Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Master Thesis No. 3506

Extending an Open Source Enterprise Service Bus for SQL Statement Transformation to Enable Cloud Data Access

Simin Xia

Course of Study: Communication Engineering and Media Technology (INFOTECH)

Examiner: Prof. Dr. Frank Leymann
Supervisor: Steve Strauch
Commenced: May 29, 2013
Completed: November 26, 2013

CR-Classification: D.2.8, D.3.3, H.2.3, H.2.4

Abstract

Cloud computing has gained tremendous popularity in the past decade in the IT industry for its resource-sharing and cost-reducing nature. To move existing applications to the Cloud, they can be redesigned to fit into the Cloud paradigm, or migrate its existing components partially or totally to the Cloud. In application design, a three-tier architecture is often used, consisting of a presentation layer, a business logic layer, and a data layer. The presentation layer describes the interaction between application and user; the business layer provides the business logic; and the data layer deals with data storage. The data layer is further divided into the Data Access Layer which abstracts the data access functionality, and the Database Layer for data persistence and data manipulation.

In various occasions, corporations decide to move the their application's database layer to the Cloud, due to the high resource consumption and maintenance cost. However, currently there is little support and guidance on how to enable appropriate data access to the Cloud. Moreover, the diversity and heterogeneity of database systems increase the difficulty of adaption for the existing presentation layer and business layer with the migrated database layer. In this thesis, we focus on the heterogeneity of SQL language across different database systems. We extend an existing open source Enterprise Service Bus with Cloud data access capability for the transformation of SQL statements used in the presentation and business layer, to the SQL dialect used in the Cloud database system backend. With the prototype we develop, we validate it against real world scenario with Cloud services, such as FlexiScale and Amazon RDS. Besides, we analyze the complexity of the algorithm we realized for parsing and transforming the SQL statements and prove the complexity through performance measurements.

Contents

List of Figures

List of Tables

List of Listings

1. Introduction

Cloud computing is the buzz word circulating around for the past decade. Even though it doesn't conform a technical definition, it has gained in the IT industry a large amount of attention for the marketing idea behind it. Providing and consuming computing capacity as daily utility like water or electricity, such concept of converged infrastructure and shared resources has been proved to be economically efficient. The computing capacity we just referred, doesn't restrict to the processing power, but extends to almost every component of a computing system. One particular component we will be focusing on is *database*. "*Database as a Service*" paradigm challenges the traditional model of data management with its intriguing Cloud features, such as seamless software upgrade, automatic data backup and restore, flexible scaling of resources, etc.

However, when switching to the Cloud environment, certain efforts have to be made to ensure the adaption of the existing applications. Minimize the modification of existing applications is the way to achieve a seamless transition. Our solution is to introduce a middle layer between the Cloud and the applications. Its functionality is to eliminate all incompatibilities of the applications in a Cloud environment. And in this thesis, we tackle one specific incompatibility, the query language between database client and Cloud database server.

1.1. Problem Statement

There are various choices of Cloud database services in the market, and different services are often backed with different flavors of database system. In order to take advantage of the Cloud and choose the best fitting alternative, Bachmann proposed a method to migrate database into the Cloud [Bac12], in which he specified specific questionnaires for the decision making of choosing the proper Cloud database based on the user's operational and economical requirements. Due to the diverse requirements, the existing database currently in use, referred as source database, may not coincide with the Cloud database been chosen, referred as target database. This will introduce compatibility issues between source and target databases, which Bachmann addressed before and during the migration process, but not after the migration with data access and data modification.

Furthermore, in order to minimize the impact made on the applications during the migration to the Cloud, Goméz Sáez, in the work *Extending an Open Source ESB for Cloud Data Access Support* [Sáe13a], provided us a prototype ESB with Cloud data access support, which acts as an intermediate layer between the application and the on and off-premise databases. It was

extended with multi-tenancy support, and importantly solved the communication inconsistency problem with different Cloud services. Moreover, it maintained the transparency that the Data Layer provided to the upper layers of the application architecture.

However, the compatibility issue still remains unsolved, due to the differentiation of the SQL language used in various relational database systems. To further ensure the transparency of the Data Layer, and minimize the adaption effort of the applications, transformation between different SQL syntax has to be implemented into the intermediate layer Goméz Sáez has provided, as mentioned above.

An SQL statement represents a certain action made onto the database. And to ensure the consistency of the applications, transforming SQL statement from one dialect to another, has to maintain its original intention. Thus, understanding the intention of an SQL statement is essential before transformation. An SQL statement follows a strict grammar, and can be broken down into clauses, each of which identifies a constituent objective of the statement. The syntax can be then further resolved to expressions, key words, schema objects, constants, etc. Therefore a parser is needed in place to manage the analysis of the SQL statement. In this thesis, we introduce an SQL parser which parses a statement with its source dialect grammar, and translates it into a hierarchy of Java classes, which is then in turn reconstructed into a new statement according to the target dialect.

Parsing statements into Java classes can be memory and time consuming, which may have significant performance impact on the existing system. On that account, we analyze the query throughput against the complexity of the statements in order to evaluate the drawback of the transformation and provide suggestions for future improvement.

1.2. Motivating Scenario

To adapt to the Cloud environment, one must choose a Cloud service provider according to their functional and non-function requirements, such as whether is there an automatic backup and restore service, how is the service metered, how is the availability, which deployment model to choose. Various factors affect the final decision of service provider and service product. And the chosen product may not always be fully compatible with the existing system.

Take the database application as an example. Applications are initially built based on a certain database system, e.g. MySQL or PostgreSQL as shown in Figure 1.1. And each database system from different vendors has their own sets of SQL syntax and functions which distinguish from the standard. This phenomenon is a result of the drastic competition between database developers, where each developer introduces new features into their product to accustom user's demand way before they become standard. Thus, migration tools and guidance across different databases are developed to help the transition between competitors. For example, Oracle offers a developer toolkit which identifies SQL statements in an application and makes appropriate changes to adapt the Oracle database system [LN12].

However, such migration will have significant impact on the existing applications, and will lead to more legacy issues.

On the other hand, the Cloud service providers offer even more alternatives, each with their own backend database systems and additional service limitation and enhancement. For example, Google Cloud SQL is actually a MySQL database that lives in the Cloud. However due to the Cloud environment, it has its own restrictions. For example, statement "`SELECT ...INTO OUTFILE/DUMPFILE`" is not supported in Google Cloud SQL [goo], contrary to MySQL.

In order to provide an unified access interface for the existing applications to Cloud and local databases, while preserving the transparency to minimize the changes that have to made for the applications, an intermediate layer is to be introduced where transforming the SQL statements between different dialects from different database vendors takes place, as shown in Figure 1.1.

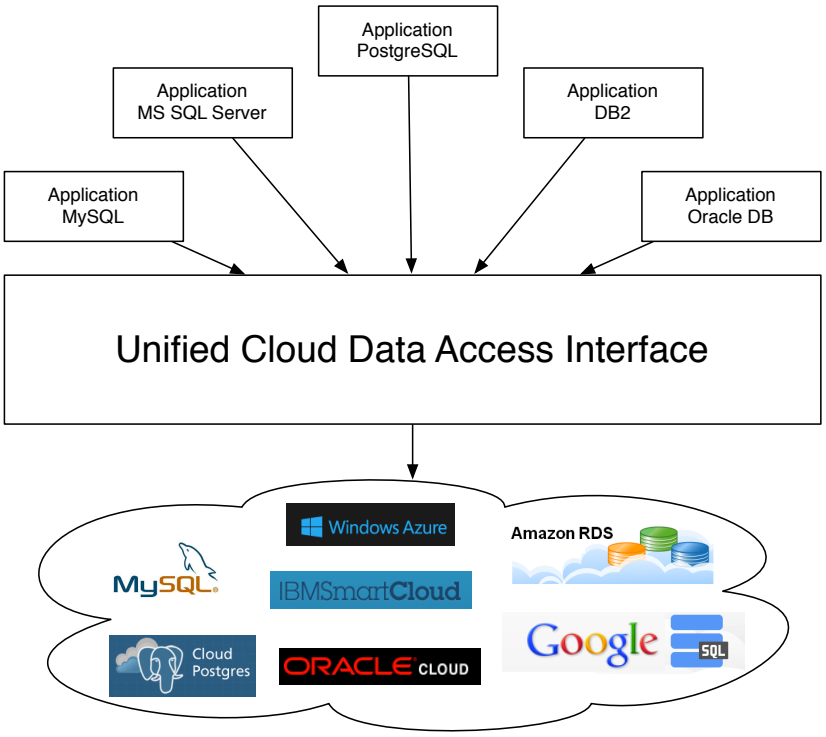

Figure 1.1.: Motivating Scenario

1.3. Definitions and Conventions

In the following section we list the definitions and the abbreviations used in this thesis for understanding the description of the work.

List of Abbreviations

The following list contains abbreviations which are used in this document.

ANSI	American National Standards Institute
API	Application Programming Interface
B2B	Business-to-Business Integration
BC	Binding Component
BLOB	Binary Large Object
CaaS	Context as a Service
CDASMix	Cloud Data Access Support in Multi-Tenant ServiceMix
CLOB	Character Large Object
DBaaS	Database as a Service
DDL	Data Definition Language
DML	Data Manipulation Language
DBMS	Database Management System
DBS	Database System
EAI	Enterprise Application Integration
ESB	Enterprise Service Bus
FDBS	Federated Database System
IaaS	Infrastructure as a Service
ISO	International Organization of Standardization
JBI	Java Business Integration
JDBC	Java Database Connectivity
LDAP	Lightweight Directory Access Protocol
MDBS	Multi-database System
MEP	Message Exchange Patterns
NM	Normalized Message
NMF	Normalized Message Format
NMR	Normalized Message Router
OSGi	Open Services Gateway initiative *(deprecated)*
ORDBMS	Object-Relational Database Management System

PaaS	Platform as a Service
RDBMS	Relational Database Management System
RDBS	Relational Database System
SaaS	Software as a Service
SCR	Service Component Runtime
SE	Service Engine
SOA	Service-Oriented Architecture
SQL	Structured Query Language
STaaS	Storage as a Service
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WSDL	Web Services Description Language
XML	eXtensible Markup Language

1.4. Outline

The rest of this document is organized as follows,

- **Chapter 2. Fundamentals** offers a broad illustrations of background knowledge that we used in our work, and are necessary for understanding this thesis, regarding Cloud Computing, Enterprise Service Bus, database system and SQL language.
- **Chapter 3. Related Work** enlightens the state of the art of existing technologies that coincide with our work of cooperating multiple database systems, and SQL transformation.
- **Chapter 4. Analysis and Specification** provides a full analysis of the system overview, functional and non-functional requirements that our design will meet, based on the system we are extending upon, that is the Multi-tenant ServiceMix with Cloud Data Access Support. It also explains the details of comparison of SQL statements between different dialects, in order to perform the transformation.
- **Chapter 5. Design** clarifies the system component structure, and the changes and extensions we need to make to enable SQL transformation between different dialects as OSGi Bundles in the ServiceMix. Moreover, the extensibility of the transformation components will be explained.

- **Chapter 6. Implementation** illustrates our implementation of the SQL Transformation components, most importantly, the implementation of SQL parser using JavaCC, as well as the effort of integrating the components into the existing system.
- **Chapter 7. Validation and Evaluation** demonstrates the methodology we used to validate and evaluate our implementation, as well as the resulting outcome. In addition, we provide an analysis of the algorithm complexity based on the performance evaluation.
- **Chapter 8. Conclusion and Future Work** concludes this thesis, and provides some perspectives on future development of the SQL transformation.

2. Fundamentals

In the former chapter, we explained that we are building a middleware solution to enable transparent Cloud database access with SQL transformation capability. In the following sections, we will introduce some basic concepts that are fundamental to our work.

First, we define several basic terms of database technology. The purpose of our work is allowing client's database applications to interact with databases on-premise and off-premise without modifying the applications. In order to do so, we need to understand the database system, and the communication between application and database system, more importantly the query language they use.

Furthermore, we need to have the knowledge of Cloud computing, especially about database in the Cloud. Moreover, we have to understand the original purpose of moving the database to the Cloud, and the subsequent limitations.

In addition, we'll explore the software architecture we extend in order to provide such a transparent access layer between applications and on-premise and off-premise databases. This solution, namely Enterprise Service Bus (ESB) serves beyond a role as middleware, also as business integration. We choose an open source implementation of ESB, the Apache ServiceMix to serve our purpose. It is in compliance with Java Business Integration (JBI) and OSGi specifications.

Last but not least, we'll introduce the Cloud Data Access Support in Multi-Tenant ServiceMix (CDASMix), the actual system we are extending. We'll briefly identify the system structure, and the area we work upon.

2.1. Relational Database

Edgar Codd, an employee from IBM, published a paper in June 1970, named *A Relational Model of Data for Large Shared Data Banks* in the *Communications of the ACM*. He utilized mathematical set theory to organize data into relation (table), which is composed of tuple (row) and attribute (column) [Cod70]. This led to the birth of the first ever relational database system, System R [CAB+81].

Later in 1985, Codd proposed the following 12 rules to govern the concept of *relational*, in order to fight against the misuse of the term. Since then, Codd's 12 rules have become the guideline of validating the "relational" characteristics of a database system [KKH08] [GW02]. Codd's 12 rules state:

1. All information in the database is organized logically into tables.

2. Stored data must be accessible using table, column and primary key.
3. NULL is generally defined as "missing information", not otherwise a string or a number
4. Information about the database is also stored in the database as regular data.
5. A single language must be able to define, manipulate, administrate the database.
6. Views and their base tables reflect each other's update.
7. Retrieve data, insert data, update data, or delete data must be done with a single operation.
8. Batch and end-user operations are logically independent from physical storage and access methods.
9. Batch and end-user operations can change the database schema without having to recreate it or the applications built upon it.
10. Integrity constraints must be also stored as regular data in the database.
11. Physical storage of the data does not affect the data manipulation language of the relational system.
12. Any row or set processing in the system, must obey the same integrity rules and constraints.

One or more databases often come with a piece of software that manages and interfaces with them, called Database Management System (DBMS). They then form together the so-called Database System (DBS). When the databases are all relational databases, the management system is then called a Relational Database Management System (RDBMS) and the whole system a Relational Database System (RDBS). Here we list the major RDBMS vendors in the world:

- **MySQL**[1], one of the most popular open source relational database management systems, is developed, distributed, and supported by Oracle Corporation. It offers both proprietary and community version of MySQL.
- **PostgreSQL**[2] is a powerful, feature-rich open source Object-Relational Database Management System (ORDBMS) best known for its excellent support for ANSI standard. Along with MySQL and SQLite, it is one of the three leading open source implementations of RDBMS.
- **Oracle Database**[3], being often cited as the first commercially available RDBMS, was developed in 1979 by the company Relational Software, Inc. (later changed to Oracle Corporation). It is the most popular database management system in the world and the leading relational database vendor by revenue [dbe] at the time of writing.

[1] MySQL: `http://www.mysql.com`
[2] PostgreSQL: `http://www.postgresql.org`
[3] Oracle Database: `http://www.oracle.com/us/products/database/overview/index.html`

- **Microsoft SQL Server**[4], developed by Microsoft Inc., was first released in 1998. It runs exclusively on the Windows platform. And it is one of the biggest commercial DBMS, besides Oracle and DB2.
- **IBM DB2**[5] traces its root back to System R, in 1974, the first relational DBMS based on the original concept of relational database by Edgar Codd from IBM.

2.2. Structured Query Language

Structured Query Language (SQL), was first developed along with the System R at IBM in the 1970s [CAB+81], as a result of the emergence of the Relational Data Model (see Section 2.1). It is a computer language specially designed for accessing and manipulating relational databases [KKH08].

SQL is a declarative language[6], which allows the user to work on the higher level of data structure. It complies with Codd's rules about the relational database and doesn't require users to specify or know about the physical storage of data. The same SQL statement can work on different database systems and databases no matter how their underlying structures are. SQL is simple, intuitive, and resembles the English language [7], which makes it easy to use and understand. Along with the success of relational data model, SQL proved to be the most popular language for it. In 1986, SQL was standardized first by American National Standards Institute (ANSI), and then followed by International Organization of Standardization (ISO) in 1987. Since then, it has been undergoing 7 revisions until the latest version *SQL:2011* [iso] at the time of writing.

2.2.1. SQL Statement

The main part of SQL language is *statement*. Each SQL statement performs a specific action (command) on the database, such as data definition, data modification, query, access control, etc. People often categorize SQL statements that modify and query database as Data Manipulation Language (DML), and those that define data structure as Data Definition Language (DDL) [BED01].

```
1 SELECT order_id FROM orders WHERE custKey = '123';
2 UPDATE customer SET payment = payment + 123 WHERE custName = 'John␣Smith';
```

Listing 2.1: SQL Statement Examples

[4]Microsoft SQL Server: `http://www.microsoft.com/en-us/sqlserver/default.aspx`
[5]IBM DB2: `http://www-01.ibm.com/software/data/db2/`
[6]Though, there are extensions to standard SQL which add procedural programming functionality. That is however out of the scope of this thesis.
[7]SQL was initially named Structured English Query Language (SEQUEL), and was later changed to SQL due to a trademark issue.

The first few tokens of a statement generally identify the action of the statement. In the example above in Listing 2.1, first line is a `SELECT` statement, and the second is an `UPDATE` statement, which respectively query and update the database. Such tokens are identified as *keywords*, same as `FROM`, `WHERE` and `SET`[8], which are a set of words that have special meanings in the SQL language and can be fairly easy to understand with their English implication. And *reserved words* are words that are reserved by the database system, which can not be used as *identifiers*, unless quoted. SQL distinguishes between the *keyword* and *reserved word* by specifying *reserved keyword* and *non-reserved keyword*.

The other tokens, namely `order_id`, `orders`, `customer`, are examples of *identifiers*. They are specified by database design or database system to identify names of tables, columns or other database objects, depending on the statement they are in, and their position in the statement. Since they have the exact same lexical structure as a *reserved word*, during design they must avoid conflict with the list of reserved words, or use quotation (e.g. "`select`"). However, quoted reserved word as identifier is generally not recommended.

Furthermore, there are *expressions*, like "`custKey = '123'`" and "`payment = payment + 100`". Besides *identifiers*, they are composed of *operators* and *literals*. *Operators* can be symbols or keywords depending on the systems, such as =, +, >, &, |, AND, OR, etc. *Literals* are constant values of various types. Depending on the database system, *literals* are implicitly-typed, such as string (`'123'`), numeric (123) and boolean (`TRUE`). They can also be explicitly-typed using standard SQL syntax *type 'string'*, such as date (`DATE '2013-11-28'`). The string literal specified will be converted to the corresponding type.

In Appendix B, we list several important SQL statements, and the comparison between different SQL dialects from different database vendors.

2.2.2. SQL Data Type

Each column of a relational table is declared with a data type [GW02]. SQL standard and each database vendor have their own set of data types they support. Generally speaking, there are the basics such as number, boolean, character, date and time, etc. And new data types are included as the technical demand grows, such as XML, JSON, Network Address, and so on. In addition, PostgreSQL also supports user defined data types, using command `CREATE TYPE` [pos]. Data types are mainly used in a `CREATE TABLE` statement, or the returning result set of a `SELECT` statement. A list of available data types in various DBMS and their comparison is presented in Appendix A.

2.3. SQL Parsing

In order to transform an SQL statement, an SQL parser is required to semantically understand the statement. An SQL statement is said to be transformable to another dialect, only if it can

[8]SQL statement is in general case-insensitive. Here the *keywords* are capitalized for easy reading.

replaced by another statement with the same semantics, that is to result the exact same action upon the database as the original statement.

We came upon an open source project *JSqlParser*[9] who parses an SQL statement and translates it into a hierarchy of Java classes, which is exactly what we can make use of. It is built mainly using *JavaCC* ™ [10], a Java parser generator. The *JSqlParser* project can be divided into two main parts. The first part compromises of a collection of Java classes representing the lexical components of an SQL statement, such as a table, a column or an expression. The second part is the *JavaCC* grammar file (JSqlParserCC.jj) which is used by *JavaCC* to generate the parser. The interesting part is you can inject Java code into the grammar file, which will be written to the generated parser as it is. The parser will then execute these code during the parsing process. *JSqlParser* utilized such feature to produce the hierarchy of Java classes during the parsing process, rather than generate the hierarchy afterwards. An example is shown in Listing 2.2.

```
1  Truncate Truncate():
2  {
3      Truncate truncate = new Truncate();
4      Table table;
5  }
6  {
7      <R_TRUNCATE> <R_TABLE>
8      table = Table()
9      {
10         truncate.setTable(table);
11     }
12     {
13         return truncate;
14     }
15 }
```

Listing 2.2: JavaCC Grammar File Snippet

This code snippet parses a Truncate statement, and returns an instance of `Truncate` class. At line 1, the first word "`Truncate`" is the class name of the return value. And "`Truncate()`" represents a definition of a lexical structure which direct follows. Line 2-5 are Java code for initialization which are executed whenever such lexical structure is parsed. At line 7, token `<R_TRUNCATE>` ("TRUNCATE") and `<R_TABLE>` ("TABLE") have to be matched, followed by a `Table()` structure which returns an instance of `Table` class. The curly-bracketed code at line 8 is the injected Java code which invokes the method "`setTable`" of the instance "`truncate`" with the "`table`" we just received. Finally at line 10, the resulted "`truncate`" instance is returned. Such grammar file is quite intuitive. It cooperates with Java to parse a statement and at the same time produce instances of Java classes we can directly refer to.

The downside with *JSqlParser* is that it uses an unknown dialect of SQL Language. Therefore

[9] JSqlParser Project: `http://jsqlparser.sourceforge.net/`
[10] Java Compiler Compiler ™ (JavaCC ™): `http://javacc.java.net/`

we have to adapt the grammar for different SQL dialects, and extend it with SQL transformation functionality.

