UNIVERSIDADE D
COIMBRA

Henrique Tavares Silva

Design and implementation of a relational data model supported in Postgres

Sakai Learning Management System

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Professor Bruno Cabral and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July of 2023

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

Henrique Tavares Silva

Design and implementation of a relational data model supported in Postgres

Sakai Learning Management System

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Prof. Bruno Cabral and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July 2023

Agradecimentos (Acknowledgements)

Embora esta tese de mestrado esteja escrita em inglês, tenho que usar o vocabulário e a gramática portuguesa para agradecer a todos os que me ajudaram, direta ou indiretamente, a concluir esta jornada que começou em 2017, com a minha entrada na Universidade de Coimbra.

Quero agradecer à Apereo Foundation pela oportunidade de participar no projeto Sakai através deste estágio, e em especial ao doutor Charles Severance pelo acompanhamento e conhecimento transmitido. Também quero agradecer ao Professor Bruno Cabral, pela disponibilidade e orientação desde o início do estágio.

Quero também agradecer à minha família pelo seu apoio incondicional. Que apesar de não ter sido um trajeto como desejariam, foram fundamentais para que eu terminasse este ciclo académico.

Também não posso deixar de agradecer a todos os amigos que fiz e mantive durante estes 6 anos, independentemente de termos partilhado uma sala de aula, um balneário ou uma tenda. Fizeram-se presentes nos meus desalentos e conquistas, e por isso vos fico grato.

Abstract

This thesis aims to study database portability in the context of the large, enterprise legacy code that is Sakai LMS, which is a free, community-source, educational software platform designed to support teaching and research, and has been under continuous development and production for the past 20 years.

The application is committed to database portability and currently supports Oracle and MySQL and wishes to support PostgreSQL going forward. The application mixes hand-constructed SQL, templated SQL, and data persistence by a Java ORM solution, JPA. The goal is not to rewrite the application to use a new way to generate SQL but to look at the effort required to convert all SQL approaches from one dialect to another.

To inform and help accomplish this engineering task, this thesis will have a deeper examination of how the application executes all database-related operations and explore how the SQL dialects diverge among the various databases. This paper will also look at strategies to add a new database dialect or convert SQL from one dialect to another, exploring the jOOQ library and then testing its effectiveness in this internship's problem.

This thesis work contributed to the progress in the challenge of inserting the PostgreSQL dialect into SakaiLMS:

- Sakai LMS can now start up with a PostgreSQL database, using a MySQL SQL service that gets translated to Postgres.
- The proposal of an architecture that can build the base for a translations server that enables Sakai LMS to be fully compatible with Postgres and, thanks to the way jOOQ's SQL translator works, other relational databases, without modifying the SQL service.

In addition to that, this work also allowed us to conclude that jOOQ's SQL translator works for most translations, but not always. So it's necessary to modify any jOOQ SQL translator layer to cope with the errors it throws. As we could conclude, manual edition and approval of unsuccessful translations are enough.

Another conclusion taken was that the SQL service in Sakai isn't the only entity responsible for executing SQL statements in Sakai. This turned out to be a limitation in the work's results because the proposed architecture wasn't expecting that behavior from Sakai.

Keywords

Sakai LMS, PostgreSQL, MySQL, Oracle, jOOQ, SQL translation

Contents

Acronyms

API Application Programming Interface.

CRUD CREATE, READ, UPDATE, DELETE.

DBMS Database Management System.

DDL Data Definition Language.

DML Data Manipulation Language.

DSL Domain Specific Language.

JDBC Java Database Connectivity.

jOOQ Java Object-Oriented Querying.

JPA Java™ Persistence API.

LMS Learning Management Systems.

ORM Object-relational mapping.

PTC Postgres Translations Component.

RDBMS Relational Database Management System.

SQL Structured Query Language.

TH Translations Hashmap.

List of Figures

List of Tables

Chapter 1

Introduction

In the realm of education, technology has played a pivotal role in transforming the way knowledge is disseminated, managed, and accessed. Over the years, Learning Management Systems (LMS) have emerged as indispensable tools for educational institutions, facilitating the delivery of online courses, collaborative learning experiences, and efficient content management. Simultaneously, relational database management systems have revolutionized the storage and retrieval of vast amounts of structured data, becoming the backbone of numerous applications across various industries.

LMS have come a long way from their early iterations, initially developed as content repositories with limited interactivity. As educators recognized the need for more dynamic and personalized learning environments, LMS evolved to incorporate features like discussion boards, assessment tools, and multimedia integration. This metamorphosis was paralleled by the advancement of RDBMS, which provided the foundation for efficient data management, data integrity, and seamless integration of diverse content types within the LMS ecosystem.

1.1 Problem Statement

Nowadays, most software applications use ORM solutions to facilitate the storage of data, not only because it allows independence from any database type, but also facilitates the manipulation of data.

Sakai LMS has remained modern and relevant in the current scenario of teaching support platforms, always with a view to adapting to the infrastructures of institutions and universities that adopt it. As a result of this factor, the databases currently supported are MySQL and Oracle. This project seeks to expand Sakai's spectrum of databases to work with PostgreSQL, making the necessary adaptations in terms of architecture to do so.

However, Sakai, as it was founded at the beginning of the century, still presents some typical traits of software programs written at that time. One of which is the large quantity of manual and templated SQL statements written. These state-

ments define the schema of the database as well as manipulate the data in it.

This project doesn't consist of only changing the database configuration options to start working with a Postgres database, as it would happen with an application that only uses ORM to communicate with a database.

This project intends to make use of that large amount of data definition and manipulation statements and create an architecture capable of translating them to make Sakai compatible with PostgreSQL. Doing this, instead of rewriting all of those statements to PostgreSQL, allowed to come up with a solution that could be advantageous in the long run and facilitate the work for the insertion of other databases in the future.

This thesis will investigate the potential of manual query conversion, automatic query conversion, and a hybrid of manual and automatic query conversion for the database-specific SQL queries in Sakai.

1.2 Main objectives and approach

The main objective and contribution of this work is the addition of PostgreSQL to the Sakai LMS as well as understanding and reporting on the challenges and approaches to migrating large legacy software with manually constructed SQL to a new database.

This could be achieved by using a query conversion tool to convert the existing hand-constructed SQL queries in the application's Relational Database Management System (RDBMS) (Oracle and MySQL) and then use the converted queries in the new RDBMS (PostgreSQL).

The approach involved building a layer that could automate the query conversion and reduce the changes in the Sakai code that actually constructs the queries. We used the Java Object-Oriented Querying (jOOQ) framework, as a library to implement our initial approach to automated translation of hand-constructed and templated SQL from one vendor variant to another.

By studying this approach, this work would contribute to Sakai LMS a layer that could not only enable developers to use the application with a translations server but also allow, in the future, to use the application with some other RDBMS. This approach helped create the base for a translations server that can, eventually, allow Sakai to perform normally with any kind of relational database, without manually translating the core of its SQL service dialects.

1.3 Results of the thesis

This thesis work contributed to the progress in the challenge of inserting the PostgreSQL dialect into SakaiLMS:

- Sakai LMS can now start up with a PostgreSQL database, using a MySQL SQL service that gets translated to Postgres.
- The proposal of an architecture that can build the base for a translations server that enables Sakai LMS to be fully compatible with Postgres and, thanks to the way jOOQ's SQL translator works, other relational databases, without modifying the SQL service.

In addition to that, this work also allowed us to conclude that jOOQ's SQL translator works for most translations, but not always. So it's necessary to modify any jOOQ SQL translator layer to cope with the errors it throws. As we could conclude, manual edition and approval of unsuccessful translations are enough.

Another conclusion taken was that the SQL service in Sakai isn't the only entity responsible for executing SQL statements in Sakai. This turned out to be a limitation in the work's results because the proposed architecture wasn't expecting that behavior from Sakai. This doesn't reflect a faulty construction of the project architecture but the gathering of inaccurate information on the Sakai LMS's data control.

1.4 Outline

Chapter 2 provides important background knowledge for the understanding of the work done in the implementation phase, explores other tools, that are used in not-so-different situations, and a closer insight into what jOOQ is and why it was chosen for the solution's architecture.

Chapter 3 discusses the approach to the problem and states the architectural drivers that will measure the success of the whole intervention.

Chapter 4 sums up the decisions, experiments, and changes in the Sakai application that led to the obtained results.

Chapter 5 points to the methodology used to guide the internship and provides a comparison between the initial plan for the second semester and the real course of events at that time.

Chapter 6 summarizes the conclusions drawn from the thesis.

Chapter 2

Background concepts and State of the Art

This chapter provides important background knowledge for the understanding of the work done in the remainder of the thesis. Firstly, explains what open-source software is. Secondly, explores the Sakai's architecturally important parts for this thesis. Then, compares the differences between Postgres and the two database dialects present in Sakai, MySQL and Oracle. After that, enumerates the currently most used schema conversion tools. And the last sections focus on providing grounding on why the chosen tool for this project, jOOQ, suits the problem in hand.

2.1 Open-source

Since the products of this thesis will insert an open-source community project, its fundamental to understand what is expected from open-source technologies and communities.

By design, open source software licenses promote collaboration and sharing because they permit other people to make modifications to source code and incorporate those changes into their own projects. They encourage computer programmers to access, view, and modify open source software whenever they like, as long as they let others do the same when they share their work. [1]

Open source software refers to computer software whose source code is made freely available to the public. In other words, users have the freedom to view, modify, and distribute the source code according to the terms of the software's open-source license. This open nature promotes collaboration, transparency, and community-driven development.

Some of the major objectives of using open-source software include: [2]

- **Cost:** Open source software is generally cheap to use (if not free), which may significantly benefit organizations and individuals who are working

with limited budgets.

- **Community:** Open source software is developed and maintained by a community of developers, which means that there is often a large and active user base that can provide support and assistance.
- **Customizability:** Because open source software is typically published under a license that allows users to modify and distribute the source code, it can be easily customized to meet the specific needs of an organization or project.

It is important to retain that any work done in an open-source project should care for facilitating future customizability and low costs.

2.2 Apereo Foundation and Sakai LMS

The Apereo Foundation [3] is a non-profit organization that supports the development and use of open-source software for education. It was founded in 2005 as the Sakai Foundation and was later renamed Apereo Foundation in 2012. The Apereo Foundation is focused on developing and supporting open-source software and tools that are used in education and research.

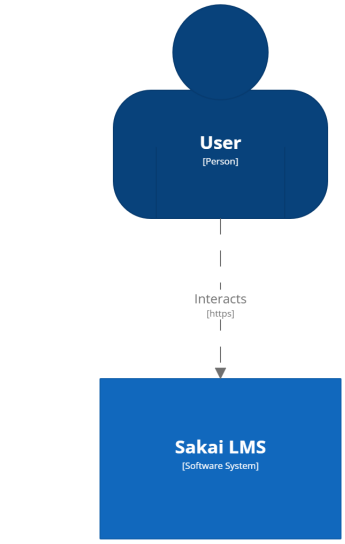
The Apereo Foundation is supported by a community of educational institutions, organizations, and individuals who contribute to the development and use of its open-source software. The foundation is committed to promoting the use of open-source software in education and research and to supporting the development of tools and resources that can improve the learning experience.

Sakai LMS [4] is an open-source learning management system (LMS) that is used by educational institutions and organizations around the world. It is a comprehensive platform that allows users to create and manage courses, assign and grade assignments, conduct online discussions, and provide resources and materials for students. Sakai is designed to be a flexible and customizable platform, and it can be tailored to meet the needs of different educational institutions and organizations. It supports a wide range of features and tools, including course and project management, collaboration, assessment, and communication.

