Master Thesis

Modular Derby

by
Pawel Kowalski

Due date
14. April 2009

Advisors:
Prof. Dr. Gustavo Alonso
Ionut Subasu

ETH Zurich, Systems Group
Department of Computer Science
8092 Zurich, Switzerland
Abstract

Database Management Systems (DBMS) rely on highly optimized engines. However the optimization process was driven by the context of self-contained systems. In the current trend of multicore CPUs and internet scale (highly parallelized) applications their monolithic design is becoming more and more of a scalability issue. While current approaches try to achieve performance by replicating or partitioning data in different ways, to minimize concurrent access, we take a different approach by modularizing an existing RDBMS (Apache Derby) into functional units, that are going to be distributed across several physical machines. In this thesis it has been shown that it is possible to modularize and distribute the phases of SQL processing of an existing RDBMS (Apache Derby) without changing its functional behavior. We describe the required analysis, refactorings and methodology used. Additionally we identify and describe several key software engineering principles necessary for such refactoring. This thesis focuses on restructuring an existing RDBMS engine to run in a distributed manner. We also suggest concrete paths to achieve improved scalability and further modularization.
Preface

This project is done as a Master Thesis at the Computer Science Department of the ETH Zürich in the Information Systems track.

I would like to extend a very special thank to Prof. Dr. Gustavo Alonso for giving me the opportunity to do this project, for his invaluable help, listening to my ideas and for his critical questions. I would also like to thank Ionut Subasu for his help during the entire project.

In addition I want to express my gratitude to all my friends and my parents for supporting me during the studies at the ETH and this project as well. A special thank goes to my girlfriend Mamela for her patience and for helping me with all the figures I’ve used in this project. Last but not least I want to thank Patric, Peter and Leo for proofreading this Master Thesis and giving me valuable feedback.

Thank You All!
## Contents

Preface v

### 1 Introduction
1.1 Modularization and Distribution 2
1.2 Focus and Contributions 2
1.3 Outline of the Thesis 2

### 2 Background
2.1 Technology overview 5
  2.1.1 Apache Derby 5
  2.1.2 OSGi 7
  2.1.3 R-OSGi introduction 9

### 3 Approach
3.1 Solution finding approach 11
3.2 Envision 13

### 4 Derby analysis
4.1 Analyze class dependencies 16
4.2 Analyze interface usage 18

### 5 Derby restructuring / Design principles
5.1 Concrete refactoring catalog 24
  5.1.1 Successful Refactorings 25
  5.1.2 Unsuccessful Refactorings 33
5.2 Abstract refactoring catalog 34
5.3 Design principles 36
5.4 Conclusion 37

### 6 System design
6.1 Abstract OSGi principles 39
6.2 Modular Derby bundle structure 40
6.3 Modular Derby query processing 40
6.4 Module registrar service 42
6.5 Two Phase bootstrap 44

### 7 Software engineering aspects
7.1 Solution space exploration 49
7.2 Test driven development and refactoring 51
7.3 Code demarcation ................................. 53
7.4 Change tracking ................................. 53

8 Generic Process ................................. 57

9 Evaluation ........................................ 61
  9.1 Functional Tests ............................... 61
  9.2 Performance Tests ............................. 62

10 Conclusions and future work ................ 65
  10.1 Future Work .................................... 65
Chapter 1

Introduction

Most existing RDBMS (Relational Database Management Systems) are designed in a monolithic way with their functions highly integrated and optimized. This is to maximize throughput, lower response times, handle many concurrent users and at the same time assure the ACID (Atomicity, Consistency, Isolation, Durability) database properties. With the raise of internet scale applications, a new challenge, the scalability issue has appeared. This issue has been approached in many different ways. Some of approaches achieve scalability through data replication, others through horizontal partitioning. Current database management systems, however, are still designed in a monolithic way. Only the data is distributed, replicated or partitioned. This inhibits scalability and adaptation to changing requirements.

The goal of this Master Thesis is to analyze, modularize and finally distribute functions of Apache Derby, an existing RDBMS. In contrast to data replication or partitioning, the focus of this project is the distributed execution of functionality that in totality constitute the full Derby RDBMS. The data is not replicated but the functions responsible for parts of the RDBMS overall system are distributed. Possibly with multiple instances of the same function used for load balancing. The challenge taken up with is in particular not the design of a distributed database management system, but the restructuring and distribution of an existing monolithic system; a system which was designed with a modular service architecture in mind, but not a modularization for distribution of functionality. The modularization and distribution targeted in this work are themselves means to pave the way for improving the scalability of the original Derby system.

Moreover, based on the experience gained by solving the problem of modularizing the Apache Derby RDBMS, the purpose of this project is to present a generic approach to solve the problem of modularizing and distributing existing systems. More concretely this approach shall present a process which can be followed to modularize and distribute systems, as well as presenting tools to support such a process and ideas about how to restructure existing systems to achieve modularization and distribution.
1.1 Modularization and Distribution

The main purpose of the Modular Derby project is to modularize Derby’s codebase and distribute the functions to improve scalability. In the context of this work the terms modularization and distribution shall have the following meaning:

**Modularization** means partitioning its code into dependent units containing classes and parts of the configuration. The code units called modules are structured in layers where no module can reference a module on a higher level. In addition to this constraint no circular references are allowed between modules. As used in this project, the modularization term has only a static meaning, i.e. the modules contain partitions of the code while at runtime they are all possibly needed to form a fully functional system.

**Distribution** means that modules can be executed remotely in a separate address space which might run on a separate physical machine. The purpose is to increase scalability through remote execution of parts of the entire Database Management System. A prerequisite for Distribution is Modularization.

1.2 Focus and Contributions

The focus and contributions of this Master Thesis are:

- **Modularization**: Derby SQL query processing codebase has been modularized, i.e. its code has been partitioned into interdependent modules. The dependency structure shall be layered, i.e. modules from lower layers may not directly use modules from layers above. Modularization is a prerequisite for the distribution of Derby functionality.

- **Distribution**: Derby SQL query processing functions can be executed as remote services. The core of the DBMS (Database Management System) delegates some tasks to those remote services providing parts of the overall DBMS functionality.

- **Generic process description**: The process developed to modularize and distribute Derby is documented in a generalized form. This shall allow application of the described process to other versions of Derby or even other systems to modularize and distribute them.

1.3 Outline of the Thesis

Chapter 2: Background. This chapter gives an overview and introduction of the technologies used to implement this project.
1.3 Outline of the Thesis

Chapter 3: Approach. A high level description of the approach used to find the solution will be given. Furthermore this chapter describes the envisioned solution.

Chapter 4: Derby analysis. It was necessary to analyze Derby before achieving the final goal of this project, i.e. splitting existing Derby functions into modules and executing them remotely to improve scalability. The results of the analysis of the existing Derby codebase gave hints on how to structure parts of the new modular version. How this analysis has been conducted and what tools have been developed and used to support this task will be described here.

Chapter 5: Derby restructuring / Design principles. After the analysis of existing Derby code it was possible to decide on how to structure the code into modules. This structuring would make it possible to distribute Derby. However after the decisions on the modules have been taken it was necessary to restructure and refactor Derby code to make this modularization possible. More details on what has been changed and how the changes have been grouped into more abstract principles will be the topic of this chapter.

Chapter 6: System design. After the restructuring of the existing monolithic system and moving parts of its original code into modules it was necessary to put those parts back together. The modules containing parts of the code have to be reassembled and brought to work in a distributed mode. Chapter 6 describes the system design and important details of how this task has been achieved. In particular it demonstrates how the OSGi and R-OSGi technologies have been used. This chapter also shows the structure of Modular Derby, i.e. to which degree the modularization has been achieved.

Chapter 7: Software Engineering aspects. This chapter describes Software Engineering techniques applied to achieve the results, to mitigate the risks of this project as well as to make it easier to understand the new modularized version of Derby.

Chapter 8: Generic Process. Distilled from the experience of modularizing and distributing Derby, a generic process to modularize and distribute software system has been described. This description will be the topic of this chapter.

Chapter 9: Evaluation. The evaluation of the realized solution will be described in chapter 9. It recapitulates the contributions of this project and shows how to retrace them.

Chapter 10: Conclusions and future work. Finally chapter 10 presents the conclusions of this work. Furthermore this chapter will show possible paths of future work related to this thesis.
Chapter 2

Background

2.1 Technology overview

This chapter contains the description of technologies used in this project. Modular Derby makes intense usage of some technologies. Its implementation is based on Apache Derby [27]. In addition to that it uses the OSGi [4] and R-OSGi (Remote OSGi [20]) technologies for modularization and distribution. For those reasons this technology overview will be given. Its purpose is to introduce the used technologies and thus make this project more accessible.

2.1.1 Apache Derby

Apache Derby is an open source RDBMS (Relational Database Management System) implemented entirely in Java [23]. The Derby RDBMS was first developed as the Cloudscape Database which was later bought by Informix to be in turn integrated into the IBM Corporation. IBM finally donated it to the Apache Foundation [25] where it is currently an Apache DB subproject [26]. It implements the SQL-99 and the SQL-2003 standards [8] and has a relatively small runtime memory footprint, i.e. 2MB for the database engine and the JDBC driver. An interesting property of Derby is that it can be either used as an embedded database executed in the same JVM (Java Virtual Machine) as the client or in the more commonly known client/server mode with the database client and the database engine communicating over the network.

Derby is a fully functional RDBMS supporting the important SQL-99 and SQL-2003 standards. However, it is still a relatively small system and easy to understand. Internally, it uses a fine-grained service design which allows the definition of services using standard Java interfaces and a configurable choice of their implementations to be used at runtime. Its service design, however, does not mean that Derby can be decomposed into independent and distributable modules without effort. Nevertheless, its service design makes the task of modularizing its code and distributing its functions easier.

Derby composes the fine grained services into higher level services like parsing
the SQL, optimizing operator trees, storing data, generating code, managing the transactions and its indices etc. Those high level services provide the basic building blocks which together form a fully functional database management system. They are good candidates for modularization and distribution.

The architecture of Derby and its high level services is depicted in figure 2.1. At the bottom there is a raw storage layer managing data pages which is then directly used by the access storage layer implementing the access methods to the stored data. The replication with its Slave and Master provide data replication functionality to Derby. The Parser, Binder, Optimizer, CodeGenerator and finally the Plan Executor are responsible for processing the incoming SQL queries and finally executing them. Embedded JDBC provides a JDBC driver to Derby which embeds the database management system inside a JDBC driver. In contrast to this, the DRDA (Derby Network Server) provides a network JDBC driver which makes possible to run Derby as a standalone server and access it over the network. On both sides of the described high level services, common low level services are used. These services are used in many areas of the core high level services.