2.4. Java Database Connectivity

Java Database Connectivity (JDBC), is a standard Java-based database access technology, which provides a standardized client-side Java Application Programming Interface (API) for interacting with a backend database system [jdb]. To take advantage of JDBC, a vendor-specific JDBC driver has to be installed prior to the connection, which implements the actual communication to the backend database system.

The important interfaces of JDBC API include `java.sql.Statement` and `java.sql.ResultSet`. The `Statement` interface is used to execute a static SQL statement and returning the results it produces, as in Listing 2.3

```
1 Statement stmt = conn.createStatement();
2 ResultSet rs = stmt.executeQuery("SELECT␣*␣FROM␣table1␣WHERE␣id␣=␣1");
3 ResultSetMetaData metaData = rs.getMetaData();
4 if(metaData.getColumnType(1) == Types.INTEGER){
5     if(rs.next()){
6         int i = rs.getInt(1);
7     }
8 }
```

Listing 2.3: JDBC Statement Execution Examples

As we can see, the statement is executed in plain text form. This allows the acceptance of any SQL statements supported in different database systems without restriction. This means the application is free to use as much SQL functionality as desired, though it runs the risk of incompatibility between SQL dialects, which is the problem we will solve in this thesis.

The `ResultSet` interface is the returning result of an SQL query (a SELECT statement or similar). It resembles a relational table containing rows of data in different data types. In contrast to the statement execution, the result set retrieved via JDBC is standardized. JDBC defines a set of generic SQL type identifiers in the class `java.sql.Types`, which correspond to the most commonly used SQL data types. And for each type, there is a method in `ResultSet` to retrieve the data as a Java object (an instance of a primitive type or a Java class) [jdb], as shown in Listing 2.3, Line 4-8.

2.5. Cloud Computing

Internet or a network, conventionally, is illustrated as an image of Cloud just like in Figure 2.1, representing a connected cluster with unknown underlying structure. This metaphor is then used to describe a computing model highly based on the computer network (typically the

Internet), that is Cloud computing. Cloud computing is not strictly a scientific or technical term, rather than a marketing concept. It comes close to the term *Distributed Computing*. It is, according to National Institute of Standards and Technology (NIST), "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [MF11].

As described in the article *The NIST Definition of Cloud Computing* [MF11], the consumer of Cloud computing can utilize the computing capabilities as needed without directly interacting with each service provider. The computing capabilities are accessible through the network with standard protocols. The computing resources of the provider are pooled together to serve multiple consumers using virtualization and multi-tenant model. Depending on the consumer's demand, resources can be allocated and released accordingly. The measurement, monitor and control of resource usage is transparent to both consumer and provider. An infrastructure, including software and hardware, that fulfills the descriptions above can be qualified as a Cloud infrastructure.

The services that the Cloud provider offers are categorized into 3 models: *Software as a Service (SaaS)*, *Platform as a Service (PaaS)*, and *Infrastructure as a Service (IaaS)* [MF11], as shown in Figure 2.1. This is a hierarchical categorization where the higher level service is somewhat an abstraction of the lower level. PaaS abstracted the underlying physical infrastructure, as in processing power, storage and network, which are all hidden from the consumers of PaaS. And SaaS offers user a pure software experience without caring about the platform it is running on.

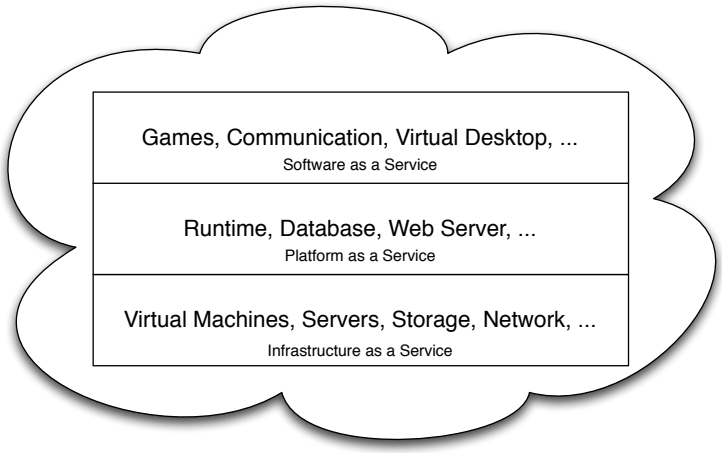

Figure 2.1.: Cloud Computing Service Model

Such abstraction also determines how much a consumer has control over the Cloud resources. IaaS consumers have control over what is built upon the provided infrastructure. They have very limited control over the underlying hardware resources. PaaS consumers have the capability to deploy applications onto the Cloud infrastructure, but little control over the

application-hosting platform. With SaaS, Cloud providers offer applications running on a Cloud infrastructure. Consumers can use the applications to their own need, but cannot modify or even control individual application capabilities. The benefit behind this is that, the consumer can focus on what they need, while the service providers take care of the underlying structure (e.g., hardware and software upgrade, data backup and restore, and network maintenance). This also decreases the cost for consumers, since the entailed cost of building the Cloud infrastructure is divided among tenants at the provider side.

Following this logic, more of the "XaaS" models appeared on the market, providing a certain computing capability over the Cloud, such as *Storage as a Service (STaaS)*, *Database as a Service (DBaaS)*, *Context as a Service (CaaS)*, etc. In our thesis, we focus on the DBaaS, where Cloud provider offers the capabilities of a DBS over the Cloud, and consumers can access the databases through network using standard protocols. Most popular Cloud providers and database vendors offer DBaaS, like Amazon RDS[11], Google Cloud SQL[12], Oracle Cloud[13], Windows Azure SQL Database[14]. And likewise, we can make use of the database systems they provided, but have limited control over the management and configuration of the systems. On the other hand, system upgrade, data backup and restore, security and system scaling are all seamlessly taken care of by the provider.

Another way of achieving database on the Cloud is to deploy our own database system onto a Cloud infrastructure. There are plenty of IaaS providers who offer a wide range of virtual or dedicated machine instances, with different processing powers, storage sizes, and network performances. Upon them, we can install any or a selective type of operating system, and database system. This approach gives us almost full control over the DBS and its running environment.

NIST also defined 4 models how the Cloud Infrastructure can be deployed. *Private Cloud* is provisioned exclusively for a single organization; *Community Cloud* is for a specific group of consumers from different organizations; *Public Cloud* opens to general public; and *Hybrid Cloud* is a combination of any two or more of different models (private, community and public) that remain unique entities.

2.6. Enterprise Service Bus

Chappell defines ESB as a standards-based integration platform that combines messaging, Web services, data transformation and intelligent routing to reliably connect and coordinate the interaction of significant numbers of diverse applications across extended enterprises with transactional integrity [Cha04].

ESB plays a central role in the Service-Oriented Architecture (SOA), where collections of discrete services collectively compose complete functionality. It provides the implementation

[11] Amazon RDS: `http://aws.amazon.com/rds/`
[12] Google Cloud SQL: `https://developers.google.com/cloud-sql/`
[13] Oracle Cloud: `https://cloud.oracle.com/mycloud/f?p=service:database:0`
[14] Windows Azure SQL Database: `http://www.windowsazure.com/en-us/services/data-management/`

backbone for a loosely coupled, event-driven SOA, with a highly distributed universe of named routing destinations across a multi-protocol message bus [Cha04]. In the integration area, the ESB concept distinguishes itself with its highly distributed bus model and open standards from other solutions, such as Enterprise Application Integration (EAI), which is based on a centralized hub-and-spoke model [RD09].

In order to better understand what an ESB is, it is easier to learn from what it does. ESB hides location information of services by providing a central platform where communication can be achieved without directly coupling service consumers and service providers. While each application utilizes different transport protocols, an ESB should be able to seamlessly integrate these applications with protocol conversion [RD09]. As message exchange is the key part of an integration solution, an ESB should provide functionality to transform messages with different formats, route messages between endpoints, and enhance based on the incoming message. Authentication, authorization, and encryption functionality should be provided by an ESB for securing message exchange to prevent malicious use of the ESB as well as satisfy the security requirements of the service provider [RD09].

2.7. Java Business Integration

To implement an integration solution for enterprise applications, tradition solutions such as EAI and Business-to-Business Integration (B2B) sought non-standard technologies to create functional systems [RD09]. This led end users locked in with specific vendors, and a hard time switching between products.

JBI is a specification developed under Java Community Process as an open-standard approach for integration solutions. It defines an architecture that allows the construction of integration systems from plug-in components, that interoperate through the method of mediated message exchange. The message exchange model is based on the Web Services Description Language (WSDL) 2.0 or 1.1 [THW05].

JBI plug-in components are responsible to provide and consume services, which are respectively named *Service Consumer* and *Service Provider*. Similarly to Web service model, a *Service Provider* describes and publishes its services using WSDL, while *Service Consumer* consumes services using WSDL *operations*, by exchange messages via a set of four WSDL-defined Message Exchange Patterns (MEP).

Based on architectural principles, JBI components are divided into two types: *Service Engine (SE)* and *Binding Component (BC)*. SEs provide business logic and transformation services to other components within JBI as well as consume such services, while BCs provide connectivity to services external to a JBI environment. Such separation of business logic and communication logic reduces implementation complexity, and increases flexibility [THW05].

The communication between components is realized by exchanging Normalized Message (NM)s through the Normalized Message Router (NMR). A NM consists of two parts: an abstract eXtensible Markup Language (XML) message and message metadata. The XML message is also referred to as “payload” which conforms to an abstract WSDL message

type, describing the abstract message type, abstract operations, etc. Another portion of the payload can be made up of attachment, referenced by the payload, and is contained within a data handler that is used to manipulate that contents of the attachment itself [THW05]. The message metadata, or message properties, hold extra data associate with the message, which can be used by developer to store identity information, context information, etc. NMR is a key operational component in a JBI environment who acts as a communication mediator. Its main functionality is routing NMs between Service Consumers and Service Providers. Such mediated message-exchange processing model also allows the NMR to perform additional processing during the lifetime of the message exchange [THW05].

2.8. OSGi Framework

OSGi Framework is a general-purpose and managed Java framework that supports the deployment of extensible and downloadable applications known as *bundles* [OSG11]. Its functionality is defined as the layered model shown in Figure 2.2.

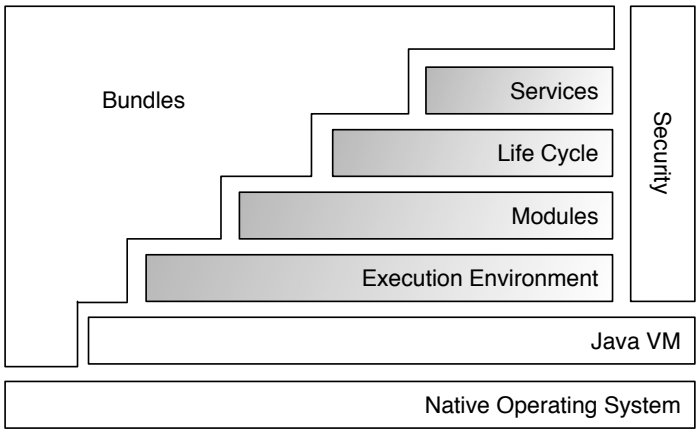

Figure 2.2.: The Layered Model of OSGi Framework [OSG]

The Security Layer is an optional layer that provides a well-controlled environment for deploying and managing applications. It is based on the Java 2 security architecture [OSG11] which defines a secure packaging format as well as the runtime interaction.

The Module Layer specifies the packaging, deploying and validating of the modularization unit, the *bundle*. A bundle is a special-formatted Java Archive (JAR) file, with additional resources and manifest specifications. It also regulates the Java package sharing between bundles, as well as resolves dependency relationships.

The Life Cycle Layer defines how bundles are started and stopped, as well as how they are installed, updated, and uninstalled. The lifecycle management requires no system reboot to install or uninstall bundles. It exposes API to bundles, which allows lifecycle managing operations during runtime [OSG11].

The Service Layer provides a programming model for developers, which decouples the service's specification and implementation, thus allows service binding with interface specification while the selection of service implementation can be deferred to runtime. The framework provides a service registry where bundles can register new services and look up existing services to adapt to the current capabilities. This makes an OGSi bundle extensible even after deployment. We'll take advantage of this feature, to make our implementation flexible to adapt to the incoming request according to its requirement, as well as extensible.

Apache Karaf is a lightweight OSGi based runtime, which is used by Apache ServiceMix 4.3.0 as the underlying OSGi server runtime. The following section will give a further in detail introduction on Apache ServiceMix.

2.9. Apache ServiceMix

Apache ServiceMix is an open-source ESB, which provides an integration container that coordinate various Apache projects into an integration platform. This thesis is based on a modified and extended version of Apache ServiceMix 4.3.0 by Goméz Sáez [Sáe13a], which we refer as CDASMix (see Section 2.10).

The significance of using Apache ServiceMix version 4, compared to the predecessors, is its compliance of OSGi framework through Apache Karaf. Initially, Apache ServiceMix is based on the JBI specification, upon which various components are developed, and inherited to the newer version, such as servicemix-camel, servicemix-bean, servicemix-jms, etc.

As introduced in Section 2.8, Apache Karaf provides a light-weight OSGi container into which various bundles can be installed. In Apache ServiceMix, there is a folder "`/deploy`" shipped under the installation location, acting as a hot deployment folder. OSGi bundles and JBI components can be directly deployed by putting them into the "`/deploy`" folder, or deleting them to uninstall. Karaf also provides a user friendly command line console to manage the life cycle of components.

Aside from the flexible lifecycle management, the service layer of OSGi is also realized via *Blueprint Service*, *Declarative Service* [OSG12], etc. *Blueprint Service* is enabled in Karaf by default. It injects proxy object as service reference, while the backing service it defers can be dynamically and independently registered or replaced, which makes the service's availability much more flexible. *Declarative Service*, on the other hand is an add-on feature to Karaf, realized by Apache Felix Service Component Runtime (SCR) bundle[15]. *Declarative Service* registers first its service component via SCR with the service registry, and the service implementation is later "lazily" loaded or instantiated until the service is actually requested by a client.

Even though documented as deprecated, Apache ServiceMix 4.3.0 is still in full support of JBI 1.0 specification, with a set of legacy JBI components deployed as OSGi bundles, which are

[15]Apache Felix Service Component Runtime (SCR): `http://felix.apache.org/documentation/subprojects/apache-felix-service-component-runtime.html`

still of great importance. For example, Apache Camel is leveraged as a routing and mediation engine, especially with the Camel NMR Component. NMR as introduced before, is a central message bus defined in JBI specification, and it's remodeled in Apache SerivceMix as an OSGi bundle with NMR API, which enables the inter-communication between JBI components and OSGi bundles [Theb].

2.10. Cloud Data Access Support in Multi-Tenant ServiceMix

To achieve a seamless adoption of Cloud database services, Goméz Sáez presented us an ESB with Cloud data access support, and the support of multi-tenancy. It acts as a data access layer of the applications, and access multiple Cloud database services, to provide transparent Cloud data access. We refer to it as CDASMix.

Its system architecture is shown as in Figure 2.3, that provides MySQL communication protocol to the applications, and routing to the backend Cloud or local databases.

CDASMix is based on a multi-tenant ServiceMix prototype which Muhler and Goméz Sáez provided in their work [Muh12] and [Sáe12], named ServiceMix-mt, which provides multi-tenant awareness to the ServiceMix by injecting tenant context in the JBI endpoint's Uniform Resource Locator (URL)s [Muh12], and by providing a Normalized Message Format (NMF) with tenant context information in its properties for routing in the NMR [Sáe12].

As shown in Figure 2.3, the Proxy Bundle communicate with the application according to MySQL communication protocol via Transmission Control Protocol (TCP), which eventually receives the SQL queries. It then acquires the according tenant and data source information from the service registry, where the information needs to be registered before hand. It's notable that a registry cache is in use to improve the performance.

A NM is then created and sent to the corresponding tenant-aware JBI endpoint in Apache Camel via NMR, where it's further routed to the *JBItoCamelJdbc* endpoint deployed in the OSGi container. At last the NM is forwarded to the *CdasmixJDBC* component, which selects the appropriate JDBC driver, creates a connection, demarshals the request, and sends the request to the backend Cloud database service [Sáe13a].

As indicated in Figure 2.3, a query and data transformation might exist between the Proxy bundle, and Camel JBI endpoints, through NMR. This is where we will insert our work, to achieve the SQL transformation between source and target dialects.

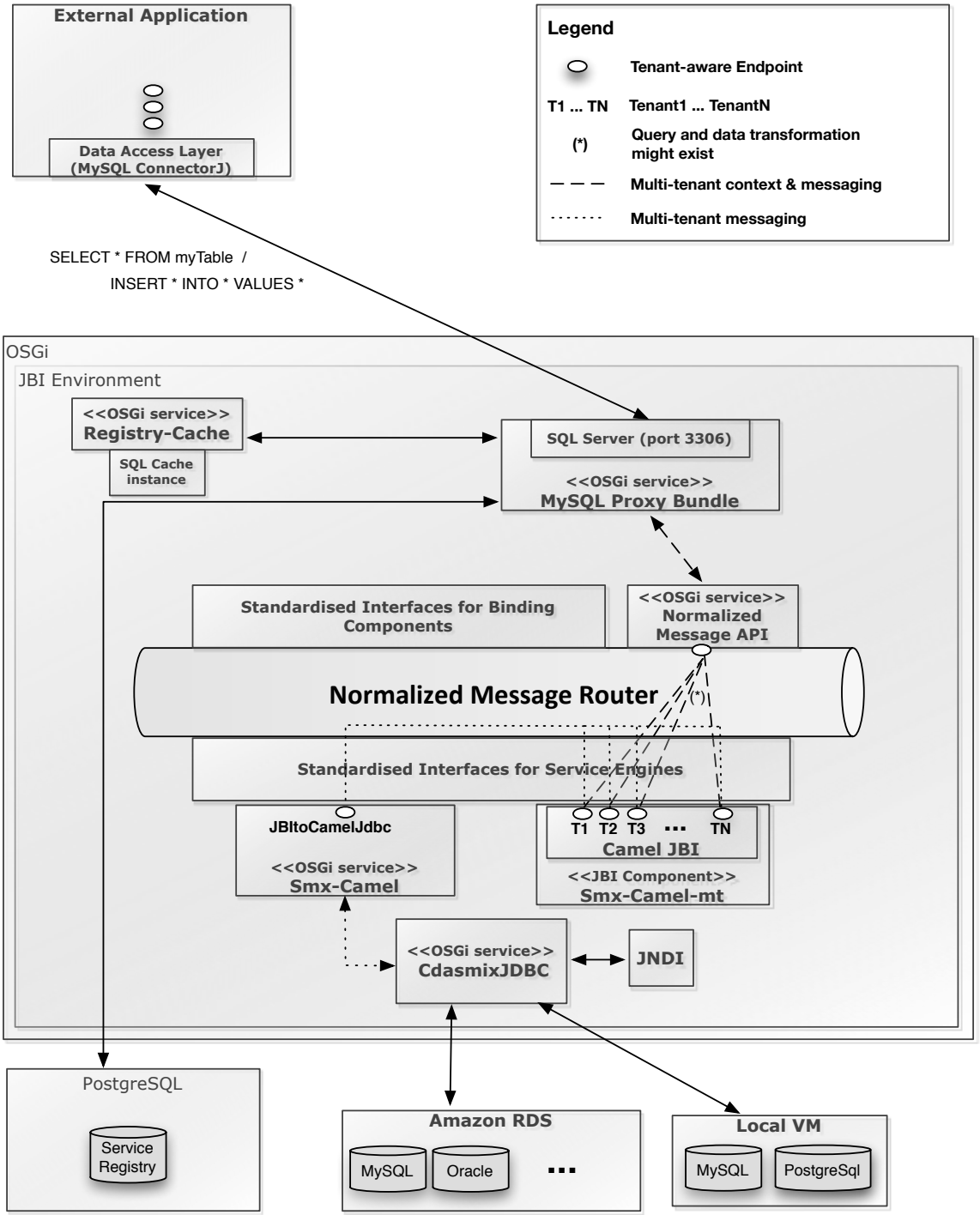

Figure 2.3.: Architectural Overview of CDASMix for providing support of relational database access [Sáe13a]

3. Related Work

In this chapter, we'll discuss other's work related to the situation of proper integration regarding migration of Database Layer to the Cloud, especially focusing on query language integration of relational databases. As an integration solution, our system provide a reliable, secure and transparent communication between on-premise applications and off-premise Cloud databases with multi-tenant, multi-protocol and multi-database support.

3.1. Multi-database System

As pointed out, we provide access to multiple backend database systems, which may categorize us as a Multi-database System (MDBS). A popular implementation architecture of MDBS is the mediator/wrapper approach, as shown in Figure 3.1, which our system structure complies to as well.

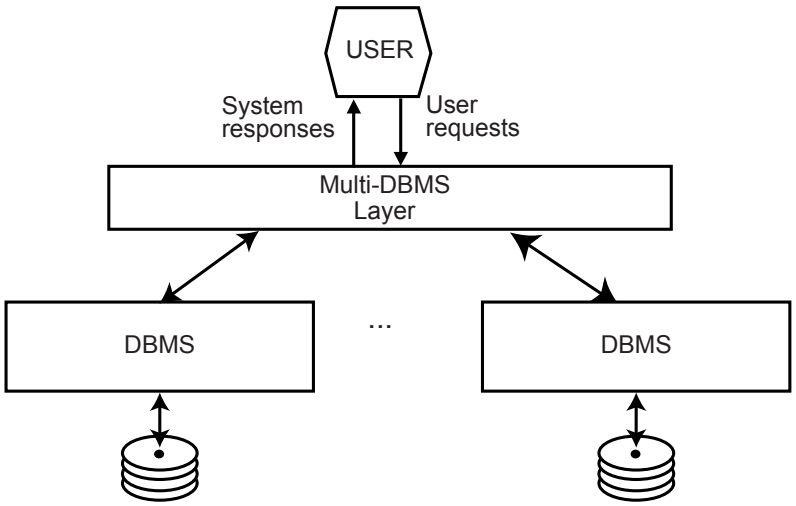

Figure 3.1.: Components of an MDBS [OV11]

A MDBS can be characterized along three orthogonal dimensions: *autonomy*, *distribution*, and *heterogeneity* [SL90]. *Autonomy* defines the decentralization of control within an MDBS, indicating that the individual DBS participating in an MDBS retains its independent control of the operation. Such system is also identified as a Federated Database System (FDBS). Our system presents no such characteristic, as all component DBSs are centrally controlled and accessed solely through our system.

