Modular Database
Eya Ben Charrada

Master’s Thesis in computer science
at the Information and Communication Research Group
April 1st, 2008 - September 13th, 2008

ETHZ Supervisors:
Prof. Gustavo Alonso
Jan S. Rellermeyer

EMSE Supervisor:
Prof. Olivier Boissier
Dedication

This work is dedicated for my beloved parents who offered me unconditional love and support, for my brothers and their special sense of humor, and for my husband who has given me support during the difficult moments and who has been a great source of courage and motivation.

Acknowledgement

I would like to express my thanks to Professor Gustavo Alonso for advising my placement and for giving me the opportunity to work among his group. I would also like to thank Professor Olivier Boissier and Jan Rellermeyer for their guidance, time and advice. I am grateful to Ionut Subasu for the helpful comments on the thesis. To each of the above, I extend my deepest appreciation.
Abstract

Database Management Systems are still consisting of a huge monolithic piece of software that should be taken and used as it is, and which can not be adapted to the user’s needs. As a result, the client is in many cases obliged to pay more to get a database that includes functionalities that he never uses. If DBMS become modular, they will become more extensible and it will be easier to adapt them to the user’s needs by adding and removing components according to requirements.

In this thesis we present our approach to increase DBMS extensibility through modularization. We aim to split a monolithic DBMS into loosely coupled services that can be adapted at runtime. We also include tests to evaluate the effects of modularization on the performance of the database engine.
Abstract

L’architecture des systèmes de gestions de bases de données n’a pas subi de grand changement durant les dernières décennies. En effet, ils ont encore une structure monolithique qui est très peu flexible et qui n’est donc pas bien adaptable aux besoins de l’utilisateur et des application. Pour plusieurs applications, on a juste besoin d’une petite base de données incluant peu de fonctionnalités, mais comme la plupart des bases de données ont été conues pour gérer un grand tas de données ainsi que pour être utilisées par un grand nombre d’utilisateur en même temps, on se trouve souvent obligé de payer pour des fonctionnalités qu’on n’utilise jamais.

Dans ce projet on propose de faire une base de données modulaire qui est basée sur une architecture de service. Le SGBD est constitué de modules indépendants et chaque module offre des services qui sont accessibles via une interface de programmation API bien définie. L’avantage d’une telle architecture c’est qu’on peut l’adapter aux besoins et aux ressources, en choisissant les modules qu’on veut inclure au système de gestion de base de données. Cela permet aussi de distribuer le système facilement en lançant les modules à partir de machines séparées.
# Contents

1 Introduction 5
   1.1 Context and Motivations ................................. 5
   1.2 Goal ................................................. 6

2 Background and Related work 7
   2.1 Concepts and Background ............................... 7
      2.1.1 Modularity ...................................... 7
      2.1.2 OSGi ............................................ 7
      2.1.3 Remote-OSGi .................................... 10
      2.1.4 Apache Derby .................................... 10
   2.2 Related Work .......................................... 14
      2.2.1 Types of Components Database Management Systems ..... 14
      2.2.2 Extensible Database Management Systems ............... 15
      2.2.3 Service Oriented Architecture for Database Management Systems ............................................. 15
      2.2.4 Synthesis ......................................... 16

3 Approach for modules separation 19
   3.1 Modularization according to Functionalities .............. 19
      3.1.1 Modules .......................................... 19
      3.1.2 Results .......................................... 20
   3.2 Modularization according to Database Architecture ........ 21
      3.2.1 Dependencies ..................................... 21
      3.2.2 Derby modules .................................... 22
      3.2.3 Dependencies ..................................... 23
      3.2.4 Monitor/modules problem ............................ 23
      3.2.5 Hidden dependencies ............................... 25

4 Tests 27
   4.1 Transaction Processing Performance Council ................ 27
      4.1.1 Benchmark B ...................................... 27
      4.1.2 Benchmark C ...................................... 28
      4.1.3 BenchmarkSQL .................................... 28
   4.2 Tests ................................................ 29
      4.2.1 Tests with Benchmark B ............................ 29
      4.2.2 Tests with Benchmark C ............................ 30
      4.2.3 Various Queries Test ............................... 30
      4.2.4 Select Queries Test ............................... 32
4.2.5 Tests with multiple clients ........................................ 33
4.2.6 Tests with Remote-OSGi .......................................... 34
4.3 Comments .............................................................. 35

5 Conclusion ............................................................... 37
Chapter 1

Introduction

For decades, Database Management Systems (DBMS) [2] have been consisting of large monolithic piece of software, which are complex and not sufficiently flexible. Although many research projects aimed to design and implement different kind of extensible database engines such as DASDB [23], POSTGRES [26] and Starbust [27], DBMS flexibility is still very limited. Modular systems are supposed to present more extensibility, flexibility and adaptability by allowing to modify and configure the software components according to the needs of the user [1, 3, 4]. In addition, modularity has the advantage of facilitating the distribution of the database. Being able to adapt the system according to the client and applications needs, by adding and removing modules according to requirements can pull down the DBMS cost and increase its performance at the same time. Taking in consideration the need of database flexibility, we intend in our project to make an extensible DBMS by transforming a monolithic database engine into a modular one, where components can be added and removed at run time.

1.1 Context and Motivations

Why a modular database?

Ironically, databases are the last major preserve of monolithic, closed design. A decision to use a particular dbms is also a decision to accept a way of managing disk space, buffers, an access method, a security scheme, a query language, an API, and more. In short, every database, relational, object oriented, or otherwise, is its own self contained world. The first challenge for the decade is to redesign databases around the concept of layered, cooperating components.

