6.2 illustrates the example of PSM database model, which corresponds to the PIM model in Figure 6.1, i.e. the draft of a schema of orders in e-shop. The PSM database model diagram is consist of:

- *Tables* - visualized as rectangles with a nonempty black name in a blue field. For instance, in Figure 6.2 *Address* or *Product* are tables.
- *Columns* - visualized as a child item of any table. Each column has a nonempty name, datatype and belongs to exactly one table. For instance, *productId*, *name*, *description* and *unitPrice* are columns in the table *Product*.
  The column can be further marked as:
  - *Primary Key* - the column represents a primary key of the table - red font color and golden key icon
  - *Foreign Key* - the column is the foreign key to the another table - blue font color and grey key icon
  - *Not Null* - the column cannot have a null value - black font color, red bead icon and the tag *NN* behind the name
  - *Null* - the column can have a null value - light grey font color and blue bead icon

– *Unique* - the column has a unique value - the tag U behind the column name, the font color and the icon according to the *Null / Not Null* property

- *Relationships* - visualized as an arrow leading from the referencing table to the referenced table. The relationship has a nonempty name. Each referencing table contains a primary key column from the referenced table as a column marked as *foreign key*.

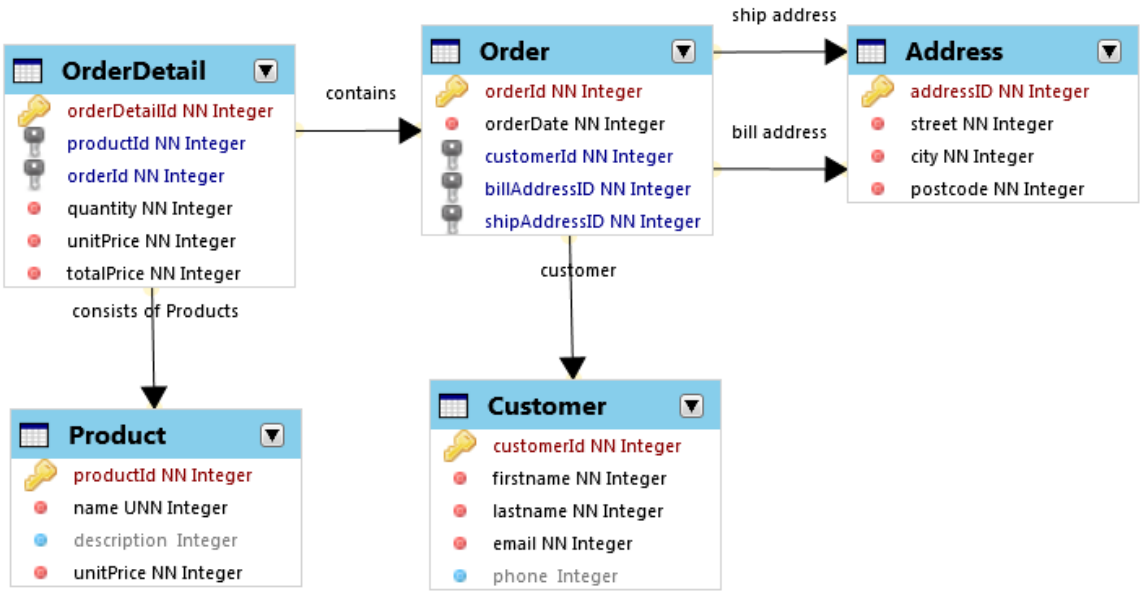


Figure 6.2: The example of PSM database model.

6.2.1 Model Constructs

The model contains the following constructs:

- **Table** represents a given table in a given database system. Each table construct can represent only one class from the PIM model. Every table construct has a name and contains column constructs. Table concstructs can be connected by relationships.

- **Column** is a construct that can exists only as a child of a table construct. Each column construct represents at most one PIM attribute. Every column construct has a name, a data type, an indicator if it is used as primary key, nullable or with a unique value.

- **Relationship** construct represents a connection between two table constructs. Every relationship construct has a role, which represents a logical label for such connection, and cardinality of relationship. There can exist more than one relationship between two table constructs, but each such relationship must have a unique role.
  The relationship between two tables is not only logical or virtual. It is based

on the concept of *keys* and *foreign keys* 2.1.2. If exists a column construct C, which is in table construct T_1 used as primary key and in table construct T_2 used as foreign key, there must exist a relationship construct connecting constructs T_1 and T_2.

Chapter 7

SQL Query Model

For possibility of evolution of SQL queries related to a given database schema there must exist a mapping between SQL query and a database schema model. This mapping helps to manage the evolution process to evolve the query related to the evolved database schema.

In this chapter we will introduce a graph-based SQL query model, which is particularly designed for evolution process. We will describe its visualization model, limitations and possibilities. The mapping between SQL query model and the database schema will be introduced as well. Also we will describe an algorithm to generate the SQL query from the query model.

7.1 Limitations of the Query Model

Full SQL language syntax is too extensive to be used in this thesis. For this reason there exist some limitations on the used subset of SQL language.

- Projection operator '$*$' is banned to use. It is always necessary to enumerate all the columns used in the SELECT clause.
- In queries it is possible to use only simple column enumeration, other expressions or functions other than aggregate functions are banned to use. This limitation relates to the *CASE* construct as well.
- The model does not support *UNION*, *INTERSECT* and *EXCEPT* constructs.
- Each condition used in the SQL query is assumed to be in *conjunctive normal form* (CNF) [30].

7.2 Graph-Based Query Model

The idea of the graph-based model comes from papers [17] or [18]. In this paper we use graph-based model for query modeling, in contrast to mentioned papers, where graph-based model is implemented as part of *database management system* (DBMS). The idea of graph-based model from [17] and [18] is adjusted and extended for purposes of this thesis.

Each SQL query is in the model represented as a directed graph with particular properties.

Definition 7.1. (Query Graph). A query graph G of the SQL query Q is a directed graph $G_Q = (V, E)$, where V is a set of *query vertices* and E is a set of *query edges*.

Definition 7.2. (Query Model). A query model M of the SQL query Q is a pentad $M_Q = (G_Q, T_V, T_E, \tau_V, \tau_E)$, where G_Q is a query graph $G_Q = (V, E)$, T_V is a set of vertex types {*AggregateFunction, Alias, BooleanOperator, CombineSource, ComparingOperator, ConstantOperand, DataSource, DataSourceItem, From, FromItem, GroupBy, Having, OrderBy, OrderByType, QueryOperator, Select, SelectItem, Where*}, T_E is a set of edge types {*Alias, Condition, ConditionOperand, DataSource,DataSourceAlias, DataSourceItem, DataSourceParent, FromItem, FromItemParent, FromItemSource, GroupBy, GroupByColumn, Having, MapColumn, MapSource, OrderBy, OrderByColumn, SelectColumn, SelectQuery, SourceTree, Where*}, a function $\tau_V : V \to T_V$ assigns a type to each vertex of the query graph G_Q and a function $\tau_E : E \to T_E$ assigns a type to each edge of the query graph G_Q.

A *query vertex* represents a particular part of the SQL query, e.g. database table, table column, comparing operator in condition, selected column in the SELECT clause, etc. A *query edge* connects parts of the SQL query together and gives a particular semantics to this connection. For instance the edge connecting a From vertex and a Where vertex means that the query contains a WHERE clause represented by the Where vertex.

Each query graph can be logically divided into smaller subgraphs. These subgraphs are called *essential components*. Each essential component has a visual equivalent in *query visualization model* described in Section 7.3.

There exist the following essential components:

- DataSource
- From
- Select
- Condition
- GroupBy
- OrderBy

The simplest SQL queries of the form *'SELECT projection FROM table'* require only DataSource, From and Select components.

7.2.1 DataSource Component

A *DataSource* component represents a general source of data, primarily it is used to model database tables of a given database schema. But we can also consider it as a database view, a result of a given database function, or something else. The DataSource component represents an input point of evolution process (see Section 7.4). The component comprises the following types of vertices or edges:

DataSource Vertex

This vertex represents a particular source of data, for instance a database table. The vertex contains a name of the source of data.

DataSourceItem Vertex

This vertex represents one attribute of a particular source of data, for instance a column of a given database table. For each attribute of a given source there exists exactly one DataSourceItem vertex. The vertex contains a name of the attribute.

DataSourceItem Edge

This edge is directing from DataSource vertex to DataSourceItem vertex. The edge indicates that the given DataSourceItem belongs to the particular DataSource. From one DataSource vertex there can lead zero to n edges of this type. The edges are indexed from 0. The index determines an order of the DataSourceItem in the source of data.

DataSourceParent Edge

This edge is directing from DataSourceItem vertex to DataSource vertex. The edge indicates that the DataSource vertex is a parent of the DataSourceItem vertex. Exactly one edge of this type must lead from one DataSourceItem vertex.

Figure 7.1 shows visualization of the query graph of the DataSource component. The depicted DataSource represents a database table *Customer* with columns *firstname* and *lastname*.

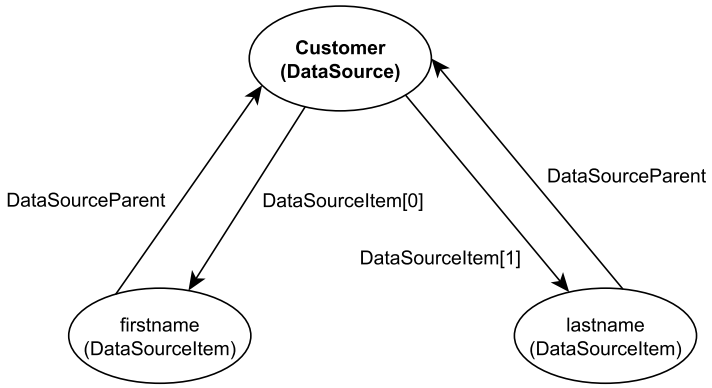


Figure 7.1: An example of DataSource query graph component.

7.2.2 From Component

A From component is used to model the FROM clause of the SQL query. The component comprises the following types of vertices or edges:

From Vertex

This vertex is the main vertex of the From component. In the context of whole SQL query it represents the FROM clause. Algorithm 7.4 for generating SQL query from the model starts at this vertex. Also it is the important point of algorithms, which distribute changes in query model and which are described in Section 8.2.

FromItem Vertex

This vertex represents one column from a possible set of query results. Each FromItem vertex corresponds to exactly one DataSourceItem vertex. Each attribute from each source of data used in the particular query must be covered by FromItem vertex. The vertex contains the full name of the covered attribute, i.e. the name with the alias of the attribute parent.

FromItem Edge

The edge of this type is directing from a vertex of type From to vertices of type FromItem. The edge indicates that the FromItem belongs to the given From vertex. The From vertex can have zero to n adjacent vertices connected by the edge of the FromItem type.

FromItemParent Edge

The edge is directing from the FromItem vertex to the From vertex. The edge indicates that the FromItem is a parent of the From vertex. Each FromItem vertex must have exactly one adjacent vertex of the type From connected by edge of the FromItemParent type.

FromItemSource Edge

This edge is directing from the DataSourceItem vertex to the FromItem vertex. The edge indicates a mapping of the DataSourceItem (e.g. a table column) into the column representation of the particular query.

Alias Vertex

This vertex contains a used alias for a particular source of data used in the FROM clause. This vertex always exists, even if there is not defined any alias in the query. In such case the alias has a value of the name of the given source of data.

DataSource Edge

This edge is leading from the AliasVertex to the DataSource vertex. It indicates that the Alias vertex is related to the given DataSource vertex. Each Alias vertex have exactly one adjacent neighbour connected by the edge of this type.

DataSourceAlias Edge

This edge is leading from the AliasVertex to the From vertex. It indicates that the Alias vertex represents an alias of the given DataSource. Each alias vertex have exactly one adjacent neighbour connected by the edge of this type.

Alias Edge

The edge of this type is used in three places:

- The edge which leads from the From vertex to the Alias vertex means membership of the given DataSource in the particular query.
- The edge which leads from the FromItem vertex to the Alias vertex represents parent of the given FromItem in the particular query.
- The edge which leads from the DataSource vertex to the Alias vertex means that the given DataSource appears in the particular query as the alias with the name of the Alias vertex value.

SourceTree Edge

This edge is leading from From vertex to the so called *MainSource vertex*, which:

- In the case of the simple SQL query with only one DataSource, this edge is leading direct to the Alias vertex of this DataSource.
- Otherwise it is leading to the root of the *combination source tree* (see later).

Each From vertex covering any DataSource has exactly one MainSource vertex.

The illustration in Figure 7.2 shows a simple example of the modeled FROM clause with only one database table. The example is equivalent to the following part of the SQL query:

FROM Order O

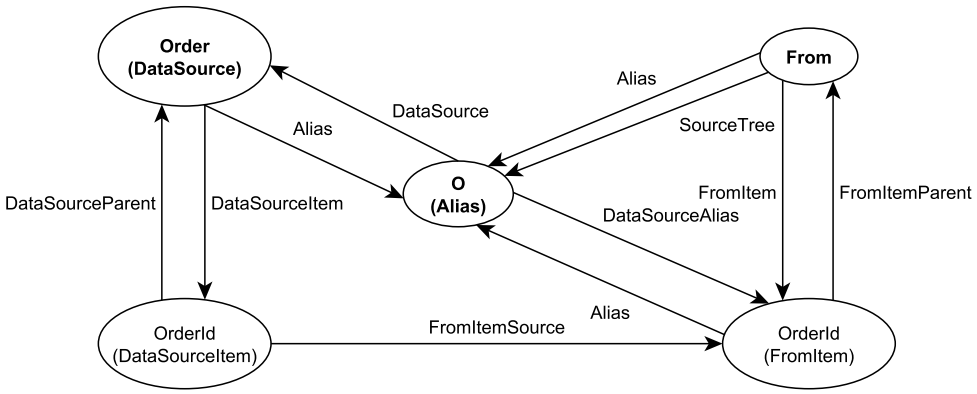


Figure 7.2: An example of the simple model of the FROM clause.

Combination Source Tree

Even simple SQL queries often contain a combination of database tables. In SQL table combinations are performed by Cartesian product or by variations of JOIN construct. For these types of queries there exists a structure called *combination source tree*.

It is an acyclic graph structure which comprises the following vertices and edges:

- **CombineSource vertex**, which represents combination of two database tables by the given type of combination (i.e. Cartesian product, Join, etc.).
- **MapSource Edge** is the type of the edge, which connects CombineSource vertex with other level of the *combination source tree*. The first edge (indexed from 0) is leading to Alias vertex, if there is no other combination. Otherwise, if there exists another combination of DataSources, it is leading to CombineSource vertex.
  The second edge is always leading to Alias vertex and it represents currently connected DataSource into the tree.
- **Condition Edge**. This edge is leading from the combine-source vertex to the root of the *condition tree* (see Section 7.2.4). This edge exists only if the combination type of combine-source vertex is not the Cartesian product, because in such case there is necessary to specify condition of the DataSource connection.

Figure 7.3 illustrates the part of the modeled FROM clause containing a combination source tree. The example is equivalent to the following part of the SQL query:

```
FROM
    Order
    JOIN OrderDetail ON condition1
    JOIN Customer ON condition2
```

7.2.3 Select Component

A Select component is used to model the SELECT clause of the SQL query. The component comprises the following types of vertices or edges:

Select Vertex

This is the main vertex of the Select component. In the context of the SQL query it represents the SELECT clause. It is the important part of algorithms which distribute changes in the query model as well.

SelectQuery Edge

This edge is directing from a From vertex to a Select vertex. It indicates that the SELECT clause represented by the Select vertex belongs to the FROM clause represented by the From vertex. From each From vertex only one edge of the

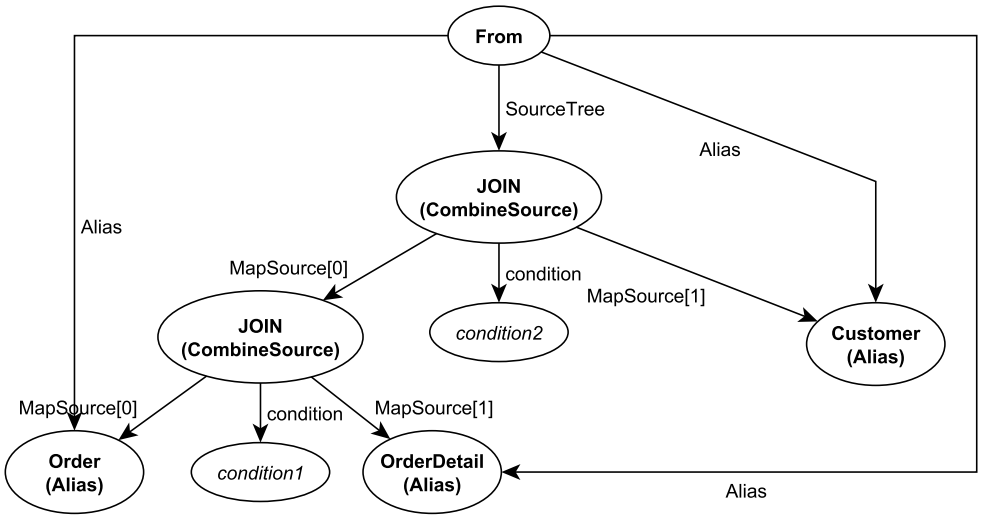


Figure 7.3: An example of the combination source tree model.

SelectQuery type can lead and to each Select vertex can lead only one edge of the SelectQuery type.

SelectItem Vertex

This vertex represents a selected column in the SELECT clause of the SQL query. The task of this vertex is to provide a name of the selected column. In case that in the selected column expression there is an alias, it contains its value. Otherwise it contains the full name of the source item (i.e. a name of the DataSourceItem including the alias used for the parent DataSource).

SelectColumn Edge

This edge connects a Select vertex with a SelectItem vertex. It indicates the membership of the selected column with the particular Select vertex. To each SelectItem vertex must lead exactly one edge of the SelectColumn type. The edges are indexed from 0, where index determines the order of the selected columns in the SELECT clause.

AggregateFunction Vertex

This vertex represents an aggregate function used in the SELECT clause. The possible represented aggregate functions are the following:

- Count
- Sum
- Average
- Maximum
- Minimum

MapColumn Edge

The edge of this type can be used in three places:

- The edge which leads from the SelectItem vertex to the FromItem vertex means a mapping of the selected column to its source column in FROM part of the query.
- The edge leading from the SelectItem vertex to the AggregateFuntion vertex means that the given aggregate function is used in the particular select column expression.
- The edge leading from the AggregateFunction vertex to the FromItem vertex indicates the usage of the given source column as the argument of the aggregate function.

Figure 7.4 shows an example of the modeled SELECT clause. The example simply illustrates a difference when an alias for the column is used or not. The example is equivatelent to the following part of the SQL query:

```
SELECT
      c.CustomerId as cid
    , c.Name
FROM
    Customer c
```

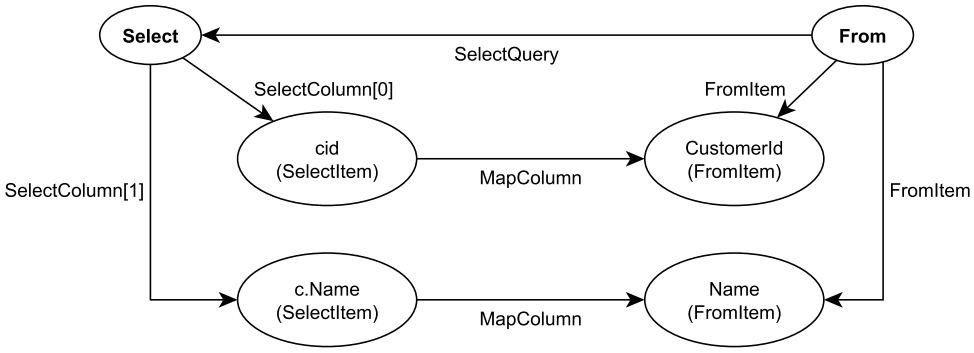


Figure 7.4: An example of the simple model of the SELECT clause.

