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

Master's Thesis No. 0838-003

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

Alketa Ramaj

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

Examiner: Prof. Dr. Dr. h. c. Frank Leymann
Supervisor: Steve Strauch
Dr. Vasilios Andrikopoulos

Commenced: April 23, 2015
Completed: October 23, 2015

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

Abstract

Cloud computing has enabled a new era in the IT industry and many organizations are interested in moving their business operations to the Cloud. This can be realized by designing new applications that follow the prerequisites of the Cloud provider or just by migrating the existing applications to the Cloud. Each application follows a multi-layered architecture defined by its design approach. Application data is of utmost importance and it is managed by the data layer, which is further divided into two sublayers, the Data Access Layer (DAL) and the Database Layer (DBL). The former abstracts the data access functionality and the latter ensures data persistence and data manipulation.

Application migration to the Cloud can be achieved by migrating all layers it consists of or only part of them. In many situations it is chosen to move only the DBL to the Cloud and keep the other layers on-premise. Most preferably, the migration of the DBL should be transparent to the upper layers of the application, so that the effort and the cost of the migration, especially concerning application refactoring, becomes minimal. In this thesis, an open source Enterprise Service Bus (ESB), able to provide multi-tenant and transparent data access to the Cloud, is extended with PostgreSQL transformation functionality. Previously the ESB could only support MySQL source databases. After the integration of two new components, a PostgreSQL proxy and a PostgreSQL transformer, we provide support for PostgreSQL source databases and dialects. Furthermore, we validate and evaluate our approach based on the TPC-H benchmark, in order to ensure results based on realistic SQL statements and appropriate example data. We show linear time complexity, $O(n)$ of the developed PostgreSQL transformer.

Contents

List of Figures

List of Tables

List of Listings

1. Introduction

Cloud computing is a term to communicate the concept of virtualizing the computing power, while hiding its internal structure and operations. The term was initially used by engineers, back in the early years of network design, to refer to the unknown parts of the network [Lou10]. Computing power is built of discrete components, including processing, data storage, and software resources. All of them cooperate and give a single, optimized computing package, which can be provided as a utility, so called *"on-demand"* delivery. Customers can obtain computing in a way similar to electricity, water or telephony, and in such a way pay providers based on what they use, so called *"pay-as-you-go"*. The main objectives of Cloud computing are to increase resource utilization rates, to centralize management and maintenance of the systems, and to offer computing power as a service at lower cost.

However, transferring data to Cloud yields a series of incompatibilities between the application and the database. A challenge nowadays is to isolate the application and enable it to simply call the database service without considering the incompatibilities generated after the data migration. This idea is encapsulated in the *"Database as a Service"* paradigm, which attempts to provide seamless scaling of database resources, back-ups, server failure handling, and provisioning. The ultimate goal is to provide this without affecting the front-end in any way. The current work focuses on a sub-area of Database as a Service (DBaaS). Specifically, it resolves the incompatibility of the query language, which occurs when the source and target database use different query languages.

1.1. Problem Statement

Nowadays in the market there are plenty of vendors offering several types of Cloud services. Choosing the Cloud solution which meets the requirements of an organization and at the same time reduces its expenditures to the maximum, seems ambiguous and hard. Bachmann developed a Cloud Data Migration Methodology and a Tool [Bac12]. The proposed migration and application refactoring methodology is based on a work by Laszewski [LN12] and is composed of seven phases as shown in Figure 1.1.

This methodology involves specific questionnaires for supporting the decision making process to find the most suitable database. In the pre-migration step, it also adapts the Data Access Layer (DAL) in order to reduce the compatibility issues, which are going to appear after the migration, as a result of the differences among the source and target databases. However, in his work, Bachmann does not treat the remaining incompatibilities that occur after the data migration.

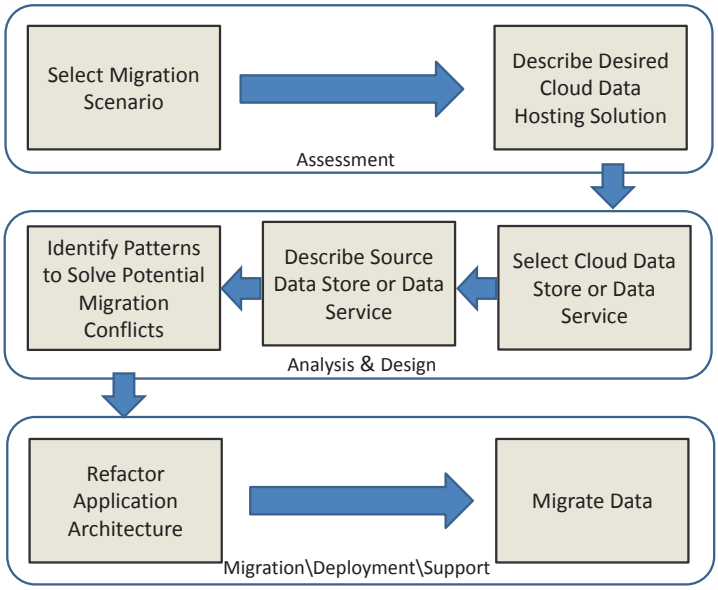

Figure 1.1.: Cloud Data Migration Tool [Bac12, SAK+14, SAKVH15]

Apart from addressing the compatibility issues, an additional goal of our system is to achieve this with minimum impact on the application side. To serve this objective, Goméz Sáez extended an open source Enterprise Service Bus (ESB) for Cloud data access and multi-tenancy support [GS13]. It acts as a middleware between the application and the local (on-premise) or Cloud (off-premise) databases. The communication inconsistencies among different Cloud services are significantly reduced. Furthermore, this prototype ensures the transparency the Data Layer provides to the Business and Presentation Layers of the application, namely, it reduces the required changes made to the application[1].

However, the aforementioned solution is not sufficient when the source and target data sources are not built based on the same database system. Incompatibilities related to the variations of the SQL dialects (syntaxes) remain unsolved. Oracle's SQL Developer tool for migration, offers the possibility to convert the SQL statements on the application side in a way that they will be compatible with the Oracle target database system [LN12]. One drawback of this solution is that it is dedicated to the Oracle target data sources and therefore, it needs to be extended to support also other dialects. The biggest problem though, is that it does not provide database transparency to applications. The goal must be to minimize application changes when migrating data to the Cloud, in order to reduce the overall migration time and effort.

New technologies allow to take the SQL statements addressed at the source database and transform them, before forwarding them to the target database, for execution. Xia extended on the work by Goméz Sáez further, by enhancing the open source ESB for Cloud data Access with the capability of transforming SQL statements [Xia13]. The SQL statements used in

[1] In this work we refer to applications that follow the three-tier architecture [app]. However, our system is compatible also with applications that follow other design approaches

the presentation and business layers of the application are passed to the middleware, where they are analyzed by the parser and translated into a hierarchy of Java classes. The SQL transformer then reconstructs the Java classes into a new set of statements compatible with the target data store dialect. Xia enhanced the already existent MySQL Proxy with the ability to interact with a newly added component, the SQL transformer, which receives SQL statements and further processes them in order to make them understandable to Oracle and PostgreSQL target databases.

The work of Xia addresses only MySQL source dialects. Hence, he worked on the MySQL proxy and MySQL transformer in order to make it possible to translate MySQL statements into Oracle or PostgresSQL statements. However, he does not handle other cases of source databases, such as PostgreSQL. This work aims at developing and integrating a PostgreSQL proxy and a PostgreSQL transformer into the existing system, in order to support parsing and transformation of PostrgreSQL statements into Oracle and MySQL statements.

1.2. Motivating Scenario

There are numerous financial, organizational, and technological reasons why an enterprise chooses to migrate its application to the Cloud. There are also plenty of Cloud services from many different vendors. Each of them offers its own special features and aspires to be the best solution. [ABLS13] discusses the different types of application migration. The enterprise organization, which decides to migrate its application to the Cloud, tries to maximize its benefits in all three aforementioned directions. To achieve this it may be required to allocate its applications' data among different Cloud storage providers. It may also be required that some data, the most frequently used and/or confidential data, to remain on-premise. On the other hand, the organization may also choose to migrate its whole application into the Cloud. The latter dramatically increases the migration time and effort and is not a case considered by this work. Our system concentrates on the migration of the Data Layer (DL) and its two sublayers, the Database Layer (DBL) and the DAL, while the rest of the application remains unchanged and on-premise.

Inspired by the notation used in [Moh11], we use the following representation to capture the migration of an enterprise application:

$$D \rightarrow D_C + D_L \rightarrow D_{OM} + D_L \tag{1.1}$$

where D is the state of the application before migration, while its DL was on-premise. D_C is the part of the application data moved to the Cloud and D_L is the data remaining in the local database. If the DL is fully migrated, then D_L is zero, as no data exists on-premise. The data on the Cloud can reside in a single storage provider or can be allocated in an optimized fashion, across different providers, in order to maximize the overall benefits for the organization. D_{OM} expresses the data being optimally migrated and is the aggregation of data among many

different Cloud storages. D_{OM} is described as follows:

$$D_{OM} \rightarrow D_{C1} + D_{C2} + ... + D_{CN} \tag{1.2}$$

where $D_{C1} + D_{C2} + ... + D_{CN}$ are the different alternatives of Cloud services used. The dispersion of data in Cloud enables an enterprise to take advantage from the different providers by using their services in a dynamic and flexible way. It is unlikely though, that all of them will be compatible with the source database. Thus, mapping a single source database into multiple, diverse database systems is the main reason why incompatibilities often appear after migration. Our goal is to address a specific subset of such incompatibilities, related to the differences in SQL dialects. Vendors, in order to acquire a larger market share, like to implement new features that will differentiate them from their competitors. For example, one of the many differences between PostgreSQL and MySQL dialects, is that PostgreSQL is case sensitive, while MySQL is not, unless the "binary" flag is set. Therefore, an intermediate layer is required, for a unified and transparent access of the applications to the Cloud and local databases, as shown in Figure 1.2. Xia implemented the seamless transformation required when migrating data from MySQL source data store into Oracle and PostgreSQL data stores. This thesis continues by handling the PostgreSQL source databases.

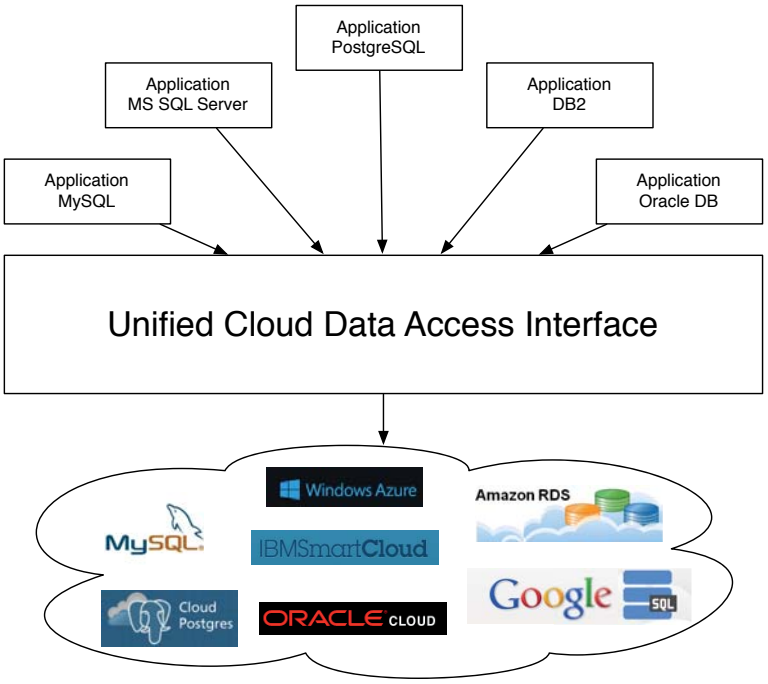

Figure 1.2.: Motivating Scenario [Xia13].

1.3. Definitions and Conventions

In the following section a list of abbreviations used in this thesis is provided, to further the understanding of this document.

List of Abbreviations

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

AWS	Amazon Web Services
BC	Binding Component
CDASMix	Cloud Data Access Support in Multi-Tenant ServiceMix
CRM	Customer Relationship Management
DAL	Data Access Layer
DBaaS	Database as a Service
DBMS	Database Management System
DBL	Database Layer
DBS	Database System
DL	Data Layer
ESB	Enterprise Service Bus
FDBS	Federated Database System
GAE	Google App Engine
IaaS	Infrastructure as a Service
IBM	International Business Machines Corporation
JBI	Java Business Integration
JBIMulti2	JBI Multi-tenancy Multi-container Support
JDBC	Java Database Connectivity
JMS	Java Message Service
JNDI	Java Naming and Directory Interface
MDBS	Multi-database System
MEP	Message Exchange Patterns
NIST	National Institute of Standards and Technology
NM	Normalized Message
NMF	Normalized Message Format
NMR	Normalized Message Router
OSGi	Open Services Gateway initiative *(deprecated)*
PaaS	Platform as a Service

RDBMS	Relational Database Management System
SA	Service Assembly
SaaS	Software as a Service
SCR	Service Component Runtime
SE	Service Engine
SOA	Service-Oriented Architecture
SQL	Structured Query Language
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
UUID	Universally Unique Identifier
WS	Web Service
WSDL	Web Services Description Language

1.4. Outline

The rest of this document is structured as follows:

- **Chapter 2. Fundamentals** discusses some fundamental and background information that is required for the understanding of this thesis.
- **Chapter 3. Related Work** provides the state of the art in topics that are the cornerstone of our work and positions them towards the work in this thesis, respectively.
- **Chapter 4. Analysis and Specification** represents a detailed analysis of the system overview, functional and non-functional requirements that our components must meet, as an enhancement of the Multi-tenant ServiceMix with Cloud Data Access Support.
- **Chapter 5. Design** describes and illustrates the internal structure of the new system components and their interaction with the other components of ServiceMix. The modifications and extensions required for the parsing and transformation of SQL statements are thoroughly described.
- **Chapter 6. Implementation** describes and illustrates our implementation of the PostgreSQL proxy and PostgreSQL transformation components, the implementation of a PostgreSQL parser using JavaCC, and the integration of the components into the existing system.

- **Chapter 7. Validation and Evaluation** presents the validation and evaluation of our implementation, and discusses the results.
- **Chapter 8. Conclusion and Future Work** concludes this thesis and discusses some directions of further development in the future.

2. Fundamentals

In this chapter background information and fundamental knowledge is provided, necessary for a sufficient understanding of the existing system and the objectives of this work.

2.1. Roots of Cloud Computing

Achievements in several fields, including hardware, Internet technologies, distributed systems and their enterprise-wide administration, prepared the ground for the advent of Cloud computing. This section provides a view on the step-by-step emergence of the Cloud through its predecessor technologies.

2.1.1. Mainframes

The need for access to a large amount of computing power and also for acquiring that power on-demand (Utility computing), has been present since the start of computing. In 1964 International Business Machines Corporation (IBM) came with a new generation of electronic computing equipment, named IBM System/360. IBM Board Chairman Thomas J. Watson promised "*more computer productivity at lower cost than ever before*" [Boy04]. In the early 1970's, mainframes evolved to time-shared machines, able to serve hundreds of users simultaneously. The arrival and rapid progress of low-cost computers based on integrated circuits made the mainframes an old-fashion technology. However, mainframes remain to be the first important step towards some of the features encapsulated nowadays by Cloud technology, such as the illusion of infinite resources, created by their aggregation, and their on-demand delivery.

2.1.2. Web Services and SOA

The next significant station towards Cloud computing can be considered to be SOA, firstly defined by Sun in the late 1990's [Qus05]. With SOA, enterprise IT is composed of software components, packaged as services [VBB11] that interact with each other over a network, in a loosely-coupled way, ideally by using WS standards [VBB11, Qus05]. These are the preferred standards used to implement SOA, as they can facilitate the communication over a network of applications running on different messaging platforms. But they should be distinguished from the SOA itself, which is the architecture paradigm and not a specific technology that can be used to implement this architecture [Qus05, WCL+05].

Both SOA and WS can provide the backbone to Cloud development. According to [KB] SOA is not a requirement for Cloud computing and vice-versa, instead they are complementary to each other. However, according to *The Open Group* definition, Cloud services are SOA services, but not all SOA services are Cloud service. A set of standardized characteristics that must be enabled simultaneously for the Cloud are just optional in SOA [BGK+11]. Either way, the pre-existing SOA gives the Cloud a well-defined and robust architecture that considerably simplifies the maintenance and migration procedures [Amo14]. Differently from SOA, WS are always part of any Cloud infrastructure and application. They are the glue used for connecting applications together in order to achieve software integration [KB]. Figure 2.1 depicts the relationships and interconnections among these three technologies: WS, SOA, and Cloud computing, with a Venn diagram.

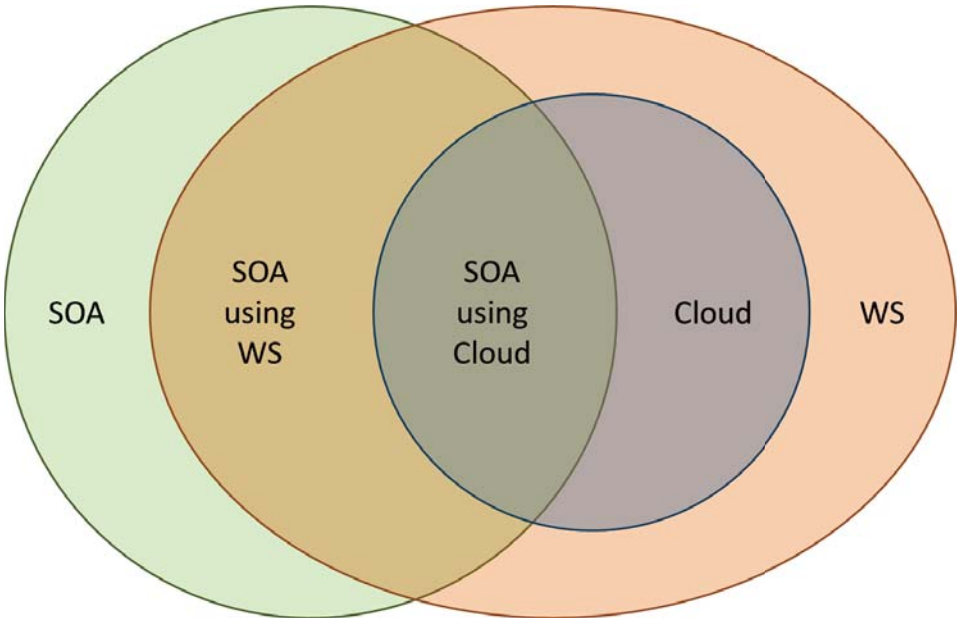

Figure 2.1.: Relationships Among WS, SOA, and Cloud Computing [KB].

2.1.3. Grid Computing

Grid computing aggregates distributed resources and forms a big infrastructure that operates as a single virtual system. Resources can be provisioned as a utility, which is switched on or off [Mye09]. Its successor, Cloud technology, adopts this idea of resource provisioning and extends it to "on-demand" resource provisioning. The access to the common pool of resources is not flat, but it can be dynamically adjusted according to a customer's needs, who pays for what she/he uses (Utility computing) [VBB11]. As a newer and more highly evolved technology, Cloud computing has a set of other advantages compared to Grid computing. Important differences come from the fact that a Cloud has evolved to make use of SOA and virtualization technology. The absence of SOA in Grid computing is the common case.

For this reason, Grid technology usually provides concrete services, such as CPU, network, memory, bandwidth etc., instead of abstract computing service types provided by a Cloud.

2.1.4. Virtualization

Virtualization of physical computing resources, such as network, server and storage, is the creation of their virtual versions, in order to ease their shared and flexible use among many users. *Network virtualization* allows the sharing of network bandwidth among different VLANs (Virtual LANs), while *server virtualization* results in multiple operating systems' images, known as virtual machines (VMs), running on the same server. *Virtualization of storage* refers to the merging of data from multiple types of storage devices into a "single storage unit" that can be managed centrally [LN12]. The virtualization model that involves the two latter types of virtualization is named *hardware virtualization*. The use of hardware virtualization in Cloud deployments improves sharing and utilization, as it gives the users the possibility to dynamically get resources and pay only for the amount used. It also leads to better manageability, by simplifying the monitoring of resource consumption and subsequently, to higher flexibility, by better management of workloads in VMs. Different VMs running on the same server are mutually isolated from each other (workload isolation). This increases the reliability, as failures at one VM will not affect the others and will not propagate through the whole Cloud infrastructure. Workload isolation has benefits regarding workload migration, known as application mobility, as well. Migration can extremely simplify hardware maintenance and disaster recovery [VBB11].

2.2. Cloud Models

Over the years, many attempts have been made to give a precise definition of Cloud computing and its unique characteristics by academia, industry, and governmental labs. According to the National Institute of Standards and Technology (NIST) [MF11]: "*Cloud computing is 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. This Cloud model is composed of five essential characteristics, three service models, and four deployment models.*" The five essential characteristics of Cloud computing are (1) on-demand self-service, which enables the users to provision the required resources automatically and at run time; (2) broad network access, which means that the resources hosted on the Cloud are available over the network; (3) resource pooling, which refers to resource aggregation in order to provide transparent access to phenomenally infinite and directly available resources; (4) rapid elasticity or in other words scalable provisioning, allows users to automatically request additional resources; and (5) measured service, denoting the possibility of transparent monitoring, controlling, and reporting of resource use.

The Service models given by NIST are Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). Through these three service models Cloud

computing is discussed in terms of creation, delivery, and consumption of Cloud services, and subsequently it is interconnected to the notions of SOA and described as part of it [BGK+11].

SaaS is a model where an application is running on a remote data center and provided as a service to customers across the Internet. The provider is responsible for software development, maintenance and upgrades, while the customer has no control of the underlying Cloud infrastructure. Google Mail and Google Docs, as well as social media, including Facebook, LinkedIn, Flickr, and Twitter are just some of the widely known cases of SaaS, accessible with no limitation from several client devices through Web browsers. Enterprises also resort to putting applications on to the Cloud for a variety of services, such as accounting, business management, collaboration, marketing, Customer Relationship Management (CRM), communication and many others. Some widely used SaaS applications serving the aforementioned goals are Concur Technologies, Inc., NetSuite, GoToMeeting, Constant Contract, Sales Force Automation, and Google Mail, respectively.

PaaS provides users with the capability to deploy (build and run) their own applications onto the Cloud using programming languages, libraries, services and tools, given and maintained by the Cloud provider. Same as in SaaS, also in PaaS users have no control of the underlying Cloud infrastructure, namely the middleware, operating system, virtualization, servers, storage, and networking. However, this service type gives more freedom to the users by allowing them to manage their own applications on Cloud and to configure the application-hosting environment. Amazon Web Services (AWS) and Salesforce.com were the pioneers of PaaS. Google App Engine (GAE), Apprenda, Red Hat OpenShift, Windows Azure Web Services and many others followed.

IaaS provides users with the capability to provision some fundamental computing resources, such as servers, storage, networks, and networking services (e.g. firewalls). The offered servers can be virtualized, when the main requirement is not the performance, but highly dynamic workloads. Otherwise, IaaS may offer so called bare-metal servers, in which the hypervisor layer is not required, because no resource sharing takes place. SoftLayer, a subsidiary company of IBM, is one of the few Cloud providers who offers bare-metal Cloud computing instances aside IaaS virtual servers. Other famous providers of IaaS are AWS, IBM SmartCloud Enterprise, Google Compute Engine. Figure 2.2 visualizes the relationship of service models and the contribution, as well as responsibility, they require by the providers and the customers, respectively.

The deployment models of the Cloud are (1) Private Cloud, which is dedicated to a single organization and can be owned and managed by the organization itself, a third party, or the combination of both; (2) Community Cloud, referring to the case where the Cloud infrastructure is to be used by a specific group of organizations with the same requirements and/or purpose (it also can be owned and managed by some of the organizations of the group/community, a third party or by some combination of them); (3) Public Cloud, a term for describing the situation when the Cloud infrastructure can serve the general public and (4) Hybrid Cloud, which can be any combination of two or more different aforementioned models [MF11].

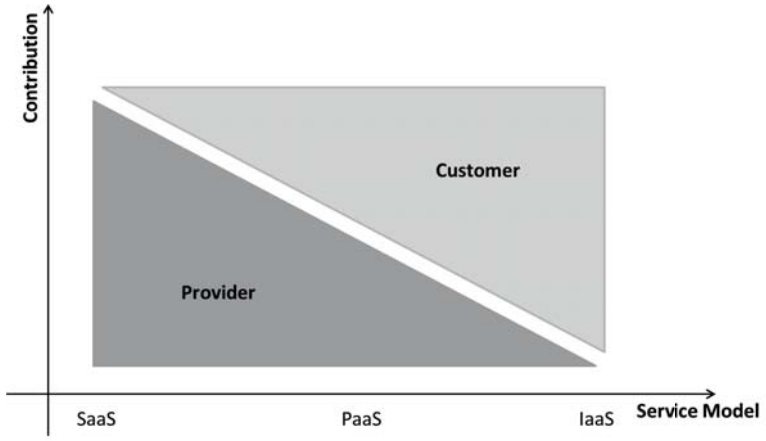

Figure 2.2.: Service Models and Their Relation to the Contribution and Responsibilities of Providers and Customers

2.3. Relational Databases

In [Cod70], Codd brings up the idea of *data transparency*, while talking about the necessity to hide the way data is represented in the machine from future users of large data banks. He realizes that changes on the internal data organization should not affect activities of users at terminals and application programs. As shown in Figure 2.3, Codd used the mathematical theory of relation to represent data in a table, known as *relation R*, which is composed of rows and columns, respectively named tuples and attributes [Cod70].

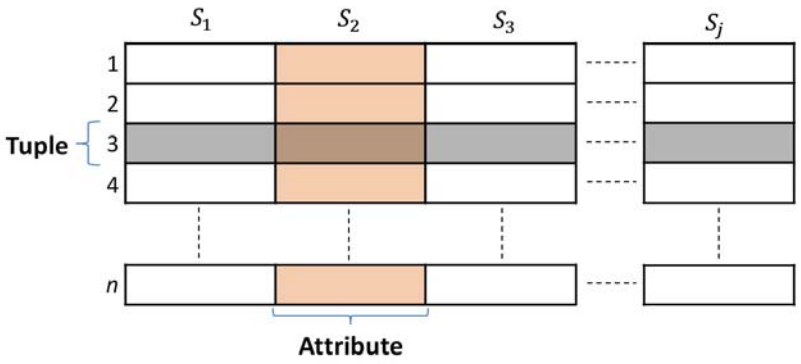

Figure 2.3.: Relational Model of Data [Cod70].

Specifically, data is considered to be grouped in attributes, or sets (S_1, S_2, S_3,..., S_j), and the relation of those sets is a set of n-tuples, each of it having its first element from component S_1, second element from component S_2 and so forth. Each n-tuple is unique and their order is not of importance. Attributes, on the other hand, can be similar, but their order is significant. With time the need to alter the information stored in the relation can arise. This is achieved by deleting existing tuples, adding new ones, or changing the components of a tuple. The entire

information of a large data bank can be considered a collection of time varying relations. Codd's relation model sets up the basis for a high level data language, able to manipulate and retrieve data stored on relations. For this purpose, the first version of the Structured Query Language (SQL) was developed [KKH08].

2.4. SQL Dialects

SQL is the standard computer database language for relational database management systems. Although it is standardized since 1986, various deviations of it are implemented and extensions of it are still being developed by different vendors. Most database vendors follow the product lock-in policy: they add their own features in order to bind their customers and render them unable to use another vendor without reasonable switching costs. This results in a wide variety of SQL dialects. On the other hand, writing portable SQL statements, namely such that perform well in multiple databases, is a must for applications that work with a range of different SQL servers. This work aims at solving SQL incompatibility problems, generated from the various server implementations and subsequently, the several dialects. It does not try to implement portability at the application level by defining restrictions on the SQL queries construction. Instead, a middle layer is added, responsible for parsing the SQL statements of a source dialect and the transformation of them into a target dialect.

The SQL standard compacts all features that are shared among the different implementations and define the core functionality of SQL. The following, are implementations of the most known deviations of SQL standard versions.