Distribution refers to the physical distribution of data over multiple locations, while the user sees the data as one logical pool. Regarding our system, as according to [Sáe13a], a source data source (physically accessed by tenant) can be connected to one or more target data sources (logically accessed by tenant, but physically accessed by the system), thus the data in a source data source can be distributed among multiple target data sources in various database services (locations). However, such distribution is only limited to physical storage of data, since we provide no support for logical access of the distributed data, in the sense that join of data cross different target data sources is not supported in our system. In other words, one SQL statement can only refer to schema objects (tables) stored in the same backend database. Such limitation would identify our system as a *remote DBMS interface*, rather than an MDBS, since it only provides access to multiple DBMSs *one at a time* (e.g., no joins across two databases) [SL90].

Heterogeneity is due to the technological differences, in our case the differences in RDBMS. As we mentioned in Section 2.1, there is a variety of database vendors with different designs of database system. Fortunately, the Codd's rules had governed the relational databases to separate its end-user operations with the underlying physical structures and access of the databases. And the wide adaption of SQL language as the end-users' tool for accessing and manipulating relational databases also eases the process of unifying different database systems. Even so, the different dialects of SQL supported by different RDBMS still contribute to the heterogeneity. In addition, the representation of the data stored within a RDBMS also depends on the implementation of the database. Such difference must also be addressed in our integration solution. A second type of *Heterogeneity* is the *Semantic Heterogeneity*, which occurs when there is a disagreement about the meaning, interpretation, or intended use of the same or related data [SL90]. This is however not our concern, as [Bac12] and [Sáe13a] guaranteed the semantic consistency throughout the database migration and access process, which in turn minimized the impact we made on the application layer and above. As acknowledged above in *distribution*, we are dealing with one target database at a time, thus the *heterogeneity* that we handle is between the source and target database. *Heterogeneity* across multiple target databases is out of the scope of this thesis.

3.2. Application Migration

Holding SQL language as the rule of thumb, the heterogeneity of the query language across different DBSs require the integration solution to offer a unified view of different dialects. In addition, vendor-specific data representation also constrain the process of integration.

In order to address the incompatibility issue across different database systems, lots of vendors actually provide tools, or compatibility features built in the DBS to help clients migrate from one database to another. *Cloud Data Migration Application* from [Bac12] took care of the database object migration scenario in our case, however one step is missing, namely *Application Migration* [LN12]. As of a database application, it means to adapt the application for the newly deployed database system, mainly the embedded SQL statements. *Oracle SQL*

Developer[1] is a toolkit developed by Oracle to help migrate third-party databases to Oracle Database. It includes an automated tool, named *SQL Migration Scanning Tool*, which identifies SQL in the applications and makes the appropriate changes [LN12]. The essence of such tool is similar to what we've implemented, which fundamentally requires an SQL parser and specific rules to translate SQL statements into another dialect. Microsoft offers a similar tool called SQL Server Migration Assistant (SSMA)[2], which provides application migration support as well. The essential purpose of these migration tools, is to change the applications to accommodate with new database system. While this disobeys the original goal of our system, which requires minimum impact over applications and the layers above, it may also introduce further problems when the backend database changes again. Another problem with such tools, is that they often deal only with static statements written in the applications. However, many applications build SQL statements dynamically based on user input in the form of variables that may contain table names, a list of column names and the predicate clause to be used in the statements, and then execute them. These statements are difficult to be identified, as they are not presented as a fully-formed SQL statement.

On the other hand, our implementation won't require the *Application Migration*, as we utilized a mediator/wrapper approach to provide a transparent migration process for the applications. Applications will send their SQL statements as before, via our proxy; the proxy will intercept the statements, and send them to a transformation service, where the statements are transformed on-the-go and eventually forwarded to the backend database. Since we provide multi-protocol and multi-database support, changing backend database won't require any further alteration of the applications. Furthermore, the statements that are executed by the applications are all processed by the transformation bundle when necessary, whether they are static or dynamic. Since the transformation happens after the application sends out the queries, it means that they are all fully complete SQL statements.

In addition to the request transformation, our system needs to handle the additional transformation of the response. In contrast to the migration approach, the response from the backend database is inconsistent to the client applications, mainly on the account of the data representation returned from the database. A mapping has to be presented to correlate the data representations (data types) in the target database and the ones used in the source database.

3.3. SQL Transformation

As pointed out in Section 3.2, there are two types of transformation tools. The *static transformer*, like *SQL Migration Scanning Tool*, transforms only static SQL statements embedded in an application. Such transformation often incorporates with a total migration and adaption to new database system, meaning the application will adapt to new database system

[1] Oracle SQL Developer: `http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html`

[2] Microsoft SQL Server Migration Assistant: `http://msdn.microsoft.com/en-us/library/hh302873(v=sql.105).aspx`

permanently. The *dynamic transformer*, like our implementation, transforms SQL statements on-the-go. It usually resembles a middle layer, which acts as a proxy and intercepts SQL statements that are sent out by the applications, then transforms them before forwarding to the destination. There are several commercial tools on the market offering such feature as a *dynamic transformer*, such as *General SQL Parser*[3], and *SwisSQL API*[4].

Despite which kind the transformer is, in order to transform an SQL statement from one dialect to another, it often involves a language parser to parse a text representation of a statement into a tree structure, which will then be reorganized into a new text representation according to a set of predefined transformation rules. Writing a parser from scratch can be a cumbersome job, hence several tools are developed as a *parser generator*, which generate language parser code in different programming language based a on predesigned context-free grammar.

SQL Migration Scanning Tool utilizes ANother Tool for Language Recognition (ANTLR)[5] as a parser generator. ANTLR is an LL parser, meaning it parses the input from *L*eft to right, and constructs a *L*eftmost derivation of the sentence. It's a free software under the BSD License, however the reference tutorial is published as a commercial book, *The Definitive ANTLR 4 Reference* [Par13]. We chose another similar tool named JavaCC, as mentioned in Section 2.3. There is no available quality and performance comparison between these tools. The choice is barely made as the result of the open source project *JSqlParser* we based on.

General SQL Parser and *SwisSQL API* are commercial tools that offer the functionality of SQL transformation. *General SQL Parser* supports major programming language like Java, C#, VB.NET, etc. It's mainly an SQL parser product, with additional SQL processing features, such as statement refactoring, SQL objects identification and SQL transformation. *SwisSQL API* is a Java library for on-the-fly query conversion. It can be deployed as a library in the application to be directly invoked from inside a Java application. In addition, it offers a JDBC Wrapper mode, where it can be used as a bridge between the application and the database specific JDBC driver, so the application can leverage seamless integration by loading the *SwisSQL* JDBC driver.

[3]General SQL Parser: `http://www.sqlparser.com/`
[4]SwisSQL API: `http://www.swissql.com/products/sqlone-apijava/sqlone-apijava.html`
[5]ANother Tool for Language Recognition (ANTLR): `http://www.antlr.org`

4. Analysis and Specification

In this chapter, we will provide a detailed analysis of the system components that our work is based on and we extend in order to integrate SQL Transformation solution into the system. Furthermore, we analyze the SQL dialects that are relative to our system based on the system and resource limitation. Last but not least, we will enlist the functional and non-functional requirements the system need to fulfill.

4.1. System Overview

To extend the CDASMix system with SQL transformation functionality, we need to first identify the current system structure, and its components that are related to our extension, as shown in Figure 4.1. As pointed out in Section 2.10, a database server proxy is presented in CDASMix to handle the communication with external applications. And a normalized message router is used as a centralized message bus for internal message exchange. In addition, a service registry stores data source information of tenants regarding source and target data sources. At the end, a JDBC component is responsible for the communication with the target data sources.

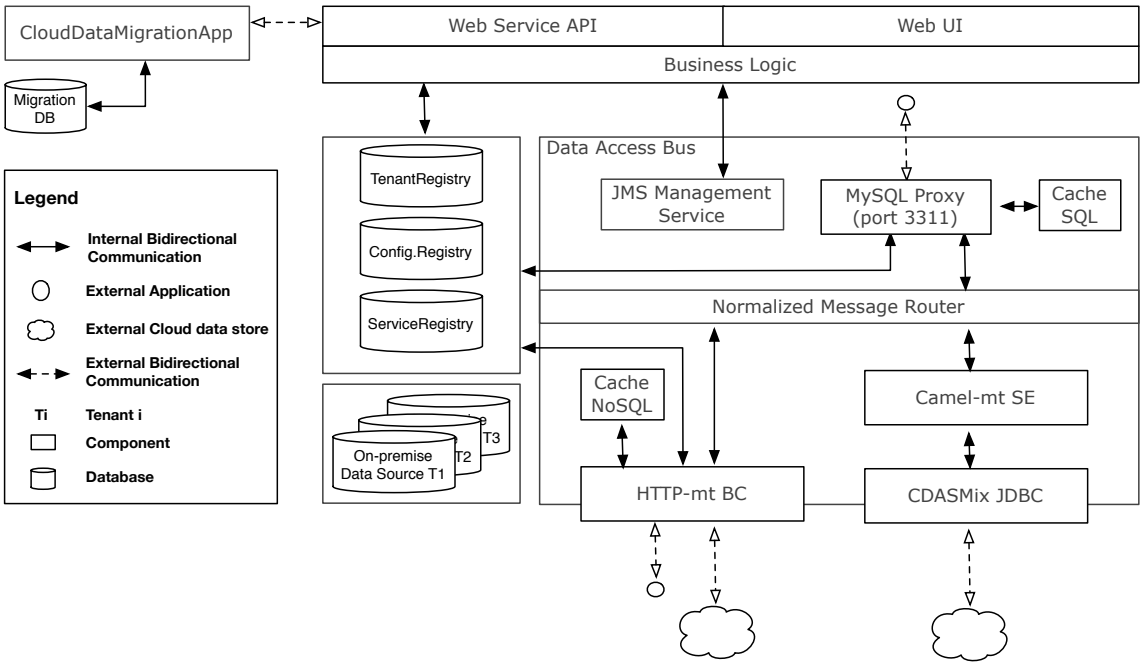

Figure 4.1.: Component Overview of CDASMix [Sáe13a]

4.1.1. Cloud Data Migration Application

Before a user can access the Cloud databases through CDASMix, it must first migrate its existing data to the destined Cloud database. *Cloud Data Migration Application* provided by Bachmann [Bac12] provides the support for migrating data from a traditional database and a Cloud one, or between Cloud databases. Before migration, the user has to provide the source and target data source information, based on which the application will identify the possible incompatibilities and present them to the tenant. After solving the incompatibilities, the application can then migrate user's data with provided access credentials.

Since we are focusing on RDBS, migrating data from one RDBS to another is our only concern. *Cloud Data Migration Application* provides support of migrating schema data, views and indexes. An important aspect of the migration process, is that the migration should minimize the impact it has over the database clients (applications). Therefore, before and during migration, the migration application preserves the names and structures of the database as much as possible. When there is conflict, for example when one table has a name that is reserved in the target database, the migration application will notify the tenant prior the migration [Bac12].

4.1.2. Database Server Proxy

A database server proxy is an application that sits between one or more database clients and servers, which communicates over the network using database specific network protocol. It's normally used for load balancing, failover, query analysis, query filtering and modification, etc. [mys]. In CDASMix, a Java implementation of MySQL Proxy developed by Continuent Inc. named Tungsten Connector [Con], is used to fit in our system as the server proxy for MySQL applications, as shown in Figure 4.1. A PostgreSQL Proxy is deployed as well for PostgreSQL applications. For convenience, we'll explain further with MySQL proxy as an example.

In MySQL communication protocol, there are 4 phases: Connection Phase, Authentication Phase, Command Phase, and Disconnect Phase. In Authentication Phase, the database client will send a packet with its user name (compromised of tenant ID and user ID [Sáe13a]), connecting database name, and password. The authentication information will be unpacked in the proxy bundle, and used to verify against the entry in the service registry. In addition, these information will be kept across the connection for future use in the Command Phase.

In the Command Phase, the database client sends command packets, importantly the query commands, to the proxy. The queries will be analyzed against its query type, and the related database objects (tables). As introduced in Section 2.10, target data source information is precedently registered in the service registry, in accordance with the source data source and tenant information. Therefore, the proxy bundle can look up the service registry, with the tenant ID, user ID, the database name, and the table name, for its actual physical storage location, namely the target data source. Up till this point, the proxy can then decide whether a transformation is needed based on the target and source data source types, i.e. the value of

dsType attribute. The format of this value follows the structure: family-mainInfoStructure-secInfoStructure-version, e.g. mysql-database-table-5.1.2 for a MySQL 5.1.2 database [Sáe13a]. As we are dealing solely with relational databases, the attribute we need to check is the family and version, e.g. mysql 5.1.2. All the information mentioned above, including source and target data source information and queries, is then marshaled into a normalized message with a predefined format and handed over to NMR, which we will introduce in the following section.

In correspondence with the request, a response is received containing the results of the queries when applicable. The result data and its metadata are in conformance with the JDBC specification (see Section 4.1.4), meaning the data are all converted to Java types (`int`, `float`, `java.sql.Timestamp`, etc.), and the data types in the metadata are specified using class `java.sql.Types`.

4.1.3. Normalized Message Format

The functionality of a NMR is introduced in Section 2.7. Another important aspect of the NMR for us is the format of the NM we use in our system, as shown in Figure 4.2.

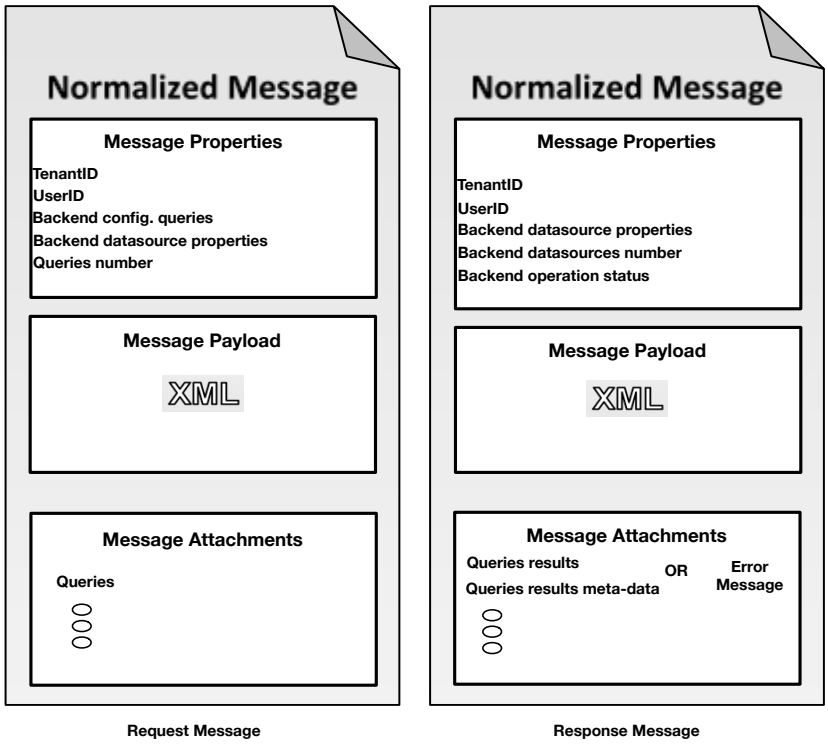

Figure 4.2.: Design of the Normalized Message Format Used in the System [Sáe13a]

There are two types of message, the request message and the response message. They are distinguished by their message properties and more importantly the content of their attachments. As we can see in the figure, backend data source properties, including source

and target data source, are stored in the message properties. The response message has a property of *backend operation status*, indicating whether the queries in the request are executed successfully. When success, the attachment of the response message will contain the results of the request, or the error message when otherwise. On the other hand, the attachment of the request message contains the queries. As we can see, multiple queries in one request is supported in our system, though for each query, only one pair of source and target data source can be specified [Sáe13a].

4.1.4. CDASMix JDBC Component

The connection to the target data sources from CDASMix is established in the *CDASMix JDBC Component*, as shown in Figure 4.1. As its name indicates, it uses the corresponding JDBC API to interact with backend databases. Based on the target data source, the bundle selects dynamically the corresponding JDBC driver to connect to the target database. JDBC technology provides an unified interface for accessing and modifying different databases, as well as an unified response from different database systems [jdb]. Most importantly the `java.sql.ResultSet` interface helps retrieving the returning result of an SQL query converted into JDBC types. The returning result set, along with its metadata is retrieved by the component according to the JDBC specification, and marshaled into a response NM as shown in Figure 4.2. This message is then sent back to the database server proxy within CDASMix.

4.2. SQL Dialects

As mentioned is Section 2.2, each relational database vendor offers their own version of SQL language, we refer to as *SQL dialect*. And we name them after the name and version of the database they belong to, such as MySQL 5.6 or PostgreSQL 9.2. When transforming (or translating) an SQL statement from one dialect to another, we name the dialect the statement already complies the *source dialect*, and the dialect the transformed statement will comply the *target dialect*.

4.2.1. Source Dialect

In Section 4.1.2, we specified that the SQL Transformation component is tightly followed after the database server proxy. The availability of the proxy component determines the SQL dialect we will transform as the source dialect. So far, we have the implementation of MySQL Proxy, and PostgreSQL proxy in place, which we will adapt to take advantage of the transformation component.

An important step of SQL transformation is parsing (see Section 2.3). A parser will parse the statement according to the SQL grammar of the source dialect. Due to this reason, the transformation process is grouped according to the source dialect. In other words, a

transformation service will transform a statement from one specific source dialect to various target dialects.

For MySQL database, we use the latest General Available (GA) Release of MySQL Community Server 5.6, as MySQL Community Server 5.7 is still in development at the time of writing. The documentation of the SQL syntax can be found in [mys], and a MySQL parser is developed according to the description in that document.

PostgreSQL released version 9.3 on 9th September 2013[1], therefore we developed the PostgreSQL parser based on the document [pos]. According to [KKH08], PostgreSQL is best known for its excellent support for the ANSI SQL standard.

4.2.2. Target Dialect

The target dialect depends on the target data source our system support, which include both on-premise and off-premise databases. On-premise or local databases, are the ones that reside on the same location as the applications. They specify their own SQL dialects and keep them well documented as those we used in the source dialect. However, for off-premise or Cloud database, it can get a little bit complicated.

There are two types of Cloud database service: *Database as a Service (DBaaS)* and *Database in a Cloud Infrastructure*.

Database as a Service, is a service model of Cloud computing that abstracts everything below a database system (including operation system, hardware, etc.) from the service consumer, and offers solely a database system as a service on the Cloud (see Section 2.5). Even though they are Cloud services, most of them are still backed with the same database systems as a local one, which means they use the same SQL dialect as the local database system does. However, since service consumer has no or little control over the underlying system of the database, that restriction may limit the functionality of the database system, which in turn limit the SQL syntax that we can use. Or simply the service provider poses limitation for some other reasons, such as security or availability. For example, Google Cloud SQL restricts the use of user defined functions or statement "`SELECT ...INTO OUTFILE/DUMPFILE`" [goo]. Therefore, it should be treated as a new SQL dialect, distinguished from its local counterpart. However, for those that have no documented limitation or differences, such as Amazon RDS, we'll treat them as the same dialect as a local database system.

Database in a Cloud infrastructure is an IaaS approach, where we deploy our own database system, as well as the underlying operation system, etc., on a Cloud infrastructure. Since IaaS abstracts only the underlying infrastructure, we have almost full control over the system built upon it, including the operating system, file system, and the database system. This also allows us to install any type of database system on the Cloud. Such setup simulates a local database system, which means we can use the same SQL dialect as we used on a local system.

[1] PostgreSQL Release Announcement: `http://www.postgresql.org/about/news/1481/`

4.3. SQL Statement Transformation

In this section, we will detail the system specification regarding the transformation functionality of SQL statements. In order to transform SQL statements across multiple dialects used by different database vendors, the transformation process is grouped according to the source dialect, each with a corresponding parser. The parsing result should be a dialect-independent representation of the SQL statement containing all its semantic context, which can later be transformed back into the text-form statement based on the target dialect. Thus, the following two functionalities must be offered by our components: *SQL Statement Parsing* and *SQL Statement Transforming*.

4.3.1. SQL Statement Parsing (FR1)

SQL is a structured language, as indicated by its name. And the main body of SQL language consists of *SQL statements*, each of which requests a specific action from the DBMS, such as retrieving data or creating new table [GW02]. The construction of an SQL statement follows a strict grammar specified by its standard, and various database vendors. Therefore a parser can be created according to the grammar, to decompose the statement and construct it into a parse tree, as shown in Figure 4.3. With the tree, the system can then easily identify every aspects of a statement, unlike the plain text formed sentence. Importantly, the tree can then be used to form statement in any dialects, with all the information of the original statement.