![Apache Derby architecture](image)

What has just been presented is a static view of Derby, i.e. what the different parts of the entire Derby codebase are and which tasks they fulfill. Figure 2.2 illustrates the dynamic behavior of Derby, i.e. how it behaves in order to process a single SQL query. The figure shows how a query is handled by Derby. Firstly, it is parsed and transformed into an operator tree, afterwards it is bound to internal datastructures using table meta-data. Subsequently the query is optimized and standard Java bytecode is generated. The bytecode is then executed to actually get the data from the datastructures and return it to the
2.1 Technology overview

An important detail of Derby architecture is the class hierarchy starting with the `QueryTreeNode`. The `QueryTreeNode` class hierarchy is responsible for modelling SQL queries as trees, i.e. each `QueryTreeNode` descendant models an SQL concept or word (select, from, row, etc.). Some `QueryTreeNode` classes model implementation specific concepts of SQL not directly representable as SQL words. They however all contribute to finally modelling SQL queries. Instances of those classes form trees representing the SQL queries. Later in this work they will be referenced to as operator trees.

2.1.2 OSGi

OSGi [4, 2] is a standard defining a dynamic service platform and an environment where modularized Java applications can be executed. The OSGi standard defines how to build application modules with their own framework managed lifecycle and how those modules can communicate with each other. Basic OSGi application modules are called bundles. The standard describes how to structure bundles and how to specify bundle metadata. OSGi compliant platforms are dynamic, i.e. the bundles can be installed, started, stopped and uninstalled at runtime. In addition to its core OSGi also defines generic utility services provided as bundles which can be used by other bundles.

Because OSGi bundles can provide services to other bundles and OSGi based systems compose those services, the standard can be seen as a SOA [29](Service Oriented Architecture) platform with service orientation within a system and not across system boundaries. The OSGi standard defines a Service Registry where bundles can register and lookup services.

The different parts of the OSGi standard are shown in figure 2.3.
The basic OSGi building block, a bundle is a group of classes, resources and a manifest file describing its meta-data. The meta-data contains information about the name of the bundle, its version, the Java packages it imports and exports, dependencies to other bundles, the Java environment where it can be executed, user specific metadata and other information. A bundle may use a bundle activator which is a Java class implementing a specific interface which will be used to start and stop the bundle. In order to provide functions, bundles can use other bundles and explore contents of other bundles. In this thesis, the name bundle will be used interchangeably with the name module.

The OSGi standard defines a service platform where the OSGi bundles can be executed. This service platform takes care of bundle lifecycle management (depicted in figure 2.4), dependency management between bundles and provides a service registry where the bundles can register and lookup services.

**Figure 2.3: OSGi standard overview**

![OSGi standard overview](image)

The OSGi standard and framework provide many different ways to solve the problems. The real challenge is to know those possibilities and to choose the appropriate one to solve a specific problem. OSGi users need to first explore the existing possibilities to be able to choose the appropriate one before using it. Guidelines on how to use the OSGi framework and how to take advantage of the great freedom of choice have been established. They are called OSGi best
practices [7]. The primitives of OSGi usage are code and service sharing. Code sharing simply means that code from within bundles is used by other bundles using normal Java techniques, i.e. by either instantiating an object, inheriting a class or implementing an interface. In case of service sharing a bundle registers an object (service) which is then available as a service to other bundles. In the latter case the consumer of a service is not responsible for its instantiation, it is the service provider. In this case the service provider does not share its code but its services, hence the name service sharing. How the primitives are used to actually implement systems based on OSGi will be the topic of chapter 6.

2.1.3 R-OSGi introduction

OSGi offers a standardized way to modularize Java software systems and provides a dynamic framework for their execution. The OSGi technology will be the base for the modularization of Derby. Another goal of this project is to distribute functions of Derby, i.e. some parts that contribute to the overall Derby behavior shall be remote services. The basic OSGi doesn’t yet provide means for the remote service execution. This gap is however filled by the R-OSGi (Remote OSGi) technology [20, 18, 19]. The R-OSGi enhances the OSGi standard with a transparent way to access remote OSGi services. Its advantage is that the remote services can be accessed in the standard OSGi way. It does not introduce another service taxonomy in addition to the standard OSGi services.

R-OSGi can be used in many ways. For this project however, only the transparent remote services are relevant. This R-OSGi use-case is illustrated in figure 2.5. The situation is the following: FooService is defined as an interface represented by the FooService box. Its implementation is contained in the FooImpl Bundle. The FooService is provided by the System A. System B contains the service consumer represented by FooConsumer. In order to be able to use the FooService the FooConsumer needs to know its interface, therefore the FooService (interface) must also be available on System B. Both systems also contain the R-OSGi bundle providing remote services. On System B R-OSGi provides a proxy implementing FooService to the service consumer represented as FooProxy box (the box represents only the concept; the proxy is not an OSGi bundle). This proxy allows for remote communication with the actual service provider residing on System A. Finally the FooConsumer can access and use the FooImpl service which resides on System A through the R-OSGi communication layer. Modular Derby will use this technique to access its remote services.
Figure 2.5: R-OSGi use case scenario
Chapter 3

Approach

This chapter gives an overview of the approach used to solve the problem of modularization and distribution of Derby SQL processing as well as the envisioned solution. In addition to that it is a guide through the subsequent chapters of this thesis. Each subsequent chapter of this thesis is devoted to a specific topic of the overall solution and a step in the entire solution finding process. Likewise this chapter with its sections represents the solution finding process in a condensed way and shows how the subsequent chapters are interrelated.

3.1 Solution finding approach

The goal of this project is to modularize the SQL processing of Derby to allow a distributed execution of its functions. To achieve this goal it was necessary to:

**Learn and Analyze** Understand the internal structure and behavior of Derby. The best way to start with modularization and distribution is to know the theory of the system. Furthermore it is necessary to know how the system accomplishes its requirements, how it is structured and how it works internally. Only by tackling the problem through understanding the system it is possible to take the right decisions, to measure the right things and analyze the important parts.

**Envision** Decide on how to partition Derby codebase into new modules. Although this decision was obvious in most cases, there were also many border cases. In such cases it was necessary to understand the dependencies within the code. To that end a tool has been developed which allowed to analyze the dependencies of the single classes. The results of the analysis helped to decide into which modules the classes had to be moved so that structural problems resulting from the partitioning were minimal. This topic will be described in more detail in chapter 4.

**Modularize** Partition and restructure the code. Moving classes into newly created modules was a large part of this step. Nevertheless it was in-
evitable that problems such as circular dependencies between modules arose due to code partitioning. Such problems were resolved by restructuring and refactoring the code after the modularization. Many cases of the restructuring were depending on local decisions and were not easily repeatable in other situations. Yet in some cases it was possible to automate the process of restructuring the code in order to make it not only faster but also less error prone. The tools allowing the restructuring automation have been implemented using the standard Java APT API (Java Annotation Processing) [14] and the Sun Compiler Tree API [13]. In this manner the effort necessary for the refactoring had been reduced.

**Distribute**  
Be able to execute the modularized Derby as a distributed system. The sheer modularization of Derby does not mean that it is possible to execute the system in a distributed mode where some database functions are executed remotely. For that purpose it was necessary to make sure that the data structures exchanged between the modules over the network can be transformed into a binary representation, i.e. they need to be serializable. In addition to that it is necessary to minimize the amount of data exchanged during remote calls. Yet even this does not suffice for distribution. Shared state in monolithic design is not automatically available in distributed settings. Shared state has to be made explicit during service calls or has to be exchanged through dedicated information services. This means that the distributed modules have to provide their services only based on information passed during the service calls. The restructuring and refactoring of Derby will be the topic of chapter 5. Even though the restructuring and refactoring of Derby constituted the biggest part of the entire project, merely changing the existing codebase would not be enough to obtain a modularized and distributed version of Derby. In addition to that it was necessary to design and implement new parts of the system due to the modularization and distribution. Those parts would glue together the newly created modules and allow remote services to cooperate. Likewise it was necessary to design the modules of Derby so that they integrate well with the OSGi framework. Those aspects of the project will be described in the System Design chapter 6.

**Test**  
Make sure that the distributed system still behaves as the original. Also make sure that others working with this project find an easy way to understand what has been done, how it has been accomplished, which parts have been changed, etc. To achieve the goals set for this thesis, Software Engineering techniques have been used. The restructuring and refactoring of the existing Derby system have been done in a test driven way, i.e. for every change a test has been developed and executed. This to make sure that the new modular version of Derby system behaves the same as the original. For the purpose of testing a small framework has been developed which eases the execution of SQL queries against Derby. For the sake of understandability, packaging conventions have been used to delimit new and existing code. In addition to that markers have been used in cases where existing code has been changed. The software engineering aspects will be the topic of chapter 7.
3.2 Envision

After understanding Derby internals by studying its code and runtime behavior it was possible to come up with a way on how to structure the Derby into modules. The envisioned structuring is depicted in figure 3.1.

![Figure 3.1: Envisioned Modular Derby structure](image)

The diagram shows how Derby shall be modularized and how the modules depend on each other (transitive dependencies are omitted). In fact, the symbol for modules with an “API” on top represents two modules: an API module containing the public interface of a service and its implementation. The two modules (indexing, transactions) framed with the “possibly” label shall be analyzed and separated as far as possible. To achieve this envisioned goal the `QueryTreeNode` class hierarchy must be modularized and pruned of code used to optimize, bind and generate code. This situation would allow distributed execution of the parser, the optimizer and the code generator (all marked with a *). Depending on the progress, the indexing and transactions would also be candidates for analysis, modularization and distributed execution.

In the Modular Derby the process to execute a query is the same as in the original. The only difference is that some services will be executed remotely. This is the first step towards a better scalability, as not only one physical system is responsible for computing the results and processing an SQL query. For that purpose it is necessary to have multiple instances of Modular Derby running. Those instances would provide the remote functions like parsing and codegenerating. It is important to note that the instances providing those functions would require the modules that the service implementations depend on. This is depicted in figure 3.2.
Figure 3.2: Envisioned Modular Derby structure
Chapter 4

Derby analysis

To achieve modularization and distribution of Derby it is necessary to restructure it. This requires several steps. Some of them involve actual restructuring of the codebase, changing it and defining interfaces so that they can be used as remote services. However, before all that can be done it is first necessary to understand how Derby is built internally. This can be done by reading Derby documentation and its source code and learning RDBMS theory. This will, however, not fully suffice to achieve the goals of this thesis. For example in order to modularize Derby it is necessary to move classes into newly created modules. In most cases, the decisions on where a class has to be moved can be done based on the understanding of its meaning and functions. In a few cases, however, it will be necessary to understand what the dependencies of a class are. Another case is data structures which transitively reference almost the entire Derby runtime structure. In case of those structures, it is necessary to understand how their interfaces are used. Based on this information it is possible to determine which data of an interface implementation is used for communication within the database software and which is used solely by a given single class or functional group like the parser.

Interface usage patterns can be extracted by reading the source code of Derby, debugging it and writing down the observations. Such manual approach would, however, be very tedious and time consuming. It is therefore necessary to find ways to reduce the time spent on the tedious and repetitive parts of the system analysis. This can be achieved by finding adequate means to automate those tasks. This in turn can be done by using already existing products for software analysis or if suitable and necessary developing custom-made tools.