David Vaskevitch - Database in Crisis and Transition: A technical agenda for the year 2001 (1994)

As the needs of companies to manage more and more data using efficient tools is growing, the complexity of developing and maintaining databases is increasing and the costs of database management systems is becoming higher. Such monolithic and high-cost databases are not necessarily suitable to small applications that are not meant to be used by many users and that manage small
amount of data. As a result the DBMS are becoming more expensive and the user is in many cases paying for functionalities that he doesn’t use. Designing a new modular database where components can be added and removed according to requirements is a solution for the problem as the functionalities proposed by the database can be configured and adapted to needs.

There are also more and more applications and domains that need to manage specific data type that are not well handled by conventional DBMS this can include multimedia data, documents, temporal data... So it would be interesting to take advantage of the existing databases and configure them according to the specific requirements. This is possible if we use a modular database where we can choose the components to add.

Another other advantage of modularization is that it gives us extensibility and distribution. Extensibility is due to the fact that we can change components at run time, and distribution is because we can chose to run modules in different machines.

1.2 Goal

We propose in this project to make an extensible database engine by splitting a monolithic DBMS into components that can be added and removed according to requirements. We also aim to test the modular DBMS performance and compare it to the monolithic one, this will give us an idea about the effects of modularization on the database. So the first step would be to make a Service-oriented DBMS as defined by Dittrich and Geppert and then transform it into a Configurable DBMS.

We are going to refactor Apache Derby [12], a database system entirely implemented in Java. We will split the database into modules and then transform it into an OSGi [7, 8]-based System, where OSGi is a platform meant for managing software modules.

In this thesis, we first present the state of the art in the field of extensible and modular databases as well as the background needed for our project in section 2. In section 3 we introduce our approach to modularize the database and we explain the main difficulties faced during the database refactoring. Then we present the main tests that were meant to evaluate the database performance and we compare them to the tests applied to the monolithic one, this will be in section 4. Finally, we conclude and discuss the work that need to be done in the future to continue this project in section 5.
Chapter 2

Background and Related work

In this Chapter we are going to present the concepts and technologies needed for the project. Then we will give a presentation of the state of the art in the field of extensible and modular DBMS.

2.1 Concepts and Background

2.1.1 Modularity

A modular application can be defined as follow:

A modular application, in contrast to one monolithic chunk of tightly-coupled code in which every unit may interface directly with any other, is composed of smaller, separated chunks of code that are well isolated. Those chunks can then be developed by separate teams with their own life cycles and their own schedules. The results can then be assembled together by a separate entity: the distributor [5].

A module can be defined as a unit whose structural elements are powerfully connected among themselves and relatively weakly connected to elements in other units [6].

2.1.2 OSGi

The OSGi technology is a platform proposed by the OSGI Alliance meant for managing software modules. It provides a Java framework that supports the deployment of modules which are called bundles in OSGi. The OSGi framework provides an environment allowing to download, install and remove bundles according to requirements. The bundles can also be installed and updated with no need to reboot the machine. With OSGi it is also possible to manage services. Any OSGi bundle can register, find and use services.
OSGi bundles

An OSGi bundle is a JAR file containing class files as well as other resources, and which have specific OSGi attributes defined in the Jar manifest. Each bundle contains an Activator that includes a start and stop methods.

- The start method is used when starting the bundle to register itself and to start any needed thread.
- The stop method is meant to clean up and stop any running threads when stopping the module.

When activated, a bundle receives a Bundle Context that provides methods to:

- Register the services provided by the bundle
- Access information about the rest of the framework
- Install new bundles.

Bundle state

Each bundle has a state. Figure 2.1 presents the different possible states for an OSGi bundle. A bundle can be “INSTALLED”. At that stage it is stored in the persistent storage of the Framework, then if it is ready to start or if it has stopped, his state is “RESOLVED”. The bundle state can also be “STARTING” or “STOPPING”. When a bundle is in the “ACTIVE” state, it means that it started correctly and is running. Finally if the bundle has been uninstalled, which means that it is no more in the persistent storage of the Framework and consequently it could not be started, his state would be “UNINSTALLED”. Every bundle can include OSGi Services, which are defined in the next paragraph.

OSGi Services

An OSGi service [7] is a registered object that can be discovered and used by other modules (Figure 2.2). The service is registered with the “Service Registry” that holds all the services registrations. An OSGi bundle has the ability to register services and to search and discover services that are registered by other bundles. It also receives information whenever a new service is registered or unregistered.

Each service is defined by a service interface that allows other bundles to know about the service’s public methods. Each service implementation is owned by a bundle that registers it so it becomes available to other bundles.

To obtain a service, a bundle specifies the name of the service interface that is implemented by the needed service and uses the Bundle Context to get it. There is also the possibility to use a filter when looking for a service, to narrow the results of the services search.
Figure 2.1: State Diagram Bundle [7]

Figure 2.2: OSGi Framework with Bundles and Services [10]
2.1.3 Remote-OSGi

R-OSGi [9] is a part of the flowOSGi project, which is an ongoing project at the Department of Computer Science at the ETH Zurich. R-OSGi, which is an acronym for Remote-OSGi, is an extension of the OSGi Framework that enables the distribution of an OSGi-based application (Figure 2.3). Differently from OSGi that runs on one machine, using R-OSGi we can run bundles in separated machines. R-OSGi is characterized by small footprint and it is adapted for small and embedded devices with limited memory.

R-OSGi [10] runs as a service on the OSGi framework and it gives the possibility to register remote services by adding some properties during service registration. One of the important characteristics of R-OSGi is transparency. In fact, for an OSGi client bundle, the remote services are not distinguished from the local ones.