Figure 7.5 shows the example of a usage of the aggregate function in the SELECT clause. The example is equivalent to the following part of the SQL query:

```
SELECT
      COUNT(c.CustomerId) as customerCount
FROM
    Customer c
```

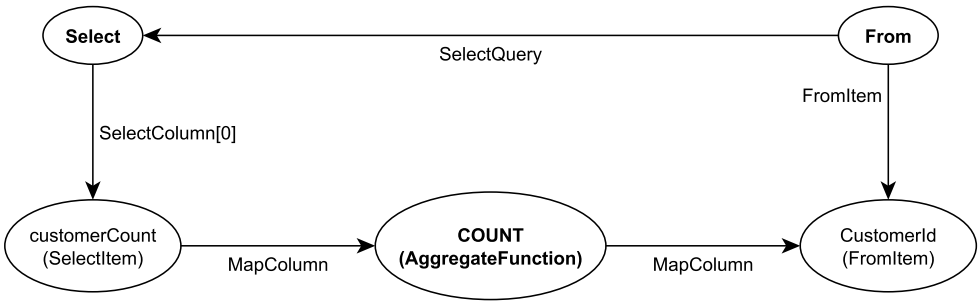


Figure 7.5: An example of a usage of an aggregate function in the SELECT clause.

SQL Query as DataSource

Each SQL query can be used as a DataSource in the FROM clause of another SQL query. The *query model* provides this possibility as well. The connection between the Select component and the From component is similar to the connection between the DataSource component and the From component, but there exist following exceptions:

- The *Alias edge* leading to the Alias vertex is directed from the Select vertex and it has the same meaning as the edge between the DataSource vertex and the Alias vertex.
- The *FromItemSource edge* is leading from each SelectItem vertex belonging to the SELECT clause to the FromItem vertex. The edge indicates a mapping of the selected column into the column representation of the particular query.

Because of usage of an SQL query as a DataSource, there exist FromItem vertices, which create a transparent interface for other components this way, instead of direct mapping between DataSourceItem vertices and for instance SelectItem vertices.

Figure 7.6 shows the example of the usage SQL query as a DataSource of another SQL query.

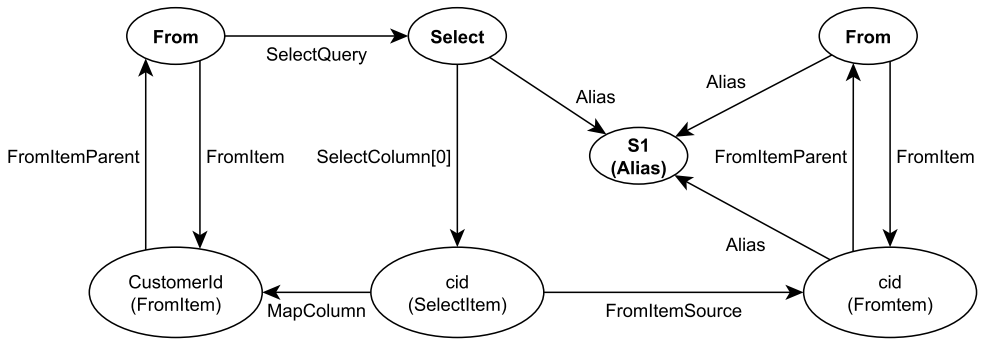


Figure 7.6: An example of a usage of the SQL query as a DataSource.

7.2.4 Condition Component

This component is used to model conditions in SQL queries. Conditions occur in the following parts of the SQL query:

- WHERE clause.
- HAVING clause.
- Condition of the DataSource combination.

Condition Tree

A *condition tree* is an acyclic graph structure, which can cover a model of each condition used in the SQL query. It comprises the following types of vertices or edges:

- **ComparingOperator Vertex**
  This vertex represents the conditions with a comparing operator. The form of these condititions is A *op* A' or A *op constant*, where A and A' refer to the table column or usage of the aggregate function with the given table column, and *op* is a binary operator ($<$, $>$, $>=$, $<=$, $=$, $!=$, $LIKE$, $NOT\ LIKE$) or a unary operator ($IS\ NULL$, $IS\ NOT\ NULL$). The vertex has one or two adjacent vertices according to the arity of used operator.

- **QueryOperator Vertex**
  This vertex represents conditions which have the form of $A\ op_b\ Q$ or $op_u\ Q$, where A refers to the table column, op_b is a binary query operator (IN, $NOT\ IN$, ANY), op_u is an unary query operator ($EXISTS$, $NOT\ EXISTS$) and Q is a query. The vertex has one or two adjacent vertices according to the arity of used operator.

- **ConstantOperand Vertex**
  This vertex represents any constant value used in conditions.

- **AggregateFunction Vertex**
  In the context of conditions, this vertex is used in the HAVING clause of the SQL query.

- **BooleanOperator Vertex**
  This vertex represents boolean operator used to joining of conditions. Literals of conditions in CNF are joined by *OR* operator, clauses of conditions are joined by *AND* operator. Each BooleanOperator vertex has at least two adjacent vertices. All modeled literals of one clause are connected with one OR operator vertex, all modeled clauses are connected with one AND operator vertex. From this claim it is obvious that each condition contains at most one AND operator vertex.

  Only the ComparingOperator vertex and the QueryOperator vertex can be adjacent vertices of the BooleanOperator vertex of the type of OR. The BooleanOperator vertex of the type of AND can have the BooleanOperator vertices of the type of OR as adjacent vertices in addition as well.

- **ConditionOperand Edge**
  This is the only type of edge which is used in the whole condition tree. All vertices mentioned before are connected by this edge. The edges leading from the particular vertex are indexed from 0. The edge can lead between following types of vertices:
  - Two BooleanOperator vertices. This edge can lead only from AND boolean vertex to OR boolean vertex.
  - BooleanOperator vertex and ComparingOperator or QueryOperator vertex. The edge indicates the membership of the literal in the clause.
  - ComparingOperator or QueryOperator vertex and FromItem vertex. The edge indicates that the table column represented by FromItem vertex is compared to other column, constant value or query.
  - ComparingOperator or QueryOperator vertex and AggregateFunction vertex. The edge indicates that the result of the aggregate function is compared to constant value or query.
  - AggregateFunction vertex and FromItem vertex. The edge indicates the usage of the aggregate function with the table column represented by FromItem vertex as its argument.
  - ComparingOperator vertex and ConstantOperand vertex. The edge indicates that the table column is compared to constant value.
  - QueryOperator vertex and Select vertex. The result of the query represented by Select vertex is used as operand to be compared with other column of result of the aggregate function usage.

WHERE and HAVING Clauses

WHERE and HAVING clauses of the SQL query are modeled as follows:

- *Where Vertex*, which represents a WHERE clause.
- *Having Vertex*, which represents a HAVING clause.
- *Condition Edge*, which is leading from the Where or the Having vertex to the root of the condition tree. Each Where or Having vertex can be connected at most with one condition tree.
- *Where Edge*, which is leading from the From vertex to the Where vertex. It indicates a membership of the WHERE clause in the SQL query. Each From vertex can be connected at most with one Where vertex and vice versa.
- *Having Edge*, which is leading from the From vertex to the Having vertex. It indicates a membership of the HAVING clause in the SQL query. Each From vertex can be connected at most with one Having vertex and vice versa.

DataSource Combination Condition

The way of the usage of the condition in DataSource combination was described in Section 7.2.2.

Figure 7.7 shows a simple example of the condition in the WHERE clause of the SQL query. The example is equivalent to the following part of the SQL query:

```
WHERE
  OrderId = 1135
```

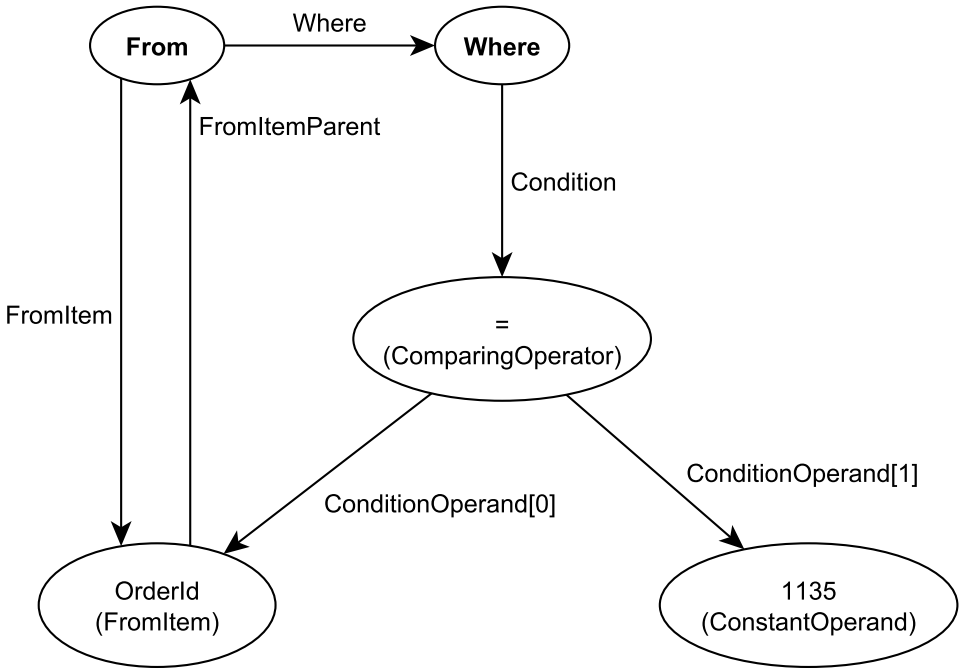


Figure 7.7: An example of the simple model of the WHERE clause.

Figure 7.8 shows a simple example of the condition in the HAVING clause of the SQL query. The example is equivalent to the following part of the SQL query:

```
HAVING
  COUNT(OrderId) > 100
```

Figure 7.9 shows a more comlex example of the condition using Boolean operators. The example is equivalent to the following part of the SQL query:

```
(orderDate >= '1.6.2011')
  AND
(city = 'New York' || city = 'Los Angeles')
  AND
(productName IN (Products))
```

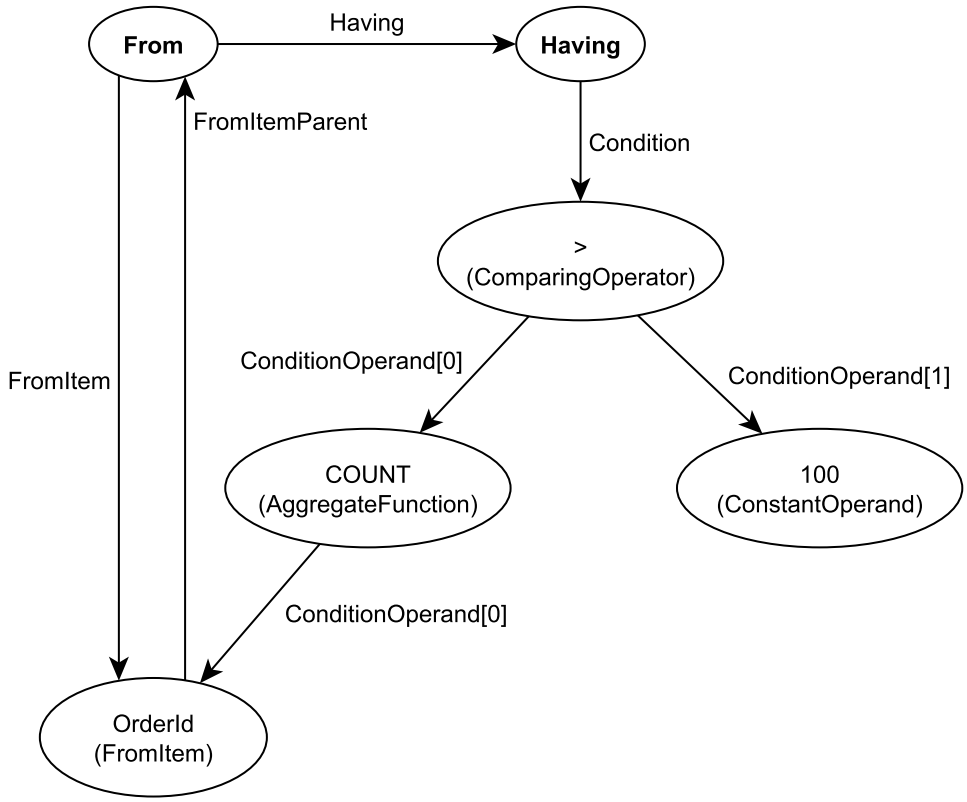


Figure 7.8: An example of the simple model of the HAVING clause.

7.2.5 GroupBy Component

A GroupBy component is used to model the GROUP-BY clause of the SQL query. The component comprises the following types of vertices or edges:

GroupBy Vertex

This main vertex of GroupBy component represents the GROUP-BY clause of the SQL query. It is the part of algorithms which distribute changes in the query model as well.

GroupBy Edge

This edge is directing from a Select vertex to a GroupBy vertex. It indicates the membership of the GROUP-BY clause in the SQL query. The edge must lead from the Select vertex, because the columns that are grouped depend on the selected columns in the SELECT clause.

GroupByColumn Edge

This edge is directed from a GroupBy vertex to a SelectItem vertex. It indicates that the given selected column is used in the GROUP-BY clause. Because of correctness of the SQL query, each SelectItem vertex which do not use an aggregate function has to be connected with the GroupBy vertex, but this applies only in

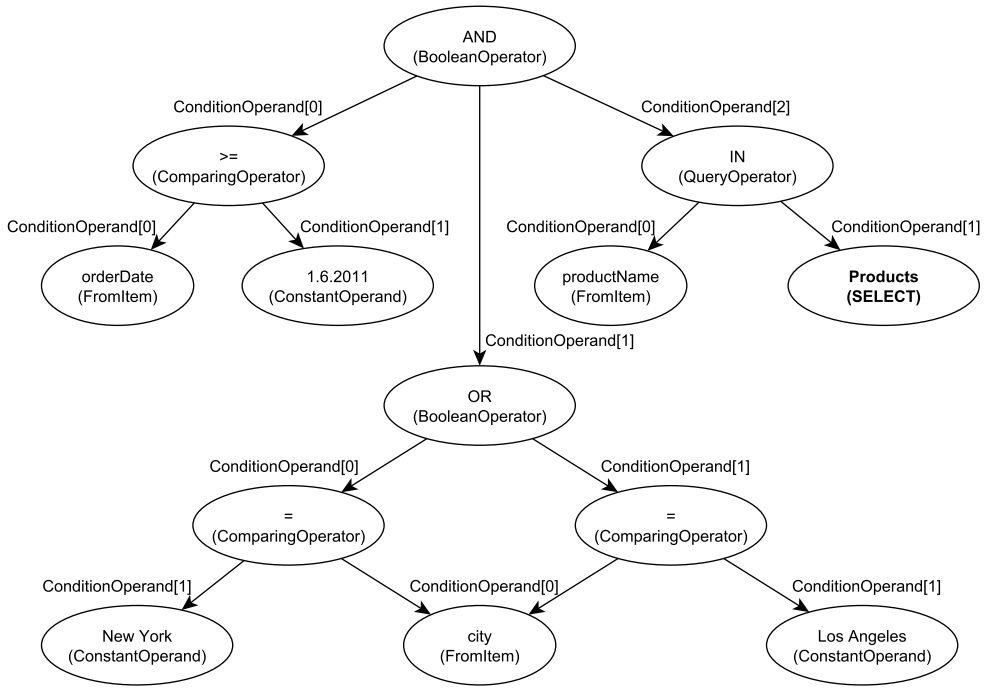


Figure 7.9: An example of the modeled complex condition.

case that there exists at least one SelectItem, which uses the aggregate function. The edges are indexed from 0 according to the order of the aggregator.

Figure 7.10 illustrates a simple example of the modeled GROUP-BY clause. The example is equivalent to the following parts of the SQL query:

```
SELECT
    CustomerId as cid,
    COUNT(OrderId) as orderCount
...
GROUP BY
    CustomerId
```

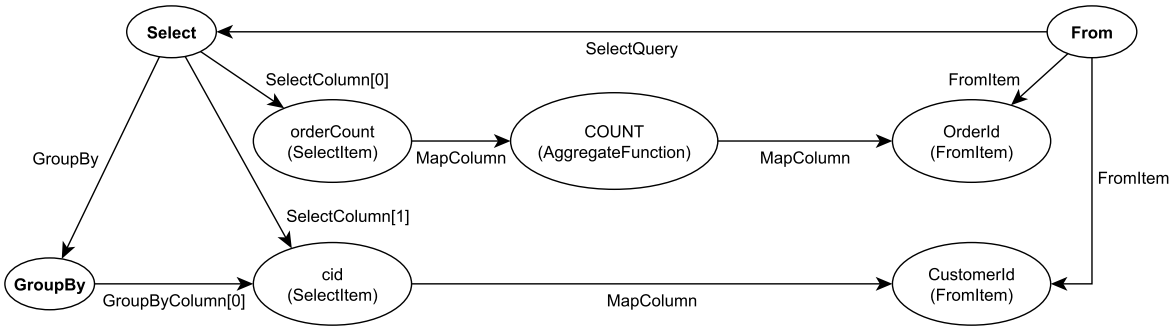


Figure 7.10: An example of a simple model of the GROUP-BY clause.

7.2.6 OrderBy Component

The OrderBy component is used to model the ORDER-BY clause of the SQL query. The component comprises the following types of vertices or edges:

OrderBy Vertex

This vertex represents the ORDER-BY clause of the SQL query. It is the part of algorithms which distribute changes in the query model as well.

OrderByType Vertex

This vertex represents a choice whether the sort by the given column is ascending or descending.

OrderBy Edge

This edge is directing from a From vertex to an OrderBy vertex. It indicates the membership of the ORDER-BY clause in the SQL query.

OrderByColumn Edge

This edge is used in three places:

- The edge directing from an OrderByType vertex to a FromItem vertex indicates that the given table column is used as an argument for sorting the SQL query result.
- The edge directing from an OrderByType vertex to a SelectItem vertex indicates that the given selected column is used as an argument for sorting the SQL query result.
- The edge directing from an OrderBy vertex to an OrderByType vertex indicates a direction of used sort by the given column.

All the edges are indexed from 0 in order to set the order of columns used in a sorting algorithm.

Figure 7.11 illustrates a simple example of the modeled ORDER-BY clause. The example is equivalent to the following part of the SQL query:

```
ORDER BY
    CustomerId ASC,
    Name DESC
```

7.3 SQL Query Visualization Model

Though graph-based query model can describe any SQL query, it is relatively complex. Even query model for a simple SQL query contains a lot of vertices and edges. For this reason we proposed a visualisation model, which simplifies an underlaying query model for users.