To analyze Derby several software analysis tools have been used. Where possible we have applied already existing products. In two specific cases, however, we have developed and used custom-made tools. They are, however, generic enough to be used in other situations. Later in this chapter those two tools will be described in more detail.

For the purpose of this project the types of analysis have been done:

**Dependency analysis** For the sake of modularization it was necessary to
determine the dependency structure of Derby. Knowing the dependency structure enhanced with the knowledge about the system requirements (RDBMS theory) was a preliminary step in the envision. It helped drawing the lines between parts of the code. They would later become module boundaries. To analyze static dependencies within the Derby codebase, the “Dependency Tool” [3] has been used. Although very useful for a preliminary dependency analysis between the packages, this tool lacked more detailed information about class dependencies. For a detailed analysis between classes a custom made tool has been developed which will be described later in this chapter.

Interface usage analysis In many cases Derby uses relatively large objects as information containers. The most prominent examples of such objects are the Context classes like the LanguageConnectionContext and the CompilerContext. They are problematic when it comes to distributing Derby’s functions as remote services. Serialization to send them as remote parameters is either impossible or very problematic due to their size. They don’t make the vision of distributing Derby per se impossible; it is possible to extract some information from those objects and pass it as remote parameters. In order to know which parts of those classes can be separated to be passed as remote parameters it is necessary to understand patterns of their usage, i.e. in which order the methods of those classes are used and by whom. Such analysis can be automated and supported by tools and will be described in more detail later in this chapter.

Size analysis Modularization of Derby is only one step which has to be done to distribute functions as remote services. Another is defining remote services and implementing them. Among other things this requires that the data exchanged with the remote services is kept minimal and can at all be serialized in a binary form. Therefore it was also necessary to analyze sizes of data exchanged between functional units of Derby. As the Java language and platform does not provide any standard means to determine sizes of data, the SizeOf [17] library has been used. It allows calculation of the deepsize of datastructures. Such information is useful for the decision on how services have to be designed which eventually will be used as remote services.

Profiling To support modularization and the envision process on how to structure Modular Derby, profiling information was required. It helped determining which functional units of the Derby were good candidates for modules and distribution. Modularizing a function which is very inexpensive in terms of time but very often used would incur unnecessary costs and almost no profit. For that purpose Derby has also been profiled with the JProfiler tool [9].

4.1 Analyze class dependencies

To support the decision process on how to draw module boundaries within the Derby codebase it was necessary to extract detailed information about class
4.1 Analyze class dependencies

dependencies. Although it is possible to do this by reading the sourcecode of a class and writing down its dependencies, it was not practicable for this project. Too much time would have been spent on reading classes and extracting their dependencies by hand. To automate this task a dependency analysis tool has been developed. The tool takes as input the compiled classfile containing the bytecodes of a class and extracts its dependencies to other classes. The dependencies of a class are extracted from the constant pool, field information and the method signatures of a class (as defined by the class file format [28]1.

Listing 4.1 shows the structure of a classfile and the sections relevant for this analysis.

Listing 4.1: Classfile structure

```java
ClassFile {
  u4 magic;
  u2 minor_version;
  u2 major_version;
  u2 constant_pool_count;
  cp_info constant_pool[constant_pool_count – 1];
  u2 access_flags;
  u2 this_class;
  u2 super_class;
  u2 interfaces_count;
  u2 interfaces[interfaces_count];
  u2 fields_count;
  field_info fields[fields_count];
  u2 methods_count;
  method_info methods[methods_count];
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

In addition to the core extract functionality, the analysis tool can also filter uninteresting dependencies and group dependencies. This additional functions allow to reduce the amount of information retrieved and concentrate on relevant information. The results of the dependency analysis are files named after the analyzed classfiles and containing the interesting dependency information as strings. An example of an output can be found in listing 4.2. Those are the analysis results of the class shown in listing 4.3

Listing 4.2: Dependency analysis output (ExecRow.depinfo)

```java
org.apache.derby.iapi.services.io.FormatableBitSet
org.apache.derby.iapi.sql.Row
org.apache.derby.iapi.sql.execute.ExecRow
org.apache.derby.iapi.types.DataValueDescriptor
```

Listing 4.3: Analyzed class for dependencies

```java
package org.apache.derby.iapi.sql.execute;
```

1In order to reduce the time necessary to develop this tool, an open source library, “The Kawa language framework” [1], has been used to read the classfiles. This helped reduce the total amount of code necessary to implement it.
As mentioned earlier, the results of the dependency analysis were good hints on where a class belongs. Thanks to those results, it was easier to decide on the affiliation of classes to modules. The following case demonstrates this: Two modules are created and most classes can be assigned to either of them. There is, however, a class which cannot be assigned based on its responsibilities. The class is analyzed and the results are that the dependencies of this class to one of the module are 10-fold the dependencies to the other module. In such a case the class will probably have to be moved into the former module. The term “probably” has been used on purpose. It is still possible that this decision is not correct. The decision itself is a creative task and has to be considered thoughtfully. The analysis tool can only support such a decision.

4.2 Analyze interface usage

Derby often makes use of relatively large data structures. The most prominent examples of such objects are the Context classes like the LanguageConnectionContext and the CompilerContext. They are used as data containers used by functional units like the parser, optimizer, the code generator, to execute queries and retrieve data, and many others. The functional units use those structures to store intermediary results. By their design the Context classes contain information used by different functional units not relevant to the others. In other situations those structures are used as communication elements. Some functional units don’t receive all the necessary data explicitly as method call parameters and neither do they return all results directly. Instead they use the mentioned contexts to pass information implicitly. In case of Modular Derby, such implicit communication is not viable. Remote service calls need to contain all the data. In exceptional cases where communication elements are used to pass information implicitly, the data structures may not transitively reference the entire Derby (which is the case for the Context classes).

To solve this problem it is necessary to analyze exact data usage and communication elements. The results of such analysis can hint on how the structures can be decomposed according to their responsibilities, i.e. as data containers for
functions or communication elements. The following example demonstrates this case:

The CompilerContext structure is composed as follows:

```plaintext
CompilerContext {
  fooOne;
  fooTwo;
  fooResult;
  barOne;
  barTwo;
  barResult;
}
```

It is used by the foo and the bar services. The foo service uses fooOne, fooTwo to store intermediary results and fooResult to store implicit information to be passed as results (those results are then possibly used by the bar service). Analogically, bar uses barOne and barTwo to store intermediary and private results and barResult to return final results. In case the foo service is made remote, it is necessary to pass the entire structure to it even if only foo* values would suffice. To solve this problem, the CompilerContext structure can be subdivided in a foo related structure and a bar related structure. The result information can either be passed explicitly or in a data container designed specifically for this purpose.

In this example it is obvious which data belongs solely to the foo service, which to the bar and which is used by both to communicate. In case of the Context classes this is seldom the case. In most cases, it is not clear which data elements are used as intermediary results not used by others and which are used to communicate. Moreover it is not clear which elements belong to which functional units, e.g. which data is used by the parser and which by the optimizer. The situation is even more obscured by the fact that the data elements are manipulated by methods in Context classes.

To shed light on the usage of data elements in structures used by many different functions and from different places, it is necessary to analyze and understand usage patterns of the interfaces. More specifically it is important to understand in which order the methods of an interface are used. Such analysis allows to see which elements are function "private” and which are used for communication. This is possible because by looking closely at the code it is possible to see which data is touched by which methods.

Again such analysis can be done manually by debugging the system, setting breakpoints on all methods, taking notes whenever the methods are called and looking at the stack traces to determine the users of the methods. Obviously a manual approach is time consuming and tedious. The same can be achieved at a much lower cost with a tool extracting the same information at runtime. Because of the importance of this analysis and the fact that doing it manually is very time consuming, a tool has been developed to simplify and automate such analysis. The tools plugs in as a transparent proxy\(^2\) between interface implementations and their users, registers all method calls, determines the callers by analyzing

\(^2\)Standard Java Dynamic Proxies have been used to implement this tool [24].
the stack traces and provides the results as graph descriptions using the dot format [21]. Later it uses the GraphViz library [22] to render the graphs and provide the information visually.

Listing 4.4: CompilerContext interface

```java
package example;

public interface CompilerContext {
    void foo();
    void boo();
    void bar();
    void foobar();
}
```

To visualize this analysis, consider the following example: The interface CompilerContext shown in listing 4.4 is used by two classes, Foo and Bar. Those two classes use the CompilerContext interface and the sequence of method calls is:

```java
CompilerContext cc = ... // compiler context instantiation

class Foo {
    ...
    cc.bar();
    cc.foo();
    cc.foo();
    cc.foobar();
    cc.foobar();
    cc.bar();
    ...
}