2.1.4 Apache Derby

Derby [11] [12] is a pure Java relational database that was developed by IBM under the name of Cloudscape. Since August 2004, Derby become an open source project under the Apache 2.0 license.

The database started as a project at the Cloudscape Inc in 1996, then it was acquired by Informix in 1999 and then by IBM in 2001. In August 2004, IBM contributed the code of the database to the Apache Software Foundation and the database was renamed Derby. In April 2008, Apache released the 10.4.1 version of Derby.

Derby is characterized by its small footprint, 2MB, and its embeddable architecture. It implements the Java Database Connectivity (JDBC) [19] API.
standard, the SQL92E [20] standard as well as many other SQL99 [21] extensions. The Derby engine supports multiple connections and multiple threads to use a connection and it works within a single system or over a network.

Here are some technical aspects that differentiate Derby engine from other database systems [11]:

- Derby is easy to administer. When embedded in a client application, the system requires no administration at all.
- Derby is embeddable. Applications can embed the Database Management System (DBMS) engine in the application process, eliminating the need to manage a separate database process or service.
- Derby can run as a separate process, using the Network Server framework or a server framework of your choice.
- Derby is a pure Java class library. This is important to Java developers who are trying to maintain the advantages of Java technology, such as platform independence, ease of configuration, and ease of installation.
- Derby needs no proprietary Java Virtual Machine (JVM). Written entirely in the Java language, it runs with any certified Java 2 Platform, Standard Edition (J2SE) JVM at a release of 1.3 or higher. Beginning with Derby 10.1, there is new Java 2 Platform, Micro Edition (J2ME) support for JSR 169, the JDBC API defined for the Connected Device Configuration (CDC) / Foundation Profile (FP).
- The Derby DBMS engine is lightweight. It is about 2MB of class files, and it uses as little as 4MB of Java heap.
- Derby provides the ability to write stored procedures and functions in Java. Derby does not have a proprietary stored procedure language; it uses JDBC.

In the next paragraphs we present the Derby internal architecture and its layers, and the different run modes for the database.

**Derby Architecture**

The Derby engine [12] comprises a monitor and modules. Each module presents some specific functionalities such as lock management or indexing. The role of the monitor consists on mapping the modules requests. It provides methods to start, find and get modules when they are needed.

**Derby Layers**

Derby is composed of four layers [13] which are the JDBC, the SQL, the Store and the Services (Figure 2.4). Each layer has defined functionalities.

1. **JDBC**
   The JDBC layer contains an implementation of the java.sql and javax.sql for JDBC2.0 and 3.0.
2. SQL
The SQL layer is responsible for the compilation and execution of queries. The compilation includes parsing, using a parser generated by JavaCC, binding, optimizing, generating the statement plan and finally loading the generated class (which represents the statement plan) and creating an instance of it. The execution consists of executing methods of the instance of the generated statement plan.

3. Store
The store is composed of two parts:

- The access, which handles table scans, index scans, index lookups, indexing, sorting, locking policies, transactions, isolation levels...
- The raw which is responsible for the storage of rows in files, transaction logging, transaction management...

4. Services
The service layer includes methods meant to provides different needed functionalities, such as lock management, cache management, error logging...

Embedded and Client/Server modes

The Derby database can be embedded into an application (Figure 2.5) which means that it runs as a part of the application. In this case, both the database and the application run in the same Java Virtual Machine. Even in the embedded mode, the database supports multiple connections from multiple application threads.

Derby can also run in a client server mode, as shown in Figure 2.6.
Figure 2.5: Derby embedded mode [11]

Figure 2.6: Client/Server Mode [11]
2.2 Related Work

Many research projects are aiming to transform DBMS architecture from monolithic structures to component and modularized ones, as shown in Figure 2.7. The disadvantage of monolithic structures is that they have to be taken as they are, and cannot be adapted to the client or application needs. This is why many research projects were carried on to make databases more flexible. In this section we first present different types of Components Database Management Systems, which are supposed to be more flexible than the monolithic ones. Then we present some of the projects that were meant to implement extensible DBMS using different approaches. The section that follows introduces another approach which relies on Service-Oriented architecture. We will also describe the similarity and difference between these projects and the modular DBMS we want to implement. Finally, we will present the different technologies and concepts that we use in our project.

2.2.1 Types of Components Database Management Systems

According to [1] there are 4 types of components database management systems:

1. Pluggable DBMS:
   The core of this type of Component DBMS includes all standard functionalities, while all non-standard features can be plugged to extend it.

2. Middleware Database:
   This kind of DBMS integrates existing data stores but leaves data under the control of their original management system.

3. Service-oriented architecture for DBMS:
   Here the DBMS is transformed into a set of standalone services.

4. Configurable DBMS:
   The configurable DBMS extends the service-oriented ones by offering the possibility to adapt the services implementations to new requirements.
2.2.2 Extensible Database Management Systems

Many research projects were carried to increase database extensibility. Among these projects we cite DASDBS [23], POSTGRES [26] and Starburst [27].

DASDBS project started in 1983. The idea of the project was to implement a system composed of a kernel, which is meant to be the lowest denominator of the data management functions needed by a large variety of applications classes and a set of application specific front ends that run on top of the common kernel system. The database was designed as a family of database systems that are tailored to individual application classes as shown in figure 2.8.

POSTGRES [28] is an extensible database engine that allows to add new datatypes for polygons, bitmaps and texts. This allows to create custom datatypes and represent complex datastructures [25]. According to the PPSTGRE website [31] “PostgreSQL server can incorporate user-written code into itself through dynamic loading. That is, the user can specify an object code file (e.g., a shared library) that implements a new type or function, and PostgreSQL will load it as required. Code written in SQL is even more trivial to add to the server. This ability to modify its operation “on the fly” makes PostgreSQL uniquely suited for rapid prototyping of new applications and storage structures”.

