Otto-von-Guericke-Universität Magdeburg

School of Computer Science
Department of Technical and Business Information Systems

Master Thesis

Design and Implementation of Customizable Query Processors

Author:
Shuai Cao

2nd June 2009

Advisor:

Prof. Dr. rer. nat. habil. Gunter Saake,
Dipl.-Inform. Marko Rosenmüller,
Dipl.-Inform. Norbert Siegmund

University of Magdeburg
School of Computer Science
Department of Technical and Business Information Systems
P.O.Box 4120, D–39016 Magdeburg
Germany

Cao, Shuai:
Design and Implementation of Customizable Query Processors
Marster Thesis, Otto-von-Guericke-Universität Magdeburg, 2009.

Acknowledgements

I would like to thank to Prof.Dr.Gunter Saake for his support and confidence in me and allowing me to work on master thesis in his group. Thank you!

I would like thank to my senior supervisor, Dipl.-Inform. Marko Rosenmueller and Dipl.-Inform. Norbert Siegmund, for their encouragement, guidance, fruitful discussions and many other helps. Without their full support, this thesis will be impossible. Thank you!

I would also like to thank M.Sc. Sagar Sunkle and M.Sc. Syed Saif ur Rahman for their productive discussions on my work. Thank you!

I would like to thank my friend Rong for correction my English many times. Thank you!

I would like special thank to my wife Lingzhi Meng and my baby Aimeng for their encouragement, love and “mental support” in good and bad times.

Thanks to all!

Contents

List of Figures

List of Tables

List of Abbreviations

ANSL	American National Standards Institute
AST	Abstract Syntax Tree
AHEAD	Algebraic Hierarchical Equations for Application Design
BNF	Backus-Naur Form
CBO	Cost-Based Optimizer
DBMS	Database Management System
DML	Data Manipunation Language
FOP	Feature-Oriented Programming
FODA	Feature-Oriented Domain Analysis
FOSD	Feature-Oriented Software Development
FOD	Feature-Oriented Decomposition
ISAM	Indexed Sequential Access Method
JTS	Jakarta Tool Suite
OOP	Object-Oriented Programming
RAT	Relational Algebra Tree

RBO	Rule-Based Optimizer
SCQL	Structured Card Query Language
SPL	Software Product Line
SPLE	Software Product Line Engineering
SQL	Structured Query Language

Chapter 1

Introduction

1.1 Motivation

With breakthroughs in different domains of computer systems, like increase in processing, connection speed, in memory and storage capacity, it is now possible to create data driven applications. For such applications, their data can be kept anywhere like on a server or a desktop machine and can be accessed irrespective of the location like from a local or a remote machine or over the network. Growth in the development of embedded systems has been triggered by increase in memory size while simultaneously a decrease in their prices. These applications are oriented towards a specific user requirement, are simple in nature and are required to perform simple to moderate data manipulation. Depending on the type of application and its usage, an application requires varying amount of access to a database like a general query support, Indexed Sequential Access Method (*ISAM*) access or synchronization with backend database system [Nor07].

Developing a complete *Database Management System (DBMS)* for small intelligent devices like cellphones, sensors, smartcards, PDAs etc. [SR98] is difficult to implement due to the lack of large stable storage like main memory, low computing power, different hardwar platforms and operating systems. These new application domains differ from traditional database applications. These new applications have a different and specialized set of operation that must be efficiently supported by the database system [CDG+90]. The traditional database technology cannot fit into some application areas due to these limits. The foremost need of the user is the development of a special-purpose database system tailored to a specific application scenario. In order to solve this problem, we apply one of the appropriate methods named Tailor-Made Data Management System[1]. This method focuses on reusing functionality in different variants of a DBMS that will satisfy the needs of different application scenarios. For example the EXODUS Extensible DBMS [CDG+90], provides basic DBMS functionality. It can be extended to a rule-based

[1]http://wwwiti.cs.uni-magdeburg.de/iti-db/workshops/SETMDM/

query optimizer generator and a persistent programming language [CDG+90].

Structured Query Language (SQL) is the basis for interaction between a user and relational database. It was proposed in 1974 by Boyce and Chamberlin and since then has gained widespread attention and the fast development. Although SQL is an *ANSI (American National Standards Institute)* standard computer language for accessing and manipulating database systems, nearly all big database vendors provide proprietary extensions in addition to the SQL standard. These extensions have caused the appearance of different versions of the SQL language, like *Structured Card Query Language (SCQL)* for smart cards [Int99], TinySQL for sensor networks [MMHH05], or StreamSQL and Oracle's CQL for stream processing [ZJM+08] etc. These SQL dialects are usually separate from the standard language or included as extension packages [RKS+09]. These extensions intensified the language complexity, making it difficult for the developers to understand the complete semantics. For example, in widely used navigation systems in cars, an application developer only needs a simple select statement, other SQL statements like insert, update, delete, are not required. Insert, update and delete just add to the complexity. Another example is a sensor network, it needs only a small subset of SQL, some complex SQL statements like joins, XML extensions, windows functions, or recursive queries are not needed. Therefore, for developer, it is not necessary to understand all the statements of SQL, he only needs to concentrate and apply his energy in developing the part demanded by the user or application scenarios [RKS+09].

Object oriented Programming is an ad-hoc way to address variations in a product family. This can be achieved by implementing functionalities directly into the code of the base program. The problem which arises here is that the resulting code is littered with an IF-THEN control statement at every function where the program chooses which variant to produce. The problem which this approach is that it lacks *modularity* and *reusability* [NS02]. A much better and modular approach would be to use *polymorphism.* Polymorphism replaces the IF-THEN with polymorphism different subclasses of a class are instantiated which represents the specification of each variation that the IF-THEM control statements represent. These subclasses then replace the IF-THEM control statemens. This approach requeries a significant amount of manual labor [BJK05]. However, this approach is unable to represent every kind of functionality because of the inherent constraints in the expressiveness provided by the OOP language.

A much better solution is to modularize the software corresponding to the functionality it provides. Modules which increments the functionality of the system are knowns as features. Features provide building blocks for software development in FOP. The fundamental principle of FOP is to produce different variations of a software by composing together different features. Feature interaction is managed by feature composition. Feature interacting is important to ensure the validity of a composition. FOP concepts are applied to implement the features of query engine. It modularizes the software development of a system in a base program which can be entended with any number of features modules to form new software variations.

In regard to the different application scenarios and SQL dialects, we need a customizable SQL query processing engine. This customizable SQL engine processes only the queries of application needed SQL dialect. For example, the SQL support not join functionality in the sensor network, the customizable SQL engine provide also not functionality corresponding to this SQL dialect. In this thesis, we apply *Feature-Oriented Programming (FOP)* and the *Software Product Line (SPL)* concept to design and realize a fully customizable query processor.

1.2 Goals

Feature-Oriented software development(FOSD)aims at modularizing software to support the development of tailor-made applications that contain only needed functionality. It is based on the decomposition of software with respect to relevant features. Thus a concrete tailor-made software product can be generated by selecting needed functionality. FOSD can also be applied to SQL which results in a family of SQL dialects where a dialect describes the queries for a particular application scenario.

The aim of this thesis is to apply FOSD to the query processing of database management systems(DBMS). This includes analysis and design of the query processing subsystem and its integration into an existing DBMS product line to enable processing of queries of different SQL dialects. It should be analyzed how the feature model of a family of SQL dialects and the feature model of the extended DBMS(including the query engine)can be combined. This should be the basis for automatic generation of a DBMS according to a given SQL dialect.

To evaluate the developed approach, an existing DBMS product line implementaion has to be extended with a prototypical query engine using feature-oriented programming. Based on the analysis of possible features, the query engine should be implemented using an existing SQL parser.

The following tasks have to be considered:

- Analysis of existing tools and modeling approaches to integrate multiple feature models.
- Analysis of possible features for a SQL query engine and its integration into the DBMS feature model.
- Analysis of dependencies between a SQL feature model and the extended DBMS model.

- Creating a model that integrates both feature models(SQL and query engine)and allows for derivation of DBMS functionality based on a tailor-made SQL dialect.
- Evaluation of the developed solution with a prototypical implementation of a query engine.

1.3 Structure of the Thesis

Chapter 2 we lay the some foundations for understanding the central ideas of this thesis. The focus will be on the essential concepts related to Software Product Line Engineering and its two sub-processes, Domain Engineering and Application Engineering, Feature-Oriented Domain Analysis, Feature-Oriented Programming and Query Processor. Consciously, we avoid getting into much detail.

Chapter 3 we introduced some basic feature models which will be used in our query processor. Then we illustrate the background feature diagram for FAME-DBMS and SQL Parser. With the aid of this diagrams, we could better understand the decomposition of Query processors. Then we describe in detail the query processors feature diagram, after which we list the constraints between these feature models.

Chapter 4 we will describe in detail how the query processing has been implemented as well as the different issues which will appear in the practical application.

Chapter 5 On the end, we will provide the evaluation in chapter 5 and lists suggestions for future work.

Chapter 2

Background

In this chapter, we explore various topics required as background for good understanding of the feature-oriented decomposition of query processing. We first review the concept of software product lines. Then we talk about feature-oriented domain analysis and feature-oriented programming. Finally, we explore the basic process of query processing.

2.1 Software Product Line Concepts

Software engineering community has long been striving for the effective reuse of software. When the size and complexity exceeds the limits of what is feasible with traditional approach, the advantages of adopting product line engineering can only then be fully comprehended in software domain. Software Product Line Engineering has shown that software systems can be built at lower costs, in short time and with higher quality [PBvdL05]. It has proven to be the methodology for developing a diversity of software products and software-intensive systems. SPLE contains two distinct development processes: Domain Engineering and Application Engineering. We will introduce them in the following sections.

2.1.1 Domain Engineering

Domain Engineering is a process of designing a whole family of software systems and does not focus on a single software system. Most software organizations work only on a single domain or a small set of domains, building similar systems repeatedly. Differnt definitions of Domain Engineering can be found in related literature. Czarnecki et al. [CE00] define Domain Engineering as follows:

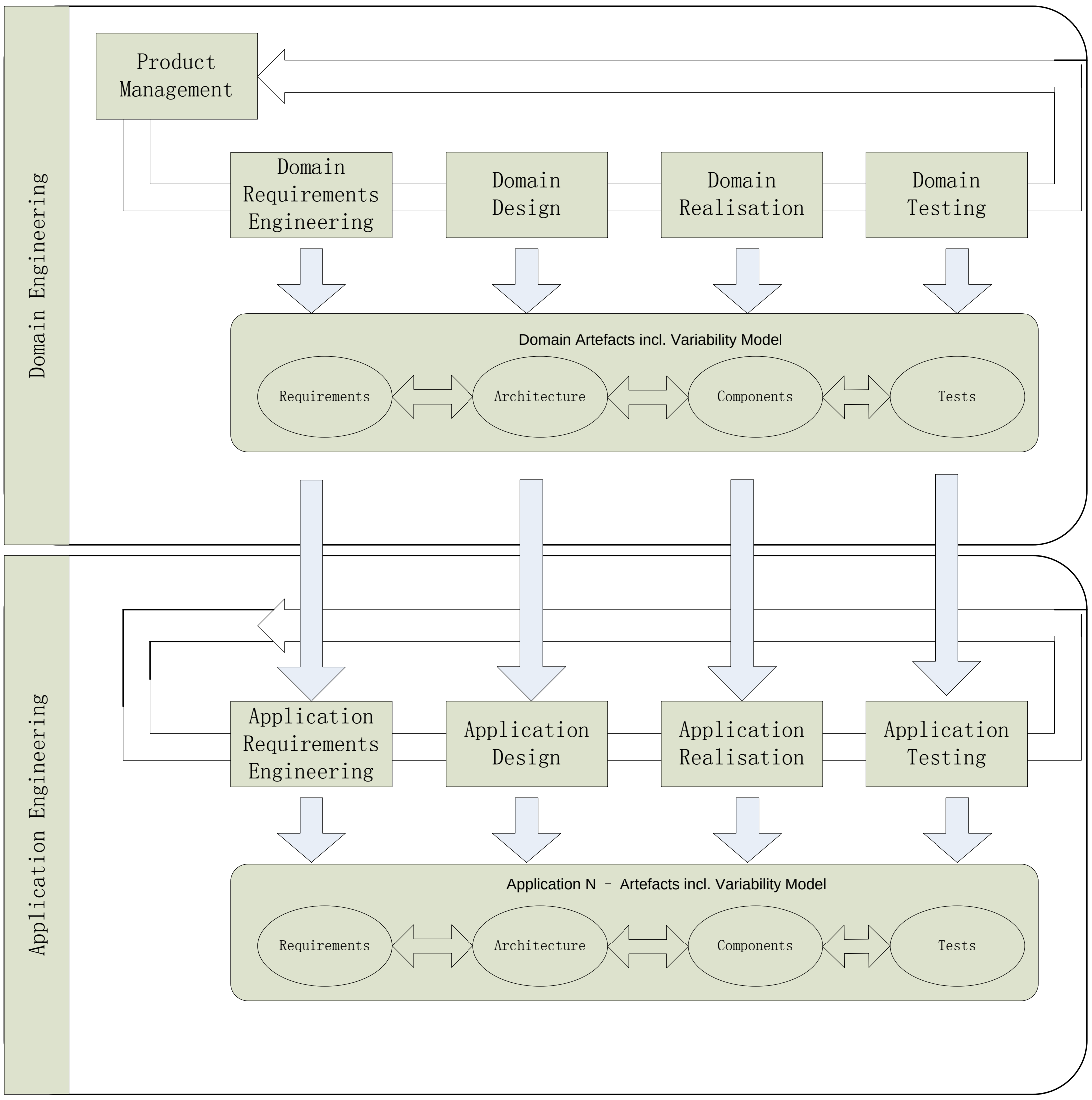

Figure 2.1: The software product line engineering framework [PBvdL05]

Domain Engineering is the activity of collecting, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets(i.e., reusable work products,) as well as providing an adequate means for reusing these assets(i.e., retrieval, qualification, disseminations, adaptation, assembly, and so on) when building new systems.

Czarnecki et al. [CE00] describes Domain Engineering encompasses *Domain Analysis*, *Domain Design*, and *Domain Implementation.* The results of Domain Engineering are reused during *Application Engineering.*

In Klaus Pohl et al. [PBvdL05] give the following definition of Domain Engineering.

Domain engineering is the process of software product line engineering in which the commonality and the variability of the product line are defined and realised.

Klaus Pohl et al. [PBvdL05] has also given the following goals of domain engineering:

- Define the commonality and the variability of the software product line.
- Define the set of applications the software product line is planned for, i.e. define the scope of the software product line.
- Define and construct reusable artefacts that accomplish the desired variablity.

The *domain engineering process* (depicted in the upper part of Figure 2.1) is composed of five key sub-processes: *product management*, *domain requirements engineering*, *domain design*, *domain realisation*, and *domain testing.*

Product Management Product Management is a sub-process of domain engineering. Its main focus is on the whole process of the product development that meets customer's needs. It also makes sure that the entrepreneurial goals that follows the concept of the software engineering process. All of the other sub-processes of domain engineering and application engineering will be related with it. In summary, product management deals with the economical aspects of the software product line.

Domain Requirements Engineering The domain requirements engineering subprocess encompasses all activities for eliciting and documenting the common and variable requirements of the product line. It is concerned with the development of detailed common and variable requirements and their precise documentation based on the initial features provided by product management. The input to the application requirements engineering is also provided by domain requirements engineering, such inputs are concerned with creating application-specific requirements artefacts [PBvdL05].