- PostgreSQL[1]: It is considered the most feature-oriented Relational Database Management System (RDBMS). It supports many built-in data types and built-in procedures, including math operations, string operations, and cryptography.
- MySQL[2]: It is flexible to operate in demanding environments, such as Web applications, and it can empower embedded applications, data warehouses, highly available redundant systems, etc. [SZT12]. It is considered the most performance-oriented implementation.
- Oracle Database[3]: It is the most popular and commercial database management system, because it is easy to use. It offers an elegant and fast architecture for handling and maintaining data.

2.4.1. PostgreSQL vs. MySQL

Businesses around the world commonly use the two leading open source relational Database Management System (DBMS), PostgreSQL and MySQL. In this document we mainly focus on the first one, however this subsection will provide a brief comparison of both.

[1] http://www.postgresql.org/
[2] http://www.mysql.com/
[3] http://www.oracle.com/technetwork/database/database-technologies/sql/overview/index.html

The most important feature of MySQL is its storage-engine architecture, the design of which separates query processing and other server tasks from data storage and retrieval. This separation lets one choose how the data is stored and what performance is to be achieved [SZT12]. For this reason MySQL has been considered for years a performance-oriented implementation. On the contrary, PostgreSQL was developed with a focus on features and standards. Thus, PostgreSQL was often considered as the most standard compliant and feature-oriented RDBMS, but slower than MySQL. Apart from plenty of supported built-in data types such as Boolean, Circles, Lines, IPv4, IPv6, and built-in procedures, it offers a better security model as well. It provides more efficient support for external authentication, security groups, and built-in SQL injection attack defenses. Its main drawbacks compared to MySQL is its lower speed and a relatively more challenging set-up and use.

Aside from the differences in architectural characteristics, data types and implemented procedures, there are also many deviations in the syntax of MySQL and PostgreSQL. In the Appendix of [Xia13] an overview of a thorough comparison of data types and functions of these two implementations of SQL is given. Here, we will point out their main syntactical and structural differences [Eis03, com]:

1. PostgreSQL uses the ANSI standard '--' (double dash) to begin a comment line. Instead, MySQL can also begin it with '#'.

2. PostgreSQL accepts only single quotes for quoting its values, while MySQL also allows double quotes.

3. In PostgreSQL double quotes are used for quoting system identifiers, such as table names, field names etc. MySQL uses the non-standard '`' (accent mark) instead.

4. PostgreSQL is case-sensitive for the former, but it is not for databases, fields, tables, and column names. Disanalogously, MySQL is case-independent for string comparisons, if the binary flag is not set, while for the latter it can be case-sensitive or not, depending on the used operating system.

5. The C-language operators for boolean logic have the same semantics in MySQL, but are not consistent with the database standards. PostgeSQL follows the standards and so, for example the `OR` operator '||' is used fo string concatenation in PostgreSQL.

6. Several constructs like `DESCRIBE`, `REPLACE`, `SHOW`, `USE`, `UNLOCK TABLE`, are not implemented in PostgreSQL.

7. `DROP TABLE IF EXISTS <table>`, is always available in MySQL, but can only be found in the later versions of PostgreSQL (after 8.2).

8. `SELECT ... INTO OUTFILE <filepath>` in MySQL is formulated as `COPY (SELECT ...) TO <filepath>` in PostgreSQL.

9. MySQL accepts both formulations `SELECT ... LIMIT <offset>, <limit>` and `SELECT ... LIMIT <limit> OFFSET <offset>`, while in PostgreSQL only the latter is valid.

10. PostgreSQL also allows procedural languages like Perl and Python, to create functions in the database, instead of coding in the Web front end.

2.5. CDASMix and its Architectural Components

This section introduces Cloud Data Access Support in Multi-Tenant ServiceMix (CDASMix), the system that this work aims at enhancing with additional functionalities. CDASMix is an extension of the multi-tenant aware version of Apache ServiceMix 4.3.0 [Muh12, GS13], an open source version of ESB. The presentation of CDASMix will be done gradually, while going through the main technologies that support it.

2.5.1. Web Services in CDASMix

Web services are a form of distributed information systems. Their definition by W3C states that "*Web service is a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web service supports direct interactions with other software agents using XML-based messages exchanged via Internet based protocols*". Hence, the Web Services Description Language (WSDL) uses XML format and this simplifies data storage and sharing, regardless of the heterogeneity of the involved systems (or service units). WSDL files define services as collections of network endpoints, or ports.

2.5.2. JBI and OSGi

The version of Apache ServiceMix used in this project is compliant with both Java Business Integration (JBI) and OSGi specifications. JBI is developed for implementing a SOA, based on a Web services model. It provides a pluggable architecture for a container that hosts service producer and consumer components. These components interact with each other by using WSDL. The WSDL files are stored in Service Assembly (SA). The SA includes metadata for "wiring" the service units together, by associating service providers and consumers, as well as for "wiring" them to external services. The central message delivery mechanism, the Normalized Message Router (NMR), delivers normalized messages via one of four Message Exchange Patterns (MEP). This provides a simple mechanism for performing composite application assembly using services. Each service can connect to the container via Binding Component (BC). This provides protocol independent communication among services. BC put the services that use a specific protocol, such as HTTP or SOAP, into the JBI NMR after converting their messages from their specific protocol to XML. This process is called *normalizing* and allows other JBI components to access these messages from the NMR. The result is that any JBI component is able to communicate over any protocol supported by the binding components deployed to the JBI runtime environment. Thus, JBI turns out to be an efficient integration solution over different applications, in a vendor independent way.

OSGi is a new platform for application development. More specifically, it is a general purpose Java framework that makes Java the leading environment for software integration. Java, regarding the code, its libraries (if they are written in pure Java), the runtime environment, and the JVM is platform independent. Hence, Java provides the needed portability to support applications on many different platforms. The OSGi framework, on the other hand, provides the standardized primary elements, which allow applications to be constructed dynamically from small, reusable, and collaborative components, so called bundles. An OSGi bundle is a Java module in a JAR file, which consists of a package of Java classes and a JAR manifest file, named `META-INF/MANIFEST.MF`. The manifest file is used to define the extension and package related data, e.g. dependency information. OSGi platforms facilitate easy and dynamic deployment and undeployment of the bundles. Deploying a bundle and making it able to join the already collaborating applications that are currently running in the OSGi container, is done simply through dropping the bundle in the deploy directory.

2.5.3. ESB

ESB is a message oriented technology that attempts to "relax" tightly coupled communication with the introduction of an intermediate component. In this way software components, which can be considered as discrete (software) systems, can communicate indirectly with each other. ESB is also used for effective system integration. Integration is considered as the "*fascia of the enterprise*" [Jak], which allows multiple systems to coordinate together in order to perform one single task. CDASMix is such an aggregation of different systems with the goal of allowing transparent data access traditionally, on-premise, and off-premise.

Effectiveness of integration is highly related to data resiliency, especially in the face of failure. Data resiliency is challenging in case of distributed systems (such as CDASMix), where data can be found not only at "rest" but also in "motion". Resilient data in motion asks for novelty in code and infrastructure. Attempts for extending the existing programming abstractions are going on [Mir]. Their target is to support not only the individual, passive values, but also the changing data sets, as they result from events involved in reactive behavior, in memory collectors or from data scattered over the Internet. The plenty of ESB implementations, on the other hand, focus on offering data resilience through infrastructure. In this project ESB serves as a container that allows to run the Apache Camel, an implementation of the idea called "Enterprise Integration Patterns". Especially, in this project, the ESB technology is given by the Apache Karaf, a lightweight container in which Apache Camel runs.

2.5.4. Apache Karaf and Apache Camel

Apache Karaf, as an OSGi based runtime system, offers hot deployment of the services, as bundles. The bundle simply needs to be dropped in the deploy directory and Apache Karaf will automatically resolve the type of the file and start its deployment. It allows also the deployment of non-OSGi applications. Hence, it becomes a flexible and easily extendible container. Apache Karaf, comes with a complete UNIX-like console, through

which a container instance can be fully managed. Moreover, multiple instances of the Apache Camel container can be managed directly from a root instance.

Apache Camel is an open source routing engine that runs in Apache Karaf. It is a JBI component deployed as an OSGi bundle. The total data for describing a particular message flow is encapsulated into the concept of exchange. A Camel exchange is a holder object that keeps the state of a conversation between systems. As shown in Figure 2.4, the exchange can support various MEP. e.g. InOnly-, OutOnly- or InOut-exchange patterns. The two former are "one-way event" messages, while the latter is a "request-reply" message exchange [The]. Other main field of an exchange are: the *Unit of Work*, a set of flags, the *Properties*, which are available for entire duration of exchange, and two messages, *In-message* and *Out-message*. In-message is a mandatory field which contains the input message and Out-message is optional message which exists only if MEP is InOut. Each message contains the data payload to be processed during the route and the corresponding headers, used to pass additional information about the message between the processors of the route. Processors are another architectural concept in Camel, serving as the base interface for the message-processing steps of the route [RD09].

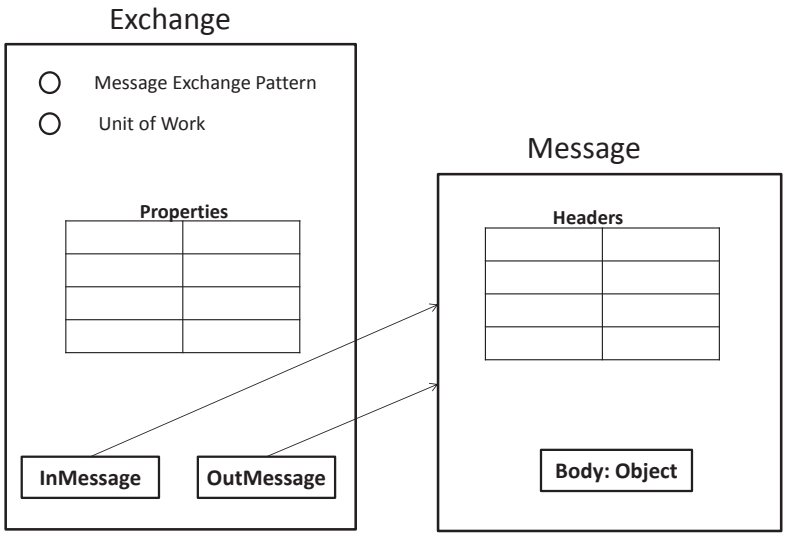

Figure 2.4.: Structure of Exchange [Jak]

Headers and also *Properties* are represented as a map (Strings to Objects). Based on the concept of exchange, in Apache Camel, each step that the message payload will go through, is architecturally independent from each other, as they do not invoke each other. A number of (processing) steps defines the route. The very initial step, which creates the exchange, is the consuming endpoint and is described as a simple Uniform Resource Identifier (URI). URI can appear as Uniform Resource Locator (URL) or Uniform Resource Name (URN). URI is a specification of a string of characters used to identify a Web resource by its name or its location. In the first case it can be called as URN and in the second as URL. In Object-Oriented

Programming (OOP), URI can be defined as a class implementation, while URN and URL, as subclasses of it. This is depicted as a class diagram of their three, in Figure 2.5.

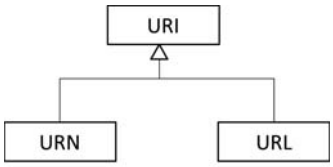

Figure 2.5.: URI Class Diagram [CK13]

After the consuming endpoint, the exchange will be passed by the Camel engine through the next steps of the route and in each step, the exchange is manipulated. This includes In-message modification or preparation of a new payload, which is set on the Out-message. In the latter, (when Out-message is set) the Camel context will move it to the In-message of the exchange before passing it to the next processing step. The Camel context is defined as the engine that handles the process of the exchange along the sequential steps within the route. Camel Context offers a loose coupling between the steps by using URIs for referring to endpoints which are created by components (endpoint factories) within routes [CK13].

2.5.5. Architecture Overview

This section discusses the architecture of CDASMix and provides a deep insight of its main components. CDASMix is an implementation approach, which provides transparent Cloud data access support for migrated data and allows for multi-tenancy, through tenant isolation, and diversity among involved database technologies, through dynamic query transformation functionality.

The system architecture of the most up-to-date version of CDASMix is shown in Figure 2.6. It is developed as an OSGi container with JBI integration functionalities. The components with dashed borders are developed in this thesis and are placed in the figure to show how our work integrates into the existing system. The current state has CDASMix providing support for the MySQL communication protocol to external applications. The requests are then routed to Cloud or local backend data stores, which support MySQL, PostgreSQL, and Oracle technologies. With this work CDASMix now also supports PostgreSQL communication to the application with all the aforementioned routing capabilities of the requests sent by the application. Following is a description of the step-by-step operation flow of our system, with an external application attempting to access the on- and off-premise data stores through CDASMix. We assume incoming MySQL messages.

One could state that the central component of this system is the *Proxy*, because it acts as a go-between server, which receives requests, authenticates the client, builds the target endpoint URLs, transforms the queries from source to target dialect, marshals messages into normalized message format, and forwards them into NMR. Then, it receives the database responses in normalized message format, finally demarshaling them and sending them back

to the application. Additionally, the *Proxy* supports caching mechanisms, communication

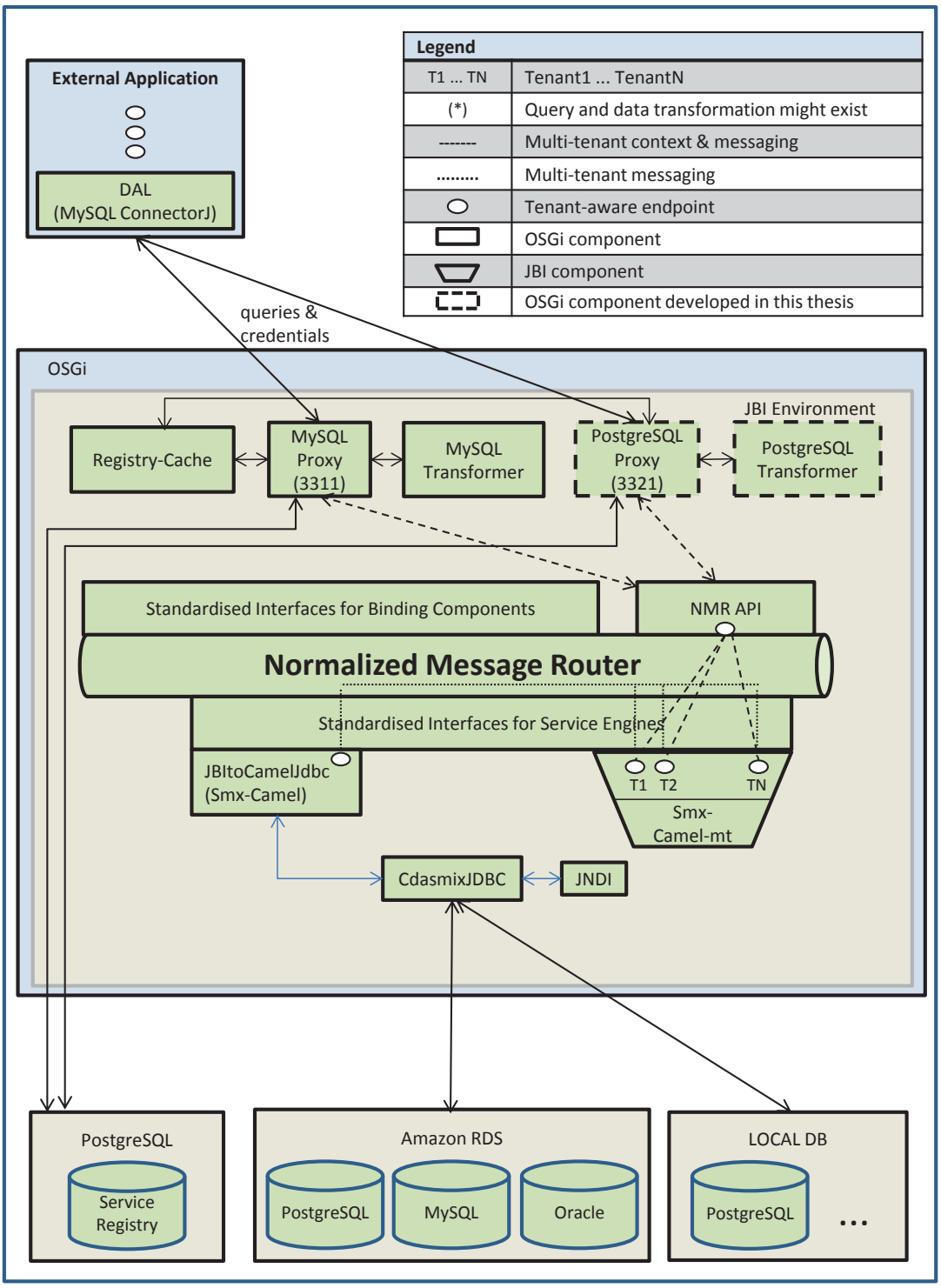

Figure 2.6.: Architectural Overview of CDASMix for Providing Support of Relational Database Access (adapted based on [GS13])

monitoring, and load balancing [mys]. In order to accomplish these tasks, it needs to interact with four different components in the system: the *NMR API*, the *Registry-Cache*, the *Service Registry*, and the *SQL Transformer*.

The operation flow is initialized by the application, which sends SQL queries to the proxy according to the MySQL communication protocol via Transmission Control Protocol (TCP). The MySQL protocol is implemented by the MySQL proxy, the MySQL backend databases and the MySQL native driver *Connector/J* (Java implemented). Integrating a MySQL server into the ESB, which will operate as an intermediate layer between the application and our system, conflicts with the main concept of ESB to be itself such an intermediate technology among applications and servers. Therefore, in [GS13] a Java version of MySQL proxy is integrated, developed by Continuent Inc.: Tungsten Connector [Cona]. However, this is not OSGi compliant and not integrated with the JBI environment. It had to be extended to an OSGi bundle that implements an OSGi- and JBI-compliant version of a Java-based MySQL proxy.

In case of multi-tenant aware communication with the backend, each tenant and user will be identified by unique *tenantID* and *userID*. Hence, ahead of sending the queries, the application sends the credentials for authentication. Before migration, the target destination is an endpoint of the source database, described as an URL. The Java Database Connectivity (JDBC) driver used in the data access layer of the application, gives the connection to the database and implements the protocol for transferring data. CDASMix attempts to support seamless migration, by modifying the applications as little as possible. However, after migration, the data access layer of the application must be slightly modified and updated with new credential information and a new target point of entry, the proxy's port. This is the single physical endpoint, through which the application accesses our system and through it connects to any of the multiple physical backend data stores. In the CDASMix implementation, the port numbers "*3311*" and "*3321*" respectively correspond to the endpoints for the MySQL and PostgreSQL proxy. These port numbers do not have to be static and can be reconfigured before the deployment of the proxy bundle.

The sent packet of credentials consists of the *tenantID*, *userID*, the hashed password and the database name. The hashed password is a 32 digit hexadecimal number produced by the *MD5 message-digest algorithm*. The proxy will connect to the local database, the so called Service Registry and inquire about the password, which corresponds to the received pair of *tenantID* and *userID*. Then the proxy will apply the *MD5 algorithm* to the obtained password. The resulted hash value has to be equal to the encrypted password received from the application. Otherwise, the authentication will fail, the connection to the backend will never be established and the connection of the application to the proxy will close. The *Service Registry* contains the tenant-aware and data source information as a table describing the relationship between source and target databases.

In case the tenant is authenticated successfully, the next step is to check if a query transformation is required. The decision is based on the tenant-aware information, stored in the *Service Registry*. From it the type of the source data store (e.g. "`mysql-database-table-5.6.22`")

is extracted and compared to the type of the target data store (e.g. "`postgresql-database-table-9.4.1`"). If they differ, the received queries from the application are input into the *SQL Transformation* component. The output is SQL queries of the target dialect. The SQL queries, transformed if required, are forwarded to the *NMR* through a *Normalized Message API* (*NMR API*). The latter implements a set of operations for accessing the *NMR* and creating message exchanges. Messages are in a *Normalized Message Format (NMF)*, and they are dynamically routed by the NMR. The target endpoint is specified dynamically by the tenant context information, the service type and the endpoint name.

NMR is able to route and exchange messages between endpoints configured on OSGi bundles and endpoints configured on JBI components. It supports loose communication between components hosted in the two containers, and therefore, it is a significant component for system integration. In CDASMix, first, the NMR routes the NMF into the tentants' JBI endpoints, which are deployed on *ServiceMix-camel-mt*. Then, the requests, still encapsulated as NMF, are routed by the NMR in *JBIToCamelJdbc* endpoint, deployed on *ServiceMix-camel*. These two routes, because of their deployment on different components (ServiceMix-camel-mt and ServiceMix-camel), are packed and deployed in different SUs and SAs.

JBIToCamelJdbc forwards the NM to the Camel *CDASMix JDBC* component (depicted in Figure 2.6 as *CDASMixjdbc block*). The latter creates and exchanges requests with external database systems via JDBC. It looks up the appropriate driver via Java Naming and Directory Interface (JNDI), establishes the connection to the target backend, demarshals the NMF and marshals the obtained requests into a stream of binaries (suitable for transport across the network), and eventually sends it to the selected backend data source. CDASMix implements the communication support for three database systems: MySQL, PostgreSQL, and Oracle, which are included in the OSGi bundle of the *CDASMixjdbc* component.

CDASMixjdbc is also the component, which receives the response of the database, marshals it back to NMF and forwards it to the NMR. The MySQL proxy bundle, demarshals the retrieved NMF, transforms the received data into the source dialect, if required, marshals it into a binary TCP stream, which is sent back to the application. The response of the backend database can be of two types, depending on the type of the statement sent for execution: in case of a data query (e.g. a `SELECT` statement), a table of values satisfying the query will be sent to the *CDASMixjdbc* component; otherwise, if the executed query is a data modification or definition statement (e.g. an `UPDATE` or `CREATE` statement), the database responds with the number of affected rows by the statement execution. The structure of the response is the same, regardless of the database system that sent it. However, the data types of the parameters involved in the response can be different from the set of data types that the source database accepts. Even when this happens, as we will see in Section 4.1, the contribution of the SQL transformer is not required. JDBC can handle the necessary transformation tasks when the source and target databases are migratable to each other.

Cache registry is used to increase the performance of our system. It contains the tenant information that was currently used, or responses from the backend regarding the statements that were recently sent.

2.6. PostgreSQL Protocol

The PostgreSQL communication protocol is divided into two phases, named A and B in Figure 2.7, respectively the start-up and the normal operation. *Phase A* consists of the following sequential events. The client initiates the communication by sending the first PostgreSQL packet to the backend. This packet includes information regarding the username of the client, the name of the database it intends to access and also the values of the PostgreSQL connection parameters: `TimeZone`, `DateStyle`, `extra_float_digits` and `client_encoding`. If the server is satisfied, it replies with the MD5 password request. The client sends the second PostgreSQL packet as a response and if it fulfills the requirement of the backend, the connection is established and the communication enters Phase B. Otherwise, the server sends an error message response and the communication is terminated. *Phase B* is driven mainly by the server, which receives the queries and the configuration queries from the client and starts the process of their execution. PostgreSQL supports two types of sub-protocols for handling the query execution: *Extended Query* and *Simple Query* protocols [posb]. In

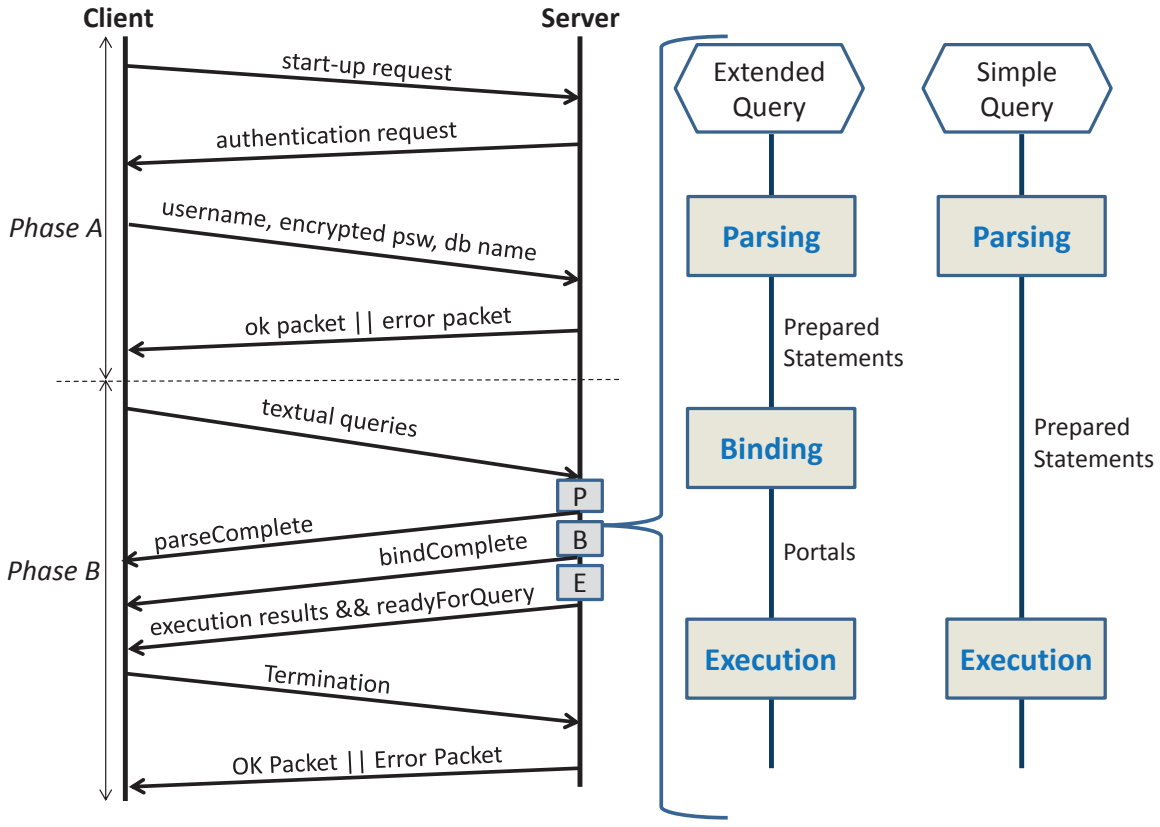

Figure 2.7.: PostgreSQL Communication Protocol

extended query mode, the backend divides the queries execution into three steps (see Figure 2.7): parsing and binding are preceding steps of the final execution. In simple query mode, the execution is performed immediately after the parsing of the queries. The parsing of

the textual SQL statements return the so called *prepared statements*. It is up to the client to choose if the prepared statements will be named or not, by sending along with the queries, the name of the prepared statements, generated on the sever-side. If successfully created, an unnamed prepared statement lasts only till the next parsing output, generating the new prepared statement. On the other hand, a named prepared-statement object lasts as long as the current session is alive, if not explicitly destroyed. Prepared statements actually are the result of query compilation and they describe the query plan and semantic. The variables of the query are placed with placeholders in the prepared statement. During the binding process they will be bound to the statement and the portals are created. Portals can also exist in named or unnamed form, depending on what the client defined in the packet, sent with the queries. The stream of portals is finally executed by the execution process. In case of the simple query protocol, the binding of the parameter values to the statement is involved in the execution process and it does not exist as a separate process.

The PostgreSQL protocol is implemented by the PostgreSQL proxy, the Service Registry and the native driver of the *CDASMixjdbc* component (See Figure 2.6), which implements the communication to the backend PostgreSQL data source.

3. Related Work

In this chapter a framework of discussion, which highlights the state of the art related to our work will be given. The seamless migration, with a minimum impact on the application side and the transparent communication between the on-premise application and the data sources, is a field of survey, which yields novel and diverse approaches. As indicated in Chapter 1, the database can be migrated to multiple database systems (on- or off-premise), in order to achieve the best fitted solution to the specified enterprise requirements. Hence, one challenge of our system is to offer, apart from multi-tenant, also multi-database and multi-protocol support. For this reason, this chapter starts with the discussion of the similarities and differences of our system to the Multi-database System (MDBS). Moreover, as we already clarified, we focus on integration of systems with different query language support. Thus, the work done in SQL transformation is also discussed here.