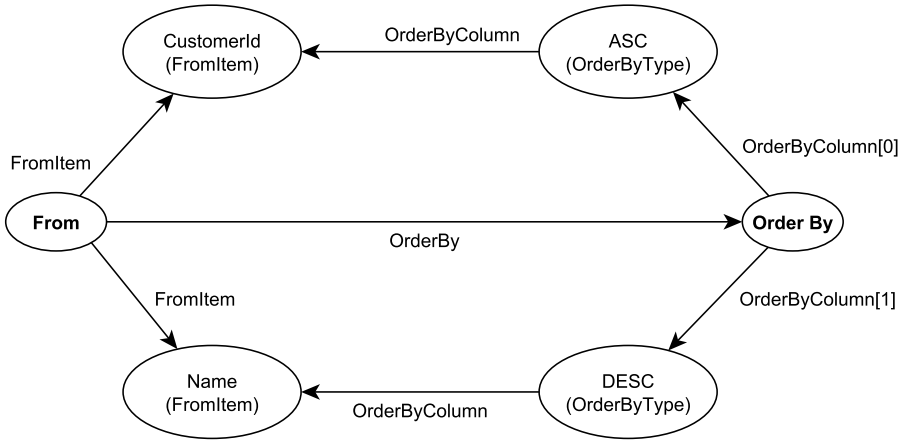


Figure 7.11: An example of a simple model of the ORDER-BY clause.

7.3.1 Visualisation Model Components

The visualisation model is divided into *essential visual components*. Each *essential component* mentioned in Section 7.2 has its visual equivalent by some essential visual component, so each visual component represents a part of the SQL query graph model.

We distinguish the following visual components:

- DataSource
- QueryComponent
- Component Connection

DataSource Visual Component

A *DataSource* visual component visualizes a DataSource essential component. Each DataSource has a name, which clearly identifies the given DataSource. The content of the DataSource component is the list of DataSourceItem visual components. The DataSource visual component itself corresponds to the DataSource vertex. The DataSourceItem visual components correspond to the DataSourceItem vertices.

Figure 7.12 shows an example of DataSource visual component. The example represents a database table *Customer* with columns:

- customerId
- firstname
- lastname
- email
- phone

Figure 7.12: An example of a DataSource visual component.

QueryComponent Visual Component

A *QueryComponent* is an universal visual essential component, which represents parts of the SQL query. A basic appearance of all query components looks the same. We distinguish the following types of query components according to the clause of the SQL query they represent:

- **Select**
  This represents the SELECT clause of the SQL query. It covers the Select essential component of the query graph.
- **From**
  This represents the FROM clause of the SQL query. It covers the From essential component of the query graph.
- **Where**
  This represents the WHERE clause of the SQL query. It covers a corresponding part of the Condition essential component of the query graph.
- **GroupBy**
  This represents the GROUP-BY clause of the SQL query. It covers the GroupBy essential component of the query graph.
- **Having**
  This represents the HAVING clause of the SQL query. It covers a corresponding part of the Condition essential component of the query graph.
- **OrderBy**
  This represents the ORDER-BY clause of the SQL query. It covers the OrderBy essential component of the query graph.

Figure 7.13 illustrates an example of the QueryComponent visual component. The example shows visualisation of the WHERE clause.

Component Connection

A *Component Connection* does not correspond directly to any essential component of the query graph. Instead of this it covers connection of two essential components to finish the correct and complete query graph. We distinquish the following types of connections:

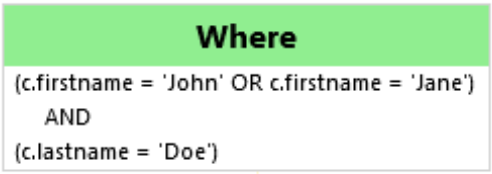


Figure 7.13: An example of a QueryComponent visual component.

- **DataSource → From**
  The connection represents the DataSource being a part of the SQL query, specifically of the FROM clause. It consists of Alias vertex, Alias edge and DataSource edge.

- **Select → From**
  The connection represents the SELECT clause belonging to the FROM clause in the SQL query. It consists of a SelectQuery edge.

- **Where → From**
  The connection represents the WHERE clause belonging to the FROM clause in the SQL query. It consists of a Where edge.

- **GroupBy → Select**
  The connection represents the GROUP-BY clause corresponding to the SELECT clause in the SQL query. It consists of a GroupBy edge.

- **Having → From**
  The connection represents the HAVING clause belonging to the FROM clause in the SQL query. It consists of a Having edge.

- **OrderBy → From**
  The connection represents the ORDER-BY clause belonging to the FROM clause in the SQL query. It consists of an OrderBy edge.

Figure 7.14 shows a visualisation model of a more complex SQL query. The modeled SQL query is the follows:

```
SELECT
      c.firstname
    , c.lastname
    , a.street
    , a.city
    , a.postcode
FROM
   Customer as c
   JOIN Address as a ON c.customerId = a.customerId
WHERE
   (c.firstname = 'John' OR c.firstname = 'Jane')
        AND
   (c.lastname = 'Doe')
ORDER BY
```

```
    c.customerId ASC
,   c.lastname ASC
,   a.postcode DESC
```

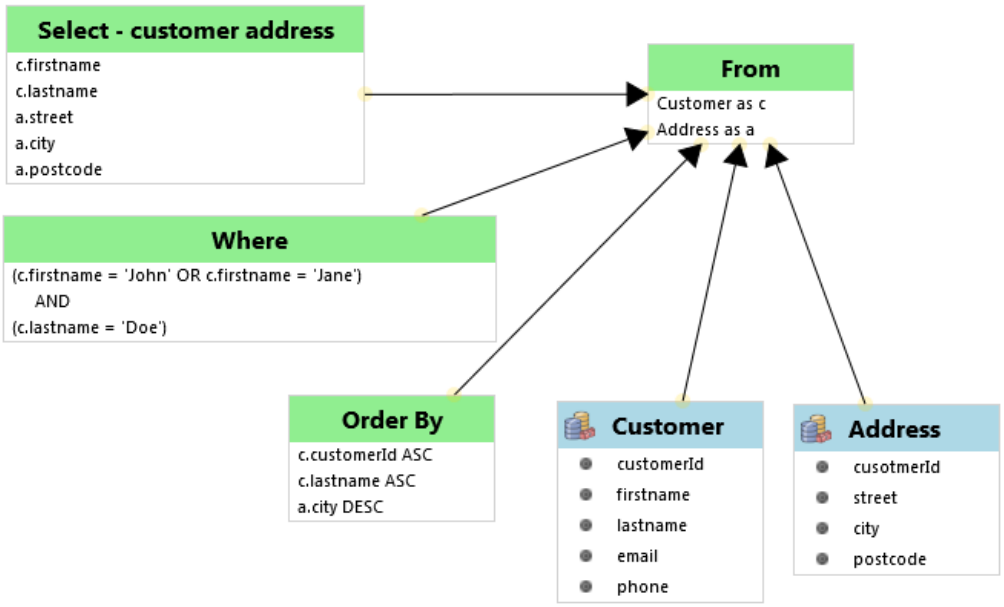


Figure 7.14: An example of a visual model of a more complex SQL query.

For comparision, the underlaying query graph model is consists of 45 vertices connected by 87 edges. A visualisation of this query graph can be found in Appendix B.

7.4 Mapping to Database Model

Since the database model consists of tables and its columns which we can interpret as a general source of data, we have a direct mapping from the database model to the query model. We do not consider database relationhips between database tables in the database model. For the purpose of the query model they are not important.

The mapping between the database model and the query model is described as follows:

- **Database table → DataSource**
  The database table in the database model is mapped to the DataSource visual component which corresponds to the DataSource vertex of the underlaying query graph.

- **Table column → DataSourceItem**
  The table column in the database model is mapped to the DataSourceItem visual component which corresponds to the DataSourceItem vertex of the underlaying query graph.

An example of the mapping is shown in Figure 7.15. The example illustrates the mapping from a database table *Customer* to a DataSource visual component named *Customer*.

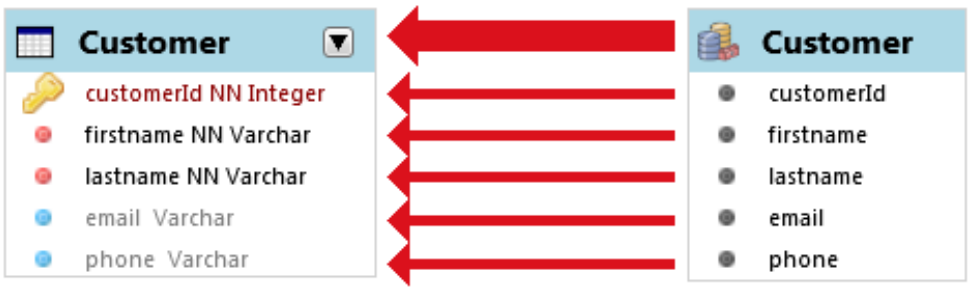


Figure 7.15: An example of a mapping between database model and query model.

The mapping does not preserve keys (primary keys, foreign keys) and any other column attribute like not-null or unique. For the purpose of the data querying this property is insignificant.

7.4.1 Mapping of Operations

In this section we describe mapping of the operations of database and query models used in the evolution process.

Definition 7.3. (Operation). An *operation* is a general function $f : (M, C) \rightarrow M'$, where M is a particular model intented to change, C represents a context describing a change of the model M and M' is a modified model.

Definition 7.4. (Atomic Opertion). An *atomic operation* is the minimal indivisible operation. It can be used to create composite operations.

All changes in the database model are done by atomic operations. All atomic operations in the database model which have an impact on the SQL query model are translated by the evolution process into corresponding atomic operations in the SQL query model (see Section 8.2).

Database Model Operations

Let us suppose a database model M_D, which is a set of tables T_i. Each table $T_i \in M_D$ has a name T_{i_N} and a set T_{i_C}, which is a set of columns c_j. Each column c_j has a name c_{j_N}.

Renaming Database Table $\alpha_T : (T_i, m) \rightarrow T_i'$ The operation returns table T_i' where $T'_{i_N} = m$ and $T'_{i_C} = T_{i_C}$.

Removing Database Table $\beta_T : (M_D, T_i) \rightarrow M'_D$ The operation removes database table $T_i \in M_D$ from the database model M_D. It returns database model M'_D where $M'_D = M_D \backslash \{T_i\}$.

Creating Table Column $\gamma_C : (T_i, c_j) \rightarrow T'_i$ The operation adds the column c_j into table T_i. It returns table T'_i where $T'_{i_N} = T_{i_N}$ and $T'_{i_C} = T_{i_C} \cup \{c_j\}$.

Renaming Table Column $\alpha_C : (c_j, m) \rightarrow c'_j$ The operation returns column c'_j where $c'_{j_N} = m$.

Removing Table Column $\beta_C : (T_i, c_j) \rightarrow T'_i$ The operation removes column $c_j \in T_i$ from the table T_i. It returns table T'_i where $T'_{i_N} = T_{i_N}$ and $T'_{i_C} = T_{i_C} \backslash \{c_j\}$.

SQL Query Model Operations

Let us suppose a query model M_Q, whose query graph G_Q consists of a set of DataSources D_i and other components, which are not important for this purpose. Each DataSource $D_i \in M_Q$ has a name D_{i_N} and a set D_{i_I}, which is a set of DataSourceItems d_j. Each DataSourceItem d_j has a name d_{j_N}.

Renaming DataSource $\alpha_D : (D_i, m) \rightarrow D'_i$ The operation returns DataSource D'_i where $D'_{i_N} = m$ and $D'_{i_I} = D_{i_I}$.

Removing DataSource $\beta_D : (M_Q, D_i) \rightarrow M'_Q$ The operation removes DataSource $D_i \in M_Q$ from the query model M_Q. It returns query model M'_Q where $M'_Q = M_Q \backslash \{D_i\}$.

Creating DataSourceItem $\gamma_I : (D_i, d_j) \rightarrow D'_i$ The operation adds DataSourceItem d_j into DataSource D_i. It returns the DataSource D'_i where $D'_{i_N} = D_{i_N}$ and $D'_{i_I} = D_{i_I} \cup \{d_j\}$.

Renaming DataSourceItem $\alpha_I : (d_j, m) \rightarrow d'_j$ The operation returns DataSourceItem d'_j where $d'_{j_N} = m$.

Removing DataSourceItem $\beta_I : (D_i, d_j) \rightarrow D'_i$ The operation removes DataSourceItem $d_j \in D_i$ from the DataSource D_i. It returns DataSource D'_i where $D'_{i_N} = D_{i_N}$ and $D'_{i_I} = D_{i_I} \backslash \{d_j\}$.

7.4.2 Complex Operations

More complex operations like *Split*, *Merge*, *Move* done in the database model can be propagated to the SQL query model as well. In the SQL query model these operations have to be composed from mentioned atomic operations.

A prototype implementation described in Section 9.2 contains only an implementation of the atomic operations.

7.5 Generating of the SQL Query

Since the SQL query model describes an SQL query, it has to be possible to generate the resulting SQL query code from the query model, i.e. from the query

graph. The resulting code has to be able to be executed on the particular DBMS. Since limitations of the query model basically correspond to the SQL-92 Standard [29], each DBMS supporting the SQL-92 Standard is suitable.

In this section we describe an algorithm for generating the correct SQL query code from the particular query graph.

7.5.1 Order of SQL Query Generating

The query model diagram can describe more than one SQL query. It is common that any modeled queries depend on other modeled queries, for instance the query is used as a DataSource in the FROM clause of other query, or the operators IN and EXISTS have the query as an argument. For this reason the queries used as a part of the another query have to be generated before this query. Obviously, the circular dependency of the queries is banned. Such query could not be executed and evaluated by any DBMS. This observation causes the generated queries have to be generated in a topological order.

In the following described algorithms we use the following auxiliary functions:

- *VertexType(V)*, where V is a vertex of the query graph. The function returns the type of the vertex V.
- *EdgeType(E)*, where E is an edge of the query graph. The function returns the type of the edge E.
- *V.GetNeighbour(type)*, where V is a vertex of the query graph and *type* is the type of the edge. The function returns first adjacent vertex of the vertex V connected by the edge of the type *type*.
- *V.GetNeighbours(type)*, where V is a vertex of the query graph and *type* is the type of the edge. The function returns all adjacent vertices of the vertex V connected by the edge of the type *type*.
- *V.GetNeighbourOfType(type)*, where V is a vertex of the query graph and *type* is the type of the vertex. The function returns first adjacent vertex of the vertex V of the type *type*.
- *V.GetNeighboursOfType(type)*, where V is a vertex of the query graph and *type* is the type of the vertex. The function returns all adjacent vertices of the vertex V of the type *type*.
- *V.GetSourceVertex(type)*, where V is a vertex of the query graph and *type* is the type of the edge. The function returns adjacent vertex from which leads the edge of the type *type* to the vertex V.
- *GetCodeForQuery(SelectVertex)*. This function returns for the given Select vertex the already generated SQL query. Obviously, this function is called by the dependant queries to complete the query.
- *StoreSQLCode(SQL Code)*. This functions stores the already generated SQL query to be fetched later by the previous function.

- *GetResultingSQLCode()*. This function returns all the generated SQL queries concatenated together.

Definition 7.5. (Inverse Graph). An inverse graph of the graph $G = (V, E)$ is a graph $G' = (V, E')$ such that if $(v_1, v_2) \in E \Rightarrow (v_2, v_1) \in E'$.

Algorithm 7.1 generates the topological order of the generated queries in the particular query model. At first, it creates a dependency graph from the query graph by Algorithm 7.3. Then it finds independent vertices in the dependeny graph. Independent vertex is a vertex, which has no adjacent vertices. Then depth-first-search (DFS) Algorithm 7.2 is used on each independent vertex in an inverse graph of the dependency graph. This algorithm at first recursively visits neighbours of an input vertex and finally it adds the input vertex to the resulting topological order. If Algorithm 7.1 finishes, the resulting order contains vertices of the dependency graph in the topological order.

Algorithm 7.1 GetTopologicalOrderOfQueries

Input: Query Graph G_Q
Output: List of vertices ordered by topological order, where each vertex represents one query

```
1: roots ← list of vertices
2: DG_Q ← CreateDependencyGraph(G_Q) {see Algorithm 7.3 }
3: resultOrder ← resulting list of vertices
4: for all vertex ∈ DG_Q.Vertices do
5:    if vertex.GetNeighbours().Count = 0 then
6:       roots.Add(vertex)
7:    end if
8: end for
9: DG_Q.Inverse() { creates an inverse graph }
10: DG_Q.InitVisit() { sets to each vertex state READY }
11: for all vertex ∈ roots do
12:    DoTopologicalOrder(DG_Q, vertex, resultOrder) {see Algorithm 7.2 }
13: end for
14: return resultOrder
```

Algorithm 7.2 DoTopologicalOrder

Input: Dependency Graph DG_Q, current *vertex*, resulting *order*
Output: *vertex* added to *order* in topological order

```
1: vertex.State ← OPENED
2: for all neighbour ∈ DG_Q.GetNeighbours(vertex) do
3:    if neighbour.State = READY then
4:       DoTopologicalOrder(DG_Q, neighbour, order)
5:    else if neighbour.State = OPENED then
6:       Error - cycle found.
7:    end if
8: end for
9: order.Add(vertex)
10: vertex.State ← CLOSED
```

Algorithm 7.3 creates a special graph useable to be topologically ordered. It consists of Dependency vertices and edges. The Dependency vertex contains a pair of From vertex and the corresponding Select vertex. This pair represents a whole SQL query.

Algorithm 7.3 CreateDependencyGraph

Input: Query Graph G_Q
Output: Dependency graph of queries in the query graph

```
1: DG_Q ← emptygraph
2: for all vertex such that VertexType(vertex) = From do
3:    selectVertex ← vertex.GetSelect() {finds corresponding select vertex}
4:    DG_Q.AddVertex(DependencyVertex(vertex, selectVertex))
5: end for
6: for all fromVertex ∈ DG_Q.Vertices do
7:    for all selectVertex ∈ DG_Q.Vertices do
8:       if DG_Q.Depends(fromVertex.From, selectVertex.Select) then
9:          DG_Q.AddEdge(selectVertex, fromVertex, Dependency)
10:      end if
11:   end for
12:   for all selectVertex ∈ fromVertex.GetConditionQueryDependecies()
      do
13:      vertex ← DG_Q.GetDependencyVertex(selectVertex) {finds dependen-
         cy vertex containing given select vertex }
14:      DG_Q.AddEdge(vertex, fromVertex, Dependency)
15:   end for
16: end for
17: return DG_Q
```

7.5.2 Generating SQL Algorithm

Algorithm 7.4 goes in the topological order through the list of queries of the query model and for each query it generates the resulting SQL code, which describes Algorithm 7.5.

Algorithm 7.4 GenerateSQL

Input: Query Graph G_Q
Output: SQL code of the queries defined by the query graph.

```
1: queries ← GetTopologicalOrderOfQueries(G_Q) {returns queries of the
   query graph in a topological order (see Algorithm 7.1) }
2: for all queryVertex ∈ queries do
3:    queryCode ← GenerateSQLOfQuery(queryVertex) {see Algorithm 7.5}

4:    StoreSQLCode(queryCode)
5: end for
6: return GetResultingSQLCode()
```

Algorithm 7.5 generates a particular SQL query. It generates the SELECT and the FROM clauses. Then checks, whether another clauses exist (i.e. WHERE,

HAVING, GROUP-BY and ORDER-BY). If does, they are generated as well.