Overall, Sakai is a powerful and feature-rich learning management system and a flexible and customizable platform that can be tailored to meet the needs of different users and environments.

2.2.1 Sakai LMS architecture

This project has the objective to add modifications to the application's communication with the database, therefore the architectural most important parts are the ones that involve the storage and retrieval of data between Sakai and the database.



[System Context] Sakai LMS

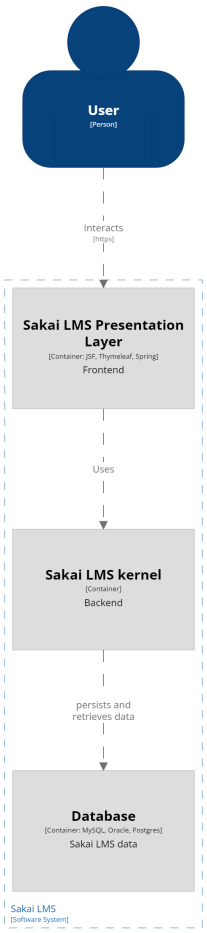
Figure 2.1: System Context diagram for Sakai LMS

Figures 2.1 and 2.2 represent, respectively, Sakai's Context and Container diagrams. These figures attempt to represent this platform's most architecturally relevant parts for the intervention.

The Sakai presentation layer fetches information from the Sakai LMS kernel and allows users to use and manipulate it.

The Sakai kernel provides core functionality for the Sakai framework. Contains the SQL Service and Object Relational Mapper that stores/persists and retrieves data in the Sakai database. Its Component Diagram is presented in figure 2.3.

The Sakai database contains all data of the application. The Sakai database can be MySQL or Oracle type.



[Container] Sakai LMS

Figure 2.2: Current Container diagram for Sakai LMS

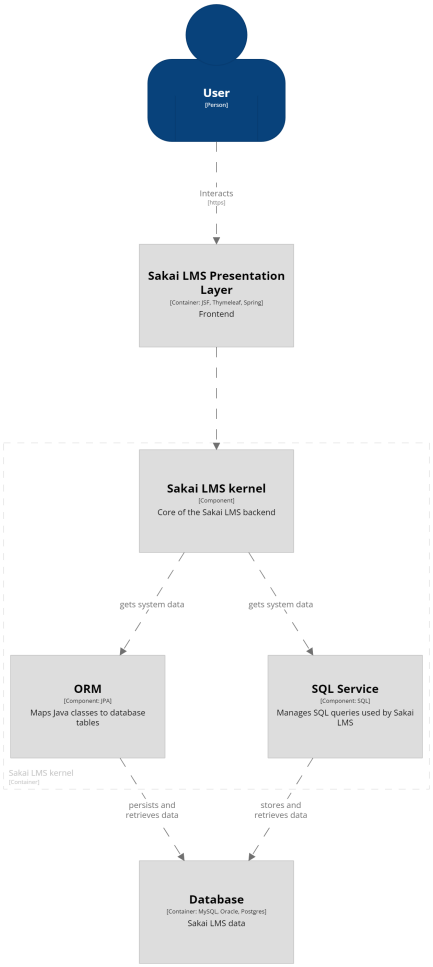


Figure 2.3: Current Component diagram for Sakai LMS kernel

2.2.2 Using the Sakai LMS

To start the Sakai LMS, the user must define the configurations in the sakai.properties file to choose from. Here is an example of how the sakai.properties file should look:

```
auto.ddl=true
hibernate.show_sql=false

## MySQL settings
    vendor@org.sakaiproject.db.api.SqlService=mysql
    driverClassName@javax.sql.BaseDataSource=org.mariadb.jdbc.Driver
    hibernate.dialect=org.hibernate.dialect.MariaDBDialect
    url@javax.sql.BaseDataSource=jdbc:mariadb://127.0.0.1:3306/sakai
    username@javax.sql.BaseDataSource=root
    password@javax.sql.BaseDataSource=password
    validationQuery@javax.sql.BaseDataSource=select 1 from DUAL


## Oracle settings
     #vendor@org.sakaiproject.db.api.SqlService=oracle
     #driverClassName@javax.sql.BaseDataSource=oracle.jdbc.driver.OracleDriver
     #hibernate.dialect=org.hibernate.dialect.Oracle10gDialect
     #url@javax.sql.BaseDataSource=jdbc:oracle:thin:@localhost:1521/xepdb1
     #username@javax.sql.BaseDataSource=system
     #password@javax.sql.BaseDataSource=password
     #validationQuery@javax.sql.BaseDataSource=select 1 from DUAL

defaultTransactionIsolationString@javax.sql.BaseDataSource=
TRANSACTION_READ_COMMITTED

bodyPath@org.sakaiproject.content.api.ContentHostingService=
${sakai.home}data

bodyPathDeleted@org.sakaiproject.content.api.ContentHostingService=
${sakai.home}deleted/bodyContentDeleted/
```

In this example, the user intends to use the MySQL database named "sakai" with the MariaDB Java Database Connectivity (JDBC) driver and Hibernate dialect. If they wanted to use the Oracle database, the definitions for the MySQL database would have to be commented (by adding a at the beginning of each line) and the Oracle ones uncommented.

2.3 SQL diversification

The Structured Query Language (SQL) standard has been revised multiple times since the 1980s, although the core features of the standard SQL have been stable since 1992. However, new variations kept appearing as database implementers working at different vendors needed to solve new problems or circumvent existing problems that were not addressed in the standard. This explains why a multiple of SQL dialects made their apparition and still co-exist today. [5]

So, although there is only one standard of the SQL language, there is a variety of SQL dialects created for the Database Management Systems (DBMSs). These dialects extend the standard SQL, in order to add features and adapt the syntax to their DBMS.

2.3.1 PostgreSQL vs MySQL

This section describes the main differences between the PostgreSQL and MySQL databases regarding their data types, Data Definition Language (DDL) and Data Manipulation Language (DML) statements, and constraints.

Data types

Both MySQL and PostgreSQL support many data types, ranging from traditional ones (e.g., Integer, Date, Timestamp) to complex ones (e.g., JSON, XML, TEXT). However, there is a difference between these two database products when it comes to the capability of catering to complex, real-time data search requirements. Let's take a look at both. PostgreSQL not only supports traditional SQL data types (e.g., Numeric, Strings, Date, Decimal, etc.) but also supports unstructured data types (e.g., JSON, XML, and HSTORE) as well as network data types, bit strings, etc. What makes PostgreSQL stand out is its support for a wider range of data types, such as ARRAYs, NETWORK types, and Geometric data types (including advanced spatial data functions) to store and process spatial data. Supported data types can be found here. The support for spatial data types and functions comes from an external module called PostGIS, which is an open-source extension. MySQL supports various data types that help a variety of applications store and process data in different formats, including the following: traditional data types to store Integers, Characters or Strings, Date with Timestamps and Time Zones, Boolean, Float, Decimal, Large Text, and BLOB to store binary data (like images). There is no support for geometric data types in MySQL. [6]

Here are available all of the data types for MySQL and Postgres

DDL and DML

MySQL is not a fully SQL-compliant database and does not support all SQL features, making it a tough choice for developers and not a great choice for data warehousing applications, as there will be a need here for advanced and complex SQLs. MySQL doesn't yet support "LIMIT & IN/ALL/ANY/SOME subquery." Also, MySQL does not support standard SQL clauses such as FULL OUTER JOINS, INTERSECT, and EXCEPT, which are commonly used. Index types, including Partial Indexes, Bitmap Indexes, and Expression Indexes, are also not supported, and these are important to speed up query performances. PostgreSQL, on the other hand, is a fully SQL-compliant database and supports all SQL standard features. Applications of pretty much any nature from any domain can use PostgreSQL as their database, which makes it a popular choice for OLTP, OLAP, and DWH environments. PostgreSQL is the best choice for developers who have to write complex SQLs. [6]

Constraints

MySQL contains the constraint of types PRIMARY KEY, UNIQUE Index, FOREIGN KEY, ENUM and SET data types [7] and CHECK [8]. Although NOT NULL is not documented as a constraint in the MySQL 8.0 Reference Manual, MySQL enables tagging NOT NULL to a column for it not to accept NULL values.

Postgres doesn't have the ENUM or SET data types, but these data types have the same effect as a string or numeric column with a check constraint. Postgres can also apply, beyond the previously mentioned, the EXCLUDE constraint.[9]

2.3.2 PostgreSQL vs Oracle

This section describes the main differences between the PostgreSQL and Oracle databases regarding their data types, DDL and DML statements, and constraints.

Data types

Data type names between these two dialects often need translation. For example, in Oracle string values are commonly declared as being of type VARCHAR2, which is a non-SQL-standard type. In PostgreSQL, use type VARCHAR or TEXT instead. Similarly, replace type NUMBER with NUMERIC, or use some other numeric data type if there's a more appropriate one. [10]

Here are available all of the data types for Oracle and Postgres

DDL and DML

Data definition language (DLL) statements define, structurally change, and drop schema objects. In Oracle, DDL statements enable to:

- Create, alter, and drop schema objects and other database structures, including the database itself and database users. Most DDL statements start with the keywords CREATE, ALTER, or DROP.
- Delete all the data in schema objects without removing the structure of these objects (TRUNCATE).
- Grant and revoke privileges and roles (GRANT, REVOKE).
- Turn auditing options on and off (AUDIT, NOAUDIT).
- Add a comment to the data dictionary (COMMENT). [11]

Postgres can produce similar DDL statements, except for the auditing options. [12] The only way to enable Postgres with auditing operations could be by developing some auditing trigger. [13]

In Oracle, DML statements are the most frequently used SQL statements and enable you to:

- Retrieve or fetch data from one or more tables or views (SELECT).
- Add new rows of data into a table or view (INSERT) by specifying a list of column values or using a subquery to select and manipulate existing data.
- Change column values in existing rows of a table or view (UPDATE).
- Update or insert rows conditionally into a table or view (MERGE).
- Remove rows from tables or views (DELETE).
- View the execution plan for a SQL statement (EXPLAIN PLAN).
- Lock a table or view, temporarily limiting access by other users (LOCK TABLE). [14]

Postgres can produce the same DML statements. [15] [16] [17]

Constraints

Oracle Database enables you to apply constraints both at the table and column level.

- **NOT NULL** Allows or disallows inserts or updates of rows containing a null in a specified column.
- **Unique key** Prohibits multiple rows from having the same value in the same column or combination of columns but allows some values to be null.
- **Primary key** Combines a NOT NULL constraint and a unique constraint. It prohibits multiple rows from having the same value in the same column or combination of columns and prohibits values from being null.

- **Foreign key** Designates a column as the foreign key and establishes a relationship between the foreign key and a primary or unique key, called the referenced key.

- **Check** Requires a database value to obey a specified condition. [18]

Postgres can apply, beyond the previously mentioned, the EXCLUDE constraint.[9]

2.4 Schema conversion tools

Schema conversion tools play a crucial role in enabling smooth database migration processes by efficiently converting and adapting database schemas to fit the requirements of the target database system. In the next subsections some of the most used schema and currently used data conversion tools are described and compared.

2.4.1 Data Loader

Data Loader [19] is a tool developed by Interface Computers, that can in a simple graphical user interface, convert a database instance into another type of database. This tool allows customizing the translations, like defining the type and name of the new columns and other customizations and choosing to maintain constraints, indexes, and default values. The trial version doesn't load more than 50 rows per table, so for schema conversion, it's still suitable.

2.4.2 MySQL migration toolkit

The MySQL migration toolkit [20] was created by MySQL AB and translates relational database systems to MySQL. The Migration Toolkit may also support migrating data to other databases that are compatible with the MySQL protocol and syntax.