3.1. Multi-database System

Sheth and Larson define MDBS as a *distributed* system of *autonomous* and potentially *heterogeneous* component database systems, which cooperate and are integrated in various degrees [SL90]. Each component can be accessed by a software responsible for their manipulation. Users can store data to different DBS' and retrieve data from them through a single endpoint, which provides a logical connection to the pool of data stores. A popular implementation architecture for an MDBS is the *mediator/wrapper* approach, as illustrated in Figure 3.1.

After migration, the application may require integrated access to two or more heterogeneous data sources and our system, as a middleware solution, must be able to provide this in a transparent way. Hence, our system could be considered an MDBS with some modifications.

We consider ESB as a single entrance point to multiple, autonomous, backend databases regardless of their physical location and the DBMS each of them supports. However, this approach is not absolutely equivalent to the mediator/wrapper approach. One could claim that ESB is used purely as a mediation between application and database systems, but instead of a wrapper we use a local database (Service Registry), which stores the information provided by the tenant during the migration.

Regarding the physical distribution of the DBS' our system is identical to the MDBS. CDAS-Mix supports the distribution of data among many target databases in different locations, on- or off-premise, while still accessible through a single logical endpoint by the tenant. But our system does not support the joining of data stored in different data sources. Subsequently,

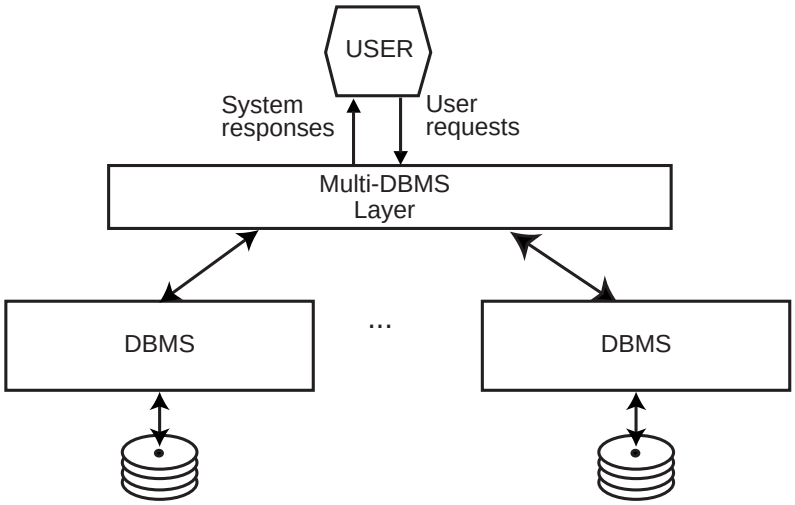

Figure 3.1.: Components of an MDBS [OV11]

it is not possible to refer to tables, etc., of databases with different locations in one concrete SQL statement. Our system can support multiple queries that refer to multiple backend databases, but one single query cannot refer to more than one. Thus, CDASMix is a remote DBMS interface, able to provide access to multiple DBS', although not simultaneously. One query execution takes place in only one DBMS, and so, only one specific database type can be accessed each time [SL90].

An MDBS is also characterized as an aggregation of autonomic components [SL90]. *Autonomy* denotes that each Database System (DBS) that composes the MDBS, controls its operations independently. Such a system is also identified as a Federated Database System (FDBS). In our approach, CDASMix provides a centralized control to every DBS component and is the only one that can "directly" access them. So, autonomy is not an attribute of our system.

Heterogeneity, on the other hand, is a common attribute of MDBS and our system. It refers to the technological differences across the components, specifically to the differences of the involved RDBMS. Codd's rules isolate the application from the details of physical storage of data and their access. Also, SQL language appeared to be the universal tool for managing relational databases. Those two facts significantly ease the unification and transparent access of individual databases, even if the multitude of database vendors offer database systems of different characteristics. The variations in SQL dialects still remain to be handled. Also, the different data types, resulting from several implementations of database systems, contribute to the heterogeneity and must be addressed. This work aims at resolving SQL language heterogeneity between source and target data sources. However, we only cover a one-to-one resolution; one source-one target database. As was clarified previously, our system may support different DBS', but as a remote DBMS interface, it provides access only to one specific type of DBS at a time.

3.2. SQL Transformation

Handling the issue of different source and target data sources requires the enhancement of the intermediate layer with a new component, the SQL transformer. When data migration is involved, the SQL statements addressed to the source database need to be transformed to the dialect that the target database understands. After transformation the semantic of the queries and subsequently its operation on the backend, must remain unchanged. SQL statements can be grouped in two categories, static and dynamic SQL statements. The former do not change at runtime and this enables their hard-coding into the application. Dynamic SQL statements however, are formed at runtime and therefore, hard-coding them into the application cannot be applied [Goo09].

The existing transformation methodologies can analogously be classified as static and dynamic. Static transformation tools work sufficiently with static SQL statements, but they cannot be used when dynamic queries are to be transformed. Apart from their limitation in handling dynamic queries, another drawback of them is the way they operate: they transform all the static statements embedded into the application, to static SQL statements of the target dialect, resulting in a permanent and one-time adaption of the application to the new database technology. Hence, they are not a tool to achieve seamless migration. On the other hand, dynamic transformers can handle all the range of possible SQL queries, after they are sent from the application, while still on their way to the backend data source. There are plenty of tools available as dynamic SQL transformers, both commercial and open source. *General SQL Parser*[1], and *SwisSQL API*[2] are the most widely known. These, as any other transformation tool, come along with a lexical analyzer and parser to decipher the source SQL statements.

The lexical analyzer breaks the statement intended for transformation into a sequence of tokens, each a string with a defined meaning. The stream of tokens is passed to the parser, which analyzes it and determines the structure of the information contained in the SQL statement. Usually, the output of the parser is an expression tree. The nodes of the tree are *expressions*, which are equivalent to the operators in *Relational Algebra*. Actually, the notion of expression trees was firstly introduced to this field to draw a logical query plan of the SQL statements. The inner nodes of the tree are operators, applied to their child or children, while the leaves are the operands, representing either variable or constant relations [Mol12]. Lexical analyzers and parsers are basically incorporated with compilers and interpreters, but SQL queries transformation is also an application of them [Rei11].

Writing a parser tends to be a long and complex task. Therefore, parser generators have more often been the subject of study. A parser generator outputs parsers and lexical analyzers based on the grammar file, which rules the transformation. There are several such tools available. ANother Tool for Language Recognition (ANTLR)[3] is one, generating parsers that parse the input from left to right, without backtracking. Because they parse the input from

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

*L*eft and construct a *L*eftmost derivation of the sentence, they are called LL parsers. Another example of an LL parser generator is JavaCC[4].

Our implementation of the query transformer component is based on an open source project, `JSqlParser`[5]. It parses an SQL statement and translates it into a hierarchy of Java classes and it is generated with JavaCC. It consists of two parts; the JavaCC grammar for SQL dialect and a set of Java classes representing the lexical components of an SQL statement. Figure 3.2 depicts a high level description of the processes involved in the query transformation and the cooperation of different components for achieving it.

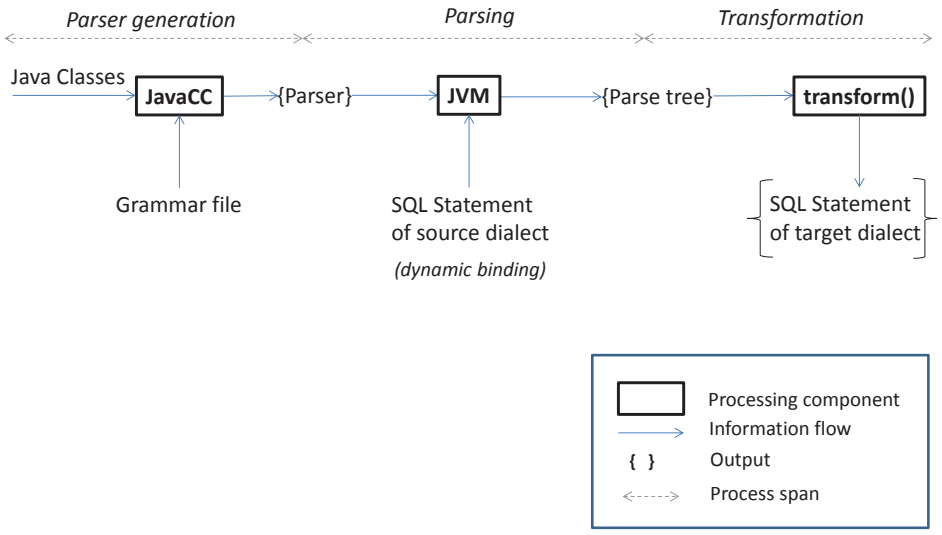

Figure 3.2.: Block Diagram of JavaCC and Transformer Cooperation

At first the parser must be generated. This is a task of the JavaCC program, which uses the grammar file (`JSqlParserCC.jj`) and the set of Java classes as input. The Java classes are the building blocks into which the parsed statement is decomposed while the grammar file contains the rules for this decomposition. Both these rules and building blocks define the parser. Modifications and extensions of the rules will be injected into the generated parser and this is the way to create the parser that serves our goal. For example, the grammar file used in the `JSqlParser` project is of general purpose and it is not compliant with the specific objective of our project. Hence, this file must be extended according to the rules of the PostgreSQL syntax. The changes in the grammar file are injected into the generated parser, which is long and complex Java source code. The source code of the parser is compiled by the Java compiler. This step is omitted in Figure 3.2 in order to keep it short and simple. The

[4]Java Compiler Compiler: `https://javacc.java.net/`
[5]JSQL Parser Project: `http://jsqlparser.sourceforge.net/`

Parser encapsulates both the source code of the parser and its bytecode, as it is created by the Java compiler. During runtime JVM binds the SQL statement to the parser's object code and generates the hierarchy of Java classes, the so called parse tree, which is the expression tree discussed previously in this section. All the components of the tree (nodes, branches and leaves), come from the set of Java classes put into JavaCC. These classes are divided into two main groups, *statement* and *expression*. The former group contains the classes that only implement the nodes of the parse tree, while the latter group contains components for representing nodes, branches as well as leaves.

This structure allows the dynamic binding of SQL statements. Moreover, the code that represents any dynamic query as parse tree can be compiled and run normally. Therefore, this approach is able to parse and then transform dynamic queries. As far as the structure of the parse tree and the components it consists of, give meaningful content during the transformation process, they can be substituted with the SQL notation of the target dialect. The resulting SQL statements have the same impact on the backend as the source SQL statement would have had.

4. Analysis and Specification

In this chapter we provide the analysis of the system and of its environment. The newly integrated components, *PostgreSQL Proxy* and *PostgreSQL Transformation*, will be analyzed regarding the functional and non-functional requirements the system must fulfill.

4.1. System Overview

Section 2.5 gave an insight of the internal components of CDASMix and the way they coordinate to achieve one goal: to provide transparent and multi-tenant access to the backend databases. In this section we aim at giving a higher level view of CDASMix and treat it as a single entity, interacting with the other surrounding technologies. The peripheral applications and systems, set-up the environment for the configuration and operation of CDASMix. Inner details of CDASMix will be hidden, however, the internal building components of it that handle the communication with external systems, are taken into consideration during the following discussion.

4.1.1. Configuration of CDASMix

In Section 2.5, it is taken for granted that the local PostgreSQL database system contains tenant-aware information. Actually, the tenant aware configuration data in the local PostgreSQL data store, is distributed among three registries: Service Registry [Muh12], Configuration Registry [SALM12], and Tenant Registry [SALM12]. In this section we describe the way the first information structures of the local registries are populated.

In [SAGS+12] a Web application for multi-tenant aware management and administration of both BCs and Service Engine (SE)s was developed. It is called JBI Multi-tenancy Multi-container Support (JBIMulti2)[1] and it allows tenant users to have a limited configuration access to the connectivity and integration services of CDASMix, by granting them to deploy configuration artifacts into it. JBIMulti2 is the tool through which the required tenant and backend information are written to the local PostgreSQL registries. It can modify all of them and the modifying operations have to be handled within distributed transactions. For this reason, JBIMulti2 is deployed in JOnAS, a JavaEE 5 application server that can manage distributed transactions, while being responsible for security, thread-pooling, and resource management. But, how can JBIMulti2 interact with CDASMix and when is it able to add tenant-aware information to the registries?

[1] The name describes the ability of the application to provide multi-tenant aware administration and management of both BCs and SEs [SAGS+12].

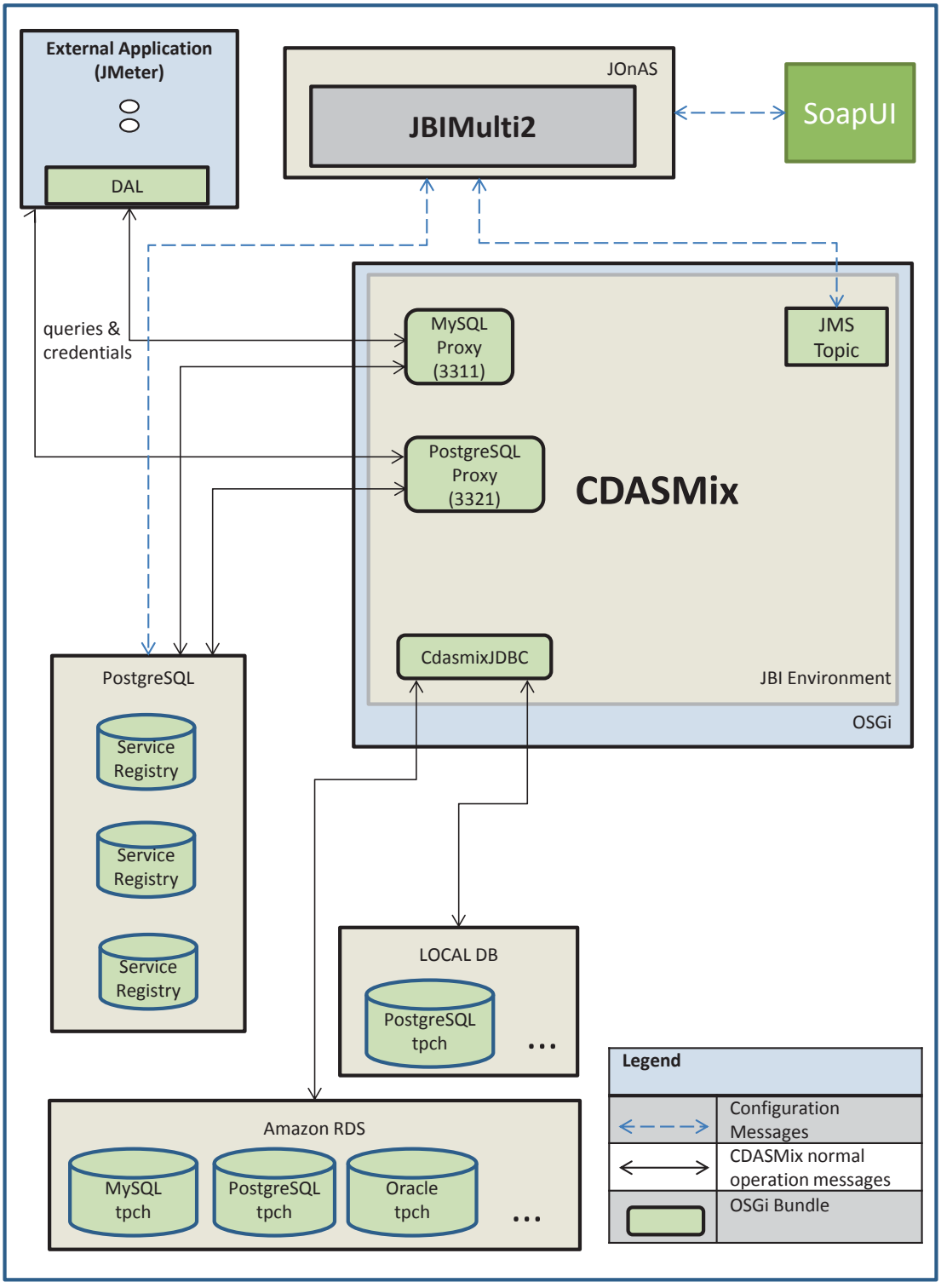

Figure 4.1.: Interactions of CDASMix and the Peripheral Technologies, During Configuration and Normal Operation

CDASMix inherits the extended version of ServiceMix, which supports multi-tenancy and further extends it with the deployment of an OSGi bundle, the `JMSManagementService` component. This is required for enabling CDASMix to communicate with the Web application, namely JBIMulti2. However, multi-tenancy of CDASMix must ensure isolation of data between tenants and therefore, they should not be able to access services of each other. Moreover, tenant users should not notice their simultaneous use of CDASMix with other users of the same or another tenant. Thus, a coordination of users of each tenant is required. Administrators of a tenant define the roles and permissions in order to authorize the tenant users' access to data and services. Each request is bound with the tenant context that describes the role and permissions. *"A configurable service instance uses the received tenant context to adjust its behavior, while a service exclusively provided to one tenant uses the tenant context for authentication"* [SAGS+12]. The requests are sent to JBIMulti2 via its Web service API, which is called by SoapUI 4.0.1[2], an open-source, SOAP-based testing tool. Specifically, while PostgreSQL, JOnAS, and CDASMix are running, the system administrator registers the CDASMix instance to the JBIMulti2. Then: (1) SOAP request messages are sent via SoapUI to JBIMulti2, for installing an extended multi-tenant aware version of Apache Camel SE and the corresponding endpoint configuration per tenant into CDASMix; (2) JBIMulti2 forwards the requests to a Java Message Service (JMS) topic; (3) JMSManagementService listens to the JMS topic for incoming management messages sent by the JBIMulti2, (4) analyzes the content of them and if JBI components or SAs are sent, (5) it respectively installs or deploys them; (6) SOAP response messages contain the UUIDs generated by JBIMulti2 for the tenant. From this point forward, the system administrator can start to add tenant information to the registries, by using the JBIMulti2.

All tenants and their users are added to the Tenant Registry. They are identified by the Universally Unique Identifier (UUID) and their properties are represented as key-value pairs, where the key is the UUID. The Service Registry stores service assemblies and service descriptions in a tenant-isolated way. The former are stored as binary ZIP files and the latter represented as XML files. Lastly, the Configuration Registry contains all the other data, tenant related or not. Specifically, it stores the configuration information generated by a tenant and its users. The service registrations and configurations stored in the Service Registry are excluded from here [SALM12].

4.1.2. Operation of CDASMix

After successful configuration, the registries contain the tenant-aware information and the normal operation of CDASMix can start. In Figure 4.1, the message flows corresponding to the normal operation are depicted as arrows in black, while those related to the configuration are depicted as dashed arrows in blue. The external application initiates the communication by sending a start-up request to the proxy port. Then proxy takes action and its operations are described in detail in Section 2.5. We used Apache JMeter[3] as an external application. There

[2]SmartBear Software, soapUI: http://www.soapui.org
[3]Apache JMeter Project: http://jmeter.apache.org

are three backend databases hosted in Amazon RDS [4] and one PostgreSQL database hosted locally. They all are populated using the TPC-H[5] tool. After a successful authentication to the proxy, JMeter is configured to forward SQL queries that are generated by the TPC-H benchmark via the TCP protocol [Tra13]. However, in our work TPC-H is not used for benchmarking, but to generate real world and realistic data, and the corresponding queries.

4.1.3. CDASMix JDBC Component

CDASMix is connected to the target databases via *CDASMix JDBC Component* (named also CDASMixjdbc), as it is shown in Figures 2.6 and 4.1. This component loads the appropriate JDBC driver for the connection to the target database dynamically, after first checking whether the connection is already registered to the JNDI or not. The interaction with the backend data stores is achieved through the JDBC API. JDBC driver provides a unified API to deal with the response from the database. Up until now, CDASMix JDBC bundle can support three database systems: MySQL, PostgreSQL, and Oracle, as we explained in Section 2.5. The capability of CDASMixjdbc to connect dynamically to any database of the aforementioned technologies and to receive the response from them, makes it a unified solution for interacting with different databases [jdb].

As discussed in Section 2.5, the CDASMix JDBC bundle transforms the NMF messages into a binary stream that is sent over the network to the target backend. Then, it receives the response from the backend, namely the result of query execution, which has the same structure, independent from the type of the target data source (see Section 2.5). However, the data types included in the results may vary depending on the dialects of the backends and may also be out of the set of data types that the source database supports. The unified API of the JDBC driver deals with the response from the database. The `java.sql.ResultSet` interface is responsible for retrieving the returning results with their metadata. Because of the unified support, the differences among data types are not considered by the CDASMix JDBC component. Thus, the result set is dynamically accessed via `ResultSet.getObject`. Objects of the result set are mapped to JDBC types, which are subclasses of the class `java.sql.Types`. Each JDBC type corresponds to a Java class, a subclass of the `Object` class. The set of Java classes is marshalled into NM and sent to the proxy bundle via NMR. The prox demarshals it and sends it back to the client via TCP connection with the database native communication protocol. JDBC already performed the data conversion of the result set into a Java class, and the proxy bundle does the conversion from the Java class to the native format of the source data store. If the data in the source database is migratable to the target database, their JDBC types can be mapped to the same Java classes. In this case, no SQL response transformation is performed in CDASMix.

[4] Amazon RDS: https://aws.amazon.com/rds/
[5] TPC-H benchmark: http://www.tpc.org/tpch/

4.2. PostgreSQL Proxy Analysis

In this thesis we will not integrate a PostgreSQL server into ESB, in order to implement the PostgreSQL proxy functionality, because as was explained in Chapter 2, this idea conflicts with the main concept of ESB to be itself a middleware technology. As was done for the MySQL proxy, we also make use of the open source, Java-based Myosotis project developed by Continuent Inc. for the PostgreSQL proxy. Specifically, we use the 'native-client' to JDBC proxy for PostgreSQL. This needs to be extended to an OSGi- and JBI-compliant component and to be integrated with the JBI environment. In the following, two approaches for integrating a PostgreSQL proxy into the CDASMix are described.

4.2.1. Approach 1

The PostgreSQL proxy communicates over the network with the external application by using the PostgreSQL client/server protocol and provides unified communication through CDASMix to one or more servers that implement one of the three supported technologies: MySQL, PostgreSQL and Oracle. When the target server is not PostgreSQL, the transformation of the queries is required. The PostgreSQL protocol was described in Section 2.6 (see Figure 2.7). Figure 4.2 depicts one way to implement this into the proxy of CDASMix.

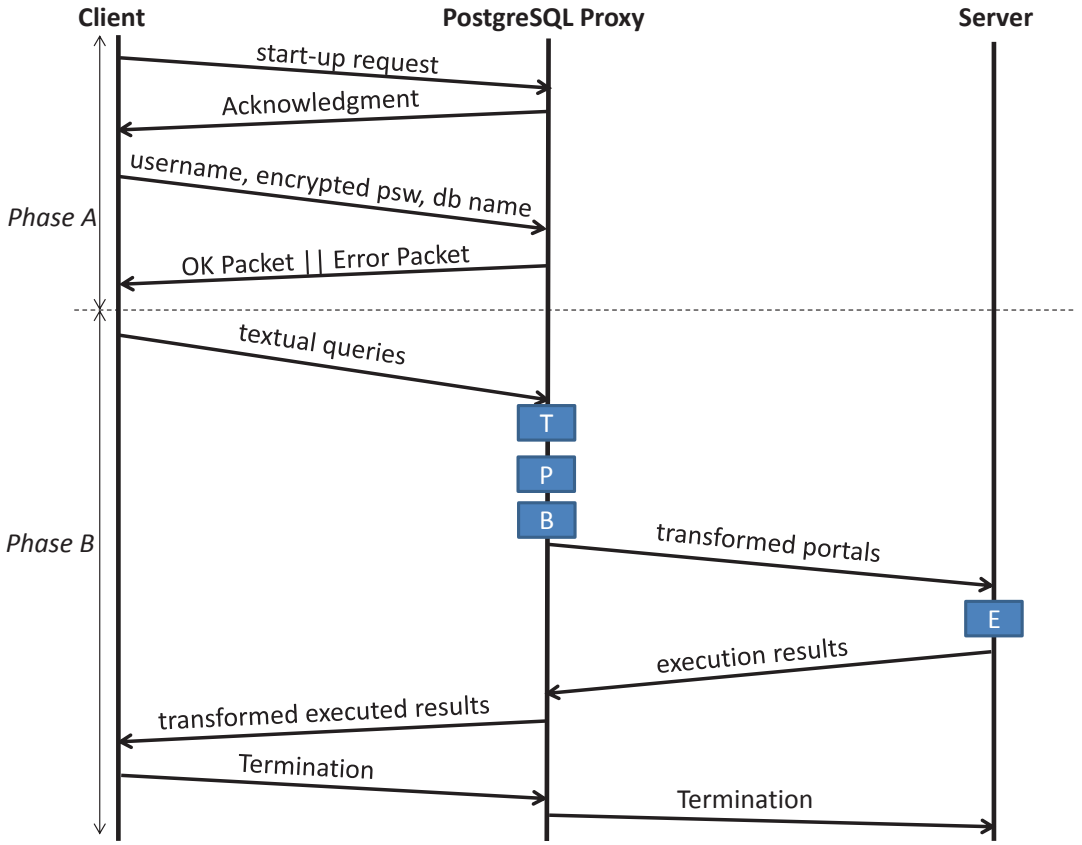

Figure 4.2.: First Approach for PostgreSQL Proxy

In this approach the PostgreSQL proxy fulfills the entire Phase A of the communication. After the successful authentication of the client to CDASMix, via the tenant UUID and password, the communication enters Phase B. Transformation of the queries from the source to the target dialect will be performed unless the backend is a PostgreSQL data source. The next task of the proxy is to pass queries for execution to the server, which in case of a PostgreSQL server, executes them following the Simple Query or the Extended Query Protocol. In this approach the Extended Query protocol is assumed. Parts of it are proposed to be implemented locally, on the proxy side. Specifically, the proxy connects to a local PostgreSQL database (the one that stores the registries) and performs the queries' parsing and binding steps there. The generated portals are sent to the backend for execution. In the end, the received results are forwarded to the client after marshaling them into a binary TCP stream.

This approach may increase the performance of communication, especially when the queries have the same structure and differ from each other only regarding the values bound to their variables. However, this idea incorporates a server into CDASMix and conflicts with the idea of ESB to be an intermediate technology among client and servers.

4.2.2. Approach 2

Figure 4.3 depicts the concept of the second approach for the PostgreSQL proxy.

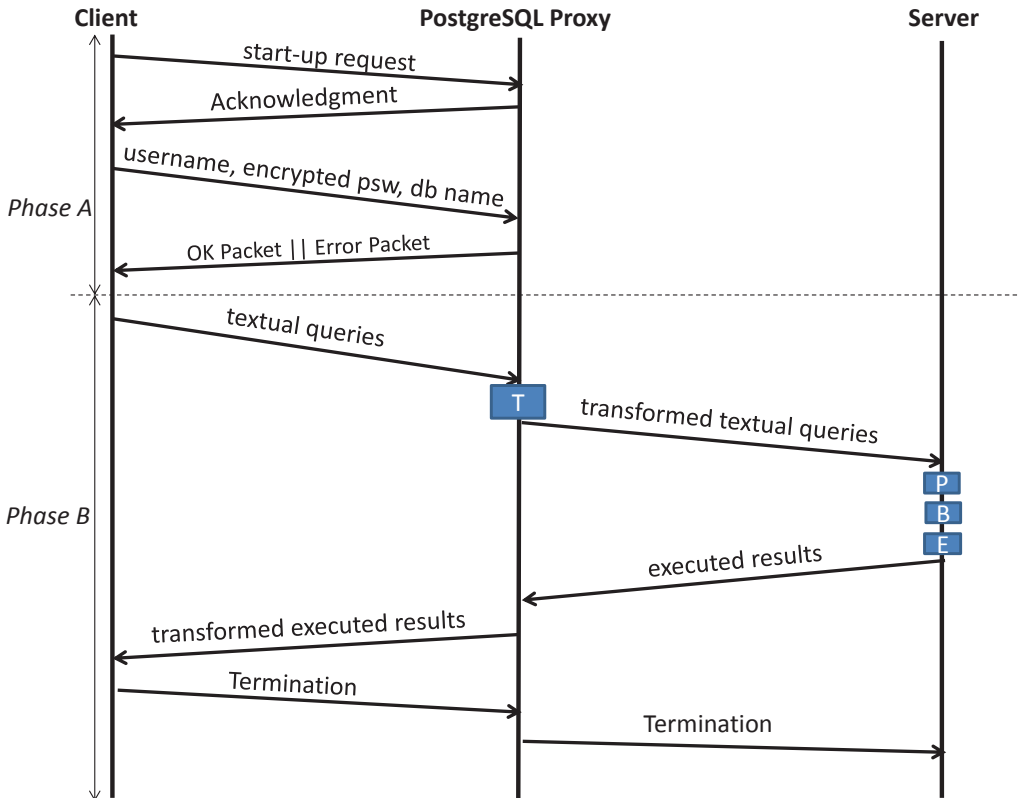

Figure 4.3.: Second Approach for PostgreSQL Proxy

As can be observed, the only operation that the proxy performs with the received queries, is the transformation to the target dialect when it is needed. The transformed queries are sent via CDASMix to the backend, which is responsible for compiling and executing them as well as sending back the generated results.