The Starburst project, which was carried on between 1984 and 1992 at IBM Almaden, has as goal to build a complete, operational prototype of an extensible relational database management system [32]. Starburst provides a skeleton architecture that allows extensions in several but predetermined ways [28].

2.2.3 Service Oriented Architecture for Database Management Systems

More recent projects were meant to increase database flexibility by turning it into a service oriented system. Here we introduce the Service-Based Data Management System (SBDMS) [28] and the Component Based Runtime Adaptable
Database (CoBRA-DB) [29], which present many point of similarity with our project.

**Service Based data management systems**

A different approach have been proposed by [28] to increase database extensibility. They propose to found the database architecture on the principles of Service Oriented Architecture (SOA) which they call Service-Based Data Management System (SBDMS). In this approach, the database consists of a set of loosely coupled services. Each Service is accessed only by means of a defined interface and without requiring any knowledge on the implementation, in this case when a new service is added, it doesn’t have impact on the rest of the database system. the services are independent, so they can be either distributed over computers or on a single machine.

In contrast to other extensible DBMS, where extensions are usually done in a predetermined way and added at the topmost level of the DBMS architecture, in the SOA approach, the system can be extended at every point in the architecture by creating new services that use existing services. For example [28], it would be possible to create a new datatype as well as a special service responsible for storing this new datatype, then this service can be used by other services of the system. The other advantage of having a service based DBMS is that "users can choose their own services based on their needs and available resources". Consequently, the user do not need to pay for functionalities that he doesn’t need anymore. The proposed architecture for the SDBMS consists of four service layers (Storage Services, Access Services, Data Services, Extension Services)

these layers have been chosen according to the layered DBMS architecture proposed by Harder and Reuter [30].

**Component Based Runtime Adaptable Database**

Not far from the SBDMS, a group from the Friedrich-Alexander University of Erlangen-Nuremberg(Germany), is working on a Component Based Runtime Adaptable Database (CoBRA-DB), which consists on a "Modular DBMS that can be adapted to different environments by assembling prefabricated modules" [29]. The goal of this project is to modularize a DBMS and be able to exchange modules at run time. Like in the SDBMS architecture, the CoBRA-DB implements some of the Harder’s layers. The difference between the two approach is that the CoBRA-DB addresses the handling of cross cutting concerns, and provides run time adaptation of the DBMS.

**2.2.4 Synthesis**

The goal of our project is to design and implement a modular DBMS, where components can be added and removed according to requirements. Modularization will not only increase DBMS extensibility and adaptability but it will also facilitate its distribution.
Similarly to the last two projects (SBDMS and CoBRA-DB), we aim to turn the database engine into a service oriented DBMS, which consists of loosely-coupled modules providing services. These services are accessed using a defined interface, so they do not know about the implementation of each other. Such a system allows users not only to adapt the DBMS to requirements by choosing the modules to be added to the database, but also gives the possibility to change the database modules at run time.

In contradiction to the SDBMS and CoBRA-DB approaches, where the choice of modules reposes on hard’s layers, we have chosen to separate modules according to functionalities, such as Compilation (parsing, optimizing...), storage, indexing... This gives the possibility to duplicate processes in order to ameliorate the database performance and adapt it to existing resources. Also, we propose to run the modules using OSGi, which is a platform to manage modular systems. Once the database is transformed into an OSGi-based system, we can distribute using Remote-OSGi, which is an extension of the OSGi platform, allowing to distribute modular systems.
Chapter 3

Approach for modules separation

The goal of our project is to increase extensibility of the database engine. We have chosen to separate modules that the user may change or distribute according to his requirements and resources. In the next part we present our two approaches to separate modules. We explain the main decisions made for each of these approach, as well as the problems and results of the implementation.

3.1 Modularization according to Functionalities

In our first approach, we have chosen to separate modules according to functionalities. Some of these functionalities were selected because they need much resources to be executed, this is the case of the indexing and the storage. We selected the four following functions to transform into modules: Query Language(compiler), Indexing, Storage and Metadata management. Each of the chosen modules is supposed to add more flexibility and extensibility to the database. In this section we justify the choice of the modules and the results of the separation.

3.1.1 Modules

We propose here a description for each module, and a justification for why have we chosen to separate it.

Query Language Module

This Query Language Module is in the charge of transforming an SQL query (string) into an Intermediate Representation, which is in our case a tree of nodes. Having such a module, allows the user to change the query language used with the database according to his background. By separating the query language module we also have the possibility to use different query languages with the same database, by running different query language services at the same time, and allowing the user to choose the language he prefers.
Index Module

All the indexing operations (creating, deleting, updating, sorting and scanning index) are done by the Index Module. Separating the index, increases the database flexibility as it gives the possibility to choose and change the indexing method according to the application and the user requirements. For example the user can choose to change a B-Tree indexing by R-Tree indexing [33]. As the indexing can rapidly become too large, it is also interesting to separate it in a way that gives us the possibility to distribute it into separate machines.

Storage Module

The storage module is responsible for the storage and the scan of the database data. If the stored data is too large, it is interesting to distribute it into different machines. This means that we will divide the database into small databases, each of them containing a part of the tables.

Metadata Module

If we have different storage systems, it is may be relevant to have a module that contains metadata copies of the different systems. Using this module we can know which database contains the tables we are looking for.

3.1.2 Results

We separated the first module (query language module) from the rest of the database. The module was in charge of doing the parsing of queries. When we tested the separated module using OSGi, it seemed to be working correctly, but when we run it with R-OSGi, the system crashed.