Algorithm 7.5 GenerateSQLOfQuery

Input: QueryVertex V
Output: SQL code of the query represented by the vertex V.

```
1: fromVertex ← V.From
2: selectVertex ← V.Select
3: code ← empty string
4: code ← GenerateSelectClause(selectVertex) {see Algorithm 7.6}
5: code ← GenerateFromClause(fromVertex) {see Algorithm 7.7}
6: whereVertex = fromVertex.GetNeighbour(WHERE)
7: if whereVertex != null then
8:    code ← GenerateConditionClause(whereVertex) {see Algorithm 7.11}
9: end if
10: groupByVertex = selectVertex.GetNeighbour(GROUPBY)
11: if groupByVertex != null then
12:    code ← GenerateGroupByClause(groupByVertex) {see Algorithm 7.9}
13: end if
14: havingVertex = fromVertex.GetNeighbour(HAVING)
15: if havingVertex != null then
16:    code ← GenerateConditionClause(havingVertex) {see Algorithm 7.11}
17: end if
18: orderByVertex = fromVertex.GetNeighbour(ORDERBY)
19: if orderByVertex != null then
20:    code ← GenerateOrderByClause(orderByVertex) {see Algorithm 7.10}
21: end if
22: return code
```

Algorithm 7.6 generates the content of the SELECT clause of the SQL query. It generates the 'SELECT' keyword and it generates for each adjacent SelectItem vertex the resulting code of the query.

Algorithm 7.6 GenerateSelectClause

Input: SelectVertex S
Output: SQL code of the SELECT clause represented by the vertex S.

```
1: code ← SELECT
2: for all itemVertex ∈ S.NeighboursOfType(SelectItemVertex) do
3:    code ← itemVertex.SQLCode
4: end for
5: return code
```

Algorithm 7.7 generates the content of the FROM clause of the SQL query. It generates the 'FROM' keyword and it uses Algorithm 7.8 in order to generate the remaining content.

Recursive Algorithm 7.8 traverses down the *combination source tree* by the DFS algorithm. If the algorithm goes to the leaf node of the tree (i.e. Alias vertex), it generates the name of the DataSource. Otherwise it generates a code for the left subtree, then type of the connection, subsequently the right subtree and finally it possibly generates a join condition.

Algorithm 7.7 GenerateFromClause

Input: FromVertex F

Output: SQL code of the FROM clause represented by the vertex F.

1: *code* ← FROM
2: *code* ← $GenerateFromClauseRecursive(F.MainSource)$ {see Algorithm 7.8 }
3: **return** *code*

Algorithm 7.8 GenerateFromClauseRecursive

Input: Vertex V

Output: SQL code of the part of the FROM clause represented by the vertex V.

1: *code* ← empty string
2: **if** V is Alias vertex **then**
3: $dataSource = V.GetNeighbour(DATASOURCE)$
4: **if** *dataSource* is DataSource vertex **then**
5: $source \leftarrow dataSource.Name$
6: **else**
7: $source \leftarrow GetCodeForQuery(dataSource)$
8: **end if**
9: *code* ← *source* as *alias*
10: **else if** V is CombineSource vertex **then**
11: $source0 \leftarrow V.GetNeighbour(MapSource[0])$
12: $source1 \leftarrow V.GetNeighbour(MapSource[1])$
13: $code \leftarrow GenerateFromClauseRecursive(source0)$
14: *code* ← type of the connection
15: $code \leftarrow GenerateFromClauseRecursive(source1)$
16: $code \leftarrow GenerateConditionClause(V)$ {see Algorithm 7.11}
17: **end if**
18: **return** *code*

Algorithm 7.9 generates the content of the GROUP-BY clause of the SQL query. It generates the 'GROUP BY' keyword and it generates for each adjacent SelectItem vertex the resulting code of the query.

Algorithm 7.9 GenerateGroupByClause

Input: GroupBy vertex G
Output: SQL code of the GROUP-BY clause represented by the vertex G.

```
1: code ← GROUP BY
2: for all itemVertex ∈ G.NeighboursOfType(SelectItemVertex) do
3:     code ← itemVertex.SQLCode
4: end for
5: return code
```

Algorithm 7.10 generates the content of the ORDER-BY clause of the SQL query. It generates the 'ORDER BY' keyword and it generates for each adjacent OrderByType vertex the resulting code of the query.

Algorithm 7.10 GenerateOrderByClause

Input: OrderByVertex O
Output: SQL code of the ORDER-BY clause represented by the vertex O.

```
1: code ← ORDER BY
2: for all orderByTypeVertex ∈ O.NeighboursOfType(OrderByTypeVertex)
   do
3:     code ← orderByTypeVertex.SQLCode
4: end for
5: return code
```

Algorithm 7.11 generates the content of the particular condition clause of the SQL query. It generates the keyword according to the type of condition and it uses Algorithm 7.12 in order to generate the remaining content of the condition.

Algorithm 7.11 GenerateConditionClause

Input: Condition vertex C
Output: SQL code of the condition represented by the vertex C.

```
1: code ← condition key word related to the type of C {WHERE, HAVING,
   ON}
2: conditionRoot ← C.GetNeighbour(CONDITION)
3: if conditionRoot != null then
4:     code ← GenerateConditionRecursive(conditionRoot) {see Algorithm 7.12
       }
5: end if
6: return code
```

Algorithm 7.12 recursively generates the condition from the *condition tree*.

- For the FromItem vertex it generates the name of the FromItem.
- For the ConstantOperand vertex it generates its value directly.

- For the Select vertex it generates the code of the whole SQL query represented by the Select vertex. Since the queries are generated in the topological order, this query has to be already generated.
- For the ComparingOperator vertex or the QueryOperator vertex it generates a first operand recursively, then it generates a corresponding operator and finally it generates a second operand recursively.
- For the BooleanOperator vertex it generates all child conditions recursively.

Algorithm 7.12 GenerateConditionRecursive

Input: Vertex V
Output: SQL code of the part of the condition represented by the vertex V.

```
1: if V is FromItem vertex then
2:    code ← V.Name
3: else if V is ConstantOperand vertex then
4:    code ← V.Value
5: else if V is Select vertex then
6:    code ← GetCodeForQuery(V)
7: else if V is ComparingOperator vertex then
8:    code ← GenerateConditionRecursive(V.GetNeighbour(ConditionOperand[0]))
9:    code ← Comparing operator
10:   if comparing operator ∉ {Is Null, Is Not Null} then
11:      code ← GenerateConditionRecursive(V.GetNeighbour(ConditionOperand[1]))
12:   end if
13: else if V is QueryOperator vertex then
14:   if query operator ∉ {Exists, Not Exists} then
15:      code ← GenerateConditionRecursive(V.GetNeighbour(ConditionOperand[1]))
16:   end if
17:   code ← Query operator
18:   code ← GenerateConditionRecursive(V.GetNeighbour(ConditionOperand[0]))
19: else if V is BooleanOperator then
20:   conditions ← V.GetNeighbours(ConditionOperand)
21:   for i = 0 to conditions.Count do
22:      if i > 0 then
23:         code ← Boolean operator
24:      end if
25:      code ← GenerateConditionRecursive(conditions[i])
26:   end for
27: end if
28: return code
```

Chapter 8

Change Propagation

In this chapter we discuss the impact of changes in the database model on the queries in the SQL query model. We introduce propagation policies, which influence the change propagation and we describe algorithms, which distribute changes accross the query graph.

8.1 Propagation Policies

Database model operations mentioned in Section 7.4.1 have an impact on the queries in the SQL query model. During the evolution process, changes in the database model have to be propagated to the SQL query model, where modeled queries are adapted to the current database model. Sometimes the direct propagation cannot be profitable. For instance, if a new column is added to the database table, we do not want to add new column to the SELECT clause of the query. For this reason we propose so called *propagation policies*, which influence a behavior of the propagation. The policies are defined on the vertices of the query graph, which participate on the change distribution process described in the section 8.2.

We distinguish the following propagation policies:

- **Propagate**
  This policy allows to perform the change directly. Subsequently, the propagation is passed on the following vertices in the change process.
- **Block**
  This policy does not allow to perform the change. The subsequent propagation is stopped and the following vertices in the change process are not visited.
- **Prompt**
  A system asks user which one of the two policies mentioned before is used to continue.

8.2 Distribution of Changes

In this section we describe operations which modify the query graph. Discusion on an impact of changes of the database model on the query model is provided

as well.

8.2.1 Graph Operations

SQL query model operations mentioned in Section 7.4.1 are atomic operations, i.e. they cannot be divided into smaller operations. In fact, these atomic operations consist of many smaller steps called *graph operations*, which modify the query graph of the SQL query model. From the global perspective, graph operations do not make any sense, if they are used separately. They have to be combined together to create a real atomic operation, which has a sense in the context of change propagation.

In the following definitions G_Q represents a query graph $G_Q = (V, E)$. We distinguish the following graph operations:

- *CreateVertex* $\gamma_v : (G_Q, v) \rightarrow G'_Q$
  The operation returns graph $G'_Q = (V \cup \{v\}, E)$.

- *CreateEdge* $\gamma_e : (G_Q, v_{source}, v_{target}, e_{type}) \rightarrow G'_Q$
  The operation creates edge $e = (v_{source}, v_{target})$ such that $EdgeType(e) = e_{type}$ and returns graph $G'_Q = (V, E \cup \{e\})$.

- *RemoveVertex* $\beta_v : (G_Q, v) \rightarrow G'_Q$
  The operation returns graph $G'_Q = (V \backslash \{v\}, E \cap \binom{V \backslash \{v\}}{2})$.

- *RemoveEdge* $\beta_e : (G_Q, e) \rightarrow G'_Q$
  The operation returns graph $G'_Q = (V, E \backslash \{e\})$.

- *ChangeLabel* $\lambda : (v, l) \rightarrow v'$
  The operation returns query vertex v', where vertex type $v'_{type} = v_{type}$ and label $v'_L = l$. For instance it returns the DataSourceItem vertex with a new name.

- *ChangeConnectionType* $\eta : (C, t) \rightarrow C'$
  This operation returns CombineSource vertex C' with connection type $C'_T = t$.

- *ResetContent* $\rho(G_Q, C)$
  Since the visualisation model visualizes the query graph, they have to be synchronized. This operation is used to signal the parent visual component C that a change in the query graph G_Q has been done and the content of the visual component has to be updated.

Definition 8.1. (Graph operation plan). An *operation plan* is a sequence of the graph operations.

8.2.2 Traversing through Query Graph

Each database model operation which has an impact on the query graph triggers an event, in which the query graph is traversed through in a particular way. The change is fully described by so called *change context*. Each change context contains a reference to the query graph G_Q being changed and an empty *graph*

operation plan, which is filled in the traversing algorithm by the graph operations, which changes the query graph according to the triggered event. After the traversing algorithm is done, the graph operation plan is executed as a part of the particular atomic operation.

Each visit of the query graph vertex with the traversing algorithm performs a check whether the propagation policy allows the given change propagation. All the following traversing algorithms suppose the propagation is allowed.

In the following described algorithms we use the following auxiliary functions:

- *ResetAllContent(FromVertex, OperationPlan, QueryGraph)*. This function adds to the operation plan ResetContent operations to reset content of all the components used in the particular query represented by From vertex.
- *RemoveVertexComplete(Vertex, OperationPlan, QueryGraph)*. This function adds to the operation plan operations needed to complete removing of the given vertex, which includes RemoveVertex and RemoveEdge operations.

8.2.3 Creating Table Column

This database model operation is translated into the *creating DataSourceItem* operation. Each query using the parent DataSource of the new DataSourceItem has to add a new item to the corresponding components, which policy allows the propagation. This means, new FromItem vertex has to be created and subsequently Select vertex and OrderBy vertex have to decide, whether the new FromItem vertex is going to be used in these clauses.

Algorithm 8.1 starts at the DataSource vertex, where it creates a new DataSourceItem vertex. Then it traverses through all Alias vertices to corresponding From vertices, where it creates a new FromItem vertex. This step is described by Algorithm 8.2.

Algorithm 8.1 DistributeCreatingDatasourceItem

Input: DataSource vertex D, change context C
Output: Graph operations to create a new DataSourceItem.

1: $newItem \leftarrow$ new $DataSourceItem(C.Name)$
2: $C.Plan \leftarrow CreateVertex(C.G_Q, newItem)$
3: $C.Plan \leftarrow CreateEdge(C.G_Q, D, newItem, DataSourceItem)$
4: $C.Plan \leftarrow CreateEdge(C.G_Q, newItem, D, DataSourceParent)$
5: $C.Originator \leftarrow newItem$
6: **for all** $aliasVertex \in D.GetNeighbours(Alias)$ **do**
7: $DistributeCreatingDatasourceItemOnAlias(aliasVertex, C)$ {see Algorithm 8.2}
8: **end for**

Algorithm 8.2 creates a new FromItem vertex in the corresponding From vertex and connects it with appropriate vertices. Subsequently it traverses to the

Select vertex, where it creates corresponding vertices and edges using Algorithm 8.3. Finally it traverses to the OrderBy vertex, where it creates corresponding vertices and edges using Algorithm 8.4.

Figure 8.1 depicts adding of a new DataSourceItem *OrderDate* to the DataSource component using Algorithm 8.1 and to the From component using Algorithm 8.2. In the figure new elements of the query graph are highlighted with a red color.

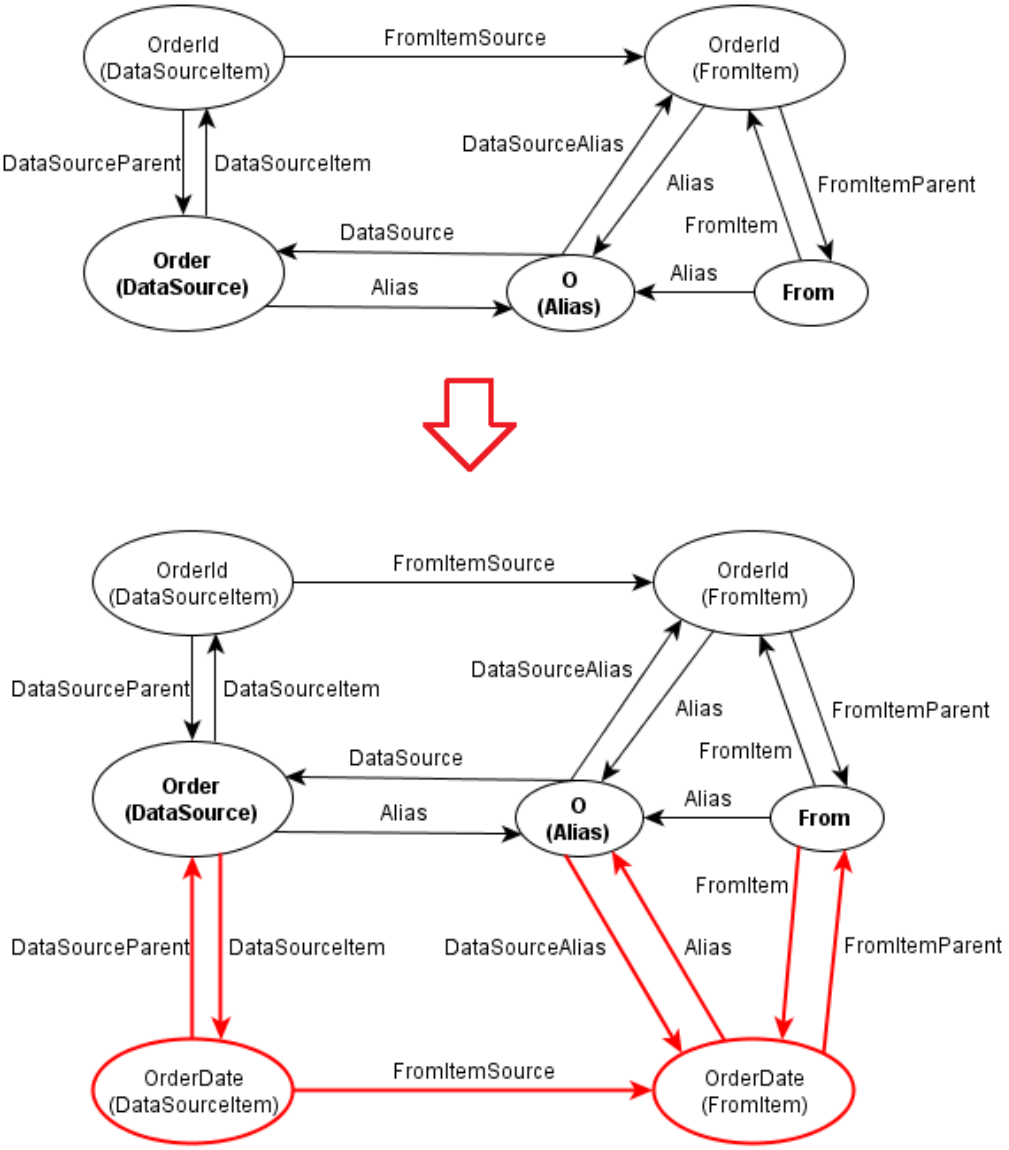


Figure 8.1: An example of adding new DataSourceItem to the DataSource and to the From components.

Algorithm 8.3 creates a new SelectItem vertex in the SELECT clause. Then it checks, whether the GroupBy vertex exists. If does, it connects the GroupBy vertex with the new SelectItem vertex. Finally it traverses to all Alias vertices of dependant queries and applies already mentioned Algorithm 8.2.

Figure 8.2 depicts a propagation of adding of a new SelectItem *Name* to the dependant query using Algorithm 8.3. In the figure new elements of the query

Algorithm 8.2 DistributeCreatingDatasourceItemOnAlias

Input: Alias vertex A, change context C

Output: Graph operations to create new DataSourceItem.

1: $fromVertex \leftarrow A.GetNeighbour(DataSourceAlias)$
2: $newItem \leftarrow$ new $FromItem(C.Name)$
3: $C.Plan \leftarrow CreateVertex(C.G_Q, newItem)$
4: $C.Plan \leftarrow CreateEdge(C.G_Q, C.Originator, newItem, FromItemSource)$
5: $C.Plan \leftarrow CreateEdge(C.G_Q, newItem, A, Alias)$
6: $C.Plan \leftarrow CreateEdge(C.G_Q, fromVertex, newItem, FromItem)$
7: $C.Plan \leftarrow CreateEdge(C.G_Q, newItem, fromVertex, FromItemParent)$
8: $C.Originator \leftarrow newItem$
9: $selectVertex \leftarrow fromVertex.GetNeighbour(SelectQuery)$
10: **if** $selectVertex$!= $null$ **then**
11: $DistributeCreatingDatasourceItemOnSelect(selectVertex, C)$ {see Algorithm 8.3}
12: **end if**
13: $orderByVertex \leftarrow fromVertex.GetNeighbour(OrderBy)$
14: **if** $orderByVertex$!= $null$ **then**
15: $DistributeCreatingDatasourceItemOnOrderBy(orderByVertex, C)$ {see Algorithm 8.4}
16: **end if**
17: $C.Plan \leftarrow ResetContent(C.G_Q, fromVertex)$

graph are highlighted with a red color. The elements highlighted with a blue color were added to the query graph by previous step of the traversing algorithm.

Algorithm 8.4 creates a new OrderByType vertex and connects it with the OrderByVertex and the source FromItem vertex.

8.2.4 Renaming Table Column

This database model operation is translated into the *renaming DataSourceItem* operation. Each query using the parent DataSource has to be translated into the new name, i.e. each FromItem vertex has to be changed and subsequently each other component using the given FromItem vertex has to be updated. Since there can exist queries, which depend on this query, these have to be updated as well.

Algorithm 8.5 starts at DataSourceItem vertex, where it changes the name of the DataSourceItem. Then it traverses to all FromItem vertices, where Algorithm 8.6 sets the new name of the FromItem vertex.

Algorithm 8.6 changes the name of the FromItem vertex. Then it traverses through the From vertex to the Select vertex.

Algorithm 8.7 finds a SelectItem vertex corresponding to the changed FromItem vertex. Then it checks, whether a name of the SelectItem vertex has to be changed. If does, it continues with traversing to all adjacent FromItem vertices of the dependant queries, where it applies already mentioned Algorithm 8.6.