4.3. SQL Statement Transformation

This section displays and analyzes the functional requirements (FR) of the system. There are two main functionalities that our components, PostgreSQL Proxy and PostgreSQL Transformer, in collaboration with the overall system, must offer: *SQL Statement Parsing* (FR1) and *SQL Statement Transforming* (FR2). Xia implemented in [Xia13] a system, which provides unified and transparent access of the applications to Cloud and local databases, but only for cases of MySQL source data stores. The integration of our components enables the system to also operate for PostgreSQL source databases.

4.3.1. SQL Statement Parsing (FR1)

SQL statements consist of blocks (e.g. `SELECT, FROM, WHERE, LIMIT`) following a syntactical structure, easily understandable to the programmer. Its initial version, developed at IBM by Donald D. Chamberlin and Raymond F. Boyce in the early 1970s, was called `SEQUEL` (Structured English QUEry Language) [CB74]. It was an attempt to provide a programming language similar to English, but with a formal syntax, hence *"structured"*. The strict structure that SQL queries follow makes their parsing a feasible task, even though the rules defining the structure vary. As we already discussed, there is a wide range of SQL dialects available on the market. From the variations of dialects, which are a result of the different formal syntaxes, arises the need for parsing the SQL statements. Parsing is a prerequisite for their transformation from one dialect to another. Output of the parsing process is a the parse tree, an abstract representation of the intention the query carries out. This means that statements from different dialects produce the same parse tree if they are semantically similar, namely, if they aim at performing the same operation to the backend. Thus, the parse tree is a unifying representation of the SQL statement among dialects. It removes the impact of any dialect from the statement and analogously can add the rules of any dialect back to it. One single parse tree can be the parsing result of statements from different dialects, if they all encapsulate the same intention. Additionally, this parse tree can be used to construct statements of any dialect, which are semantically equivalent to each other and to the original statement as well. On the other hand, the parser itself, different from its output, is strongly related to the specific dialect of the original statement. A different dialect means different syntactic rules, which implement the grammar file differently and so, the generated parser differs as well. In Figure 4.4 the parse tree of a real PostgreSQL statement used for the validation of our system is shown. Actually, it is a simplification of the a query generated from the TPC-H benchmark.

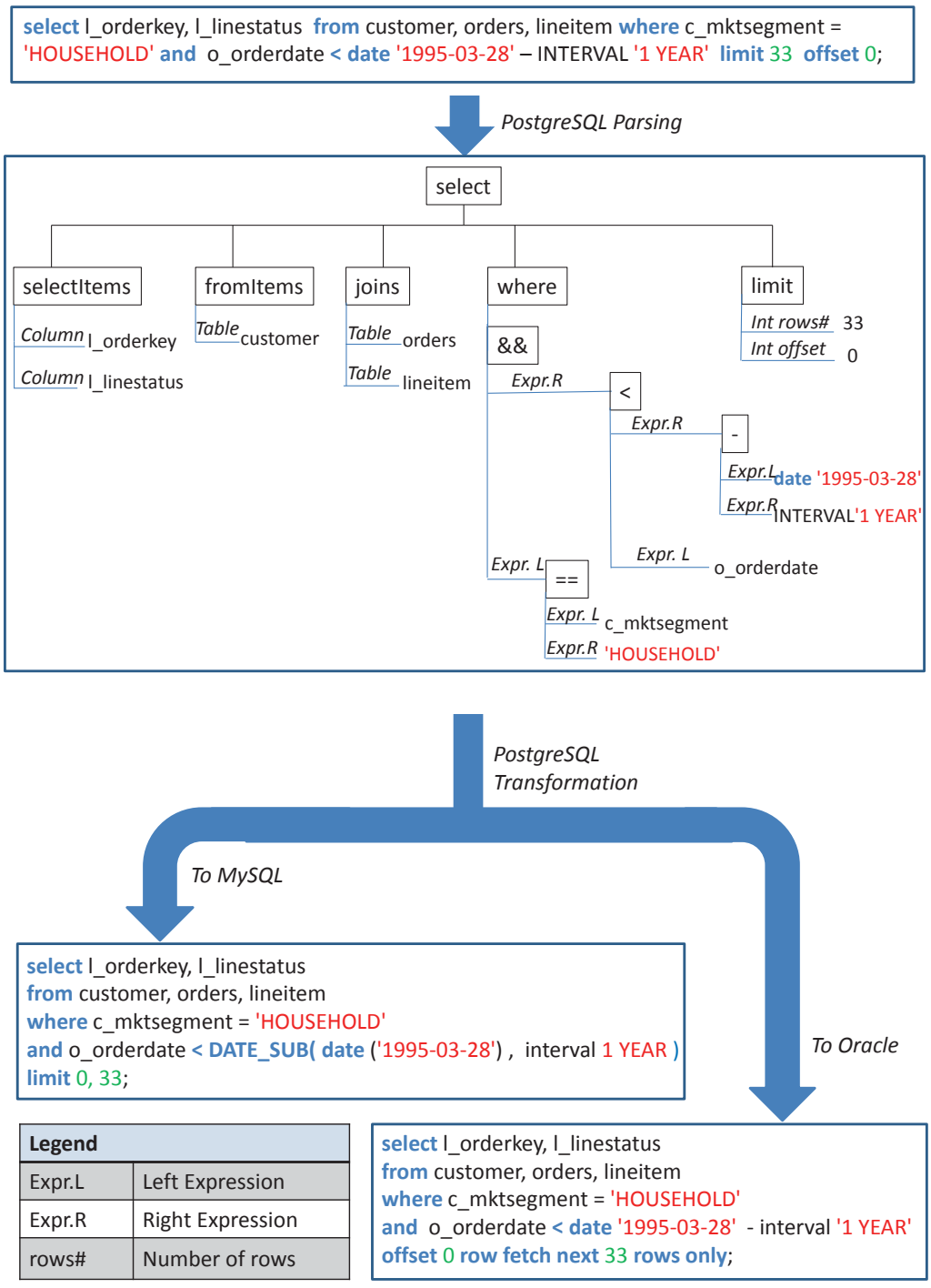

Figure 4.4.: Parsing of a PostgreSQL SELECT Statement into a Parse Tree and Transforming it into MySQL and Oracle SQL.

4.3.2. SQL Statement Transforming (FR2)

As we explained the parse tree can be transformed into an SQL statement of any dialect. In Figure 4.4 the transformation to MySQL and Oracle SQL is shown. The three depicted statements, the original and the two statements obtained after the transformation, slightly differ from each other.

For example, the node `Limit` in the figure expresses that the indexes of the returned rows after the execution of the statement must be within a given range `[offset, offset + numberOfReturnedRows]`. This clause is represented differently, though with the same semantic, in the three dialects covered by this thesis. For example, in `PostgreSQL 9.4.1`, it's written as `"LIMIT 33 OFFSET 0"`, in `Oracle 12c`, it's `"OFFSET 0 ROW FETCH NEXT 33 ROWS ONLY"` and in `MySQL 5.6.22` it is written as `"LIMIT 0, 33"`. Any of these textual representations for limit needs to bind two parameters acquired from the tree: `offset` and `row_count`. Another example of inclinations between the aforementioned dialects is the `DATE_SUB` function. It subtracts a time value (an interval) from a date value and is implemented only in MySQL. As seen in Figure 4.4, the expression `"DATE_SUB(date('1995-03-28'), interval 1 YEAR)"` is substituted with `"(date '1995-03-28'- interval '1 YEAR')"` in the other two dialects. Take note that the date value in MySQL has a different representation, as it puts it in brackets: `"date ('1995-03-28')"`; contrary to Oracle and PostgreSQL: `"date '1995-03-28'"`. In Appendices A and B of [Xia13] a thorough comparison of datatypes and statements from these three dialects is provided. In general, language-specific and user-defined functions and data types make the query transformation process a challenging task, which must evolve continuously. Otherwise, it soon becomes an incomplete, out-of-date and insufficient process.

4.4. Use Cases

In [Xia13] four use cases for the SQL transformation in CDASMix are provided. The first two represent the event steps of SQL transformation, whose actor is the system itself (more specifically: the proxy component). Moreover, Xia details two more use cases for extending the existing SQL transformation functionality, to provide support for additional source and target SQL dialects [Xia13]. Our work is enabled by these four use cases, while it extends the first two, as shown in Figure 4.5 and in the subsequent tables.

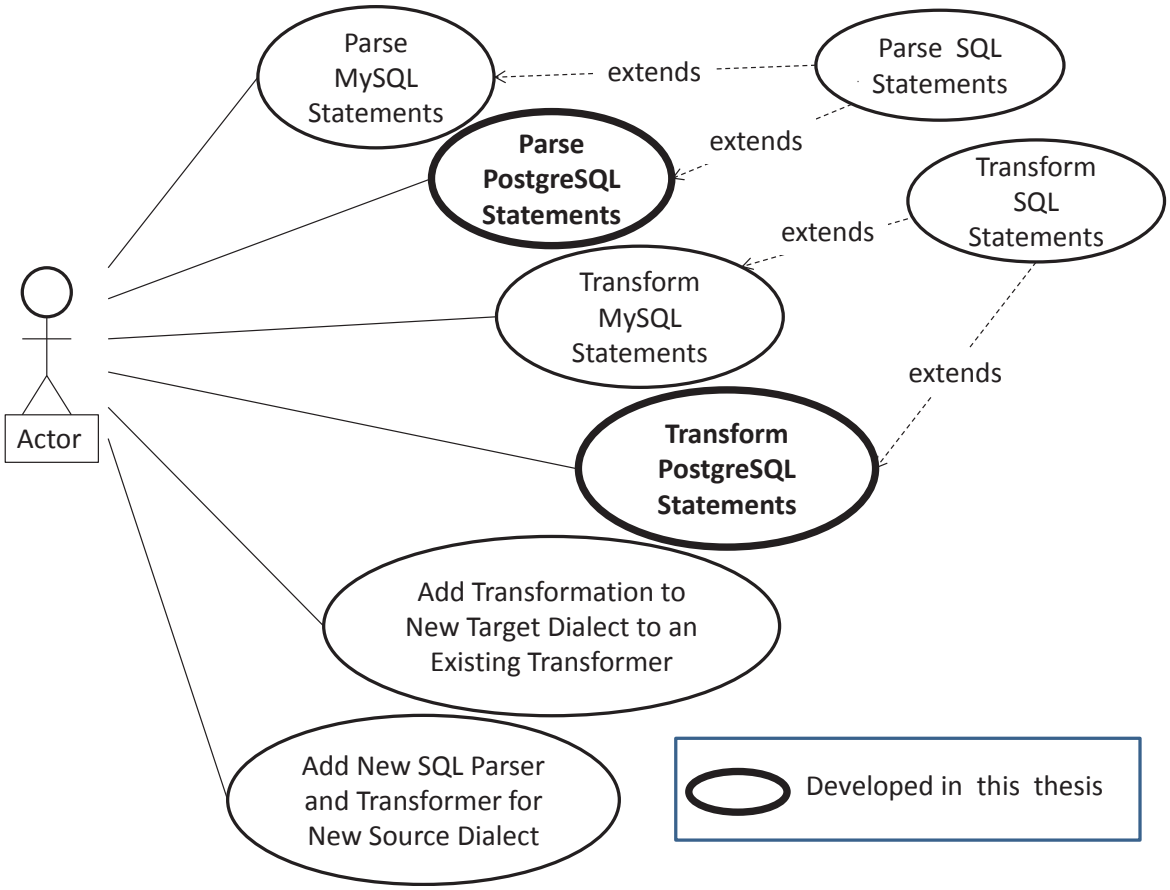

Figure 4.5.: Use Case Diagram for the SQL Proxy

Name	**Parse PostgreSQL Statement**
Goal	To parse a PostgreSQL statement in plain text form into a parse tree
Actor	PostgreSQL Proxy Component
Pre-Condition	A PostgreSQL statement in text form is presented, as well as a corresponding PostgreSQL parser.
Post-Condition	The PostgreSQL statement is parsed into a parse tree
Post-Condition in Special Case	The PostgreSQL statement is not successfully parsed into a parse tree
Normal Case	1. The PostgreSQL proxy component looks up the transformer service associated with the PostgreSQL dialect of the statement. 2. The PostgreSQL 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 PostgreSQL dialect of the source 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 PostgreSQL Statement*

Name	**Transform PostgreSQL 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 PostgreSQL source dialect is successfully parsed into a parse tree; the target dialect must be either MySQL or Oracle; 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 PostgreSQL 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 PostgreSQL Statement*

4.5. Non-Functional Requirements

This section provides an overview of the non-functional requirements that we take into consideration during the design and implementation of our system. The list of the non-functional requirements was specified in a predecessor of our work, in [Xia13], where a more detailed description of them can be found [Xia13]. These requirements rule the development, deployment, and adoption of the new components.

4.5.1. Extensibility (NFR1)

There are plenty of SQL dialects available, which are also constantly evolving. Hence, making our components easily extensible in order to support new dialects and new features of the already supported dialects is important. It must be able to add new components with minimal alteration of existing components.

4.5.2. Integratability (NFR2)

The SQL transformation must be developed as an enhancement of the functionalities of CDASMix. It should be integrated into the system (CDASMix), without affecting its basic functions and requirements. Moreover, the deactivation of the new component or its failures must be isolated from the rest of the system.

4.5.3. Performance (NFR3)

Adding the SQL transformer as an intermediate operation between the proxy and the backend, increases the transmission time and the requirements in other resources, such as memory and processing power. Implementing an effective parsing mechanism can compensate the impact of the SQL transformation in the system performance. Additionally, the position of the transformation functionality also affects the overall performance of the system. It must be chosen in such a way that it reduces internal communication methods.

4.5.4. Scalability (NFR4)

For each supported source dialect, the proxy and the SQL transformer are implemented as two separate OSGi bundles. Each time will be deployed the bundles related to the service we want to consume and not all of them. In this way we do not exceed the limits of the OSGi container and the unnecessary system resource consumption is avoided.

4.5.5. Maintainability and Documentation (NFR5)

Documentation is an important part of this work. There is a constant needs for extensions on the existing bundles, due to the enhancements of the existing dialects with new features and also a need to develop new bundles, in order to integrate the support for new dialects. This makes the delivery of a source code of high readability and a detailed description of the developing steps a big necessity, in order to ensure the continuation of this work and to enable its frequent update.

5. Design

This chapter examines the software architecture and the discipline of the system after the integration of the PostgreSQL proxy and PostgreSQL transformer. The new components, which follow the OSGi specification, fulfill the functional and non-functional requirements, as they are defined in Chapter 4.

5.1. System Architecture

Figure 2.6 shows a representation of CDASMix after the integration of the components developed in this work. They are drawn with dashed borders and their place in the system is chosen in a way that it reduces the different internal communication methods. The goal is to add the new functionalities while decreasing their negative impact on the system performance to a maximum. Figure 5.1 depicts the new added components, only showing the parts that directly communicate with them. This results from Figure 2.6, after extracting the parts of our main interest.

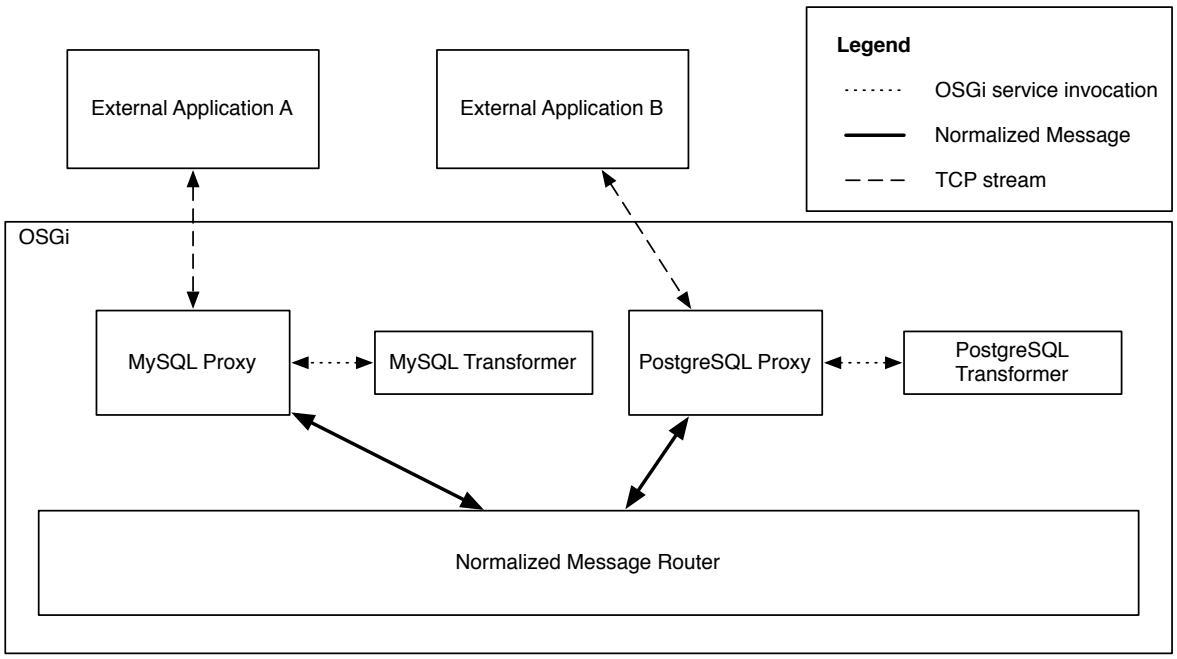

Figure 5.1.: Second Approach - Direct Transformation From Proxy Bundle With Transformer Services

For each type of source data store, the corresponding proxy bundle is deployed to handle their specific communication. As already discussed, the MySQL Proxy that implements the MySQL communication protocol of CDASMix with the external MySQL application was developed in [Xia13]. This thesis develops and deploys the PostgreSQL proxy, which aims at serving PostgreSQL applications through CDASMix. Each proxy receives statements of a particular dialect, specifically the one of the database system it supports. CDASMix needs to be enhanced with new, protocol-specific proxy bundles, in order to support application databases, different from MySQL and PostgreSQL. The main tasks of the proxies are similar, regardless of the protocol they implement, and they are discussed thoroughly in Section 2.5. In this work, the PostgreSQL proxy is extended with SQL transformation functionality. The PostgreSQL transformer shares its transformation service specifically with the PostgreSQL proxy component. In general, the transformation service of one type is shared only with the proxy bundle of the same type. The proxy is responsible for determining if transformation is needed, by comparing the type of source and target data stores that it reads from the Service Registry. If they differ, it calls the transformation service and provides it with the queries and the target dialect. CDASMix supports three types of target data sources: `mysql-database-table-5.6.22`, `postgresql-database-table-9.4.1` and `oraclesql-database-table-11.2.0.4.v3`.

By calling the transformation service, the proxy acquires the transformed queries, which are then marshaled into Normalized Message (NM) and forwarded to the NMR. The fact that the proxy will transform the SQL statements it received before marshaling them into NM reduces the overall internal communications necessary and thus benefits the system performance, as is required in NFR3. The transformation service is implemented as an OSGi bundle. This satisfies the requirements in NFR1, NFR2 and NFR4, as it makes the deployment an easy and seamless operation, isolated from the rest of the system.

The communication of the proxy and its corresponding transformer is an inter-bundle communication and can be implemented in multiple ways. In this work it is chosen to import the transformer package into the proxy bundle, as will be discussed further in Chapter 6 (See Figure 6.1). However, in CDASMix also one JBI component is deployed, the ServiceMix-camel-mt (see Figure 2.6). The use of NMR in CDASMix facilitates a message-based communication, where bundles, but also JBI components, can exchange messages both synchronously and asynchronously. Our system uses Apache ServiceMix 4.3, which provides a complete, enterprise ready ESB, exclusively powered by OSGi. It spans both the OSGi and the JBI container through the Apache ServiceMix NMR that includes a rich Event, Messaging, and Audit API[1].

5.2. SQL Transformation Service

In Section 4.1 we referred to the Service Registry, while describing the endpoint configurations, deployed by the tenants and stored via service assembly entities to the Service Registry in a tenant-isolated manner. In this section we aim at describing the SQL transformation service

[1] Apache ServiceMix: http://servicemix.apache.org/

and we show the contribution of the Service Registry on registering and consuming the OSGi services. The Service Registry, which was first developed in [Muh12] and further extended in [GS13, Sch14], contains in a WSDL file information related to the bundle services and the policies that can be dynamically obtained by the tenants.

Both the proxy and the SQL transformer are designed as OSGi bundles, in order to benefit from the OSGi container that `Apache ServiceMix 4.3.0` provides. Implementing the OSGi technology into the CDASMix, fulfills our requirements, FR1 and FR2. Building CDASMix as a software environment of collaborative OSGi components makes its development a less complex task, by separating it into different and reusable modules that can be developed independently, built more easily and deployed in a more manageable manner. Thus, CDASMix functionalities can easily be extended by integrating into it new or modified OSGi modules, so called bundles.

The OSGi service layer, as defined in the layered representation of the OSGi specification [OSG11], connects the bundles in a dynamic manner by following a publish, find, and bind model for the plain old Java objects, which encapsulate the OSGi services. Each service, which is implemented by its corresponding bundle, is registered in the Service Registry under the name of one or more Java interfaces. The typical way, defined in [OSG12], is that bundles themselves have the responsibility to register their services after creating them. The Java interfaces that a service object implements, group the methods and constants related to the service. A service object is created by its respective bundle and implements the methods it inherits from the interfaces. The service may additionally define the implementation of other properties when registered.

Almost all the functionality of CDASMix is implemented as OSGi services (an exception is the JBI component). Which bundles do we start? If we start them all, the system will work but it will use more memory than required for a particular job. Also, the start-up times will be slow. Hence, starting all bundles at once contradicts with NFR4 and NFR1. The approach followed in CDASMix selects and starts only the bundles which need to be started.

However, bundles may be dependent to each other. For example, the Proxy service has a dependency on SQL transformation service, because in case of different source and target dialects, it uses the SQL transformation functionality to accomplish its tasks. On the other hand, bundles have total freedom to register, unregister, or not register their services at all. Because of this high flexibility, it is necessary, when building our system as a set of collaborative components, to follow the preconditions of all services, which must be carefully documented in advance. This approach is hard and very impractical for large applications [Bar], such as CDASMix.

Separating the two responsibilities, for creating an implementation of a service and for publishing the service in the service registry, is the solution proposed in the *Declarative Services Specification* [OSG12, Bar]. Creating the service object by following the specification of the corresponding interface, remains a bundle's duty. But, the responsibility for publishing services is transferred to a special bundle, the Service Component Runtime (SCR). Figure 5.2 shows the life cycle of OSGi Declarative Services as UML sequence diagram.

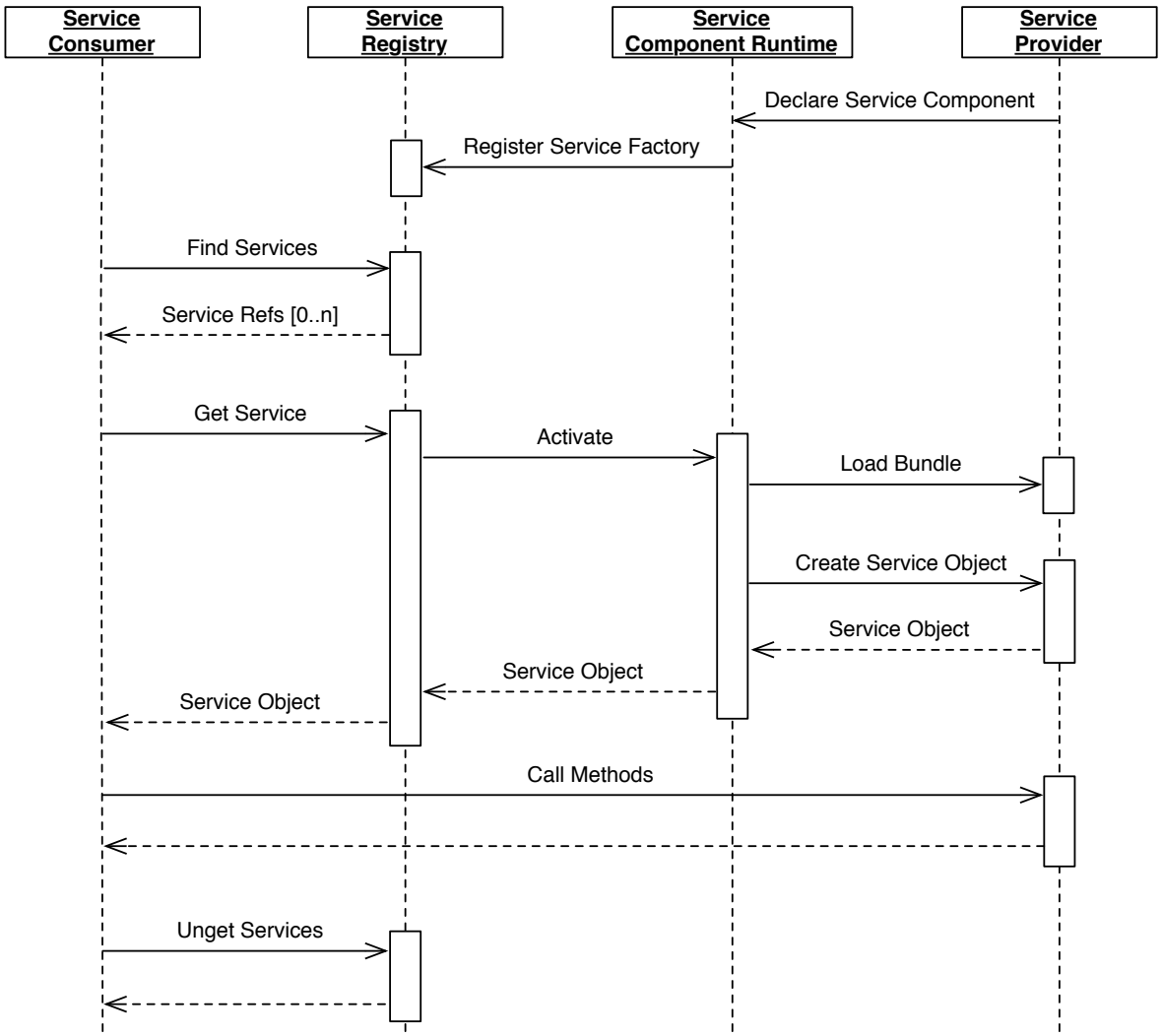

Figure 5.2.: The Life Cycle of Declarative Service of OSGi [Bar]

Each bundle contains, apart from the service to be registered, also one or more XML files with the bundle descriptions. The SCR scans the bundles in search of these XML files and uses the information provided in them, in order to register the services. The SCR may do a so called "lazy registration". In this case, it first creates and registers a proxy object in the Service Registry that operates as a placeholder. From this point on, the consumer sees the service available for use, but only when trying to access the service, will SCR ask the OSGi runtime to load and activate the bundle.