class Bar {
    ...
    cc.bar();
    cc.boo();
    ...
}

class Foo {
    ...
    cc.bar();
    ...
}
```

The analysis tool now allows recording all method calls done on the CompilerContext and presenting them visually. The information is presented as a directed graph. In this graph, the nodes represent method calls and the vertices the order in which the method calls are done. On top of the nodes the method names are contained and below the information about their callers. The numbers show in which order the methods are used. In case a method is used by the same caller, the information is compressed and no ordering is explicitly given.
4.2 Analyze interface usage

Based on this information it is possible to deduce which data elements (touched within the methods) are used by Foo, which by Bar and which are used for communication. The results of the analysis are shown in figure 4.1. The figure shall be interpreted as follows: first the method “bar” is called by the Foo class (represented as the node with “bar” on top and the “Foo 0” on the second line). The next method called is “foo” and the caller is again “Foo”. The vertex with the label “0” means that the “foo” method is called after the “bar” method. Then the “foobar” method is called twice by the “Foo” class etc. All numbers represent the sequence of methods called. In case a method is always called by the same class, the sequence numbers are omitted.

![Diagram of CompilerContext callgraph](image)

The visual representation shows what can be deduced from the usage sequence, namely that the bar is used by both classes, Foo and Bar. The others are “private” methods only used by either Foo or Bar. This information can be easily deduced by looking at the call sequence. In Derby or in normal cases however, this information is not that obvious. The analysis tool described above helps to determine it. Knowing about the patterns in interface usage helps determining how data contained in an object is used and by whom. This in turn supports the decision process on how to restructure classes that are data containers, which is necessary for the purpose of modularizing and distributing Derby.
This chapter contains a description of the restructuring and refactoring applied to the existing Derby code. This restructuring and refactoring of the existing code was necessary to provide the basis for modularizing the code and finally to distribute the functions that constitute the totality of the database management system.

The first step in the process to modularize and distribute Derby was to analyze its structure, its internal workings and the relationships between different parts of the code. As a result of this analysis it was possible to imagine how the existing codebase can be partitioned into different modules. This finally would make possible to achieve the goal of distributing database management functions like parsing, optimizing, etc. However, as the Derby System was not designed with distribution in mind it was not simply possible to move parts of the code into separate modules without changing its structure and refactoring it [31, 16]. For that reason, after deciding on how to split original Derby code into modules, it was necessary to make changes to the code. Only after those changes would Derby compile and work properly again.

Firstly, the changes in the original code, also called refactorings, will be described in detail. They will give a concrete overview of what had to be changed in the Derby codebase and how it has been done. Secondly, an attempt is made to group those refactorings and present them in a generalized and abstract form. The purpose of the second step is to give an outline of possible types of refactorings that have to be applied in order to modularize a system. It’s purpose is to provide a catalogue of general refactoring patterns. And thirdly, the changes will be presented in the more abstract form of principles that all the previously mentioned refactorings comply with. The idea is that a design following those principles is easier to modularize and distribute.

The last idea of abstract principles bridges this concrete project, whose purpose is to modularize and distribute Derby, and the generic challenge of building new systems or restructuring existing systems for distribution. The existing Derby
codebase had to be refactored and changed to finally comply with the abstract principles to make it distributable. At the same time designing a system with those principles in mind allows to distribute parts of the system; they comply with those principles by their design. Derby had to be retrofitted to comply with them to make it possible to modularize it and run it as a distributed system.

This hierarchy of refactorings, i.e. the concrete changes, the abstract ones and finally the principles represented graphically forms a pyramid as shown in figure 5.1.

![Figure 5.1: Consecutive tasks / work steps](image)

The following section will contain a catalog of the most important concrete refactorings of Derby. Describing all refactorings of Derby would make this chapter unreadable and too bloated. Besides, their description would not contribute to the understanding of this work. Not all refactorings will be therefore mentioned and described in detail here.

All envisioned refactorings could not be finished in the scope of this thesis. As a result, the envisioned solution was only partially achieved. To contrast the successful refactorings and those which could not be finished, the refactorings catalog will be divided into two sections: the successful and the unsuccessful refactorings. In case of the latter ones, details will be provided explaining why the envisioned changes could not have been achieved. This division will also make it clearer which parts of the plan described in chapter 3 section “Envision” have been achieved and which not. A higher level overview of what has been achieved will be given later in chapter 6.

5.1 Concrete refactoring catalog

All the concrete refactorings applied to the original Derby codebase will be presented using a common structure. This shall facilitate understanding and comparing the refactorings as well as refining them into a more abstract form presented in the next section. The common structure for their presentation will be the following:
5.1 Concrete refactoring catalog

<table>
<thead>
<tr>
<th>Name of the refactoring</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>A short description of the refactoring.</td>
<td></td>
</tr>
<tr>
<td>Problem statement</td>
<td></td>
</tr>
<tr>
<td>Reasons why the current situation is problematic and thus the reason why the current refactoring had to be done.</td>
<td></td>
</tr>
<tr>
<td>Solution description</td>
<td></td>
</tr>
<tr>
<td>Description of the refactoring and the solution for the problem. Code examples demonstrating in more detail how this refactoring is implemented. Where applicable also a comparison of old and new code.</td>
<td></td>
</tr>
<tr>
<td>Solution justification</td>
<td></td>
</tr>
<tr>
<td>Explanation why the refactoring solves the problem.</td>
<td></td>
</tr>
<tr>
<td>Old situation</td>
<td></td>
</tr>
<tr>
<td>(Optional element) Code snippet or illustration before the refactoring.</td>
<td></td>
</tr>
<tr>
<td>New situation</td>
<td></td>
</tr>
<tr>
<td>(Optional element) Code snippet or illustration after the refactoring.</td>
<td></td>
</tr>
</tbody>
</table>

5.1.1 Successful Refactorings

This section contains a catalog of successfully done refactorings.

<table>
<thead>
<tr>
<th>Composite/Visitor QueryTreeNode</th>
<th>qtn_vis</th>
</tr>
</thead>
</table>

Description
The class hierarchy starting with the QueryTreeNode is responsible for representing SQL operator trees. After the refactoring this hierarchy uses the Composite/Visitor patterns [12]. Firstly, the QueryTreeNode descendant instances represent the composite pattern holding the information. Secondly, the various tasks which have to traverse this structure to obtain results are now implemented as visitors following the Visitor pattern.

Problem statement
In the original version of Derby, the classes inheriting the QueryTreeNode directly contain the processing code necessary to implement tasks like the optimization or the code generation. This simplifies accessing the information stored in the operator tree. On the other hand, the code responsible for the various independent tasks is mixed in the same hierarchy. Hence the QueryTreeNode descendants mix various responsibilities at the same location. In this situation, distributing those responsibilities as remote services is not possible. It is not possible to have the code responsible for optimization in a separate module because it is colocated with the code generation, other processes and with the QueryTreeNode hierarchy. Hence it would not be possible to modularize and distribute optimization.
Solution description
The QueryTreeNode will be refactored so that it complies with the Composite pattern. It will implement a Visitable interface so that the tasks like optimization can be implemented as Visitors. This also allows the Visitors to be mutually independent which in turn allows for their modularization and distribution. Moreover, this would allow to purge the QueryTreeNode operator tree instances from data used only by the particular processing like code generation.

Solution justification
With this design, tasks like code generation or optimization operating on the QueryTreeNode can be implemented in an independent way. The only dependency is from the tasks to the operator tree instantiated from the QueryTreeNode descendants. Consequently, this makes the tasks easier to modularize and distribute.

Old situation
Figure 5.2 shows an abstract representation of a QueryTreeNode class. It contains the methods implementing the various processing done on the QueryTreeNode, the data required solely for them and the shared data.

<table>
<thead>
<tr>
<th>QueryTreeNode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specific Methods</td>
</tr>
<tr>
<td>Specific Data</td>
</tr>
<tr>
<td>Common Data &amp; Methods</td>
</tr>
</tbody>
</table>

Figure 5.2: QueryTreeNode before the refactoring

New situation
The QueryTreeNode has been changed so that it implements the QModelVisitable. The various processing tasks are now visitors traversing the operator tree (instances of the QueryTreeNode classes). They depend on the purged QueryTreeNode classes. The new situation is represented in figure 5.3.

| Code generation implemented as a Visitor | cdg_vis |
| Description |
The code generation is now implemented as a Visitor following the QueryTreeNode Composite/Visitor patterns.
Problem statement
The original Derby uses the `QueryTreeNode` class hierarchy and implements the code generation within this hierarchy. In this situation, the code generation can neither be modularized nor distributed (see qtn_vis refactoring).

Solution description
The code generation is now implemented as a `Visitor` visiting the operator trees (instances of `QueryTreeNode` descendants). The challenge for this refactoring is that almost every class from the `QueryTreeNode` hierarchy provides its own implementation of the code generation. A visitor providing the same functionality would become really large and complex. To solve this problem, a parallel class structure to the `QueryTreeNode` has been created which only models the code generation. The code generation visitor only forwards the requests to the adequate class from the parallel class structure. The creation of the code generation visitor and the parallel class structure has been automated using the CompilerTree API and the Java Annotation Processing API [13, 14]. Another challenge for this refactoring is that because code generation is now an external service, it has no access to the Derby store layer. As a result the new code generation service had to be purged from the store accessing code. The required information is prefetched before the actual code generation is done.

Solution justification
This refactoring allows for the distribution of the code generation as a remote service. The core of the Modular Derby database prepares the operator trees and sends them to a code generating service. Thanks to this refactoring it was possible to implement the code generating service using the code generation visitor. The code generation service now only generates code and does not need no access to the data in the database, i.e. the code generation responsibility has been purged from other tasks. Moreover, the reason why the code generation can be used as a remote service is that it directly sends the bytecodes of the generated classes as a result. The
classes are only defined on the receiver side, which obviously needs access to classes from the execution layer to define the generated classes.

Although this refactoring has been successful, there are some disadvantages to this solution that have to be considered:

• It is necessary to introduce parallel class structures for holding the actual code generation processing.
• In some cases, to implement the prefetching it is necessary to repeat control structures which otherwise was implemented only once.

Old situation
Before the refactoring the code generator was responsible for fetching data and generating code. This is a mix of responsibilities which make the distribution of the code generator difficult. It is depicted in figure 5.4.

New situation
The code generator is implemented as a Visitor and a parallel class hierarchy to the QueryTreeNode hierarchy. Moreover, it is only responsible for code generation. The data required for this task is prefetched before the actual code generation is done. Examples of such prefetching are the newly implemented PostOptimizerBinder and the PrepareCodegeneration visitors which prepare the operator tree. After those visitors process the operator tree, it can be sent to a remote code generation service (this refactoring) which has no access to the store layer. This new situation is shown in figure 5.5.
## 5.1 Concrete refactoring catalog

<table>
<thead>
<tr>
<th>Parser</th>
<th>parser</th>
</tr>
</thead>
</table>

### Description
Refactor the parser so that it is possible to run it as a remote service.

### Problem statement
Although the parser needs almost no changes for modularization, it heavily depends on data structures which cannot be serialized and sent as parameters to a remote service. The main data structure inhibiting distributing the parser is the CompilerContext. The reason is simply that this structure transitively references the entire database. Sending it over the wire would mean serializing the entire runtime structure of the database.

### Solution description
A solution to the described problem is the partitioning of the large CompilerContext into sub-structures which are used by functions like the parser. A partition of this structure used only by the parser is small enough to be sent over the wire as a parameter to the remote parser service. The partitioning of this structure can be done based on knowledge about how it is used and which parts of the structure are used solely by the parser. This, in turn, can be done by analyzing the interface protocol as described in chapter 4, section 4.2.

Unfortunately, it was not possible to finish the refactoring of the Parser and the required data structures completely. At the end of this project, the distributed parser only works partially, mostly for read-only queries.

### Solution justification
It is possible to prepare the parser service for distribution by partitioning large data structures so that only relevant information must be sent to it. Seen on a higher level of abstraction, this means that the large datastructures are partitioned according to the responsibilities. The parser only receives information relevant for the functions it provides and returns its results.

### DataDictionaryImpl dependencies
dd_glcf

<table>
<thead>
<tr>
<th>DataDictionaryImpl dependencies</th>
<th>dd_glcf</th>
</tr>
</thead>
</table>

### Description
DataDictionaryImpl depends on the GenericLanguageConnectionFactory. Change DataValueFactory creation in the DataDictionaryImpl to direct initialization instead of getting it through the LanguageConnectionContext.

### Problem statement
In the former version of the DataDictionaryImpl, the DataValueFactory is retrieved by first booting the LanguageConnectionContext, then retrieving it indirectly through the DataValueFactory and finally getting it from the LanguageConnectionContext.
Solution description
After the refactoring, the DataValueFactory instance is booted and returned directly in the DataDictionaryImpl.

Solution justification
The problem here is a reference from a service offered by a bundle from a lower layer to a layer above. In this case, the DataDictionaryImpl being part of the core bundle instantiates the LanguageConnectionContext in form of the GenericLanguageConnectionFactory. The LanguageConnectionContext implementation (GenericLanguageConnectionFactory) is located in the jdbc bundle above the core bundle. After the refactoring the responsibilities are more concentrated.

Old situation
Layering violation.

New situation
The DataValueFactory is located in the base bundle which is in a layer below the core. The DataDictionaryImpl from the core bundle now directly boots the DataValueFactory. Using this method booting a service implementation from the layer above, in this case the GenericLanguageConnectionFactory from the jdbc bundle is no longer required. Therefore the dependency on a bundle from a layer above is broken. This refactoring solves a layering violation.

<table>
<thead>
<tr>
<th>Interface usage</th>
<th>ifc_use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Use services by their interfaces, not their implementations.</td>
</tr>
<tr>
<td>Problem statement</td>
<td>In various places where services are required, the concrete implementations are referenced. This introduces unnecessary dependencies on implementation classes, possibly violating the layering. Those dependencies are not necessary because only the interface-specific functionality is used. Because Derby uses a central service management (the monitor), it is possible to get already created services and program solely against their interfaces.</td>
</tr>
<tr>
<td>Solution description</td>
<td>Reference solely the interfaces, as implementation specific features are not used. In some cases where the implementation of a service provided a function in addition to the interface, it was possible to pull up the function and enrich the interface.</td>
</tr>
<tr>
<td>Solution justification</td>
<td>This relatively low-cost refactoring doesn’t change how the services are used. It only removes the dependency on service implementations, hence making it possible to split the code into modules and introduce layering.</td>
</tr>
</tbody>
</table>
5.1 Concrete refactoring catalog

Situations

- StatementCache now uses the generic ExecPreparedStatement type instead of a concrete implementation
- IndexRowToBaseRowResultSet now uses ExecPreparedStatement instead of a concrete implementation
- CachedStatement instantiates the GenericPreparedStatement using the generic Statement type, not its implementation GenericStatement:
  
  old: ps = new GenericPreparedStatement((GenericStatement) key);
  new: ps = new GenericPreparedStatement((Statement) key);

- ExecPreparedStatement receives a method pulled up from a concrete implementation: GenericPreparedStatement.getStatement(). This enables using the interface ExecPreparedStatement instead of its implementation; as a result, the GenericPreparedStatement has to provide an implementation of this method.

- Statement gets a method from a concrete implementation GenericStatement.getCompilationSchema(); this enables using the interface instead of its implementation.

LanguageConnectionFactory

<table>
<thead>
<tr>
<th>LanguageConnectionFactory</th>
<th>lcc</th>
</tr>
</thead>
</table>

Description

LanguageConnectionFactory contains references to services which are only held to be passed to their users.

Problem statement