2.4.3 Ora2pg

Ora2pg [21] was developed by Gilles Darold. It connects an Oracle database, scans it automatically and extracts its structure or data, then generates SQL scripts that you can load into your PostgreSQL database. It can also apply the same process to a MySQL database. It doesn't provide the most user-friendly interaction, because the desired modifications involve manually changing a configurations file (ora2pg.conf) and running the program in a command prompt.

2.4.4 EDB Postgres Migration Portal

The EDB Postgres Migration Portal [22] was produced by EnterpriseDB Corporation. The Migration Portal consists in a web-based interface that generates data definition language (DDL) statements that are compatible with EDB Postgres Advanced Server, and, consequently, Postgres.

2.4.5 Ispirer SQLWays Toolkit

Ispirer Systems created the Ispirer SQLWays Toolkit [23]. SQLWays Toolkit is an easy-to-use cross-database migration tool with a basic graphical user interface. It allows migrating an entire database schema, including SQL objects, tables, and data from source to target databases. This tool isn't open source and doesn't support MySQL conversion to Postgres.

2.5 jOOQ

Java Object-Oriented Querying (jOOQ) is an open-source library for accessing and manipulating data stored in a relational database. It is written in Java and is designed to provide a simple and intuitive way to write SQL queries in a Java application.

jOOQ supports a wide range of database management systems, including MySQL, Oracle, and PostgreSQL. It provides a variety of features and tools, including support for stored procedures, functions, and triggers, as well as support for advanced SQL features such as window functions and common table expressions.

2.5.1 Code generation

Source code generation is one of jOOQ's main assets to increase developer productivity. jOOQ's code generator takes a database schema and reverse-engineers it into a set of Java classes modeling tables, records, sequences, POJOs, DAOs, stored procedures and user-defined types. jOOQ generates Java code from a database schema, which allows developers to work with a high-level, type-safe API that represents the structure of the database. This can make it easier to write and maintain complex SQL queries, as the developer does not need to be concerned with the details of the underlying database schema. [24]

2.5.2 DDL generation from objects

When using jOOQ's code generator, a whole set of metadata is generated with the generated artifacts, such as schemas, tables, columns, data types, constraints, default values, etc. This metadata can be used to generate DDL CREATE statements in any SQL dialect, in order to partially restore the original schema again

on a new database instance. This is particularly useful, for instance, when working with an Oracle production database, and an H2 in-memory test database. [25]

2.5.3 SQL Parser API

In the context of a database, the primary function of a SQL parser is to break down the SQL statements into smaller components, such as keywords, table names, column names, operators, and values. It checks the syntax and structure of the SQL statement to ensure it conforms to the rules of the SQL language. If the statement is not well-formed or contains errors, the parser will typically raise an error or exception.

The jOOQ's SQL Parser API [26] is accessible from the DSL API [27], which is the part of jOOQ that adds lexical convenience for programmers on top of the model API [28]. jOOQ's SQL Parser API can produce jOOQ API elements from arbitrary SQL string fragments.

SQL parser as a SQL translator

jOOQ's SQL Parser API can act as a SQL translator when merging two of jOOQ's features: the parsing of SQL queries to the jOOQ's internal domain-specific language in Java and the rendering of the query object model again into a SQL string written in the desired SQL dialect [29].

The result of this feature combination can be used online.

2.6 Is jOOQ an ORM?

Object-relational mapping (ORM) is a technique that is used to map data between a database and an object-oriented programming language. It is used to bridge the gap between the data model used by a DBMS and the object model used by an application.

This chapter will explain the main differences between jOOQ and ORMs by comparing jOOQ with JPA, a ORM solution used by Sakai LMS.

2.6.1 JPA

Data Persistence is a means for an application to persist and retrieve information from a non-volatile storage system. The Java™ Persistence API (JPA) provides a mechanism for managing persistence and object-relational mapping and functions since the EJB 3.0 specifications. JPA represents a simplification of the persistence programming model. The JPA specification explicitly defines the object-relational mapping, rather than relying on vendor-specific mapping implementations. [30]

Overall, JPA is a useful tool for developers who need to access and persist data in a Java application. It provides a simple and standard way to map Java objects to a relational database, which can help to simplify the development process and improve the maintainability of the code.

2.6.2 jOOQ vs JPA

The jOOQ API has a built Domain Specific Language (DSL) Application Programming Interface (API), that emulates SQL in Java. This allows to write complex SQL statements in Java code when it really is manipulating jOOQ objects that can be rendered to a SQL query to be executed in a database.

However, jOOQ shouldn't be seen as a JPA replacement, not only when creating a new application, but also when inserting jOOQ into an existing application that uses JPA.

The main differences between both solutions rely on the relation between the application and the database [31]:

1. Will the application design drive the data model, or will the data model drive the application design?
    - If the purpose of the application's database is just to persist data, Hibernate is the simplest choice, especially in a project's early stages. Because the database schema can be easily generated from the Entity model.
    - But in cases where the database is expected to outlive the application and be accessed by multiple applications, database design is a priority. So, jOOQ could be more appropriate, since it allows easy construction and manipulation of objects that can generate SQL queries that fit the database.
2. Complex reading and light writing, or easy reading and heavy writing?
    - If the application's objects were created with the jOOQ's code generator, jOOQ can perform simple CREATE, READ, UPDATE, DELETE (CRUD) operations in a database [32]. But its main focus is executing actual SQL statements. In other words, jOOQ's DSL API allows writing complex SQL statements with more safety and less coding. JPA can also perform more inventive SQL statements with the Java Persistence Query Language [33], however, it doesn't support DML operations and the syntax is limited by SQL standards.
    - jOOQ is prepared to support elaborate reading/writing operations from a set theory context. But when these operations implicate loading a complex object graph with multiple entities involved into memory, performing optimistic locking on it, modifying it in many different ways, and then persisting it again in one go, then jOOQ won't be helpful. This is what Hibernate was originally created for.

2.6.3 jOOQ is not the typical ORM

Like JPA and other ORMs, jOOQ facilitates CRUD by using a specific API. But jOOQ could not be considered a usual ORM, because it is not capable of robust writing to solve the object graph persistence problem. Instead, jOOQ should be seen as a solution for embedding SQL in Java code.

jOOQ is the answer

jOOQ suits this thesis problem because it doesn't act as a tool that persists data in a vendor-agnostic way, but as a tool that standardizes SQL. jOOQ's SQL Parser API can reduce any given query written in MySQL or Oracle to its own data model and render it to a Postgres query as it was a live translation technique, which isn't something that can be achieved by the other tools reviewed in this chapter. Therefore, and given that Sakai LMS relies on a big amount of pre-built SQL queries, jOOQ fits the best, theoretically, as a solution to this project. The approach design and implementation phases will prove that jOOQ can be as suitable as it seems.

Table 2.1 compares the referenced tools in 2.4 with jOOQ.

	Live Translation	Source schema conversion and target schema import	Target database Definition Queries Generation	Open-source	MySQL to Postgres conversion	Oracle to Postgres conversion
Data Loader	✗	✓	✗	✓	✓	✓
MySQL Migration Toolkit	✗	✓	✓	✓	✗	✗
Ora2Pg	✗	✗	✓	✓	✓	✓
EDB Postgres Migration Portal	✗	✗	✓	✓	✗	✓
Ispirer SQLWays Toolkit	✗	✓	✓	✗	✗	✓
jOOQ	✓	✓	✓	✓	✓	✓ Not available in the open source version

Table 2.1: Schema conversion tools

Chapter 3

Approach

This chapter contains the architectural drivers of the proposed software, a description of the experiments made to validate the fundamentals of the chosen approach, and the changes that will be made in Sakai's architecture.

3.1 Architectural drivers

Architectural drivers are requirements that shape architecture. Therefore every iteration of the software architecture design depends on its fit with the architectural drivers. Only after positive approval from the drivers' perspective, can the architecture design be executed.

3.1.1 Functional Requirements

The following functional requirements explain what is expected the system to do. For the development of the architecture design, only the high-level functionality of each requirement is expected. A use case will be defined to describe each functional requirement.

Use cases

The first requirement states that Sakai must be able to start and run with PostgreSQL as its database (Table 3.1).

Sakai contains a SQL service component that provides for SQL queries in two dialects, MySQL and Oracle. So, for the second functional requirement, it is predicted that Sakai can translate its SQL service queries to Postgres (Table 3.2).

Name	Using a Postgres database
Description	If a user wants to run Sakai with a Postgres database, they must be able to do it
Actor	Sakai LMS end-user
Precondition	The user set the sakai.properties file to run Sakai on a Postgres database
Flow	User starts Sakai. After startup, the user can use Sakai. If the user decides to check the database defined in sakai.properties, the user will observe that the database is filled with over 320 tables that contain the data created by Sakai LMS

Table 3.1: Use case 1 - Using a Postgres database

Name	Translate queries
Description	Sakai must be able to translate the SQL service queries (written in MySQL or Oracle dialect) to Postgres
Actor	Sakai LMS kernel
Precondition	The Sakai kernel invokes a SQL query from its SQL service.
Flow	The SQL service returns a translated Postgres query, instead of the original MySQL or Oracle query

Table 3.2: Use case 2 - Translate queries

3.1.2 Technical/Business constraints

This thesis's work will be integrated with Sakai LMS, which is a large enterprise legacy system. As a result, the design decisions that shape the Sakai architecture will inevitably shape this project's architecture.

Table 3.3 represents the main constraints that affected the architecture design.

Technical/Business constraint	**Description**
Java	In order to implement the necessary changes in Sakai, all code had to be written in Java since that's the language by which the Sakai kernel runs
Open-source	Since this is a project for an open-source community, it would be fundamental that the developed work was oriented keeping open-source technologies and communities in mind. This means prioritizing low costs and easy customizability

Table 3.3: Technical Constraints

3.2 Experiments

This section clarifies the experiments taken to test the viability and expected difficulties in the integration of jOOQ and Postgres into the Sakai SQL service.

3.2.1 jOOQ

To test the jOOQ SQL translator, two independent projects were created. Each project contained queries that could reproduce identic databases one in Oracle and another one in MySQL. In each project, the queries were translated to Postgres with jOOQ, and then executed, to check for its validity.

In the Oracle project, out of 37 queries, jOOQ failed to translate one query (Java throws error "org.jooq.impl.ParserException" in the translation process) and also provided one translation that wasn't effective in the target database (Java throws error "org.postgresql.util.PSQLException").

jOOQ didn't seem to have any problem converting MySQL statements to Postgres. However, as seen in the Oracle translation, it is possible that jOOQ may not be always effective. Therefore, the same level of concern should be given to both dialect conversions. These tests also showed that jOOQ may divide a single query into multiple queries, in order to provide an effective translation.

SQL scripts to count and compare the constraints in each database, that were created to use in this experiment and in the implementation phase, proved that the jOOQ translations and the manually corrected translations generated similar databases to their original ones.

Overall, this experiment showed that, despite some incorrect translations, jOOQ was effective in translating most of the queries.

Pictures A.1 and B.1 display the results of the standalone projects used to test jOOQ.

3.2.2 Postgres connection

Another key element of the project would be the connection to a Postgres database. Since Sakai uses JDBC to connect to a database, the solution would implicate redefining the properties file that defines the database used by the application:

```
vendor@org.sakaiproject.db.api.SqlService=mysql #or oracle
driverClassName@javax.sql.BaseDataSource=org.postgresql.Driver
hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
url@javax.sql.BaseDataSource=
        jdbc:postgresql://localhost:5432/sakai_pg_db
username@javax.sql.BaseDataSource=postgres_user
password@javax.sql.BaseDataSource=postgres_password
```

```
validationQuery@javax.sql.BaseDataSource=
        SELECT 1 FROM pg_catalog.pg_tables LIMIT 1
```

This part wasn't exposed to some direct testing, because, inevitably, the Sakai connection to Postgres would be tested in the next experiment.