In conclusion, the approach followed in CDASMix regarding the bundles' activation can be summarized as:

1. CDASMix is built from a set of OSGi bundles and this allows its development and maintenance to fulfill the requirements in NFR1, NFR2, and NFR4.

2. It activates only the required subset of its bundles when it starts and not all of them.

Hence, it satisfies the requirements in NFR3 and NFR4.

3. CDASMix follows the Declarative Service specification and thus, it allows SCR to handle the service registration and service dependencies' resolution. This facilitates the extensibility of the bundles and the overall system. The requirement in NFR1 is satisfied.

4. In CDASMix a "lazy" bundle activation takes place. The SCR enables the loading of the bundles to the registered placeholders after detecting an attempt of a consumer to use the respective service. Therefore it saves system resources and reduces the start-up time, as required in NFR3 and NFR4.

6. Implementation

In this chapter, we describe the implementation of the enhancements of CDASMix, as they are specified and designed in Chapter 4 and Chapter 5. First we show the realization of the new components as OSGi bundles and the implementation of their services. We focus on their interaction as consumer-producer. Next, the concrete implementations of the PostgreSQL proxy and PostgreSQL transformer is presented with the help of UML diagrams. The extensions we added in order to enable their integration with the rest of the system are made clear. We further discuss the PostgreSQL parser and its cooperation with the SQL transformer.

6.1. Transformation Service Implementation

In CDASMix the SCR is implemented by the `org.apache.felix.scr` bundle. The SCR first searches the Manifest file for the path of declaration file. The latter is an XML-based file that contains the bundle description as shown in Listing 6.1. It can be placed anywhere within the bundle, although we chose to place it in `OSGI-INF` folder.

```
1  -- OSGI-INF/PostgreSQLTransformer.xml --
2
3  <?xml version="1.0" encoding="UTF-8"?>
4
5  <!-- xmlns attribute defines a namespace for the prefix -->
6  <components xmlns:scr="http://www.osgi.org/xmlns/scr/v1.0.0">
7
8      <!-- Immediate: defines if the component will create its object immediately (true) or on
           -demand (false),
9      Name: component identifier -->
10     <component immediate="false" name="PostgreSQLTransformer">
11
12          <!-- fully qualified name of the class that impements the component -->
13         <implementation class="iaas.unistuttgart.de.sqltransformer.PostgreSQLTransformer"/>
14
15         <!-- the name of the interface under which the service will be registered is listed
               in the "provide" element -->
16         <service>
17             <provide interface="iaas.unistuttgart.de.sqltransformer.api.SQLTransformer"/>
18         </service>
19
20         <!-- configuration properties being available through the ComponentContext -->
21         <property name="source_dialect" value="PostgreSQL"/>
22         <property name="service.pid" value="PostgreSQLTransformer"/>
23     </component>
24 </components>
```

```
25
26
27 -- META-INF/MANIFEST.MF --
28
29 ...
30 Service-Component: OSGI-INF/PostgreSQLTransformer.xml
```

Listing 6.1: OSGi Declarative Service Component Descriptor

The path of the bundle description file is added to the bundle manifest file (`MANIFEST.MF`), under the header `Service-Component`: `Service-Component:OSGI-INF/iaas.unistuttgart.de.sqltransformer.PostgreSQLTransformer.xml`. Thus, after scanning the `MANIFEST.MF`, the SCR is able to find the bundle description and, if the bundle is to be registered as a service, the SCR realizes the service registration under the interface that is provided in the declaration (See the "provide" element of Listing 6.1, in Line 17). Apache Felix Maven SCR Plugin[1] is used to automatically generate the declaration file and to add its path to the `Service-Component` header of the bundle `MANIFEST.MF`. All the bundles for SQL Transformation implement the same interface, which is shown in Listing 6.2. Each proxy supports a specific source dialect and so, it will attempt to access and consume the service object that handles the same source dialect.

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

Listing 6.2: SQL Transformer Service Definition

Figure 6.1 shows the dependencies of the mainly involved classes of the two bundles, PostgreSQL proxy and PostgreSQL transformer, developed in this work, as well as the pre-existed MySQL transformer as UML class diagram. The class diagram is generated with the ObjectAid UML diagram[2], an Eclipse Plugin and our goal is to visualize the associations among them. They follow the producer-consumer model; the PostgreSQL proxy consumes the PostgreSQL transformation service in the following way:

First, as shown in the Figure 6.1, the class `PostgreSQLProtocolHanlder`, of the proxy bundle, depends on the class `SQLTransformation`, of the same bundle. The former class uses the latter one by creating an instance of it, which handles the look-up and consumption of the transformation service.

`SQLTransformation` class, which is shown in Listing 6.3, retrieves an array of `ServiceReference` objects that satisfy the filter shown in Line 10. A `ServiceReference` object references to a registered service and encapsulates properties of it, as well as other metadata related to it, but not the service object itself. The service objects are instances of the component

[1] Apache Felix Maven SCR plugin: http://felix.apache.org/documentation/subprojects/apache-felix-maven-scr-plugin.html
[2] ObjectAid UML diagram: http://www.objectaid.com/

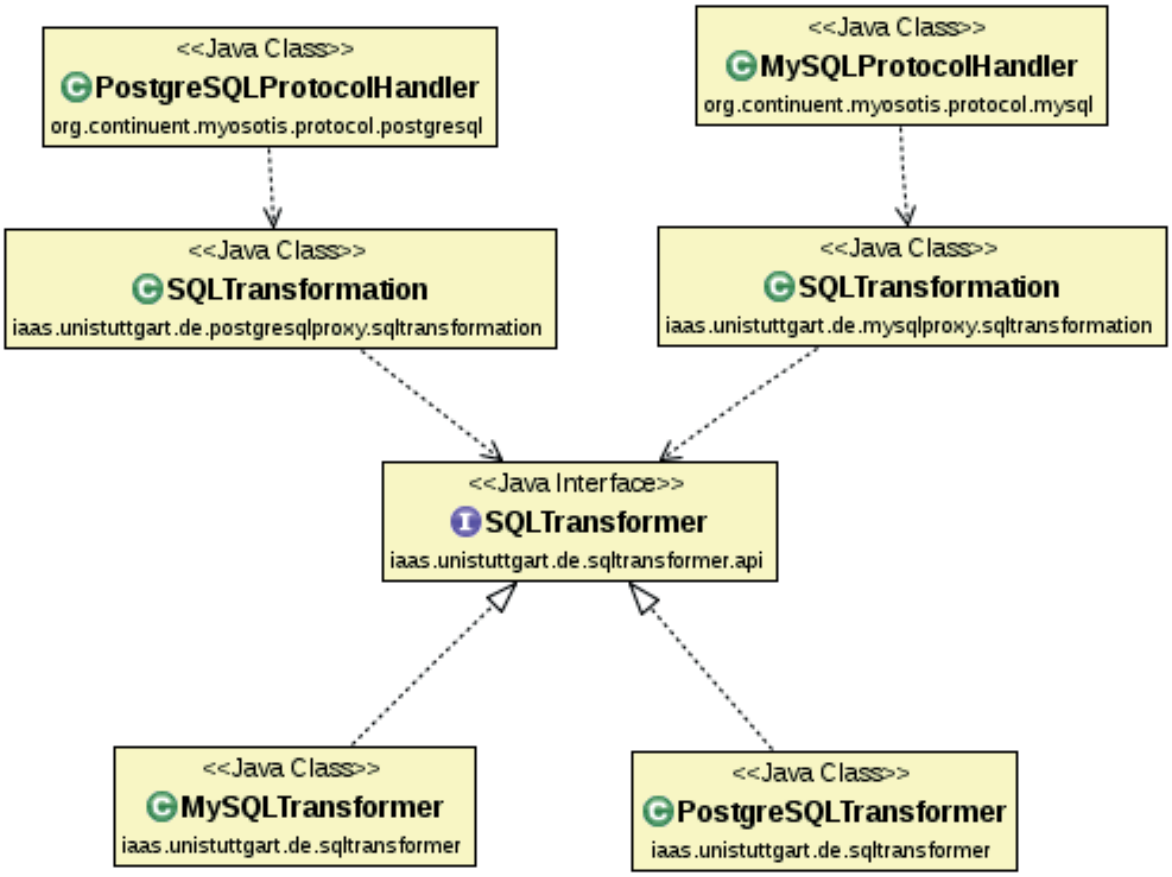

Figure 6.1.: Class Diagram that Shows the Relationships Among the PostgreSQL Proxy and the SQL Transformation Components.

classes, `MySQLTransformer` and `PostgreSQLTransformer`, which both implement the `SQLTransformer` interface, part of the `sqltransformer.api` bundle. The `SQLTransformation` class is able to reference instances of the services, namely objects of the classes `MySQLTransformer` and `PostgreSQLTransformer`, as instances of the `SQLTransformer` interface. The filter `source_dialect=PostgreSQL` is used as shown in Line 6 and so, only the `PostgreSQLTransformer` service is accessed. Java does not allow to create instances of the interface. Instead, instances of the classes that implement the interface can be created. Those instances can be referenced as instances of the interface (see Lines 14 and 20, where *transformer* is an instance of `PostgreSQLTransformer` class, up-casted to the interface and *transform(statement, target)* is the overriden method).

```
1  BundleContext context;
2  ...
3  public String transform(String statement, Dialect source, Dialect target) throws
       NotImplementedException, UntransformableException, SQLParseException,
       TransformerNotFoundException {
4      SQLTransformer transformer = null;
5      ServiceReference[] services = null;
6      String filter = "(" + SQLTransformer.SOURCE_DIALECT_PROP + "=" + source.name() + ")";
```

```
7       // filter= (source_dialect=PostgreSQL)
8
9
10      services = context.getServiceReferences(SQLTransformer.class.getName(), filter);
11      // services= [[iaas.unistuttgart.de.sqltransformer.api.SQLTransformer]]
12
13      if (services != null && services.length > 0) {
14          transformer = (SQLTransformer) context.getService(services[0]);
15          // transformer =iaas.unistuttgart.de.sqltransformer.MySQLTransformer@1897af1
16
17      } else {
18          throw new TransformerNotFoundException(target);
19      }
20      String transformed = transformer.transform(statement, target);
21      return transformed;
22  }
```

Listing 6.3: OSGi Service Lookup With Filter

Activating, when starting CDASMix, all the involved services would contradict with our requirements NFR3 and NFR4 as we discussed in Section 5.2. Figure 6.2 shows the procedure of *lazy registration* of the PostgreSQL transformer service.

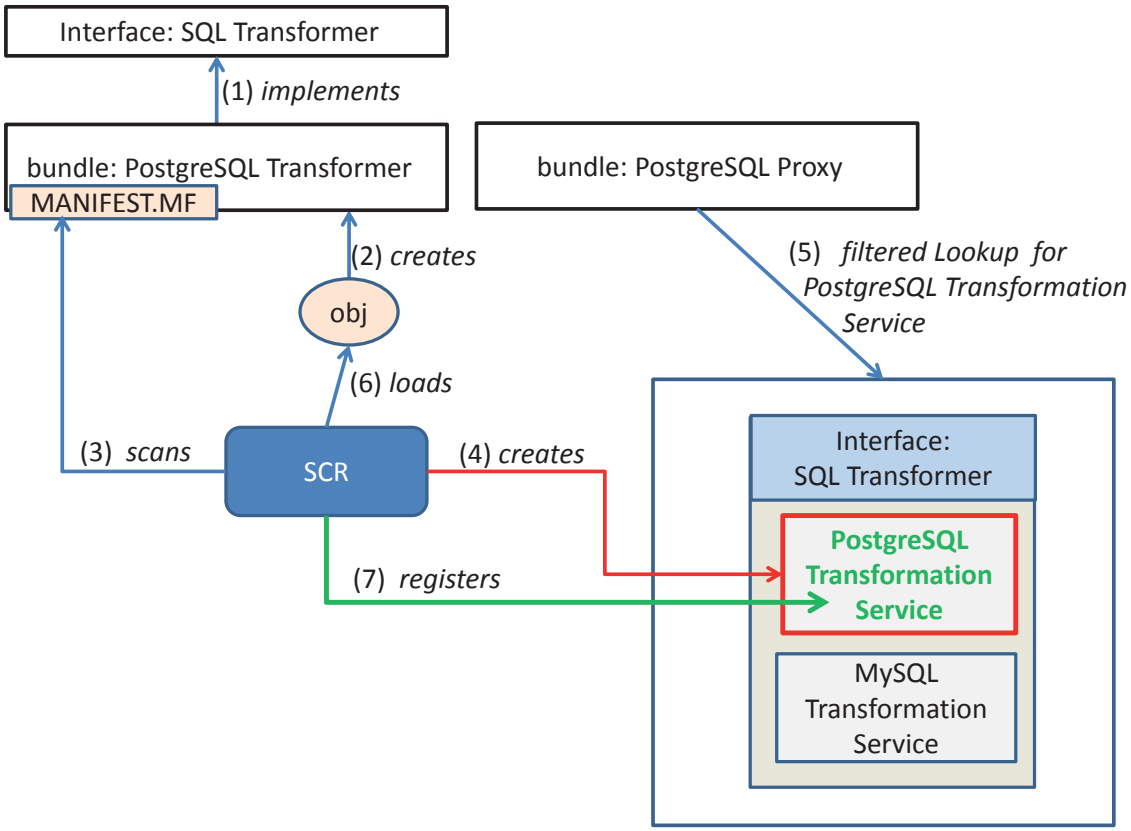

Figure 6.2.: Lazy Service Registration Scenario

In Listing 6.1, in Line 10, the *"immediate"* is defined as false, and this indicates lazy registration for the PostgreSQL transformer service (the same holds also for MySQL). The sequential steps of the lazy registration are enumerated in Figure 6.2 and discussed below:

1. The implementation class of the component, so called component class, must implement the (service) interface.
2. The component class, creates an instance of itself, which actually is the service to be registered.
3. SCR scans the file `MANIFEST.MF` of the bundle. There it finds the path of the bundle declaration file. An example of this file is shown in Listing 6.1.
4. Because *"immediate"* element is set to "false", in the Service Registry will be created a placeholder for the service under the interface it implements.
5. At some point, the PostgreSQL proxy will try to access and consume the PostgreSQL transformation service.
6. SCR "sees" the proxy's attempt to obtain the service. Therefore, it loads the service,
7. and finally registers the service to the Service Registry, to the pre-saved place.

Differently from the transformation component, the proxy does not follow the Declaratice Service Specification, and therefore, there is no *Service-Component* header in its bundle manifest. Instead, the *Bundle-Activator* header is added in the proxy's manifest file: `Bundle-Activator:iaas.unistuttgart.de.mysqlproxy.osgi.OSGIHandler`. The class `OSGIHandler` is called at bundle activation and deactivation time and it implements the interface `org.osgi.framework.BundleActivator`. The proxy component, through the `OSGIHandler` is set responsible to start itself in the ServiceMix OSGi container. To do so, it looks-up and consumes the service of `org.apache.servicemix.nmr.api.NMR` bundle, loads the server configuration from the `src/main/resources/cdasmix.server.cfg` and establishes the connection to the ehcache OSGi bundle. Additionally it creates an object of the class `iaas.unistuttgart.de.mysqlproxy.sqltransformation.SQLTransformation`, which is responsible to look-up and get the SQL transformation service of the dialect that satisfies the filter.

6.2. PostgreSQL Proxy Implementation

The OSGi bundle of the PostgreSQL proxy is developed by the Continuent Tungsten Connector, as we already discussed in the previous chapters. It is a Java PostgreSQL proxy, able to connect directly with the backend PostgreSQL database system. In this work, the proxy is extended and adapted in order to integrate it with CDASMix and reach the objectives about transparency, multi-tenancy, caching, SQL transformation and dynamic connection to the backend data stores of various dialects, that can reside locally or in the Cloud. Figure 6.3 represents the UML diagram of the main classes, which build the PostgreSQL proxy bundle or facilitate its integration into the system. Most of the classes have a huge amount of attributes

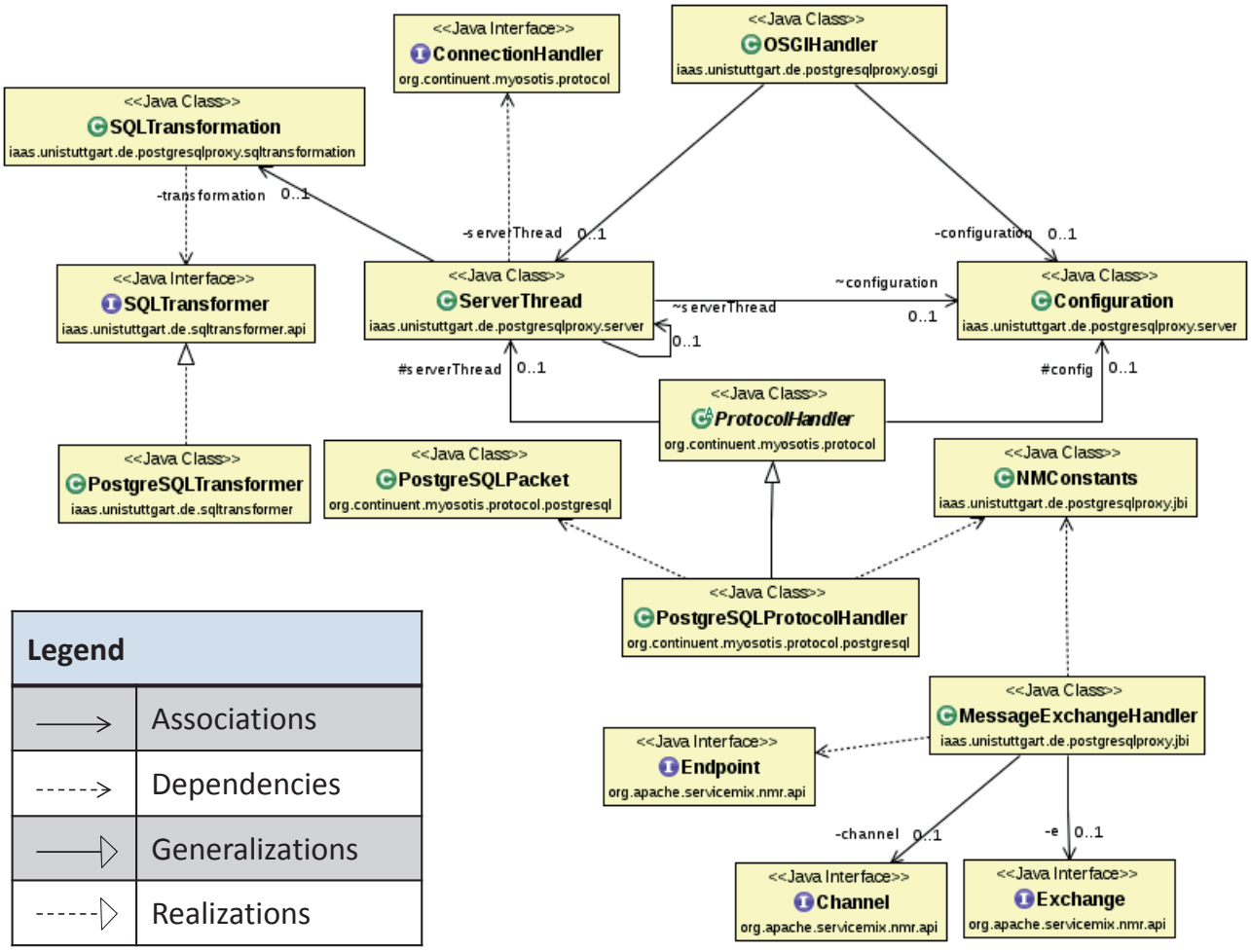

Figure 6.3.: PostgreSQL Proxy as an OSGi Bundle, Integrated into CDASMix

and operations that it is not possible to be all included in this picture. Thus, we omit them for the sake of simplicity in order to ease understanding.

CDASMix is build as an OSGi container and therefore, PostgreSQL proxy must be adapted and extended to an OSGi bundle, in order to be compliant with the rest of the system. The PostgreSQL proxy is enhanced with the class `OSGIHandler`, which implements the `BundleActivator` interface. During the bundle activation process, the OSGi container creates an instance of `OSGIHandler` and through its operations the proxy is added to the OSGi container. As we explained in Section 6.1, this class constructs the *Configuration* object, which encapsulates the server properties, as they are defined in the `cdasmix.server.cfg` file. Then, it creates an instance of the `ServerThread` class, which takes the server Configuration object as argument.

Apart from being able to activate itself and register its service, the PostgreSQL proxy bundle must be enabled to lookup and consume services from third party bundles. `OSGIHandler` is responsible for this as well. It accesses the `NMR`[3] service, and it also instantiates the *SQL-Transformation* class, which does the lookup and consumption of the `PostgreSQLTransformer` service.

The next step is to integrate the OSGi bundle of the PostgreSQL proxy with the JBI environment. This is achieved with the classes of the `iaas.unistuttgart.de.postgresqlproxy.jbi` package. The class `NMMarshaler` provides methods for marshaling incoming SQL statements and metadata to NMF and for demarshaling the received responses backwards. The `MessageExchangeHandler` class adds to the proxy bundles methods that directly invoke the dynamic routing operations in the NMR API. It creates a Camel Exchange object that is sent synchronously over NMR. The destination is a JBI endpoint URI that is dynamically created by the tenant and user UUID, as we discussed in Section 2.5. The `MessageExchangeHandler` thread waits till receiving a messeage from the JBI endpoint as a response. The *NMConstants* class contains the defined standardized naming for the properties and attachments in the NMF. `ServiceMix-camel-mt` component (see Figure 2.6), copies the properties and attachments from Camel Exchange to `JBI MessageExchange`, by using as keys the names of them defined in NMConstants class.

6.3. PostgreSQL Transformer Implementation

In the Sections 6.1 and 6.1 we discussed how the PostgreSQL transformation service is registered and how it is accessed by the PostgreSQL proxy. In this section we focus on the construction of the PostgreSQL transformer component. In Figure 6.4 is shown a simplified class diagram of it.

PostgreSQLTransformer class implements the SQLTransformer interface and overrides its single method, `transform(String, String)` (see Figure B.2), as shown in Lines 7-21 of Listing A.1. The parameters of the implemented method, `original` and `target`, are the placeholders of the source and target dialect, respectively. This method checks if source and target dialect are identical. In such a case it will directly output the original statement. However, this checking is redundant, because proxy will not lookup for the transformation service in case of similar source-target dialects. The `transform` method instantiates the `PostgreSQLParser` and calls the `Statement()` function through this instance. The parse tree will be generated and returned back. Then, the parse tree is navigated in depth-first pre-order[4], using the visitor pattern (see Appendix C). The "visit" to each node will call the `transform` function of the respective JAVA class, which will realize the node's representation according to the rules of the target dialect. In this thesis we implemented the `transform` method for each potential node.

[3]The relationships of `org.apache.servicemix.nmr.api.NMR` with the classes of Figure 6.3 are not shown for the sake of simplicity. It has associations with `OSGIHandler.class`, `ServerThread.class`, `PostgreSQLProtocolHandler.class`, and `MessageExchangeHandler.class`.

[4]Start from the root and visit each node before visiting any of its children [SW11].

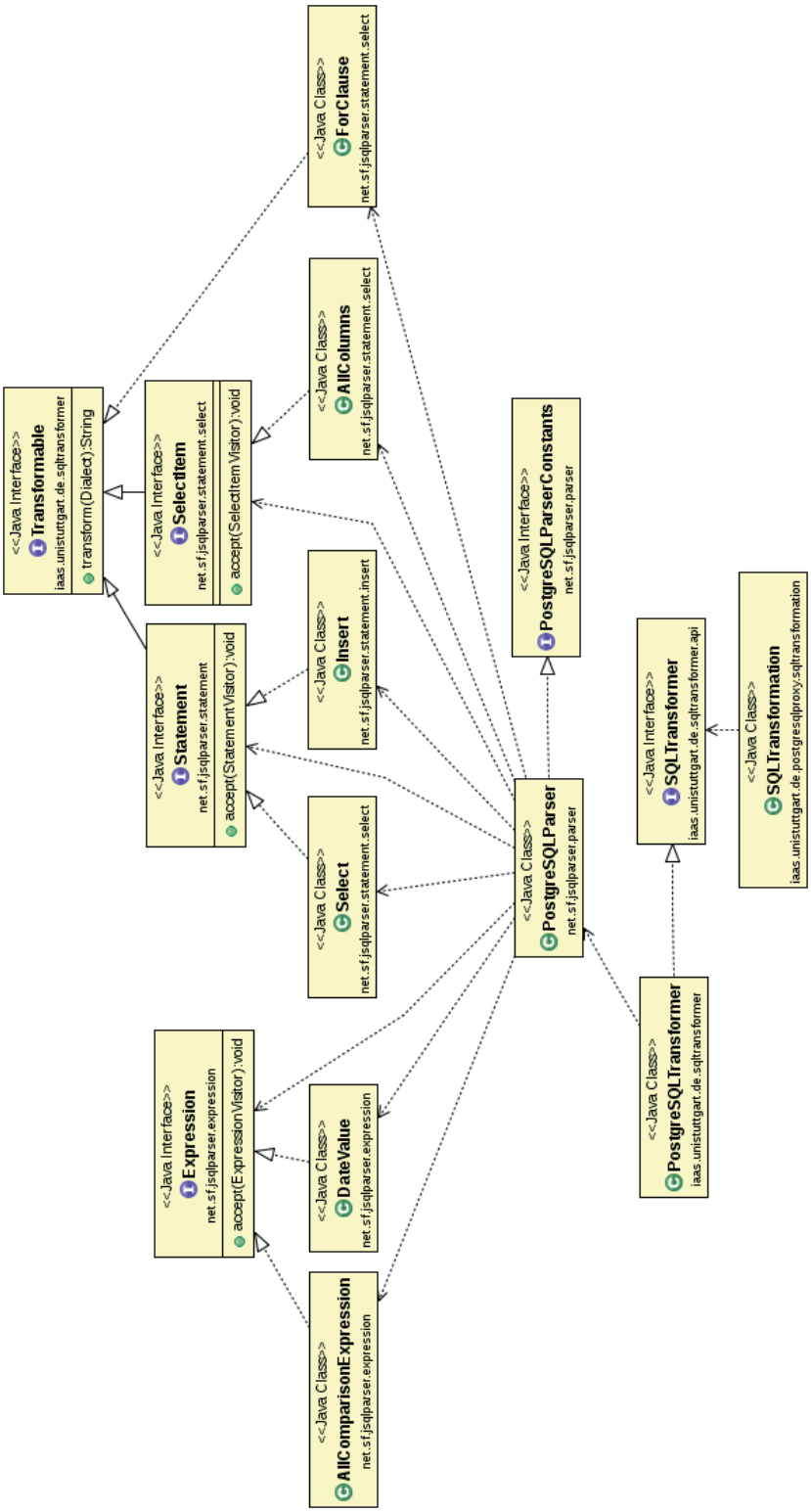

Figure 6.4.: PostgreSQL Transformer as an OSGi Bundle, Integrated into CDASMix

Each Java class encapsulates one build block of the SQL statement, (or, in other words, a node of the parse tree) and thus, there is a high amount of them. Figure 6.4 is a simplified class diagram of the transformation component, because it cannot include and represents all the Java classes that may appear in a parse tree.

6.4. SQL Parsers

The syntactical comparison among the SQL technologies involved in CDASMix was thoroughly made in [Xia13]. In this section we focus on describing how the PostgreSQL and MySQL grammar implementations are affected from the syntactical deviation of the two dialects and lead to different SQL parsers. The Listings 6.4 and 6.5 represent the PostgreSQL and MySQL systaxes of the statement `DROP table` [posa, mys].

```
1 DROP TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Listing 6.4: Syntax of the `DROP table` Statement in the PostgreSQL Dialect [posa]

```
1 DROP [TEMPORARY] TABLE [IF EXISTS]
2     tbl_name [, tbl_name] ...
3     [RESTRICT | CASCADE]
```

Listing 6.5: Syntax of the `DROP table` Statement in the MySQL Dialect [mys]

The parser logics required for each of them are shown in the segments of the respective grammar files, Listings 6.6 and 6.7, accordingly. As we discussed in Section 3.2, the source SQL statement is seen from the parser as a stream of tokens. The parse tree is built as a hierarchy of Java classes, in depth-first pre-order, based on the token the parser reads from the given statement. But before, the parser must be generated, and therefore the grammar file is required. So, after researching about the syntax of the PostgreSQL dialect, the next and most significant task of our work is to define the appropriate tokens and parsing rules[5] in terms of the tokens.