Czarnecki et al. [CE00] express the opinion that the product management and domain requirements engineering sub-processes provide the same functionality as the *domian analysis.*

Domain Design The domain design sub-process is used to produce a product line reference architecture [PBvdL05]. A reference architecture provides a common, high-level structure for all product line applications. A domain architect provides common and variability features, which can be decided according to the requirements of the customer. Domain design is very closely related to the domain requirements engineering, domain realisation, and application design. It provides a reference architecture for the software product line as the input to domain realisation and application design. The user has the ability to select and configure reusable software artefacts because of the characteristic of this architecture [PBvdL05].

Domain Realisation Domain realisation is the sub-process between domain design and domain testing. It develops the reference architecture which contains a list of reusable software artefacts from the sub-process domain design. Similarly, it provides the detailed design and implementation assets of reusable software components to domain testing. Reusable components and interfaces are the main parts of the reusable software assets points. In addition, domain realisation incorporates configuration mechanisms that enable application realisation to select variants and build an application with the reusable components and interfaces [PBvdL05].

Domain Testing The goal of domain testing is to veirfy the output of all other sub-processes of domain engineering, and then provide an efficient overall testing process. Domain testing focuses on the specification, i.e. requirements, architecure, and design artefacts. In addition, domain testing provides reusable test artefacts to reduce the effort for application testing [PBvdL05].

2.1.2 Application Engineering

Klaus Pohl et al. [PBvdL05] define Application Engineering as *Application engineering is the process of software product line engineering in which the applications of the product line are built by reusing domain artefacts(domain artefacts are reusable development artefacts created in the sub-processes of domain engineering) and exploiting the product line variablity.* The key goals of the application engineering process are [PBvdL05]:

- When developing and defining a product line application, application engineering achieves almost all the reuse of the domain assets.

- While developing the product line application, the commonality and the variability of the software product line should be exploited.
- Application artefacts like application requirements, architecture, components and tests should be documented and related to the domain artefacts.
- Keeping in view the application needs from requirements over architecture, to components and test cases and should be bound to variability.
- The impacts of the differences between application and domain requirements on architecture, components and tests should be estimated.

Application Engineering (depicted in the lower part of 2.1) is composed of following sub-processes: *application requirements engineering*, *application design*, *application realisation*, and *application test*. Each of the sub-processes uses domain artefacts and produces application artefacts. In this thesis, we will not provide detailed description of these sub processes.

Separation of Concerns in Domain Engineering and Application Engineering

Separation of concerns is at the core of software engineering. For a particular concept, goal, or purpose, separation of concerns means the ability to identify, encapsulate, and manipulate those parts of software. Dijkstra [Dij97] and Parnas [Par76] have applied the fundamental of *divide-and-conquer* to software development. The idea is that *it is easier to manage a problem by breaking it down into smaller pieces than to solve the problem as is.* These pieces refer to the concerns of a software system. Concerns are the main method of organizing and decomposing software into smaller, more manageable and comprehensible parts [1]. For example, objects and classes are modeled as sepearate concerns in object-orinted methods. In structural methods, concerns are reprensented as precedures.

The goal of separation of concerns is to focalize, separte and encapsualte the representations of concerns in a software system. Using separation of concern in software development, will give much more useful characteristics like *comprehension, reuse, maintenance, customization* [Ape07].

To build a robust platform and to build customer specific applications in short time, splitting is advantageous because of the separation of the two concerns. In order to make these two processes effective, they should interact in a way that benefits both. For example, platform design should be useful for application development and application

[1]http://www.research.ibm.com/hyperspace/workshops/icse2000/index.htm

development should aid in using the platform. This separation indicates the separation of concerns with respect to variability. Ensuring that the available variability is relevant for producing the applications is the responsibility of domain engineering. In many reusable artefacts, the platform is defined with the right amount of flexibility. Reuse of the platform and binding the variability according to the requirements of different applications makes up a large part of application engineering [PBvdL05].

2.2 Feature-Oriented Domain Analysis

Feature-Oriented Domain Analysis (FODA) is a special method for Domain Analysis. Czarnecki et al. [CE00] shows that domain analysis is the process of analyzing and creating a model of a specific domain. It consists of two activities:

- Identifiacation and definition of the domain, the scope of the domain, and the relevant stakeholders.
- Creation of the domain model that describes the vocabulary of the domain, the common and variable properties of all systems of the domain and the dependencies between these properties.

The second activity of domain analysis also contains the finding of common and variable features of all systems belonging to the domain. It focuses on the *features* of the systems of a domain. First we introduce some definitions of feature.

What Is a Feature?

Different researchers have been proposing different views of what a feature is or should be. Batory et al. define a feature as:"*an increment in program functionality*" [Bat05] and also it is a product characteristic that customers view as important in describing and distinguishing programs within a family of related programs [BEHM02]. Czarnecki et al. [CE00] define feature in Domain Engineering as:" *An end-user-visible characteristic of a system or a distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept* ".

[Ame85] [CE00] describes feature "*A prominent and user-visible aspect, quality, or characteristic or a software system or systems. For example, when a person buys an automobile, a decision must be made about which transmission feature (e.g., automatic or manual) the car will have.*"

[ALMK08] shows that "*a feature is a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement*

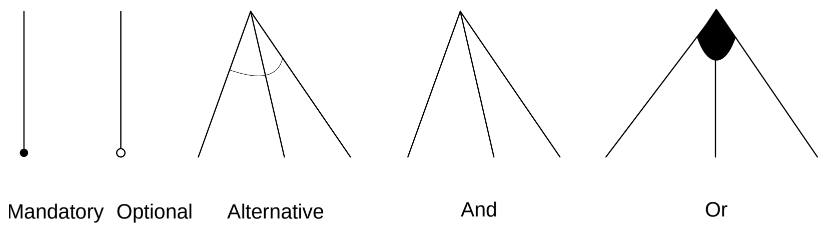

Figure 2.2: Feature Diagram Notations

and encapsulate a design decision, and to offer a configuration option." This definition provides a ground that is common to most (if not all) work on Feature-Oriented Software Development.

In this thesis in the chapter 3, we have dealed with various feature models. The feature in the feature model describes an end-user-visible characteristic of a system or a distinguishable characteristic of a concept.

2.2.1 Feature Model

Feature modeling has been applied to many numerous diverse domains like telecom systems, template libraries, network protocols and embedded systems [CHE04]. It has been proposed as a part of the Feature-Oriented Domain Analysis method. These models are independent of the implementation and are used to describe only the abstract commonalities of the code program. Common and variable features of concept instance and the dependencies between the variable features are represented by feature models [CE00]. These models are generated during feature modeling. It is the main approach to acquire and govern the common and variable features of systems in a system family or a product line. Feature diagrams, feature descriptions, binding times, priorities, stakeholders together make up feature models [CHE04].

Feature diagram

A feature diagram is a tree with the root representing a concept (e.g., a software system), and its descendent nodes are features [CHE04]. Relationships between a parent feature and its child features are categorized as: *Mandatory*, *Optional*, *Alternative*, *Or*, *And*. Common graphical notations are depicted in Figure 2.2.

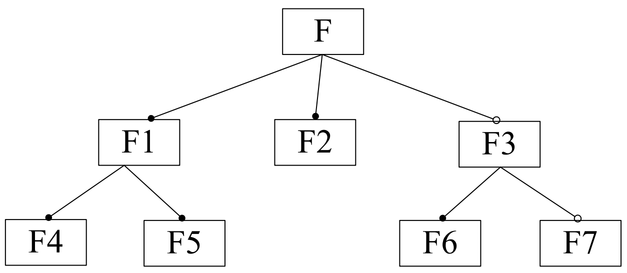

Figure 2.3: Madartory and Optional Features

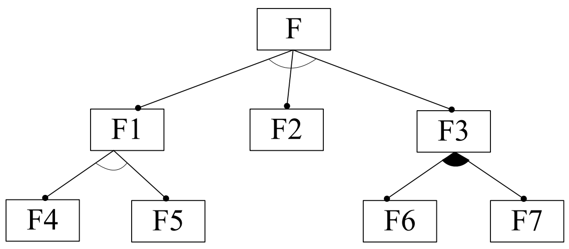

Figure 2.4: Alternative and Or Features

Mandatory Features A *mandatory feature* is included in the description of a concept instance if and only if its parent is included in the description of the instance [CE00]. It is described with an edge ending with a filled circle. It shows that this particular feature is mandatory. According to Figure 2.3, every instance description of Concept F, F1, F2, F4, F5, are always included.

Optional Features An *optional feature* is described with a edge ending with an unfilled circle. Only when its parent is included in the description, it may be included in the description of the instance. Consider Figure 2.3, F3 and F7 are optional features, F6 is a mandatory feature. If and only if F3 is included in the instance description, F6 has to be included.

Alternative Features An *alternative feature* is represented by connecting edges with an arch. So the feature consists of exactly one of its child features [CRC03]. In Figure 2.4, Feature F1, F2, F3 are alternative features and F4, F5 are also alternative features. Every time only one of the features from F1, F2, F3 could be selected; and only one of the features from F4 and F5 is selected. In Figure 2.5, there is an optional feature in the alternative features, such a set of alternative features is equivalent to the situation that all the alternative features in this set are optional [CE00].

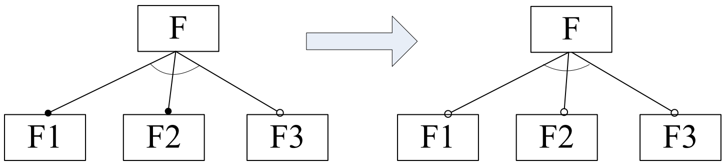

Figure 2.5: Convertion Alternative Features into another Alternative Features

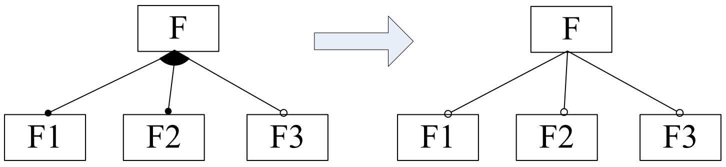

Figure 2.6: Conversion of Or Feature into Feature Diagram with all optional Features

Or Features These are set of features from which any non-empty subset can be included in the instance description, provided that their parent was also includes [CE00]. According to Figure 2.4, F6 and F7 are OR features. In Figure 2.6, there is an optional child feature in the OR feature, such a set of OR features is equivalent to the situation that all the child features are optional.

And Features A *and features* are a set of features which are included in the instance description depending on the type of each feature node [CE00]. In Figure 2.3, for concept feature F, two instance descriptions are possible. One with feature F3 and one without feature F3, while including all other features in both.

2.3 Feature-Oriented Programming

For many years researchers try to reuse software perfectly but it has proven to be a very difficult task to achieve. Object libraries can only describe the software reuse at a very low level which is be difficult for the application designer to reuse. Code encapsulation is a great problem when developing software for reuse. Object-oriented programming (OOP) can resolve this issue, however, since system feature implementation can cross cut several objects, changing something in this system features might influence other objects.

Feature Oriented Programming (FOP) refers to synthesizing programs by composing features [Bat03]. It is a criteria for developing software product lines. Consistent artifacts that define a program are systhesized when features are composed. The concept of FOP is to use algebraic techniques to specify and manipulate program designs [TBD07]. Feature oriented programming is advantageous for the following reasons: [Pre97]

- It is desirable that objects with individual services can be composed from a set of features. This will give more flexibility. Its advantage can be observed in situations where different variations of a software component are required or if new functionality needs to be incorporated into a software component frequently.
- Clearity of dependencies between features and structure is achieved by separating the core functionality from interaction handling. The benefit of this is independent reusable code by making subclasses an independent entity and not a subclass. It also benefits in making class refactoring [OJ90] much easier. This idea is same as the idea of abstract classes but in addition it also encompasses dependencies between features.
- Type dependencies between two features can occur but this can be specified within the setting stated. Parameterized features (similar to templates) also works fine with interactions and liftings.
- The simplicity of a model is achieved by considering only liftings or interactions between two features at a time. Liftings between two features could still be adequate if there are dependencies between several features by considering only auxiliary features.

2.3.1 Mixin Layers

Modularity has played an important role in the history of software design and programming languages. Modules encapsulate functionality that can be reused in other applications [SB02]. It has been experienced gradually from the small scale (functions) to the large-scale (components or packages-suites of interrelated classes) process, the reasons for simple application design and easier to bulid from fewer and larger parts [SB02]. However, researcher find that reuse opportunities become fewer as a module becomes larger. Mixin layers seem like a good solution to this problem [SB02].

The *step-wise refinement* is a very useful paradigm for a programmer to develope a complex program from a simple base program by adding what the customer needs [Dij76]. In this thesis we focus on *feature refinements*. The idea of FOP is to develop software which is composed of *features*. Such features describe the characteristic that customers views as distinguishing within a family of related programs [Gri00]. Features refine other features are increments of the program functionality that can affect multiple dispersed implementation entities (functions, classes, etc.) [SB02]. More general than traditional

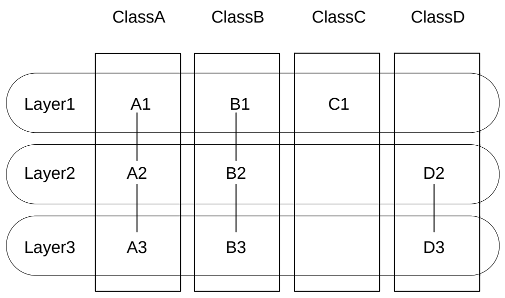

Figure 2.7: Example stack of Mixin Layer

packages that encapsulate sets of complete classes, a feature refinement can also encapsulate *fragments* of multiple classes [BSR04].

Mixin Layers encapsulate fragments of serveral different calsses so that all fragments are composed consistently, a single class is also named Mixin. Mixin Layers are an approved implementation technique for component-based layered designs. Figure 2.7 depicts a package of four classes, ClassA - ClassD, and a stack of three Mixin Layers from top to bottom defined as layer1 - layer3. A Mixin Layer consists of a set of collaborating Mixins, which cross-cuts such mutltiple classes (A-D) and implements the encapsulated fragments of the classes. The vertical lines denote class *refinement chains*. From the start layer1 also called *constants* encapsulates three classes (A1-C1). Layer2 refines two classes (A1 that is refined by A2, B1 that is refined by B2) and adds another class. That means layer2 encapsulates a cross-cut that refines class A1, B1 (represented by mixins A2, B2) and increment class D2. The same process applies to layer3 which encapusaltes layer2.

Linear refinement chains are common in this type of implementaion. We represent mixins (class refinements) as functions [BSR04]. Mixin A3 is a function that is applied to mixin A2 and mixin A2 is applied to base class A1. For function implemented in Java language, only the terminal classes (shaded circle in Fig 2.8 a) of the refinement chains could be instantiated, nonterminal classes (those that are unshaded in Fig 2.8 b) are never instantiated [BSR04].