3.2.3 Sakai SQL service integration with jOOQ and Postgres

In the initial approach, the biggest change in the Sakai LMS kernel would be the integration with jOOQ and Postgres, by inserting a Postgres Translations Component that would receive every query that went through the SQL service, jOOQ-translate it and execute in Postgres. For this experiment, we registered every SQL query that was translated, and its translation. From 3.2.1, we knew that there were going to be faulty translations, so we decided to check them. We identified 3 types of queries that would lead to inexecutable jOOQ translations (Tables 3.4, 3.5, 3.6).

Description	jOOQ translation failed to execute properly
Error	org.postgresql.util.PSQLException: ERROR: operator does not exist: character = integer
Postgres Translation query	create table SAKAI_SITE_PAGE (PAGE_ID varchar(99) not null, SITE_ID varchar(99) not null, TITLE varchar(99) null, LAYOUT char(1) null, SITE_ORDER int not null, POPUP char(1) null, check (POPUP in (1, 0)))
MySQL query	CREATE TABLE SAKAI_SITE_PAGE (PAGE_ID VARCHAR (99) NOT NULL, SITE_ID VARCHAR (99) NOT NULL, TITLE VARCHAR (99) NULL, LAYOUT CHAR(1) NULL, SITE_ORDER INTEGER NOT NULL, POPUP CHAR(1) NULL CHECK (POPUP IN (1, 0)))

Table 3.4: Error example 1 - Unsuccessful jOOQ translation

From these errors we can understand that:

- Human validation of all queries must be enabled, to correct faulty translations;
- Other components in the Sakai LMS kernel bind params to queries and retrieve the information needed from the database. Trying to figure out where

Description	jOOQ failed to translate original query
Error	org.postgresql.util.PSQLException: ERROR: syntax error at or near "org"
Postgres Translation query	org.jooq.impl.ParserException: Unexpected clause: [1:111] ..._NAME='SAKAI_SESSION' ORDER BY CREATE_TIME LIMIT 1[*];
MySQL query	select TABLE_ROWS FROM information_schema.TABLES WHERE TABLE_NAME='SAKAI_SESSION' ORDER BY CREATE_TIME LIMIT 1;

Table 3.5: Error example 2 - jOOQ Error

Description	Templated SQL without binding params causes a syntax error
Error	org.postgresql.util.PSQLException: ERROR: syntax error at or near ";"
Postgres Translation query	select USER_ID from SAKAI_USER_ID_MAP where EID=?;
MySQL query	select USER_ID from SAKAI_USER_ID_MAP where EID=?;

Table 3.6: Error example 3 - Templated SQL

are those statements built and executed, would require more alterations on code and an understanding of the current Sakai architecture, which isn't documented or known, than changing the SQL statements when going through the SQL Service.

In the end, it could be a good idea to responsibilize the Postgres Translation Component with only the translation of queries, rather than take the SQL service place in the communication with the database.

3.3 Architecture design

The next subsections define the architecture design iterations for the project and the reasons that make them valid or not.

3.3.1 First architecture design iteration

At the beginning of the semester, the only predicted change was the insertion of a Postgres Translation Component in the communication between the SQL service and the Sakai database. This would mean that the Postgres Translation Component would receive queries from the SQL service, translate them, and execute them in the Postgres database. Figure 3.1 shows what was initially planned in this stage.

However, after running the experiments in 3.2, we concluded that this architecture suggested very basic and unrealistic means to the desired goal, running Sakai LMS with Postgres.

This happened because, although this iteration satisfied the technical constraints, from the functional requirements point of view this architecture wasn't developed enough to fulfill the functional requirements, translate queries correctly and, consequently, run Sakai on a Postgres database.

3.3.2 Second architecture design iteration

After the experiments in 3.2, we knew that the next architecture design iteration would have to maintain a direct connection between the SQL service, as well as enable human approval and correction of the jOOQ-generated queries.

The figures 3.2 and 3.3, represent the new Container diagram for Sakai LMS and the new Component diagram for the Sakai LMS kernel. The differences from the previous architecture are:

1. **The SQL service won't change its usual communication with the Sakai database** - The SQL service will receive the translated Postgres queries from the Postgres Translation Component and execute them into the Sakai database.
2. **The edition and approval of jOOQ translations** - The edition and approval of jOOQ translations will happen with the use of the Translations database, which contains a table that relates a SQL service query with its translation to Postgres, as well as a flag that informs if the translation has been revised or not. On every Sakai run this table gets unregistered queries, that can be edited and must be approved, to be used on the next run.

From the functional requirements perspective, this architecture is approved since it plans for Sakai to run in a Postgres database, by translating the SQL service queries.

At this point, the proposed architecture didn't seem to need more refinement, as it was authorized from the standpoint of the requirements, and there were no more uncovered errors or experiments to perform.

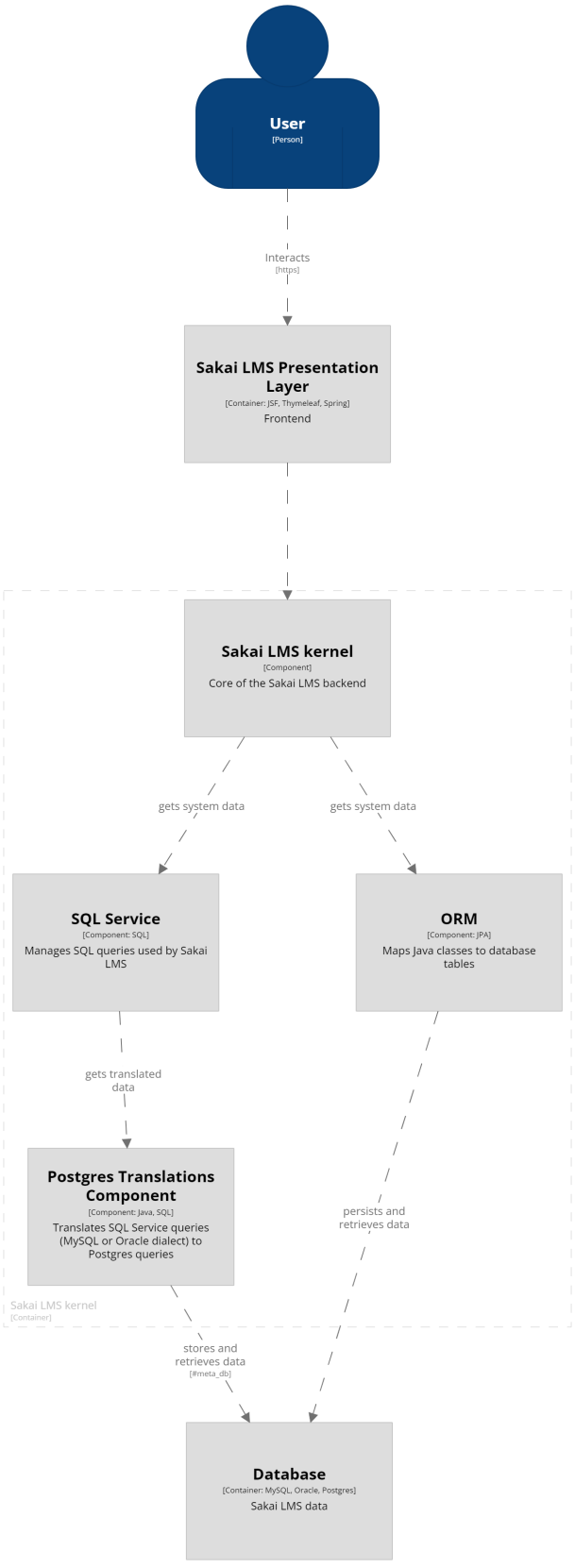
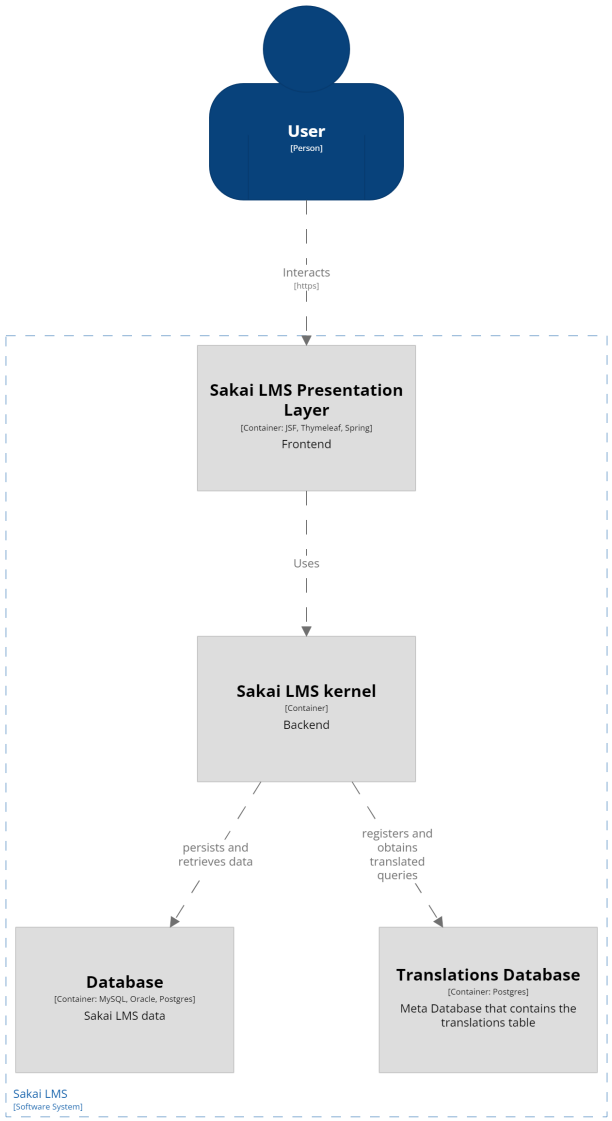