```
1  DropTable DropTable():
2  {
3      DropTable dropTable = new DropTable();
4      List tables;
5  }
6  {
7      <K_DROP><R_TABLE>
8    [
9    LOOKAHEAD(2)
10   <K_IF>
11   <K_EXISTS>
12   { dropTable.setIfExists(true); }
```

[5] A *parse rule* is also known as *production*

```
  ]
  tables = TableList()
  { dropTable.setTables(tables); }
    [
  <K_CASCADE>
  { dropTable.setCascade(true); }
  |
  <K_RESTRICT>
  { dropTable.setRestrict(true); }
  ]
    {
        return dropTable;
    }
}
```

Listing 6.6: Snippet of the PostgreSQL Grammar File, Responsible for the `DROP table` Statement

```
DropTable DropTable():
{
    DropTable dropTable = new DropTable();
    Table table;
    List tablesList = new ArrayList();
}
{
    <R_DROP>
  [
  <K_TEMPORARY>
  { dropTable.setTemporary(true); }
  ]
    <R_TABLE>
    [
  <R_IF >
  <R_EXISTS>
  { dropTable.setIf_exists(true); }
  ]
    table = Table()
  { tablesList.add(table); }
  (
  "," table = Table()
  { tablesList.add(table); }
  )*
    [
  <R_RESTRICT >
  { dropTable.setRestrict(true); }
  |
  < R_CASCADE >
  { dropTable.setCascade(true); }
  ]
    {
```

```
33          dropTable.setTablesList(tablesList);
34          return dropTable;
35      }
36  }
```

Listing 6.7: Snippet of the MySQL Grammar File, Responsible for the `DROP table` Statement

In both dialects, the SQL reserved keywords are prefixed with R_ to avoid name clashes, while the SQL non-reserved keyword with K_. The former can never be used as identifiers, while the latter have special meaning in particular contexts and can be used as identifiers otherwise. As shown in Listings 6.6 and 6.7, the two dialects considered in this section have differences on the sets of reserved and non-reserved keywords [keyb, keya]. Each token definition is enclosed in angle-brackets, (< and >). We see that the tokens `DROP`, `IF`, `EXISTS`, `RESTRICT`, `CASCADE` have the prefix K_ in PostgreSQL, while in MySQL they are defined as reserved keywords, namely with the prefix R_. The token `TABLE` on the other hand is treated as a preserved keyword in both dialects, and thus, in both grammar files it appears as `R_TABLE`.

In the Appendix A, the Listings A.4 and A.5 show the emitted Java codes for the PostgreSQL and MySQL parsers that will transform the statement into a parse tree, identical for both dialects. Along with the parser `PostgreSQLParser.java` (or `MySQLParser.java`), is generated the file `PostgreSQLParserTokenManager.java` (or `MySQLParserTokenManager.java`) as well, which is the lexical analyzer. The latter breaks the given SQL statement into a sequence of tokens and identifies the kind of each token. For example, the kind of the token `DROP` or `drop` is `K_DROP` and `R_DROP` in PostgreSQL and MySQL, respectively.

The method `jj_consume_token` (e.g. see Line 6 in Listing A.4) takes as an argument an expected token kind defined by the grammar and tries to obtain a token of that kind from the lexical analyser, which reads the actual input statement. If the next token has a different kind, then an exception is thrown. The expression `(jj_ntk==-1)?jj_ntk():jj_ntk`[6] calculates the kind of the next unread token (e.g. see the Lines 17 and 20 of the Listing A.4).

The LOOKAHEAD(2) directive (see Line 9 in the Listing 6.6), tells JavaCC that it must check the next *two* symbols before making any decision about the parse rule. The number of the next symbols needed to be checked, is defined by the argument. It is placed at the choice point[7], where the decision must be made whether the input statement contains the pair of tokens (IF, EXISTS) or not. The LOOKAHEAD directive could be omitted as well (see Listing 6.7) and in such a case it is assumed that its argument has the default value 1. If we include the LOOKAHEAD directive (with an argument value different from the default 1), JavaCC assumes that the programmer knows what she/he is doing and so, it generates the parser without throwing warnings. However, the larger the argument value, the slower the generated parser.

In conclusion, we can state that grammar file actually is an aggregation of regular expressions, defining portions of text to be matched. The proper implementation of it is the cornerstone

[6] the abbreviation *ntk* stands for Next ToKen
[7] Points in the grammar file where more than one rule might match.

for a functional and efficient parser, which will lead to an efficient transformation service as well.

7. Validation and Evaluation

In this chapter, we validate and evaluate the PostgreSQL transformer component developed in this thesis, as a standalone module. Our goal is to verify that the enhancement of CDASMix system with the PostgreSQL transformation functionality fulfills the requirements defined in Section 4.5. The new component first is examined separately from CDASMix, before its integration into it, in order to isolate it from external operation and performance variables.

7.1. Validation of SQL Parser and Transformation

The integration of the PostgreSQL transformer into the CDASMix is discussed in the previous sections and depicted in Figures 2.6, 4.1, and 6.2. In the two following sections we validate and evaluate the PostgreSQL transformer as a standalone module. After the integration into CDASMix, the overall system functionality can be validated and evaluated via Apache JMeter. It offers listeners[1] for an automated *Summary Report* and *Response Time Graphs*. The former aggregates all information regarding the runtime and throughput of the system as well as the occurred errors. The results are automatically visualized with the *Response Time Graph*.

We created three database instances in Amazon RDS[2], one for each target database technology supported by CDASMix. The characteristics of them are listed in Table 7.1. Each of them is populated with the TPC-H database schema shown in Figure 7.1, modified according to the rules of the target database. The primary keys of the tables are underlined. One notices that the tables `PARTSUPP` and `LINEITEM` use the combination of foreign keys, `(PARTKEY, SUPPKEY)`, as a unique primary key. The queries used for the system validation and evaluation are generated based on the TPC-H benchmark, unless denoted explicitly otherwise. Following the TPC-H, database schema 1 GB of example data is generated and stored into the three database instances running on Amazon RDS.

DB Name	DB Type	DB Instance Class
tpch	MySQL 5.6.22	db.t1.micro
psqldb	PostgreSQL 9.4.1	db.t1.micro
oracledb	Oracle SE 11.2.0.4.v3	db.t1.micro

Table 7.1.: Tenant Data Source Registration

[1] Feature of Apache JMeter that "listens" to the test results and also provides means to view, save, and read test results.

[2] Amazon Relational Database Service (Amazon RDS): http://aws.amazon.com/ rds/.

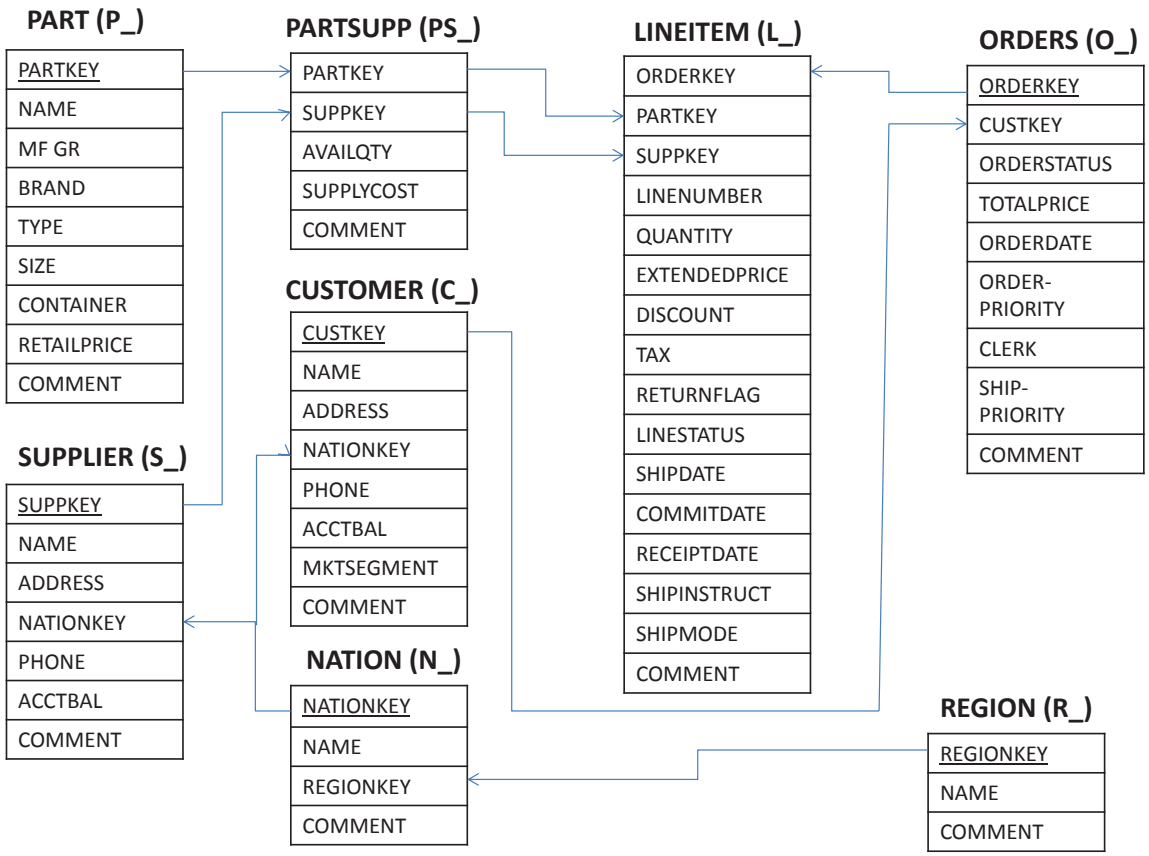

Figure 7.1.: TPC-H Database Schema Diagram [Tra13].

In Appendix A one can find Listings A.2 and A.3, which are examples of the JUnit test cases, used for the validation and evaluation of the PostgreSQL transformer, respectively. The first test case ensures the proper operation of the transformer by comparing the transformed source statement to the corresponding target statement. For each source SQL statement we can precisely predict the expected target statement after transformation. The test case in the Listing A.3 outputs the runtime of the transformer, and by inversing it we can calculate the throughput as well. By excluding Line 12 (`transformed = stmt.transform(Dialect.MySQL);`) from the Listing, we extract the parsing time and calculate its throughput.

7.2. Validation of PostgreSQL Transformation

In this section, the functionality of the PostgreSQL transformer is validated as a standalone module and the results are shown in the Table 7.2. For the validation JUnit test cases are used. An example is shown in Listing A.2.

Dialect	Statement
PostgreSQL	CREATE TABLE nation(n_nationkey NUMERIC(3,0) NOT NULL PRIMARY KEY, n_regionkey INTEGER NOT NULL, n_comment VARCHAR(152));
MySQL	CREATE TABLE nation (n_nationkey TINYINT NOT NULL, n_name CHAR(25) NOT NULL, n_regionkey INTEGER NOT NULL, n_comment VARCHAR(152));
Oracle	CREATE TABLE nation (n_nationkey NUMBER(3,0) NOT NULL, n_name CHAR(25) NOT NULL, n_regionkey INTEGER NOT NULL, n_comment VARCHAR(152));
PostgreSQL	DELETE FROM lineitem WHERE l_linestatus='O';
MySQL	DELETE FROM lineitem WHERE l_linestatus='O';
Oracle	DELETE FROM lineitem WHERE l_linestatus='O';
PostgreSQL	DROP TABLE supplier CASCADE;
MySQL	DROP TABLE supplier CASCADE;
Oracle	DROP TABLE supplier CASCADE CONSTRAINTS;
PostgreSQL	SELECT l_returnflag, l_linestatus, SUM(l_quantity) as sum_qty, SUM(l_extendedprice * (1 -l_discount)) as sum_disc_price, AVG(l_quantity) as avg_qty, COUNT(*) as count_order FROM lineitem WHERE l_shipdate <= date ('1998-12-01') - interval '1' day LIMIT 10 OFFSET 3;
MySQL	SELECT l_returnflag, l_linestatus, SUM(l_quantity) as sum_qty, SUM(l_extendedprice * (1 -l_discount)) as sum_disc_price, AVG(l_quantity) as avg_qty, COUNT(*) as count_order FROM lineitem WHERE l_shipdate <= DATE_SUB((date '1998-12-01' , interval '1' day) LIMIT 3, 5;
Oracle	SELECT l_returnflag, l_linestatus, SUM(l_quantity) as sum_qty, SUM(l_extendedprice * (1 -l_discount)) as sum_disc_price, AVG(l_quantity) as avg_qty, COUNT(*) as count_order FROM lineitem WHERE l_shipdate <= date ('1998-12-01') - interval '1' day FETCH NEXT 10 ROWS ONLY;
PostgreSQL	SELECT n_name FROM customer, orders WHERE c_custkey = o_custkey AND o_orderdate < date '1998-10-09' + interval '1' year
MySQL	SELECT n_name FROM customer, orders WHERE c_custkey = o_custkey AND o_orderdate < DATE_ADD(date('1998-10-09'), interval 1 year);
Oracle	SELECT n_name FROM customer, orders WHERE c_custkey = o_custkey AND o_orderdate < date '1998-10-09' || interval '1' year;
PostgreSQL	INSERT INTO orders (o_orderkey, o_custkey, o_orderstatus, o_totalprice, o_orderdate, o_orderpriority, o_clerk, o_shippriority, o_comment, o_image) VALUES (5970822, 69644, 'O', 71366.76, TIMESTAMP '1996-03-29 10:20:30.0', 'Clerk#000000868', 0, 'comments long text', E'\\x53696d696e0d0a');
MySQL	INSERT INTO orders (o_orderkey, o_custkey, o_orderstatus, o_totalprice, o_orderdate, o_orderpriority, o_clerk, o_shippriority, o_comment, o_image) VALUES (5970822, 69644, 'O', 71366.76, TIMESTAMP '1996-03-29 10:20:30.0', 'Clerk#000000868', 0, 'comments long text', 0x53696d696e0d0a));
Oracle	INSERT INTO orders (o_orderkey, o_custkey, o_orderstatus, o_totalprice, o_orderdate, o_orderpriority, o_clerk, o_shippriority, o_comment, o_image) VALUES (5970822, 69644, 'O', 71366.76, TIMESTAMP '1996-03-29 10:20:30.0', 'Clerk#000000868', 0, 'comments long text','53696d696e0d0a');
PostgreSQL	UPDATE orders SET o_orderdate=NOW() WHERE o_orderstatus='F';
MySQL	UPDATE orders SET o_orderdate=NOW() WHERE o_orderstatus='F';

Dialect	Statement
Oracle	UPDATE orders SET o_orderdate=CURRENT_TIME() WHERE o_orderstatus='F';

Table 7.2.: SQL Transformation Validation with Cloud Databases

SQL statement transformation consists of data types transformation and adaption to the syntactical rules of the target dialect. There are also many cases where the transformation from one dialect to the other simply needs to regenerate the source statement. Table 7.2 lists the different types of statements that are covered by the `PostgreSQLParserCC.jj` grammar file and implemented in the PostgreSQL transformer (see Section 6.3). We provide support for all the statement types existing in SQL. However, there are restrictions on the data types and syntactical features we can transform. The capabilities of the PostgreSQL transformer developed in this thesis are depicted in the examples of the said Table 7.2.

7.3. Performance Evaluation

CDASMix performance can be defined as the amount of successfully executed statements over time and resources used. The transformation service, as an add-on enhancement to the existing CDASMix, comes with an unavoidable performance drawback. However, the higher the performance of the transformation functionality, the less its negative effects to the performance of the CDASMix system. The total transformation time of a statement consists of the time needed to parse it (parsing time) and the time to construct the target statement from the parse tree (transforming time). In this section the performance of the added PostgreSQL transformer is evaluated as a standalone module, in terms of complexity of the parsing and transformation. We measure the transformation time of the statement and the statement throughput, which is defined as the amount of statements parsed and transformed in one unit of time.

However, two instances of the same statement transformation can give different timing characteristics due to several sources of non-determinism involved [Osi10]. Some non-deterministic factors can be: *memory allocation*, which can assign different pages of the allocated virtual addresses, for different instances of the process: *thread scheduling* and *system events*, which can randomly interrupt the execution of a transformation. Moreover, for applications running in JVM, such as CDASMix, Just-In-Time (JIT) compilation[3] and garbage collection are additional sources of non-determinism [GBE07].

These result in random variations of the measured time and throughput of statement transformation. Statistic theory can be used to handle the runtime non-determinism and to interpret the measured data [GBE07]. For this we need to measure the variable, namely the transformation time, multiple times. The test case shown in Listing A.3 of the Appendix A is used. As we can see, the time required to perform 10.000 sequential transformations of a statement

[3] In many virtual machines the same program may have different execution times due to the use timer-based optimizations [AFG+05]

is measured. This measurements are done 100 times. Hence, we perform the transformation process 1 million times for each statement, in order to define the speed of the transformer as the average time of all (mean value). By inverting the time consumption, the throughput (number of statement/second) is calculated. The mean value leads us to the most accurate approach of the transformation time.

In the case of PostgreSQL transformer evaluation, the time measurements of statement transformation are independent from each other, as each execution is a new process. They are also random because of the non-determinism added by the environment in which they run, as we discussed previously.

The *Law of the Large Numbers* ensures that we resolve the dependency of each measurement on the non-deterministic runtime environment, by getting a large amount of measurements. According to the law, the average of the measurements should be close to the expected value and will tend to become closer for a larger number of measurements [Lin93]. Additionally, the *Central Limit Theorem* states that *the distribution of the average of a large number of independent, identically distributed random variables (or measurements in our case) will be approximately normal, regardless of the underlying distribution* [Bar11]. Hence, we chose to perform the same transformation process 1.000.000 times, get multiple measurements, and consider their average as the expected runtime value.

The parsing and transforming, which are the two sequential steps of the PostgreSQL transformer, as shown in Figure 3.2, are both expected to be of linear time complexity $O(n)$, where n is the number of nodes. From the *superposition principle*, the PostgreSQL transformer will also be linear to the number of nodes. Let's assume that $p(n)$ is the parsing time for a source statement, which will be recomposed into a parse tree of n nodes, and $t(n)$ is the transforming time of generating the target statement from the parse tree. The superposition principle states that:

$$if \quad s_1 = p(n_1) \quad and \quad s_2 = t(n_2) \quad then \quad T(n_1 + n_2) = p(n_1) + t(n_2) \tag{7.1}$$

In the case of query transformation n_1 is equal to n_2, because the number of nodes one query has, do not change during the transformation process.

But, why linear time complexity $O(n)$? The result of the parsing is the parse tree and an example of it is shown in Figure 7.2. It is called a *k-ary tree*, because each node of it has no more than k children.

During the parsing step the heapifying[4] of the parse tree is realized and the transforming follows with a depth-first pre-order search of the tree [SW11]. Because of the way the tree is constructed the number of nodes of depth i is exactly k^i for all $i < l$. Let l be the maximum depth of any node in a subtree, and h the depth of the leaves (namely, the height of the tree). The number of nodes at depth l of the k-tree is exactly:

[4]The goal of heapifying is to build and establish the heap property in the whole parse tree.

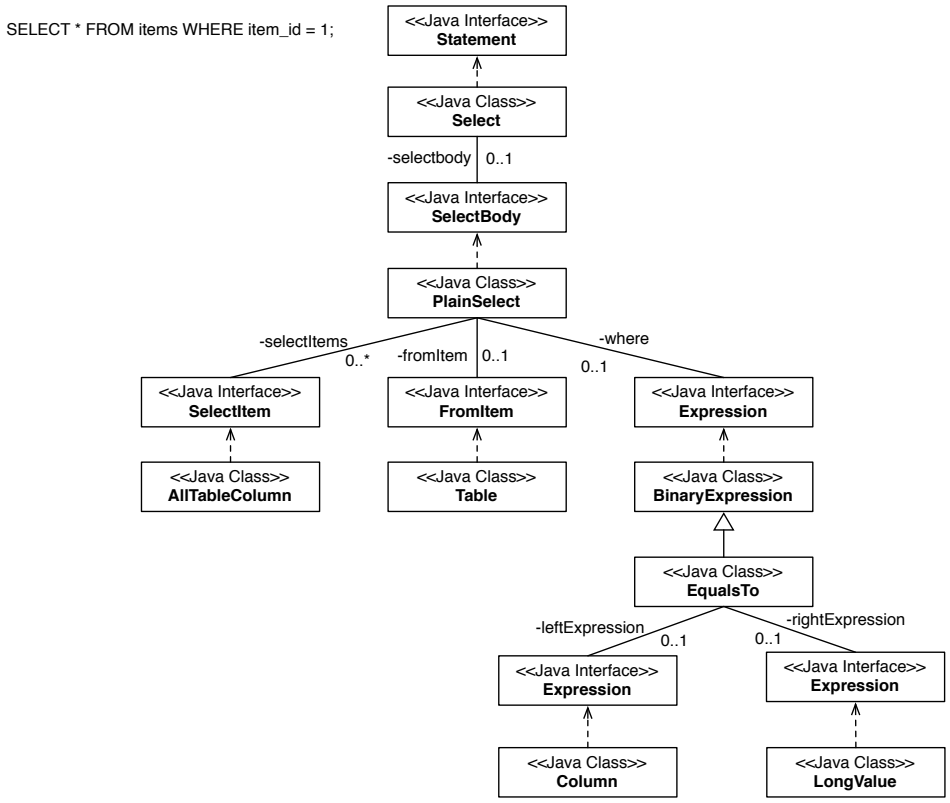

Figure 7.2.: SELECT Statement's Parse Tree in Class Diagram [Xia13].

$$n - \sum_{i=0}^{l-1} k^i = n - (k^l - 1) \tag{7.2}$$

The number of comparisons for the leaves is $O(h)$, with $O(1)$ time for each comparison. Therefore the running time is at most:

$$\begin{aligned} \sum_{i=0}^{h-1} k^i \cdot O(h-i) &= O\left(\sum_{i=0}^{h-1} k^i (h-i)\right) = O\left(\sum_{i=0}^{h-1} k^i \sum_{j=i+1}^{h} 1\right) \\ &= O\left(\sum_{j=1}^{h} \sum_{i=0}^{j-1} k^i\right) = O\left(\sum_{j=1}^{h} k^j\right) = O(k^{h+1}) = O(n). \end{aligned} \tag{7.3}$$

In the second equation $h-1$ is written as a sum of ones, and in the third equality the order of summations is exchanged[5].

[5]The Equations 7.2 and 7.3 are reused, but adapted from [Mil12], where they are applied for a special case of k-trees, the binary trees (where $k = 2$).

With this assumption, we then proceed by categorizing SQL statements based on their resulting parse trees' total number of nodes, and evaluate the time consumption and the throughput of parsing and transforming the statements. In the Tables 7.3 and 7.4 are represented the times to parse a statement and the overall transformation time (parsing and transforming) for both MySQL and PostgreSQL transformer. The target dialect is `PostgreSQL 9.4.1` and `MySQL 5.6.22`, respectively. The majority of the statements is obtained from the TPC-H benchmark. They are listed by number of nodes in ascending order. The former table contains a mixture of various statements, while the latter contains only SELECT statements.

Statements	Nodes	MySQL		PostgreSQL	
		Parsing Time (*s*)	**Total Time** (*s*)	**Parsing Time** (*s*)	**Total Time** (*s*)
`drop table lineitem;`	3	0.000007105	0.000028497	0.000005934	0.000006277
`drop table if exists lineitem, region;`	5	0.000006816	0.000029394	0.000007272	0.000006609
`select * from lineitem;`	8	0.000008829	0.000033343	0.000010124	0.000010164
`create table region2 (r_regionkey int, r_name varchar);`	10	0.000009962	0.000038029	0.000008199	0.000011027
`update lineitem set l_orderkey=15000 where l_partkey=100000000000;`	13	0.000013892	0.000042708	0.000017798	0.000017043
`select n_name from nation where n_regionkey=1;`	17	0.000019789	0.00004689	0.000018737	0.000019872
`insert into region (r_regionkey, r_name, r_comment) values (5, 'ASIA', 3+2);`	18	0.000015267	0.000043151	0.000016181	0.000015355
`select sum(o_totalprice) from orders where o_orderstatus='O' group by o_orderkey limit 20 offset 10 ;`	25	0.000024342	0.000059162	0.000027731	0.000026209
`select n_regionkey from nation where n_name in ( select r_name from region where r_regionkey= 10);`	31	0.000044037	0.000086838	0.000045659	0.000048532
`select c_name, c_address from customer, orders where o_orderdate=NOW() union select ps_suppkey from partsupp where ps_partkey=1 order by ps_suppkey limit 100 offset 0;`	41	0.000035539	0.000077677	0.000036035	0.000077677
`select c_custkey, o_orderkey, sum(o_totalprice) from (select * from orders cross join customer where o_orderpriority = 'URGENT' and c_custkey> 1000 order by o_totalprice limit 100 offset 0) as sum;`	51	0.000052286	0.000108058	0.0000433	0.000108058

✓[1] 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	0.000037307	0.000122866	0.000042333	0.000055283
select c_name, c_address from (select * from customer) join (select * from supplier where s_phone='USER_08873') join (select * from part) join (select * from orders) where o_orderdate=TODAY() AND o_orderstatus='F' limit 10 offset 0;	71	0.000052684	0.000114584	0.000049461	0.000063572
✓[1] (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 100 offset 0;	80	0.000056304	0.000124342	0.000053372	0.000124342
✓[1] 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 1000 offset 1;	100	0.000079816	0.000163986	0.000078605	0.000087163
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 100 offset 0;	125	0.000090581	0.000207711	0.000094025	0.000119122

`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	0.000102352	0.000218036	0.000084183	0.000119624
`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	0.000141075	0.000286147	0.000132546	0.000155821
`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	0.00016848	0.000302252	0.000144116	0.000167569

Table 7.3.: Time Evaluation of Various Statements

Statements	Nodes	MySQL		PostgreSQL	
		Parsing Time (*s*)	**Total Time** (*s*)	**Parsing Time** (*s*)	**Total Time** (*s*)
`select * from lineitem;`	8	0.000008829	0.000033343	0.000010124	0.000010164
`select * from nation where n_regionkey=1;`	15	0.000015544	0.000045239	0.000016898	0.000017486
`select n_name from nation where n_regionkey=1;`	17	0.000019789	0.00004689	0.000018737	0.000019872
`select avg(o_totalprice) from orders where o_orderstatus='0';`	20	0.000022024	0.000051509	0.000020699	0.000020972
`select sum(o_totalprice) from orders where o_orderstatus='0' order by o_orderkey limit 20 offset 10 ;`	25	0.000024342	0.000059162	0.000027731	0.000026209
`select n_regionkey from nation where n_name in ( select r_name from region where r_regionkey= 10);`	31	0.000044037	0.000086838	0.000045659	0.000048532
`select c_name, c_address from customer, orders where o_orderdate=NOW() union select ps_suppkey from partsupp where ps_partkey=1 order by ps_suppkey limit 100 offset 0;`	41	0.000035539	0.000077677	0.000036035	0.000077677
`select c_custkey, o_orderkey, sum(o_totalprice) from (select * from orders cross join customer where o_orderpriority = 'URGENT' and c_custkey> 1000 order by o_totalprice limit 100 offset 0) as sum;`	51	0.000052286	0.000108058	0.0000433	0.000108058
✓[1] `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	0.000055155	0.000109287	0.000048619	0.000057968
`select c_name, c_address from (select * from customer) join (select * from supplier where s_phone='USER_08873') join (select * from part) join (select * from orders) where o_orderdate=TODAY() AND o_orderstatus='F' limit 10 offset 0;`	71	0.000052684	0.000114584	0.000049461	0.000063572

✓[1] (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 100 offset 0;	80	0.000056304	0.000124342	0.000053372	0.000124342
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	0.000052339	0.00011454	0.000048215	0.000055184
✓[1] 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 1000 offset 0;	100	0.000079816	0.000163986	0.000078605	0.000087163
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 100 offset 0;	125	0.000090581	0.000207711	0.000094025	0.000119122
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	0.000102352	0.000218036	0.000084183	0.000119624

`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	0.000141075	0.000286147	0.000132546	0.000155821
`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	0.00016848	0.000302252	0.000144116	0.000167569

Table 7.4.: Time Evaluation of SELECT Statements

1. The queries of the Tables 7.3 and 7.4 started with a checkmark (✓), are gotten from the set of evaluation queries used in [Xia13].

The following Diagrams, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, and 7.9 visualize the results of Tables 7.3 and 7.4 regarding the relationship between the number of nodes over the consumed time, as well as the throughput, for statement parsing and transformation. We notice that in all cases, the time follows a linear trend over statement complexity, which is described by the number of nodes.

The first pair of the Diagrams, 7.3 and 7.4, shows the relationship of time and throughput over the number of nodes for the case of the PostgreSQL transformer. Two trendlines are depicted in each diagram, one for the set of `SELECT` statements and the other for the mixed set of statements (which contains `SELECT`, `UPDATE`, `INSERT INTO`, `CREATE TABLE`, `DELETE` and `DROP TABLE` statements). For both load testings the time gives a linear interpolation while the throughput interpolation is a hyperbolic function. They both prove the linearity of the

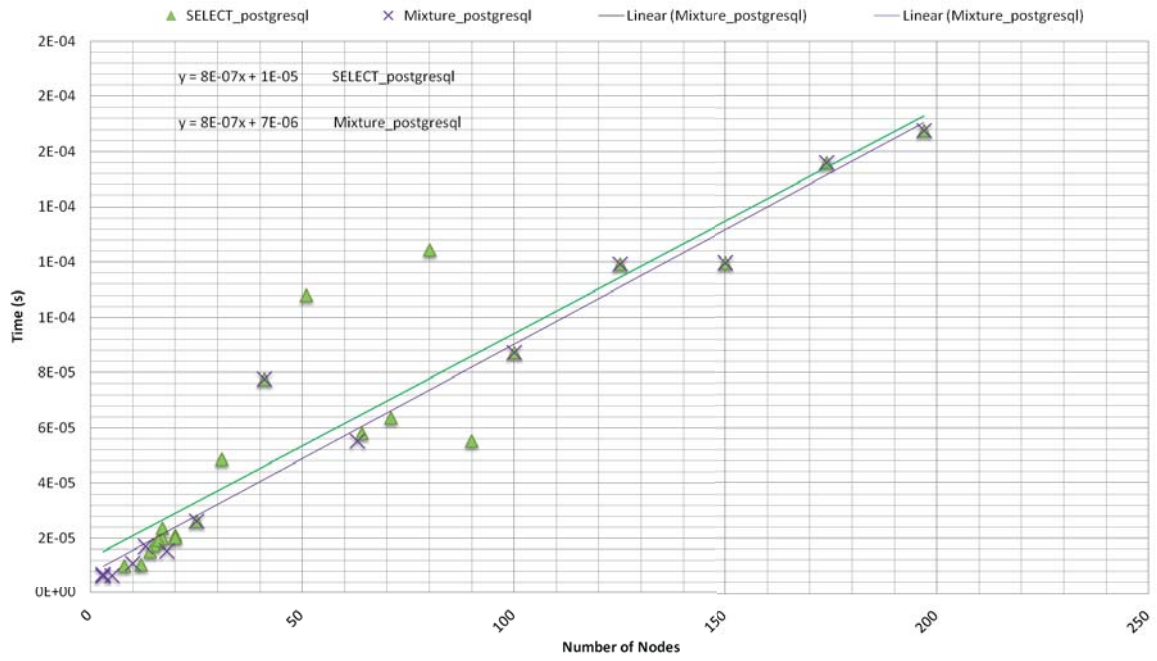

Figure 7.3.: Plot of the Time over Number of Nodes for the PostgreSQL Transformer, for two Load Testings

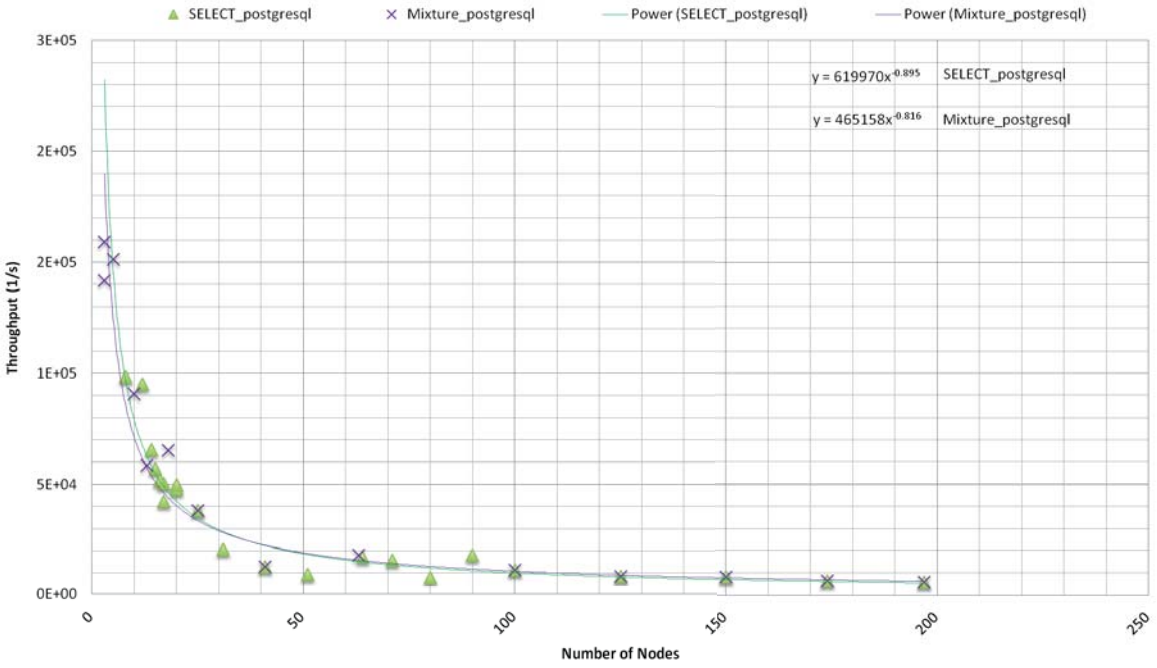

Figure 7.4.: Plot of the Throughput over Number of Nodes for the PostgreSQL Transformer, for two Load Testings

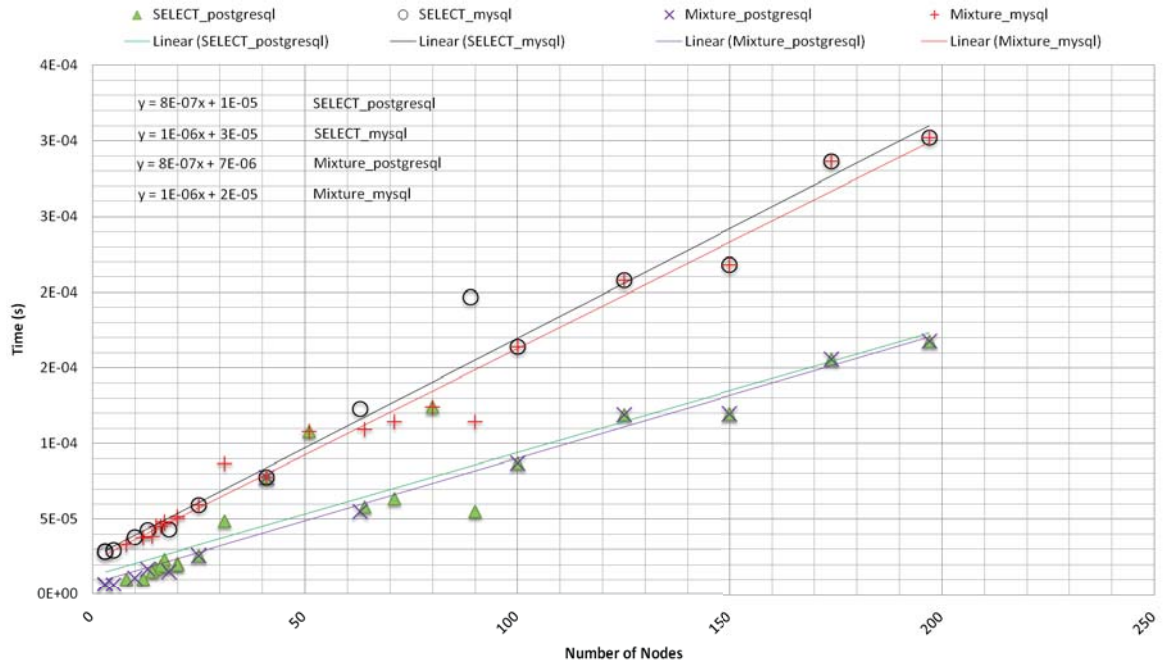

Figure 7.5.: Plot of the Time Consumption over Number of Nodes for the PostgreSQL and MySQL Transformer

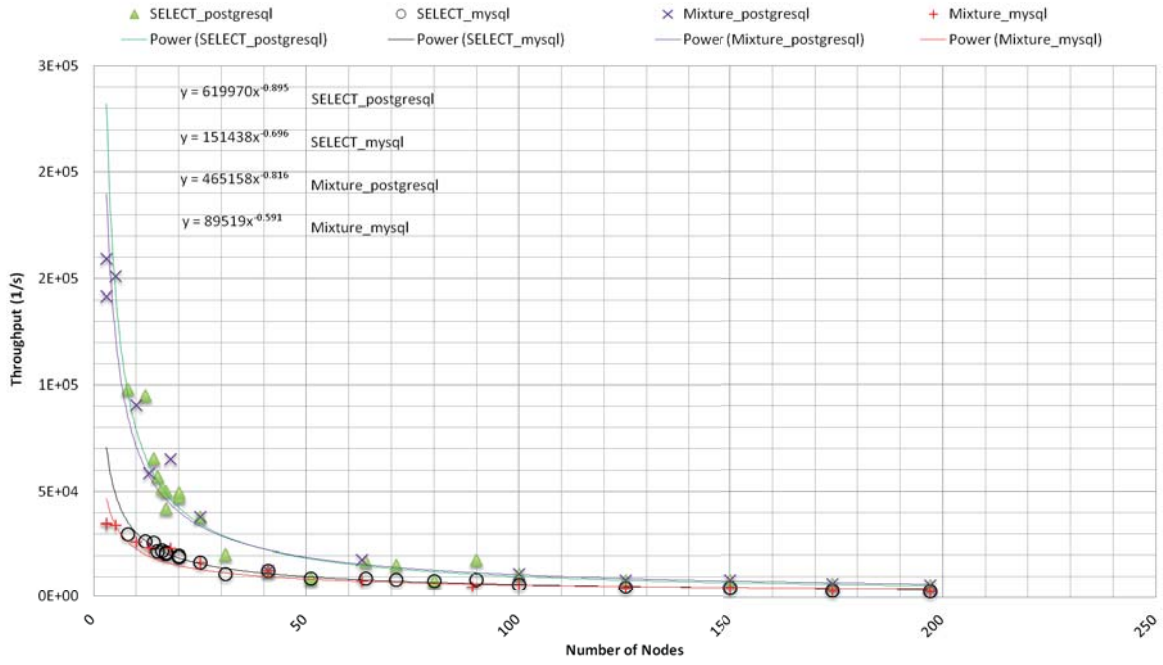

Figure 7.6.: Plot of the Throughput over Number of Nodes for the PostgreSQL and MySQL Transformer

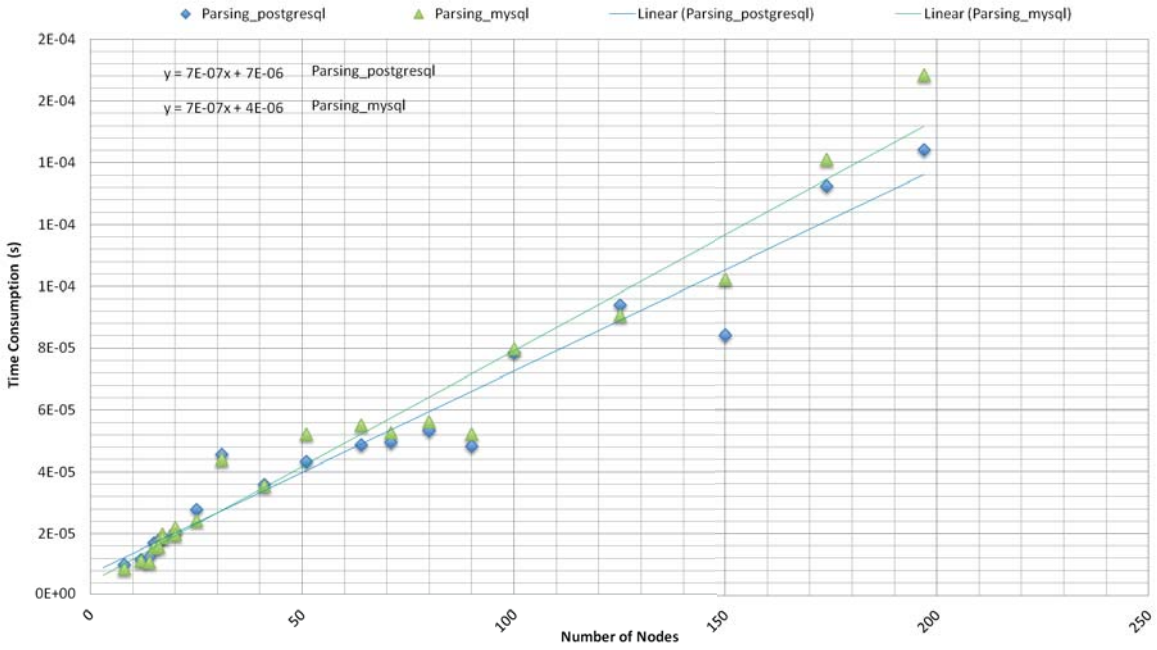

Figure 7.7.: Plot of the Parsing Time Consumptions of SELECT Statements over the Number of Nodes for the PostgreSQL and MySQL Transformer

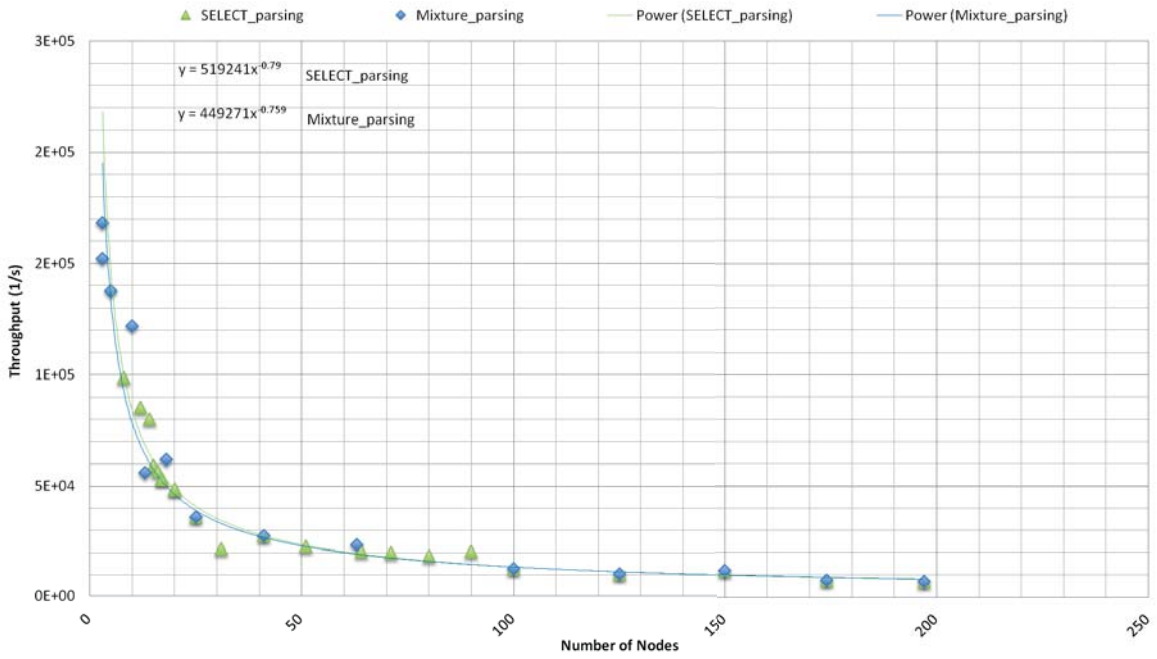

Figure 7.8.: Plot of the Throughput over Number of Nodes for PostgreSQL Transformer for both Load Testings

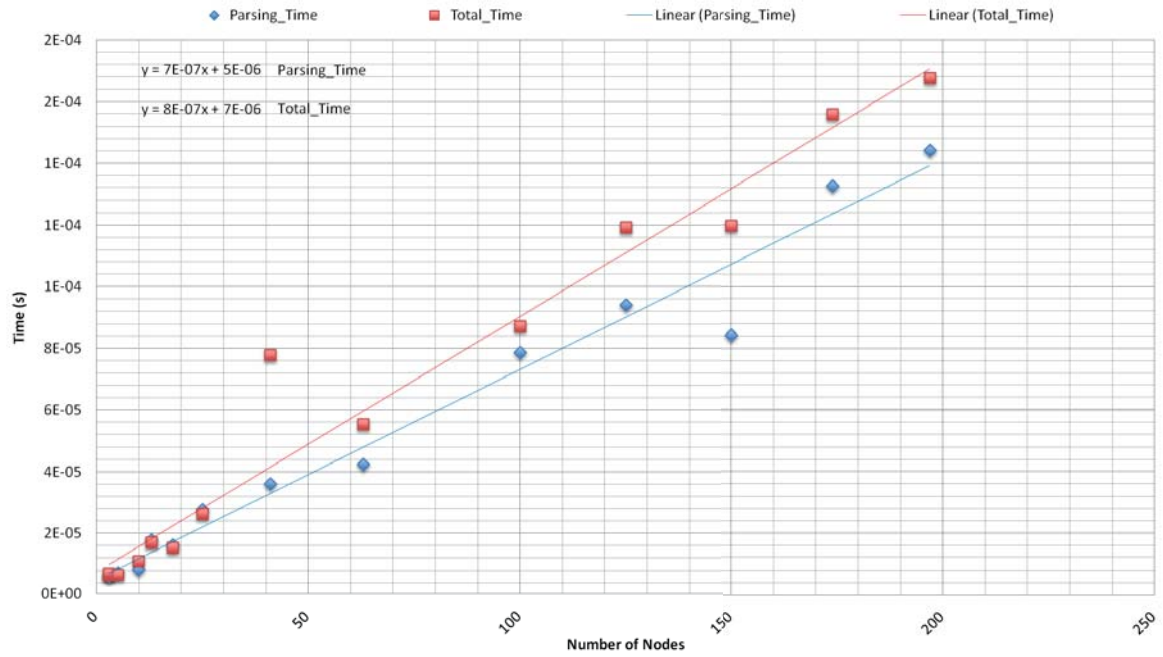

Figure 7.9.: Plot of the Transformer Time Consumption and Parsing Time Consumption, over Number of Nodes for the PostgreSQL Transformer

PostgreSQL transformer over the number of nodes and thus, the $O(n)$ time complexity of this component.

The second pair of Diagrams, 7.5 and 7.6 is obtained after adding to the Diagrams 7.3 and 7.4 the respective results of the MySQL transformer (see Tables 7.3 and 7.4), as they were presented in [Xia13]. For both transformers the basis for the measurements are the same queries, of course adapted in order to be compliant with the specific dialect features. We see that both transformers coincide with the $O(n)$ notion. However, the PostgreSQL transformer is more efficient, especially for larger number of nodes, because, as we can see in the Diagram 7.5, the gradient of its trendline is lower. The reason is the deviation of the two dialects and subsequently the need to handle them with different grammar files[6]. This leads to different parsers and different implementations of the `transform()` methods.

In the Diagram 7.7 the parsing times of MySQL and PostgreSQL transformer for SELECT statements are compared. We see that for queries with a large number of nodes, the PostgreSQL parser tends to be more efficient. However, this result is highly dependent on the design of the grammar file. Future extensions of the grammar files may reverse the current outcome.

In the Diagram 7.8 the throughput of the PostgreSQL parser over the number of nodes for the

[6]The grammar file of the MySQL parser is built based on the syntax guideline given in "https://dev.mysql.com/doc/refman/5.0/en/sql-syntax-data-definition.html", while the PostgreSQL grammar file follows the syntactical rules given in "http://www.postgresql.org/docs/9.0/static/sql-syntax.html".

two loads is depicted and we see that the interpolation remains hyperbolic.

Lastly, Diagram 7.9 represents the relationship of PostgreSQL parsing and total transformation time, over the number of nodes, for the load of mixed statements. We see that the deviation of the two trendlines increases with the number of nodes. This result is in accordance with our expectations from the Equation 7.1; the deviation of these is equal to the transforming time, which is also increasing in a linear trend over the number of nodes.

8. Conclusion and Future Work

A big challenge of this thesis, apart from the development of the components as standalone modules, was their integration to the existing system. Dealing with system integration is hard as it requires a good understanding of all the involved components and technologies. On the other hand, time constrains make it unfeasible to elaborate with all of them. Finding out in which depth your understanding and knowledge over the surrounding environment should reach is the key to success.

8.1. Conclusion

Nowadays, the trend in companies is to look for moving business services into the Cloud. As was expected in [Tec], technologies that realize the migration of existing applications to Cloud and/or access support for the migrated data, took off in 2015. CDASMix is a system that enables multi-tenant communication with Cloud or local databases of different technologies. A strong advantage of CDASMix is that it supports the seamless migration of the applications; adaptations made on the DAL of one application suffices for establishing the communication of it with the Cloud backend data store, and therefore, the migration is transparent to the higher layers of the application.

However, the current market of database technologies and Cloud service providers is wide and vividly changing. In order to be a useful data access tool, CDASMix must be able to operate as a unified cloud data access interface that will serve the communication for different database vendors, even if the source and target data source are built on different database technologies. The initial CDASMix system that we had at the beginning of this thesis, was able to successfully support the MySQL communication protocol and to provide access to MySQL, PostgreSQL, and Oracle backend data stores. A transformation service able to convert MySQL queries into PostgreSQL or Oracle was already integrated. The goal of this work was to extend on CDASMix component and enable the support for a PostgreSQL communication protocol as well as the transformation of said PostgreSQL statements into MySQL and Oracle.

In the previous chapters we present the step-by-step realization of this goal. In Chapter 2 the fundamental knowledge and background information related to this thesis were provided. In Chapter 3 the state of the art this work is based on, was discussed and positioned towards the work, respectively. Chapter 4 analyzed the old version of CDASMix, clarified the needed enhancements and defined the functional and non-functional requirements the extended system must fulfill. Based on these requirements, in Chapter 6 we illustrated and thoroughly discussed the extensions of CDASMix. Code segments, class diagrams, and Figure 6.2 were

given to clarify the functionality of the PostgreSQL proxy and PostgreSQL transformer, as well as the declaration of the transformation service and the service lookup.

The output of our work enables CDASMix to receive, transform, and deliver PostgreSQL statements to MySQL or Oracle data stores and to receive their response. In Chapter 7 we present the validation and evaluation of the transformation functionality. It is shown that the runtime of the PostgreSQL transformer is linear over the complexity of the SQL queries. The complexity of a query is defined by the number of nodes its parse tree is made of. The results regarding the performance of the PostgreSQL transformer are the indicators of the performance drawback added to CDASMix with the integration of the new functionality. Our measurements show that the PostgreSQL transformer has a speed advantage over the MySQL transformer.

8.2. Future Work

There are several issues arising from this work which should be pursued in the future.

First of all, CDASMix could be extended further, in order to provide access for a larger range of source and target SQL dialects. Currently it supports MySQL and PostgreSQL as source dialects and it is able to access MySQL, PostgreSQL, and Oracle backend data sources. In the future, other SQL proxies and transformation functionalities could be integrated. In addition, the OSGi bundle of the Camel CDASMix JDBC component (depicted in Figure 2.6 as CDASMixjdbc block) could be extended to provide communication support to more target database systems.

For the two supported source dialects, there is a limitation in the SQL statement and data types that can be handled by the system. The grammar files of both MySQL and PostgreSQL dialects could be broadened to support the parsing of more dialect features. Transforming capabilities must be extended accordingly. Moreover, there is a need to maintain and keep up-to-date the transformation support we currently provide. Due to the changes and enhancements occurring constantly to the existing dialects, our implementation can become insufficient and out-dated soon.

The current implementation does not consider any performance optimization of the grammar files and subsequently of the generated parsers. In Chapter 7 we found that the performance of PostgreSQL transformer is higher compared to the MySQL transformer. However, while developing the PostgreSQL grammar file we focused on achieving the desired functionality, without considering any performance objective. In the future, the code architecture could be analyzed mathematically in advance and optimization techniques could be applied that will lead to a faster transformation functionality.

The SQL parser of the transformation service could be used further, to define the type of an SQL statement, retrieve the alternated first information structure (table), etc. The proxy bundle can use those functionalities beside the transformation and this may benefit the performance of CDASMix.

Regarding the PostgreSQL proxy, currently the *Simple Query Protocol* (see Figure 6.3) is implemented. Realizing the implementation of the *Extended Query Protocol* could increase the performance of CDASMix. Structurally similar queries, namely queries that can be analyzed into identical parse trees with different leaves' values, can be handled significantly faster. For this, several components of CDASMix, involved in message routing, marshaling and demarshaling, need to be modified and enabled to deal with prepared statements and portals.

Appendix A.

Source Code Segments

The following lists we display the code segment used for implementing OSGi declarative transformation service and two JUnit test cases for validating the desired functionality of the PostgreSQL transformer and for evaluating its transformation speed, respectively. Also, the generated MySQL and PostgreSQL parsers are displayed in the two last lists.

A.1. OSGi Declarative Service Implementation