Suppose that the Class A is the superclass of the Class B and C, and Class C is the superclass of Class D, as shown in Fig 2.8a depicts the class hierarchie. The subclass is described with the bold lines. The calss refinement stated in Figure 2.8b can also be depited using the Figure 2.7. The refinement chains that we synthesize are the same as those in Fig 2.7 expect that both class B1 and C1 are declared to be the subcalsses of the *synthesized class* [BSR04]. This inheritance implements both subclassing (the bold lines in Fig2.9b) and emulates refinement (relationships shown in thin lines). With this design scheme which refines arbitrary subclassing hierarchies by adding new classes and refining existing classes, leads us to buliding the *Jakarta Tool Suite (JTS)* [BSR04].

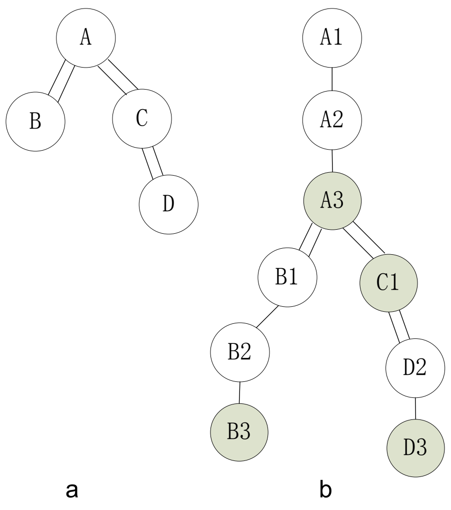

Figure 2.8: Refinement class of inheritance hierarchies

2.3.2 Feature C++

Java language has been mostly used for implementing FOP e.g. *AHEAD*. C++ despite being widely used in a large number of applications like operating systems, realtime and embedded systems, databases and middleware, is rarely considered for FOP. Currently templates [SB02], simple language extensions [SB93], or preprocessing directives are used for providing solutions. As these approaches are complicated, hard to understand and not applicable to larger software systems, therfore, Feature C++[2] has been proposed for FOP in C++.

Feature C++ is an extention of C++ which supports FOP. It uses the keyword *refines* for class refinement which has been specifically introduced in C++ for this purpose.

In Feature C++ Mixin Layers are represented by directories of the file system, they have no programmatic representation. Mixins are represented by included source files, an equation file specifies which features are required for a configuration [ALRS05].

Feature C++ offers many benifits in comparision to other programming languages. It provides solutions to different problems of the object-oriented languages like:

- The constructor problem: Minimal extentions have to be initialized even if they are not required, which causes constructor problem [EBC00] [SB00].
- Extensibility problem: occurs due to the mixture of class extensions and variations [FF98].
- Hidden Overloaded methods: in C++ hinders in step-wise refinements [ALRS05].

[2]http://wwwiti.cs.uni-magdeburg.de/iti-db/fcc/

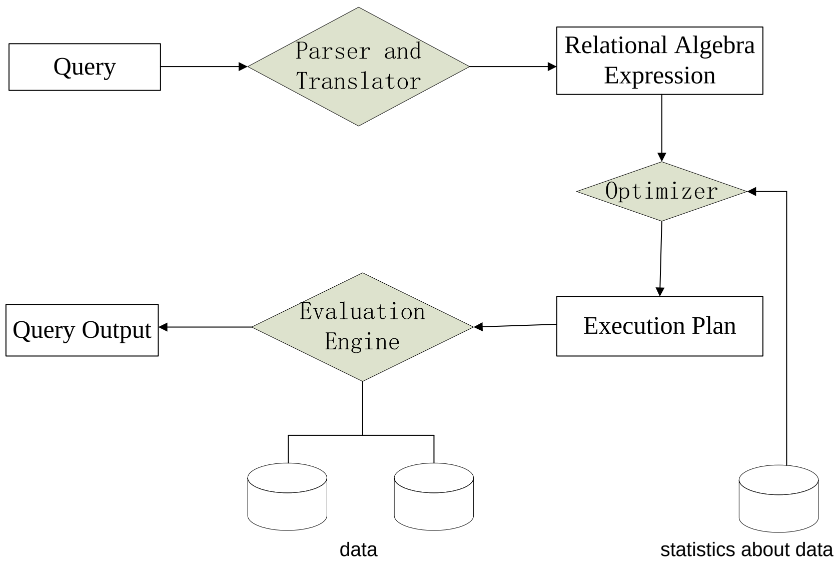

Figure 2.9: Baic Steps in Query Processing [SKS96]

2.4 Query Processing

Query processing refers to the range of activities involved in extracting data from database. These activities include translation of high level user query languages (for example, SQL) into low level data manipulation commands that can be implemented at the physical level of the file system, different query-optimizing strategies, and cost evaluations for each operation [SKS96].

Query processing and optimization have always been one of the important components of database technology. This component mainly deals with the user-desired data from an often large database and efficiently returns the results with an acceptable accuracy [YM98]. Figure 2.9 depicts the process of query engine. When the system receives a SQL query, the query processor first checks the correctness of SQL query (for example, whether the query syntax is correct, whether the relations and attributes are stored in the database, etc.) with support of SQL Parser. If the query is acceptable, then a relation algebra expression is generated, also called *initial logical query plan* [GMUW00]. Since initial logical query plan can be expressed with a large number of equivalent forms, the query optimizer is used to find only the best logical query plan. Query optimizer can be ignored by application programmer in some scenarios, for example, in the network and hierarchical models, because the Data Manipulation Language (DML) statements of such models are usually embedded in a host programming language. It makes it difficult to transform a network or hierarchical query into an equivalent form without knowing the entire application program. In contrast, relational-query languages are either declarative or algebraic. Relational query language with this form can easily generate relatively large number of equivalent plans [SKS96].

The logical query plan must be transformed into a physical execution plan. Such an execution plan contains details of each operation. For each relational operation, there are a number of methods that can be used. For example, the order of join generates different access cost. We can choose nested loop, merge sort, and other join methods according to the requirements of a stakeholder. We will explain this in more detail in Chapter 3.

2.4.1 Feature-Oriented Decompositon of SQL

SQL has grown quickly in recent years. It has been applied to different domains like sensor networks, embedded systems etc. Such applications have caused the appearance of different versions of the SQL language, like SCQL [Int99], TinySQL [MMHH05], and so on. These SQL dialects are usually separate from the standard language or included as extension packages [RKS$^+$09]. But current standards of SQL are really complex and hard to manange. But in some case like in a sensor network, it needs only a small subset of SQL, some complex SQL statements like joins, XML extensions are not required. Therefore, it is not necessary for developer to understand all the parts of SQL, he only needs to focus on the demanded part of SQL. For these reasons have caused the decompositon of standard SQL. We have applied feature-oriented programming approach based on software product line engineering which can be used to create customizable SQL parsers. Figure 2.10 [Sun07] depicts the main feature diagram of SQL:2003. It represents the basic decomposition of SQL:2003. Each of the sub-features could be further decomposed, for example, SQL foundation could be decomposed to different SQL statement classes like data manipulation statements, data definition statements, query expressions, sql transaction, etc [Sun07]. A user could arbitrary select combination of such features or packages, in order to derive a tailor-made SQL parser.

2.4.2 SQL Dialects

The current standard of SQL is much more complex and unmanageable. Because all big database vendors can provide proprietary extensions in addition to the SQL standard, These extensions have caused the appearance of different SQL dialects. For example, Structured Card Query Language (SCQL) for smart cards [Int99], TinySQL for sensor networks [MMHH05], or Stream SQL for stream processing [ZJM$^+$08]. Different users have interest in different aspects of an SQL engine like performance, the execution speed of query, the semantic of query etc., thus needing a suitable cutomizable SQL parser which can select only the needed functionality in different scenarios. In the following, we review some application domains (for example, web databases, sensor networks, and stream processing systems) where we will need different SQL dialects [RKS$^+$09].

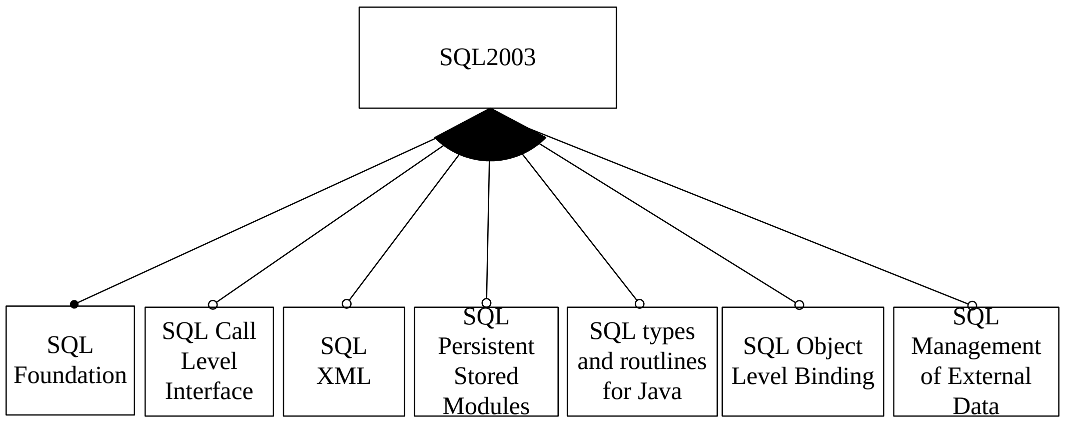
SQL2003
SQL Foundation
SQL Call Level Interface
SQL XML
SQL Persistent Stored Modules
SQL types and routlines for Java
SQL Object Level Binding
SQL Management of External Data

Figure 2.10: Main Feature Diagram of SQL:2003 [Sun07]

Web Databases Web databases are accessible from web through form-based query interfaces. Web database systems support a small subset of SQL functionality like selection-projection-join queries, aggregation, tranactions, XML etc. MySQL is an example often used for web development, despite the fact that it doesn't support nested queries, foreign keys and referential integrity, stored procedures, triggers, or views.

Sensor Networks Sensor networks also support only a small subset of SQL functionality. TinySQL [MMHH05] and Cougar [YG02] are examples used for sensor networks. Similar to the standard SQL, the query form in the TinySQL also contains select-from-where-groupby-having. It supports select, project, join and aggregation. The TinySQL dialect explicitly supports sampling intervals and windowing.

Stream Processing Stream processing is different from data processing of traditional relational database systems. In stream processing engines, data is processed as it arrives and before it is stored. Stream processing system support a stream-oriented query language. They are essentially all SQL extensions, which contain a concept of a window on a stream as a way to convert an infinite stream into a finite relation in order to apply relational operators [JMS+08] . StreamSQL or Oracle's CQL [ZJM+08] are representative work for stream processing. In these systems, they might support multistream and aggregation functions on different windows like rowing, ranging, partition and sliding windows. Since most stream processing applications apply only sliding windows, therefore, this type of window can also be provided by a simple SQL dialect [RKS+09].

In the above paragraphs three different application scenarios have been presented. The SQL dialect selected for each scenario provides only a part of functionality of the complete standard SQL. We apply the concept of domain analysis [KCH+90] and software

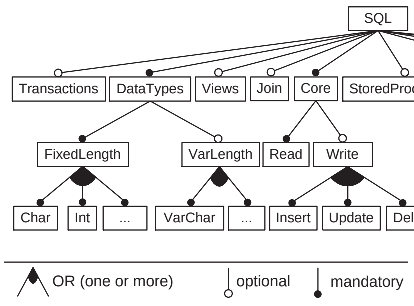

Figure 2.11: Feature diagram for a subset of SQL that supports core functionality and several extensions [RKS+09]

product line engineering [CE00] to define an initial feature diagram for a subset of SQL. Figure 2.11 [RKS+09] shows the family model of SQL that is suitable for these scenarios. This feature diagram describes the differences and commonalities of the above three scenarios. The root of the feature diagram is concept and all the other nodes are features. If the web database needs a SQL dialect, we could provide only its needed features, i.e. Data Types, Core, Transactions, Join, Aggregation, and all the Core sub-features. If the user choses the sensor networks, that means the SQL dialects could select only a small subset of SQL, like Data Types, the sub-features of Core, Temporal and Spatial features. In case of sensor networks, they do not provide important information, like sensor position (potentially aggregated), density and connectivity, system workload and network stability [YG02]. Which in turn makes the optional *Spatial* feature useless for them(sensor networks). If the user uses temporal queries, then the *Temporal* feature must be selected by the SQL dialect. At the same time, the Temporal Data Type (e.g. Data Time) must be supported by the DBMS and SQL. Such constraint will be introduced in the last section of the chapter 3. Similarly, we can describe SQL dialects for stream processing using the Core feature without write functionality and the Stream Processing feature. If user wants to represent operation on multiple streams, the Multistream feature will be provided [RKS+09].

2.4.3 A customizable SQL parser

Because of the existence of the various SQL dialects, a customizable SQL parser for differnt scenarios is desirable. SQL parser is needed for processing queries, the customizable SQL parser provides only needed functionality, such as some specific SQL statements.

Rosenmueller et al. [RKS+09] described in Figure 2.12 how various SQL parsers are generated according to SQL dialects. The SQL grammar is decomposed into small SQL sub-grammars according to the feature derived from the family of SQL dialect. Such SQL sub-grammars are composed into different grammars according to the needed SQL dialect. Composed grammars are used to create various SQL parsers with the aid of

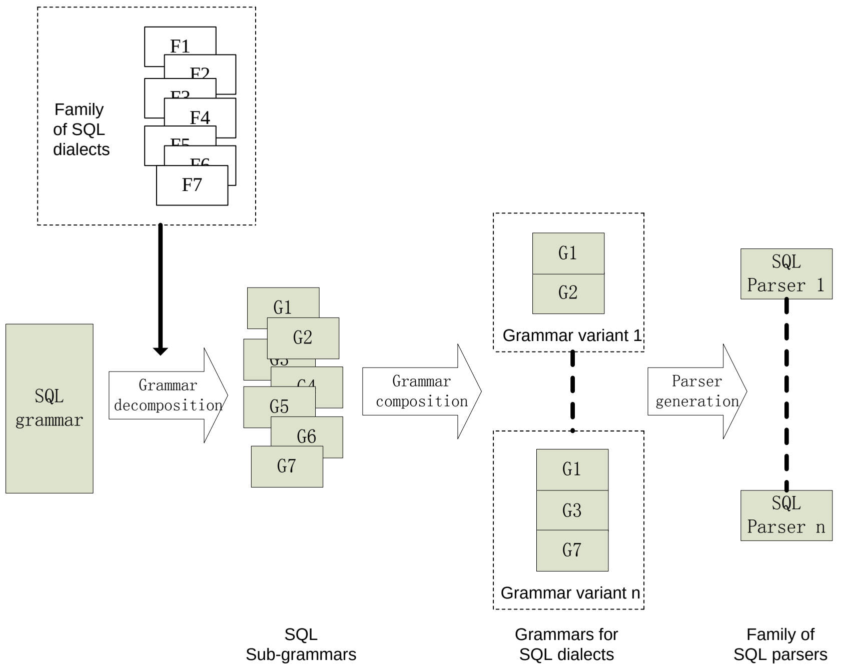

Figure 2.12: Generating a family of SQL parsers by decomposing the SQL grammar [RKS+09]

parser generators.

2.5 Summary

Software Product Line Engineering concepts are applied to query processors to design customizable query processors. The fundamental principles of software engineering are separation of concerns and stepwise development. There are two sub-processes of the Software Product Line Engineering, namely, Domain Engineering and Application Engineering. The feature-oriented domain analysis is part of the phase of Domain Engineering. We will need feature-oriented programming approach of Feature C++ in order to implement customizable query processors. We have also provided a short review of the basic process of query processor. In order to implement a customizable query processor, we must first obtain a customizable SQL parser. Implement a feature-oriented SQL parser based on the feature-oriented decomposition of SQL. This feature-oriented decomposition of SQL consider about serval different SQL dialects.