The original LanguageConnectionFactory contained references to services unrelated to its core functionality. In many cases its purpose was to hold the services without managing or changing their state. The mere holding of service is not a problem per se. However it becomes problematic whenever the holding of services introduces new dependencies, which in turn violate layering and inhibit modularization.

Solution description

Some services were only held to be passed to other parts of Derby. The refactoring was to retrieve them directly whenever they were needed.

Solution justification

This solution removes the unnecessary dependencies on services. In cases where this was done, the services are not managed but only held, hence there is no need to keep them in places where they are not needed. The monitor is now responsible for service management. This solution concentrates the responsibility of managing services at the monitor.
Situation
A concrete case was the JavaFactory and the ClassFactory. Instead of booting them within the GenericLanguageConnectionFactory, passing them then to the CompilerContextImpl only to be retrieved by the ExpressionClassBuilder they are now simply booted directly in the ExpressionClassBuilder. There was no need to keep references in so many places and always retrieve them from the CompilerContextImpl. The only problem was to provide appropriate access to the statement cache, formerly accessible through the ClassFactory.

<table>
<thead>
<tr>
<th>Context split</th>
<th>ctx_split</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Derby makes heavy use of Context classes. They contain a plethora of data. However not all data elements in the Context classes are related.</td>
</tr>
<tr>
<td><strong>Problem statement</strong></td>
<td>The data contained in Context classes is not always related. In fact it would be possible to partition the data within those classes into groups used by functional units like the parser, the code generator or others. This topic has been already described in more detail in chapter 4.</td>
</tr>
<tr>
<td><strong>Solution description</strong></td>
<td>Analyze the Context classes and determine how to partition them. Based on this analysis, create classes containing the partitions of the data which are used by the functional units.</td>
</tr>
<tr>
<td><strong>Solution justification</strong></td>
<td>This solution partitions the Context classes into parts which are coherent. They can be passed to functional units. Most importantly, instances of the newly created classes (partitions of Context classes) contain only the data required for a specific function. Hence it is possible to pass them to remote services without unnecessary overhead. In some cases they make it possible for a service to be called remotely; the original Context classes transitively contain almost the entire database. It would not be possible to pass instances of such classes to remote services.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shared information element or function</th>
<th>shared_elem</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>This description represents a group of refactorings. They were necessary to break down circular dependencies between modules.</td>
</tr>
</tbody>
</table>
| **Problem statement** | Two modules reference each other to exchange information or to use a
common function. An example of this are the original DRDA and tools modules. The DRDA module provides functions allowing to use Derby as a standalone SQL server. The tools bundle is responsible for starting drda. Hence there is a dependency from the tools to the drda module. The complication is that tools contain functions and data providing localized input and output required by the drda module. This situation is a circular dependency between modules.

Solution description
Analyze which is the common part of both modules and either move it into the other module or create a third module which will only contain this element. In case the element used by both modules from the circular dependency is a concept possibly used elsewhere, creating a dedicated module gives the concept and identity and resolves the circular dependency. If the element is not a stand-alone concept it can be moved into the other module to resolve the dependency.

Solution justification
The solution resolves the circular dependency as desired.

Old situation
Figure 5.6 represents the old situation before the refactoring.

![Circular dependency before refactoring](image)

Figure 5.6: Circular dependency before refactoring

New situation
Figure 5.7 shows the situation with the resolved circular dependency after the refactoring.

![Resolved circular dependency](image)

Figure 5.7: Resolved circular dependency

5.1.2 Unsuccessful Refactorings

This section contains refactorings which could not have been completed in the scope of this thesis. Unfortunately, it was impossible to fully implement what has been envisioned. The most important parts which have not been implemented will be described in this section.
Optimizer  The code responsible for operator tree optimization is scattered around and partially contained in the `QueryTreeNode` class hierarchy. The desired envisioned solution is to create a concentrated and independent optimizer unit of functions which would take care of operator tree optimization task. However, the optimizer requires access to the store to retrieve statistics information about stored relations and data. However the store has only been modularized but not distributed. Although modularizing and distributing the optimizer would have been possible, due to the lack of an independent Store service it would not have been possible to execute it independently from the core. Pursuing the modularization and distribution of the optimizer was therefore not useful.

QueryTreeNode cleanup  The purpose of this refactoring was to purge from code responsible for processing tasks like the optimization or data binding the `QueryTreeNode` class hierarchy. The purged `QueryTreeNode` class hierarchy could then more easily be used as an independent module whenever operator trees are processed. Failing to do so doesn’t inhibit distribution of the functions. Nevertheless it has the flaw that whoever wants to receive operator trees for processing, requires all dependencies, e.g. to the optimizer and transitively the store. They are not used actively, but only required as dependencies, or are in other words only dead code. This for example results in a situation where the remote service providing code generation for Derby requires the StoreAPI and StoreImpl bundles along with the encryption bundles to start even if their functions are never used.

Indexing and Transactions analysis  Due to the concentration on other tasks it was not possible to do an in-depth analysis of the indexing and transaction management. Analyzing those areas and envisioning how they could be modularized and distributed would have provided interesting and important information on how those efforts can be pursued.

5.2 Abstract refactoring catalog

The concrete refactorings described above can be grouped into abstract refactorings which can be applied in different situations. After abstracting away situation specific information the following abstract refactorings have been found:

Visitor pattern introduction  Use the `Visitor/Composite` patterns for complex structures like the `QueryTreeNode`. The `QueryTreeNode` instances are operator trees representing SQL queries. In Derby the `QueryTreeNode` class hierarchy contains both data and methods which can operate on those trees and process them. Although this design has its advantages, (the data can be easily accessed and the methods can access private members of the classes), it also has major flaws. The various processing methods have to be colocated, bloating the classes. In addition, adding new processing introduces new dependencies between the structure and the processing methods and dependencies. To see the consequences of this design, the following case is considered: The `QueryTreeNode` structure has various processing methods in its structure. Some of those methods are related
5.2 Abstract refactoring catalog

to code generation, others to optimizing. If now the code generation is
refactored and distributed as a remote service, as it requires the Query-
TreeNode classes for data, it automatically has a dependency on classes
related to optimization. This makes it necessary to include the depen-
dencies on optimization code even if is never used by the remote code
generation service.

Refactoring complex structures to comply with the Visitor/Composite
patterns resolves this problem. The processing methods that need to op-
erate on the composite structures can be implemented as separate visitors.
Whenever one of those processing types needs to be provided as a remote
service, it is only required that the composite object tree be serializable.
No unnecessary dependencies are created.

Double Dispatch used to distinguish Types Use double dispatch in or-
der to distinguish between types. This technique allows easy distinguish-
ing among decisions based on types. It is not necessary to implement
lengthy instanceof if statements. They are firstly tedious to implement,
secondly hard to understand and thirdly brittle because adding a type to
the type hierarchy requires changing every concerned if statement. Failing
to do so will almost certainly lead to introducing run time errors.

To illustrate why this can happen, the QueryTreeNode type hierarchy
is considered. There are already about 120 types inheriting the Query-
TreeNode and forming a complex inheritance tree. Type based decisions
are done in at least 3 different locations in each type. First is the binding
based differently depending on the type. The second is optimization and
the third code generation. In all locations different behavior is required
for every type. Using multiple if (t instanceof Type) would require very
long if statements with always about 120 branches. Adding a new type
necessitates finding all those complex if blocks and adding the type if nec-
essary. It might easily happen that not all those if blocks are found and
it is forgotten to add the new instanceof if statement.

In case of double dispatch it is enough to add the newly added type to the
Visitor interface. All its implementations will have compile-time errors
which is obviously easier to track and to find all places where the new
type related behavior has to be implemented. This solution is less brittle,
easier to understand and also less tedious to implement.

Data responsibility Redesign the data structures so that they contain data
used by single functional units. This allows sending instances of those data
structures to services without creating dependencies on others. Moreover
such single responsibility data structures can be used as arguments to
remote services.

Program against interfaces Use interfaces instead of concrete implementa-
tions. If a concrete implementation is used, analyze why and if possible
push the operation into the interface and don’t use the concrete imple-
mentation anymore. Instead of instantiating the concrete interface im-
plementations directly, use an indirection like IoC (Inversion of Control)
or a instance factory which hides the concrete implementations away. In
case where methods or implementation specific features are used, consider pulling the features up into the interface.

5.3 Design principles

Based on the concrete refactorings, later distilled in the abstract refactoring catalog it is possible to deduce the principles that lie behind all the changes. In all those cases described previously, the problems that had to be solved in order to be able to modularize and distribute Derby stem from violating the Single Responsibility Principle [15]. This means that one responsibility is handled by code in one single location and the code in one location shall not be responsible for multiple responsibilities. In spite of a modular design and dedicated units providing services within Derby, the original code generally violates this principle. This violation makes it difficult to modularize Derby and distribute its functions. The restructurings that have to be applied to Derby make sure that the Responsibility Principle is not violated.

More concretely, the Single Responsibility Principle manifests itself in case of Derby as a Process and Data Responsibility Principle. The Process Responsibility means that a single process is handled by code in one location and no other processes share the same location. The Data Responsibility is analogous to the Process Responsibility with the Data stored in the same runtime structures.

Below there are two examples that illustrate how the Single Responsibility Principle is violated in the original Derby codebase:

**Single Process Responsibility:** The original Derby uses a Class hierarchy rooted at the QueryTreeNode for modeling Relational Opearator trees. The class hierarchy contains the metadata about the operators and in addition to that it provides methods which allow optimizing the operator trees, binding them to data structures and generating code which can be executed to actually retrieve the data. This mix of operations co-located in descendants of the QueryTreeNode class violates the Single Process Responsibility Principle. Refactoring the QueryTreeNode hierarchy so that the binding, optimizing and code generating processes are located in separated locations makes possible to comply with the Single Process Responsibility principle. As a result it is possible to modularize and distribute those functions.

**Single Data Responsibility:** Derby uses the Context class and its descendants to convey data across different subsystems and method calls. The context instances are used by the different subsystems to communicate within and across them. However the Context descendants contain data from different domains, e.g. the CompilerContext contains data used by the parser but also data used by the optimizer and the code generator. This violates the Single Data Responsibility principle. The data has to be separated into different non-interleaving contexts which can be used by the mentioned processes. For cases where the data has to be exchanged between subsystems, dedicated context or communication objects should
be introduced. Refactoring the context so that they contain only single purpose datasets makes possible to modularize and distribute Derby.

5.4 Conclusion

Derby had to be restructured to make its modularization possible. This was also necessary to allow its distributed operation. The restructuring of Derby meant refactoring its codebase. Without refactoring parts of the codebase, modularization would have been impossible. Reasons for this were circular dependencies, violation of function layering and usage of specific interface implementations where the interface usage would be possible. The concrete refactorings have been grouped and presented as a catalog of abstract refactorings. Moreover it was possible to find principles which were behind all refactorings. The concrete and abstract refactorings as well as the principles form together a pyramid presented in figure 5.1. The principles found manifest themselves as the Single Data and Process Responsibilities.

However the most important implication of the found principles is that we believe that the refactorings which would have to be applied in order to further modularize Derby and distribute its functions would all follow these principles. From another perspective, the violation of these principles makes modularization and distribution difficult. Following these principles during design time of a system should make its future modularization and distribution easier.
Chapter 6

System design

A general overview of the achieved solution will be provided in this chapter. The purpose is to describe the system of Modular Derby and give more information about specific parts relevant for this project. It describes how the original Derby has been modularized and how its functions have been distributed. Moreover, important design decisions and their implementation will be put under the spotlight. Although the original codebase has been refactored, the RDBMS specific functions did not change and the code implementing them behaves equivalently.

6.1 Abstract OSGi principles

Modular Derby uses OSGi technologies for modularization and the R-OSGi technology for distribution of its functions. The challenge of OSGi is to use the right means out of a plethora of possibilities offered to solve the problems. For the purpose of this project some abstract principles have been established, which have been followed as a guidance to achieve the results. Their purpose is to build a uniform way of how to use OSGi, thus making it easier to understand what has been done and lowering the burden for further development. The abstract principles are the following:

Split the service API and the implementation. The public service API should be located in a separate bundle from its implementation. Users of a service use its API by importing the code from the API bundle. The concrete service implementation is retrieved using the OSGi service registry. The service consumer doesn’t instantiate the service implementation but retrieves it from the registry. Hence he does not have to know its implementation.

Avoid start ordering dependencies. When using direct service lookup, it must be possible for a bundle to retrieve the required services while it is starting. This works only if all the bundles providing the required services are started beforehand. In case a service has not yet been registered at the time it is requested, the service requester (consumer) will get a null
reference to the service. The consumer will not be able to start properly. A better solution to get services is to use ServiceTrackers. They make it possible that a service consumer will be notified once the service is available. Once the service becomes available, the framework will inform interested service consumers that the service is available. This approach is less error prone and reduces the necessity to write exception handling code.

6.2 Modular Derby bundle structure

In this thesis Derby has been modularized and split into modules (OSGi bundles). Figure 6.1 shows the current modules that precisely constitute the Modular Derby. The dependencies between the modules are represented as arrows. Thanks to the refactorings, it was possible to achieve the presented modularization. Nevertheless, due to the refactorings that could not have been completed, some modules still have dependencies on others. This leads to a situation depicted in figure 6.2. To execute the Parser remotely, it is necessary to include the module containing the store code even if this code is never used.

![Figure 6.1: Modular Derby modules](image)

6.3 Modular Derby query processing

The SQL processing phases and their order have not been changed in Modular Derby. Some of the query processing phases have been distributed. In Modular Derby they are used as remote services. The services that have been successfully distributed are the CodeGenerator and the Parser. The Parser functions, however, have only been restructured so that they support read-only queries when executed as a remote service. The distributed SQL query processing situation is represented in figure 6.3 (it shows the processing in a simplified way: it omits that query processing is controlled centrally from the Derby core).
6.3 Modular Derby query processing

Figure 6.2: Modular Derby modules

Figure 6.3: Distributed SQL processing setup
6.4 Module registrar service

The original system is designed around fine granular services represented as interfaces. Whenever a service is required within Derby, the service consumer asks the Monitor for a service provider. In most cases the actual implementation of the service remains hidden. The service consumer does not need to be aware of the class implementing a service interface. Such a design makes it possible to adapt to the environment easily. The monitor is responsible for selecting the right implementation for a service based on the requirements and the environment. Which implementations are available is controlled through a property file named “modules.properties”. Service implementors are registered for their interfaces in a “name=implementation_class” manner. Listing 6.1 shows an excerpt of this file.