```
1  @Component(name = "PostgreSQLTransformer", immediate = false)
2  @Service(value = SQLTransformer.class)
3  @Property(name = SQLTransformer.SOURCE_DIALECT_PROP, value = "PostgreSQL")
4  public class PostgreSQL implements SQLTransformer {
5
6      @Override
7      public String transform(String original, String target) throws NotImplementedException,
           UntransformableException, SQLParseException
8      {
9          if(target.equals(getSourceDialect())){
10             return original;
11         }
12         PostgreSQLParser 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 A.1: OSGi Declarative Service Implementation with Felix SCR annotations

A.2. Validation Test Case

```
1
2
3  public SelectTest extends TestCase{
4      ...
5  public void test() throws JSQLParserException {
6      String statement = "SELECT␣*␣FROM␣lineitem␣WHERE␣quantity␣=11␣LIMIT␣10␣OFFSET␣3;";
7      String transformed = "SELECT␣*␣FROM␣lineitem␣WHERE␣quantity␣=␣11␣LIMIT␣3,␣10;";
8  Select select = (Select) parser.parse(new StringReader(statement));
9  assertEquals(3, ((PlainSelect)select.getSelectBody()).getLimit().getOffset());
10 assertEqauls(statement, select.toString());
11 assertEquals(transformed, select.transform(Dialect.PostgreSQL));
12         ...
13     }
14     ...
15 }
```

Listing A.2: JUnit Test Case Example for Validation

A.3. Evaluation Test Case

```
1
2  public class SpeedTest {
3  ...
4      public static void main(String[] args) {
5    try {
6           str = "SELECT␣*␣FROM␣lineitem␣WHERE␣quantity␣=␣11␣LIMIT␣10␣OFFSET␣3;";
7            for (int i = 0; i < 100; i++) {
8            long t0 = System.currentTimeMillis();
9            for (int j = 0; j < 10000; j++) {
10             parser = new MySQLParser(new StringReader(str));
11                    stmt = parser.Statement();
12             transformed = stmt.transform(Dialect.MySQL);
13             parser = null;
14             stmt = null;
15             transformed = null;
16            }
17           long e0 = System.currentTimeMillis() - t0;
18           System.out.println("Overall␣time", e0);
19
20           }
21
22   } catch (Throwable e) {
23       e.printStackTrace();
24   }
25     }
26 }
```

Listing A.3: JUnit Test Case Example for Evaluation

A.4. Generated PostgreSQL Parser

```
final public DropTable DropTable() throws ParseException {

        DropTable dropTable = new DropTable();
        List tables;

        jj_consume_token(K_DROP);
        jj_consume_token(R_TABLE);
        if (jj_2_54(2)) {
            jj_consume_token(K_IF);
            jj_consume_token(K_EXISTS);
            dropTable.setIfExists(true);
        } else {
             ;
        }
        tables = TableList();
        dropTable.setTables(tables);
        switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
            case K_CASCADE:
            case K_RESTRICT:
            switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
                case K_CASCADE:
                    jj_consume_token(K_CASCADE);
                    dropTable.setCascade(true);
                    break;
                case K_RESTRICT:
                    jj_consume_token(K_RESTRICT);
                    dropTable.setRestrict(true);
                    break;
                default:
                    jj_la1[221] = jj_gen;
                    jj_consume_token(-1);
                    throw new ParseException();
            }
                break;
                default:
                jj_la1[222] = jj_gen;
                ;
        }
        {if (true)
      return dropTable;
    }
        throw new Error("Missing_return_statement_in_function");
 }
```

Listing A.4: Code Snippet of the Generated PostgreSQL Parser, Responsible for Parsing the `DROP table` Statements

A.5. Generated MySQL Parser

```
final public DropTable DropTable() throws ParseException {

        DropTable dropTable = new DropTable();
        Table table;
        List tablesList = new ArrayList();

        jj_consume_token(R_DROP);
        switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
            case K_TEMPORARY:
                jj_consume_token(K_TEMPORARY);
                dropTable.setTemporary(true);
                break;
            default:
                jj_la1[286] = jj_gen;
                ;
        }
        jj_consume_token(R_TABLE);
        switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
            case R_IF:
                jj_consume_token(R_IF);
                jj_consume_token(R_EXISTS);
                dropTable.setIf_exists(true);
                break;
            default:
            jj_la1[287] = jj_gen;
            ;
        }
        table = Table();
        tablesList.add(table);
        label_37:
        while (true) {
            switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
                case 150:
                   ;
                   break;
                default:
                   jj_la1[288] = jj_gen;
                   break label_37;
            }
            jj_consume_token(150);
            table = Table();
            tablesList.add(table);
        }
        switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
            case R_CASCADE:
            case R_RESTRICT:
            switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
                case R_RESTRICT:
                    jj_consume_token(R_RESTRICT);
                    dropTable.setRestrict(true);
                    break;
```

```
52                  case R_CASCADE:
53                      jj_consume_token(R_CASCADE);
54                      dropTable.setCascade(true);
55                      break;
56                  default:
57                      jj_la1[289] = jj_gen;
58                      jj_consume_token(-1);
59                      throw new ParseException();
60              }
61              break;
62              default:
63                  jj_la1[290] = jj_gen;
64                   ;
65          }
66          dropTable.setTablesList(tablesList);
67          {if (true)
68        return dropTable;
69      }
70          throw new Error("Missing return statement in function");
71  }
```

Listing A.5: Code Snippet of the Generated MySQL Parser, Responsible for Parsing the `DROP table` Statements

Appendix B.