Chapter 3

Feature-Oriented Design of Customizable Query Processors

The scope of the thesis is to modeling the features of a query processor for FAME-DBMS, a customizable DBMS for embedded devices. We describe constraints between the feature models of FAME-DBMS, the customizable Query engine and the feature-oriented SQL parser. We give an example of how the features can be used for implementing a product line of query processors. The complete implementation of various features is beyond the scope of this thesis. In this chapter we introduce various feature models like SQL Engine, FAME-DBMS and so on. These feature models are used to describe only the abstract commonalities and variabilities of different software produline line.

Basis for Modelling Features for Query Processors A complete feature model consists of a feature diagram and other additional pieces of information, including semantic description, rationale, exemplar systems, constraints and default dependency rules, availability sites, and so on [CE00]. But we only consider about the semantic description and constraints as follows:

- **Semantic description** Each feature should contain a short description describing its detail semantics. The developer can quickly undstand what the feature means from such information. The semantic description includes different models in appropriate formalisms (e.g. an interaction diagram, pseudocode, equations, and so on [CE00]). In this thesis, we only give small description of the feature. Some semantic description may use interaction diagram to express the feature more elaborately.

- **Constraints** Feature diagrams include not only variable features but also dependencies between them. These dependencies are expressed in the form of constraints.

Constraints allow us to establish an automatic configuration. In this thesis, we have used a labeled arc to describe a ‘require‘ condition in a feature diagram.

3.1 Feature Diagrams for FAME-DBMS, SQL and SQL Engine

Developing a complete DBMS for different small devices like cellphones, sensors, smart-cards, PDAs, etc [SR98] is often not possible. Developing a special purpose database system specifically tailored to a specific application scenario like web databases, sensor networks, stream processing etc. is both desirable and easy to achieve. When various SQL dialects are considered for differnt application scenarios, a suitable underlying DBMS must support the special functionality required by the SQL dialect. For example, in the sensor network scenario, if a SQL dialect includes the Temporal feature, the DBMS must provides the corresponding Data Type, e.g. Date Time. Such changes may affect several layers of the DBMS. In the last section we will introduce constraints of different features in the feature models of FAME-DBMS, Query Engine and SQL.

The FAME-DBMS (Family of Embedded DataBase Management Systems)[1] *project explores techniques to implement highly customizable data management solutions, and illustrates how such systems can be created with a software product line approach. With this approach a concrete instance of a DBMS is derived by composing features of the DBMS product line that are needed for a specific application scenario.* The feature model of the FAME-DBMS prototype is depicted in Figure 3.1[2]. This prototype of FAME-DBMS is developed using the experiences of the decomposition of Berkeley DB and a prototypical Storage Manager product line. It contains the core functionality OS-Abstraction, Buffer Manager, Storage, and Access. Each of these features can be decomposed into more sub-features according to the needs of different applications. For example, in widely used navigation systems in cars, the user only needs a simple select statement, other SQL statements like insert, update, delete, are not required [RKS$^+$09]. For such a case, in the Access layer of FAME-DBMS, the Get feature is only allowed and omit the functionality of Put, Remove and Update. When implementing a number of similar DBMS, these features could be often reused by the developer. This approach offers many benefits, like descreasing development costs, easy to maintain, faster development of the product and shipment to markt and increasing software quality. We argue that FAME-DBMS is highly suitable for various embedded systems. It supports a very fine granularity of features and variability that are required in such constrained environments.

[1]http://fame-dbms.org/

[2]http://fame-dbms.org/prototype.shtml

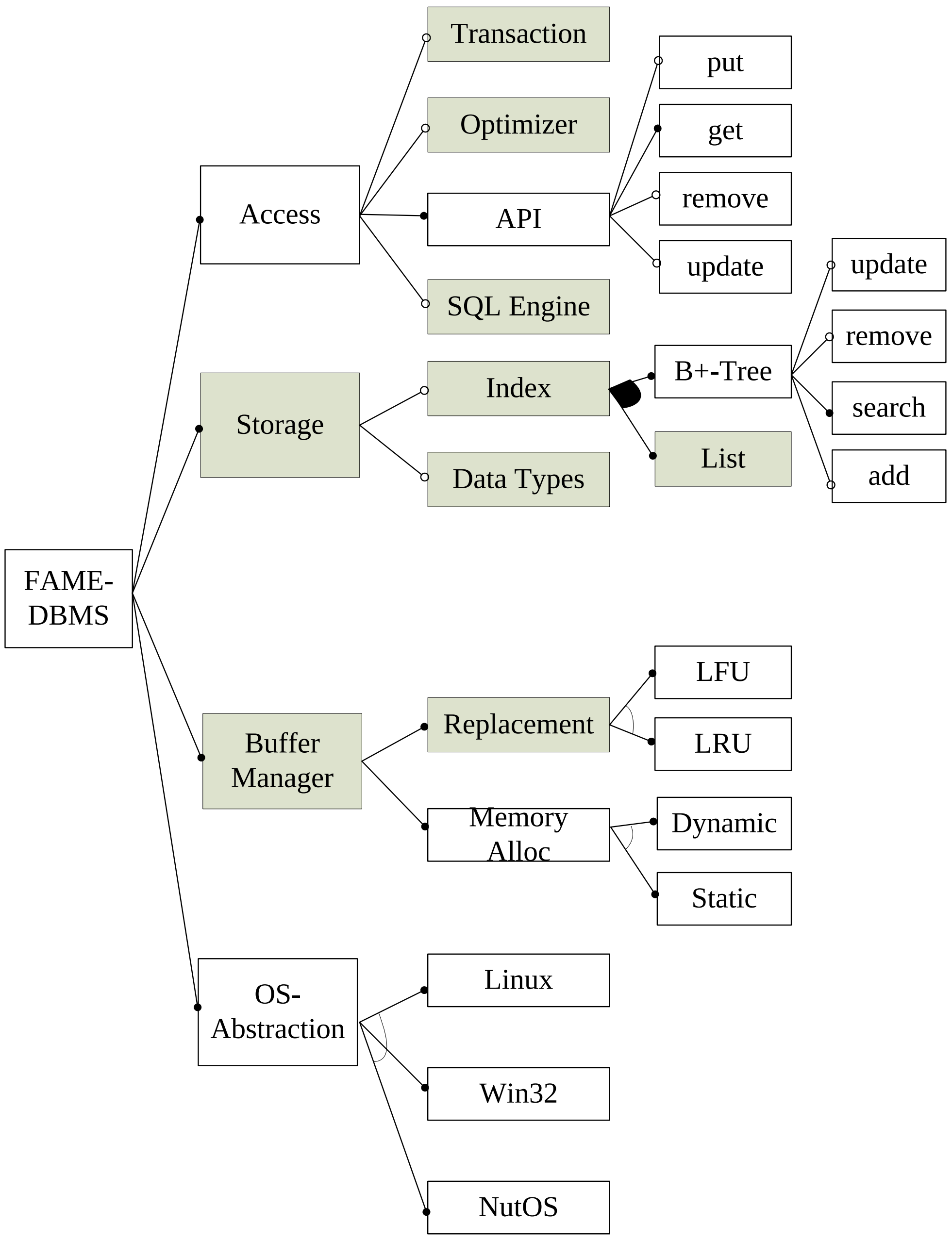

Figure 3.1: Feature diagram of FAME-DBMS [RSS+08]

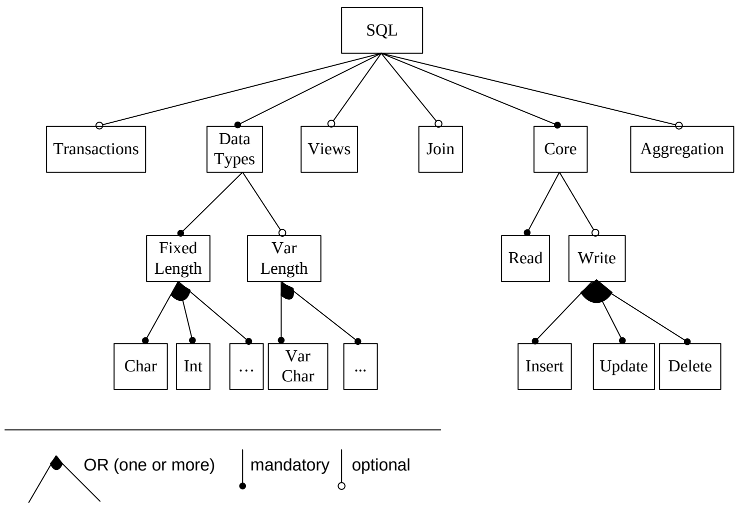

Figure 3.2: Feature diagram for SQL subset extracted from Figure 2.11

3.2 A Model for a Custom Query Engine

For developing a customizable and extensible DBMS, query processing plays an important role in it. For example, datawarehouse analytics applications require multi-dimension queries; streaming applications may require some form of stream SQL; web database applications may require XML supports and so on. As we know, there is not even a single query language that supports all the functionality needed by various embedded application scenarios. In order to satisfy different requirements of the users, we use the novel software engineering techniques, like software product lines and feature-oriented programming to develop customizable query engines. We use the software product line approach to reduce the complexity of building different solutions and enhance the maintainability of the components.

A customizable query processor cannot provide the complete functionality of the whole SQL family. It can only address some parts of the SQL [RKS+09]. In this thesis, we only provide a small part of the SQL functionality corresponding to the subset of SQL shown in Figure 3.2. This SQL subset support the *Core* feature which represent basic functionality and some basic data types like char, int and so on. The other features defined as optional features. These features are selected only according to the user's requirement. For example, the optional feature *Join* is not always needed. In the initial model of FAME-DBMS only one table is supported, join operation is not necessary for this case. But FAME-DBMS could support more tables after extended, in this way, SQL should support the feature of *Join*. We will also explain the constraints related to these optional features in the last section. In Figure 3.3, we show an initial feature diagram on the basis of differences and commonalities of the family of SQL. In the following, we

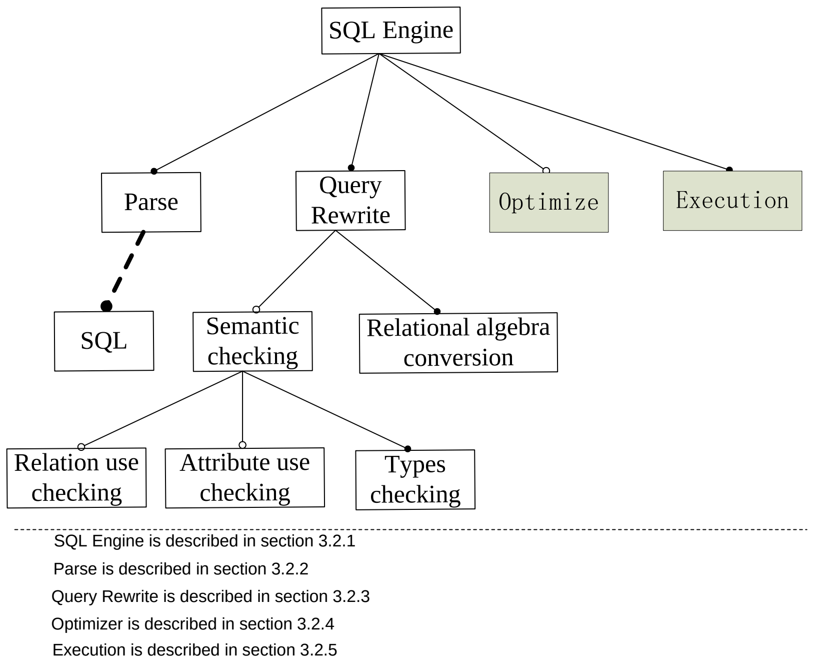
SQL Engine
Parse
Query Rewrite
Optimize
Execution
SQL
Semantic checking
Relational algebra conversion
Relation use checking
Attribute use checking
Types checking
SQL Engine is described in section 3.2.1
Parse is described in section 3.2.2
Query Rewrite is described in section 3.2.3
Optimizer is described in section 3.2.4
Execution is described in section 3.2.5

Figure 3.3: Feature diagram for the SQL engine

discribe the features shown in Figure 3.3.

3.2.1 SQL Engine

Semantic Description

Figure 3.3 describes the main feature diagram of SQL engine. It also shows the most coarse-grained decomposition. The whole SQL Engine include four main features : mandatory features *Parse, Query Rewrite, Execution,* and optional feature *Optimize.* The shadowed feature can be decomposed further. Before query processing can begin, the query engine must first parse the query, which builds a tree structure from the textual form of the query. Then the query rewrite performs semantic checks on the query. It also performs some tree transformations to turn the parse tree into a tree of algebraic operators representing the initial query plan. After this, the query optimizer transforms the initial query plan into the best available sequence of operations on the actual data. In the end, the execution engine executes each step of the generated query plan.

3.2.2 SQL Parser

This feature references the work from Sunkle Sagar [Sun07]. Since the whole standard SQL is very complex. There is quite a large number of SQL dialects, therefore, Sunkle proposed to decompose the standard SQL. This SQL Engine process depends on the

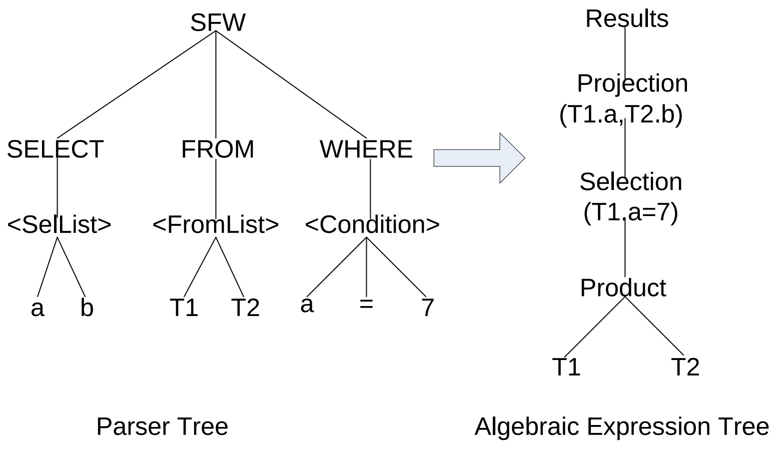

Figure 3.4: Translation Of a Parse Tree to An Algebraic Expression Tree

features selected for this dialect. We have already talk about this SQL dialect in section 2.4.2. In the Appendix A, we refer a sub-feature diagram of SQL 2003 to show how the simple SELECT-FROM-WHERE forms can be parsed.

3.2.3 Query Rewrite

This feature includes an optional feature *Semantic checking* and mandatory feature *Relational algebra conversion.* The main function of the Query Rewrite is to translate the parse tree which was generated by the SQL Parser into an initial query plan. Semantic checking is not required if the SQL is well-formed before it is processed. (In fact, even if the query is valid syntactically, it may violate one or more semantic rules on the use of names). So we use the Semantic checking feature in order to generate a valid parse tree. If the parse tree is not valid, then an appropriate diagnostic is issued, and no further processing occures.

Types checking

Types checking is a mandatory feature. All attributes must be of a type appropriate to their uses. Likewise, operators are checked to see that they apply to values of appropriate and compatible types. For instance, if the SQL dialect is applied in the sensor networks, it provides temporal functionality, which further require a DATETIME data type. The *Types checking* feature must ensure whether this data type exists or not.