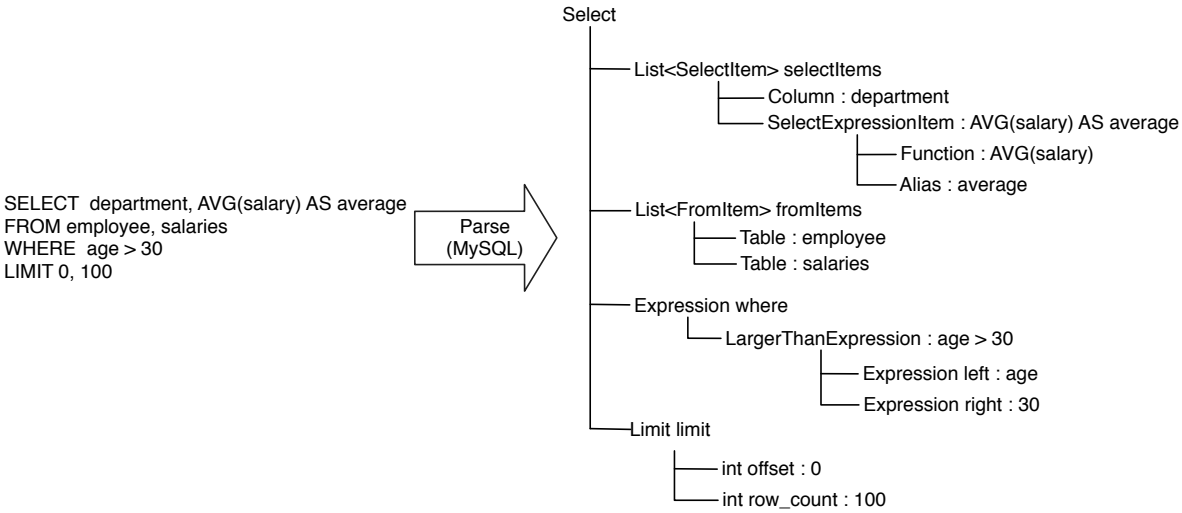

Figure 4.3.: Parsing of a SELECT Statement Into a Parse Tree

4.3.2. SQL Statement Transforming (FR2)

With the parse tree we acquired from the parser, we can identify every intention of the statement in its source dialect. With that in mind, we can then find the corresponding representation in another dialect, that has the exact same meaning of the original statement.

For example as shown in Figure 4.4, the node `Limit` of the tree indicates that the original statements has an intention to constraint the number of rows returned by the statement (the returned result should start from the beginning with an offset of 0, and has maximum 100 rows). In different dialect, this clause is presented differently, though with the same meaning. For example, in PostgreSQL 9.2, it's written as "`LIMIT 100 OFFSET 0`". And in Oracle 12c, it's "`OFFSET 0 ROW FETCH NEXT 100 ROWS ONLY`". And these texts can be easily formed with the two parameters we have in the tree: `offset` and `row_count`.

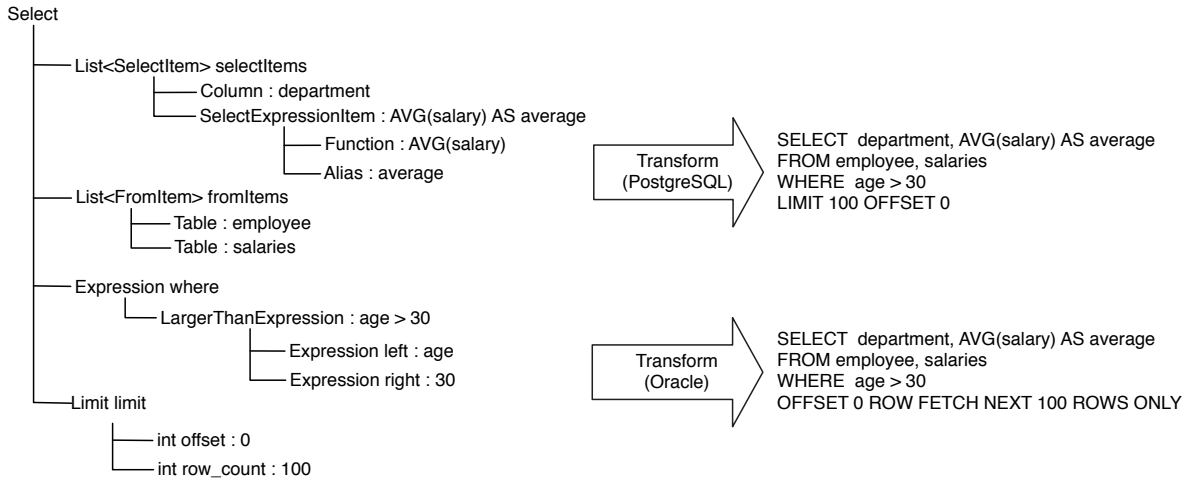

Figure 4.4.: Transforming a Parse Tree Into SQL Statements of Different Dialects

4.4. SQL Response Transformation

Same as the SQL statement, each database system may have their own syntax of the response. Fortunately, as stated in Section 4.1.4, CDASMix uses JDBC driver to connect to the target database. It provides us an unified API to deal with the response from the database.

Generally, there are two types of response from executing an SQL statement. When the statement is a data query, e.g. a SELECT statement, the response will be a table of values satisfying the query. And when the statement is a data modification or definition statement, e.g. an INSERT statement, JDBC will return the number of affected rows for the statement. In addition, JDBC can retrieve the generated keys of the executed statement when available with a specific flag. As we can see, the response we receive will have the same structure independent from the database system. Thus, when considering the compatibility of the response with the source database, we need only to take care of the data and their data types stored in the response, which can be accessed with the JDBC API: `java.sql.ResultSet`.

Camel JDBC Component is responsible for retrieving the result set and marshal it into a NM. At this point, data type is irrelevant. Thus, the dynamic data access method of `java.sql.ResultSet` is used: `ResultSet.getObject`. We know that the data retrieved from database are mapped to JDBC types expressed with class `java.sql.Types`; and for each type

there is a corresponding Java class[2] to store the data, which are all subclasses of the `Object` class.

In the proxy bundle, the result set data is sent back to the client via TCP connection with the database native communication protocol. For different data types the protocol requires different data formats, therefore the data need to be converted back to their original form in Java classes. The type information of the data is kept in the `ResultSetMetadata` specified with `java.sql.Types` class. JDBC defined a map between the JDBC type and Java type [jdb], which the proxy bundle used to convert the objects retrieved from the *Camel JDBC component*, to their corresponding Java classes.

As we can see, JDBC already takes care of the data conversion from native database format to Java class, and proxy bundle took care of the conversion from Java class to native database format. The only dilemma left is that whether the data received from the target database is mapped to the same Java class as the source database would expect. This is however guaranteed by the migration process which dealt with the compatibility of the data between source and target database [Bac12]. Since the data in the source database is migratable to target database, we can assume that they can be mapped to the same Java classes. The same applies to the DDL statements executed through CDASMix, which will be transformed to manage the data type compatibility. However such assumption requires an extensive validation, which we will provide in Chapter 7. In conclusion, SQL response transformation is not necessary in CDASMix, with the limitation that the source and target data should be mapped to the same Java class.

4.5. Use Cases

In this section, we provide four use cases regarding SQL transformation in CDASMix. The first two are related to the functionality of SQL transformation, whose actor is the system itself (more specifically the Proxy Component). In addition, we detail two more use cases for extending the existing SQL transformation functionality, to provide support for additional SQL dialects as both source and target dialect.

[2]For primitive types, such as `int`, `double`, their wrapper class is used, namely `java.lang.Integer` and `java.lang.Double`

Name	**Parse SQL Statement**
Goal	To parse an SQL statement in plain text form into a parse tree
Actor	Proxy Component
Pre-Condition	An SQL statement in text form and its dialect is presented, as well as a corresponding SQL parser.
Post-Condition	The SQL statement is parsed into a parse tree
Post-Condition in Special Case	The SQL statement is not successfully parsed into a parse tree
Normal Case	1. The proxy component looks up the transformer service associated with the dialect of the SQL statement. 2. The proxy component uses the service's parser to parse the statement into a parse tree.
Special Cases	1. There is no transformer service associated with the dialect of the SQL statement. a) The system informs the application with an error message. 2a. There is a syntax error in the statement. a) The system informs the application with an error message. 2b. The statement's syntax is not supported by the parser. a) The system informs the application with an error message.

Table 4.1.: Description of Use Case *Parse SQL Statement*.

Name	**Transform SQL Statement**
Goal	To transform a statement's parse tree into a statement with the specified target dialect
Actor	Proxy Component
Pre-Condition	The SQL statement in source dialect is successfully parsed into a parse tree; source and target dialect are different; and the transformation for the target dialect is supported in the transformer service.
Post-Condition	An SQL statement in the specified target dialect is returned.
Post-Condition in Special Case	The SQL statement is not transformed into the specified target dialect.
Normal Case	The proxy component uses the previously acquired parse tree to transform the statement to the target dialect.
Special Cases	1a. The syntax of the original statement in source dialect cannot be transformed into the target dialect. a) The system informs the application with an error message. 1b. The transformation is not implemented for the target dialect. a) The system informs the application with an error message.

Table 4.2.: Description of Use Case *Transform SQL Statement*.

Name	**Add Transformation to New Target Dialect to an Existing Transformer**
Goal	To add transformation support for new target dialect in the existing transformer
Actor	System Administrator
Pre-Condition	The system administrator has access to the source code of the existing transformer, and the access permission to the OSGi container
Post-Condition	The transformer service is updated to support the new target dialect
Post-Condition in Special Case	The new target dialect stays unsupported in the transformer service
Normal Case	1. Update the source code of the transformer service to add support for the new target dialect, and package it into an OSGi bundle with an increased version number. 2. Perform an update with the newly created bundle using the OSGi console.
Special Cases	The bundle fails to be updated. a) OSGi console prompts an error message and aborts the update.

Table 4.3.: Description of Use Case *Add Transformation to New Target Dialect to an Existing Transformer*.

Name	**Add New SQL Parser and Transformer for New Source Dialect**
Goal	To add new SQL parser and transformer for new source dialect
Actor	System Administrator
Pre-Condition	The system administrator has access permission to the OSGi container
Post-Condition	The new transformer service is added to the OSGi container to support new source dialect
Post-Condition in Special Case	The new transformer service is not deployed to support the new source dialect
Normal Case	1. Develop the SQL parser and transformation for the new source dialect. 2. Declare it as a transformer OSGi service and package it in an OSGi bundle. 3. Deploy the bundle into the OSGi container.
Special Cases	The bundle fails to be deployed. a) OSGi console prompts an error message and aborts the deployment.

Table 4.4.: Description of Use Case *Add New SQL Parser and Transformer for New Source Dialect*.

4.6. Non-Functional Requirements

In this section, we will specify a list of non-functional requirements that we take into consideration during the design and implementation of our system. It spreads over the development, deployment, and adoption of the components we develop.

4.6.1. Extensibility (NFR1)

Since there exists a wide range of SQL dialects, both as source and target dialect, we need to make the components we develop easily extensible for adding additional support. As for source dialect, additional parser and transformation bundles can be added to the system without extra alteration of existing components or restarting the whole system. And when adding support for new target dialects, changes applied to the existing components should be minimum, and independent from the already running system.

4.6.2. Integratability (NFR2)

SQL transformation is developed as an additional functionality. Thus it should be able to smoothly integrate into the existing system (CDASMix), without affecting its basic functions and requirements. In addition, plugging out or failure of the transformation should be as undisturbed as the integration.

4.6.3. Performance (NFR3)

The additional processing of SQL statement can be time and memory consuming, especially when parsing statements. This will have a definite impact on the system performance, in terms of request throughput, and system resource consumption. We can increase the effectiveness of the parser to speed up the process. The position we place the transformation functionality also affects the performance due to different internal communication methods.

4.6.4. Scalability (NFR4)

In order to provide support for various source dialects, a separate bundle for each source dialect needs to be deployed. Such deployment model will restrict the scalability of the system due to the limit system resource in an OSGi container. Therefore we should consider to limit the service instantiation and components loading based on the service we actually use to avoid unnecessary resource consumption.

4.6.5. Maintainability and Documentation (NFR5)

Due to the extensibility (NFR1), a continuous development of transformation bundles will take place to add further support. Therefore, a detailed description of developing and extending the bundles should be documented to ease the process. This also means fully commented source code, and extensive logging in order to support the debugging.

5. Design

In this chapter, we present the architectural designs we took into consideration to integrate transformation functionality, and the final decision we made between two variants. Furthermore we'll discuss how the transformation process is realized as an OSGi service, and how the realization fulfills the functional and non-functional requirements we proposed in Chapter 4.

5.1. System Architecture

As we discussed in Section 4.1.2, the database server proxy bundle provided by CDASMix gathered all the information we need to perform an SQL transformation: the statements, and their source and target dialects; then wrap them into a normalized message before passing it to the NMR.

Before further discussion, we need to first take a look at the inter-bundle communication in CDASMix. We know CDASMix is based on Apache ServiceMix 4.3, which complies with both OSGi and JBI specification (see Section 2.9). There are multiple ways to communicate between the bundles. Communication through NMR is a message-based communication where bundles can exchange messages, both synchronously and asynchronously. In our case, synchronous message exchange is used by the proxy to ensure the the correspondence of request and response, as well as message order [Sáe13a]. The reason NMR is used, is that it spans both the OSGi container and the JBI container, thus enables communication between OSGi bundle and JBI bundle.

On the other hand, OSGi service is the natural way to communicate within an OSGi container. It is a publish, find and bind model, which provide normal Java objects as services registered with a registry [OSG11]. Therefore the the message exchange with OSGi service is achieved by invoking a Java method with parameters and receiving a returned value. It's a straight forward synchronous communication, however only available in the OSGi container.

Therefore we proposed two approaches based on different prospective on modularization and performance, using the above mentioned communication methods.

5.1.1. First Approach

In the earlier development, we considered the design presented in Figure 5.1. It modularizes the transformation functionality into a separate OSGi bundle, the *SQLTransformer*, which exposes itself as an endpoint on the NMR. Database server proxies will relay their NMs to

the *SQLTransformer* endpoint when the source and target data source are different. As a side note, the Uniform Resource Identifier (URI) of the tenant-aware Camel endpoint where the NM should be sent to (see Figure 2.3) is stored in the property of the NM.

In the *SQLTransformer* bundle, the received NM will be unmarshaled, to examine the source and target dialects of the queries stored in its attachment. Based on the source dialect, it will look up the corresponding transformation service registered in the OSGi service registry, which we will introduce later. With the acquired service, it can then transform the queries to their target dialect. With all queries transformed, they will be again marshaled into the NM and forwarded to the tenant-aware Camel endpoint via NMR.

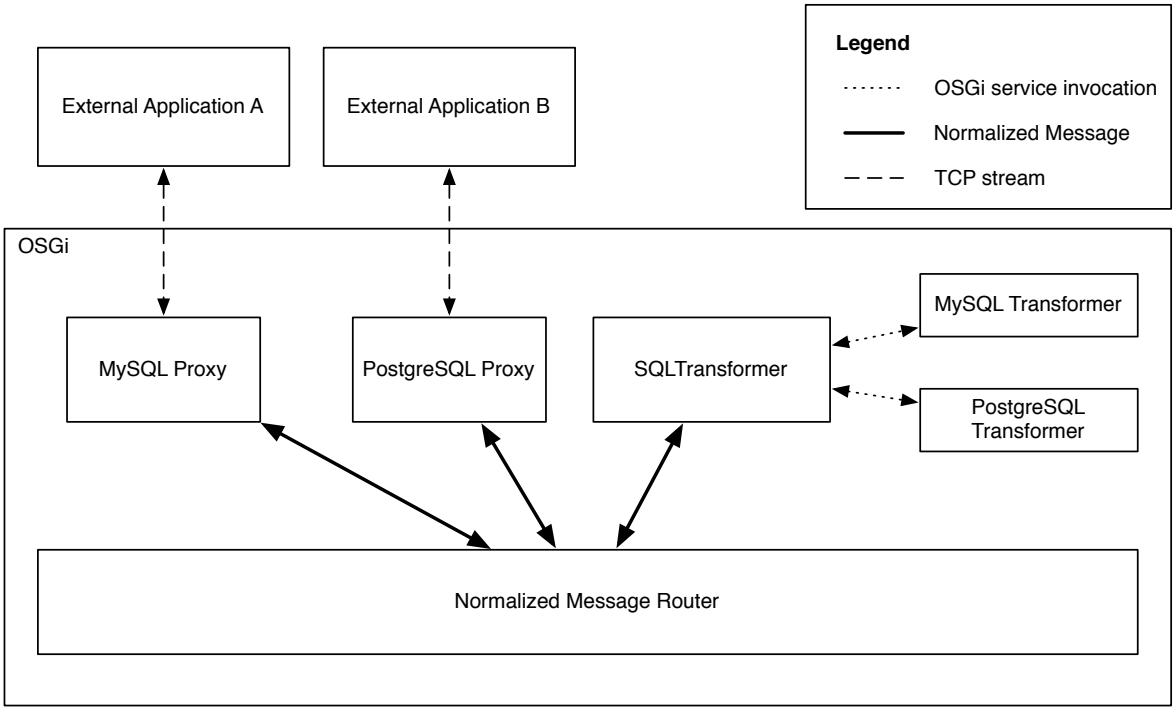

Figure 5.1.: First Approach - Transformer as Separate OSGi Bundle

5.1.2. Second Approach

In the second approach, we will transform the SQL statements with the transformation service directly in the proxy bundle. When the proxy detects that the source and the target dialect of the statement disagree, it will then look up a corresponding transformation service, and use it to transform the statement before marshaling it into the NM. Otherwise the statement will proceed without transformation.

As we specified in Section 4.1.2, for each type of database client, a server proxy bundle is deployed to handle the communication (MySQL Proxy for MySQL applications and PostgreSQL Proxy for PostgreSQL applications), hence one proxy corresponds to one type of database system. Therefore the statements received at a proxy always belong to the dialect of that

particular database system. Depending on the version of source dialect, a proxy bundle will require only a particular set of transformer services. Thus, a different architecture is then designed and chosen to use for our final prototype, as shown in Figure 5.2.

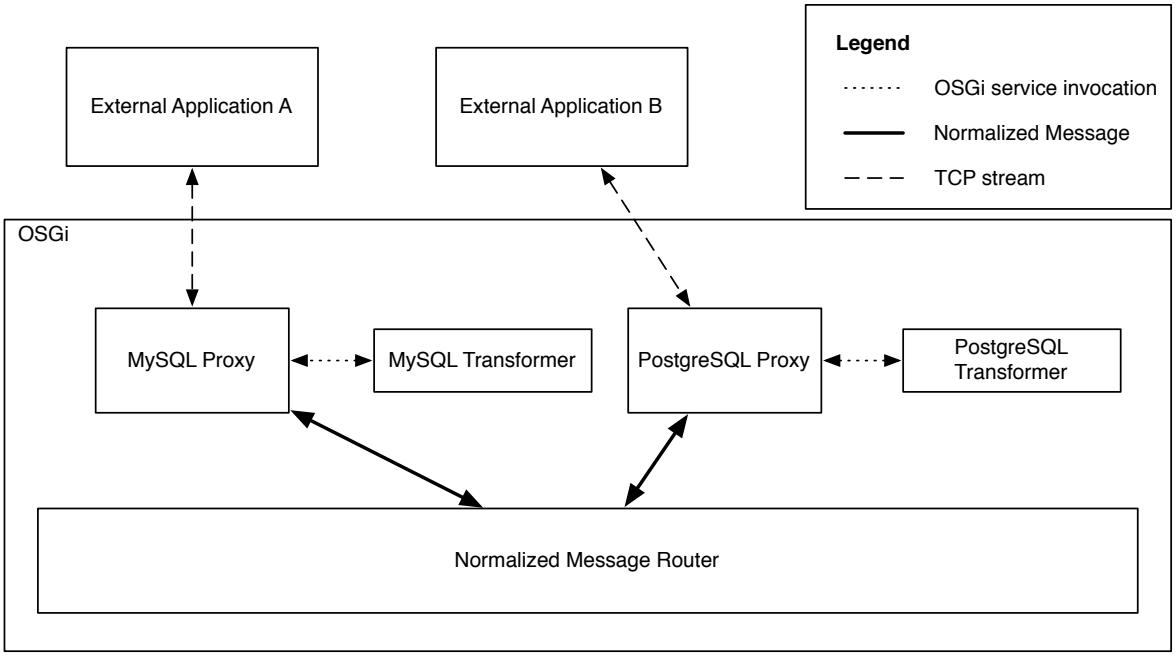

Figure 5.2.: Second Approach - Direct Transformation From Proxy Bundle With Transformer Services

Another advantage of this architecture over the previous one, is that the proxy will transform the SQL statements it received prior to the marshaling of NM. As stated previously, the *SQLTransformer* bundle needs to unmarshal and then again marshal the NM in order to transform the statements stored in the attachment. Such process is then eliminated using the second design, which will benefit the overall performance (request throughput) we required as stated in NFR3.

Furthermore, in this design, the proxy bundle is directly responsible to look up the transformation service. Therefore it knows beforehand the availability of the services and can act accordingly. In the first approach, the proxy bundle will send the request to the *SQLTransformer* when transformation is needed, without knowing if there is actually a service available to transform the statements.

In addition, for future consideration, the parser used in the transformation service could potentially further improve the effectiveness and stability of the proxy. As pointed out in Section 4.1.2, the statements are analyzed in the proxy bundle to retrieve its query type and related schema objects. In current implementation of CDASMix by Goméz Sáez [Sáe13a], direct string matching is used to examine the statements. However, the transformation service can be easily extended to use the parse tree we acquired (see Figure 4.3) to identify the information in an SQL statement with a much higher robustness and effectiveness.