Figure 3.1: Inital Component diagram for Sakai LMS kernel proposal



[Container] Sakai LMS

Figure 3.2: New Container diagram for Sakai LMS

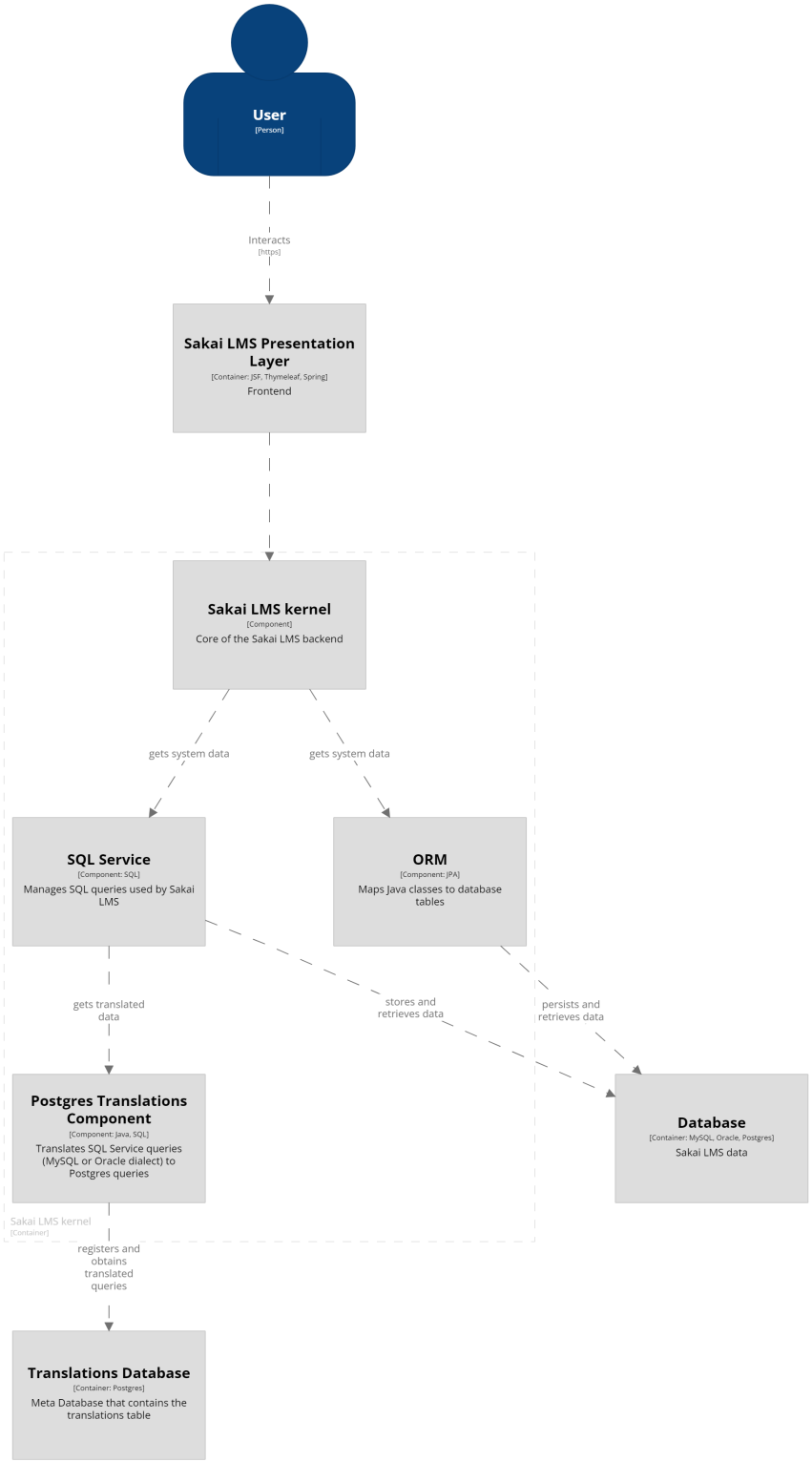


Figure 3.3: New Component diagram for Sakai LMS kernel

Chapter 4

Implementation

This chapter has the purpose to provide a closer look at the development of the work that led Sakai to work with a Postgres database, by translating its SQL service. First, we define an endpoint that allows guidance and a final evaluation for this work's implementation. Then, we'll describe in detail how the main object of the implementation, the Postgres Translations Component (PTC), works, its relation with the rest of the implementation objects, and its impact on the Sakai LMS architecture. After illustrating the core implementation, it's presented the steps to recreate the evolution of the translations table, and its influence on Sakai LMS's goal to run using Postgres. Finally, we'll make use of the initial endpoint defined at the beginning of the chapter, to analyze the implementation's result.

4.1 Defining parameters for a final comparison

Defining a possible endpoint for this project could help guide the implementation and allow more concrete conclusions in the evaluation phase.

For this, we used the jOOQ's functionalities of code generation and ddl queries generation to create a Postgres database. First, the code generator would take an instance of the Sakai database schema in MySQL and generate the Java classes, then jOOQ would get, from the classes that were created, the ddl queries that could replicate the initial MySQL database schema into Postgres. This predicted database originated from a MySQL one, was because despite jOOQ successfully generating the Java classes from Oracle, it failed to generate the ddl queries that would recreate a Postgres database.

This jOOQ-generated Postgres database was created from an instance of the MySQL database used by Sakai LMS after a complete startup because the goal of the project is to study database portability in Sakai, rather than to translate the whole application's database.

The only evidence that this database was a correct replication of the original MySQL database, was the fact that both databases had the same number of tables and constraints, 328 and 2845, respectively. But if we got the end result of the

implementation to match this expected database schema and, on top of that, start the Sakai LMS, this could add to the set of proofs that jOOQ can be a very useful database converter.

4.1.1 What will be compared exactly?

We will compare the jOOQ database and the implementation schemas' tables and constraints.

To combine all existing constraints in the database into a single table, we created a SQL statement that joins information from tables in the non-public schemas in Postgres (information_schema and pg_catalog). Each row of this constraint table contains information on the constraint's name, type, table, column, condition (for the check and not null constraints cases), and the referenced column (for the foreign key constraints).

This way it's possible to know what tables and constraints are present in the jOOQ-generated database, that weren't created in the implementation's database.

4.2 The Postgres Translation Component

The PTC is the main object of this thesis implementation. This component is more active in its startup and when is required to provide a translation for a MySQL or Oracle query.

The Translations Hashmap

To avoid multiple connections to retrieve the translation for a pre-existent query in the Translations Database, PTC contains a Translations Hashmap (TH) that is loaded in one go with all the approved translations and their original queries. For the remainder of the Sakai run, the TH is the entity that provides the translations for known queries in the project.

4.2.1 Startup

At any Sakai startup, the PTC will retrieve the queries and their translations (if approved), from the Translations Database, and store them into the TH that will be used during runtime.

When a PTC instance is created, it requires two parameters, the SQL dialect used by the SQL service and a boolean value that informs if the current Sakai run intends to use a default TH or the dynamic one in the Translations Database.

If the developer doesn't want to use a default TH, the PTC will create the connection to the Translations Database (this database must have been created by the

developer before starting the Sakai LMS), create the translations table if it doesn't exist, and read the translations table to insert the approved translations in the TH.

4.2.2 Translation Process

All the SQL queries that go through the SQL service, will be looked up in the PTC's TH, in order to obtain a translation.

In case there is any SQL statement that is not in the TH, jOOQ will translate it to be used in the current run, and register a new row in the Translation Database table for the original dialect.

For example, if the MySQL service is being used, and a new query is not recognized by the TH, the PTC will execute, in the Translations Database, the query:

```
INSERT INTO MySQL_translations (mysql_query, jooq_translation, approved)
VALUES ('original_mysql_query', 'translated_postgres_query', 'NOT TESTED')
ON CONFLICT DO NOTHING
```

The translations table (in this case MySQL_translations) has a unique index in the original query column (mysql_query), which doesn't allow the insertion of rows with a query that already exists in the table.

It is also important to say that, this new jOOQ-translation won't be used in the TH of the next Sakai LMS run unless it's approved. A translation is approved when the developer edits the translations table in the Translation Database, and changes the "approved" column to something other than 'NOT TESTED'.

Finally, even if the jOOQ fails to translate, the translations table will receive the jOOQ error in the column of the jOOQ translation:

```
'original_mysql_query', 'jooq_error', 'NOT TESTED'
```

But the original query will be executed in the current run, and hopefully, it can be successful.

Figure 4.1 is a representation of what the translation process in PTC looks like.

4.3 Starting Sakai with a Postgres database

Having created the necessary software elements for the execution of this work, we can now choose one of the Sakai SQL service dialects, MySQL or Oracle, to translate to Postgres. And now it can start the recursive process that aims to make Sakai able to run Postgres.

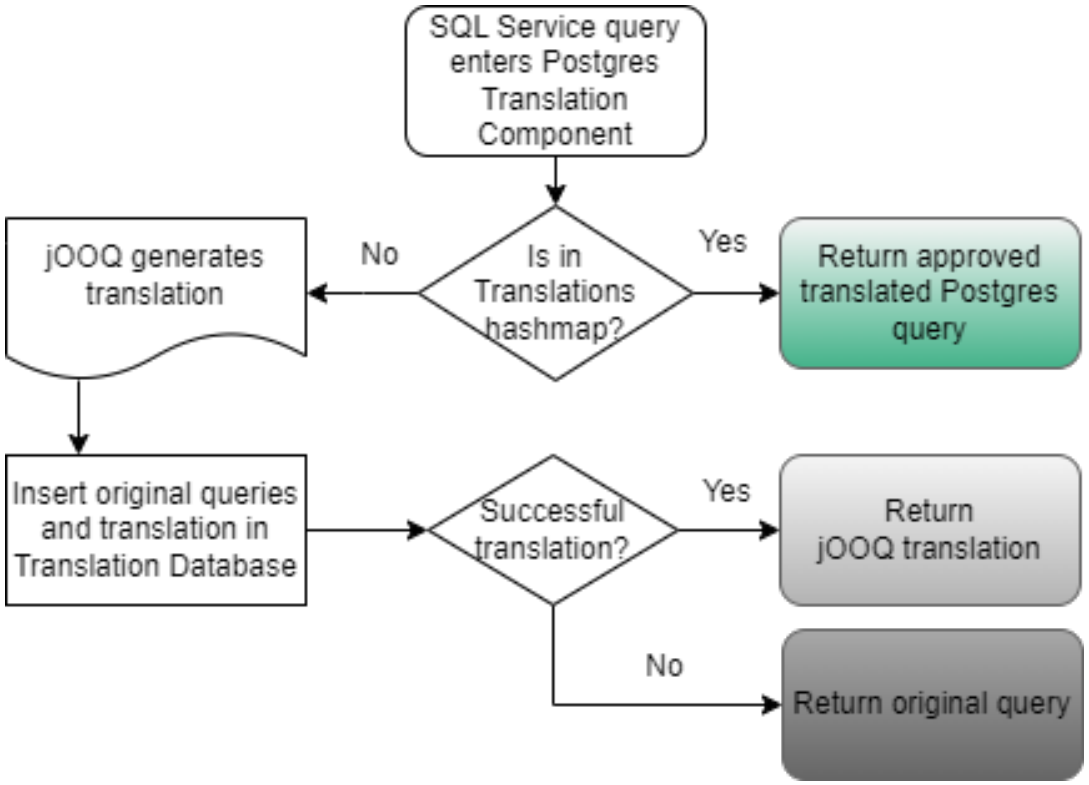


Figure 4.1: Translation Process

4.3.1 Compare dialects for a starting point

The Sakai LMS user can choose one of two dialects to run the Sakai LMS, these are MySQL and Oracle. For reasons of time efficiency, it was decided that the project goal should be achieved by focusing on translating only one of these dialects to Postgres.

To get to a decision, we would have to compare jOOQ's performance on each of the dialects. So we'd start Sakai LMS with each of the two SQL services, translate every query, and execute every translation in a Postgres database. From those executions, we get the number of successful queries, the number of tables created, and the number of constraints created (Table 4.1).

	Oracle	**MySQL**
Successful translations	83 in 91	107 in 116
Tables created	264	271
Constraints created	2205	2154

Table 4.1: Performance of jOOQ in Oracle and MySQL

Just because jOOQ received more queries for translation from the SQL service, doesn't prove that jOOQ actually performs better on MySQL queries than Or-

acle queries. So to determine the difference in jOOQ performance between the two dialects, we can use the values of successful translations in a z-test for two independent samples.

Considering the null hypothesis to be $H0$: "jOOQ shows the same performance for both dialects at a significance level of 0.05", and the sample sizes and proportions for the experiment, the number of queries and successful translations, respectively, the value of z is -0.2686, making the value of p 0.78716. The result is not significant at $p < 0.05$, so there is no evidence that jOOQ generates more successful translations from one dialect to another.

When we look at the number of tables and constraints created, it's important to say that most Sakai tables are created from Hibernate, instead of SQL queries generated from jOOQ. However, comparing both results with the expected database results (328 tables and 2845 constraints), it's pretty eye clear that the jOOQ performance doesn't differ much in the generation of tables and constraints.