```
1 derby.module.dvfCDC=org.apache....CDCDataValueFactory
2 derby.module.database=org.apache....BasicDatabase
3 derby.module.rawStore.trn=org....XactFactory
4 derby.module.streams=org.apache....SingleStream
```

An important detail of this design is the fact that the Monitor instantiates the service implementors and manages them. Service providers only provide the class which is instantiated by the Monitor. The Monitor is a class which is known by almost every part of Derby. This situation is depicted in figure 6.4. The Monitor is at the very core of Derby. It instantiates the multitude of services available in Derby.

![Monitor instantiating services](image)

Figure 6.4: Monitor instantiating services

In Modular Derby this design however leads to a problematic situation. Being at the very core of Modular Derby, the Monitor doesn’t necessarily have access
to all classes providing an implementation of a service. This situation appears because of the layered structure of Modular Derby. Some service implementations are now located in layers above the layer of the Monitor. Allowing the Monitor to access those layers would result in circular dependencies from the very bottom (or core) to upper layers and back. This is firstly undesired and secondly unallowed by OSGi. The situation with layered Modular Derby and the Monitor at the core is shown in figure 6.5.

A solution to this problem is provided by the newly introduced concept of the `ModuleRegistrarService`. It allows service providers to register their service implementations. The `ModuleRegistrarService` keeps a list of service implementors and makes them available to the Monitor. In this solution the Monitor is not obsolete, but the task of knowing the implementors is deferred to the `ModuleRegistrarService`. The monitor is still responsible for instantiating the services providers, however, it now does it dynamically without knowing the implementation classes directly. Service providers register their implementation classes by the service interface names. Later the Monitor retrieves the implementing classes, instantiates them and returns the service instances to the service consumer. This situation is depicted in figure 6.6.

This design makes possible to register a service implementation from a given layer and make it available to service consumers on other, possibly lower, layers. In the case of Modular Derby it also allows to keep the existing design untouched from the perspective of Monitor users. The Monitor implementation had to be changed, but its interface and usage remains the same, allowing for a smooth transition phase towards modularity and distribution. For the same purpose it might be possible to use the standard OSGi approach for service registration. Such an approach would solve the problem. However, it would require a complete redesign of the Derby service system. This, however, is not a viable solution, because doing so would exceed the scope of this thesis.
6.5 Two Phase bootstrap

In Modular Derby the entire functionality is partitioned into modules (OSGi bundles). Generally bundles require service from other bundles to function properly. To start a system it is necessary to bootstrap all bundles in the correct order. Failing to do so causes bundles not to start properly as some required services might not have been registered yet. OSGi provides a simple mechanism that solves the bootstrap problem. Each bundle is assigned a start level. The OSGi framework then starts bundles according to their start level. Bundles with start level 0 will be started first, level 1 bundles afterwards and so on.

For a complex system like the Modular Derby, however, this is challenging, especially when the functions are separated into modules one by one, resulting in a ever changing bundle structure. Using the start level mechanism requires the developer to assign start levels every time a new bundle is added to the system. Likewise it is necessary to consider start levels for the part of Derby which provides some remote services like the code generation or parsing. All the bundles used to provide those services need also be assigned start levels.

Overall this is a tedious, uncreative and time consuming process. To provide a remedy to this problem, a solution has been designed. This solution allows starting all bundles on the same start level, yet have the system bootstrap correctly independently on the start order of the bundles. To that end the bootstrapping of each bundle has been split into two phases, hence the name “two phase bootstrap”:

1. Request required services: In this phase, the bundle requests necessary services from the Service registry. Here, thanks to the standard OSGi
ServiceTracker mechanism, all already registered services will be provided. The services not yet announced will not be requested. This means that no “null” service references will be retrieved. The important part is that this “Request” phase is not accomplished unless all required services have been retrieved.

2. Announce provided services: In this phase, the bundle registers services provided by itself with the Service registry. This phase will only be entered once all the dependencies (in form of required services) from the previous phase are satisfied. The fact that this phase will not be entered unless the dependencies are satisfied allows this bundle to start all its provided services correctly, as all required services are present.

A previous section, 6.1 described an abstract principle called “Avoid start ordering Dependencies”. Although this is a part of the two phase bootstrap, it does not solve the problem altogether. Avoiding start order dependencies requires using the ServiceTrackers which will be informed about services once they are available. However, it does not prevent a bundle offering its services before all required services have been acquired. This in turn causes problems as some required services might not have been acquired before own offered services are consumed. As a remedy, the two phase bootstrap mechanism has been used.

For a bundle to take part in the two phase bootstrap with other bundles it is necessary to implement its Activator as shown in listing 6.2. The important ingredient is the satisfier. The satisfier takes care of bookkeeping which required services have become available. The satisfier is finally responsible for triggering the second bootstrap phase, i.e. announcing own services. Important excerpts of the satisfier are shown in listing 6.3. The satisfy method has to be called every time a required service is acquired. This is a hint to the satisfier that such a service is available.

Listing 6.2: Two phase bootstrap Bundle Activator