5.2. SQL Transformation Service

The actual SQL transformation functionality we introduced in Section 4.3 (FR1 and FR2), is designed as an OSGi service, to take advantage of the OSGi container in CDASMix. The nature of OSGi framework fulfills our requirement NFR1, as the OSGi service packaged in an OSGi bundle can be easily installed, upgraded, and uninstalled as specified in the Life Cycle Layer of OSGi specification (see Section 2.8).

OSGi service as defined in the Service Layer of OSGi specification [OSG11], follows a publish, find and bind service model, which register normal Java objects under one or more Java interfaces with the service registry. The Java interfaces which the service objects implement, contribute as service description that identifies the input parameters and their types, as well as the return values and exceptions. A service object may specify additional service properties when registered.

There are various ways to define an OSGi service as specified in the OSGi Service Compendium [OSG12]. According to NFR4 we require minimum resource consumption, as well as a simplified programming model to satisfy NFR1. Based on these requirements, we narrowed our choices to the *service component model* specified in the *Declarative Services Specification* [OSG12].

This model simplifies the task of authoring OSGi services by performing the work of registering the service and handling service dependencies. It minimizes the amount of code we have to write, which eases the extensibility (NFR1) of our system to develop additional bundles. It also allows components to be loaded only when they are needed, as shown in Figure 5.3. The service bundle is only loaded when the service it provides is requested, as well as the instantiation of service object. As a result, the system start up time is reduced, as well as the memory footprint, which fulfills the requirement NFR4.

As we concluded in Section 4.3, the transformation process is grouped according to the source dialect. That is to say, for each source dialect we will deploy a corresponding OSGi service component. Each service component implements the same service as defined in Listing 5.1. It has only one method which transforms a statement from the source dialect to the specified target dialect, and defines a set of exceptions that can occur.

```
1 public interface SQLTransformer {
2     String transform(String original, Dialect target) throws NotImplementedException,
          UntransformableException, SQLParseException;
3 }
```

Listing 5.1: SQL Transformer Service Definition

In order to distinguish the deployed service components which support different source dialects, we need to define an extra service property for the service component to identify the source dialect it supports. The service property is registered along with service component in the OSGi service registry when the bundle is deployed. More importantly, a particular

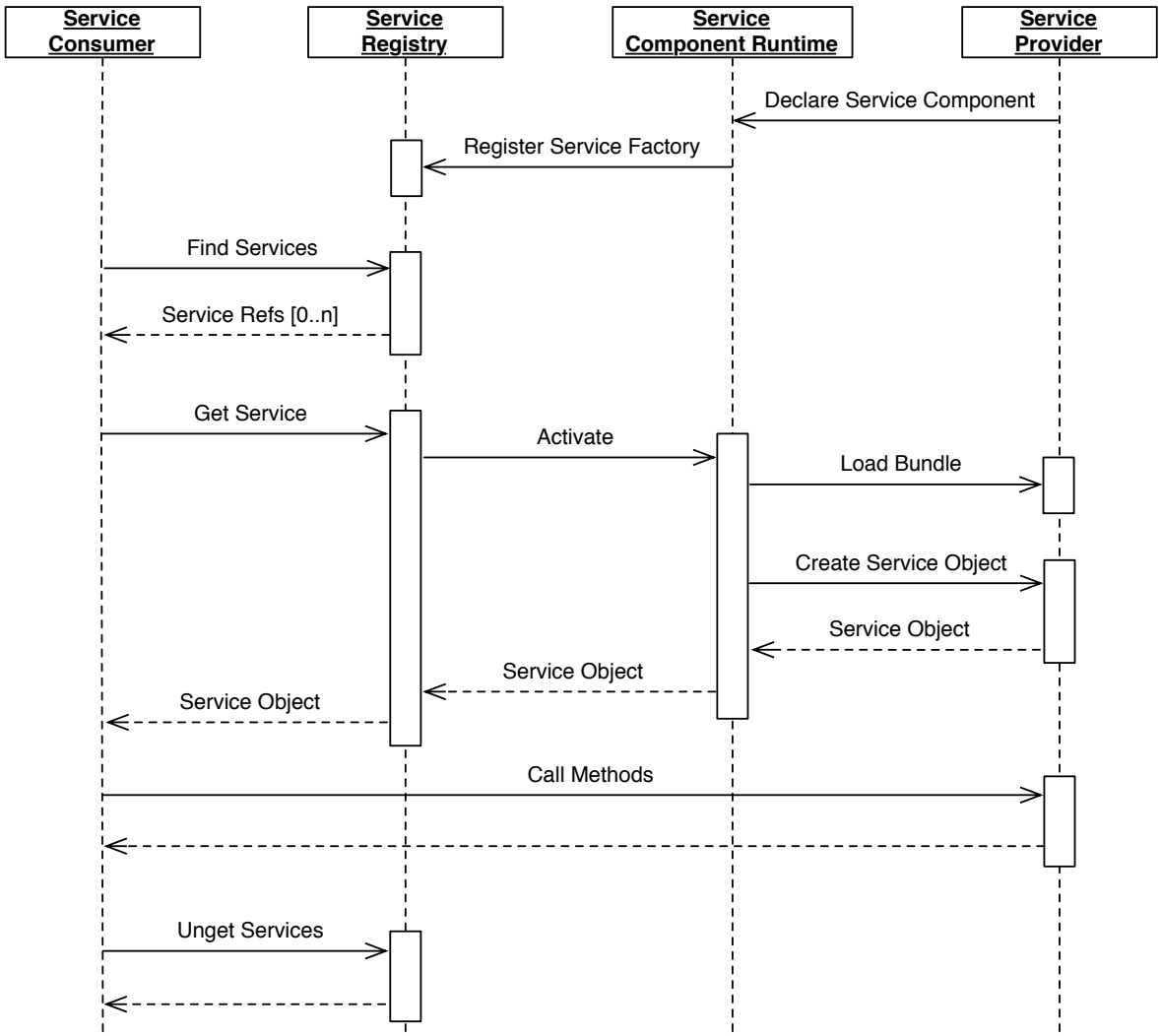

Figure 5.3.: The Life Cycle of Declarative Service of OSGi [Bar]

service can then be looked up by specifying a filter which includes the source dialect property and the desired value.

6. Implementation

In this chapter, we will detail the implementation of the SQL transformation functionality, as well as its integration with CDASMix. The transformation functionality is realized as OSGi services, packaged in individual OSGi bundles. The integration requires extension to the existing proxy bundles in CDASMix. The majority of the implementation is realized with Java language, while certain parts are written in XML in cope with the OSGi specification.

6.1. SQL Parser and Transformation

In Section 2.3, we introduced *JavaCC*, the parser generator, and the fact that it can generate a parser from the grammar file (Listing 2.2) to parse SQL statement into a parse tree. Listing 6.1 shows us with pseudo code how the generated parser parse a statement.

```
1  public class MySQLParser {
2      ...
3      final public Statement Statement() {
4          Statement stmt;
5          switch(first_word){
6              case R_SELECT:
7                  stmt = Select();
8              ...
9          }
10         return stmt;
11     }
12
13     final public Select Select() {
14         consume_token(R_SELECT);
15         Select select = new Select();
16         List selectItems = null;
17         FromItem fromItem = null;
18         ...
19         selectItems = SelectItemsList();
20         if(R_FROM){
21             consume_token(R_FROM);
22             fromItem = FromItem();
23         }
24         ...
25         return select;
26     }
27
28     final public FromItem FromItem(){
29          if(match_Table){
```

```
30              return Table();
31          }
32      }
33      ...
34  }
```

Listing 6.1: Pseudo Code of MySQL Parser

As we can see, what the parser does is building a parse tree based on the token it reads from the input. It will build from the root node (`Select`) and then to the left-most leaf node until it reaches the terminal node before it moves on to the next leaf node. Such order is referred as depth-first pre-order.

In addition to the JavaCC grammar file, we need to implement the parse tree as Java classes, as well as the transformation functionality. An example is shown in Listing 6.2, which illustrates the `Select` class in MySQL dialect. For demonstration reason, the class is simplified from the actual implementation to fit the example we used in Figure 4.3.

```
1   public class Select implements Statement {
2       private List<SelectItem> selectItems;
3       private List<FromItem> fromItems;
4       private Expression where;
5       private Limit limit;
6       ...
7       @Override
8       public String toString() {
9           String sql = "SELECT␣";
10          sql += StringUtil.listToString(selectItems);
11          sql += "␣FROM␣" + StringUtil.listToString(fromItems);
12          sql += (where != null) ? "␣WHERE␣" + where : "";
13          sql += (limit != null) ? limit : "";
14          return sql + ";";
15      }
16      @Override
17      public String transform(Dialect target) throws UntransformableException,
            NotImplementedException {
18          switch (target) {
19              case PostgreSQL:
20                  String sql = "SELECT␣";
21                  sql += StringUtil.transformListToString(selectItems, target);
22                  sql += "␣FROM␣" + StringUtil.transformListToString(fromItems, target);
23                  sql += (where != null) ? "␣WHERE␣" + where.transform(target) : "";
24                  sql += (limit != null) ? limit.transform(target) : "";
25                  return sql + ";";
26              ...
27          }
28      }
29  }
```

Listing 6.2: Select Class and its Transform Method

Worthy of note, in Line 2-4, `SelectItem`, `FromItem`, and `Expression` are Java interfaces we use as types to reference specific SQL context. Since different SQL syntax can appear at various position of an SQL statement in different context, Java interface provides an alternative to avoid multiple inheritance which is not supported in Java. For example, `SubSelect` implements `FromItem`, since `SubSelect` can appear after `FROM` indicating the place from which to retrieve data. `SubSelect` also implements `Expression`, as it can appear in an SQL expression, such as an `InExpression`.

The `toString` method[1], that every Java class inherits, is used to reconstruct the class into a statement in its original dialect. It may differ from the statement that is passed as input of the parser, but only on the insignificant parts, such as case of keywords, meaningless empty spaces or brackets, etc. This method is useful to output an SQL statement into a standard format.

The important method for us is the `transform` method which will be implemented by every class of the parse tree as well. It takes only one parameter, the target dialect, to which the statement will be transformed. In Listing 6.2, Line 20-25, we can see that there are no additional changes required on the `Select` statement level; only the components in the statement need to be transformed individually. And Listing 6.3 shows us how the `Limit` class is transformed. The `Statement` interface in Listing 6.2 is a subclass of the `Transformable` interface. In such way, when the `transform` method (as well as the `toString` method) is called on the `Statement`, a recursive method calling will carry out along the tree which finally complete a depth-first pre-order tree traversal and return a complete transformed SQL statement. It's in the exact same order as how the tree is build which we introduced previously.

```
1  public class Limit implements Transformable{
2      ...
3      @Override
4      public String toString(){
5          return "␣LIMIT␣" + offset + ",␣" + row_count;
6      }
7      @Override
8      public String transform(Dialect target){
9          switch(target){
10             case PostgreSQL:
11                 return "␣LIMIT␣" + row_count + "␣OFFSET␣" + offset;
12                 ...
13         }
14     }
15 }
```

Listing 6.3: Limit Class and its Transform Method

[1] In Listing 6.2, Line 12-13, `where` and `limit` are implicitly cast to `String`, where their `toString` method will be implicitly called. See Listing 6.3, Line 3-6.

6.2. Transformation Service Implementation

As the next step, we need to expose the transformation functionality as an OSGi declarative service, as discussed in previous sections. Fortunately, the SCR as defined in OSGi specification [OSG12] will take care of the service registration, bundle loading, and service object instantiation of a service component (see Figure 5.3). All we need to do is declare a service component. This is achieved with the SCR descriptor, as shown in Listing 6.4. Such an XML file may be placed anywhere within the bundle, as long as its path is listed in the bundle's manifest header `Service-Component` [Thea].

```
1  -- OSGI-INF/MySQLTransformer.xml --
2
3  <?xml version="1.0" encoding="UTF-8"?>
4  <components xmlns:scr="http://www.osgi.org/xmlns/scr/v1.0.0">
5      <component immediate="false" name="MySQLTransformer">
6          <implementation class="iaas.unistuttgart.de.sqltransformer.MySQLTransformer"/>
7          <service>
8              <provide interface="iaas.unistuttgart.de.sqltransformer.api.SQLTransformer"/>
9          </service>
10         <property name="source_dialect" value="MySQL"/>
11         <property name="service.pid" value="MySQLTransformer"/>
12     </component>
13 </components>
14
15
16 -- META-INF/MANIFEST.MF --
17
18 ...
19 Service-Component: OSGI-INF/MySQLTransformer.xml
```

Listing 6.4: OSGi Declarative Service Component Descriptor

It has several noteworthy settings regarding the service component. The `component` element defines one service component with an unique `name`. The `immediate` attribute defines whether the component is to be instantiated immediately (`true`) or on-demand (`false`). The `implementation` element specifies the full qualified name of the class that implements the service(s), while the service(s) is defined under the `service` element. A service component can implement multiple services. Furthermore, we can characterize the service component with additional properties. The property is essential for us, as we depend on them to distinguish different transformation service components based on their source dialect.

In addition to the straightforward declaration with component descriptor, Apache Felix provides a Maven SCR plugin[2] to help generating the SCR descriptor and adding the `Service-Component` header to the bundle manifest automatically. Since we are already building our components with Maven, such plugin can help ease the process, as it can keep the descriptor and the code in sync during the development.

[2] Apache Felix Maven SCR plugin: `http://felix.apache.org/documentation/subprojects/apache-felix-maven-scr-plugin.html`

Apache Felix Maven SCR plugin uses Java annotations to define the service component, as shown in Listing 6.5. It can be quite easy to see the resemblance with what we defined in Listing 6.4. Actually, the descriptor in Listing 6.4 is generated by the Maven SCR plugin with our definition in Listing 6.5.

```
1  @Component(name = "MySQLTransformer", immediate = false)
2  @Service(value = SQLTransformer.class)
3  @Property(name = SQLTransformer.SOURCE_DIALECT_PROP, value = "MySQL")
4  public class MySQL implements SQLTransformer {
5
6      @Override
7      public String transform(String original, Dialect target) throws NotImplementedException,
           UntransformableException, SQLParseException
8      {
9          if(target.equals(getSourceDialect())){
10             return original;
11         }
12         MySQLParser parser = new MySQLParser(new StringReader(original));
13         String transformed = null;
14         try {
15             Statement stmt = parser.Statement();
16             transformed = stmt.transform(target);
17         } catch (ParseException e) {
18             throw new SQLParseException(e.getMessage());
19         }
20         return transformed;
21     }
22     ....
23 }
```

Listing 6.5: OSGi Declarative Service Implementation with Felix SCR annotations

The actual service implementation is quite simple, as the transformation functionality is handle by the parser and the parse tree which we introduced in Section 6.1. We only need to pass the original statement to the parser (`MySQLParser` in Line 12) that we generated with *JavaCC*, and parse the statement into a parse tree (Line 15) as seen in Listing 6.2. This will fulfill our functional requirement FR1. Then we can transform the parse tree into a statement of the target dialect (Line 16), to fulfil requirement FR2.

6.3. Transformation Service Lookup and Consumption

In order to use a service object and call its methods, a bundle must first obtain a `ServiceReference` object, which references to a registered service. It encapsulates the properties and other meta-information about the service object it represents, but not the service object itself. This is to avoid complex service dependencies between bundles when a bundle needs to know about a service but does not require the service object itself [OSG11].

The `BundleContext` interface of the OSGi bundle provides several methods to locate the `ServiceReference` object. In Listing 6.6, we use `getServiceReferences(String clazz, String filter)` method, which returns an array of `ServiceReference` objects whose services implemented and were registered under the `clazz` service interface, and satisfy the search `filter`.

```
1  BundleContext context;
2  ...
3  public String transform(String statement, Dialect source, Dialect target) throws
       NotImplementedException, UntransformableException, SQLParseException,
       TransformerNotFoundException {
4      ServiceReference[] refs = null;
5      SQLTransformer transformer = null;
6      String filter = "(" + SQLTransformer.SOURCE_DIALECT_PROP + "=" + source.name() + ")";
7      refs = context.getServiceReferences(SQLTransformer.class.getName(), filter);
8      if (refs != null && refs.length > 0) {
9          transformer = (SQLTransformer) context.getService(refs[0]);
10     } else {
11         throw new TransformerNotFoundException(target);
12     }
13     String transformed = transformer.transform(statement, target);
14     return transformed;
15 }
```

Listing 6.6: OSGi Service Lookup With Filter

It's obvious that we are looking for service objects that implement the service (interface) shown in Listing 5.1. And we can specify the source dialect in the filter (Line 6) to match the service property we defined in Section 6.2. The filter syntax used by the OSGi service registry is based upon the string representation of Lightweight Directory Access Protocol (LDAP) search filters [OSG11].

As we can see, there could be more than one `ServiceReference` objects returned, since multiple services can be registered with the same interface and properties. We will always choose the first element, which has the highest ranking order [OSG11]. With the acquired `ServiceReference` object, we can then retrieve the service object (Line 9), on which the service method can be invoked (Line 13).

Since OSGi is a highly dynamic environment, service objects can be registered and unregistered at any given time, therefore it's useless to store the service object as it may become unavailable. A flexible strategy is that we look up the service object every time when we need it.

7. Validation and Evaluation

In this chapter, we will validate the prototype we provided in Chapter 5 and Chapter 6, and evaluate its performance, to ensure that the system fulfills the requirements we proposed in Section 4.6. For the validation, we provide a two step validation to separately verify the correctness of the standalone transformer, and its integrity when the system operates with Cloud database services. The performance evaluation of transformation is done as an individual module to isolate the performance variables, so that we can have a clearer view how the transformation works.

7.1. Validation of SQL Parser and Transformation

The SQL parser and transformer work in a static way that we can precisely predict the result for a provided SQL statement. Therefore, to validate the correctness of the SQL parser and transformer, we use an extensive collection of test cases based on JUnit test framework[1].

```
1  public SelectTest extends TestCase{
2      ...
3      public void test() throws JSQLParserException {
4          String statement = "SELECT␣*␣FROM␣mytable␣WHERE␣col␣=␣9␣LIMIT␣3,␣10;";
5          String transformed = "SELECT␣*␣FROM␣mytable␣WHERE␣col␣=␣9␣LIMIT␣10␣OFFSET␣3";
6          Select select = (Select) parser.parse(new StringReader(statement));
7
8          assertEquals(3, ((PlainSelect) select.getSelectBody()).getLimit().getOffset());
9          assertEqauls(statement, select.toString());
10         assertEquals(transformed, select.transform(Dialect.PostgreSQL));
11         ...
12     }
13     ...
14 }
```

Listing 7.1: JUnit Test Case Example

As shown in Listing 7.1, we used a test statement on Line 4, from which we can easily recognize its structure and predict the outcome parse tree. Hereafter, we can then verify each part of the parse tree with our expected value as in Line 8, using the `assertEquals` method from class `junit.framework.TestCase`, which asserts that the expected value and the actual value are equal. Another compact way to verify the parser is to check the `toString` method as in Line 9. We introduced in Chapter 6 that the `toString` method reconstructs the

[1] JUit Test Framework: `http://www.junit.org`

parse tree into a standard SQL statement. When we provide the statement (Line 4) in the standard format, we can expect that the statement will equal to the reconstructed one. In this way, every aspect of the parse tree is verified in one single line of code. Validating the transformation is similar as in Line 10, where we verify the transformed result against our expected value specified at Line 5.

To run the unit tests, we take advantage of the Maven Surefile Plugin[2], which runs the unit tests and generates reports when we build our bundle. This guarantees the integrity of the bundle of the corresponding transformer service.

7.2. Validation with CDASMix and Cloud Database Services

In this section, we verify the integration of SQL transformation service with CDASMix, and our final goal of achieving to access various Cloud database services with transformed SQL statements. As the proper functionality of Cloud data access is provided in [Sáe13a], we only focus on validating the correctness of transformed SQL statements and the results we get from the Cloud database.

7.2.1. Deployment and Initialization

We follow the same setup of CDASMix as provided in [Sáe13a]. There is an additional document, *Manual for the CDASMix Initialization* [Sáe13b], which helps us to setup CDASMix. Refer to those documents for detailed information on setup and initialization of CDASMix, which we won't discuss here. In addition, we need to deploy our own bundle to the ServiceMix-mt (see Section 2.9), to integrate the transformation service. That includes:

- CDASMix MySQL Proxy 2.0: an upgraded CDASMix proxy bundle with the service look up functionality we introduced in Section 6.3.
- MySQL Transformer 1.0: the transformation service implementation for MySQL source dialect which is defined as an declarative service.
- Apache Felix SCR Bundle 1.8: an implementation of the Declarative Service specification, which handles the service registration, bundle loading and service instantiation of declarative services (see Figure 5.3).
- SQL Transformation API 1.0: the API bundle which includes the service interface (see Listing 5.1), SQL dialect definitions, and exception classes shared between the proxy bundle and transformer bundle.

Furthermore, we have several Cloud databases set up for test:

[2]Maven Surefire Plugin: `http://maven.apache.org/surefire/maven-surefire-plugin/`

- Amazon RDS db.t1.micro DB instance with MySQL 5.5 database system. This is used as a comparison to ensure when target and source databases are of the same type, no SQL transformation will take place. It can also verify that the syntax of the original statement is correct, as well as the result set of queries.
- PostgreSQL 9.2 on an Ubuntu 10.04.4 VM image in the FlexiScale Cloud infrastructure[3].
- Amazon RDS db.t1.micro DB instance with Oracle 11.2 database system.

7.2.2. Validation

To stimulate the scenario where database application is built upon one database system, while it actually access through CDASMix with a target database of another type, we need to register a pair of source and target data source with the CDASMix. We can register existing Cloud data source with a source data source through a Web service interface [Sáe13a]. For the detail information about registering data sources, please refer to document [Sáe13a]. In our case, we registered three pairs of data sources as in Table 7.1. Since we have only the MySQL proxy for testing, we use only MySQL database as source data source. And MySQL 5.1.3 and MySQL 5.5 use the same SQL syntax due to its backward compatibility.