Since there are no major differences in the jOOQ performance in the conversion of each dialect to Postgres, it was decided to proceed with the implementation with the MySQL SQL service, as a consequence of:

1. The approach's business constraint in 3.1.2, of prioritizing open-source technologies. There's greater support in the Sakai and open-source communities for MySQL than for Oracle, and the jOOQ open-source version works with MySQL, while Oracle is only supported by the paid versions.

2. The number of total original queries processed, meaning that the Sakai startup got further with a translated MySQL service than with a translated Oracle service. Although jOOQ performance isn't affected by the original dialect, it shows that Sakai acts differently from one dialect to another.

3. The fact that the jOOQ-generated Postgres database originated from MySQL

4.3.2 Improve the translations table

After choosing a dialect to translate, the evolution of the translations table started.

Starting with an empty Translations Database, the PTC creates, in its first startup, an empty translations table. In each Sakai run, the PTC will load into its TH the approved translations in the translations table and register new queries, found during that Sakai run, and their jOOQ translations into the translations table.

This process has the objective to supplement the translations table, to not only have a more complete default TH in PTC that doesn't need to connect to a database but also to understand the impact of each iteration of the translations table evolution in the transition to a successful Sakai startup.

It only took 6 iterations until there were no more MySQL queries to be discovered or translations to be corrected, so it wasn't possible to add new tables or constraints in this approach.

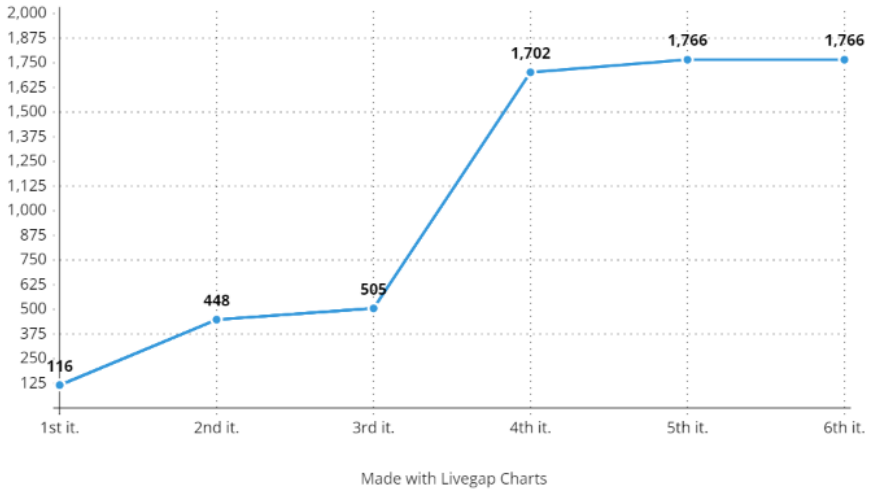


Figure 4.2: Evolution of the translations table in the number of translated queries

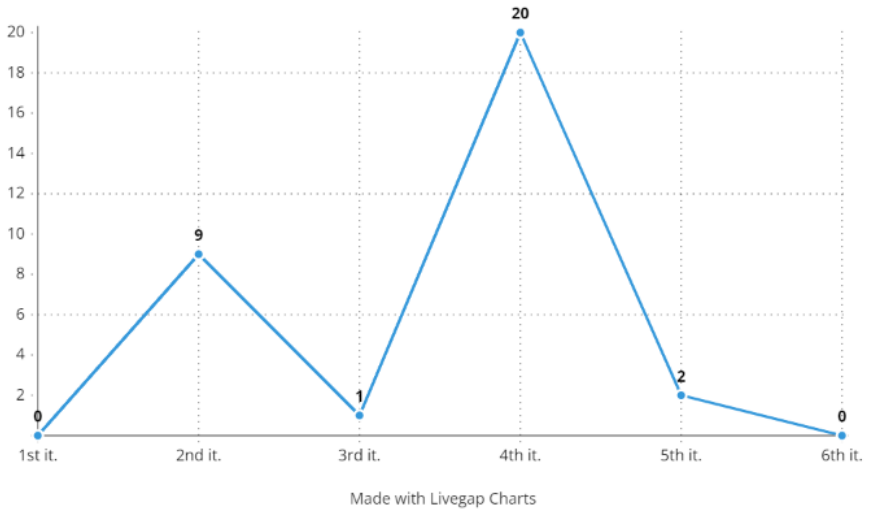


Figure 4.3: Evolution of the translations table in the number of corrected translations from the previous iteration

Figures 4.2 and 4.3 show the number of translations contained in each iteration of the evolution of the translations table, and how many of them were corrected right before that iteration. Figures 4.4 and 4.5 also show the impact of the evolution of the translations table in the creation of Sakai tables and constraints.

Figure 4.6 links the iteration of the evolution of the translations table with the output screen that appeared after a Sakai startup with that state of the translations table. A value of 0 means that such iterations couldn't even start Sakai. With a value of 1, that iteration led to the screen in figure 4.7. The last iterations could show the proper Sakai screen, 4.8

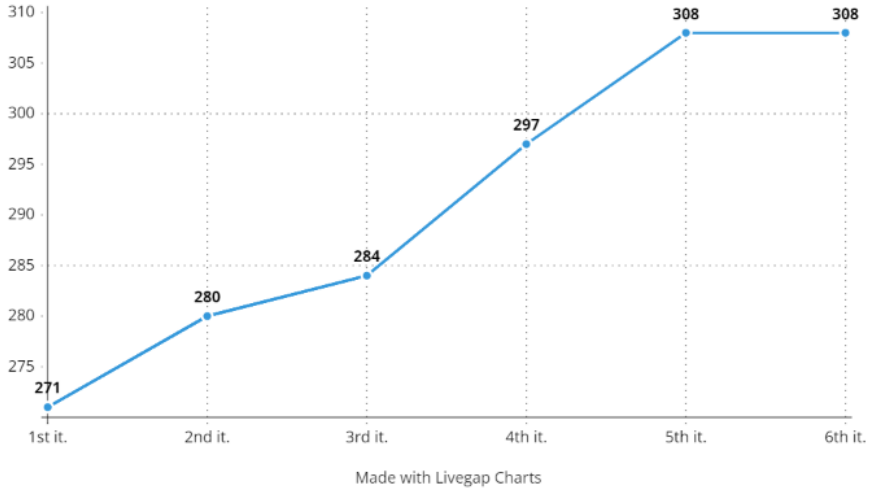


Figure 4.4: Impact of the translations table on the number of tables in the Sakai database

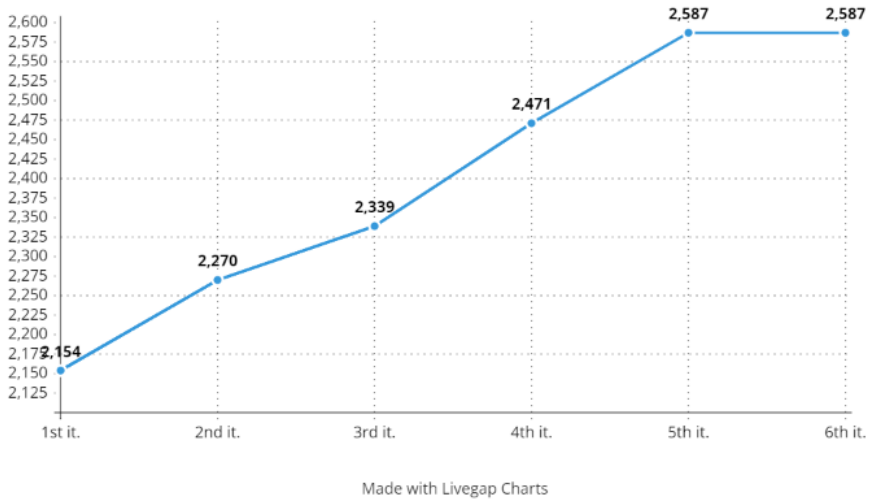


Figure 4.5: Impact of the translations table on the number of constraints in the Sakai database

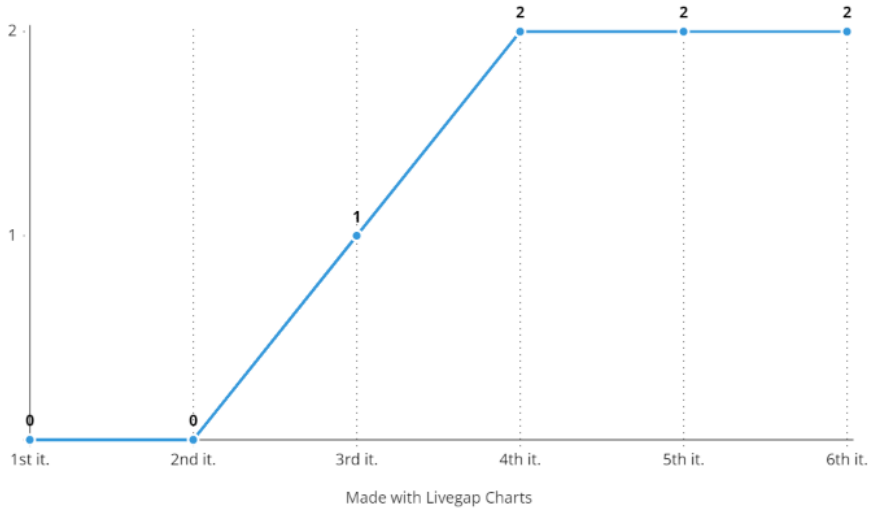


Figure 4.6: Startup screens after Sakai used each iteration of the translations tables

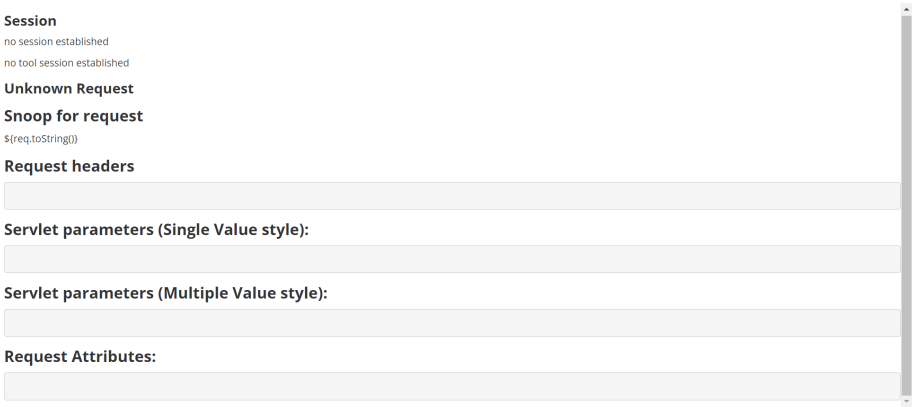


Figure 4.7: Screen 1 - screen after starting up Sakai in the third iteration of the translations table

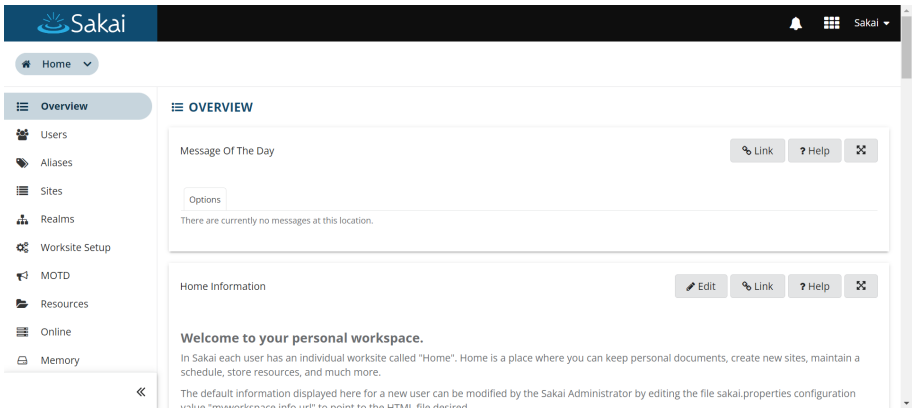


Figure 4.8: Screen 2 - screen after running Sakai in the fourth, fifth, and sixth iteration of the translations table

4.4 Implementation result

At the end of the improvement of the translations table, there were still tables, and consequently constraints, that weren't created in comparison with the jOOQ-generated database. These tables had no apparent influence on Sakai's startup with Postgres but were still inserted into the database to help realize if jOOQ's database replication in 4.1 was viable or not.