Figure 8.3 depicts the process of a distribution of a table column name change. In the figure elements affected by Algorithm 8.5 are highlighted with a red color,

Algorithm 8.3 DistributeCreatingDatasourceItemOnSelect

Input: Select vertex S, change context C

Output: Graph operations to create new DataSourceItem.

1: $newItem \leftarrow$ new $SelectItem(C.Name)$
2: $C.Plan \leftarrow CreateVertex(C.G_Q, newItem)$
3: $C.Plan \leftarrow CreateEdge(C.G_Q, S, newItem, SelectColumn)$
4: $C.Plan \leftarrow CreateEdge(C.G_Q, newItem, C.Originator, MapColumn)$
5: $groupByVertex \leftarrow S.GetNeighbour(GroupBy)$
6: **if** $groupByVertex$!= $null$ **then**
7: $\quad C.Plan \leftarrow CreateEdge(C.G_Q, groupByVertex, newItem, GroupByColumn)$
8: **end if**
9: $C.Originator \leftarrow newItem$
10: **for all** $aliasVertex \in S.GetNeighbours(Alias)$ **do**
11: $\quad DistributeCreatingDatasourceItemOnAlias(aliasVertex, C)$
12: **end for**
13: $C.Plan \leftarrow ResetContent(C.G_Q, S)$

Algorithm 8.4 DistributeCreatingDatasourceItemOnOrderBy

Input: OrderBy vertex O, change context C

Output: Graph operations to create new DataSourceItem.

1: $newItem \leftarrow$ new $OrderByType(Ascending)$
2: $C.Plan \leftarrow CreateVertex(C.G_Q, newItem)$
3: $C.Plan \leftarrow CreateEdge(C.G_Q, O, newItem, OrderByColumn)$
4: $C.Plan \leftarrow CreateEdge(C.G_Q, newItem, C.Originator, OrderByColumn)$
5: $C.Plan \leftarrow ResetContent(C.G_Q, O)$

Algorithm 8.5 DistributeRenamingDatasourceItem

Input: DataSourceItem vertex D, change context C

Output: Graph operations to change the DataSourceItem name.

1: $C.Plan \leftarrow ChangeLabelOperation(D, C.NewName)$
2: **for all** $fromItemVertex \in D.GetNeighbours(FromItemSource)$ **do**
3: $\quad DistributeRenamingDatasourceItemOnFromItem(fromItemVertex, C)$ {see Algorithm 8.6}
4: **end for**

Algorithm 8.6 DistributeRenamingDatasourceItemOnFromItem

Input: FromItem vertex F, change context C

Output: Graph operations to change the name of the column represented by the FromItem vertex.

1: $C.Plan \leftarrow ChangeLabelOperation(F, C.NewName)$
2: $fromVertex \leftarrow F.GetNeighbour(FromItemParent)$
3: $selectVertex \leftarrow fromVertex.GetNeighbour(SelectQuery)$
4: $C.Originator \leftarrow F$
5: $DistributeRenamingDatasourceItemOnSelect(selectVertex, C)$ {see Algorithm 8.7}
6: $ResetAllContent(fromVertex, C.Plan, C.G_Q)$

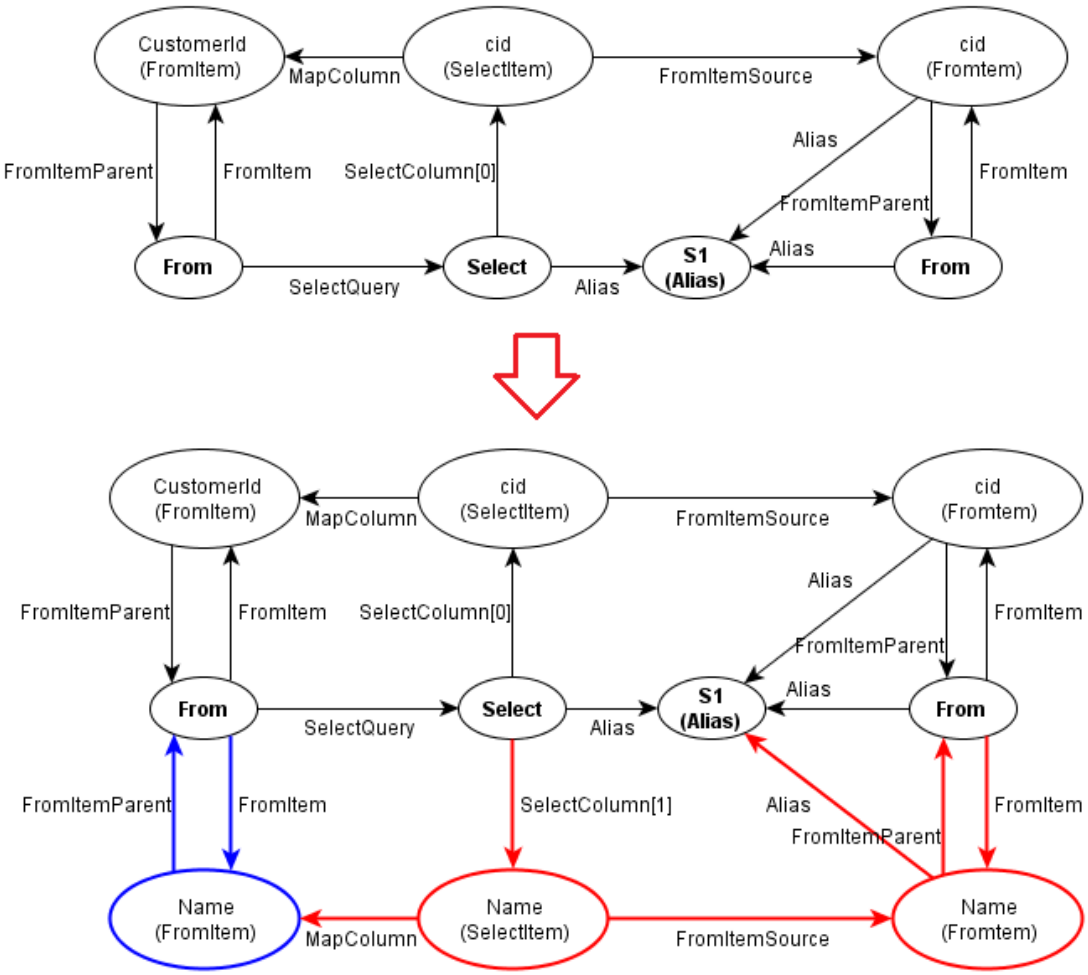


Figure 8.2: An example of a propagation of adding of a new SelectItem to a dependant query.

the elements affected by Algorithm 8.6 are highlighted with a blue color and the elements affected by Algorithm 8.7 are highlighted with a green color.

8.2.5 Renaming Database Table

This database model operation is translated into the *renaming DataSource* operation. Each query using the given DataSource has to be checked, whether the change of its name has an impact on the query, i.e. whether the original DataSource name is used directly as an alias of this DataSource in the query. In such case all columns used in the query has to be rewritten by using new alias refering to the new DataSource name.

Algorithm 8.8 changes the name of the DataSource vertex and then checks each adjacent Alias vertex, whether the used alias of the DataSource has to be changed. If does, it changes the given alias.

Algorithm 8.7 DistributeRenamingDatasourceItemOnSelect

Input: Select vertex S, change context C

Output: Graph operations to change the name of the column occuring in the SELECT clause.

1: $selectItemVertex \leftarrow GetCorrespondingSelectedItem(S, C.Originator)$ {returns the SelectItem vertex refering to the originator, i.e. mapped to the FromItem vertex}
2: **if** $selectItemVertex$ has not an alias **then**
3: $C.Originator \leftarrow selectItemVertex$
4: $C.NewName \leftarrow selectItemVertex.Name$
5: **for all** $fromItemVertex \in selectItemVertex.GetNeighbours(FromItemSource)$ **do**
6: $DistributeRenamingDatasourceItemOnFromItem(fromItemVertex, C)$
7: **end for**
8: **end if**

Algorithm 8.8 DistributeRenamingDatasource

Input: DataSource vertex D, change context C

Output: Graph operations to change the DataSource name.

1: $C.Plan \leftarrow ChangeLabelOperation(D, C.NewName)$
2: **for all** $aliasVertex \in D.NeighboursOfType(AliasVertex)$ **do**
3: **if** *alias has to be changed* **then**
4: $C.Plan \leftarrow ChangeLabelOperation(aliasVertex, C.NewName)$
5: $fromVertex \leftarrow aliasVertex.GetNeighbour(DataSourceAlias)$
6: $ResetAllContent(fromVertex, C.Plan, C.G_Q)$
7: **end if**
8: **end for**

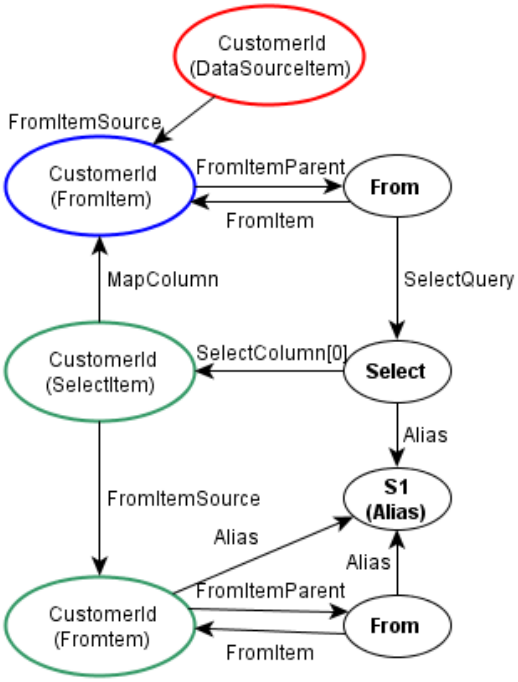


Figure 8.3: An example of distribution of a table column name change.

8.2.6 Removing Table Column

This database model operation is translated into the *removing DataSourceItem* operation. This operation is the most complex one. The propagation of this change concerns all the components of all queries in the query model. It includes removing of the particular DataSourceItem vertex and all the corresponding FromItem vertices, which continues with removing corresponding vertices and edges mapped to the particular FromItem vertex.

Figure 8.4 shows an example of removing of a DataSourceItem *Name* from a DataSource *Customer*. In the figure all elements highlighted with a red color are being removed.

Algorithm 8.9 starts at DataSourceItem vertex, where it completely removes the vertex including all the edges. Then it traverses to all the FromItem vertices.

Algorithm 8.9 DistributeRemovingDatasourceItem

Input: DataSourceItem vertex D, change context C
Output: Graph operations to remove the DataSourceItem.

1: **for all** $fromItemVertex \in D.NeighboursOfType(FromItemSource)$ **do**
2: $\quad DistributeRemovingFromItem(fromItemVertex, C)$
3: **end for**
4: $RemoveVertexComplete(D, C.Plan, C.G_Q)$ {see Algorithm 8.10}

Algorithm 8.10 completely removes the given FromItem vertex and through the parent From vertex it traverses to all corresponding components, i.g. Select, Where, Having, OrderBy. Similarly, the vertex has to be removed from all

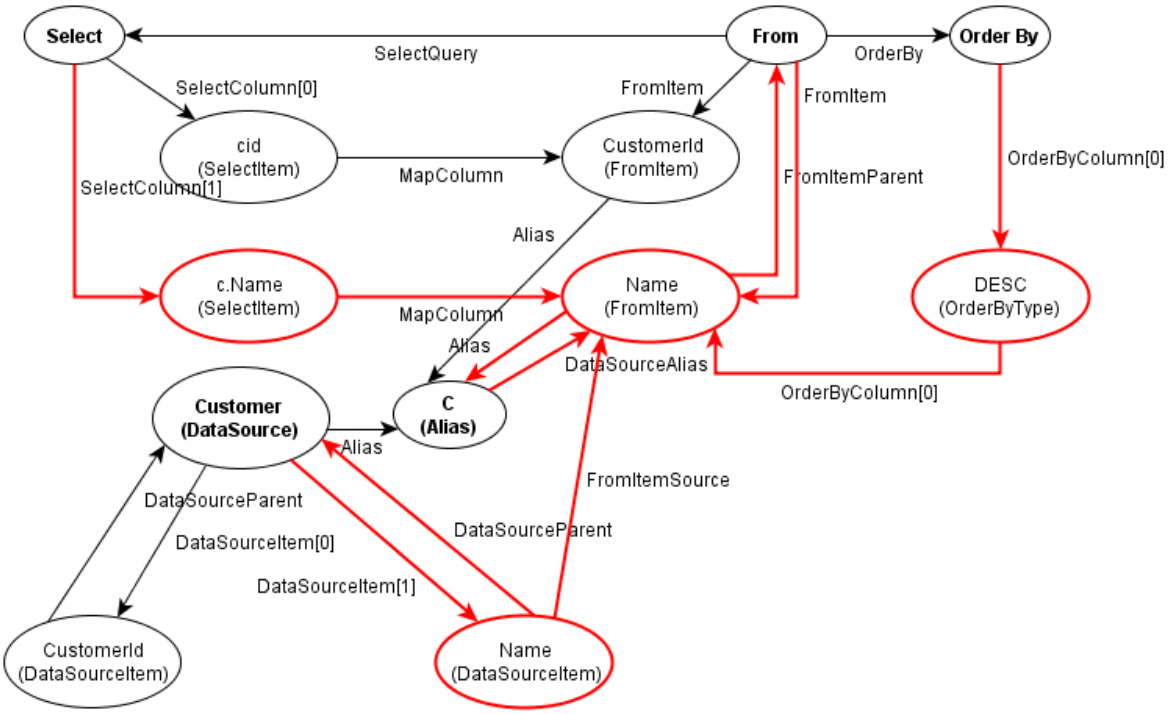


Figure 8.4: An example of affected elements of a query graph because of the DataSourceItem being removed.

conditions in the *combination source tree*. This step is performed by Algorithm 8.11.

Algorithm 8.11 traverses the combination source tree starting at the root node towards the leaf nodes, and at each level it reorganizes the used condition. If the whole condition has to be removed, the type of connection is changed to the Cartesian product.

Algorithm 8.12 removes all SelectItem vertices corresponding to the FromItem vertex being removed. Then it continues with traversing to all adjacent FromItem vertices of the dependant queries, where it applies already mentioned Algorithm 8.10.

Algorithm 8.13 removes the FromItem vertex from the *condition tree* using Algorithm 8.14. If it is necessary, it changes the root of the condition tree.

Algorithm 8.14 recursively traverses the condition tree and according to the condition tree level it checks if the condition has to be removed, because the FromItem vertex (originator of *ChangeContext*) is used in the condition:

- At BooleanOperator vertex level: Algorithm 8.15 checks all child conditions (i.e. clauses or literals). If all the child conditions have to be removed, the BooleanOperator vertex has to be removed as well. If only one child condition is remaining, the BooleanOperator vertex has to be removed as well, but at the higher level of the condition tree the root of the subtree has to be changed to the remaining child condition.

- At the QueryOperator or ComparingOperator vertex level: It checks if any of its child conditions has to be removed. If does, the operator vertex has to be removed as well.

Algorithm 8.10 DistributeRemovingFromItem

Input: FromItem vertex F, change context C

Output: Graph operations to remove the FromItem vertex.

1: $C.Originator \leftarrow F$
2: $fromVertex \leftarrow F.GetNeighbour(FromItemParent)$
3: $selectVertex \leftarrow fromVertex.GetNeighbour(SelectQuery)$
4: **if** $selectVertex$!= $null$ **then**
5: $DistributeRemovingFromItemOnSelect(selectVertex, C)$ {see Algorithm 8.12}
6: **end if**
7: $whereVertex \leftarrow fromVertex.GetNeighbour(Where)$
8: **if** $whereVertex$!= $null$ **then**
9: $ReorganizeCondition(whereVertex, C)$ {see Algorithm 8.13}
10: **end if**
11: $havingVertex \leftarrow fromVertex.GetNeighbour(Having)$
12: **if** $havingVertex$!= $null$ **then**
13: $ReorganizeCondition(havingVertex, C)$
14: **end if**
15: $orderByVertex \leftarrow fromVertex.GetNeighbour(OrderBy)$
16: **if** $orderByVertex$!= $null$ **then**
17: $orderType \leftarrow OrderByType$ vertex corresponding to F
18: **if** $orderType$!= $null$ **then**
19: $RemoveVertexComplete(orderType, C.Plan, C.G_Q)$
20: **end if**
21: **end if**
22: $ReorderFromTree(fromVertex, C)$ {see Algorithm 8.11}
23: $RemoveVertexComplete(F, C.Plan, C.G_Q)$
24: $ResetAllContent(fromVertex, C.Plan, C.G_Q)$

Algorithm 8.11 ReorderFromTree

Input: From vertex F, change context C

Output: Graph operations to remove FromItem vertex from the *combination source tree*.

1: $sourceVertex \leftarrow F.GetNeighbour(SourceTree)$
2: **while** $VertexType(sourceVertex)! = Alias$ **do**
3: $complete \leftarrow ReorganizeCondition(sourceVertex, C)$
4: **if** *complete* is true **then**
5: $C.Plan \leftarrow ChangeConnectionType(sourceVertex, CartesianProduct)$
6: **end if**
7: $sourceVertex \leftarrow sourceVertex.GetNeighbour(MapSource[0])$
8: **end while**

Algorithm 8.12 DistributeRemovingFromItemOnSelect

Input: Select vertex S, change context C

Output: Graph operations to remove the FromItem vertex corresponding to SelectItem vertices of the Select vertex S.

1: **for all** $selectItemVertex \in S.GetNeighbours(SelectColumn)$ such that $selectItemVertex$ corresponds to $C.Originator$ **do**
2: $C.Originator \leftarrow selectItemVertex$
3: **for all** $fromItemVertex \in selectItemVertex.NeighboursOfType(FromItemSource)$ **do**
4: $DistributeRemovingFromItem(fromItemVertex, C)$
5: **end for**
6: $RemoveVertexComplete(selectItemVertex, C.Plan, C.G_Q)$ {including the AggregateFunction vertex and the edge from GroupBy component}
7: **end for**

Algorithm 8.13 ReorganizeCondition

Input: Condition vertex V, change context C

Output: Graph operations to reorganize the condition and a flag indicating that the whole condition tree was removed.

1: $conditionVertex \leftarrow V.GetNeighbour(Condition)$
2: $result \leftarrow RemoveFromItemFromCondition(conditionVertex, C)$ {see Algorithm 8.14}
3: **if** $result.Type \in \{Adjust, Remove\}$ **then**
4: $RemoveVertexComplete(conditionVertex, C.Plan, C.G_Q)$
5: **if** $result.Type \in \{Adjust\}$ **then**
6: $C.Plan \leftarrow CreateEdge(C.G_Q, V, result.NewRoot, Condition)$
7: **end if**
8: $C.ConditionTreeRemoved \leftarrow result.Type = Remove$
9: **end if**

- At the FromItem or AggregateFunction vertex level: It check, whether the vertex corresponds to the the vertex being removed.
- Any other case does not cause any change.

Figure 8.5 depicts a condition reorganization using Algorithm 8.13, because the FromItem vertex *orderDate* is being removed. In the figure elements highlighted with a red color are being removed because a given condition subtree contains the removing FromItem vertex. The BooleanOperator vertex *AND* and the remaining edge highlighted with a blue color are removed consequently, because only one subtree of the BooleanOperator vertex remains. The root of the remaining subtree highlighted with a green color will become a new root of the whole condition tree.