Data Source Name	Location ID	Data Source Type
mysqldb	source	mysql-database-table-5.1.3
	target	mysql-database-table-5.5
postgresqldb	source	mysql-database-table-5.1.3
	target	postgresql-database-table-9.2
oracledb	source	mysql-database-table-5.1.3
	target	oraclesql-database-table-11.2

Table 7.1.: Tenant Data Source Registration

In order to thoroughly validate the transformer, we need to build test cases which cover every SQL statements that are supported in the transformation service, and such statements should be transformable to all target dialects we provided. Therefore, we make up our own database schema and statements rather than using standard benchmark tools like TPC-H [Tra13], though we still used as a reference. Since we use only MySQL as source data source, the source dialect we use will also be MySQL. In Table 7.2, the input row is the test statement we write using MySQL dialect syntax. The following rows (MySQL, Oracle, PostgreSQL) are the statements that are actually executed on the corresponding target database. The tick mark indicates the correctness of the execution of that statement, that includes successful execution

[3]FlexiScale Cloud Computing Platform: `http://www.flexiscale.com`

of the statements, as well as the correct result set of the queries. An X indicates otherwise. This can be easily judged with basic database knowledge; also we can compare the results from other target databases with MySQL target database. In addition, we verify the statement execution by connecting directly to the target databases with vendor-specific database clients. Detailed explanations of the table follow directly after with the numbers.

Input	`CREATE TABLE items (id MEDIUMINT NOT NULL PRIMARY KEY, name VARCHAR(128), quantity DOUBLE, comments MEDIUMTEXT, image BLOB);`[1]	
MySQL	`CREATE TABLE items (id MEDIUMINT NOT NULL PRIMARY KEY, name VARCHAR(128), quantity DOUBLE, comments MEDIUMTEXT, image BLOB);`	✓[2]
Oracle	`CREATE TABLE items (id NUMBER(7, 0) NOT NULL PRIMARY KEY, name VARCHAR2(128), quantity FLOAT(24), comments CLOB, image BLOB)`	✓[2]
PostgreSQL	`CREATE TABLE items (id NUMERIC(7, 0) NOT NULL PRIMARY KEY, name VARCHAR(128), quantity DOUBLE PRECISION, comments TEXT, image BYTEA);`	✓[2]

Input	`CREATE TABLE orders (order_id BIGINT NOT NULL PRIMARY KEY, item_id MEDIUMINT, status BOOLEAN DEFAULT FALSE, shiptime DATETIME, CONSTRAINT fk_item FOREIGN KEY (item_id) REFERENCES items (id) ON DELETE CASCADE);`[1]	
MySQL	`CREATE TABLE orders (order_id BIGINT NOT NULL PRIMARY KEY, item_id MEDIUMINT, status BOOLEAN DEFAULT FALSE, shiptime DATETIME, CONSTRAINT fk_item FOREIGN KEY (item_id) REFERENCES items (id) ON DELETE CASCADE);`	✓[2]
Oracle	`CREATE TABLE orders (order_id NUMBER(19, 0) NOT NULL PRIMARY KEY, item_id NUMBER(7, 0), status NUMBER(1) DEFAULT 0, shiptime TIMESTAMP, CONSTRAINT fk_item FOREIGN KEY (item_id) REFERENCES items (id) ON DELETE CASCADE)`	✓[2]
PostgreSQL	`CREATE TABLE orders (order_id BIGINT NOT NULL PRIMARY KEY, item_id NUMERIC(7, 0), status BOOLEAN DEFAULT FALSE, shiptime TIMESTAMP, CONSTRAINT fk_item FOREIGN KEY (item_id) REFERENCES item (id) ON DELETE CASCADE);`	✓[2]

Input	`INSERT INTO items (id, name, quantity, comments, image) VALUES (1, 'name1', 100, 'comments1 long text', IMG1_BINARY`[3]`), (2, 'name2', 101.1, 'comments2 long text', IMG2_BINARY), (3, 'name3', 123, 'comments3 long text', 0x53696d696e0d0a);`	
MySQL	`INSERT INTO items (id, name, quantity, comments, image) VALUES (1, 'name1', 100, 'comments1 long text', IMG1_BINARY), (2, 'name2', 101.1, 'comments2 long text', IMG2_BINARY), (3, 'name3', 123, 'comments3 long text', 0x53696d696e0d0a);`	X[4]
Oracle	`INSERT ALL INTO items (id, name, quantity, comments, image) VALUES (1, 'name1', 100, 'comments1 long text', IMG1_BINARY) INTO items (id, name, quantity, comments, image) VALUES (2, 'name2', 101.1, 'comments2 long text', IMG2_BINARY) INTO items (id, name, quantity, comments, image) VALUES (3, 'name3', 123, 'comments3 long text', '53696d696e0d0a') SELECT 1 FROM DUAL`	X[5]
PostgreSQL	`INSERT INTO items (id, name, quantity, comments, image) VALUES (1, 'name1', 100, 'comments1 long text', IMG1_BINARY), (2, 'name2', 101.1, 'comments2 long text', IMG2_BINARY), (3, 'name3', 123, 'comments3 long text', E'\\x53696d696e0d0a');`	X[4]

Input	`INSERT INTO orders (order_id, item_id, status, shiptime) VALUES (100, 1, DEFAULT, NULL), (101, 2, DEFAULT, {ts '2013-11-11 11:11:11'});`[4]	
MySQL	`INSERT INTO orders (order_id, item_id, status, shiptime) VALUES (100, 1, DEFAULT, NULL), (101, 2, DEFAULT, {ts '2013-11-11 11:11:11'});`	✓[7]

Oracle	`INSERT ALL INTO orders (order_id, item_id, status, shiptime) VALUES (100, 1, DEFAULT, NULL) INTO orders (order_id, item_id, status, shiptime) VALUES (101, 2, DEFAULT, TIMESTAMP '2013-11-11 11:11:11.0') SELECT 1 FROM DUAL`	X[5]
PostgreSQL	`INSERT INTO orders (order_id, item_id, status, shiptime) VALUES (100, 1, DEFAULT, NULL), (101, 2, DEFAULT, TIMESTAMP '2013-11-11 11:11:11.0');`	✓[7]
Input	`UPDATE orders SET status = true, shiptime = NOW() WHERE item_id IN (SELECT id FROM items WHERE quantity > 100);`	
MySQL	`UPDATE orders SET status = true, shiptime = NOW() WHERE item_id IN (SELECT id FROM items WHERE quantity > 100);`	✓[7]
Oracle	`UPDATE orders SET status = 1, shiptime = CURRENT_TIME() WHERE item_id IN (SELECT id FROM items WHERE quantity > 100)`[6]	✓[7]
PostgreSQL	`UPDATE orders SET status = true, shiptime = NOW() WHERE item_id IN (SELECT id FROM items WHERE quantity > 100);`	✓[7]
Input	`SELECT DISTINCT orders.order_id, orders.item_id, orders.status, orders.shiptime, items.name, items.comments, items.image FROM orders JOIN items WHERE name LIKE 'name%' AND quantity > 100 GROUP BY orders.order_id ORDER BY items.id LIMIT 1, 5;`	
MySQL	`SELECT DISTINCT orders.order_id, orders.item_id, orders.status, orders.shiptime, items.name, items.comments, items.image FROM orders JOIN items WHERE name LIKE 'name%' AND quantity > 100 GROUP BY orders.order_id ORDER BY items.id LIMIT 1, 5;`	✓[8]
Oracle	`SELECT DISTINCT orders.order_id, orders.item_id, orders.status, orders.shiptime, items.name, items.comments, items.image FROM orders JOIN items WHERE name LIKE 'name%' AND quantity > 100 GROUP BY orders.order_id ORDER BY items.id OFFSET 1 ROWS FETCH NEXT 5 ROWS ONLY`	✓[8]
PostgreSQL	`SELECT DISTINCT orders.order_id, orders.item_id, orders.status, orders.shiptime, items.name, items.comments, items.image FROM orders JOIN items WHERE name LIKE 'name%' AND quantity > 100 GROUP BY orders.order_id ORDER BY items.id LIMIT 5 OFFSET 1;`	✓[8]
Input	`DELETE FROM items WHERE id = 1;`	
MySQL	`DELETE FROM items WHERE id = 1;`	✓[9]
Oracle	`DELETE FROM items WHERE id = 1`	✓[9]
PostgreSQL	`DELETE FROM items WHERE id = 1;`	✓[9]
Input	`DROP TABLE orders CASCADE;`	
MySQL	`DROP TABLE orders CASCADE;`	✓[10]
Oracle	`DROP TABLE orders CASCADE CONSTRAINTS`	✓[10]
PostgreSQL	`DROP TABLE orders CASCADE;`	✓[10]

Table 7.2.: SQL Transformation Validation with Cloud Databases

1. The majority of transforming the CREATE TABLE statement is the data type transformation. We introduced in Section 2.2.2 and Appendix A, that different database vendors

have their own set of data types. For most of them, we can find a similar data type in the target database to substitute them with the closest context meaning.

2. To validate the correctness of the CREATE TABLE statement, we check at each target database if the table is created using `LIST TABLE` command (or similar). Besides we use `DESCRIBE TABLE` command (or similar) to verify the table's schema and constraint. Of course it can be also verified when we execute the following DML statements.

3. For the INSERT statement, we will use the `PreparedStatement` from JDBC to help insert binary data (an image file) for the statement. `IMG1_BINARY` and `IMG2_BINARY` here are just placeholders for the binary data we will insert.

4. An important aspect of transforming INSERT statement is the literal data transformation. Here for example, we can see the different expressions of hexadecimal data. However, as a limitation of CDASMix, MySQL Proxy Component will encode binary literal with the syntax (`_binary'data'`) into Base64 format and pass it onto the database. And for all binary query result it received, it will decode the data [Sáe13a]. This raises a problem when we insert binary data with other format such as hexadecimal data. It will not be encoded when inserted, however it will get decoded when we select them. A solution for this will be to encode all binary data types.

5. For Oracle SQL, inserting multiple values is not naturally supported as other databases system do. But it can be achieved in a tricky way, using the multi-table insert syntax with a dummy SELECT statement "`SELECT 1 fROM DUAL`". `DUAL` is a default table in all Oracle database containing one column `DUMMY` defined as `VARCHAR2(1)` [ora]. However with such syntax, we lose the ability to retrieve the generated keys by the statement and will raise exception in CDASMix. This can be solved when we support transforming one statement to multiple statements.

6. As boolean value is not supported in Oracle database, we use a single digit number for substitution (see 1). Since we transformed the data type, the data literal also has to be transformed (true to 1, false to 0). To be noted, JDBC can still retrieve boolean value when we query the data as JDBC naturally support casting other data types to boolean value (1 to false, 0 to true) [jdb].

7. To validate INSERT and UPDATE statements, first we check the result set for auto-generated keys produced by this statement. Then we use SELECT statement to check the data in the target database. Especially for the binary data, we will write the selected data into a file so that we can see if it matches the image file we inserted before.

8. For SELECT statement, we can see the transformation of the `LIMIT` clause for different target database. Moreover, we verify the integrity of the data we inserted and updated before. We inserted more data as shown here to check the ordering and row limiting clause. As we can see, for some data we use different but compatible data types in different databases. And we can still retrieve them from different databases with JDBC as they are the same data. This proves the assumption we made in Section 4.4 that the result set transformation is unnecessary, however with a limitation that the data should be presentable with the same Java class. Boolean value is an exception (see 6).

9. To validate DELETE statement, we use SELECT statement to check whether the data is deleted. This also verifies the foreign key constraint (`fk_item`) we created before, which will lead to a cascade deletion in `orders` table as well.

10. We use `LIST TABLES` command (or similar command) or SELECT statement to check whether the table is successfully dropped.

7.3. Performance Evaluation

An important aspect of performance of our system is the statement throughput, which measures how many statements can be executed in one unit of time. Since transformation functionality is an add-on feature to the existing CDASMix system using OSGi service, its performance drawback can be evaluated as a standalone module.

As we can see in Chapter 6, the transformation of a statement is made up of two parts: parsing and transforming. Both follow a depth-first pre-order tree traversal as we introduced earlier. And the tree we are referring to is the parse tree as presented in Figure 7.1. In graph theory, such tree structure is named *k-ary tree* which is a rooted ordered tree in which each node has no more than k children. And the time complexity of the traversal for such tree would be $O(n)$ where n is the number of nodes.

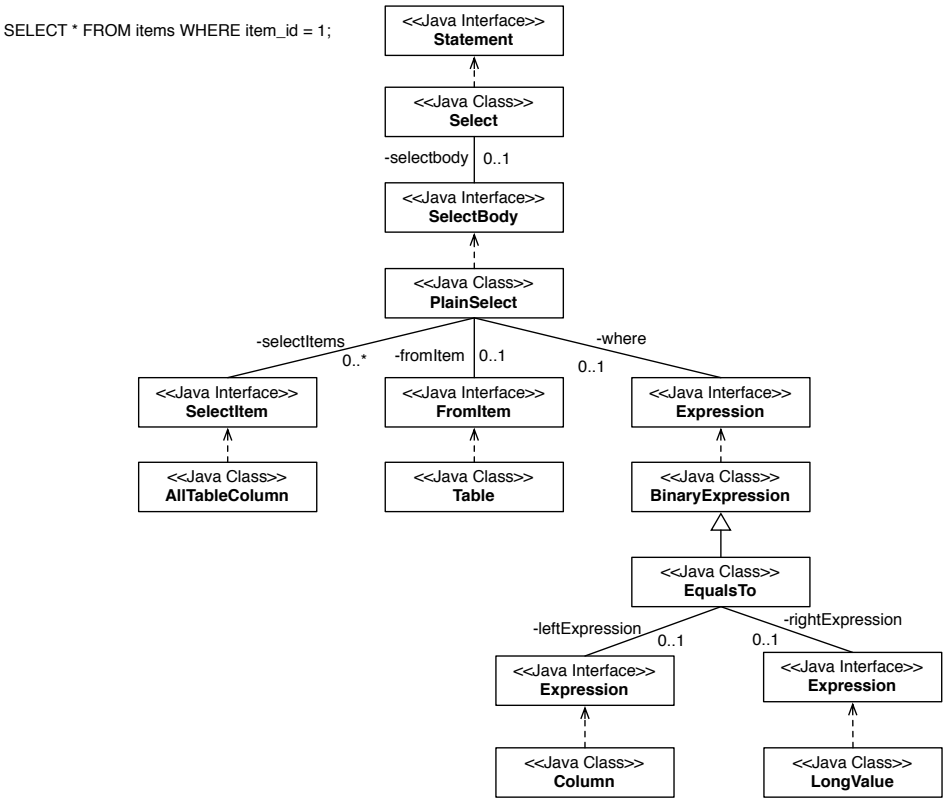

Figure 7.1.: SELECT Statement's Parse Tree in Class Diagram

With this assumption, we then proceed by categorizing SQL statements based on their resulting parse tree's total number of nodes, and evaluate the time consumption and the throughput of parsing and transforming the statements. To fairly measure the throughput, we monitor the time it takes to parse and transform a statement for 10000 times, and measure this value for 100 times. We then take the average of these 100 values for the time consumption (second/statement). With the time consumption we can then calculate the throughput (number of statement/second) by inverting it. We performed a load testing with one representative load containing a mixture of various statements, and another load with only select statements. Table 7.3 and Table 7.4 show us the evaluation for statements with different number of nodes. And Figure 7.2 and Figure 7.3 visualizes the relationship between the number of nodes and the time consumption, as well as the throughput.

Statement	Number of Nodes	Time (s)	Throughput (1/s)
`drop table t;`	3	0.000028497	35091.4131312068
`drop table if exists t, t1;`	5	0.000029394	34020.5484112404
`select * from items;`	8	0.000033343	29991.3025222685
`create table t (a int, b varchar);`	10	0.000038029	26295.7216860817
`update items set item_id = 1 where item_id = 1;`	13	0.000042708	23414.8168961319
`select a from items where item_id = 1;`	17	0.00004689	21326.5088505012
`insert into items (a, b, c) values(1, 2, c + 2);`	18	0.000043151	23174.4339644504
`select sum(price) from inventory where name = 'banana' order by id limit 10, 20;`	25	0.000059162	16902.7416246915
`select id from item where producer in (select name from production where nation = 'germany');`	31	0.000086838	11515.6958935028
`select name, address from contact where birthday = TODAY() union select employ from company where name = 'IBM' order by id Limit 0, 100;`	41	0.000077677	12873.8236543636
`select inventory.id, orders.id, sum(orders.price) as sum from (select * from inventory cross join orders where orders.status = false and inventory.quantity > 1000 order by inventory.id limit 0, 100);`	51	0.000108058	9254.28936311981
`CREATE TABLE employees_demo (employee_id NUMBER(6), first_name VARCHAR2(20), last_name VARCHAR2(25) NOT NULL, email VARCHAR2(25) NOT NULL, phone_number VARCHAR2(20), hire_date DATE NOT NULL DEFAULT SYSDATE, job_id VARCHAR2(10) NOT NULL, salary NUMBER(8, 2) NOT NULL, commission_pct NUMBER(2, 2), manager_id NUMBER(6), department_id NUMBER(4), dn VARCHAR2(300), CONSTRAINT emp_email_uk UNIQUE (email));`	63	0.000122866	8138.94812234467
`select firstname, lastname from (select * from contact) join (select * from employ where company='IBM') join (select * from employer) join (select * from uni) where birthday=TODAY() AND birthplace='stuttgart' limit 0,10;`	71	0.000114584	8727.22195070865

(select firstname, lastname from contact where age > 25) union (select salary, workingage from employee where company ='IBM' and location='stuttgart') union (select spouse, parent from registry where region='bw' or region='bayern') order by firstname limit 0, 100;	80	0.000124342	8042.33485065384
UPDATE employees a SET salary = (SELECT 1.1*AVG(salary) FROM employees b WHERE a.department_id = b.department_id) WHERE department_id IN (SELECT department_id FROM departments WHERE location_id = 2900 OR location_id = 2700);	89	0.000196803	5081.22335533503
select firstname, lastname, address, birthday from contact join (select * from employee, schedule where age>20 and workday=TODAY()) join (select * from employer where name='steve') where birthday=TODAY() or age=24 group by age having max(age)<60 order by firstname, lastname limit 1,1000;	100	0.000163986	6098.0815435464
select l_returnflag, l_linestatus, sum(l_quantity) as sum_qty, sum(l_extendedprice) as sum_base_price, sum(l_extendedprice*(1-l_discount)) as sum_disc_price, sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge, avg(l_quantity) as avg_qty, avg(l_extendedprice) as avg_price, avg(l_discount) as avg_disc, count(*) as count_order from lineitem where l_shipdate < date '1998-12-01' and l_shipdate > date '1998-11-01' group by l_returnflag, l_linestatus order by l_returnflag, l_linestatus limit 0, 100;	125	0.000207711	4814.38152047797
select n_name, sum(l_extendedprice * (1 - l_discount)) as revenue, avg(l_extendedprice*(1-l_discount)*(1+l_tax)) as avg_revenue from customer, orders, lineitem, supplier, nation, region where c_custkey = o_custkey and l_orderkey = o_orderkey and l_suppkey = s_suppkey and c_nationkey = s_nationkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'germany' and o_orderdate >= date '2011-11-11' and o_orderdate < date '2012-11-11' group by n_name order by revenue desc;	150	0.000218036	4586.39857638188
select s_acctbal, s_name, n_name, p_partkey, p_mfgr, s_address, s_phone, s_comment from part, supplier, partsupp, nation, region where p_partkey = ps_partkey and s_suppkey = ps_suppkey and p_size = 100 and p_type like '%type' and s_nationkey = n_nationkey and n_regionkey = r_regionkey and ps_supplycost = (select min(ps_supplycost) from partsupp, supplier, nation, region where p_partkey = ps_partkey and s_suppkey = ps_suppkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'germany') order by s_acctbal desc;	174	0.000286147	3494.70726584588

`select s_acctbal, s_name, n_name, p_partkey, p_-mfgr, s_address, s_phone, s_comment from part, supplier, partsupp, nation, region where p_partkey = ps_partkey and s_suppkey = ps_suppkey and p_-size = 10 and p_type like 'type1' and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'region' and ps_supplycost = (select min(ps_supplycost) from partsupp, supplier, nation, region where p_partkey = ps_partkey and s_suppkey = ps_suppkey and s_nationkey = n_nationkey and n_-regionkey = r_regionkey and r_name = 'region') order by s_acctbal desc, n_name, s_name, p_partkey;`	197	0.000302252	3308.49754509482

Table 7.3.: Throughput Evaluation of Various Statements

Statement	Number of Nodes	Time (s)	Throughput (1/s)
`select * from items;`	8	0.000033343	29991.3025222685
`select * from items where item_id = 1;`	15	0.000045239	22104.8210614735
`select a from items where item_id = 1;`	17	0.00004689	21326.5088505012
`select avg(abcdefg) from items where item_id = 1;`	20	0.000051509	19414.0829757906
`select sum(price) from inventory where name = 'banana' order by id limit 10, 20;`	25	0.000059162	16902.7416246915
`select id from item where producer in (select name from production where nation = 'germany');`	31	0.000086838	11515.6958935028
`select name, address from contact where birthday = TODAY() union select employ from company where name = 'IBM' order by id Limit 0, 100;`	41	0.000077677	12873.8236543636
`select inventory.id, orders.id, sum(orders.price) as sum from (select * from inventory cross join orders where orders.status = false and inventory.quantity > 1000 order by inventory.id limit 0, 100);`	51	0.000108058	9254.28936311981
`select address, concat(firstname, lastname) as name from contact join (select salary, employer from company where name = 'IBM' and location = 'germany') where age>40 and age<60;`	64	0.000109287	9150.21914774859
`select firstname, lastname from (select * from contact) join (select * from employ where company='IBM') join (select * from employer) join (select * from uni) where birthday=TODAY() AND birthplace='stuttgart' limit 0,10;`	71	0.000114584	8727.22195070865
`(select firstname, lastname from contact where age > 25) union (select salary, workingage from employee where company ='IBM' and location='stuttgart') union (select spouse, parent from registry where region='bw' or region='bayern') order by firstname limit 0, 100;`	80	0.000124342	8042.33485065384