The problem was related to a class called CompilerContext in the derby source code. The CompilerContext is a structure that contains all the necessary information used during compilation, so it is heavy. It is used by the parser and by the rest of the database. When we start the database, the CompilerContext is sent from the core to the parser. As this structure is not serializable, we cannot send it through the network and this is why the modules don’t work using R-OSGi. Thanks to these tests we discovered that it is not relevant to modularize the database if the CompilerContext still exist. Because if we send it from the core to the parser it is just like sending almost all the core to the parsing module.

When trying to separate the index from the rest of the database, we realized that separating the index is quite complicated as there are too many dependencies between the index and the rest of the database.

After working on this first decomposition of Derby, we noticed that our choice of modules was not adapted to the Engine internal architecture. Consequently, separating those modules can take much longer time than we expected to modularize the DBMS. So we decided to rethink what modules should be separated and how can we do the modularization in a better and more efficient way. The new approach is presented in the next part.
3.2 Modularization according to Database Architecture

The Derby source code is divided into packages, each package comprises all the classes needed for a specific functionality. Among the different packages of Derby we have the JDBC, Authentication, Store, SQL, Shared, Error, Services, Type, Util... In contrast to our first approach, this time we decided to take the Derby architecture into consideration and to separate modules according to the packages structure. To do so, we moved some packages from the Derby project to new separated projects. These new projects will be considered as our new modules. The shifted packages are the following: (each package is a module)

- SQL
- JDBC
- Store
- Encryption

The rest of the database represents an other module that we call the core.

Each of the packages is responsible for providing some specific functionalities that will be presented later. The packages are not completely independent, in fact there are many dependencies (shared classes and methods) between the separated packages as well as between the packages and the core.

3.2.1 Dependencies

The different Derby packages are not totally independent. In fact there are many classes and methods that are shared between packages and layers. As our goal is to split the database into independent modules so that they can be added and removed, we need to eliminate all existing dependencies between them. At the end each module should be used through its API, and the rest of the system need not know about modules implementation.

To eliminate the dependencies between the different modules, various refactorings was performed. Some of them consist on creating new interfaces for the classes that are used by many modules. In this case, the modules would not know about the implementation any more and they will know about the api instead, which removes the dependencies between the modules.

When a module needs to use methods that are included to classes from other modules, we replace the methods by OSGi services that are registered when launching the database and that can be used by other modules.

We also faced other problems that are particularly related to the Derby architecture, like the problem related to the Derby’s monitor and modules as well as the problem related to the Derby LanguageConnectionContext (LCC). These problems will be presented in more details later in the thesis.
3.2.2 Derby modules

After applying our new approach, we have got 5 modules, which are: Core, Cstore, SQL, JDBC and Encryption. The choice of the modules was done according to the packages architecture of the current Derby Database Engine. Here is a description of each of the separated modules.

Store

This module contains all the packages related to the storage function. The Store layer is split into two main areas, access and raw. The access presents a conglomerate (table or index)/row based interface to the SQL layer. It handles table scans, index scans, index lookups, indexing, sorting, locking policies, transactions and isolation levels. The access sits on top of the raw store which provides the raw storage of rows in pages in files, transaction logging, transaction management (from the Apache Derby site). The store module also contains the replication package. As we mentioned in our first approach, separating the storage gives the possibility to distribute it into different machines.

SQL

The SQL module is meant to do compilation and execution. Compilation consists of parsing, binding, optimizing, generating the statement plan, loading the class and creating an instance representing the state of the query. The execution is done by calling execute methods on the instance of the generated class. Separating the SQL module gives the possibility to duplicate it which may ameliorate the database performance.

JDBC

Consists of implementations of the java.sql and javax.sql classes for Java Database Connectivity (JDBC) 2.0 and 3.0. As many applications uses Open Database Connectivity (ODBC) instead of the JDBC, it would be interesting to have the possibility to chose the database connectivity of the DBMS.

Encryption

This module is used for encryption/decryption. As there are different encryption methods, such as Advanced Encryption Standard (AES), Data Encryption Standard (DES), Pretty Good Privacy (PGP), etc. separating this module allow the user to choose the method that is more adapted to his application and needs. This also gives the possibility to change and update the encryption program at run time.
Core

The core is the main part of the database. It contains the API of the different modules (SQL, store, JDBC, and encryption) as well as many derby services (lock management, cache management, error logging...) and some shared packages (authentication, catalog, security, reference, error...).

3.2.3 Dependencies

As we have mentioned before, each module should be independant and need not to know about other modules implementation. Modules are used through their API which is included to the Core. So every module knows about the API packages of the Core (Figure 3.1). In addition to the API, some of the Core packages that are accessible by other modules include shared classes and methods like errors, authentication, catalog...

3.2.4 Monitor/modules problem

Separating some modules lead to problems related to the monitor/modules architecture of the database. Before introducing the problem faced, here is a brief presentation of Derby’s monitor/module, from the Apache Derby website.

Derby architecture

The Derby database engine is a system comprised of a monitor and a collection of modules.
A module is a set of discrete functionality, such as a lock manager, JDBC driver, indexing method etc. A module’s interface is typically defined by a set of Java interfaces, e.g. the java.sql interfaces define an interface for a JDBC driver. All callers of a module do so purely through its interface to separate API from implementation. A module’s implementation is a set of classes that implement the required behavior and interfaces. Thus a module implementation can change or be replaced with a different implementation without affecting the callers’ code.

Modules are either system wide (shared), e.g. error logging, or per-service with a service corresponding to a database, e.g. a lock manager would be a module in a service (database)

This architecture allows different modules to be loaded depending on the environment.