Class Diagrams

In the following, a class diagram that depicts the relationships among the PostgreSQL proxy and the SQL transformation components is shown. Here, a more detailed overview of the class diagram of Figure 6.1 is provided. The methods each of the involved classes implements or overrides are shown.

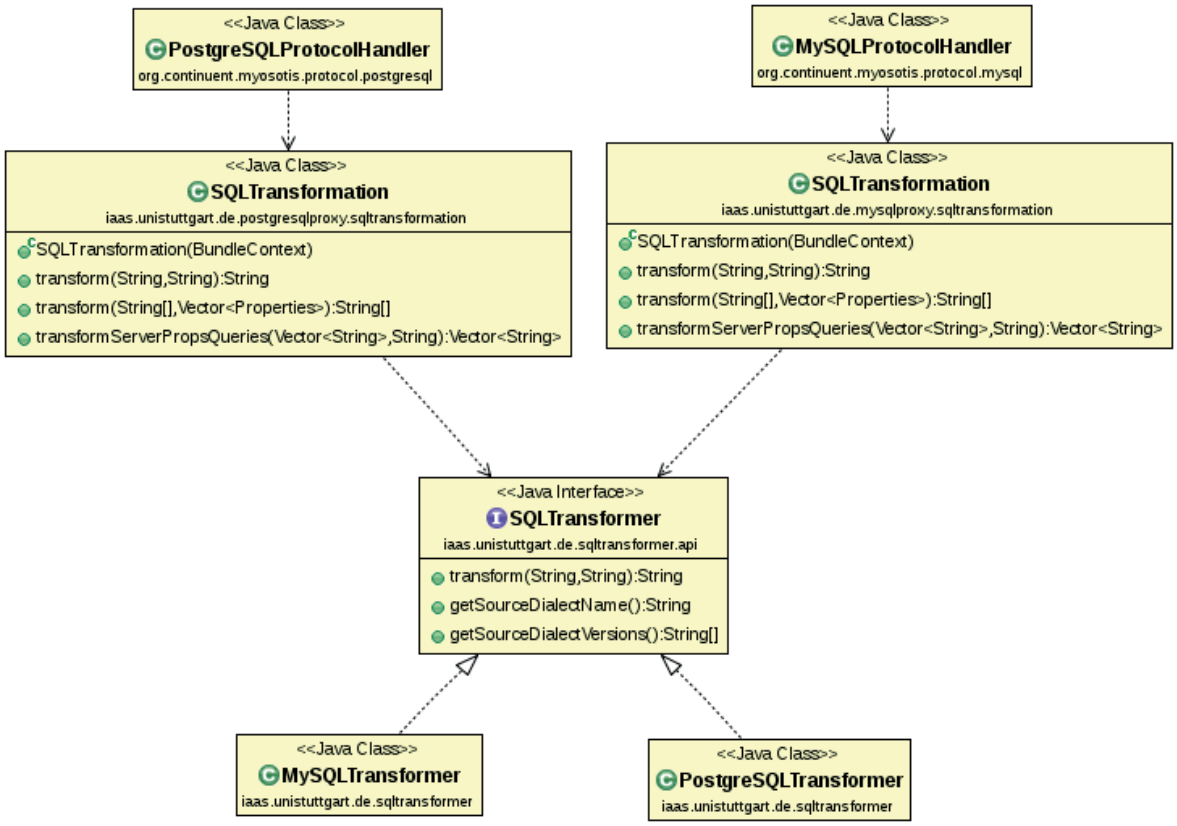

Figure B.1.: Class Diagram that Shows the Relationships Among the PostgreSQL Proxy and the SQL Transformation Components

The following class diagram depicts the relationships among the PostgreSQL proxy and classes of other components of CDASMix, such as *ServiceMix-Camel* and *CDASMixJDBC*.

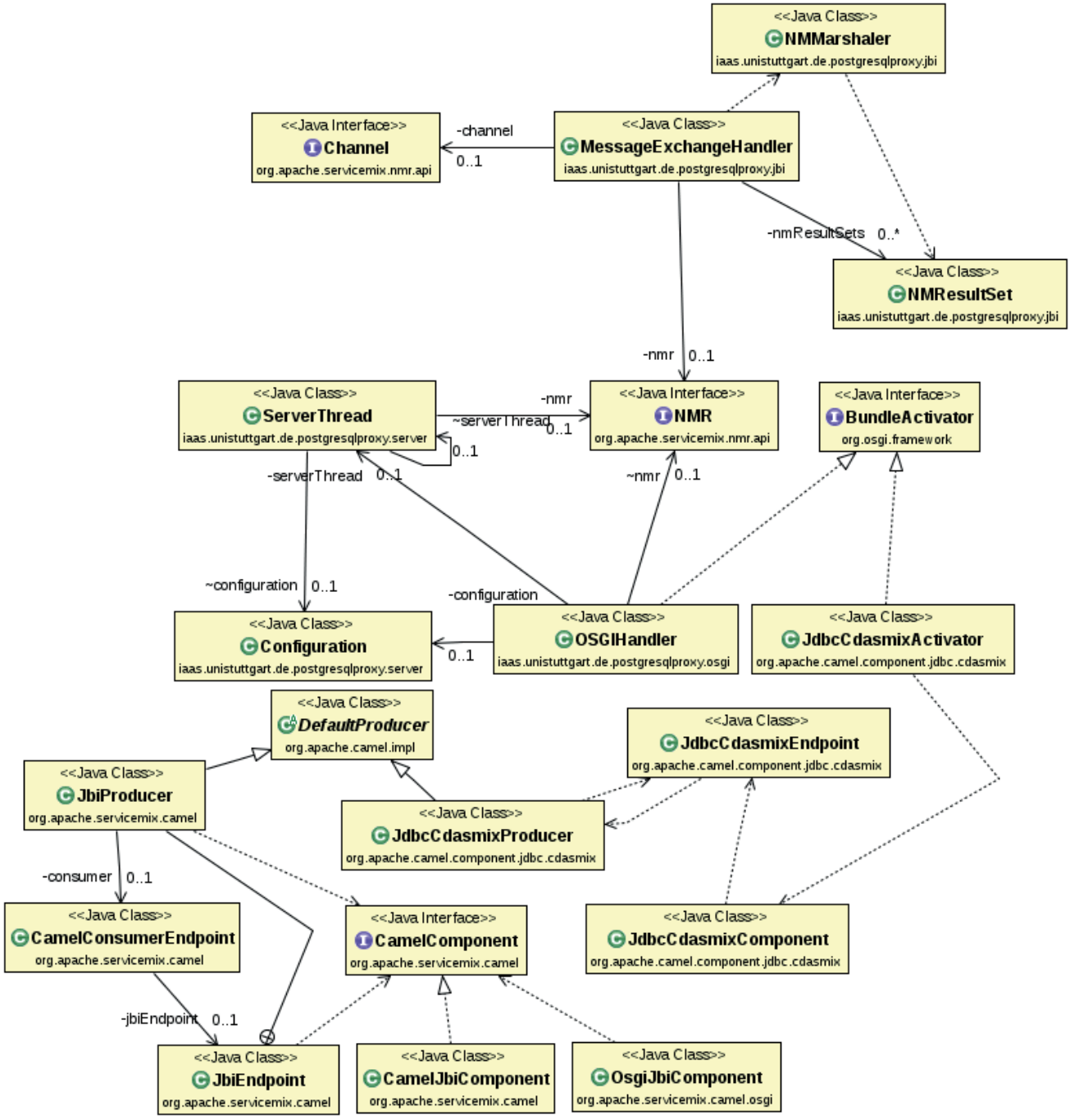

Figure B.2.: Class Diagram that Shows the Relationships Among the PostgreSQL Proxy and Other Components of CDASMix

Appendix C.

Visitor Pattern in JSqlParser

As we discussed in Section 3.2, we develop the PostgreSQL transformation based on the open source `JSqlParser`, which analyzes SQL statement and generates from it a hierarchy of Java classes in depth-first pre-order. The generated hierarchy can be navigated using the *visitor pattern* [Conb] (also in depth-first pre-order). This allows the addings of new methods to the existing hierarchy without modifying the hierarchy [Mar02]. This is actually the reason why the same class hierarchy can be easily re-used for implementing different types of parsers [MA06]. It was used in [Xia13] for generating the MySQLParser and it is used in this work for the PostgreSQLParser.

The visitor represents an operation, encapsulated as a Java method, to be performed on the elements and allows us to define new operations without changing the classes. In such a way we are able to plug-in the *transform* functionality to the existing class hierarchy that will convert each node to an SQL component of the target statement.

The visitor architecture organizes the class hierarchy of JSqlParser in 7 groups, as shown in Figure C.1. Figure C.2 focuses on the group of classes that implement the Statement Interface. These classes are the so called elements. Each element implements an *accept* method that takes the respective visitor as an argument. In the example of Figure C.2, the *accept* methods that each element overrides, take as argument the *StatementVisitor*. The implementation of the called *accept* method is chosen based on the dynamic type of the element and the static type of the visitor, while the implementation of the called *visit* method is chosen based on the dynamic type of the visitor and the static type of the element. Hence, the double dispatch is effectively implemented.

One should notice that our implementation uses the visitor pattern architecture to navigate the Java hierarchy and not to directly implement transform functionality. Each Java class of the elements is enhanced with a new method, the *transform* method, which handles the transformation of the SQL feature encapsulated by this class, into the implemented target dialects.

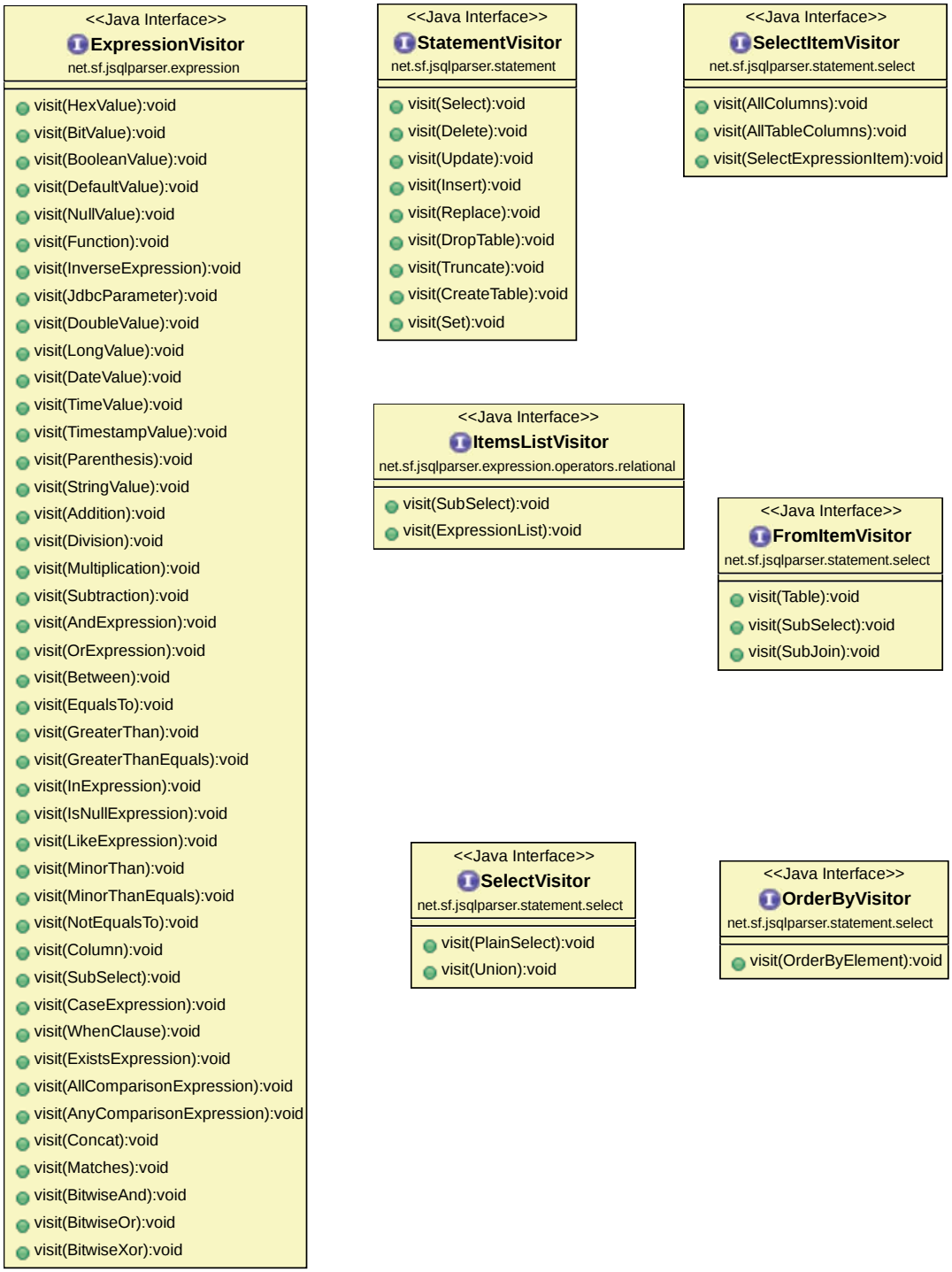

Figure C.1.: Visitor Interfaces of the JSqlParser's Class Architecture

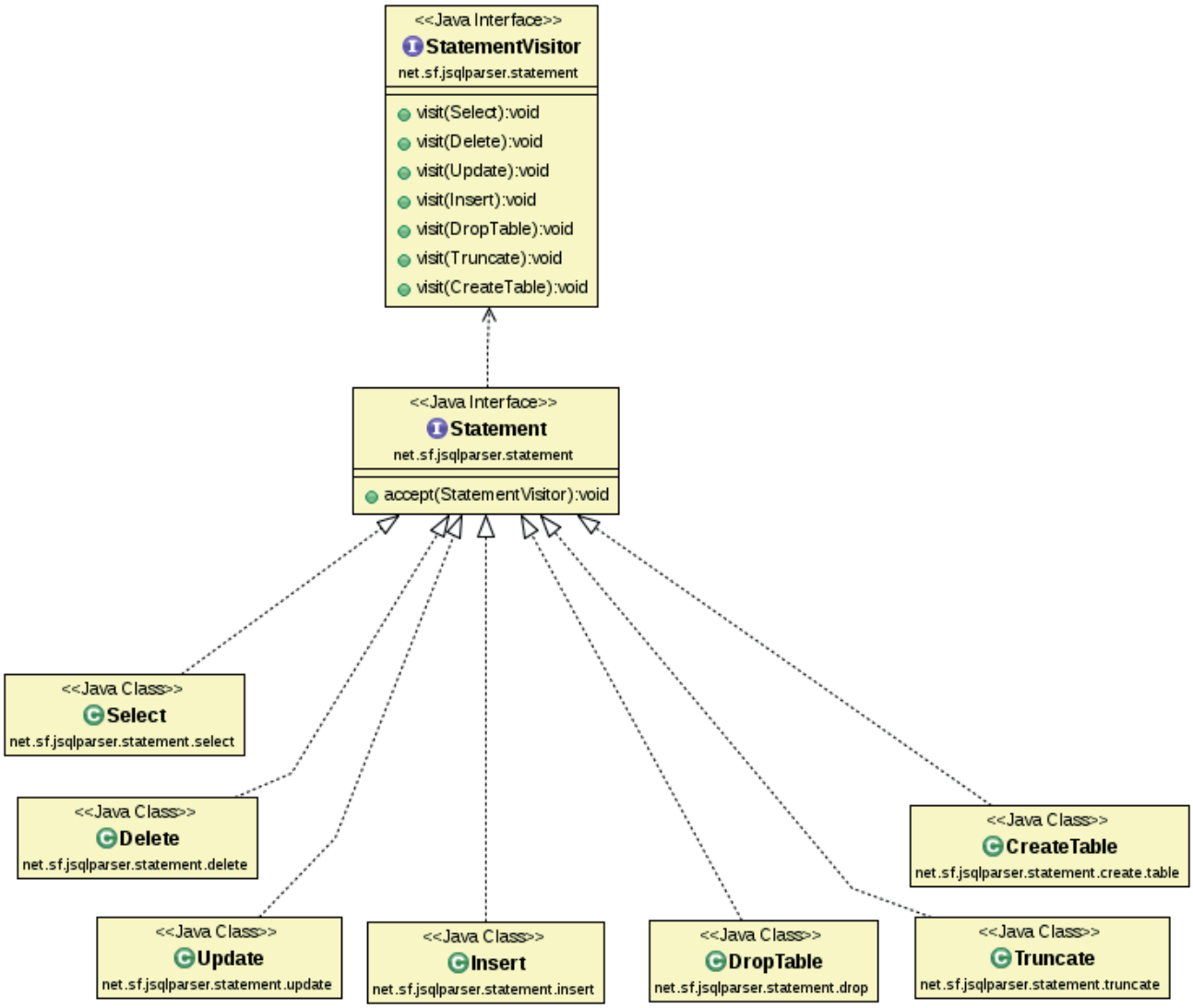

Figure C.2.: Visitor Architecture of the Group of Classes that Implement the Statement Interface

Bibliography

[ABLS13] V. Andrikopoulos, T. Binz, F. Leymann, and S. Strauch. How to Adapt Applications for the Cloud Environment. *Computing*, 95:493–535, 2013.

[AFG+05] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. *Proceedings of the IEEE*, 93(2):449–466, 2005.

[Amo14] G. Amorim. The Importance of SOA to Cloud Computing. *Service Technology Magazine*, 2014.

[app] Application Architecture Guide - Chapter 9 - Layers and Tiers. `http://www.guidanceshare.com/wiki/Application_Architecture_Guide_-_Chapter_9_-_Layers_and_Tiers`.

[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/`.

[Bar11] I. Barukcic. *Causality II. A Theory Of Energy, Time And Space*. lulu.com, 2011.

[BGK+11] M. Behrendt, B. Glasner, P. Kopp, R. Dieckmann, G. Breiter, S. Pappe, H. Kreger, and A. Arsanjani. Introduction and Architecture Overview, IBM Cloud Computing Reference Architecture 2.0. *Draft Version V*, 1(0), 2011.

[Boy04] C. Boyer. The 360 Revolution. *IBM Corp*, 2004.

[CB74] D. D. Chamberlin and R. F. Boyce. SEQUEL: A Structured English Query Language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74, pages 249–264. ACM, 1974.

[CK13] S. Cranton and J. Korab. *Apache Camel Developer's Cookbook (Solve Common Integration Tasks With Over 100 Easily Accessible Apache Camel Recipes)*. Packt Publishing, 2013.

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

[com] MySQL vs. PostgreSQL. `https://www.wikivs.com/wiki/MySQL_vs_PostgreSQL`.

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

[Conb] Continuent, Inc. Sourceforge. http://jsqlparser.sourceforge.net.

[Eis03] P. Eisentraut. *PostgreSQL – Das Offizielle Handbuch*. Verlag Moderne Industrie, 2003.

[GBE07] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. *SIGPLAN Not.*, 42(10):57–76, 2007.

[Goo09] J. Goodson. *The Data Access Handbook : Achieving Optimal Database Application Performance and Scalability*. Prentice Hall, 2009.

[GS13] S. Gómez 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.

[Jak] Jakub Korab. Effective System Integrations with Apache Camel. https://skillsmatter.com/skillscasts/5074-effective-system-integrations-with-apache-camel.

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

[KB] D. K. Barry. Web Services and Cloud Computing. http://www.service-architecture.com/articles/cloud-computing/web_services_and_cloud_computing.html.

[keya] MySQL Documentation: Keywords and Reserved Words. https://dev.mysql.com/doc/refman/5.5/en/keywords.html.

[keyb] PostgreSQL 9.4.5 Documentation: Appendix C. SQL Key Words. http://www.postgresql.org/docs/7.3/static/sql-keywords-appendix.html.

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

[Lin93] B. Lindgren. *Statistical Theory, Fourth Edition (Chapman & Hall/CRC Texts in Statistical Science)*. Chapman and Hall/CRC, 1993.

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

[Lou10] P. Louridas. Up in the Air: Moving Your Applications to the Cloud. *IEEE Software*, 27(4):6–11, 2010.

[MA06] B. Meyer and K. Arnout. Componentization: The Visitor Example. *Computer*, 39(7):23–30, 2006.

[Mar02] R. C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Pearson, 2002.

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

[Mil12] N. Milosavljević. Algorithms and Data Structures, Course Notes, University of Stuttgart, 2012.

[Mir] Mira Mezini. Programming Abstractions for Applications in Cloud Environment (PACE). `http://pace-erc.eu/assets/PACE-Synopsis.pdf`.

[Moh11] T. S. Mohan. *Migrating into a Cloud*, pages 43–56. John Wiley and Sons, Inc., 2011.

[Mol12] L. Molková. *Theory and Practice of Relational Algebra: Transforming Relational Algebra to SQL*. LAP LAMBERT Academic Publishing, 2012.

[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.

[Mye09] J. Myerson. Cloud Computing Versus Grid Computing. *Service Types, Similarities and Differences, and Things to Consider, IBM*, 3, 2009.

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

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

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

[Osi10] J. Osis. *Model-Driven Domain Analysis and Software Development: Architectures and Functions (Premier Reference Source)*. IGI Global, 2010.

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

[posa] PostgreSQL 9.4.0 Documentation. `http://www.postgresql.org/docs/9.4/interactive/index.html`.

[posb] PostgreSQL Documentation, Chapter 43. `http://www.postgresql.org/docs/9.4/static/protocol.html`.

[Qus05] Qusay H. Mahmoud. Service-Oriented Architecture (SOA) and Web Services: The Road to Enterprise Application Integration (EAI), 2005. `http://www.oracle.com/technetwork/articles/javase/soa-142870.html`.

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

[Rei11] A. J. D. Reis. *Compiler Construction Using Java, JavaCC, and Yacc*. Wiley-IEEE Computer Society Pr, 1 edition, 2011.

[SAGS+12] S. Strauch, V. Andrikopoulos, S. Gómez Sáez, F. Leymann, and D. Muhler. Enabling Tenant-Aware Administration and Management for JBI Environments. In *Proceedings of the 5th IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2012*, pages 206–213. IEEE Computer Society, 2012.

[SAK+14] S. Strauch, V. Andrikopoulos, D. Karastoynova, F. Leymann, N. Nachev, and A. Staebler. Migrating Enterprise Applications to the Cloud: Methodology and Evaluation. *International Journal of Big Data Intelligence*, 1(3):127–140, 2014.

[SAKVH15] S. Strauch, V. Andrikopoulos, D. Karastoyanova, and K. Vukojevic-Haupt. *Migrating eScience Applications to the Cloud: Methodology and Evaluation*, book chapter 5, pages 89–114. Cloud Computing with E-science Applications. CRC Press/-Taylor & Francis, 2015.

[SALM12] S. Strauch, V. Andrikopoulos, F. Leymann, and D. Muhler. ESB^{MT}: Enabling Multi-Tenancy in Enterprise Service Buses. In *Proceedings of the 4th IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2012*, pages 456–463. IEEE Computer Society, 2012.

[Sch14] C. Schmid. Development of a Java Library and Extension of a Data Access Layer for Data Access to Non-Relational Databases. Diploma Thesis No. 3679, Institute of Architecture of Application Systems, University of Stuttgart, 2014.

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

[SW11] R. Sedgewick and K. Wayne. *Algorithms (4th Edition)*. Addison-Wesley Professional, 2011.

[SZT12] B. Schwartz, P. Zaitsev, and V. Tkachenko. *High Performance MySQL: Optimization, Backups, and Replication*. O'Reilly Media, Inc., 2012.

[Tec] TechTarget, SearchCloudComputing. Cloud Computing Technology Trends-in-2015. `http://searchcloudcomputing.techtarget.com/feature/Cloud-computing-technology-trends-in-2015`.

[The] The Apache Software Foundation. Apache Camel. `http://camel.apache.org/exchange-pattern.html`.

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

[VBB11] W. Voorsluys, J. Broberg, and R. Buyya. *Introduction to Cloud Computing*. John Wiley and Sons, Inc., 2011.

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR, 2005.

[Xia13] S. Xia. Extending an Open Source Enterprise Service Bus for SQL Statement Transformation to Enable Cloud Data Access. Diploma Thesis No. 3506, Institute of Architecture of Application Systems, University of Stuttgart, 2013.

All links were last followed on October 22, 2015.

Acknowledgement

I am heartily thankful to my supervisor Steve Strauch from the University of Stuttgart for trusting me and giving me the opportunity to enter the exciting field of system integration and Cloud computing. Without his encouragement, guidance, persistent support, and patience, the completeness of this thesis would not have been possible. I am grateful to my parents as well, who have always been the source of inspiration and the greatest supporters for me. I want to thank my colleague and boyfriend Oliver Feldmann, for the encouragement, support, and countless advices. Last but not least, I thank my sister and young programmer, Eva Ramaj, as well as my fellow and friend, Omar Elazhary. The long and fruitful discussions with them were extremely helpful.

Alketa Ramaj

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, 23 October 2015 _____

(Alketa Ramaj)