`select s_acctbal from part, supplier, partsupp, nation where p_partkey=ps_partkey and s_suppkey=ps_suppkey and p_size=10 and p_type like 'type1' and s_nationkey=n_nationkey and n_regionkey=r_regionkey and r_name='region' order by s_acctbal desc, n_name;`	90	0.00011454	8730.57447180024
`select firstname, lastname, address, birthday from contact join (select * from employee, schedule where age>20 and workday=TODAY()) join (select * from employer where name='steve') where birthday=TODAY() or age=24 group by age having max(age)<60 order by firstname, lastname limit 1,1000;`	100	0.000163986	6098.0815435464
`select l_returnflag, l_linestatus, sum(l_quantity) as sum_qty, sum(l_extendedprice) as sum_base_price, sum(l_extendedprice*(1-l_discount)) as sum_disc_price, sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge, avg(l_quantity) as avg_qty, avg(l_extendedprice) as avg_price, avg(l_discount) as avg_disc, count(*) as count_order from lineitem where l_shipdate < date '1998-12-01' and l_shipdate > date '1998-11-01' group by l_returnflag, l_linestatus order by l_returnflag, l_linestatus limit 0, 100;`	125	0.000207711	4814.38152047797
`select n_name,sum(l_extendedprice*(1-l_discount)) as revenue, avg(l_extendedprice*(1-l_discount)*(1+l_tax)) as avg_revenue from customer, orders, lineitem, supplier, nation, region where c_custkey = o_custkeyand and l_orderkey = o_orderkey and l_suppkey = s_suppkey and c_nationkey = s_nationkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'germany' and o_orderdate >= date'2011-11-11' and o_orderdate < date'2012-11-11' group by n_name order by revenue desc;`	150	0.000218036	4586.39857638188
`select s_acctbal,s_name,n_name,p_partkey,p_mfgr, s_address, s_phone, s_comment from part,supplier, partsupp,nation,region where p_partkey = ps_partkey and s_suppkey = ps_suppkey and p_size = 100 and p_type like '%type' and s_nationkey = n_nationkey and n_regionkey = r_regionkey and ps_supplycost = (select min(ps_supplycost) from partsupp, supplier, nation, region where p_partkey = ps_partkey and s_suppkey = ps_suppkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'germany') order by s_acctbal desc;`	174	0.000286147	3494.70726584588
`select s_acctbal, s_name, n_name, p_partkey, p_mfgr, s_address, s_phone, s_comment from part, supplier, partsupp, nation, region where p_partkey = ps_partkey and s_suppkey = ps_suppkey and p_size = 10 and p_type like 'type1' and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'region' and ps_supplycost = (select min(ps_supplycost) from partsupp, supplier, nation, region where p_partkey = ps_partkey and s_suppkey = ps_suppkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'region') order by s_acctbal desc, n_name, s_name, p_partkey;`	197	0.000302252	3308.49754509482

Table 7.4.: Throughput Evaluation of SELECT Statements

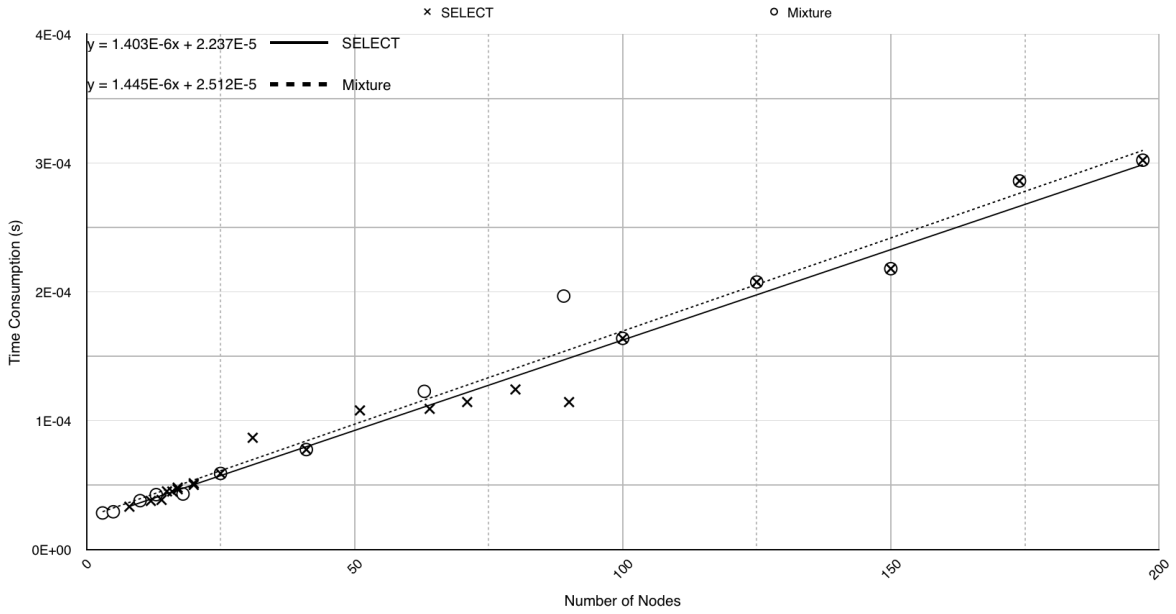

Figure 7.2.: Plot of Time Consumption Over Number of Nodes

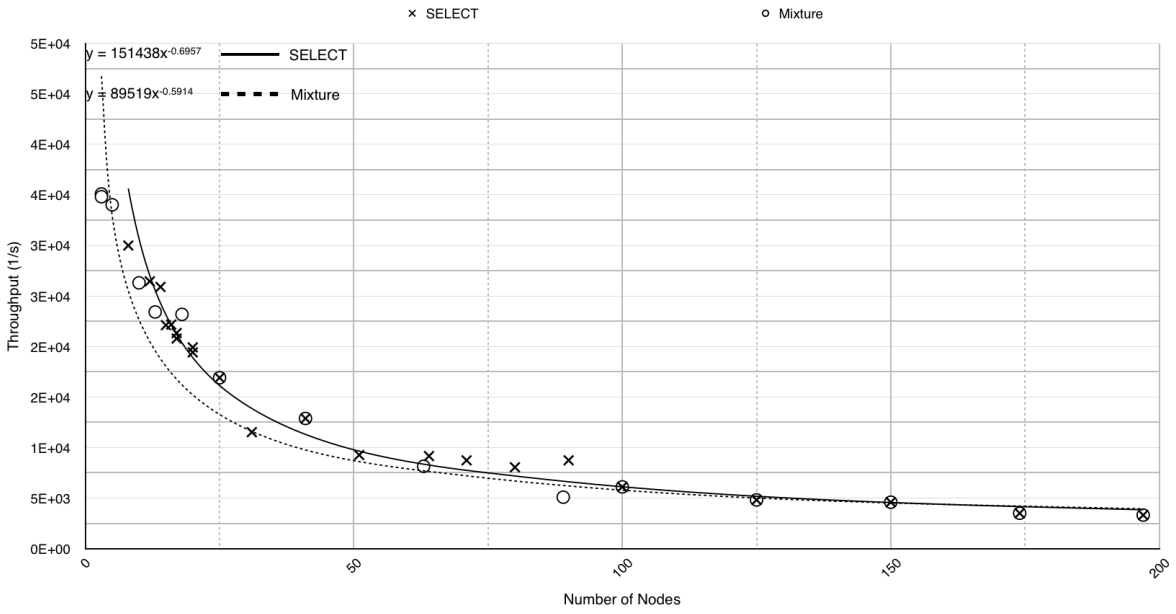

Figure 7.3.: Plot of Throughput Over Number of Nodes

For the first load testing, we used a mixture of statements of various types, to represent the normal usage of the parser. And a second load use only SELECT statements. From Figure 7.2 and Figure 7.3, we can see as a trend that the time consumption increases along with the number of nodes with an interpolation function of $y = 1.403\text{E-}6x + 2.237\text{E-}5$ and $y = 1.445\text{E-}6x + 2512\text{E-}5$ for the two loads respectively. Whereas throughput presents an interpolation function of $y = 151438x^{-0.6957}$ and $y = 89519x^{-0.5914}$. This coincides with the $O(n)$ notion we proposed previously for the complexity of our SQL parser and transformer.

8. Conclusion and Future Work

8.1. Conclusion

To migrate existing application's database layer to the Cloud and take advantage of Cloud computing, there are various issues to address to adapt both the migrated and non-migrated layers. Different Cloud database requires different communication protocols. They have different data structures, and various access languages. *Multi-Tenant ServiceMix with Cloud Data Access Support (CDASMix)* by Gómez Sáez provided us a transparent Cloud data access layer with multi-protocol and multi-tenant support. Applications can access the migrated database without knowing its actual location or system type. However, for relational databases, their access languages (SQL) as introduced in Chapter 2 still differ from each other which poses difficulty for the exiting database application to use legacy queries with the migrated database system when they are of different types. In this thesis, we focus on such issue, to provide a transparent way to transform the SQL statements in CDASMix so that they can be executed upon the migrated Cloud database.

In Chapter 2, we offered a broad illustration of the necessary background information to support this thesis. Especially, we detailed the relationship between relational database and SQL language, and how SQL is used to access and manipulate relational database independently from the physical storage of the database system. Also, we analyzed the syntax of SQL statement and how it can be parsed into a parse tree. Furthermore, we talked about our system environment regarding Apache ServiceMix with its OSGi and JBI container, as well as a brief introduction of CDASMix.

In Chapter 3, we discussed the research we did on other's work that is related to our topic. There is Multi-database system which resembles our system structure that provides an intermediate layer for multiple autonomous, distributed, and/or heterogeneous database systems. However, our system is limited to access one database at a time, which differs from the definition of a Multi-database system. Then there is another method to adapt to migrated database, which is called *Application Migration*. It requires changes to be made with the database applications that alters the SQL statements resided within the applications in order to adapt to the migrated database system. However, such method disobeys our original goal to minimize the impact on the applications. In addition, we researched some existing commercial products that provide similar functionality of SQL transformation as ours.

After analyzing the system structure of CDASMix and various dialects of SQL language, we proposed several functional and non-functional requirements our components need to fulfill. Most importantly, we specified that the SQL transformation needs to be realized with a two-step functions, SQL statement parsing (FR1) and SQL statement transforming (FR2).

Based on these requirements, we then went on proposing two different approaches to integrate SQL transformation into CDASMix. The first approach is to modularize the transformation functionality and isolate it from other parts of the system. The communication between the system and transformation component is realized with NM through NMR. However, the actual transformation functionality is modeled as OSGi service. The transformation component needs to look up the the corresponding transformation service to transform the SQL statements it received from the proxy component and relay them to the tenant-aware Camel endpoint via NMR. This design becomes redundant when we realize that for each proxy component which communicates with one type of database system, it requires only the transformation service(s) which deals that particular SQL dialect used by the database system. Thus, we proposed and decided on a second approach where the proxy component communicate directly with the transformation service. This design removed the intermediate transformation component, and let the proxy bundle directly look up transformation services and transform the SQL statements before they are marshaled into a NM. Subsequently, we detailed the SQL transformation service design, which is defined as OSGi declarative service. It fulfills our functional requirements FR1 and FR2, as well as takes advantage of the dynamism and flexibility of OSGi container which satisfies our non-functional requirements.

In Chapter 6 and Chapter 7, we illustrated the implementation, validation and evaluation of our system prototype, which complies with the system requirements and design we proposed in previous chapters. We clarified with code how the SQL transformation works, how the transformation service is declared, and how the service lookup is done. For validation, we separately verified the correctness of SQL transformation, and its integration with CDASMix and Cloud databases. The performance of the transformation is evaluated by its throughput, meaning how many statements can be parsed and transformed in one unit of time. We concluded that the throughput is inversely proportion to the complexity of the SQL statement, that is the number of nodes of its parse tree.

8.2. Future Work

For future consideration, there are several aspects of work need to be further carried on.

In our prototype, we support only a limited types of SQL statements to be parsed. The existing parser should be extended to support more SQL statement types. Parser for more SQL dialect should also be developed to support more source dialects. More target dialects should also be added as well to broaden the transformation support.

Regarding SQL transformation, in the current implementation, we support only one to one transformation. That is one SQL transformation can be only transformed to another statement. In some cases, the effect of one SQL statement in one database system, can be realized in another database system with two or more statements, such as when inserting multiple values into a table in Oracle database (see Section 7.2.2).

Also as pointed out in Section 7.2.2, we have a problem when inserting and retrieving binary data. In CDASMix binary data with prefix "`_binary`" in an SQL statement is encoded

into Base64 string representation to avoid character set problem for the SQL parser in the transformation [Sáe13a]. And all binary data retrieved from the backend database is then decoded back to the original data. However, there is more than one expression for binary data, such as hexadecimal data (`X'val' or 0xval` in MySQL). And these data will not be encoded when inserted, but will be decoded when retrieved. A possible solution for this would be to encode all types of binary data. Or we can substitute the binary data with a placeholder before passing the statement to the parser and place it back after transformation. This way it could speed up the transformation process when the binary data get significantly large, which normally is the case.

As the transformation service includes an SQL parser, the service interface can be extended to include more functionalities, such as deciding SQL statement type, retrieving the affected database schema object (table), etc. Since they are realized with a full edge parser, the effectiveness and robustness can be guaranteed even when the input statement is malformed. These functionalities can then become useful to the proxy bundle besides transforming SQL statement.

Appendix A.

Data Types

The following table lists a collection of data types that are respectively supported by different database systems [mys] [pos] [ora] [KKH08]. Letter Y indicates that the data type listed is supported as it is in the corresponding database system. And the others that listed under the database systems are of equivalence to that data type, used by the database system. The empty entry indicates that the corresponding data type is not supported in this database system.

Data Type	MySQL 5.6	PostgreSQL 9.3	Oracle 12c Release 1 (12.1)
Numeric[1]			
TINYINT[2]	Y; INT1	NUMERIC(3)	NUMBER(3)
SMALLINT[2]	Y; INT2	Y; INT2	Y
MEDIUMINT[2]	Y; INT3	NUMERIC(7)	NUMBER(7)
INTEGER, INT[2]	Y; INT4	Y; INT4	Y
BIGINT[1]	Y; INT8	Y; INT8	NUMBER(19)
REAL	Y; REAL(M, D)	Y; FLOAT4	FLOAT(n)
DOUBLE	Y; DOUBLE(M, D)	DOUBLE PRECISION	FLOAT(n)
DOUBLE PRECISION	Y; DOUBLE PRECISION(M, D)	Y; FLOAT8	FLOAT(n)
FLOAT[(p)]	Y, FLOAT(M, D)	Y	Y
DECIMAL[(p [,s])]	Y, DEC[(p [,s])], FIXED[(p [,s])]	Y	NUMBER[(p, s)]
NUMERIC[(p [,s])]	Y	Y	NUMBER[(p[, s])]
NUMBER[(p [,s])]	NUMERIC[(p [,s])]	NUMERIC[(p [,s])]	Y
BINARY_FLOAT			Y
BINARY_DOUBLE			Y
SMALLSERIAL[3]		Y, SERIAL2	
SERIAL[3]	Y [4]	Y, SERIAL4	
BIGSERIAL[3]		Y, SERIAL8	

Data Type	MySQL 5.6	PostgreSQL 9.3	Oracle 12c Release 1 (12.1)
BOOL, BOOLEAN	Y, TINYINT(1)	Y	
Characters			
CHARACTER[(n)], CHAR[(n)]	Y	Y	Y
CHARACTER VARING(n), CHAR VARING(n), VARCHAR(n)	Y	Y, CHARACTER VARING, CHAR VARING, VARCHAR	Y, VARCHAR2(n)
NATIONAL CHARACTER[(n)],NATIONAL CHAR[(n)], NCHAR[(n)]	Y	Y	Y
NATIONAL CHARACTER VARING(n), NATIONAL CHAR VARYING(n)	Y, NATIONAL VARCHAR(n), NVARCHAR(n)	Y, NATIONAL CHARACTER VARING, NATIONAL CHAR VARYING	Y, NCHAR VARING(n), NVARCHAR2(n)
TINYTEXT	Y		VARCHAR2(n)
TEXT	Y	Y	CLOB
MEDIUMTEXT	Y		CLOB
LONGTEXT	Y		CLOB
Date and Time			
DATE	Y	Y	Y
DATETIME[(p)]	Y		TIMESTAMP[(p)]
TIMESTAMP[(p)]	Y	Y, TIMESTAMP[(p)] WITHOUT TIME ZONE	Y
TIMESTAMP[(p)] WITH TIME ZONE		Y, TIMESTAMPTZ[(p)]	Y
TIME[(p)]	Y	Y, TIME[(p)] WITHOUT TIME ZONE	
TIME[(p)] WITH TIME ZONE			
YEAR[(2 | 4)]	Y		
INTERVAL YEAR [(yp)] TO MONTH		INTERVAL YEAR TO MONTH	Y
INTERVAL DAY[(dp)] TO SECOND[(p)]		INTERVAL DAY TO SECOND[(p)]	Y
INTERVAL [*field*] [(p)][5]		Y	INTERVAL YEAR [(year_precision)] TO MONTH, INTERVAL DAY [(day_precision)] TO SECOND [(fractional_seconds_precision)]
Raw data			
BIT[(n)]	Y	Y	

Data Type	MySQL 5.6	PostgreSQL 9.3	Oracle 12c Release 1 (12.1)
BIT VARYING[(n)], VARBIT[(n)]		Y	
BINARY[(n)]	Y		
VARBINARY[(n)]	Y		
BYTEA		Y	
LONG			Y
RAW			Y
RAW LONG			Y
TINYBLOB	Y		
BLOB	Y		Y
MEDIUMBLOB	Y		
LONGBLOB	Y		
CLOB			Y
NCLOB			Y
BFILE			Y
Enumeration			
ENUM(value, ...)	Y	Y[6]	
SET(value, ...)	Y		
Spatial Type			
GEOMETRY	Y		
POINT	Y	Y	
CURVE	Y		
LINESTRING	Y		
SURFACE	Y		
POLYGON	Y	Y	
GEOMETRYCOLLECTION	Y		
MULTIPOINT	Y		
MULTILINESTRING	Y		
MULTIPOLYGON	Y		
BOX		Y	
CIRCLE		Y	
LINE		Y	
LSEG		Y	
PATH		Y	
SDO_GEOMETRY			Y

Data Type	MySQL 5.6	PostgreSQL 9.3	Oracle 12c Release 1 (12.1)
SDO_TOPO_GEOMETRY			Y
SDO_GEORASTER			Y
Misc			
CIDR		Y	
INET		Y	
MACADDR		Y	
MONEY		Y	
OID		Y	
ROWID			Y
UROWID[(n)]			Y
XML		Y	
XMLTYPE			Y
JSON		Y	

Table A.1.: Comparison of SQL Data Types of Various Vendors.

1. In MySQL, there are several nonstandard attributes for numeric types [mys]. `ZEROFILL` attribute is used in conjunction with the *display width* value to specify the display representation of the numeric value. For example, if a column is declared as `INT(4) ZEROFILL`, value 5 is then retrieved as 0005. This does not constraint the range of values that can be stored in the column. All numeric types can have another optional attribute `UNSIGNED`, indicating the column only accept non-negative numbers. If you specify `ZEROFILL` for a numeric column, MySQL automatically adds the `UNSIGNED` attribute to the column. Integer or floating-point data types can have an additional `AUTO_INCREMENT` attribute which makes the column `AUTO_INCREMENT` indexed, meaning when you insert a value of `NULL` or 0 into such column, the inserted value will be set to the next sequence value (*value + 1*), where *value* is the largest value for the column currently in the table. `AUTO_INCREMENT` sequence begins with 1.

2. `TINYINT`, `SMALLINT`, `MEDIUMINT`, `INT` and `BIGINT` are integers with 1, 2, 3, 4 and 8 byte(s) respectively [mys], and they may be substituted with `NUMERIC(p, 0)` or `NUMBER(p, 0)`, which will have a larger range. For example, `TINYINT` has a range of (-128, 127), while `NUMERIC(3, 0)` has a range of (-999, 999).

3. `SMALLSERIAL`, `SERIAL` and `BIGSERIAL` are auto-incrementing integers. For MySQL, the keyword "`AUTO_INCREMENT`" can be used to enable such feature, such as "`SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE`" can replace `SMALLSERIAL` in MySQL. And in Oracle DB, we can use sequence instead.

4. In MySQL, data type `SERIAL` is an alias for `BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE` [mys].

5. The *field* can take value: `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `YEAR TO MONTH`, `DAY TO HOUR`, `DAY TO MINUTE`, `DAY TO SECOND`, `HOUR TO MINUTE`, `HOUR TO SECOND`, and `MINUTE TO SECOND`. And precision *p* applies only to `SECOND` [pos].
6. In PostgreSQL, ENUM type has to be created first using `CREATE TYPE type_name AS ENUM(value1, value2, ...)`, which then can be used with "`type_name`" in the `CREATE TABLE` statement [pos].

Appendix B.