Relation use checking

It is an optional feature. Every relation mentioned in a FROM-clause must be a relation or view in the schema against which the query is executed.

Attribute use checking

It is an optional feature. Every attribute that is mentioned in the SELECT- or WHERE-clause must be an attribute of some relation in the current scope. Although the SQL query statisfies valid syntax, but it actually may violate some semantic rules on the use of names. For example, one simple SQL query like SELECT A FROM B WHERE A = 7 ; We can find that the syntax of this query is valid, but if the attribute A belongs not to relation B, this will cause the error. If the user have more experiences with the database, he may avoid to make this mistakes, so we define both Relation use checking feature and Attribute use checking feature are optional features.

Relational algebra conversion

It is a mandatory feature. It transforms SQL parse trees to algebraic logical query plans. For instance, it converts a simple SELECT-FROM-WHERE (SFW) construct to relational algebra. If we have a < Query > that is a < SFW > construct, and the <Condition> in this construct has no subqueries, then we can translate the constuct with select-list, from-list, and condition to a relational-algebra expression, from bottom to top. For example, if user gives a SQL query like:
SELECT T1.a, T2.b
FROM T1, T2
WHERE a = 7
In Figure 3.4 describe how translate of a parse tree to an algebraic expression tree.

- We first translate the < Fromlist > with product of all the relations.
- Then the < Condition > expression in the construct being replaced with a selection σ.
- At the last, the < SelList > in the construct being replaced with a projection π.

3.2.4 Query Optimizer

Semantic Description

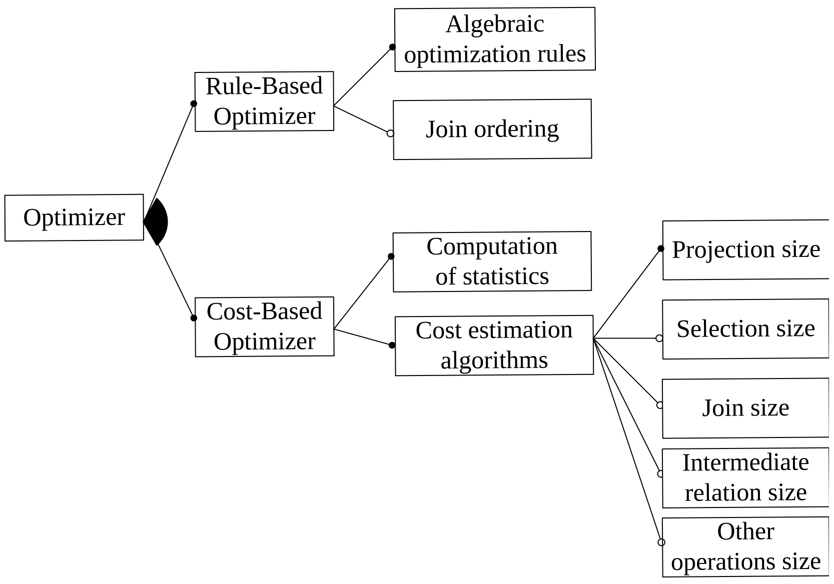

Figure 3.5: Feature Diagram Of the Optimizer

The Figure 3.5 describes the main feature diagram of SQL Optimizer. SQL Optimizer is a useful part of the SQL processing. It can have a huge impact on the speed of the SQL execution. In Figure 3.3 we have defined the feature *Optimizer* as an optional feature. Because in some cases, the speed of the SQL execution might not be an issue for the user. Without the SQL optimizer, the query processing can still works. Feature diagram of Optimizer includes two mandatory features : *Rule-Based Optimizer - RBO* and *Cost-Based Optimizer - CBO*. Both the RBO and CBO have benefits and it is up to the user to tune each SQL query using the proper optimizer. So we define these two mandatory features as Or features. This means that the user can choose both of these two methods or anyone of them.

RBO

It is very elegant for its simplicity and often made faster execution choices than the CBO. Because the cost of CBO is very expensive, many systems use RBO to reduce the number of choices that must be made in a cost-based fashion. RBO uses a heuristic method to select among serveral alternative access paths with the help of some algebraic optimization rules. All possible paths were ranked and chosened the lowest one [3]. RBO feature includes two sub-features, one mandatory feature *Algebraic optimization rules* and one optional features *Join ordering*.

Algebraic optimization rules

[3]http://www.lc.leidenuniv.nl/awcourse/oracle/server.920/a96533/rbo.htm

This feature listed here reorder an initial query-tree representation such that the operations that reduce the size of intermediate results are applied first. It contains many different equivalence transformation rules. For example, selection operations are commutative, conjunctive selection operations can be composed into a sequence of individual selections. More detailed rules are shown in the Appendix B. With the aid of these rules we can use the heuristic optimization to reduce the cost of execution. Such as performing selection and projection operations as early as possible. Because early selection can reduce the number of tuples and early projection can reduce the number of attributes.

Join ordering

In some cases, the user does not use the join operation, so we define the *Join ordering* feature as an optional. As we know, a good ordering of join operators is important for reducing the size of temporary results.

CBO

It determines which execution plan is most efficient by considering available access paths and by factoring in information based on statistics for tables or indexes accessed by the SQL statement. The CBO performs the following steps[4]:

- The optimizer generates a set of potential plans for the SQL statement, based on available access paths.
- The optimizer estimates the cost of each plan, based on statistics in the data dictionary for the data distribution and storage characteristics of the tables, indexes, and partitions accessed by the statement. The **cost** is an estimated value proportional to the expected resource use needed to execute the statement with a particular plan. The optimizer calculates the cost of access paths and join orders, based on the estimated computer resources, including I/O, CPU, and memory.
- The optimizer compares the costs of the plans and chooses the one with the lowest cost.

Computation of statistics

[4]http://www.lorentzcenter.nl/awcourse/oracle/server.920/a96533/optimops.htm

It is a mandatory feature. The CBO uses statistics that are collected from the table in the DBMS. Query engine uses these metrics, such as number of rows and number of rows per block, average row length, total number of database blocks in a table and so on, in order to intelligently determine the most efficient way of servicing the SQL query. As we know, the database is not static. It will always changes through data input and data modifiaction. These changes cause current statistics to become inaccurate and generate the "best" plan for the SQL engine that actually is not the accurate best plan. So query optimizers prefer the periodic computation of statistics because these statistics tend not to change radically in a short time [GMUW00].

Cost estimation algorithms

CBO uses cost estimation algorithms to estimate the cost of each physical plan. After calculating the cost, CBO selects the physical query plan with the lowest cost. The selected plan is then forwarded to the query execution engine. There is an algorithm corresponding to every operation in the logical plan.

Selection size

It evaluates different selection operations and picks the cheapest cost.

Projection size

It evaluates different projection operations like sort-based projection or hash-based projection and chooses the better one. We define projection size are mandatory feature. That is becaue we consider about for a very simple SQL query may only contains SELECT statement and FROM statement, without WHERE statement. In this case, FROM statement can also contain only one table.

Join size

It decides which join algorithm like sort merge join, hash join or nested loop join should be used.

Intermediate relation size

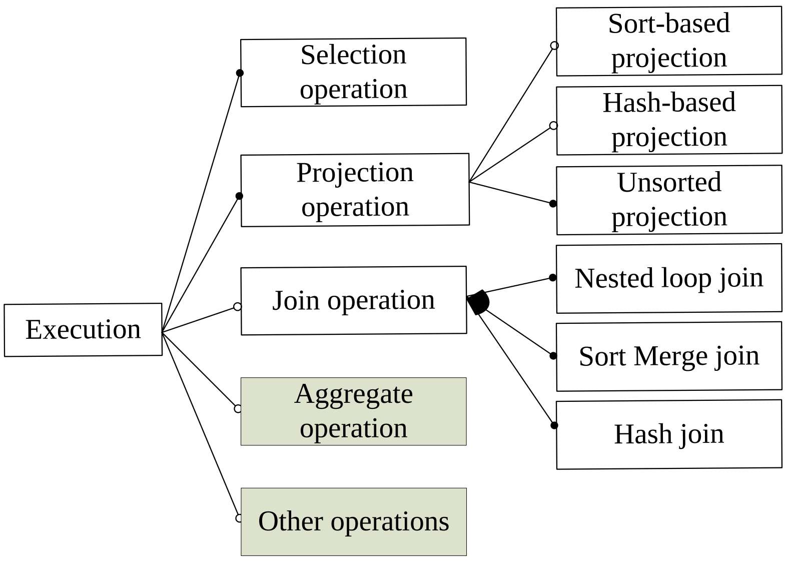

Figure 3.6: Feature Diagram Of the Execution

Knowledge of how the intermediate relations are represented is known when expression of the logical plan involves several operators. Relations which are arguments of an expression can be stored in many different ways like clustered or unclustered, indexed or unindexed. Relations computed during query execution can be stored on a disk in a cluster occupying as few blocks as possible [GMUW00].

Other operations size

It evaluates other operations like aggregate operation, set operation and so on and chooses the lowest cost operation.

3.2.5 Query Execution

Semantic Description

The Figure 3.6 describes the feature diagram of SQL execution. It is the responsibility of the execution engine to execute each step of the chosen query plan. Interaction between execution engine and most of the components of DBMS takes place using buffers or directly [GMUW00].

Selection operation

The selection σC(R) takes a relation R and a condition C. The condition C may involve [GMUW00]:

- Arithmetic (e.g. +, -, *) or string operations (e.g. concatenation or LIKE) on constants and/or attributes.
- Comparisons between these arithmetic, e.g. x > y or x - y = 1 and
- Boolean connectives AND, OR and NOT applied to the terms constructed in step 2.

In query processing, *file scan* is the lowest level operation performed by query proceesing to access data. File scan can be categorized as search algorithm. All records fulfilling a selection condition are located and retrieved by file scan [SKS96].

Projection operation

Projection helps in deleting unwanted columns after a relation has been scanned. However, the situation gets complicated due to the presence of DISTINCT directive. The problem arise because of the presence of duplicate tuples as a result of the projection operation. These duplicate tuples must be eliminated. Sorting and hashing [KBL05], the two sub-features of projection operation can be used for finding identical tuples.

Sort-based projection

It is also used for scanning the orignal relation. It removes the tuple components that are to be projected out, sorts the resulting tuples and writes the result back to disk. The result can then again be scanned and duplicate tuples can easily be removed since they are right next to each other [KBL05].

Hash-based projection

Hash-based projection is another way to quickly identify the duplicates using a hash function. In this thesis, we will not go into details of the Hash-based projection.

Join operation

Join operation is an optional feature. In the design of FAME-DBMS, there is only one table, therefore, join operation is not needed. Nevertheless, it is a very useful operation in relation algebra and is used to join information from two or more tables. The size of the resulting data obtained after applying join operation is two times the size of the input. Although in comparison to exponential complexity of some algorithms, quadratic complexity of join operation seems quite acceptable. But considering the large amount of data and the relatively slow disk I/O operation, quadratic complexity is prohibitive in database query evaluation. There are three main methods for computing joins: Nested loop join, Sort merge join and Hash join. We define them as sub-features of Join operation [KBL05].

Nested loop join

Nested loop join is useful when small subsets of data are being joined and if the join condition is an efficient way of accessing the second table. It is very important to ensure that the inner table is driven from (dependent on) the outer table. If the inner table's access path is independent of the outer table, then the same rows are retrieved for every iteration of the outer loop, degrading performance considerably.

Sort Merge join

Sort merge join is used to join tuples from two independent sources. Generally, the performance of hash join is better than sort merge join, but in some cases sort merge join can perform better than hash join. This happens if both of the following conditions exists:

- The tuples have already been sorted.
- The join condition between two tables is an inequality condition (but not a nonequality) like <, <=, >, or >=.

For large datasets, sort merge join performs better than nested loop join. You cannot use hash join unless there is an equality condition.

Hash join

Hash join is used for joining large datasets. A hash table on the join key is built by the optimizer in memory using the smaller of the two tables or data sources. It then scans the large table, seraching the hash table to find the joined rows. This method works well when the smaller table fits completely in the available memory, reducing the cost to a single read pass over the data for the two tables.

Aggregate operation

Apart from retrieving data, computation or summarization on data could also be performed. This computation can be performed using arithmetic expressions like MIN and SUM provided by SQL. These features represent a significant extension of relational algebra [Ram97].

Other operations

Other relational operations such as set operations (union, intersection and set-difference operations) can be implemented according to customer needs.

3.3 Constraints Between Features

We express constraints between different features or feature models with *Requires* or *Excludes* [CE00]. In our feature models we present in all of the feature diagrams not with *Excludes* constraint. It does not mean that exlusion conditions do not exist at all but at least not exist in the feature modeling sense and at the *feature constraints* level. The exclusion constraints are present in the form of what we have called implementation constraints. At the statement level and in terms of *feature constraints*, it is the *Requires* constraints that are more prominent than *Excludes* constraints and therefore only *Requires* constraints are presented. In the following, table 3.1 describe constraints between the feature models FAME-DBMS, SQL, and SQL Engine. We only consider domain dependencies and talk about implementation description in Chapter 4. In table 3.1, we describe two columns with Constraint and Constraint description. Constraint describe the constraints between different features. Constraint description explains why we use this constraints. In this table, the constraint type of all the constrains use the *Require*. We use constraints from upper layers to lower layers, because if we configure upper layers, the lower layers should be configured automatically.

3.4 Summary

In this chapter introduced the basic idea of how the customizable query processors design. We first apply the domain engineering approach to analyze the family of query processor. Then we used feature-oriented approach to design a basic feature modelling of query processor (Figure 3.3). In order to find the dependencies between differnt software product lines, we are reviewed short about the software product line FAME-DBMS (Figure 3.1). At the same time, we abstract a simple SQL feature diagram (Figure 3.2) to

Constraint	Description
SQL.Data Types (Figure 3.2) ⇒ FAME-DBMS.Data Types (Figure 3.1)	Without Data Types of FAME-DBMS, the Data Types of SQL can not work
SQL.Transaction (Figure 3.2) ⇒ FAME-DBMS.Transaction (Figure 3.1)	Without Transaction of FAME-DBMS, the Transaction of SQL can not work
SQL.Insert (Figure 3.2) ⇒ FAME-DBMS.Put (Figure 3.1)	Without FAME-DBMS API-Put feature, the Insert of SQL can not work(* This feature will cause feature derivatives problem, we will discuss in chapter 4)
SQL.Delete (Figure 3.2) ⇒ FAME-DBMS.Remove	Without FAME-DBMS API-Remove feature, the Delete of SQL can not work(* This feature will cause feature derivatives problem, we will discuss in chapter 4)
SQL.Update (Figure 3.2) ⇒ FAME-DBMS.Update Figure 3.1)	Without FAME-DBMS API-Update feature, the Update of SQL can not work(* This feature will cause feature derivatives problem, we will discuss in chapter 4)
SQL.Join (Figure 3.2) ⇒ SQL Engine.Execution.Join operation (Figure 3.6)	Without Join operation feature of Execution, the Join functionality of SQL can not work
SQL.Aggregation (Figure 3.2) ⇒ SQL Engine.Execution.Aggregate operation (Figure 3.3)	Without Aggregate operation feature of Execution, the Aggregate functionality of SQL can not work
SQL Engine.Query Rewrite.Semantic Checking.Types checking(Figure 3.3) ⇒ FAME-DBMS.Data Types (Figure 3.1)	Without Data Types feature of FAME-DBMS, the Types checking of Query Rewrite can not work. For example, SQL query have a where condition A=10, the DBMS must support the Int Datatype
SQL Engine.Optimizer.Computation of Statistics (Figure 3.5) ⇒ FAME-DBMS.Data dictionary (Figure 3.1)	Without Data dictionary feature of FAME-DBMS, the Computation of Statistics of CBO can not work
SQL Engine.Execution.Sort-based projection (Figure 3.6) ⇒ FAME-DBMS.Index (Figure 3.1)	Without Index feature of FAME-DBMS, the Sort-based Projection of Execution can not work
SQL Engine.Execution.Hash-based projection (Figure 3.6) ⇒ FAME-DBMS.Index (Figure 3.1)	Without Index feature of FAME-DBMS, the Hash-based Projection of Execution can not work

Table 3.1: Constraint of Features

meet the software product line query engine. We are only provided a quite coarse-grained decomposition of query engine, it contains only the basic functionaly, more fine-grained decomposition maybe implement in the future. In order to implement automatic configuration of SPL, we describe the constraints between variable features. These constraints specify which feature combinations are valid or invalid. In Chapter 4 we introduce some feature implementation detail.