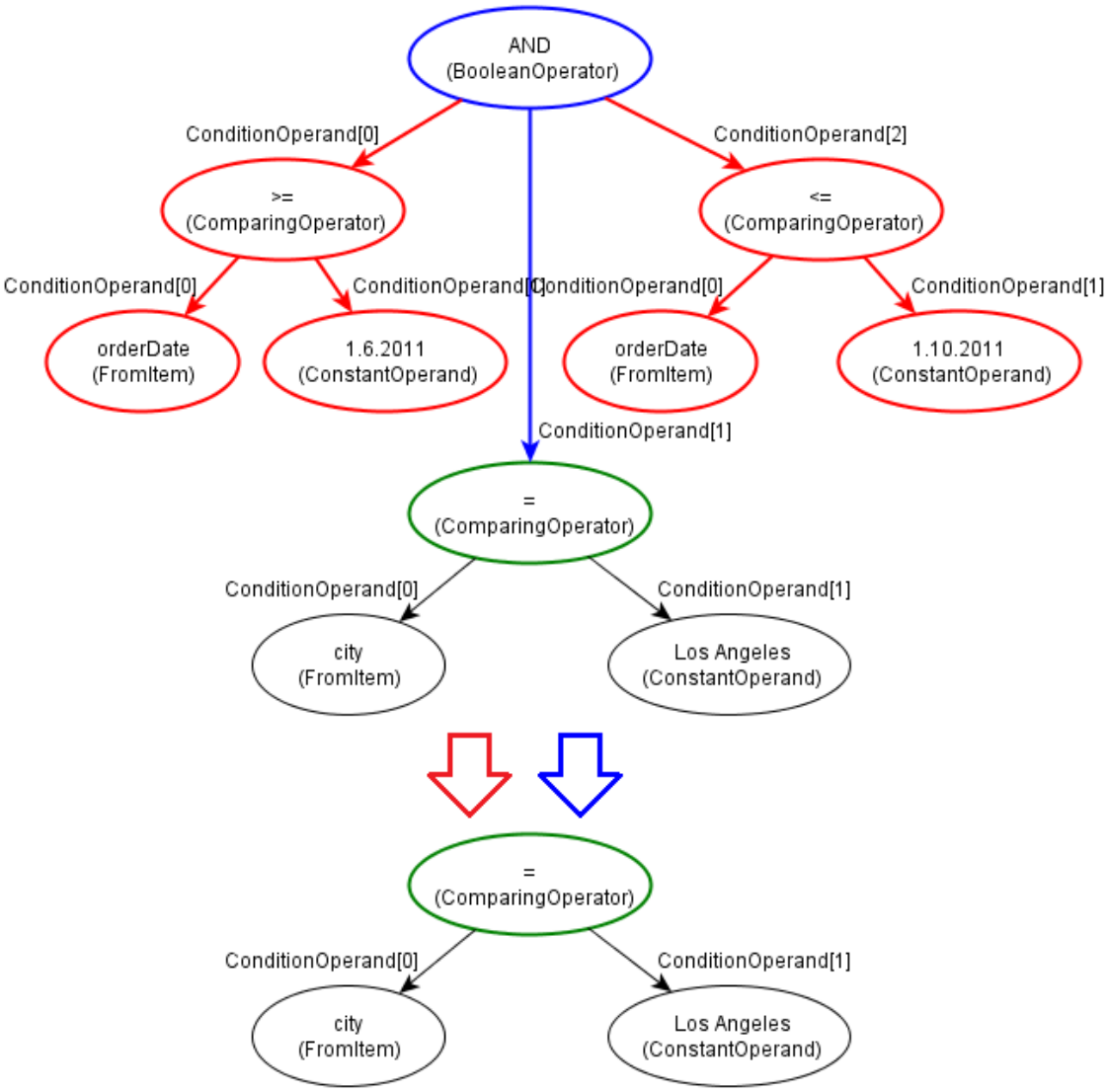


Figure 8.5: An example of reorganization of a condition because of removing of a FromItem vertex.

Algorithm 8.14 RemoveFromItemFromCondition

Input: Vertex V, change context C

Output: Graph operations to reorganize condition, the type of condition reorganization and an eventual new root of the condition subtree.

1: $result.Type \leftarrow NoChange, result.NewRoot \leftarrow null$
2: **if** V is FromItem vertex corresponding to $C.Originator$ **then**
3: $result.Type \leftarrow Remove$
4: **else if** V is AggregateFunction vertex such that the argument of the aggregate function corresponds to $C.Originator$ **then**
5: $RemoveVertexComplete(V, C.Plan, C.G_Q)$
6: $result.Type \leftarrow Remove$
7: **else if** $V \in \{ComparingOperator, QueryOperator\}$ vertex **then**
8: $operand0 \leftarrow V.GetNeighbour(ConditionOperand[0])$
9: $operand1 \leftarrow V.GetNeighbour(ConditionOperand[1])$
10: $result0 \leftarrow RemoveFromItemFromCondition(operand0, C)$
11: $result1 \leftarrow RemoveFromItemFromCondition(operand1, C)$
12: $result.Type \leftarrow result1.Type$
13: **if** $result0.Type = Remove$ **then**
14: $C.Plan \leftarrow RemoveEdge(C.G_Q, V, ConditionOperand[0])$
15: $result.Type \leftarrow Remove$
16: **if** $operand1$ is ConstantOperand vertex **then**
17: $C.Plan \leftarrow RemoveVertex(C.G_Q, operand1)$
18: **end if**
19: **end if**
20: **if** $result1.Type = Remove$ **then**
21: $C.Plan \leftarrow RemoveEdge(C.G_Q, V, ConditionOperand[1])$
22: **end if**
23: **else if** V is BooleanOperator vertex **then**
24: $result \leftarrow RemoveFromItemFromBooleanCondition(V, C)$ {see Algorithm 8.15}
25: **end if**
26: **return** $result$

Algorithm 8.15 RemoveFromItemFromBooleanCondition

Input: BooleanOperator vertex V, change context C

Output: Graph operations to reorganize Boolean condition, the type of condition reorganization and an eventual new root of the condition subtree.

```
1: result.Type ← NoChange, result.NewRoot ← null
2: conditions ← V.GetNeighbours(ConditionOperand)
3: removedConditionsCount ← 0
4: for all i = 0 to conditions.Count do
5:    res ← RemoveFromItemFromCondition(conditions[i], C)
6:    if res.Type = Remove then
7:       C.Plan ← RemoveEdge(C.G_Q, V, ConditionOperand[i])
8:       removedConditionsCount + +
9:    else if res.Type = Adjust then
10:      C.Plan ← RemoveEdge(C.G_Q, V, ConditionOperand[i])
11:      C.Plan ← CreateEdge(C.G_Q, V, result.NewRoot, ConditionOperand)
12:      result.NewRoot ← res.NewRoot
13:   else
14:      result.NewRoot ← conditions[i]
15:   end if
16: end for
17: if removedConditionsCount = conditions.Count then
18:   C.Plan ← RemoveVertex(C.G_Q, V)
19:   result.Type ← Remove
20: else if removedConditionsCount = conditions.Count − 1 then
21:   C.Plan ← RemoveVertex(C.G_Q, V)
22:   result.Type ← Adjust
23: end if
24: return result
```

8.2.7 Removing Database Table

This database model operation is translated into the *removing DataSource* operation. The basis of traversing Algorithm 8.16 is removing of all DataSourceItems of the particular DataSource by traversing Algorithm 8.9.

Algorithm 8.16 starts at the DataSource vertex. From this vertex it raises an event to remove all the DataSourceItems of the particular DataSource. Then it traverses through all Alias vertices to the From vertices.

Algorithm 8.16 DistributeRemovingDatasource

Input: Datasource vertex D, change context C
Output: Graph operations to remove the DataSource.

1: **for all** $datasourceItemVertex \in D.GetNeighbours(DataSourceItem)$ **do**
2: $itemContext \leftarrow$ new $ChangeContext(datasourceVertex, RemoveDataSourceItem)$
3: $DistributeRemovingDatasource(datasourceItemVertex, itemContext)$
4: $C.Plan \leftarrow itemContext.Plan$
5: **end for**
6: $RemoveVertexComplete(D, C.Plan, C.G_Q)$
7: **for all** $aliasVertex \in D.GetNeighbours(Alias)$ **do**
8: $fromVertex \leftarrow aliasVertex.GetNeighbour(DataSourceAlias)$
9: $C.Originator \leftarrow aliasVertex$
10: $DistributeRemovingDatasourceOnFrom(fromVertex, C)$ {see Algorithm 8.17}
11: $RemoveVertexComplete(aliasVertex, C.Plan, C.G_Q)$
12: **end for**

Algorithm 8.17 determines the kind of removing of the Alias vertex related to the removing DataSource vertex from the combination source tree. The determination is performed on the basis of a location of the Alias vertex in the combination source tree. After that Algorithm 8.18 is used in appropriate way to complete the Alias vertex removing.

Algorithm 8.17 DistributeRemovingDatasourceOnFrom

Input: From vertex F, change context C
Output: Graph operations to remove the DataSource from the From component.

1: $alias \leftarrow C.Originator$
2: $source0 \leftarrow alias.GetSourceVertex(MapSource[0])$
3: **if** $source0$!= $null$ **then**
4: $ReorganizeSourceTree(source0, F, 1, C)$ {see Algorithm 8.18}
5: **end if**
6: $source1 \leftarrow alias.GetSourceVertex(MapSource[1])$
7: **if** $source1$!= $null$ **then**
8: $ReorganizeSourceTree(source1, F, 0, C)$
9: **end if**
10: $ResetAllContent(F, C.Plan, C.G_Q)$

Algorithm 8.18 describes how the alias vertex is removed from the combination source tree. The basis of the algorithm is to replace the parent of the removed

alias with its sibling. The situation is depicted in Figures 8.6 and 8.7. Figure 8.6 illustrates simple replacing of the parent with the sibling. Figure 8.7 depicts the situation, where any sibling has to be set as the root of the combination source tree. In the figures a color of an arrow corresponds to the vertex being removed.

Algorithm 8.18 ReorganizeSourceTree

Input: CombineSource vertex V, From vertex F, index i of the sibling edge, change context C

Output: Graph operations to remove alias vertex from the *combination source tree*.

1: $siblingVertex \leftarrow V.GetNeighbour(MapSource[i])$
2: $C.Plan \leftarrow RemoveEdge(C.G_Q, V, MapSource[i])$
3: $sourceVertex \leftarrow V.GetSourceVertex(MapSource[0])$
4: **if** $sourceVertex$!= $null$ **then**
5: $\quad C.Plan \leftarrow RemoveEdge(C.G_Q, sourceVertex, MapSource[0])$
6: $\quad C.Plan \leftarrow CreateEdge(C.G_Q, sourceVertex, siblingVertex, MapSource[0])$
7: **else**
8: $\quad C.Plan \leftarrow RemoveEdge(C.G_Q, F, SourceTree)$
9: $\quad C.Plan \leftarrow CreateEdge(C.G_Q, F, siblingVertex, SourceTree)$
10: **end if**
11: $C.Plan \leftarrow RemoveVertexComplete(V, C.Plan, C.G_Q)$

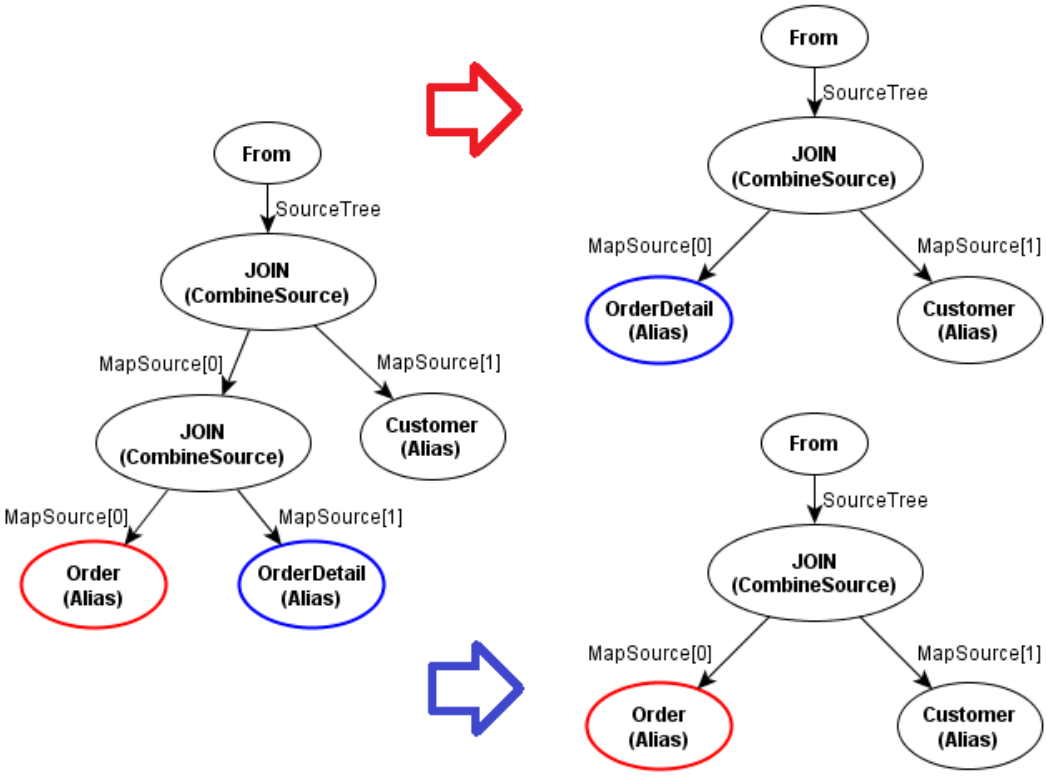


Figure 8.6: An example of a simple replacing of the parent of the removed alias with its sibling in the combination source tree.

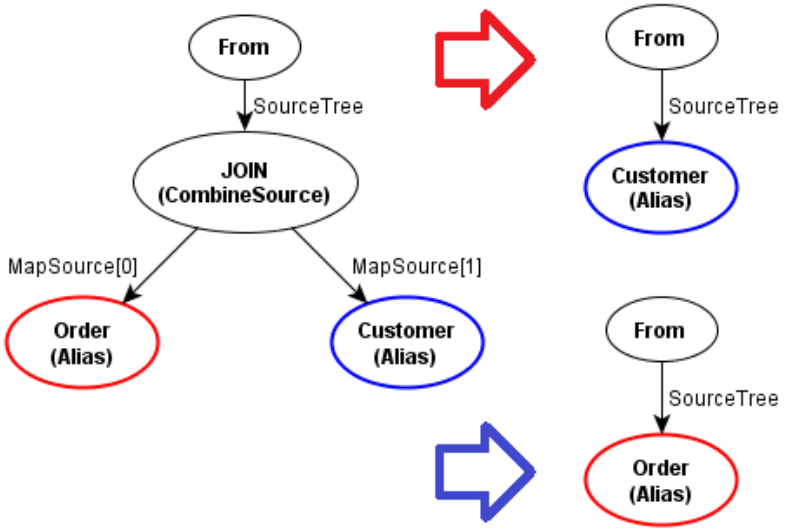


Figure 8.7: An example of a change of the combination source tree root.

Chapter 9

Implementation and Experiments

In this chapter we describe a prototype implementation of the query model described in Chapter 7 and algorithms described in Chapters 7 and 8. It was implemented as an extension of DaemonX, a general framework for modelling and processing of evolution [32].

9.1 DaemonX

DaemonX [32] is a pluginable CASE tool framework for data and/or process modeling. It was developed as a software project at the Faculty of Mathematics and Physics, of the Charles University in Prague. The aim of the framework is to provide funcionality for processing of evolution among various models. The models are implemented as plug-ins, which use *application programmable interface* (API) of the framework. Implementation of additional plug-ins comprises the definition of the models and setting rules for propagation of atomic operations from a source model to a target model. A detailed description of the framework can be found on the website of the DaemonX project [32].

9.2 Implementation

A prototype implementation uses existing simplified PSM database modeling plug-in, which was developed as a part of the first release of the DaemonX project and adds two new plug-ins. The first plug-in is a plug-in for SQL query modeling (user documentation of this plug-in is available on the attached CD). The second plug-in is used for evolution propagation from the PSM database model to the SQL query model.

The prototype implementation has the following features:

- Plug-in for modeling of SQL queries
  - It implements all components necessary for SQL query model described in Chapter 7, i.e. query graph and visualisation model.
  - It provides generating of an SQL query from the query model.
  - It implements algorithms described in Chapter 8 for propagation of changes in a query graph.

- Evolution plug-in for propagation from the PSM database model to the SQL query model
  - It enables creation of a mapping between the PSM database model and the SQL query model.
  - It supports propagation of operations from the source model to the target model.

9.3 Experiments

For performing experiments to provide the proof of the concept of the query model and all used algorithms we used an existing database project Adventure Works [33]. We modeled in the database model the following tables of given database schema:

- Person.BusinessEntity
- Person.BusinessEntityAddress
- Person.Address
- Person.AddressType
- Person.StateProvince
- Person.CountryRegion
- Sales.Store
- Production.Product
- Production.ProductModel
- Production.ProductModelProductDescriptionCulture
- Production.ProductDescription
- Person.Person
- HumanResources.Department

Figure 9.1 depicts a part of the modeled database model. The complete illustration of the modeled database schema can be found in Appendix B.

From this database model we derived to the query model tables as DataSources, which we used to model the following queries and views:

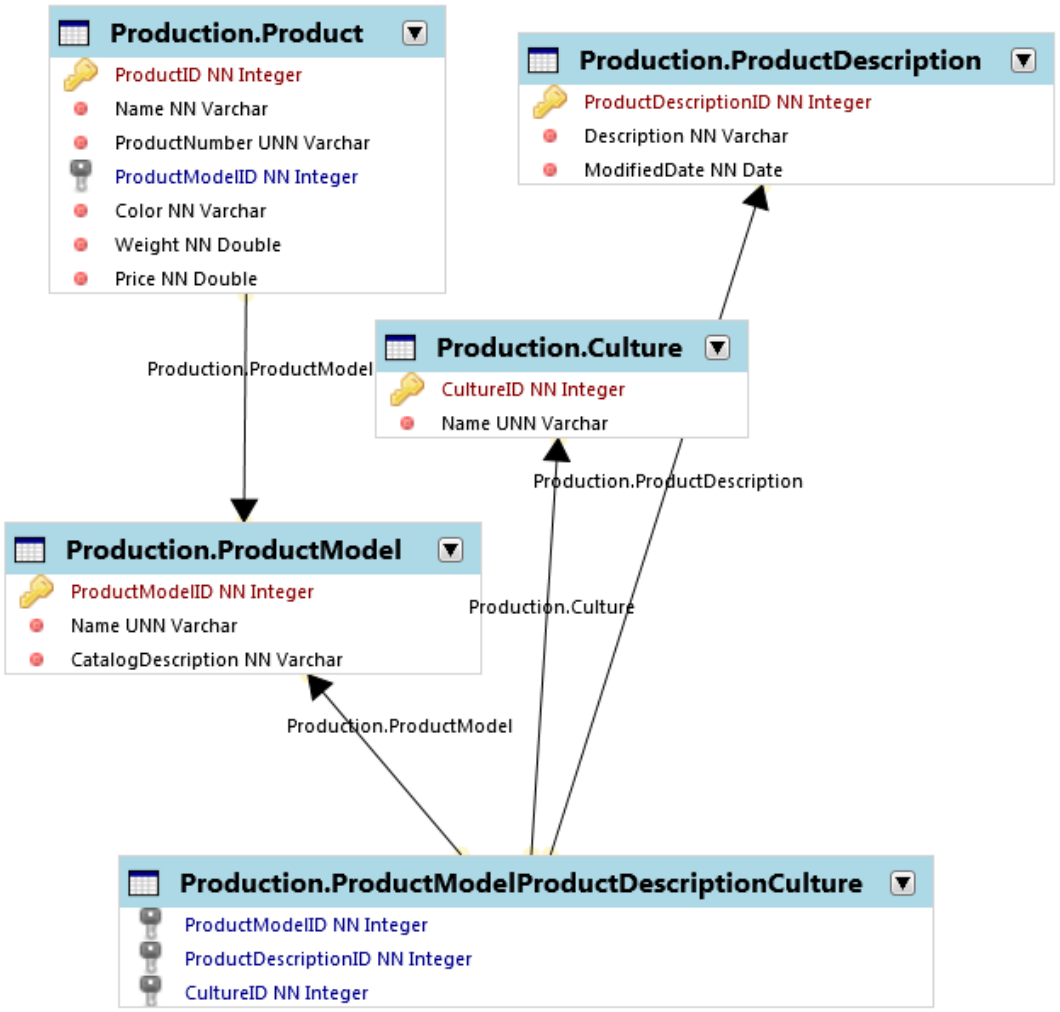


Figure 9.1: A part of the modeled database model.