The monitor maps the module requests, to implementations based upon the request and the environment. E.g. with JDK 1.3 the internal request for a JDBC driver the monitor selects Derby’s JDBC 2.0 implementation, while in JDK 1.4 the driver is the JDBC 3.0 implementation. This allows Derby to present a single JDBC driver to the application regardless of JDK and internally the correct driver is loaded.

Problem

Here we explain the problem we face because of the monitor. We will call the modules used by Derby ”intern modules” so we can differentiate them from the modules we have separated. When we launch the database, the ”Monitor” looks for the different available “intern Modules” and starts the ones that are needed for the database launching. Then, when an ”intern module” is needed, the monitor searches the class representing that module and starts it. The Monitor is a part of the core, so it doesn’t have access to the other modules, and this is why it cannot start any ”intern module” that is part of the SQL, Encryption, JDBC or Estore modules.

Proposed solutions

There are different solutions for this problem. We can try to get rid of the monitor, or try to register each intern module as an OSGi service. But these solutions require an extensive time for understanding and implement, therefore they have not been implemented at this stage. But we propose the following solution which is easier to implement and which partially solves the problem.

Implemented solution

As the monitor needs to use some classes from other bundles to launch its internal modules, we can register a service that we call RemoteClassGetterService and which looks for the classes in the remote modules and return them. The services get a class/interface name as input and returns the right class as output. This solution seems to be working fine, as the database starts and runs correctly after refactorings.
3.2.5 Hidden dependencies

Among the advantages of modularization there is the possibility to change a module at run time. We tried to test this with the modular Derby. To do so, we have made copies of some modules that we use during the tests. For example we used two SQL bundles during a test. The test first runs using the first SQL module that we call sql1 and then we stop sql1, start sql2 and restart the test.

Problem

At the beginning such a test did not work, because the JDBC bundle has a reference to the LanguageConnectionContext (LCC) which is part of the SQL bundle. So even when sql1 is stopped and sql2 is started, the JDBC still has a reference to sql1. When we restart the test we get a NullPointerException. Such a result shows that even after separating the modules, removing all the direct dependencies between them and resolving all the compilation problems, there are still some hidden dependencies which we cannot discover until running this kind of test. In the next part we propose a method to eliminate the LCC problem.

Proposed solution

The LCC problem will possibly be solved if we remove the reference that the JDBC bundle has to the LCC. The solution was implemented as follows: When starting a new connection to a database, a new LCC is created. Instead of returning a reference of the LCC to the JDBC bundle, we associate an id to the LCC and return the id. At the same time, the LCC is added to a vector contained by the SQL module. A new service was also created to allow the JDBC bundle to use the LCC. The service is called LccService and it is registered by the SQL bundle.

With this configuration, whenever the JDBC module needs to use a method from the LCC, it will look for the LccService and ask it to run it. The JDBC will give as inputs the need LCC id, and the LccService will look for that LCC in the vector containing all the LCC, then it will run the method and return the result to the JDBC.

Results

After implementing this method, when we tested the modular database with one client, it seemed to be working correctly when replacing the SQL module by another one and refreshing the modules. But when we run it with multiple clients, many queries failed to complete.

The problem here is possibly related to caching. In fact the LCC is cached and can be used again instead of creating a new one. When the database was refactored, instead of looking for cached LCC, a new LCC is created for each connection. The consequence is that too many LCC are created and none of them is reused, which makes the database a lot slower when we have many clients and this can be the source of the problem.
Chapter 4

Tests

Different tests were done to compare the modular Derby to the monolithic one. These tests give us an idea about the performance changes due to refactorings and modularization. To do the tests, we used benchmarks of the Transaction Processing Performance Council [14] (TPC) as well as some other tests that we have made specially to get divers information about the database performance. Before introducing the results, here is a description of TPC and the used benchmarks.

4.1 Transaction Processing Performance Council

TPC is a non-profit corporation that defines transaction processing and database benchmarks. These benchmarks are meant to evaluate the performance of different systems. As in the business world, the TPC consider a transaction as a commercial exchange of goods, services or money. The TPC simulates an environment similar to the real business world that can include transactions related to inventory control, airline reservations or banking. To measure Transaction processing and database performance, TPC benchmarks calculate the number of performed transactions per unit of time. The results are in tpm (transactions per minute) or tps (transactions per second). Among the TPC members, there are many computer and information Technology corporations such as IBM, NEC, hp, Sun, intel...

4.1.1 Benchmark B

TPC-B is the second benchmark approved by the TPC. it measures how many transactions per second a system can perform. In the TPC-B, Transactions are submitted by many programs executing concurrently. After submitting a transaction, a program waits for the transaction end to submit another one. In the TPC website it is mentioned that the TPC-B benchmark is characterized by:

- Significant disk input/output
- Moderate system and application execution time
• Transaction integrity


When running a TPC-B benchmark we can chose the number of clients as well as the number of transactions per client in the test.

4.1.2 Benchmark C

The TPC-C benchmark is an on-line transaction processing (OLTP) that was approved in July 1992. It includes five different types of transactions with different complexity.