Chapter 4

Implementation Detail

After the above discussion about our design, in this chapter we will give a more detailed discussion of how our system works and how it is implemented. We developed a prototype to demonstrate how feature oriented programming can be used to generate a highly customizable query processor. The implementation described here is based on our test implementation of query processor, which is mainly based on query parser, rewrite and execution modules. FeatureC++ is used for implementation and FAME-DBMS is used as database, which is highly customizable embedded DBMS.

Automatic Generation of Customizable Query Processors

In the subsection 2.4.3 we have introduced how a family of SQL parsers can be generated by decomposing the SQL grammar. Similarly, we describe in Figure 4.1 to explain how customizable query processors are generated according to the family of SQL parsers. As we can see from Figure 4.1, the SQL sub-grammars which are obtained by decomposing SQL grammar and are composed into different grammars according to the needed SQL dialect. Composed grammars are used to create various SQL parsers. These different parsers with features which is selected from the family of query engines are used to generate the various query engines according to different application scenarios. For example in our implementation, we developed a query engine with parsing, rewrite and execution features. In which, rewrite feature is optional and can be removed from query engine. We can extend the functionality of query engine by including more optional features like optimization.

4.1 Characteristics of FAME-DBMS

In order to implement the query processor, we first review some characteristic of FAME-DBMS. Because SQL Engine is the feature of the FAME-DBMS. In chapter 3 we have

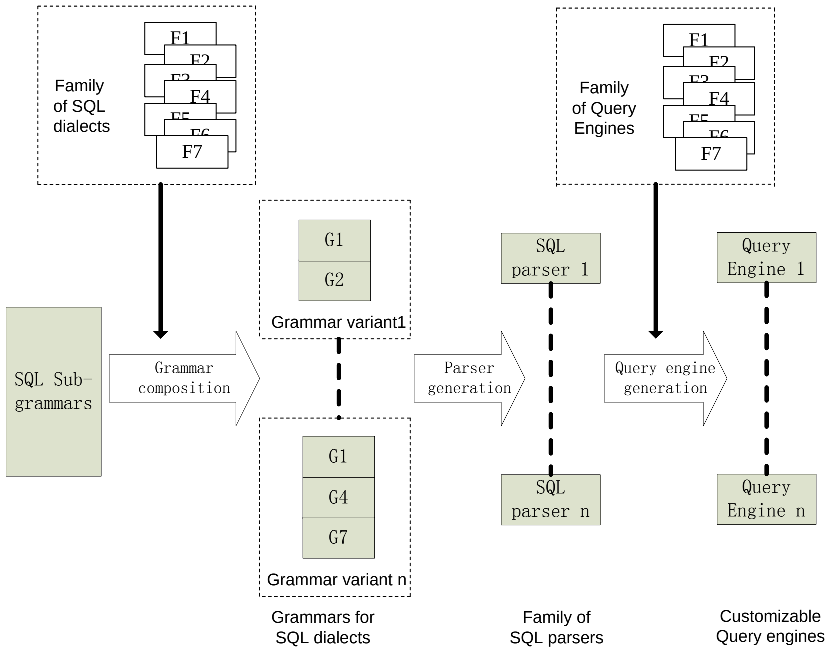

Figure 4.1: Generating Customizable Query Processors (adopted from Figure 2.12) [RKS+09]

introduced some constraints between SQL Engine and FAME-DBMS. In the following we give the main features in the FAME-DBMS, such feature have closely related to the implementation of customizable SQL Engine.

4.1.1 Feature: DataAndAccessManager

This mandatory feature that realizes the API for FAME-DBMS. This API include functions like PutData(), GetData() and Delete().

Class DataAndAccessManager

DataAndAccessManager class contains the functionality of the API. The purpose of keeping the API separate was to make sure, changes to internal interfaces have less impact on the external API and we make sure that API is consistent for different versions and builds of FAME-DBMS. Class details shows below:

bool GetData(RECORD r) Get the data from the database as per the key provided in the RECORD structure as argument. This method will be used by end user to retrieve data from database. This method simply forwards this call to storage manager.

Class DataDictionary

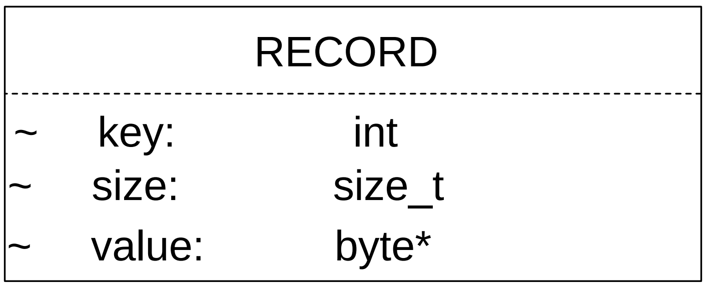

Figure 4.2: Class Record

DataDictionary class manages the FAME-DBMS Meta-Data information. This class is responsible to load Meta-Data information when database is opened and this class saves the Meta-Data information when database is closed.

bool PutData(RECORD r) Put the data to the database as per the tuple provided in the RECORD structure as argument. This method will be used by end user to insert data to database. This method simply forwards this call to storage manager.

bool Delete(RECORD r) Delete the data from the database as per the key provided in the RECORD structure as argument. This method will be used by end user to delete data from database. This method simply forwards this call to storage manager.

Class-RECORD

Record is used to store each tuple in FAME-DBMS. It is also exposed through API. End-user gives records in the form of RECORD structure to FAME-DBMS. Figure 4.2 define the class Record.

4.1.2 Different Implementation Variants for FAME-DBMS

For the initial FAME-DBMS, it supports no tables. Considering about the limited resource on the devices (e.g., memory), a very simple query processor is also needed. In Figure 4.2 we have clearly know the class of record. All data in the FAME-DBMS express with <key, value>. In this case, the FAME-DBMS contains no columns and also no tables. If we implement query engine for this case, we can do the query like this:

SELECT <columns>
FROM <table>

WHERE <condition>

The SELECT <columns> in this case can only query SELECT KEY and SELECT *, in fact the * meaning here only the key, value. because the FAME-DBMS contains no columns. And searching for values does not make sense since these are without any type, the values is only byte array. If the query have only SELECT * without FROM and WHERE sentences, the output of query print all data in the FAME-DBMS. When the query engine process FROM <table> sentence, the table could be mapped directly to a database in FAME-DBMS. If we define a name of database, we could use the name as the table name for the FROM sentence. But also the FROM sentence could be ignored since there are actullay no tables in FAME-DBMS. The WHERE <condition> in the initial FAME-DBMS can query WHERE key = "condition". In general, in our implementaion we predefine the key and value as columns. Key can also be used within WHERE clause. For example, we support a simple SQL query like SELECT key, value WHERE key = 10 + (10-5) ;

4.2 Query Engine Implementation

In our implementation we developed a hightly customizable query engine prototype. The main components of query engine are Parser, Query Rewriter and Query Execution components. In figure 3.3 optimization also exist as optional feature. However, it is currently not part of this prototype implementation.

Since FAME-DBMS which we used at back end for testing our prototype, currently only support key/value pair, we restricted our implementaion to available functionality. However, we laid down an architecture that is highly customizable and can be extended to support full fledged DBMS query execution.

To support high customizability, we used feature oriented programming approach. Sunkle [Sun07] decomposed the SQL based on feature and then realized those features in our prototype using FOP. We used C++ for our prototype development. C++ give benefit of high performance at the same time giving the benefits of object oriented programming. We used Feature C++ for implementing our prototype. To verify the functionality of our prototype, executed some test queries and verified the functionality of the query engine.

The functionality of the prototype is in Figureas 4.3 as follow:

We provide the query as input to query engine which is first processed by Parser. Parser after processing the query generates the *Abstract Syntax Tree (AST)* for the query as output. The resulted AST is submitted to Query Rewrite component. Query Rewrite component converts AST into *Relational Algebra Tree (RAT)*, however Query Rewrite

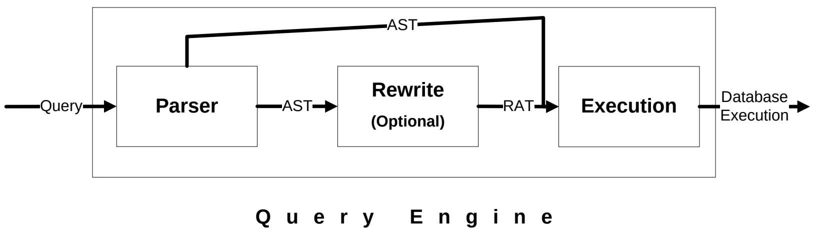

Figure 4.3: Process of Query Engine Implementation

in our prototype implement as an optional component and can be removed from the query engine as needed. Execution component is able to execute the AST as well as RAT based on the seleted components. If we do not have the feature Query Rewrite, the query execution execute the operation based directly on AST. If we provide the feature Query Rewrite, the query execution will execute the operation based on RAT. Once we provide AST or RAT to Execution component it performs the execution on the database. Query execution traverses the AST or RAT. For each type of node it performs the execution task for that node. For example, for a CREATE query, first node encountered is Keyword node in case of AST based execution. Once execution component reads that it is CREATE node, it knows that it should contain the child with table name. It reads the child for table name and then execute the appropriate method calls of FAME-DBMS.

4.2.1 Parsing the SQL Query

Parser is one of the most important components in the query engine. It checks for correct syntax of query and builds a data stucture (in our implementation this data structure has been described as Abstract Syntax Tree (AST)) to describe the input tokens. Parsers include hand-writen parser and Parser generators like ANTLR, Yacc and so on. The parser generators often use a grammar written in BNF form.

It is not always possible or desirable to use a Generated Parser. In our implementation the languages grammer is not very complex, therefore, the development cost of writing a parser by hand is lower than that of using a parser generator. Generated Parsers also have an inherent problem that they lack extensibility mechanism. This might become an issue if the semantics of the language demands extensibility. It is also difficult to provide extensible grammars and sophisticated error reporting with Generated Parsers. In our case, the simplicity of our implementation allows us to ignore the use of generated parsers [JDJ04]. Another important thing is that there is currently no working solution based on features as described by Sunkle [SKS+08]. Our implementation is based an AST which fulfills many of our needs, therefore, it was important for us to build our own parsing engine, as AST are not completable with existing parsers are built for complete

SQL while we are only concerned with very specifiv elements.

The customizable query engine is based on a customizable parser which was provided as a starting point for this thesis. The parser is manually implemented with FeatureC++ and supports only a small subset of SQL. This Parser components contain scanning functionality that scans the query for delimiters, whitespaces and keywords etc. and generates the tokens for query. These tokens are then used to generate the AST for the query. At present, this feature oriented parser supports only features SELECT statement and INSERT statement. This feature oriented parser can in future be extended to parser more statements like, DELETE, UPDATE etc. The features we want to use have been extracted from sunkle thesis [Sun07]. In our implementation we focus only an SELECT statement.

Design of Abstract Syntax Tree

The SQL parser generates an AST by passing the input query. The nodes of the AST contains tokens. The importent syntactical elements of the query are saved in the nodes of the AST. In other words, AST is the internal representation of a query in our system.

The benefit of using AST is that all the necessary and important parts of the input query are saved in the nodes of the AST for further processing while it also helps in omitting the unnecessary systactic details. The difference between the AST and concrete syntax trees is that AST omities the tree nodes which represents punctuation marks such as semi-colon to terminate statements or commas to separate funetion arguments. Also in AST there are no tree nodes for unary productions in the grammar. Such these informations are directly represented in ASTs by the structure of the tree [JDJ04].

For example, if the user give the simple SQL query like this:

SELECT T1.Key, T2.Value
FROM T1, T2
WHERE Key = (5+10)-10
Use the parser, we can get the AST in Figure 4.4.

4.2.2 Query Translation

Figure 3.3 showed that query rewrite is a mandatory feature of Query Engine. In our implementation, we implement only feature relational algebra conversion. However, we kept it optional in our prototype implementation since we don't have complex query execution need in our current DBMS implementation. We know that after parsing the query, we can get an abstract syntax tree. But in the inner of relational DBMS, lower - level operations are similar to relational algebra operations. Relational algebra is

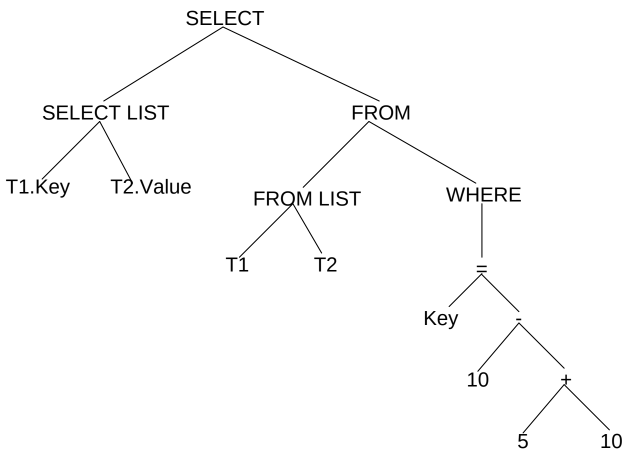

Figure 4.4: One Example of Abstract Syntax Tree

a mathematical formalism that is used to express queries. One advantage of using relational algebra is that it makes alternative forms of SQL query more easy to explore. Relation algebra expression forms the basis of the procedural language of the execution plans. So in order to get a logical query plan for this query, we must translate the AST into relational algebra expression. An expression in relational algebra describes a sequence of operations that can be applied to a relation and which produces a relation as a result. The primary operations of the relational algebra are projection, selection and joins.

RAT makes a little change of AST. The nodes of AST describe for each rule of a grammar. The nodes of RAT are based on the nodes of AST. The RAT create a PROJECTION node for each SELECT node of the AST and also create a SELECTION node for each WHERE node of the AST. Actually, internally of SELECT node and PROJECTION node, WHERE node and SELECTION node are same. In our implementaion, WHERE node actually supports not AND. It supports only one condition. We have also added two special nodes ,one is Attribute node, another is Relation node for RAT. That is because in relational algebra column is called attribute, and relational algebra table is called relation.