9.3.1 Basic Select

Figure 9.2 shows a model of a simple Select query. The modeled query is as follows:

```
SELECT
      d.DepartmentID
    , d.Name as DepartmentName
FROM
    HumanResources.Department as d
```

9.3.2 Basic Group-By

Figure 9.3 shows a model of a simple usage of the GROUP-BY clause. The modeled query is as follows:

```
SELECT
      d.GroupName
    , COUNT(d.Name) as NumberOfDepartments
```

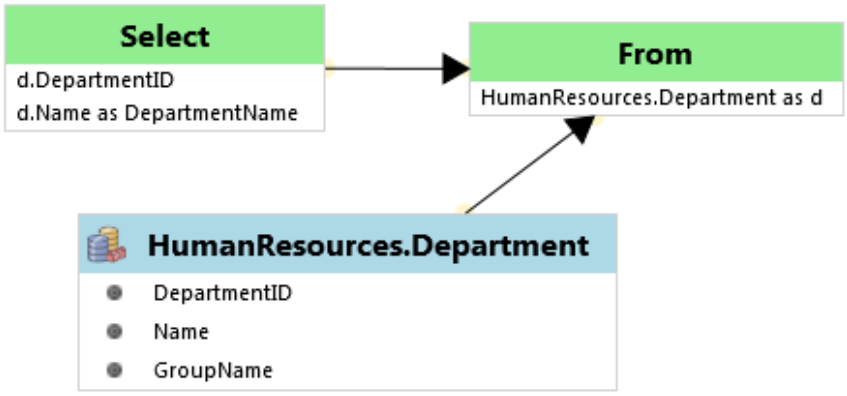


Figure 9.2: The model of the simple SELECT query.

```
FROM
    HumanResources.Department as d
GROUP BY
    d.GroupName
```

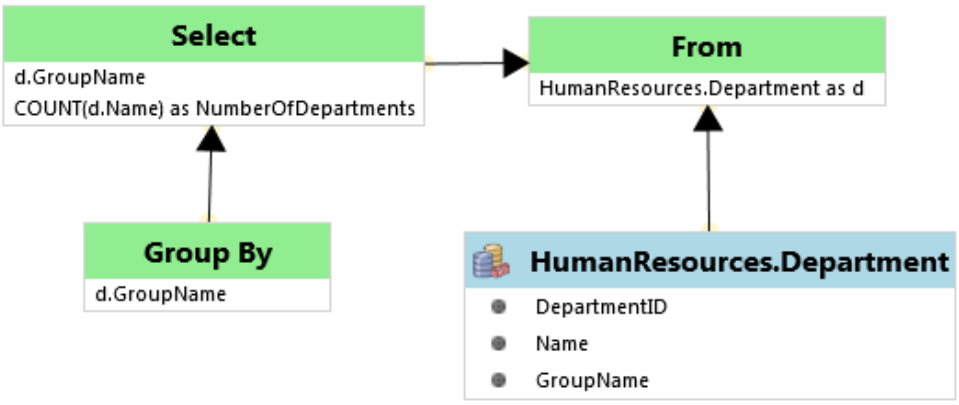


Figure 9.3: The model of the simple usage of the GROUP-BY clause.

9.3.3 Complex Group-By

Figure 9.4 shows a model of a complex usage of the GROUP-BY clause, together with usage of the HAVING clause. The modeled query is as follows:

```
SELECT
      d.GroupName
    , COUNT(d.Name) as NumberOfDepartments
FROM
    HumanResources.Department as d
GROUP BY
    d.GroupName
HAVING
```

```
    COUNT(d.Name) > 2
ORDER BY
    NumberOfDepartments DESC
```

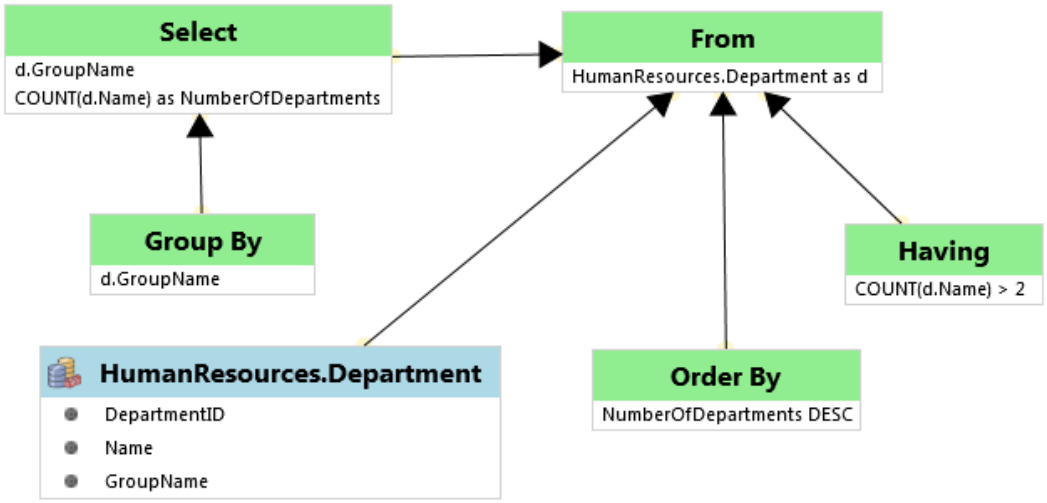


Figure 9.4: The model of the complex usage of GROUP-BY clause.

9.3.4 Query as DataSource

Figure 9.5 shows a model of a usage of a query as a DataSource in another query. The modelled query is as follows:

```
SELECT
      d.DepartmentID
    , d.GroupName
    , d.Name as DepartmentName
FROM
    HumanResources.Department as d
    JOIN
    (
      SELECT
            d.GroupName
          , COUNT(d.Name) as NumberOfDepartments
      FROM
          HumanResources.Department as d
      GROUP BY
            d.GroupName
      HAVING
        COUNT(d.Name) > 2
    ) as d2 ON (d.GroupName = d2.GroupName)
ORDER BY
      d.GroupName ASC
    , d.Name ASC
```

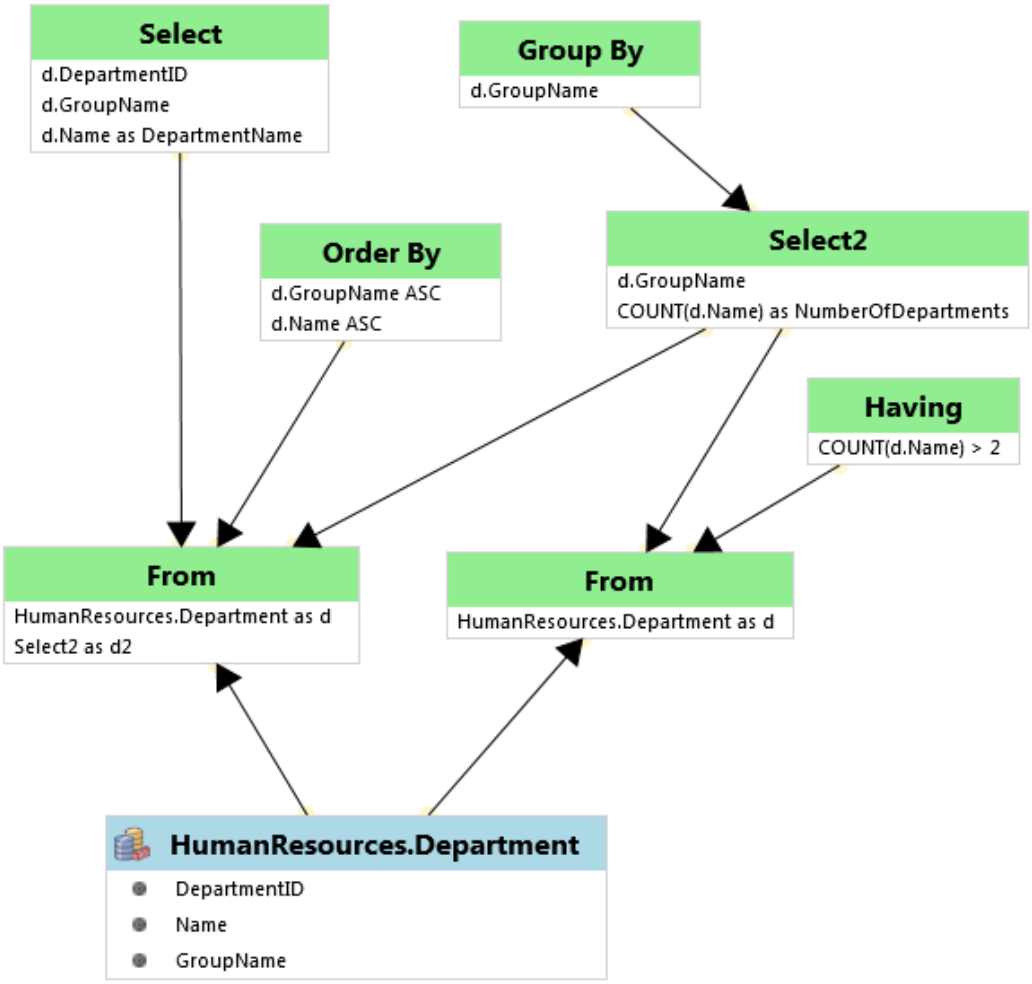


Figure 9.5: The model of the usage query as a DataSource.

9.3.5 Complex Where

Figure 9.6 shows a model of a complex usage of the WHERE clause. The modeled query is as follows:

```
SELECT
      x.FirstName
    , x.LastName
FROM
    (
    SELECT
          p.FirstName
        , p.MiddleName
        , p.LastName
    FROM
        Person.Person as p
    WHERE
        (p.MiddleName IS NOT NULL )
            AND
```

```
        (p.LastName = 'Adams')
            AND
        (p.FirstName LIKE 'A\%' OR p.FirstName LIKE 'K\%')) as x
ORDER BY
        x.FirstName ASC
```

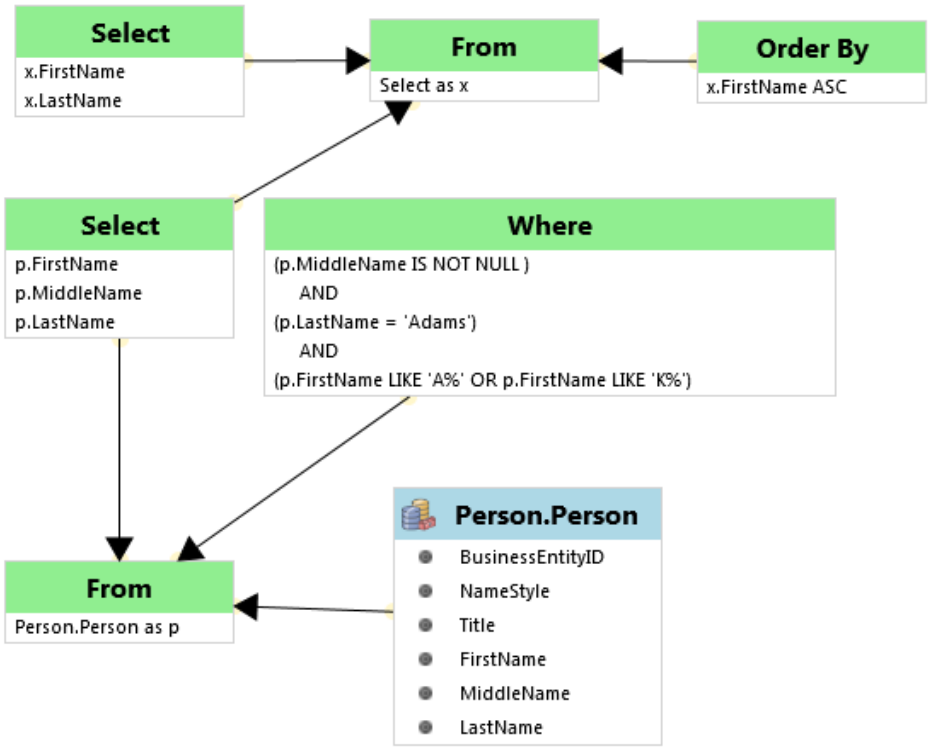


Figure 9.6: The model of the complex usage of the WHERE clause.

9.3.6 View vStoreWithAddresses

Figure 9.7 shows a model of the *vStoreWithAddresses* view. The underlaying query of the *vStoreWithAddresses* view is as follows:

```
SELECT
        s.BusinessEntityId
    ,   s.Name
    ,   at.Name as AddressType
    ,   a.AddressLine1
    ,   a.AddressLine2
    ,   a.City
    ,   sp.Name as StateProvinceName
    ,   a.PostalCode
    ,   cr.Name as CountryRegionName
FROM
    Sales.Store as s
    JOIN
```

```
Person.BusinessEntityAddress as bea ON
  s.BusinessEntityId = bea.BusinessEntityId
JOIN
Person.Address as a ON
  a.AddressId = bea.AddressId
JOIN
Person.StateProvince as sp ON
  sp.StateProvinceID = a.StateProvinceID
JOIN
Person.CountryRegion as cr ON
  cr.CountryRegionCode = sp.CountryRegionCode
JOIN
Person.AddressType as at ON
  at.AddressTypeID = bea.AddressTypeID
```

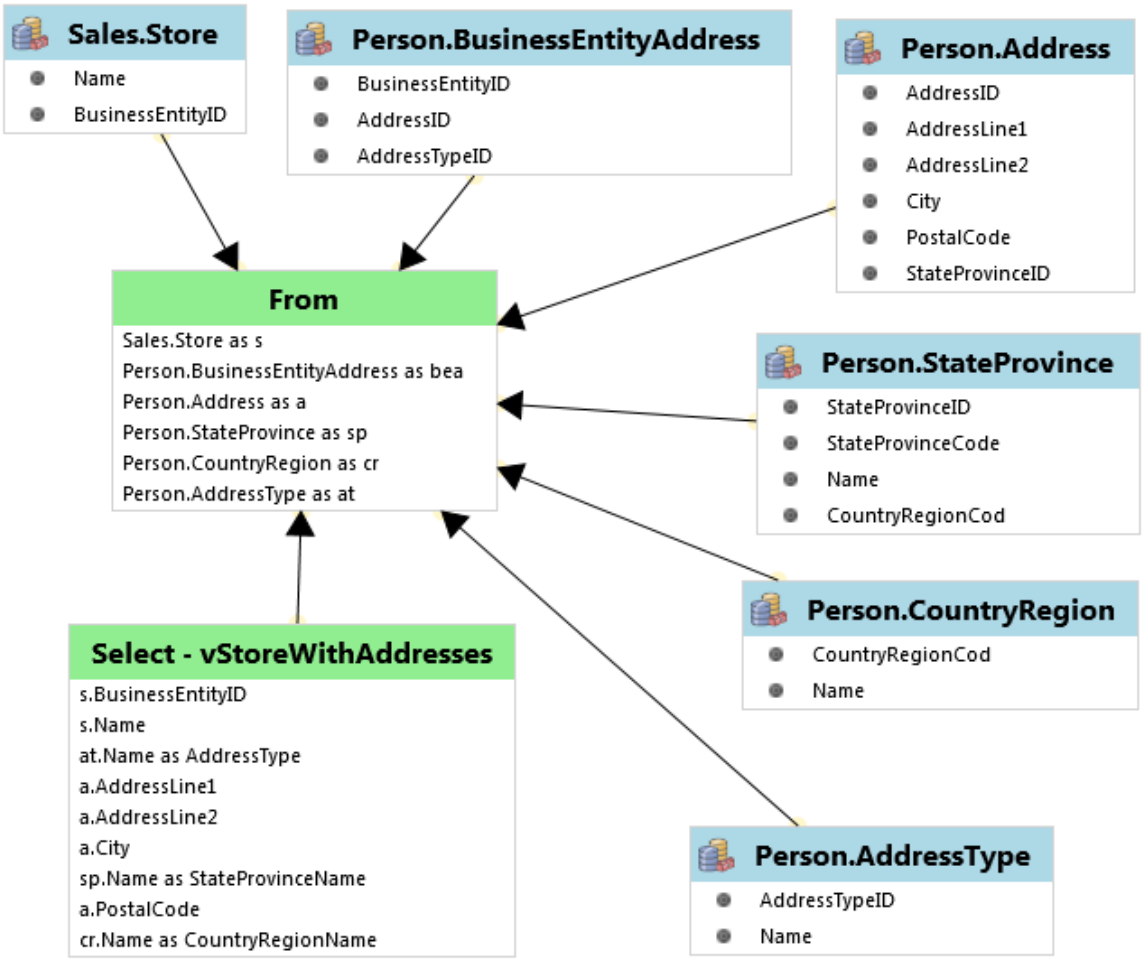


Figure 9.7: The model of *vStoreWithAddresses* view.

9.3.7 View vProductAndDescription

Figure 9.8 shows a model of the *vProductAndDescription* view. The underlaying query of the *vProductAndDescription* view is as follows:

```
SELECT
    p.ProductID
  , p.Name
  , pm.Name as ProductModel
  , pmx.CultureID
  , pd.Description
FROM
  Production.Product as p
  JOIN
  Production.ProductModel as pm  ON
    p.ProductModelID = pm.ProductModelID
  JOIN
  Production.ProductModelProductDescriptionCulture as pmx
ON
    pm.ProductModelID = pmx.ProductModelID
  JOIN
  Production.ProductDescription as pd  ON
    pmx.ProductDescriptionID = pd.ProductDescriptionID
WHERE
  pmx.CultureID = 'en'
ORDER BY
    p.Name ASC
  , pm.Name ASC
```

The test process can be described as follows:

1. We generated the code of the SQL queries.
2. The generated code was tested using SQL Server 2008 R2 [34]. All queries were executed correctly and returned correct results.
3. We applied various changes described in Chapter 8. All updates were executed correctly in all cases as well. Some of performed changes are described in Section 9.3.8.
4. We tested the adapted SQL queries using SQL Server. All queries were executed correctly and returned correct results.

9.3.8 Results of Performed Changes

Application of Creating Table Column Operation on Complex GroupBy Query

The column *GroupID* was added to the table *HumanResources.Department*. This change was propagated into the *Complex GroupBy* query. The new column was added to all components of this query.