```java
class BundleActivator {
    public static BundleContext ctx;
    private Satisfier satisfier;

    public void start(BundleContext ctx) {
        BundleActivator.ctx = ctx;
        satisfier = new Satisfier(
            new Class[] {
                // all service classes of required services:
                Foo.class,
                Bar.class
            },
            new SatisfierCustomizer() {
                // Second Phase (announce services)
                public void onDependenciesSatisfied() {
                    // announce provided services
                    ctx.registerService(AService.class.getName,
                        new AService(), new Hashtable());
                }
            }
        );
    }
}
```
// First Phase (request services)
// register a foo tracker
ServiceTrackerCustomizer fooCust = new FooTrackerCustomizer();
ServiceTracker fooTracker =
    new ServiceTracker(ctx, Core.class.getName(), fooCust);
fooTracker.open();

Listing 6.3: Satisfier excerpts

public class Satisfier {
    ... public void satisfy(Class dep) {
        boolean result = true;
        for (int i = 0; i < deps.length; i++) {
            if (deps[i].dep.isAssignableFrom(dep)) {
                deps[i].sat = true;
                if (!result) {
                    return;
                }
            } else {
                result &= deps[i].sat;
            }
            if (result) {
                satisfierCustomizer.onDependenciesSatisfied();
            }
        }
    }
    ... 
}

The following example explains the two phase bootstrap. The figure 6.7 shows
the bundles, their services and dependencies. Independently of the exact start
ordering, the bundle dependencies will be wired correctly thanks to the mech-
anism described above.

To demonstrate the bootstrap mechanism, the start ordering as shown in figure
6.8 is considered. This results in the following sequence of operations and system
states:

Currently available services: ()
1. Bundle JDBC is started.
   Request services:
6.5 Two Phase bootstrap

Figure 6.7: Two phase bootstrap situation

Figure 6.8: Two phase bootstrap - Example 1
2. Bundle Base is started.
   Request services:
   > all satisfied, enter phase two
   Offer services:
   - LocaleFinder, Kernel
   > JDBC request for the LocaleFinder is satisfied

Currently available services: (LocaleFinder, Kernel)

3. Bundle RunModular is started.
   Request services:
   - Driver
   Offer services:

Currently available services: (LocaleFinder, Kernel)

4. Bundle Core is started.
   Request services:
   - Kernel
   > all satisfied, enter phase two
   Offer services:
   - Core
   > JDBC request for the Core is satisfied
   > JDBC enters phase two
   > JDBC announces the Driver
   > RunModular request for the Driver is satisfied
   > RunModular enters phase two

All bundles and services available.
Currently available services: (LocaleFinder, Kernel, Core, Driver)
Chapter 7

Software engineering aspects

This chapter describes the software engineering aspects of this Modular Derby thesis. This work was not only a theoretical consideration of modular relational databases, but a concrete attempt to analyze, restructure and finally modularize Derby. For a successful realization of such a project software engineering challenges had to be thought about. Those challenges were less about modularizing an existing database management system, but rather about the way to achieve the envisioned goals in an efficient and pragmatic way. Finding solutions to those challenges would at the same time allow mitigating the risks related to modularizing existing Derby codebase. Moreover the same considerations could be applied to another modularization and distribution problem unrelated to this work.

This chapter will describe how the various possible solution paths have been explored, how the system has been tested and restructured in a test driven way [32]. Furthermore, a section is devoted to packaging and naming conventions used to demarcate new and existing code as well as markers used to group changes to existing code into topics.

7.1 Solution space exploration

Analyzing, restructuring, modularizing and finally distributing Derby was not always a task easy to predict and to plan. In some situations it was possible to achieve the envisioned results in the planned time, in other situations it was not. Restructuring and changing an existing system introduced the challenge of knowing when to abandon an approach early enough to concentrate on other doable tasks. More specifically it was necessary to know whether a certain task or step in the process of solution finding was a blind alley, because its consecutive steps are not achievable. This situation is shown in figure 7.1. The green branches are solution finding steps (tasks) that can be achieved. The red ones are tasks which are not achievable and where after a preliminary
analysis it is obvious that they are not. The black ones are the tasks that are the challenge. They can be achieved, but finally their consecutive tasks are not achievable. Such tasks are blind alleys. In order not to lose time for such tasks it is necessary to abandon those paths early enough and save the time to concentrate on those which can be achieved in useful time.

![Figure 7.1: Solution space exploration](image)

To foresee early enough which paths of consecutive tasks lead to expected results and are not blind alleys the following approach has been used: First, a plan of consecutive tasks has been envisioned. In the following step the first task is analyzed and parts of the planned work are done as a proof of concept. Neither the analysis nor the work is done completely. Afterwards the second consecutive task is analyzed and parts of the planned work is done. However only a smaller proportion of the total analysis and work of the second consecutive task is done. The purpose is not to finish the work completely, but only to see whether it will be possible to complete the planned task. As a next step the third task on the path is partially analyzed. This shall allow to plan it and to foresee whether it will be possible to complete it. This partial analysis and actual work done on the path of consecutive tasks are shown schematically in figure 7.2: the sections labelled with “Step x” represent the total amount of work to be done. In case of “Step 1” only a third of the total work has been done and two thirds have been analyzed. Task labelled “Step 2” has been analyzed up to its third. Only a small proportion of work has been done as a proof of concept. For the third task only the first analysis steps have been accomplished to see whether it is feasible.

![Figure 7.2: Consecutive tasks / work steps](image)
This approach makes it possible to abandon paths of tasks that are blind alleys early. Using this approach it was possible to reduce the amount of work which didn’t contribute to the final solution.

7.2 Test driven development and refactoring

Adding new features and changing small parts of a system can possibly break it. Changing and restructuring an existing system like Derby will almost certainly lead to problems and newly introduced bugs, making the system unusable. To mitigate the risks of introducing bugs and breaking the restructured system a test driven development [6, 5] and refactoring approach [11] has been used. Before restructuring and refactoring the Derby engine for modularization, tests have been developed. Whenever a given feature of Derby had to be refactored for modularization, first tests have been implemented which use the feature and demonstrate its correct behavior. This makes sure that all new and refactored features are actually tested. Moreover, it demonstrates that the Modular Derby system still behaves the same as the original one. Without rigorous testing this wouldn’t be possible.

For the implemented tests the following assumptions have been made:

- **System tests**: It is only necessary to implement overall system tests because unit testing is already covered by the standard Derby test suite. The newly implemented system tests are SQL statements which test the overall functioning of the system and provide data which can be used to compare the original and the modularized Derby.

- **Positive tests only**: The implemented tests cover the correct functioning of the system. System behavior in error conditions is covered by the standard test suite.

It is possible to argue that the original Derby test suite could have been used to test the behavior of the Modular Derby system. With the tests implemented solely for the purpose of this project, it was possible to test slices of the whole system. This means that not only single functions (like in unit testing) have been tested but entire execution paths encompassing different functions and parts of the system. An example of such a system slice is the execution of a query containing the coalesce operator. Executing it involves different aspects of the database, i.e. the SQL parser, binder, optimizer, code generator, the execution layer, indexing, transactions, the storage layer and the coalesce function implementation. A system slice test is not about testing those functions in isolation but the cooperation of those functions even if they encompass only parts of function groups, e.g. only specific parts of the parser are tested.

Such system slice testing made it easier to see the results of specific changes done to the system. It also allowed isolating the sources of problems, i.e. which refactoring caused bugs in the refactored engine. This in turn reduced the time spent on debugging and correcting introduced bugs. In addition to the normal debugging it was also possible to compare execution of the original and the
refactored system. Whenever bugs were introduced due to refactorings it was possible to run the failed test against the original and the modular system. Then, the execution paths of both systems can be compared with respect to the refactoring done, to see where a bug has been introduced. Comparing the behavior of the original and the refactored system helped reducing the time necessary to find and fix the problems even more.

Altogether the refactoring testing, debugging, comparing of the results and the execution paths of the systems can be wrapped up in a cyclic process of system modularization. This cyclic process has the following structure:

1. implement a test (skip this if there is already a specific test)
2. run the test against modular Derby
   -> compile problems
   - refactor to get rid of the problems
   -> bugs
   - run original Derby to see the difference
   - run modularized refactored Derby code
   - compare execution
   - get rid of newly introduced bugs
   -> everything ok, tests passed
   go to step 1

To simplify the development of the tests a test running framework has been developed. The framework allows writing single SQL queries in text files and executing them. In addition to that it renders the results for the purpose of comparing them against the results of the original reference system, i.e. the original Derby.

An example of this approach is a case where the tests are used to first detect compile problems and in a later stage to make sure that no bugs have been introduced. The situation is the following:

- The `QueryTreeNode` implements a `QModelVisitor` which requires the concrete descendants of the `QueryTreeNode` to implement a method. Without implementing this method the test which uses directly the `CreateAliasNode` class is executed, which in turn results in the following message

```
Caused by: java.lang.Error: Unresolved compilation problem:
The type CreateAliasNode must implement the inherited abstract method QModelVisitable.accept(QModelVisitor)
```

- Next step is to continue the implementation or the refactoring to see the code compile correctly. In this stage bugs are possible. After debugging of the refactored features the test can be rerun and the results of the test can be compared to expected results. Once the results are as expected it is possible to move forward.
7.3 Code demarcation

The Modular Derby project is based on the existing Derby codebase. In most cases the work done consisted of thorough analysis of the original system, restructuring its code and modularizing its functionality into independent services. In some cases it was also necessary to write completely new code. Such code was required to support the new modularized system architecture.

To make it possible to distinguish between new and original restructured code, the new code has been put into different packages. In other cases where only small changes had to be done, the existing and new code have been marked. Overall the following conventions have been used:

- **Original Code:** Remains in original packages. However to improve readability and make it possible to compare what has been refactored, important changes have been marked with tags according to conventions described in Section 7.4. In addition to that the original code has been commented out and new code marked as new. This makes it possible to easily compare what has been changed without the necessity to consult revision changes stored in the version control system. A template of the new / old code distinction looks as follows:

  ```
  // OLD:
  // <original-code>
  // NEW:
  <new-code>
  // END
  ```

- **New Code related to original functions:** New code related to existing Derby functionality was placed in packages whose names were derived from the original packagenames by prefixing with `ch.ethz.systems` and abbreviating `org.apache.derby` to only `derby`, e.g. the new `CodegenService` interface, related to the original `ClassBuilder` interface from the `org.apache.derby.iapi.services.compiler` package was placed in the `ch.ethz.systems.derby.iapi.services.compiler` package.

- **New Code not directly related to original functions:** New code implemented for the specific purpose of modularizing the original Derby system, but not directly related to a specific original functionality has been placed in packages prefixed with `ch.ethz.systems.derby`

Not only do those techniques allow to distinguish what has been added or changed to Derby, they also cater for easier familiarization with the internals of Modular Derby.

7.4 Change tracking

In Modular Derby, all changes have been marked with specific short names. More specifically, not every change has been given a unique short name, but
the changes have been clustered and the clusters have been named. The Eclipse IDE [10] used for the implementation of this project was set up to detect those short names and provide a list with all places where the code has been changed. This allows to locate faster the places in the codebase where changes have been made. On one hand it facilitates the location of places where changes to the original code have been done and on the other hand it also makes it possible to learn about the internals faster. The Eclipse IDE gives an overview of those tasks, but in case another tool is used, it is simply possible to grep [30] the sources for the specific tags. The figure 7.3 shows a screenshot of the Tasklist view in the Eclipse IDE. The complete list of change cluster names along with a description of work done in the clusters is:

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODEGEN</td>
<td>Separate the code generation as a service.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>Context spliting.</td>
</tr>
<tr>
<td>DERBYNET</td>
<td>Prepare the DRDA Derby network driver adapter for modularization.</td>
</tr>
<tr>
<td>LAYER</td>
<td>General layering restructuring.</td>
</tr>
<tr>
<td>MDERBY</td>
<td>First high level Derby modularization efforts. Prepare all modularization and distribution of specific modules like parser, optimizer, code generator.</td>
</tr>
<tr>
<td>MODJDBC</td>
<td>Separate the jdbc code from the rest. Makes possible to use the code generation as a service without removing binding code from the QueryTreeNodes.</td>
</tr>
<tr>
<td>OPTIMIZER</td>
<td>Optimizer modularization and distribution.</td>
</tr>
<tr>
<td>PARSER</td>
<td>Move the parser into a separate bundle and provide a parser service.</td>
</tr>
</tbody>
</table>
7.4 Change tracking

STORE
Separate the store code and refactor as a module.

VISITOR
Refactoring of the QueryTreeNode and its descendants to comply with the Composite/Visitor patterns [12].
Chapter 8

Generic Process

Apart from describing how the specific task of modularizing and distributing Derby has been solved and how the corresponding process is structured, the solution has been brought to a more abstract level. This laid the basis for developing an abstract process which can be applied on other problems of modularizing and distributing software systems.

To find and describe such a process, it was necessary to reconsider what has been done with Derby to achieve its modularization and distribution, then to abstract away Derby specific details and finally to find the abstract process structure. This generic process can be described on different levels of abstraction. On the highest level, its description would simply be: build a comprehensive test suite for the system and don’t be afraid to change the system. The test suite will assure that the changes done to the system did not change its behavior. This very abstract description is however not very useful without more specific details of the process. On a lower level the process can be described as follows:

1. Understand what the system has to do. Do this by understanding the requirements of the system and the theory about it.
2. Analyze and understand how the system accomplishes its task. This can be done by analyzing the internal structure and behavior of the system. Concretely this can be accomplished by reading the sourcecode of the system and by analyzing its runtime behavior.
3. Restructure the system according to responsibilities.
4. Modularize the system.
5. Distribute the functions of the system.

The generic process is driven by understanding the requirements and the theory of the system. Furthermore, it is important to consider how the system realizes the requirements. This means it is important to know its internal structure, processes and its details. Knowing this permits to envision how the system can be modularized. The modularization, however, cannot be accomplished just by
Figure 8.1: Generic modularization and distribution process
moving and grouping parts of the system. As the internal structure might have cyclic dependencies and be generally intertwined, it is necessary to restructure and refactor the system. Only after a restructuring of the system it is possible to create modules covering partitions of the overall system. Most refactorings done in this work are based on the abstract responsibility principle described in more detail in chapter 5.3. Once the system is restructured it can be modularized (see chapter 1 for the distinction between modularization and distribution). In a further step the modules of the system can be distributed. This requires that the services can be executed separately and called remotely. For that purpose it is necessary that the data exchanged is serializable, i.e. it can be transformed in a binary representation. It is also necessary that all information is passed explicitly. This because shared state cannot be accessed as easily as in monolithic systems. It should also be minimized in order to reduce communication overhead.

During the entire process it is necessary to test the changed parts of the system. Changing too much without testing might result in a situation where a lot of problems introduced during the restructuring accumulate. In such a situation it might be difficult to find the causes of the problems and make it difficult to recover from the problematic state of accumulated bugs.

The important idea is that the modularization and distribution process is driven by creative thoughts and intelligent decisions based on the knowledge of the system, while the tedious tasks can be simplified by existing or custom developed tools. The tools cannot drive the process but only support it. A graphical representation of this generic process is shown in figure 8.1
Chapter 9

Evaluation

The results of this project have been evaluated in two different ways. On one hand, functional tests have been done. On the other, the performance of the original and Modular Derby has been measured and compared. The main purpose of this thesis was to show that it is possible to modularize and distribute the SQL processing of the Derby database engine without changing its functional behavior. To that end, the original and modular version of Derby have been tested with a functional test suite and the results have been compared (more information about the test suite can be found in chapter 7).

9.1 Functional Tests

The test suite developed for the purpose of this project encompasses database setup, querying the database in various ways, modifying the data as well as the relations and meta-data. To evaluate the solution developed in this project the test suite was executed against the original monolithic Derby. The results of the test suite have been stored for later comparison. In a later step the modular database was tested using the same tests. The test results then have been compared: the results yielded by the Modular Derby are the same as those yielded by the original monolithic Derby. In a last step the Modular Derby has been tested in a setup where the code generation was executed remotely. Also in the third case the results were the same as in case of the original Derby.

In addition to those tests Modular Derby has also been tested with the parser and the code generator used as remote services. As described in the “Concrete Refactorings” section of the chapter 5 it was not possible to fully restructure the SQL parser. As a result the current remote parser is only able to handle read-only queries. Nonetheless to be able to test with this limitation taken into account a reduced test setting has been used. The test setup encompassed solely the read-only tests from the entire test suite. In the reduced setting the parser and the code generator have been used as remote services by the core of the database engine. Also in this setting the results yielded by the original and the distributed Derby were the same.
The tests done with the monolithic, modular and distributed Derby demonstrate that the functional behavior has not been altered by the restructuring, modularizing and distributing of its functionality. Modular Derby still behaves the same as the original Apache Derby. The test results demonstrate that it was possible to modularize the SQL parsing functionality and distribute parts of it without changing its functional behavior. It was possible to use the code generation as a remote service with respect to the entire test suite. Using the parser as a remote service was successful for all read-only tests. The test setup with the code generation and the parser used as remote services is depicted in figure 9.1. It shows in a simplified manner how a SQL query is processed on Modular Derby in a distributed setting.

![Figure 9.1: Distributed test setup](image)

### 9.2 Performance Tests

Basic performance evaluation has also been done with Modular Derby. The performance evaluation measured the number of transactions which completed per second. For the performance evaluation the following setups have been used:

First run of all measurements has been done with the statement cache enabled, the second run with the cache disabled\(^1\). In both cases, i.e. with cache enabled and disabled the performance of the original monolithic Derby has been measured along with Modular Derby. The latter has been measured with all modules on one JVM and also in a distributed setting with the code generator executed as a remote service. The different measurements done are represented in figure 9.2. The test setup used for the distributed situation is shown in figure 9.3.

The measurements with the statement cache enabled show that in average all three versions of Derby have the same performance. The reason for this is that

---

\(^1\)The statement cache is responsible for storing already parsed, bound and optimized operator trees for later use. It is used for normal and prepared statements alike. It allows reusing already processed operator trees for queries that have already been processed.
the modular version of Derby in a non-distributed setting accomplishes its tasks in the same way as the monolithic version. Its restructuring for modularization should have no major impact on performance, which is shown with this performance measurement. Moreover the fact that Modular Derby uses a OSGi framework should not change its performance. The OSGi frameworks are not intrusive, i.e. although they provide common services and manage the life-cycle of modules, they hardly interfere with the system built on top of them.

Modular Derby with its code generation distributed should on the average have the same performance figures as the Modular Derby executed in the same JVM. The reason is that the queries are processed only the first time. Later they are retrieved from the cache already processed, hence the remote code generation service is not used anymore and creates no overhead. However, for the first time a query is executed, the overhead resulting from distributed processing should be visible.

The measurements done with the statement cache disabled show the following: Monolithic and Modular Derby executed in a single JVM have the same performance. Modular Derby with code generation shows a decrease in performance by roughly 30%. This can be explained by the overhead incurred from the remote code generation. The absolute measurement figures are shown in chart 9.4. Chart 9.5 shows normalized percentage values. The charts show how many transactions have been completed per second.
Figure 9.4: Derby absolute performance figures

Figure 9.5: Derby normalized performance figures
Chapter 10

Conclusions and future work

In this thesis it has been shown that it was possible to modularize and distribute parts of the Derby RDBMS. In particular, it was possible to modularize Derby as shown in previous chapter on figure 6.1 and to distribute the code generator and the SQL Parser which both are important parts of the SQL query processing in Derby.

The modularization and distribution have been achieved by refactoring Derby’s code, partitioning it into modules and restructuring its functions in a way which allows to run them as remote services. It was only possible to achieve these steps by first knowing the structure of Derby through understanding of its requirements and behavior. The knowledge about its internal structure like dependencies, interface usage protocols and usage of objects data was accrued by targeted analysis partly supported with tools. The tools developed specifically for the purposes of this project have been presented in chapter 4. They have helped to reduce time spent on tedious and uncreative tasks.

Furthermore, the process to achieve modularization and distribution of Derby has been distilled into a more abstract generic process described in more detail in chapter 8. Likewise, the refactorings applied to Derby have been grouped and brought into a more abstract form and finally described as the responsibilities principles lying behind the changes.

10.1 Future Work

The goals of this project are only a few small steps towards improving the scalability and adaptability of Derby and RDBMS in general. Before continuing the work on Modular Derby it will be necessary to analyze how and whether at all it is possible to modularize and distribute indexing and transactions. Answering these questions would substantially contribute to determining how far the vision of modularizing and distributing an existing RDBMS like Derby
can be taken.

Other, not less important topics to analyze, explore and finally realize are:

**Storage layer** Analyze how the storage layer can be restructured, modularized and finally provided as a remote service. Once the analysis is done, do the actual separation and distribution. Separation of the store is a very important prerequisite for the separation of other services like the query tree optimization. Furthermore the separation of the store is also one of the major steps which after being completed, would show how far the overall modularization and distribution of Derby can be brought.

**Optimizer** After the separation of the Storage layer it should be possible to continue the modularization and distribution of the Derby query optimizer. The separation of the Optimizer is very interesting because it is a playground for discovering how to design a system (Derby) which can evolve over time without downtime. During the execution time of the entire Derby engine, the optimizer might be changed and its changed version might replace the old while the system is running.

**Offline Job Processing** Derby uses an offline job processing mechanism for periodical management tasks not directly related to user interaction. Analysis on whether it is possible and desirable to modularize and distribute the offline job processing could be done.

**Remote service loadbalancing** Explore the possibility to use multiple instances of a service type. For instance use multiple instances of the code generator with a loadbalancer used to dispatch the work among them. Pursuing the loadbalancer experiments would provide answers on how to improve the scalability of Derby beyond the point of sheer distributing its functions on separate systems.

**Automatic configuration** Once the functions of Derby are modularized and their services remotely accessible it will be necessary to configure the distributed Derby. The configuration means: where the remote services reside, how many of them are deployed, how they communicate, etc. An automatic configuration would be able to determine based on the environment where and how to deploy the services which together provide the entirety of Derby.
Bibliography