Figure 4.5 describe a relation algebra tree translate from the abstract syntax tree in figure 4.4. Because our FAME-DBMS currently support not tables, so in the implementation of query rewrite component, we have ignored the FROM statement. The PROJECTION node in our prototype query engine support only attribute Key and Value. The RAT also evaluate the complex query expression in figure 4.5 with simplified expression in figure 4.4.

Visitor Pattern

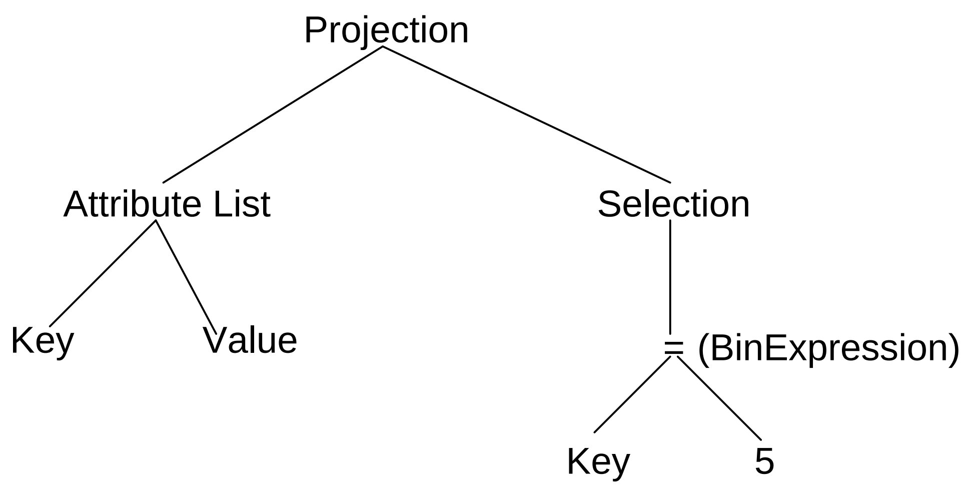

Figure 4.5: Relation Algebra Tree

To avoid frequent type casts or recompilation Visitor pattern has been used. It describes a mechanism for interacting with the internal structure of composite objects. The advantage of Visiteor Pattern is that a new operation can be defindan on an Object structure without changing the classes of the objects on which it operates. This saves us from writing dedicated methods for each programming task and afterwards recompiling. The main objective is to have an *accept* method in every class which passes control back to the visitor. The Visitor behaves like a repository for the new methods [PJ98].

If the operations are distributed across the various node classes, the resultiong system becomes hard to understand, maintain and change [GHJV94]. If a new operation is added, then we will have to recompile all of the classes, we have used visitor to package related operations from each class in a separate object, then we pass it to node of AST as it is traced. Once a visitor has been “accepted“ by a node, a request is sent to the visitor that encodes the node's class, it also includes the node as an argument. The visitor then executes the operation for the node - the operation that is in the class of the node.

Figure 4.6 illustrates the relationship between RANode class and RAVisitor class in our implementation. Classes RAListNode, RAKeywordNode, RAValue and RAExpression inherit from a higher-level class RANode. Class RAPrintVisitor inherits from clas RAVisitor. RAVisitor provides different operations on the RAT. Every class that inherits from RANode contains methods *AcceptVisitor()* and *DoneVisitor()*. *DoneVisitor()* methods do nothing at present but can be implemented later. The function argument to these methods is an object of type RAVisitor to select through dynamic binding, the appropriate visit method to be called. The RAVisitor implements a ”double- Dispatch” machanism [MA06]:

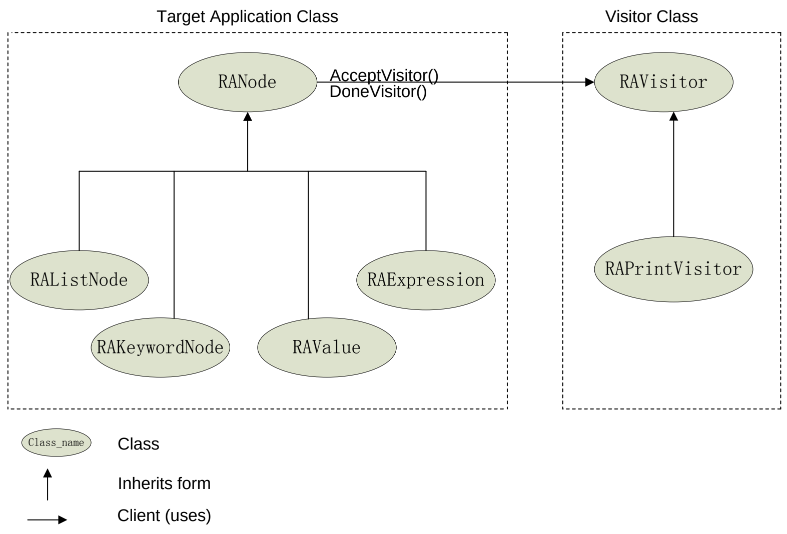

Figure 4.6: Visitor Pattern Architecture

- In a call to *AcceptVisitor()* on a certain RANode, the appropriate version of *AcceptVisitor()* is determined by the type of that RANode.
- When that version is executed on an argument of type *RAVisitor*, the appropriate visitor routine version is dertermined by the type of the RANode attached to that argument as given by one of the RAVisitor classes, for example RAPrintVisitor.

The strong point of the pattern is that it is easy to add new functionalities to a class hierarchy: simply write a new inheritance of RAVisitor to traverse the structure in a different way and perform some other task.

For generating the RAT we used visitor pattern. First we traverse the AST to generate the RAT. Based on visitor patter, we override the methods for each type of node. As we encounter some typed of node in AST, we generate a RAT node in RAT that contains the same meaning in relational algebra form. For example, when we encounter WhereNode in AST, we generate the Selection node in RAT. Similarly, we traverse AST, generates similar nodes in RAT and at the same time we simplify the concepts where applicable. For example, if where clause contains the k = 7 + (10 - 5) as expression. While generating RAT we will simplify it to k = 12. It is a simple example. However, we can extend this concept for simplifying the complex queries for execution during ReWrite. Once RAT is generated, we can traverse the RAT for execution.

4.2.3 Implementation Query Execution

Query Execution manipulate the data of the database. In the part of Query Rewrite translate the query into relation algebra expression. Query Execution execute the oper-

	Parser	Query Rewrite	Query Execution
Select	×	×	×
Insert	×	×	×
Update	×	×	×
Delete	×	×	×
DbOperations	×	×	×

Table 4.1: Some Feature Interactions

ations of relational algebra. Consider about in practical application, DBMS will quite complete, it should at least support tables. We must translate the AST into RAT in order to optimizer the logical query plan and generate a physical query plan based on the relation algebra expression tree.

But in our initial FAME-DBMS store only KEY/Value pair. In this case, it does not need the functionality of Query Optimizer. In our implementation of query engine prototype, actually query rewrite does not make sense. Because of this reason, we implement the Query Execution with two ways based on the selected components. One approach implement based directly on the AST. Another approach implement based on the RAT. Both of this two approaches are possible in our initial FAME-DBMS.

4.3 Feature Interactions

We can use FeatureC++ to modularize crosscutting concerns. But interactions between different crosscutting features occur often in the implementation. These feature interactions often enforce a particular composition order of features and can result in special interaction code, know as *derivatives* [1].

Since we have developed our prototype based on feature oriented decomposition mechanism. We have seperated, end-user visible functionalities based on features. However, not all features are mandatory and this results in feature optionality problem. In our prototype mechanism we have used feature derivative approach to solve this issue. We have talked about in table 3.1 some constraints about different features. We have found that feature Select, Insert, Update, Delete and DbOperations in the implementation have crosscut features Parser, Query Rewrite, Query Execution. In Table 4.1 shows this feature interactions.

In our implementation, we have two approaches to resolve this feature interaction problem.

- One approach adds directly related feature below each of crosscuted features. For

[1]http://wwwiti.cs.uni-magdeburg.de/iti-db/fcc/

example, Select feature crosscut feature Parser, Query Rewrite, Query Execution. We add feature Query Rewrite/Select, Query Execution/Select for feature Query Rewrite and Query Execution part.

- Another approach uses the feature derivatives to solve this as we describe below.

We can use the first approach to implement the functionalities like Select, Insert in our prototype. But it cause not consistence of feature domain lay.

In our prototype mechanism we have used feature derivative approach to solve this issue. Because the special code is needed to implement the interaction of different features, feature derivatives can be modularized and separated from the interacting features [LBL06]. A derivative occurs only in the interacting features are present in an SPL instance. Implementing with FeatureC++, derivatives are stored in a separate folder which used only for derivatives. The symbolic links within folders of the corresponding features are used to ease navigation[2]. When SPL building, the FeatureC++ precompiler generates binary code for derivatives. For example, in our implementation the selection of features in Query Rewrite and Execution component is based on the feature selection in Parser. In parser we have DbOperations, Project, Select and Insert features that we used for feature derivative selection in Query Rewrite and Execution components. Query Rewrite and Execution components are composed based on the Parser component feature selection. If we remove the Insert feature from the parser, Insert functionality will automatically be removed from the Query Rewrite and Execution components. This enables use high customization in our query engine prototype. In our implementation we have generated 10 feature derivatives. There are 4 feature derivatives, i.e., DbOperations, Select, Project and Insert crosscut three features, i.e., Parser, ReWrite and Execution. There are two feature derivatives that crosscut ReWrite and Execution feature, as well as Execution and QueryEngine. If ReWrite feature is not selected these two derivatives, it should automatically generate code that should execute based on AST.

4.4 Summary

This chapter describes the main ideas of how implement a customizable query processor. We first describe that how can generate customizable query engine based on customizable parser. We based on a simple customizable SQL parser to implement query rewrite and query execution. Because our initial FAME-DBMS do not support tables, just contains a pair of value (key,value), we have ignored the query optimizer part for the implementation. At the last, we talked about how can solve the feature interaction problem in the implementaion.

[2]http://wwwiti.cs.uni-magdeburg.de/iti-db/fcc/

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we demonstrated how feature-oriented decomposition can be used to develop highly customizable query processor. In this chapter, we present the conclusions that we reach during the presented work. Highly customizable query engine is very critical for customizable DBMS. Customizable DBMS are tailor-made DBMS based on some important criteria. Functionalities within these DBMS are not same across different DBMS products. All important user visible functionalities are identified and decomposed as features. Each variant is generated based on the selection of feature. For these customizable DBMS, if we use query engine with all functionalities, it will lead to waste of resources. Query engine will have large binary size, high complexity, and complex execution.

In this thesis, we envision the customization of DBMS based on query engine customization. We can identify the required feature of query engine based on the knowledge of domain. As we customize the query engine, DBMS customization should be automatically done. However, for existing applications, it is possible to identify all needed features of DBMS by analyzing applications queries. Based on query analysis, we can generate the configuration for query engine customization and so on we can also customize the DBMS.

We applied Software Product Line concept to address query engine variations by keeping common assets inside a core element and combining it with a set of extra functionalities that might be selected to provide variation. These core elements are defined as mandatory feature in feature diagram. Extra functionalities are defined as optional features. Product Line instances are the result of combining the core element and a set of extra functionalities. For example, the high-level query engine feature diagram contains core functionalities, i.e., *Parser*, and *Executor*. These core functionalities actually address the basic requirement of query engine. Each of these core features can also be decomposed according to the application need. Combining these features in different but valid combinations will generate various software products.

Feature-oriented decomposition of SQL can be used to generate a customizable SQL parser. For each decomposition a different parser can be generated. One approach could be to use customizable parser generator. Using customizable parser generator, it is possible to generate parser by writing grammar. It is possible to write grammar for each decomposition of SQL and then generate parser for it. This approach is efficient for parser generation but other functionalities are implemented manually. Since, parser generator generates parser based on grammar, to make parser customizable there should be mechanism in existing tools for writing grammar using feature oriented approach. However, there is not such support in existing parser generators. For each SQL variant, we will have to write a separate SQL grammar which is not viable. Another approach is to have hand-written parser that we also followed in this thesis. Using handwritten parser, we can write parser based on software product line approach. We decompose parser into user visible features and generate variants of parser for different SQL decompositions. Now this give us control on how parser should be generated and what features it should contain based on the grammar it should parse. This also gives us possibility to generate other query engine components as well as DBMS based on the parser feature selection.

A customizable SQL processor is composed of customizable SQL parser and different features in the family of SQL engine. For k features, theoretically 2^k feature combinations are possible. But it does not mean that every combination is possible. There exist optional features. Interactions between different features that are being composed can lead to an invalid composition. This feature optionality problem reduces the benefits of software product line approach. There exists extensive literature on the mechanism of solving feature optionality problem. In order to avoid this invalid composition, we have described constraints between different features of various software product lines (i.e., *FAME-DBMS*and *Query Engine*). The most highly used approach to overcome feature optionality problem is feature derivative approach. We used this approach in resolving feature optionality problem and to ensure that generated combination of SQL engine is valid and correct. We identified that there are certain implementation dependencies between features that are not identified at domain level in feature diagram. We have to satisfy these dependencies during implementation. We used feature derivative approach, but we do not found it highly scalable. In our implementation, we have six derivatives for only one optional feature of ReWrite. We identified that number of derivatives grow approximately exponentially as the number of optional features and cross-cutting features grows.

We have implemented a simple customizable SQL parser specifically implementing query rewrite and query execution. As currently we have pair of data (i.e., key, value) and we do not have tables support in FAME-DBMS, parser can parse tables and columns but nodes with table information are ignored during query rewrite and execution. Similarly only pair of data (i.e., key, value) is executed. Our implementation is highly customizable and with addition of functionalities in FAME-DBMS, it will be possible to add functionality in query engine.

Highly customizable query engine are important for support different SQL dialect efficiently. In future as more application areas for data management emerges, customizable DBMS will play an important role and the most important part of it will be customizable query engine.

5.2 Future Work

There are some area of research that can be done in the near future along the direction of this thesis.

Prototype Implementation

Although we have partially implemtented our system based on FAME-DBMS, there is still some more work to do for a prototype system of our design. In our current design, we had to ignore some features of query languages, such as transactions management, view supports, etc. Also, the design of the whole system is still not exquisite. For a more realistic prototype, a more careful and comprehensive design is needed. Our current testing experiment is only based on FAME-DBMS. Although, according to our studies, generating query processors for other DBMS should be quite similar to our current test. So prototypes based on other DBMS should also be constructed.

Query Optimization

In our thesis, we have not considered query optimization strategies in detail. But query optimization is one of the crucial parts of query processing. Although many Query optimization strategies designed for traditional SQL can still be applicable to FAME-DBMS but with FAME-DBMS many more complex situations are encountered where traditional query optimization stratigies can not be applied, as these algorithms are mainly based on the predefined operations for the predefined types in the system. In such situations new specifically tailored optimization algorithms have to developed. The designers provide some optimization hints (operetion algebric rules, logic rules or code segment transformation rules) based on the operation specification or implementation should be very useful to the system.

Granularity of a SQL Engine Decomposition

In Section 3.2, we have provided a coarse-grained decompostion of SQL Engine but we can also show that a much more fine-grained decomposition is possible. The main challenge in future research will be to find the appropriate granularity for a SQL Engine decomposition. The main hurdle is the resulting complexity of a DBMS implementation which increases as the level of granularity increases. This however has the benefit that a fine - grained decomposition produces an SQL engine that better fits to application scenarios.