"TPC-C simulates a complete computing environment where a population of users executes transactions against a database. The benchmark is centered around the principal activities (transactions) of an order-entry environment. These transactions include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. While the benchmark portrays the activity of a wholesale supplier, TPC-C is not limited to the activity of any particular business segment, but, rather represents any industry that must manage, sell, or distribute a product or service." (From the TPC official website)

According to the TPC Benchmark C standard specification, Revision 5.9, the benchmark is characterized by :

• The simultaneous execution of multiple transaction types that span a breadth of complexity
• On-line and deferred transaction execution modes
• Multiple on-line terminal sessions
• Moderate system and application execution time
• Significant disk input/output
• Transaction integrity (ACID properties)
• Non-uniform distribution of data access through primary and secondary keys
• Databases consisting of many tables with a wide variety of sizes, attributes, and relationships
• Contention on data access and update

4.1.3 BenchmarkSQL

The presented TPC-C tests of this document are done with BenchmarkSQL [15] [16], which is an open source implementation of the TPC-C benchmark. It is based on the JTPCC [17] benchmark which is an open source Java implementation of the TPC-C. The primary difference [18] between JTPCC and BenchmarkSQL is that BenchmarkSQL has been rewritten to use all JDBC prepared statements rather than dynamic SQL.
4.2 Tests

4.2.1 Tests with Benchmark B

We run the TPC-B benchmark with different configurations.

**TPC-B test 1**

Number of clients: 5 - 10 and 50

Number of transaction per client: 10 Tx/ client

Each of the tests was run four times at least, and the values presented here are the average of the tests result values.

![Figure 4.1: TPC-B : 10tx/Client](image)

**Comments** For this test, the number of executed transactions per second is slightly higher for the modular database.

**TPC-B test 2**

Number of clients: 5 - 10 and 50

Number of transaction per client: 100 tx/ client

Each of these tests was run three times at least.

**Comments** The number of executed tx/s is almost the same for the modular Derby and the monolithic one.
4.2.2 Tests with Benchmark C

For the TPC-C benchmark, each of the tests was run for a duration of 10 minutes at least.

Number of terminals: 1 - 5 and 10

The used database contains around 600,000 rows

Comments In average, we have the same number of executed transaction per minute for both databases.

4.2.3 Various Queries Test

In this test, we launch a program that executes different types of queries against a database, and we have at the end the number of the executed queries for each type. Each of the tests was run at least twice for a duration of 2 minutes. The used database and queries for these tests are those of the TPC-C benchmark. The database contains tables with very different number of stored data. Here we have the number of rows in each of the used tables:

- Customer: 30,000
- District: 10
- Item: 100,000
- Order: 30,000
The executed queries are:

- Join select (where clause)

\[
\text{SELECT O\_ID FROM oorder, customer, district WHERE o\_id = c\_id AND c\_w\_id = 1 AND c\_d\_id = ? AND c\_w\_id = d\_w\_id AND c\_d\_id = d\_id}
\]

- Join select (join clause)

\[
\text{SELECT O\_ID FROM oorder JOIN customer ON o\_c\_id = c\_id AND c\_w\_id = 1 AND c\_d\_id = ? JOIN district ON c\_w\_id = d\_w\_id AND c\_d\_id = d\_id}
\]

- Update item (update 1)

\[
\text{UPDATE item SET i\_data=? where i\_id=?}
\]

- Update stock (update 2)

\[
\text{UPDATE stock SET s\_quantity=? where s\_i\_id=?}
\]

- Insert item

\[
\text{INSERT INTO item (i\_id, i\_name, i\_price, i\_data, i\_im\_id) VALUES (?, ?, ?, ?, ?)}
\]
Comments The graphic shows that the time for executing the different kind of queries is almost the same for the two databases.

4.2.4 Select Queries Test

Here we try to measure the time for executing a select query with a join between two, three or four tables. The used database and the queries for these tests are those of the TPC-C benchmark. The database content has been described in the previous section.

The duration of the test was 45 minutes, and the used queries are the following:

- Two tables join
  - where clause:
    
    "SELECT d_name FROM warehouse, district WHERE w_id = d.w_id AND d_id = ?"

  - join clause:
    
    "SELECT d_name FROM warehouse JOIN district ON w_id = d.w_id AND d_id = ?"

- Three tables join
- where clause :

"SELECT O_ID FROM oorder, customer, district WHERE o_c_id = c_id AND c_w_id = 1 AND c_d_id = ? AND c_w_id = d_w_id AND c_d_id = d_id"

- join clause :

"SELECT O_ID FROM oorder JOIN customer ON o_c_id = c_id AND c_w_id = 1 AND c_d_id = ? JOIN district ON c_w_id = d_w_id AND c_d_id = d_d_id"

• Four tables join

- where clause :

"SELECT O_ID FROM oorder, customer, district, warehouse WHERE o_c_id = c_id AND c_w_id = 1 AND c_d_id = 1 AND c_w_id = d_w_id AND c_d_id = ?"

- join clause :

"SELECT O_ID FROM oorder JOIN customer ON o_c_id = c_id AND c_w_id = 1 JOIN district ON c_w_id = d_w_id AND c_d_id = ? JOIN warehouse ON w_id = d_w_id"

During the test, the previous queries were executed many times. At the end we have the number of executed queries during 45 minutes as well as the time spent to execute each type of query. Each kind of query (2 tables join, 2 tables where join, 3 tables...) was executed more than 3000 time. In the following schema, we have the number of executed queries per second. As the values for the 2 tables join are too much greater than the 3 and 4 tables join, we have represented them in separated schemas.

Comments  According to the graphs, the modular Derby seems to be slightly faster than the monolithic one in executing queries with joins. Except for the 2 tables join, the difference between the number of executed queries per second for the two databases does not exceed 3 or 4%.

4.2.5 Tests with multiple clients

Here we try to run tests with different number of clients. For this test, we use the TPC-C database and the queries of the "Queries test 2", which include 6 types of select queries with joins. we run the test with 1 - 3 - 5 and 7 clients.

Each of the tests was done at least twice for duration of 1 minute. so the results represented here are the number of queries executed during 1 minute.