The original query

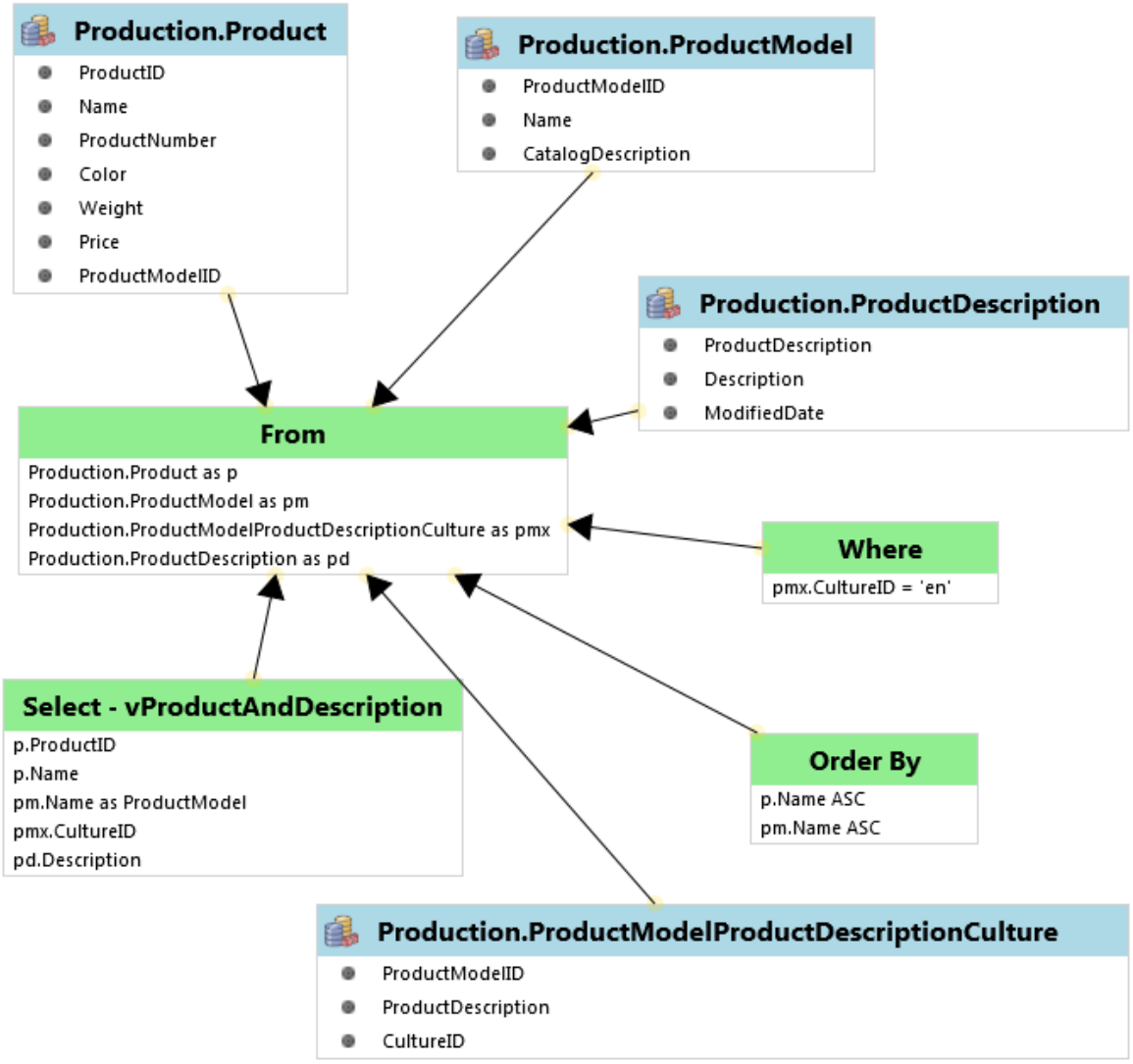


Figure 9.8: The model of *vProductAndDescription* view.

```
SELECT
      d.GroupName
    , COUNT(d.Name) as NumberOfDepartments
FROM
    HumanResources.Department as d
GROUP BY
    d.GroupName
HAVING
   COUNT(d.Name) > 2
ORDER BY
    NumberOfDepartments DESC
```

was transformed by Algorithm 8.1 to the query

```
SELECT
      d.GroupName
    , COUNT(d.Name) as NumberOfDepartments
```

```
    , d.GroupID
FROM
    HumanResources.Department as d
GROUP BY
      d.GroupName
    , d.GroupID
HAVING
   COUNT(d.Name) > 2
ORDER BY
      NumberOfDepartments DESC
    , d.GroupID ASC
```

Figure 9.9 depicts an updated model of the *Complex GroupBy* query.

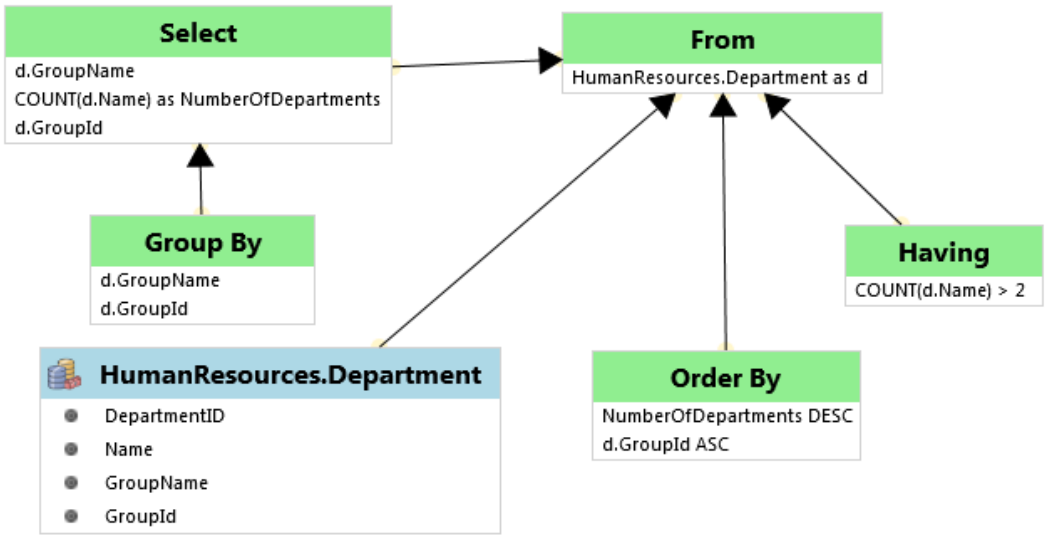


Figure 9.9: The updated model of the *Complex GroupBy* query.

Application of Removing Table Column Operation on Complex Where Query

The column *FirstName* was removed from the table *Person.Person*. This change was propagated into the *Complex Where* query. The new column was removed from all components of the query, including the dependant query, which is not using mapped column *FirstName* directly.

The original query

```
SELECT
      x.FirstName
    , x.LastName
FROM
    (
    SELECT
          p.FirstName
        , p.MiddleName
```

```
    , p.LastName
  FROM
    Person.Person as p
  WHERE
    (p.MiddleName IS NOT NULL )
        AND
    (p.LastName = 'Adams')
        AND
    (p.FirstName LIKE 'A\%' OR p.FirstName LIKE 'K\%')) as x
ORDER BY
    x.FirstName ASC
```

was transformed by Algorithm 8.9 to the query

```
SELECT
    x.LastName
FROM
  (
  SELECT
      p.MiddleName
    , p.LastName
  FROM
    Person.Person as p
  WHERE
    (p.MiddleName IS NOT NULL )
        AND
    (p.LastName = 'Adams')) as x
```

Figure 9.10 depicts an updated model of the *Complex Where* query.

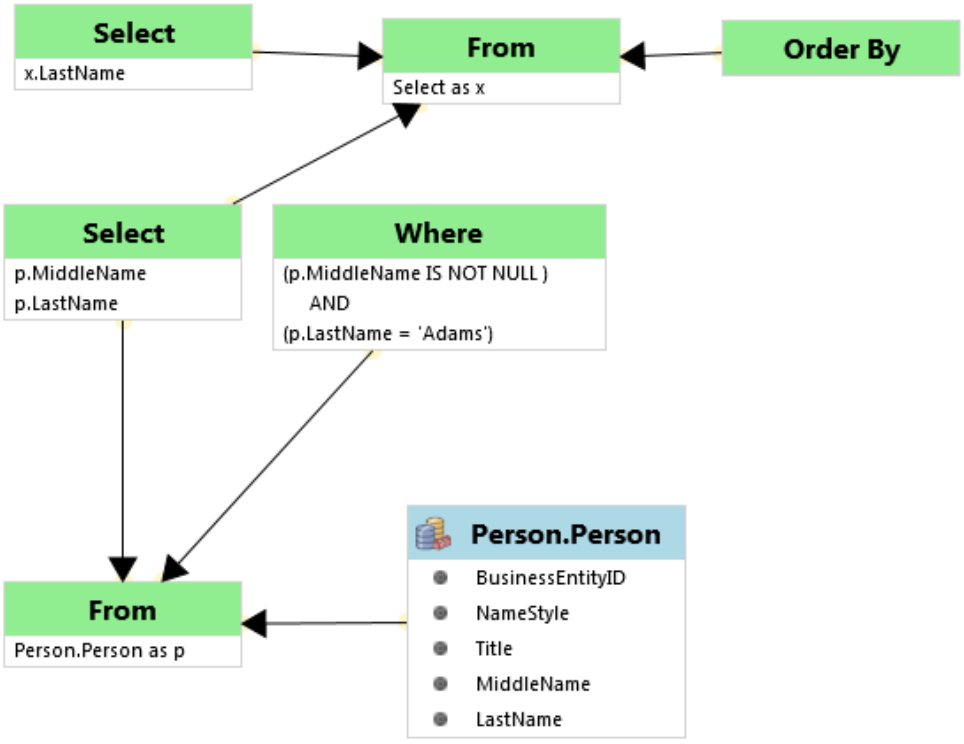


Figure 9.10: The updated model of the *Complex Where* query.

Chapter 10

Conclusion

In this thesis we presented an approach to adaptation of SQL queries, which is an issue related to adaptation of relational database schema problem. We proposed a model, which can be used in CASE tools to model SQL queries over the particular database schema. Similarly, we proposed algorithms, which can determine the impact of changes performed in the database model on the SQL queries and which update queries in order to be correctly defined and return correct results.

We began with introducing of adaptation of relational database schema problem in general. We introduced a problem with changing of database schema and the related problem of incompatibility with related queries (see Chapter 1).

In Chapter 2 we describe the relational data model and selected parts of SQL language. Chapter 3 introduces a base concept of a model driven architecture, which we use to define roles of proposed models.

In Chapter 4 we describe related issues of relational database schema adaptation and discuss key problems and solutions. Chapter 5 contains an analysis of related works dealing with a selected database schema adaptation problem.

In Chapter 7 the main contribution of this thesis is described. We propose a graph-based SQL query model, particularly designed for evolution process. The platform-specific visualisation model of the query is described as well. An important part of this chapter is focused on mapping between the SQL query model and the database model. The end of this chapter describes an algorithm to generate SQL query from the proposed query model.

In Chapter 8 we discuss the impact of changes in the database model on the queries in the SQL query model. The main part of this chapter is focused on a description of algorithms which distribute changes in the query graph according to changes performed in the database model.

The implementation of the proposed solution is described in Chapter 9 together with test experiments. A prototype implementation of the proposed model and algorithms was implemented as an extension of the DaemonX framework. It is available on the attached CD.

10.1 Main Contributions

The main contribution of our approach is the ability to model SQL queries in a CASE tool, to analyze changes performed in the database model and to update the queries to preserve their compatibility and correctnes. It enables us to perform

changes in the database model without an inspection of all the related views / queries by looking for an incompatibility of the queries and the new database schema. Changes in the database schema model are propagated immediately to the SQL query model thanks to the mapping between these two models.

10.2 Open Problems

Even though the approach is complex and robust, there exists some problems that are not covered by this thesis. The main problem is the semantics of database schema elements.

There is no way how to add semantics to the database table or table columns used in SQL queries. This reflects during the propagation of changes:

- A column was deleted, its occurence has to be deleted from all parts of the SQL query, including conditions. Was the column important in the semantics of the given query, or not? For instance, suppose a view *MarriedPerson* that selects persons who are married. The person's status is evaluated according to a value of a given column. But when this column is removed, it is also removed from the condition of the *MarriedPerson* view. Now the semantics of this view is broken, because it selects all the persons instead of only those persons, who are married.
- A table was deleted. Does the query after performed changes have any sense?
- A new column was added to the database table and during the propagation of changes it was added to the Select clause. Is it necessary to add any condition in order to retain semantics of the given query?

The semantics in SQL queries is an open problem in general. We could find simply other examples of parts of the SQL query, where the semantics could be broken.

10.3 Future work

Our approach covers a large subset of SQL constructs and for this reason it can be successfully used in practice. To further extend its practical applicability, the future research can examine the following areas.

10.3.1 Richer SQL Syntax

In Section 7.1 we described limitations for supporting of SQL language in the query model. These limitations are not very restricting, but for many cases it is not sufficient. The possible enlargment of the syntax (e.g. support of expressions) covers an extension of the query model (defining of new types of vertices, defining of new types of edges, extension of traversing algorithms, etc.), expansion the analysis of changes performed in the database model and their propagation to the query model. However, this will make the performing propagation much more complicated.

10.3.2 Extension of Query Model

The query model extension does not relate only to the previous future work recommendation, but the query model could be also extended by a support of database stored functions, procedures, constraints or triggers. Such an extension could cover a complete database schema. But in this case the problem with semantics is much more important and it would require an extensive research.

Appendix A

Attachments

The attached CD contains:

- PDF version of this thesis – *thesis.pdf*.
- Installer of the DaemonX framework with appropriate plug-ins in the folder *implementation*.
- Examples of the database models and SQL query models determined to evolution process in the folder *examples*.

Appendix B

Used Database Schemas and SQL Queries

B.1 Example of the Query Graph

The visualisation of the query graph of a query depicted in Figure 7.14 is saved on the attached CD in the file *examples/queryGraph.png*.

B.2 Model of the Database Schema

The complete illustration of the modeled database schema used in Section 9.3 is saved on the attached CD in the file *examples/databaseSchema.png*.

B.3 Basic Queries

The DaemonX project with the model of basic queries and the corresponding database schema is stored in the file *examples/basicQueries.dx*.

B.4 View vStoreWithAddresses

The DaemonX project with the selected part of the AdventureWorks database schema and the model of the database view *vStoreWithAddresses* is saved on the attached CD in the file *examples/vStoreWithAddresses.dx*.

B.5 View vProductAndDescription

The DaemonX project with the selected part of the database schema and the model of the database view *vProductAndDescription* is stored in the file *examples/vProductAndDescription.dx*.

Bibliography

[1] Lukovic, I., Ristic, S., Mogin, P., Pavicevic, J. *Database schema integration process - a methodology and aspects of its applying.* Novi Sad: J. Math, 2006.

[2] Choobinch J., Mannio V. M., Nunamaker F. J., Konsynski R. B. *An Expert Database Design System Based on Analysis of Forms.* IEEE Transactions on Software Engineering, Vol. 14, No. 2 Feb. 1988, pp. 242-253.

[3] Beeri C., Bernstein P. A. *Computational Problems Related to the Design of Normal Form Relational Schemas.* ACM Transactions on Database Systems, Vol. 4 No. 1, March 1979, pp. 30-59.

[4] Roddick J.F. *A survey of schema versioning Issues for database systems.* In Information Software Technology, 37(7): p.383-393.

[5] Ventrone V., Heiler S. *Semantic heterogeneity as a result of domain evolution.* SIGMOD 1991. Rec. 20(4):16-20.

[6] Batini C., Lenzerini M., Navathe, S.B. *A comparative analysis of methodologies for database schema integration.* ACM Computing Surveys. Vol. 18, No. 4, December 1986

[7] Shneiderman B., Thomas G. *An architecture for automatic relational database system conversion.* ACM Transactions on Database Systems 7(2):235-257, 1982.

[8] Hudicka J.R. *An Overview of Data Migration Methodology.* Select Magazine. April 1998

[9] Gilbert L., Hewitt J. *http://searchdatacenter.techtarget.com/definition/OLTP/.* Last update in August 2010.

[10] Chaudhuri S., Dayal U. *An overview of data warehousing and OLAP technology.* SIGMOD Rec., Vol. 26, No. 1. (March 1997), pp. 65-74.

[11] Deutsch A., Popa L., Tannen V. *Physical Data Independence, Constraints and Optimization with Universal Plans.* In VLDB, pages 459–470, 1999.

[12] Deutsch A., Tannen V. *Mars: A system for publishing XML from mixed and redundant storage.* In VLDB, 2003.

[13] Deutsch A., Tannen V. *XML Queries and Constraints, Containment and Reformulation*. TCS, 336(1):57–87, 2005.

[14] Curino C., Moon H. J., Zaniolo C. *Graceful database schema evolution: the PRISM workbench*. Very Large DataBases (VLDB), 1, 2008.

[15] Drumm Ch., Schmitt, M., Do H.-H., Rahm E. *QuickMig: Automatic Schema Matching for Data Migration Projects*. Proceedings of the sixteenth ACM conference on Conference on information and knowledge management (CIKM). p. 107–116, 2007.

[16] Do H.-H. *Schema Matching and Mapping-based Data Integration*. Verlag Dr. Müller (VDM), 2006. ISBN 3-86550-997-5.

[17] Papastefanatos G., Vassiliadis P., Vassiliou Y. *Adaptive Query Formulation to Handle Database Evolution*. In Proceedings of CAiSE Forum, 2006.

[18] Papastefanatos G., Vassiliadis P., Simitsis A., Aggistalis K., Pechlivani F., Vassiliou Y. *Language Extensions for the Automation of Database Schema Evolution*. In Proceedings of ICEIS (1). 2008, 74-81.

[19] Codd, E.F. *The Relational Model for Database Management: Version 2*. Reading, Mass, Addison-Wesley. 1990.

[20] Pokorný J. *Relational Data Model*. Introduction to Database Systems, lecture notes. 1999.

[21] Sreenivasa, K. *Relational Model*. Introduction to Database Systems and Design, lecture notes. 2006.

[22] Yi-Shin Ch. *Relational Data Model*. Introduction to Database Systems, lecture notes. 2007.

[23] Maier D. *Theory of Relational Databases*. Computer Science Pr., 1983. ISBN: 0914894420

[24] Peter Pin-Shan Chen. *The entity-relationship model: toward a unified view of data*. SIGIR Forum 10, 3 (December 1975), 9-9. `http://doi.acm.org/10.1145/1095277.1095279`

[25] Chawathe S.S. *A Quick Introduction to Relational Algebra*. University of Maine, October, 2006.

[26] Spector, P. *Introduction to SQL*. Statistical Computing Facility, University of California, Berkeley, 1999.

[27] Sreenivasa, K. *The SQL Standard*. Introduction to Database Systems and Design, lecture notes. 2006.

[28] Melton, J. *(ISO-ANSI Working Draft) Foundation (SQL/Foundation)*. ANSI TC NCITS H2, ISO/IEC JTC 1/SC 32/WG 3, Database, 2003.

[29] Brashear D. *ANSI SQL-92 Standard.*
http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt

[30] Korovin K. *CNF and clausal form.* Logic in Computer Science, lecture notes. The University of Manchester, 2006.

[31] Miller J., Mukerji J. *MDA Guide Version 1.0.1.* Object Management Group, 2003.
http://www.omg.org/mda/mda_files/MDA_Guide_Version1-0.pdf.

[32] DaemonX Team. *DaemonX.*
http://daemonx.codeplex.com/, June 2011.

[33] Adventure Works team. *Adventure Works 2008R2.*
Released November 2010,
http://msftdbprodsamples.codeplex.com/.

[34] Microsoft Corporation. *SQL Server 2008 R2.*
http://www.microsoft.com/sqlserver/.

[35] Object Management Group. *Unified Modeling Language.*
http://www.uml.org/

[36] SAP Corporation. http://www.sap.com/.