SQL Statement Comparison

In this section, we compare several important DML and DDL statements between MySQL 5.6 [mys], PostgreSQL 9.3 [pos], and Oracle SQL 12.1 [ora]. The clauses that are aligned in the same row indicates that they are of the same meaning, which means they might be transformable to each other, unless specified otherwise. When then entry is empty, the corresponding database doesn't have an equivalent clause to that row. Uppercase words such as `SELECT`, `WHERE` are keywords, while lowercase words are expressions (`where_condition`), identifiers (`table_name`), or external definition of syntaxes (`from_list`) which are not presented here. When bold italic font is used, such as ***create_definition*** in Section B.7, the syntax structure is then presented later in the table.

B.1. Select Statement

MySQL 5.6 http://dev.mysql.com/doc/refman/5.6/en/select.html	PostgreSQL 9.3 http://www.postgresql.org/docs/9.3/interactive/queries.html	Oracle 12c Release 1 (12.1) http://docs.oracle.com/cd/E16655_01/server.121/e17209/statements_10002.htm
	[WITH [RECURSIVE] with_query [, ...]]	[WITH with_clause]
SELECT	SELECT	SELECT [hint]
[ALL | DISTINCT | DISTINCTROW]	[ALL | DISTINCT [ON (expression [, ...])]]	[DISTINCT | UNIQUE | ALL]
[HIGH_PRIORITY] [STRAIGHT_JOIN] [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT] [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]		
select_expr [, select_expr ...]	* | expression [[AS] output_name] [, ...]	[t_alias.]* | expr [[AS] alias] [,...]
	[INTO [TEMPORARY | TEMP | UNLOGGED] [TABLE] new_table]	
[FROM table_references	[FROM from_item [, ...]]	FROM from_list
[PARTITION partition_list]		
[WHERE where_condition]	[WHERE condition]	[WHERE condition]
		[CONNECT BY [NOCYCLE] condition [START WITH condition] | START WITH condition CONNECT BY [NOCYCLE] condition]
[GROUP BY {col_name | expr | position} [ASC | DESC], ... [WITH ROLLUP]]	[GROUP BY expression [, ...]]	[GROUP BY {expr | { ROLLUP | CUBE } expr_list | GROUPING SETS (list) } [,...]
[HAVING where_condition]	[HAVING condition [, ...]]	[HAVING condition]]
	[WINDOW window_name AS (window_definition) [, ...]]	[MODEL model_clause]
[UNION [ALL | DISTINCT] select]	[{ UNION | INTERSECT | EXCEPT } [ALL | DISTINCT] select]	[{ UNION [ALL] | INTERSECT | MINUS } subquery]
[ORDER BY {col_name | expr | position } [ASC | DESC], ...]	[ORDER BY expression [ASC | DESC | USING operator] [NULLS { FIRST | LAST }] [, ...]]	[ORDER [SIBLINGS] BY { expr | position | c_alias } [ASC | DESC] [NULLS FIRST | NULLS LAST] [, ...]]
[LIMIT {[offset,] row_count | row_count OFFSET offset}]	[LIMIT { count | ALL }] [OFFSET start [ROW | ROWS]]	[OFFSET offset { ROW | ROWS}]
	[FETCH { FIRST | NEXT } [count] { ROW | ROWS } ONLY]	[FETCH { FIRST | NEXT } [{ rowcount | percent PERCENT }] { ROW | ROWS } { ONLY | WITH TIES }]

MySQL 5.6 `http://dev.mysql.com/doc/refman/5.6/en/select.html`	PostgreSQL 9.3 `http://www.postgresql.org/docs/9.3/interactive/queries.html`	Oracle 12c Release 1 (12.1) `http://docs.oracle.com/cd/E16655_01/server.121/e17209/statements_10002.htm`
[INTO OUTFILE 'file_name' [CHARACTER SET charset_name] export_options | INTO DUMPFILE 'file_name' | INTO var_name [, var_name]]		
[FOR UPDATE | LOCK IN SHARE MODE]]	[FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [OF table_name [, ...]] [NOWAIT] [...]]	FOR UPDATE [OF [column [, . . .]] [{ NOWAIT | WAIT integer | SKIP LOCKED}]

Table B.1.: Comparison of SQL Select Statements of Various Vendors

B.2. Delete Statement

MySQL 5.6 http://dev.mysql.com/doc/refman/5.6/en/delete.html	PostgreSQL 9.3 http://www.postgresql.org/docs/9.3/interactive/sql-delete.html	Oracle 12c Release 1 (12.1) http://docs.oracle.com/cd/E16655_01/server.121/e17209/statements_8005.htm
	[WITH [RECURSIVE] with_query [, ...]]	
DELETE	DELETE	DELETE
[LOW_PRIORITY] [QUICK] [IGNORE]		[hint]
FROM tbl_name [, . . .]	FROM [ONLY] table_name [*] [[AS] alias]	[FROM] [ONLY] dml_table_expr [alias]
[USING table_references][1]	[USING using_list]	
[PARTITION (partition_name,...)]		
[WHERE where_condition]	[WHERE condition |	[WHERE condition]
	WHERE CURRENT OF cursor_name]	
[ORDER BY ...] [LIMIT row_count] [1]		
	[RETURNING * | output_expression [[AS] output_name] [, ...]]	[RETURN | RETURNING expr [,...] INTO data_item [,...]]
		[LOG ERRORS [INTO table] [(simple_expr)] [REJECT LIMIT int | UNLIMITED]]

Table B.2.: Comparison of SQL Delete Statements of Various Vendors

1. In MySQL, USING clause applies only to Multi-Table delete. ORDER BY and LIMIT clauses on the other hand only apply to single-table delete.

B.3. Update Statement

MySQL 5.6 `http://dev.mysql.com/doc/refman/5.6/en/update.html`	PostgreSQL 9.3 `http://www.postgresql.org/docs/9.3/interactive/sql-update.html`	Oracle 12c Release 1 (12.1) `http://docs.oracle.com/cd/E16655_01/server.121/e17209/statements_10008.htm`
	[WITH [RECURSIVE] with_query [, ...]]	
UPDATE	UPDATE	UPDATE
[LOW_PRIORITY] [IGNORE]		[hint]
table_reference	[ONLY] table_name [*] [[AS] alias]	[ONLY] dml_table_expr [alias]
SET col_name1=expr1 | DEFAULT [, col_name2={expr2 | DEFAULT}] ...	SET { column_name = { expression | DEFAULT } | (column_name [, ...]) = ({ expression | DEFAULT } [, ...]) } [, ...]	SET {{ column_name = expression | DEFAULT | (column_name [, ...]) = (expression | DEFAULT [, ...]) } [, ...] | VALUE (t_alias) = expr | subquery }
	[FROM from_list] [1]	
[WHERE where_condition]	[WHERE condition |	[WHERE condition]
	WHERE CURRENT OF cursor_name]	
[ORDER BY ...] [LIMIT row_count]		
	[RETURNING * | output_expression [[AS] output_name] [, ...]]	[RETURN | RETURNING expr [,...] INTO data_item [,...]]
		[LOG ERRORS [INTO table] [(simple_expr)] [REJECT LIMIT int | UNLIMITED]]

Table B.3.: Comparison of SQL Update Statements of Various Vendors

1. In PostgreSQL, the *from_list* is a list of table expressions, allowing columns from other tables to appear in the WHERE condition and the update expressions.

B.4. Insert Statement

MySQL 5.6 http://dev.mysql.com/doc/refman/5.6/en/insert.html	PostgreSQL 9.3 http://www.postgresql.org/docs/9.3/interactive/sql-insert.html	Oracle 12c Release 1 (12.1) http://docs.oracle.com/cd/E16655_01/server.121/e17209/statements_9014.htm
	[WITH [RECURSIVE] with_query [, ...]]	
INSERT	INSERT	INSERT
[LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]		[hint]
[INTO] tbl_name	INTO table_name	INTO dml_table_expr [t_alias]
[PARTITION (partition_name,...)]		
[(column_name[,...])]	[(column_name [, ...])]	[(column_name [, ...])]
	{ DEFAULT VALUES |	
{{VALUES | VALUE} ({expr | DEFAULT},...),(...), |	VALUES ({ expression | DEFAULT } [, ...]) [, ...] |	{ VALUES ({ expression | DEFAULT } [, ...])[1]
		[{RETURN | RETURNING } expr [,...] INTO data_item [,...]] |
query |	query }	query }
SET col_name=expr | DEFAULT, . . . }		
[ON DUPLICATE KEY UPDATE col_name=expr [, col_name=expr] ...]	[RETURNING * | output_expression [[AS] output_name] [, ...]]	[LOG ERRORS [INTO table] [(simple_expr)] [REJECT LIMIT { int | UNLIMITED }]]

Table B.4.: Comparison of SQL Insert Statements of Various Vendors

1. In Oracle, INSERT statement only support inserting one row of values into a single table using VALUES clause.

B.5. Drop Table Statement

MySQL 5.6 http://dev.mysql.com/doc/refman/5.6/en/drop-table.html	PostgreSQL 9.3 http://www.postgresql.org/docs/9.3/interactive/sql-droptable.html	Oracle 12c Release 1 (12.1) http://docs.oracle.com/cd/E16655_01/server.121/e17209/statements_9003.htm
DROP	DROP	DROP
[TEMPORARY]		
TABLE	TABLE	TABLE
table_name [,...]	table_name [,...]	table_name
[RESTRICT | CASCADE]	[CASCADE | RESTRICT]	[CASCADE CONSTRAINTS] [PURGE]

Table B.5.: Comparison of SQL Drop Statements of Various Vendors

B.6. Truncate Table Statement

MySQL 5.6 http://dev.mysql.com/doc/refman/5.6/en/truncate-table.html	PostgreSQL 9.3 http://www.postgresql.org/docs/9.3/interactive/sql-truncate.html	Oracle 12c Release 1 (12.1) http://docs.oracle.com/cd/E16655_01/server.121/e17209/statements_10007.htm1
TRUNCATE [TABLE]	TRUNCATE [TABLE]	TRUNCATE TABLE
table_name [,...]	[ONLY] table_name [*] [,...]	table_name
	[RESTART IDENTITY | CONTINUE IDENTITY]	[PRESERVE | PURGE MATERIALIZED VIEW LOG]
	[CASCADE | RESTRICT]	[DROP [ALL] | REUSE STORAGE] [CASCADE]

Table B.6.: Comparison of SQL Truncate Statements of Various Vendors

B.7. Create Table Statement

MySQL 5.6 `http://dev.mysql.com/doc/refman/5.6/en/create-table.html`	PostgreSQL 9.3 `http://www.postgresql.org/docs/9.3/interactive/sql-createtable.html`	Oracle 12c Release 1 (12.1)[1] `http://docs.oracle.com/cd/E16655_01/server.121/e17209/statements_7002.htm`
CREATE [TEMPORARY]	CREATE [[GLOBAL | LOCAL] TEMPORARY | TEMP | UNLOGGED]	CREATE [GLOBAL TEMPORARY]
TABLE [IF NOT EXISTS] table	TABLE [IF NOT EXISTS] table	TABLE table
(*create_definition*, ...)	([*create_definition*, ...])	[(*create_definition*, ...)]
[table_option [,] ...]	[INHERITS (parent_table [, ...])]	
[partition_options]	[WITH (storage_parameter [= value] [, ...]) | WITH OIDS | WITHOUT OIDS]	
	[ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP }]	[ON COMMIT { DELETE | PRESERVE } ROWS]
	[TABLESPACE tablespace_name]	
[[IGNORE | REPLACE] [AS] query]	[AS query [WITH [NO] DATA]]	[AS query]
	or	
	CREATE [[GLOBAL | LOCAL] TEMPORARY | TEMP | UNLOGGED]	CREATE [GLOBAL TEMPORARY]
	TABLE [IF NOT EXISTS] table_name OF type_name	TABLE table OF type_name
	[({ column_name WITH OPTIONS [*column_constraint* [...]] | *create_definition* } [, ...])]	[(*create_definition*, ...)]
	[WITH (storage_parameter [= value] [, ...]) | WITH OIDS | WITHOUT OIDS]	
	[ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP }]	[ON COMMIT { DELETE | PRESERVE } ROWS]
	[TABLESPACE tablespace_name]	[OBJECT IDENTIFIER IS { SYSTEM GENERATED | PRIMARY KEY }]

MySQL 5.6 `http://dev.mysql.com/doc/refman/5.6/en/create-table.html`	PostgreSQL 9.3 `http://www.postgresql.org/docs/9.3/interactive/sql-createtable.html`	Oracle 12c Release 1 (12.1)[1] `http://docs.oracle.com/cd/E16655_01/server.121/e17209/statements_7002.htm`
	create_definition:	
col_name data_type [***column_constraint*** [...]]	col_name data_type [***column_constraint*** [...]]	col_name data_type [***column_constraint*** [...]]
| [CONSTRAINT [symbol]] PRIMARY KEY [index_type] (index_col_name,...) [index_option] ...	| [CONSTRAINT name] PRIMARY KEY (column_name,...) index_parameters	| [CONSTRAINT name] PRIMARY KEY (column_name,...)
| [CONSTRAINT [symbol]] UNIQUE [INDEX | KEY] [index_name] [index_type] (index_col_name,...) [index_option] ...	| [CONSTRAINT name] UNIQUE (column_name,...) index_parameters	| [CONSTRAINT name] UNIQUE (column_name,...)
| [CONSTRAINT [symbol]] FOREIGN KEY [index_name] (index_col_name,...) REFERENCES tbl_name (index_col_name,...) [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE] [ON DELETE action[2]] [ON UPDATE action[2]]	| [CONSTRAINT name] FOREIGN KEY (col_name,...) REFERENCES reftable [(refcolumn,...)] [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE] [ON DELETE action[2]] [ON UPDATE action[2]]	| [CONSTRAINT name] FOREIGN KEY (col_name,...) REFERENCES reftable [(refcolumn,...)] [ON DELETE { CASCADE | SET NULL }]
| CHECK (expr)	| [CONSTRAINT name] CHECK (expression) [NO INHERIT]	| [CONSTRAINT name] CHECK (expression)
| { INDEX | KEY } [index_name] [index_type] (index_col_name,...) [index_option] ...		
| { FULLTEXT | SPATIAL } [INDEX | KEY] [index_name] (index_col_name,...) [index_option] ...		
	| [CONSTRAINT name] EXCLUDE [USING index_method] (exclude_element WITH operator [, ...]) index_parameters [WHERE (predicate)]	
	[[NOT] DEFERRABLE] [INITIALLY { DEFERRED | IMMEDIATE }]	[[NOT] DEFERRABLE] [INITIALLY { DEFERRED | IMMEDIATE }] [RELY | NORELY] [using_index_clause] [ENABLE | DISABLE] [VALIDATE | NOVALIDATE] [exceptions_clause]
| LIKE old_tbl_name	| LIKE source_table [{ INCLUDING | EXCLUDING } { DEFAULTS | CONSTRAINTS | INDEXES | STORAGE | COMMENTS | ALL }]	

MySQL 5.6 `http://dev.mysql.com/doc/refman/5.6/en/create-table.html`	PostgreSQL 9.3 `http://www.postgresql.org/docs/9.3/interactive/sql-createtable.html`	Oracle 12c Release 1 (12.1)[1] `http://docs.oracle.com/cd/E16655_01/server.121/e17209/statements_7002.htm`
	column_constraint:	
		[SORT] [VISIBLE | INVISIBLE] [DEFAULT [ON NULL] expr | identity_clause] [ENCRYPT encryption_spec]
	[CONSTRAINT constraint_name]	[CONSTRAINT constraint_name]
[NOT] NULL	{ [NOT] NULL	{ [NOT] NULL
| DEFAULT default_value	| DEFAULT default_expr	
| AUTO_INCREMENT		
| UNIQUE [KEY]	| UNIQUE index_parameters	| UNIQUE
| [PRIMARY] KEY	| PRIMARY KEY index_parameters	| PRIMARY KEY
| [COMMENT 'string']	| CHECK (expression) [NO INHERIT]	| CHECK (expression) [NO INHERIT]
| [COLUMN_FORMAT {FIXED | DYNAMIC | DEFAULT}]		
| REFERENCES tbl_name (index_col_name [,...]) [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE] [ON DELETE action[2]] [ON UPDATE action[2]]	| REFERENCES reftable [(refcolumn)] [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE] [ON DELETE action[2]] [ON UPDATE action[2]] }	| REFERENCES reftable [(refcolumn [,...])] [ON DELETE { CASCADE | SET NULL }] }
	[[NOT] DEFERRABLE] [INITIALLY { DEFERRED | IMMEDIATE }]	[[NOT] DEFERRABLE] [INITIALLY { DEFERRED | IMMEDIATE }] [RELY | NORELY] [using_index_clause] [ENABLE | DISABLE] [VALIDATE | NOVALIDATE] [exceptions_clause]

Table B.7.: Comparison of SQL Create Table Statements of Various Vendors

1. This is an incomplete description of Oracle's CREATE TABLE statement. Only the comparable parts to MySQL and PostgreSQL are presented here.
2. The ON UDPATE and ON DELETE action can be one out of NO ACTION, RESTRICT, CASCADE, SET NULL, SET DEFAULT(only for PostgreSQL).

Bibliography

[Bac12] T. Bachmann. Entwicklung einer Methodik für die Migration der Datenbankschicht in die Cloud. Diploma Thesis No.3360, Institute of Architecture of Application Systems, University of Stuttgart, 2012.

[Bar] N. Bartlett. A Comparison of Eclipse Extensions and OSGi Services. `http://www.eclipsezone.com/articles/extensions-vs-services/`.

[BED01] J. Bowman, S. Emerson, and M. Darnovsky. *The Practical SQL Handbook: Using SQL Variants*. Addison-Wesley, 4th edition, 2001.

[CAB+81] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost. A History and Evaluation of System R. *Commun. ACM*, 24(10):632–646, October 1981.

[Cha04] D. A. Chappell. *Enterprise Service Bus*. O'Reilly Media, 2004.

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, June 1970.

[Con] Continuent, Inc. Continuent Tungsten Connector. `http://sourceforge.net/apps/mediawiki/tungsten/index.php?title=Introduction_to_the_Tungsten_Connector`.

[dbe] DB-Engines Ranking. `http://db-engines.com/en/ranking`.

[goo] Google Cloud SQL. `https://developers.google.com/cloud-sql/`.

[GW02] J. R. Groff and P. N. Weinberg. *SQL: The Complete Reference*. McGraw-Hill/Osborne, second edition, 2002.

[iso] ISO/IEC 9075-1:2011 - Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework). `http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=53681`.

[jdb] JDK 6 Java Database Connectivity (JDBC)-related APIs & Developer Guides. `http://docs.oracle.com/javase/6/docs/technotes/guides/jdbc/`.

[KKH08] K. E. Kline, D. Kline, and B. Hunt. *SQL in a Nutshell*. O'Reilly, third edition, 2008.

[LN12] T. Laszewski and P. Nauduri. *Migrating to the Cloud : Oracle Client/Server Modernization*. Syngress, 2012.

[MF11] P. Mell and T. France. The NIST Definition of Cloud Computing. National Institute of Standards and Technology, 2011.

[Muh12] D. Muhler. Extending an Open Source Enterprise Service Bus for Multi-Tenancy Support Focusing on Administration and Management. Diploma Thesis No.3226, Institute of Architecture of Application Systems, University of Stuttgart, 2012.

[mys] MySQL Documentation: MySQL 5.6 Reference Manuals. `http://dev.mysql.com/doc/refman/5.6/en/index.html`.

[ora] Oracle Database SQL Language Reference 12c Release 1. `http://docs.oracle.com/cd/E16655_01/server.121/e17209/toc.htm`.

[OSG] OSGi Alliance. The OSGi Architecture. `http://www.osgi.org/Technology/WhatIsOSGi`.

[OSG11] OSGi Alliance. OSGi Service Platform Core Specification. Release 4, Version 4.3, 2011.

[OSG12] OSGi Alliance. OSGi Service Platform Service Compendium. Release 4, Version 4.3, 2012.

[OV11] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Springer, third edition, 2011.

[Par13] T. Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

[pos] PostgreSQL 9.3.0 Documentation. `http://www.postgresql.org/docs/9.3/interactive/index.html`.

[RD09] T. Rademakers and J. Dirksen. *Open Source ESBs in Action*. Manning Publications Co., 2009.

[Sáe12] S. G. Sáez. Integration of Different Aspects of Multi-tenancy in an Open Source Enterprise Servis Bus. Student Thesis No.2394, Institute of Architecture of Application Systems, University of Stuttgart, 2012.

[Sáe13a] S. G. Sáez. Extending an Open Source Enterprise Service Bus for Cloud Data Access Support. Diploma Thesis No.3419, Institute of Architecture of Application Systems, University of Stuttgart, 2013.

[Sáe13b] S. G. Sáez. Manual for the CDASMix Initialization, 2013.

[SL90] A. P. Seth and J. A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3), 1990.

[Thea] The Apache Software Foundation. Apache Felix Service Component Runtime (SCR). `http://felix.apache.org/documentation/subprojects/apache-felix-service-component-runtime.html`.

[Theb] The Apache Software Foundation. Apache ServiceMix Documentation. `http://servicemix.apache.org/docs/4.5.x/index.html`.

[THW05] R. Ten-Hove and P. Walker. JSR 208, Java Business Integration (JBI) 1.0. Technical report, Sun Microsystems, Inc, 2005.

[Tra13] Transaction Processing Performance Concil. TPC BenchMark H Standard Specification Revision 2.16.0, 2013.

All links were last followed on November 25, 2013

Acknowledgement

I am heartily thankful to my supervisor Steve Strauch from the University of Stuttgart for his encouragement, guidance and support in all the phases of this thesis. I'm also grateful to my college Santiago Gómez Sáez for his helpful advices.

Simin Xia

Declaration

I hereby declare that the work presented in this thesis is entirely my own. I did not use any sources and references other than those listed. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Stuttgart, 26 November 2013 _____

(Simin Xia)