FAME-DBMS Extension

Because initial FAME-DBMS support only KEY/Value pair data. It restrict the implementation of the Query Rewrite and Query Exectution. In the future, we can extend our prototype FAME-DBMS to support tables and much more data types. Based on this support of tables, we can also extend our customizable query engine to support tables. In the implementation layer of Query Rewrite, it can support FROM statement. Similarly, in the Query Execution can also provides Join Operation feature.

Bibliography

[ALMK08] Apel, S.; Lengauer, C.; Moeller, B.; Kaestner, C.: *An Algebra for Features and Feature Composition.* Pearson Addison Wesley, University of Passau et al, 2008.

[ALRS05] Apel, S.; Leich, T.; Rosenmueller, M.; Saake, G.: *Feature C++: Feature-Oriented and Aspect-Oriented Programming in C++.* University of Magdeburg, Germany, 2005.

[Ame85] *The American Heritage Dictionary.* Houghton Mifflin, Boston, 1985.

[Ape07] Apel, S.: *The Role of Features and Aspects in Software Development.* University of Magdeburg,Magdeburg, 2007.

[Bat03] Batory, D.: *A Tutorial on Feature-oriented Programming and Product-lines.* In Proceedings of the 25th International Conference on Software Engineering, Washington USA, 2003.

[Bat05] Batory, D.: *Software Product Line Conference 2005-Feature Models,Grammars,and Propositional Formulas.* University Of Texas at Austin,Austin, 2005.

[BEHM02] Batory, D.; E.Lopez-Herrejon, R.; Martin, J.-P.: *Generating Product-Lines of Product-Families. Automated Software Engineering*, University Of Texas at Austin,Austin, 2002.

[BJK05] Batory, D.; Jung, E.; Kapoor, C.: *Automatic code generation for actuator interfacing from a declarative specification.* 2005.

[BSR04] Batory, D.; Sarvela, J.; Rauschmayer, A.: *Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering*, 2004.

[CDG$^+$90] Carey, M. J.; Dewitt, D. J.; Graefe, G.; Haight, D. M.; Richardson, J. E.; Schuh, D. T.; Shekita, E. J.; V, S. L.: *The EXODUS Extensible DBMS Project: An Overview.* Reading in Object-Oriented Database Systems, 1990.

[CE00] Czarnecki, K.; Eisenecker, U. W.: *Generative Programming - Methods,Tools,and Applications.* Pearson Addison Wesley, 2000.

[CHE04] Czarnecki, K.; Helsen, S.; Eisenecker, U.: *Staged Configuration Using Feature Models.* SPLC 2004: software product lines, University of Waterloo,Canada,University of Applied Sciences Kaiserslautern,Germany, 2004.

[CRC03] Cao, F.; R.Bryant, B.; C.Burt, C.: *Automating Feature-Oriented Domain Analysis.* University of Alabama,Birmingham, 2003.

[Dij76] Dijkstra, E.: *A Discipline of Programming.* Prentice Halle, 1976.

[Dij97] Dijkstra, E. W.: *A Discipline of Programming.* Prentice Hall, 1997.

[EBC00] Eisenecker, U.; Blinn, F.; Czarnecki, K.: *A Solution to the Constructor Probelm of Mixin-Based Programming in C++.* In *GCSE'2000 Workshop on C++ Template Programming*, 2000.

[FF98] Findler, R.; Flatt, M.: *Modular Object-Oriented Programming with Units and Mixins.* In *Proc. of the 3rd Int. Conf. on Fuctional Programming*, 1998.

[GHJV94] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. M.: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1994.

[GMUW00] Garcia-Molina, H.; Ullman, J. D.; Widom, J.: *Datebase System Implementation.* Prentice Hall, America, 2000.

[Gri00] Griss, M.: *Implementing Product Line Features by Composing Component Aspects.* Proc.First Int'l Software Product Line Conf., Aug 2000.

[Int99] *International Organization for Standardization(ISO).* In Identification Cards- Integrated Circuit(s) Cards with Contacts, ISO/IEC 7816-7, 1999.

[JDJ04] Jones, J.; Dunnavant, C.; Jay, T.: *A Pattern Language for Language Implementation.* Tuscaloosa,USA, 2004.

[JMS$^+$08] Jain, N.; Mishra, S.; Srinivasan, A.; Gehrke, J.; Widom, J.; Balakrishnan, H.; Cetintemel, U.; Cherniack, M.; Tibbetts, R.; Zdonik, S.: *Towards a Streaming SQL Standard.* Proceedings of the 34th International Conference on Very Large Data Bases, Auckland, 2008.

[KBL05] Kifter, M.; Bernstein, A.; Lewis, P. M.: *Database Systems - A Plication Case.* Pearson Addison Wesley, State University of New York,Stony Brook, 2. Auflage, 2005.

[KCH^{+}90] Kang, K. C.; Cohen, S. G.; Hess, J. A.; E.Novak, W.; Peterson, A.: *Feature-Oriented Domain Anlaysis(FODA)Feasibility Study.* Software Engineering Institute, University of Camegie Mellon, 1990.

[LBL06] Liu, J.; Batory, D.; Lengauer, C.: *Feature Oriented Refactoring of Legacy Applications.* Proceedings of the international conference on Software engineering (ICSE), 2006.

[MA06] Meyer, B.; Arnout, K.: *Componentization: the Visitor example.* 2006.

[MMHH05] Madden, S.; M.J.Franklin; Hellerstein, J.; Hong, W.: *TinyDB: An Acquisitional Query Processing System for Sensor Networks.* ACM Transactions on Database Systems, 2005.

[Nor07] Nori, A. K.: *Mobile and Embedded Databases.* Microsoft Corporation, 2007.

[NS02] Nakkrasae, S.; Sophatsathit, P.: *A formal approach for specification and classification of software components.* Proceedings of the 14th international conference on Software engineering and knowledge engineering, USA, 2002.

[OJ90] Opdyke, W.; Johnson, R.: *Refactoring: An Aid in Designing Application Frameworks.* ACM-SIGPLAN, September 1990.

[Par76] Parnas, D. L.: *On the design and development of program families. IEEE Transactions on Software Engineering*, 1976.

[PBvdL05] Pohl, K.; Boeckle, G.; Linden, F. v. d.: *Software Product Line Engineering - Foundations,Principles,and Techniques.* Springer, State University Of Duisburg-Essen,Essen, 2005.

[PJ98] Palsberg, J.; Jay, C.: *The Essence of the Visitor Pattern.* USA, 1998.

[Pre97] Prehofer, C.: *Feature-Oriented Programming: A Fresh Look at Objects.* University of Technisch Muenchen,Muenchen, 1997.

[Ram97] Ramakrishnan, R.: *Database Management Systems.* Tom Casson, 1997.

[RKS^{+}09] Rosenmueller, M.; Kaestner, C.; Siegmund, N.; Sunkle, S.; Apel, S.; Leich, T.; Saake, G.: *SQL ‘a la Carte - Toward Tailor-made Data Management.* In Proceedings of the GI-Fachtagung Dantenbanksysteme fuer Business, March 2009.

[RSS^{+}08] Rosenmueller, M.; Siegmund, N.; Schirmeier, H.; Sincero, J.; Apel, S.; Leich, T.; Spinczyk, O.; Saake, G.: *FAME-DBMS: Tailor-made Data Management Solutions for Embedded Systems.* ACM International Conference Proceeding Series, 2008.

[SB93] Singhal, V.; Batory, D.: *P++: A Language for Large-Scale Reusable Software Components.* In Workshop on Software Reuse, 1993.

[SB00] Smaragdakis, Y.; Batory, D.: *Mixin-Based Programming in C++.* In *Proc. of GCSE*, 2000.

[SB02] Smaragdakis, Y.; Batory, D.: *Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. ACM Transactions on Software Engineering Methodology*, 2002.

[SKS96] Silberschatz, A.; Korth, H. F.; Sudarshan, S.: *Database System Concepts.* McGraw-Hill, 3. Auflage, 1996.

[SKS+08] Sunkle, S.; Kuhlemann, M.; Siegmund, N.; Rosenmueller, M.; Saake, G.: *Generating Highly Customizable SQL Parsers.* Workshop on Software Engineering for Tailor-made Data Management(SETMDM), 2008.

[SR98] Sen, R.; Ramamritham, K.: *Efficient Data Management on Lightweight Computing Devices.* IEEE Computer Society Press, Indian Institute of Technology India, 1998.

[Sun07] Sunkle, S.: *Feature-Oriented Decomposition of SQL:2003.* University of Magdeburg,Magdeburg, 2007.

[TBD07] Trujillo, S.; Batory, D.; Diaz, O.: *Feature Oriented Model Driven Development:A Case Study for Portlets.* Accepted for Publication,International Conference on Software Engineering(ICSE)2007, University of the Basque Country and University of Texas at Austin, 2007.

[YG02] Yao, Y.; Gehrke, J.: *The Cougar Approach to In-Network Query Processing in Sensor Networks.* Sigmod Record, September 2002.

[YM98] Yu, C. T.; Meng, W.: *Principles of Database Query Precessing for Advanced Applications.* Morgan Kaufmann Publishers, 1998.

[ZJM+08] Zdonik, S.; Jain, N.; Mishra, S.; Srinivasan, A.; Gehrke, J.; Widom, J.; Blalkrishnan, H.; Cherniack, M.; Cetintemel, U.; Tibbetts, R.: *Towards a Streaming SQL Standard.* Proceedings of the VLDB Endowment, 2008.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den 2nd June 2009

Shuai Cao

Appendix A

Feature ID -Query Specification

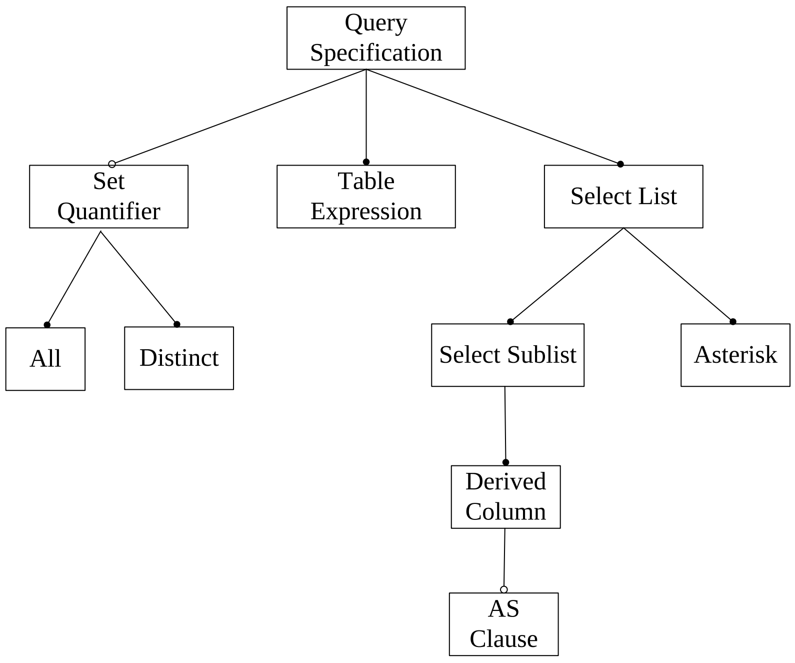

Figure 5.1: Query Specification Feature Diagram [Sun07]

Semantic Description - Figure 5.1shows the feature diagram for *Query Specification*. The *Query Specification* feature specifies a SELECT statement. The [1..*] cardinality notation for *Select Sublist* feature indicates that one or more columns can be selected. The *Set Quantifier* feature tests for duplicate rows and returns all rows or distinct rows depending on the features ALL and DISTINCT. The *Asterisk* feature indicates that all columns of specified table are selected. The *AS Clause* is used to name or rename result columns [Sun07].

Feature ID -Table Expression

Semantic Description - Figure 5.2shows the feature diagram for *Table Expression*. The *Table Expression* feature specifies a table or a grouped table. The *From Clause*feature is mandatory. The *Where Clause* feature can be used to apply conditions to columns in a table expression. With the *Group By* feature can be used to group columns values while using aggregate and other grouping functions. *Having Clause* feature is supposed to be used along with Group By to apply conditions to grouping of columns. If it is used without *Group By Clause* feature then it applied to all rows that satisfy given condition. Since Having Clause can be used independently of *Group By Clause* feature, no 'requires' arc has been shown between the two [Sun07].

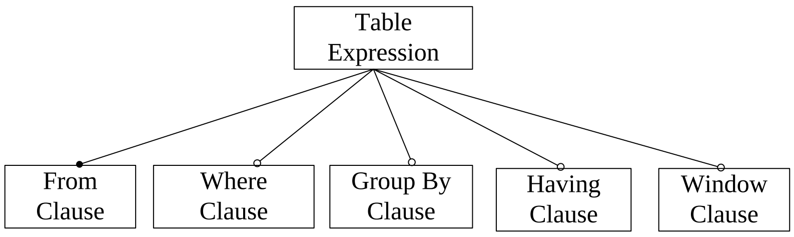

Figure 5.2: Table Expression Feature Diagram [Sun07]

Appendix B

*1 means that if the selection is σc, then we can only push this selection to a relation that has all the attributes mentioned in C, if there is one. We shall show the laws below assuming that the relation R has all the attributes mentioned in C [GMUW00].

*2 means that M is the list of all attributes of R that are either join attributes (in the schema of both R and S) or are input attributes of L, and N is the list of attributes of S that are either join attributes or input attributes of L [GMUW00].

*3 means that M and N are the lists of all attributes of R and S, respectively, that are input attributes of L [GMUW00].

*4 means that C is the condition that equates each pair of attributes from R and S with the same name, and L is a list that includes one attribute from each equated pair and all the other attributes of R and S [GMUW00].

Different Algebraic Laws for Improving Query Plans	Expression of Laws
Commutative and Associative Laws	R×S = S×R; (R×S)×T = R×(S×T)
Commutative and Associative Laws	R∞S = S∞R; (R∞S)∞T = R∞(S∞T)
Commutative and Associative Laws	R⋃S = S⋃R; (R⋃S)⋃T = R⋃(S⋃T)
Commutative and Associative Laws	R⋂S = S⋂R; (R⋂S)⋂T = R⋂(S⋂T)
Laws Involving Selection	σc1 AND c2(R) = σc1(σc2(R))
Laws Involving Selection	σc1 OR c2(R) = (σc1(R)) ⋃S (σc2(R))
Laws Involving Selection	σc1(σc2(R)) = σc2(σc1(R))
Selection Laws for Union	σc(R⋃S) = σc(R)⋃σc(S)
Selection Laws for Difference	σc(R-S) = σc(R)-σc(S)
Selection Laws for Product	σc(R×S) = σc(R)×S *1
Selection Laws for Join	σc(R∞S) = σc(R)∞S *1
Selection Laws for Intersection	σc(R⋂S) = σc(R)⋂S *1
Laws Involving Projection	πl(R∞S) = πl(πm(R)∞πn(S)) *2
Laws Involving Projection	πl(R×S) = πl(πm(R)×πn(S)) *3
Laws About Joins	R∞S = πl(σc(R×S)) *4
Laws involving Duplicate Elimination	δ(R×S) = δ(R)×δ(S)
Laws involving Duplicate Elimination	δ(R∞S) = δ(R)∞δ(S)
Laws involving Duplicate Elimination	δ(σc(R)) = σc(δ(R))

Table 5.1: Different Algebraic Laws for Improving Query Plans [GMUW00]