Comments  The results of the multiple clients test are very similar for both databases.
4.2.6 Tests with Remote-OSGi

Lunching the modular derby using R-OSGi produces an exception and thus the database cannot start. The problem seems to be related to the Derby’s monitor, because the thrown exception shows that the monitor could not get some needed classes from the remote modules.
4.3 Comments

According to the different tests, the modular Derby seems to be, in most cases, slightly more efficient and faster than the original Derby engine. But these tests are far from being enough to conclude that the modularization improved the database performance. In fact, the results of the tests are not accurate enough to make such a conclusion. In the previous graphs we presented the average of the different results we have got. But there is still something important that we noticed during the tests and that is not visible in the graphs. Each time we run a test we get new values that are more or less different from the previous ones. Especially after rebooting the machine to re-run the tests, some tests give results that are noticeably different, which means that these results are not accurate and can not be sufficient to do the comparison.

The problem can be related to the fact that the machine run many processes at the same time, which can affect the results we get. During the tests, we stopped all the running applications and we only launched Eclipse which we use to start the tests, but there are still many other processes that are started by the system and that can affect the results. To get more precise and reliable results, we should use a machine that is dedicated for the tests and which only runs the very essential processes that are needed by the machine to work.
Chapter 5

Conclusion

The goal of our project was to make a modular database. Modularization presents various advantages, among them there is the possibility to turn the system more adaptable to the client’s need and more extensible by allowing to add, remove and replace modules at runtime. The other advantage of modularity is that we can transform a modular system into a distributed one if we run the modules on different machines.

To implement our project, we have chosen to use an open source, pure Java database, the Apache Derby database. We refactored the database engine and we split it into five OSGi modules. The modules where chosen according to the Derby packages architecture. The separated modules are: SQL (compilation + execution), Store, JDBC, Encryption and Core (the rest of the database). The modules where tested using some TPC benchmarks as well as with other tests that were conceived to have more information about the database performance. The tests showed that the modular Derby performance is slightly higher than the monolithic one, but as the tests results varied noticeably each time, we concluded that they are not too accurate and that they were probably influenced by other processes running in the machine. Therefore the carried tests are not sufficient to conclude that the modular database is better than the original one. Some other tests showed that there are still some dependencies between the different modules. The existing dependencies are probably due to the Derby monitor that has access to classes belonging to different modules. Because of these dependencies, we could not run the modular database using R-OSGi, nor change modules at run time.

Future work

At this stage we managed to separate 4 modules from the Derby database, which are the SQL module that is meant to do the compilation and the execution of queries, the Storage Module that includes all the functionalities related to Storage, the JDBC Module and the Encryption Module. The rest of the database constitute another module that we call Core, and all of the SQL, Store, JDBC and Encryption modules depend on the Core.
The modules seem to be working correctly with OSGi but not with R-OSGi. The problem is probably related to some dependencies that are still existing between the modules. So the first thing that needs to be done, is to remove these dependencies so that the modules become completely independent. As the dependencies are due to the Derby’s monitor which has access to all the classes of the different modules, and which is in many cases used by modules to get a reference to an object from other modules, we need to get rid of this monitor. If we succeed to remove the monitor, the dependencies between the modules will be removed as well.

Then it is important to test the performance of the modular Derby and compare it to the original one. The tests need to be accurate and reliable, and should not be influenced by the processes running in the machine during the test. So they need to be run on a dedicated machine that only starts the processes needed to launch the system and the tests.

The following step is to test the possibility of duplicating some modules. If the duplication succeeds, it is then important to find if there is a gain in performance that is due to the duplication and measure it.

Among the goals of our project, we want to have a system that can be improved without stopping or rebooting it. To do so, we need to try changing some modules at run time and see if the database still work correctly.

Finally, if the modular database works correctly and if its performance is acceptable, we can go further and separate new modules. Among the modules that can be separated, we propose to split the SQL module into two modules, one for the compilation and one for the execution. Then it can also be relevant to split the compiler into many smaller modules, where each one of them is responsible for a finer functionality like parsing, optimizing, binding...
Bibliography


[18] Postgresql: http://www.postgresql.org/about/news.443


Appendices
Test : Time for parsing, binding and optimizing

At the beginning of the project, one of the goals was to separate some components from the compiler (parser, optimizer...), so it was interesting to have an idea about the time for parsing, binding, optimizing and generating classes. This can help us deciding whether it is interesting or not to separate these processes and include them to separated modules.

For the moment, in our modular derby, we do not have these processes separated, as all the compilation and execution procedures are still in the SQL module, but we still want to compare the time token by each of these processes and compare it between the monolithic and the original Derby.

After running some tests with different number of clients, we noticed that the results are not comparable as they are, because the recorded time for each process varies too much each time. Here we have the min/max values obtained during the tests. These results can only give us an idea about the duration of parsing, binding but they are not very adapted to do a comparison between the monolithic and the modular Derby because the values change considerably at each test. The durations are in milliseconds.

**Monolithic derby**

- Number of clients = 1
  
  parsing: 1 - 5  
  binding: 1 -7  
  optimizing: 1 - 70  
  generating classes: 1 -15

- Number of clients = 10
  
  parsing: 1 - 10  
  binding: 5 -25  
  optimizing: 10 - 450  
  generating classes: 5 -15

**Modular derby**

- Number of clients= 1
  
  parsing: 1 - 5  
  binding: 1 -10  
  optimizing: 1 - 70  
  generating classes: 1 -30

- Number of clients = 10
  
  parsing: 1 - 40  
  binding: 1 -100  
  optimizing: 20 - 400  
  generating classes: 10 -200