Those 20 missing tables and their constraints were created using the respective jOOQ ddl queries and after this manual insertion, we could verify the same number of tables and constraints, in both jOOQ-generated and implementation databases.

Could this mean that the manual validation failed at some point in the evolution of the translations table, which led to missing tables? No, because there was no query in the translations table that mentioned any of these tables. This tells us that the missing tables are created in some other part of the Sakai LMS that isn't the Sakai LMS kernel. After some research, the definitions of these tables were found in .sql files scattered across the Sakai LMS source code.

Chapter 5

Methodology and planning

This chapter describes the workflow of the first and second semesters. To enhance our organization and gain a clearer understanding of the tasks at hand, we opted to generate a Gantt chart illustrating the key assignments and their corresponding durations for this project's second semester.

5.1 First semester

Between October and January, monthly meetings were held with Professor Bruno Cabral and the internship supervisor, Dr. Charles Severance, to make sure that the internship was meeting its objectives.

Nevertheless, meetings on an almost weekly basis with the internship supervisor Dr. Charles Severance were fundamental to the development of the thesis document, as well as to the definition of the whole project.

5.2 Second semester Planning

This semester was expected to continue, just like the previous one, with the regular meetings with Dr. Charles Severance, as well as the monthly meetings with Professor Bruno Cabral.

In the second semester's planning, the project was divided into two major parts. As figure 5.1 shows, it was expected that the first month and a half of the internship would set the work for the real purpose of the project, the conversion of the Sakai LMS's SQL service, in the following two and a half months.

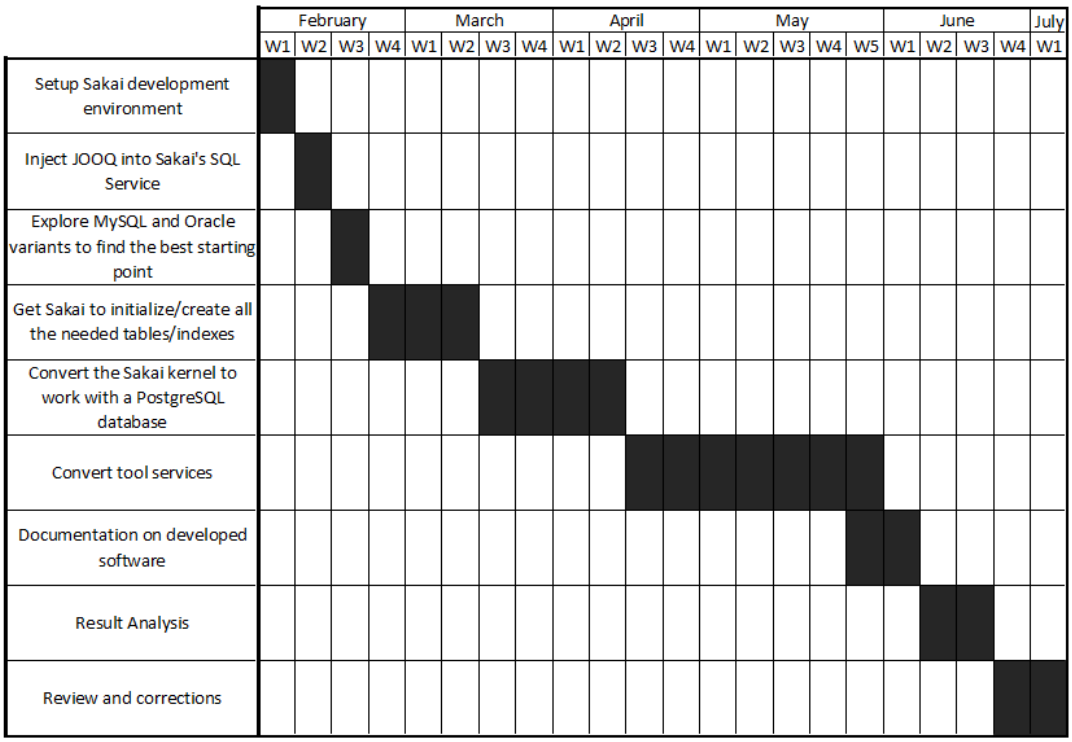


Figure 5.1: Gantt diagram for the second-semester initial planning

However, as the development setup evolved and the Sakai LMS was modified to work with jOOQ and Postgres, it started to become clear that the planning for the semester wasn't accurate and needed to be adjusted. Therefore, new planning was defined (figure 5.2), which resulted in a more complete and precise architecture design. This time the main priority would be to optimize the translation process, then compare the new software's performance in both SQL services and ultimately finalize the translation process by improving the Translations Hashmap.

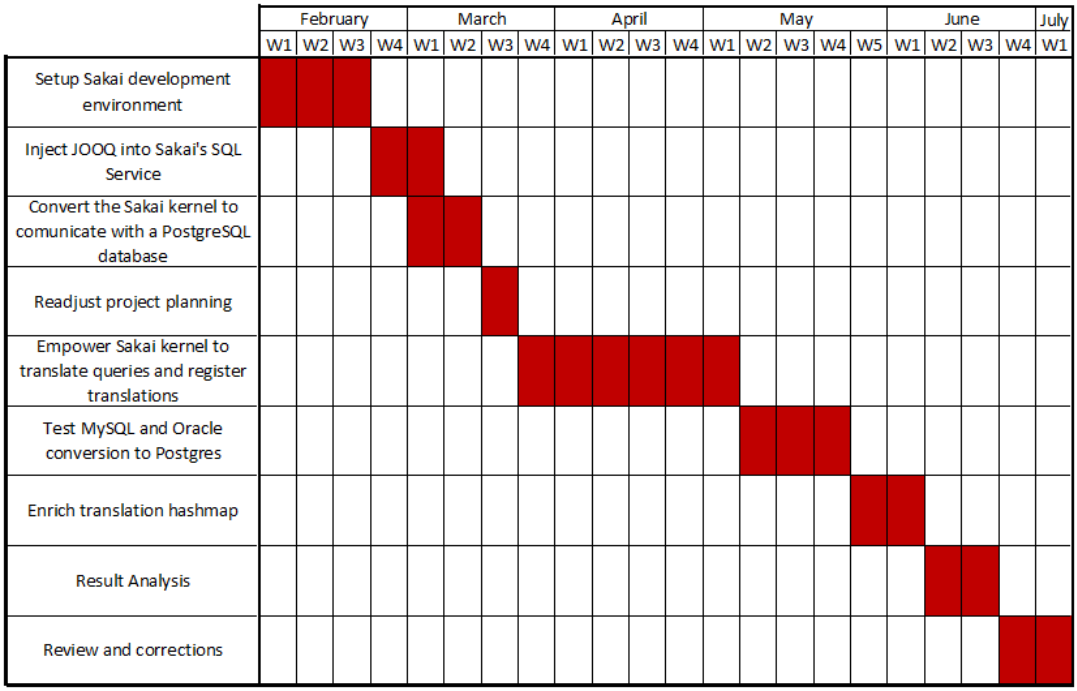


Figure 5.2: Real Gantt diagram of the second semester

Chapter 6

Conclusion

This chapter delivers a concise overview of the primary contributions and discoveries made in this work. It also provides some investigation that can be done to extend the work from this thesis.

6.1 Contributions and findings

This thesis work contributed to the progress in the challenge of inserting the PostgreSQL dialect into SakaiLMS:

- Sakai LMS can now start up with a PostgreSQL database, using a MySQL SQL service that gets translated to Postgres.

- The proposal of an architecture that can build the base for a translations server that enables Sakai LMS to be fully compatible with Postgres and, thanks to the way jOOQ's SQL translator works, other relational databases, without modifying the SQL service.

In addition to that, this work also allowed us to conclude that jOOQ's SQL translator works for most translations, but not always. So it's necessary to modify any jOOQ SQL translator layer to cope with the errors it throws. As we could conclude, manual edition and approval of unsuccessful translations are enough.

Another conclusion taken was that the SQL service in Sakai isn't the only entity responsible for executing SQL statements in Sakai. This turned out to be a limitation in the work's results because the proposed architecture wasn't expecting that behavior from Sakai. This doesn't reflect a faulty construction of the project architecture but the gathering of inaccurate information on the Sakai LMS's data control.

6.2 Future work

Concerning future work, we consider that there are other experimentations that can be done in this domain:

- Explore the translation of the SQL queries that don't pass throw the SQL service, by translating their .sql files or redirecting such queries to the Postgres Translations Component.
- Try to translate the Oracle SQL service to Postgres. To get a more concise decision on the jOOQ difference in performance between these two dialects, in the Sakai LMS context.

References

[1] What is open source? - www.opensource.com/resources/what-open-source.

[2] Open-Source Software Benefits - What to Know - www.heavybit.com/library/article/open-source-software-benefits-advantages.

[3] Apereo Foundation - www.apereo.org.

[4] Sakai LMS - www.sakailms.org.

[5] How To Find Your Way Through the Different Types of SQL - www.towardsdatascience.com/how-to-find-your-way-through-the-different-types-of-sql-26e3d3c20aab.

[6] PostgreSQL vs. MySQL: A 360-degree Comparison [Syntax, Performance, Scalability and Features] - www.enterprisedb.com/blog/postgresql-vs-mysql-360-degree-comparison-syntax-performance-scalability-and

[7] 1.6.3 How MySQL Deals with Constraints - https://dev.mysql.com/doc/refman/8.0/en/constraints.html.

[8] 13.1.20.6 CHECK Constraints - https://dev.mysql.com/doc/refman/8.0/en/create-table-check-constraints.html.

[9] 5.4. Constraints - https://www.postgresql.org/docs/current/ddl-constraints.html.

[10] Porting from Oracle PL/SQL - https://www.postgresql.org/docs/current/plpgsql-porting.html.

[11] Data Definition Language (DDL) Statements - https://docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/sql.html#GUID-C25B548B-363A-4FE5-B4EE-784502BAAD08.

[12] Chapter 5. Data Definition - https://www.postgresql.org/docs/current/ddl.html.

[13] Working with Postgres Audit Triggers - https://www.enterprisedb.com/postgres-tutorials/working-postgres-audit-triggers.

[14] Data Manipulation Language (DML) Statements - https://docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/sql.html#GUID-90EA5D9B-76F2-4916-9F7E-CF0D8AA1A09D.

[15] Chapter 6. Data Manipulation - https://www.postgresql.org/docs/current/dml.html.

[16] MERGE - https://www.postgresql.org/docs/current/sql-merge.html.

[17] LOCK - https://www.postgresql.org/docs/current/sql-lock.html.

[18] Types of Integrity Constraints - https://docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/data-integrity.html#GUID-1C9665AD-A444-4AFB-984F-6385FCBEA64E.

[19] Data Loader - www.dbload.com.

[20] MySQL Migration Toolkit - https://downloads.mysql.com/docs/migration-toolkit-en.pdf.

[21] ora2pg - www.ora2pg.darold.net.

[22] EDB Migration Portal - www.enterprisedb.com/docs/static/9c9c01a9056ac239b4dedae97f56132e/migration_portal_v4_documentation.pdf.

[23] Ispirer SQLWays Toolkit - www.sqlways.com.

[24] jOOQ Code Generation - www.jooq.org/doc/latest/manual/code-generation/.

[25] jOOQ Generating DDL from objects - www.jooq.org/doc/latest/manual/sql-building/ddl-statements/generating-ddl/.

[26] jOOQ SQL Parser API - www.jooq.org/doc/latest/manual/sql-building/sql-parser/sql-parser-api/.

[27] jOOQ DSL API - www.jooq.org/doc/latest/manual/sql-building/dsl-api/.

[28] jOOQ Model API - www.jooq.org/doc/latest/manual/sql-building/model-api/.

[29] SQL translator - www.jooq.org/doc/latest/manual/sql-building/sql-parser/sql-parser-translator/.

[30] Java Persistence API (JPA) - www.ibm.com/docs/en/was-liberty/nd?topic=overview-java-persistence-api-jpa.

[31] jOOQ vs Hibernate - https://blog.jooq.org/jooq-vs-hibernate-when-to-choose-which/.

[32] jOOQ Simple CRUD - www.jooq.org/doc/latest/manual/sql-execution/crud-with-updatablerecords/simple-crud/.

[33] Java Persistence Query Language - https://docs.oracle.com/html/E13946_01/ejb3_langref.html.

Appendices

Appendix A

jOOQ conversion test: Oracle to Postgres

ORACLE	POSTGRES
CREATE TABLE "users" (id NUMBER PRIMARY KEY, name VARCHAR2(50), email VARCHAR2(100), address VARCHAR2(200));	create table "users" (ID numeric, NAME varchar(50), EMAIL varchar(100), ADDRESS varchar(200), primary key (ID))
CREATE TABLE table1 (id NUMBER, name VARCHAR2(50), age NUMBER default 22, email VARCHAR2(100), address VARCHAR2(200), user_id NUMBER);	create table TABLE1 (ID numeric, NAME varchar(50), AGE numeric default 22, EMAIL varchar(100), ADDRESS varchar(200), USER_ID numeric)
CREATE TABLE table2 (id NUMBER, name VARCHAR2(50), age NUMBER, email VARCHAR2(100), address VARCHAR2(200), user_id NUMBER);	create table TABLE2 (ID numeric, NAME varchar(50), AGE numeric, EMAIL varchar(100), ADDRESS varchar(200), USER_ID numeric)
CREATE TABLE table3 (id NUMBER, name VARCHAR2(50), age NUMBER, email VARCHAR2(100), address VARCHAR2(200), user_id NUMBER);	create table TABLE3 (ID numeric, NAME varchar(50), AGE numeric, EMAIL varchar(100), ADDRESS varchar(200), USER_ID numeric)
CREATE TABLE table4 (id NUMBER, name VARCHAR2(50), age NUMBER, email VARCHAR2(100), address VARCHAR2(200), user_id NUMBER);	create table TABLE4 (ID numeric, NAME varchar(50), AGE numeric, EMAIL varchar(100), ADDRESS varchar(200), USER_ID numeric)
CREATE TABLE table5 (id NUMBER, name VARCHAR2(50), age NUMBER, email VARCHAR2(100), address VARCHAR2(200), user_id NUMBER);	create table TABLE5 (ID numeric, NAME varchar(50), AGE numeric, EMAIL varchar(100), ADDRESS varchar(200), USER_ID numeric)
ALTER TABLE table2 ADD CONSTRAINT pk_table2 PRIMARY KEY (id);	alter table TABLE2 add constraint PK_TABLE2 primary key (ID)
ALTER TABLE table2 ADD CONSTRAINT fk_table2_user FOREIGN KEY (user_id) REFERENCES "users"(id);	alter table TABLE2 add constraint FK_TABLE2_USER foreign key (USER_ID) references "users" (ID)
ALTER TABLE table3 MODIFY age DEFAULT 22;	org.jooq.impl.ParserException: Unexpected clause: [3:40]
ALTER TABLE table3 MODIFY name NOT NULL;	alter table TABLE3 alter NAME set not null
CREATE INDEX idx_address ON table4(address);	create index IDX_ADDRESS on TABLE4(ADDRESS)
ALTER TABLE table5 ADD CONSTRAINT uk_address UNIQUE (address);	alter table TABLE5 add constraint UK_ADDRESS unique (ADDRESS)
INSERT INTO table1 (id, name, age, email, address) VALUES (1, 'John', 25, 'john@example.com', '123 Main St');	insert into TABLE1 (ID, NAME, AGE, EMAIL, ADDRESS) values (1, 'John', 25, 'john@example.com', '123 Main St')
INSERT INTO table2 (id, name, age, email, address) VALUES (1, 'Emma', 27, 'emma@example.com', '987 Walnut St');	insert into TABLE2 (ID, NAME, AGE, EMAIL, ADDRESS) values (1, 'Emma', 27, 'emma@example.com', '987 Walnut St')
INSERT INTO table2 (id, name, age, email, address) VALUES (2, 'David', 32, 'david@example.com', '741 Birch St');	insert into TABLE2 (ID, NAME, AGE, EMAIL, ADDRESS) values (2, 'David', 32, 'david@example.com', '741 Birch St')
INSERT INTO table3 (id, name, age, email, address) VALUES (1, 'Michael', 33, 'michael@example.com', '369 Cedar St');	insert into TABLE3 (ID, NAME, AGE, EMAIL, ADDRESS) values (1, 'Michael', 33, 'michael@example.com', '369 Cedar St')
INSERT INTO table3 (id, name, age, email, address) VALUES (2, 'Olivia', 26, 'olivia@example.com', '258 Pine St');	insert into TABLE3 (ID, NAME, AGE, EMAIL, ADDRESS) values (2, 'Olivia', 26, 'olivia@example.com', '258 Pine St')
INSERT INTO table3 (id, name, age, email, address) VALUES (3, 'Daniel', 23, 'daniel@example.com', '147 Walnut St');	insert into TABLE3 (ID, NAME, AGE, EMAIL, ADDRESS) values (3, 'Daniel', 23, 'daniel@example.com', '147 Walnut St')
INSERT INTO table4 (id, name, age, email, address) VALUES (2, 'Matthew', 25, 'matthew@example.com', '369 Cedar St');	insert into TABLE4 (ID, NAME, AGE, EMAIL, ADDRESS) values (2, 'Matthew', 25, 'matthew@example.com', '369 Cedar St')
INSERT INTO table4 (id, name, age, email, address) VALUES (3, 'Amelia', 30, 'amelia@example.com', '258 Pine St');	insert into TABLE4 (ID, NAME, AGE, EMAIL, ADDRESS) values (3, 'Amelia', 30, 'amelia@example.com', '258 Pine St')
UPDATE table3 SET name = 'Updated Name' WHERE id = 2;	update TABLE3 set NAME = 'Updated Name' where ID = 2
UPDATE table4 SET age = 29 WHERE id = 1;	update TABLE4 set AGE = 29 where ID = 1
SELECT * FROM table1;	select * from TABLE1
SELECT id, name, age FROM table2 WHERE age > 30;	select ID, NAME, AGE from TABLE2 where AGE > 30
SELECT * FROM (SELECT * FROM table3 ORDER BY name) WHERE ROWNUM <= 5;	org.postgresql.util.PSQLException: ERROR: column "rownum" does not exist
SELECT COUNT(*) FROM table4 WHERE name LIKE 'J%';	select count(*) from TABLE4 where cast(NAME as varchar) like 'J%'
SELECT name, address FROM table5 WHERE id IN (1, 3, 5);	select NAME, ADDRESS from TABLE5 where ID in (1, 3, 5)
SELECT AVG(age) FROM table1;	select avg(AGE) from TABLE1
SELECT DISTINCT age FROM table2;	select distinct AGE from TABLE2
SELECT name, COUNT(*) FROM table3 GROUP BY name;	select NAME, count(*) from TABLE3 group by NAME
SELECT * FROM table4 WHERE address IS NULL;	select * from TABLE4 where ADDRESS is null
SELECT id, name FROM table5 WHERE name LIKE '%son%';	select ID, NAME from TABLE5 where cast(NAME as varchar) like '%son%'
SELECT * FROM table1 WHERE age BETWEEN 25 AND 35;	select * from TABLE1 where AGE between 25 and 35
SELECT SUM(age) FROM table2;	select sum(AGE) from TABLE2
SELECT name, age FROM table3 WHERE age = (SELECT MAX(age) FROM table3);	select NAME, AGE from TABLE3 where AGE = (select max(AGE) from TABLE3)
SELECT MIN(age) FROM table4;	select min(AGE) from TABLE4
SELECT id, AVG(age) FROM table5 GROUP BY id HAVING AVG(age) > 30	select ID, avg(AGE) from TABLE5 group by ID having avg(AGE) > 30

Figure A.1: Result of jOOQ translations of Oracle queries to Postgres

Appendix B

jOOQ conversion test: MySQL to Postgres

MySQL	POSTGRES
CREATE TABLE users (id INT PRIMARY KEY, name VARCHAR(50), email VARCHAR(100), address VARCHAR(200))	create table users (id int, name varchar(50), email varchar(100), address varchar(200), primary key (id))
CREATE TABLE table1 (id INT, name VARCHAR(50), age INT default 22, email VARCHAR(100), address VARCHAR(200),user_id INT)	create table table1 (id int, name varchar(50), age int default 22, email varchar(100), address varchar(200), user_id int)
CREATE TABLE table2 (id INT, name VARCHAR(50), age INT, email VARCHAR(100), address VARCHAR(200), user_id INT)	create table table2 (id int, name varchar(50), age int, email varchar(100), address varchar(200), user_id int)
CREATE TABLE table3 (id INT, name VARCHAR(50), age INT, email VARCHAR(100), address VARCHAR(200), user_id INT)	create table table3 (id int, name varchar(50), age int, email varchar(100), address varchar(200), user_id int)
CREATE TABLE table4 (id INT, name VARCHAR(50), age INT, email VARCHAR(100), address VARCHAR(200), user_id INT)	create table table4 (id int, name varchar(50), age int, email varchar(100), address varchar(200), user_id int)
CREATE TABLE table5 (id INT, name VARCHAR(50), age INT, email VARCHAR(100), address VARCHAR(200), user_id INT)	create table table5 (id int, name varchar(50), age int, email varchar(100), address varchar(200), user_id int)
ALTER TABLE table2 ADD CONSTRAINT pk_table2 PRIMARY KEY (id)	alter table table2 add constraint pk_table2 primary key (id)
ALTER TABLE table2 ADD CONSTRAINT fk_table2_user FOREIGN KEY (user_id) REFERENCES users(id)	alter table table2 add constraint fk_table2_user foreign key (user_id) references users (id)
ALTER TABLE table3 ALTER COLUMN age SET DEFAULT 22	alter table table3 alter age set default 22
ALTER TABLE table3 MODIFY COLUMN name VARCHAR(50) NOT NULL	do $$ begin alter table table3 alter name type varchar(50); alter table table3 alter name set not null; end $$
ALTER TABLE table4 ADD INDEX idx_address (address)	create index idx_address on table4(address)
ALTER TABLE table5 ADD UNIQUE (address)	alter table table5 add unique (address)
INSERT INTO table1 (id, name, age, email, address) VALUES (1, 'John', 25, 'john@example.com', '123 Main St')	insert into table1 (id, name, age, email, address) values (1, 'John', 25, 'john@example.com', '123 Main St')
INSERT INTO table2 (id, name, age, email, address) VALUES (1, 'Emma', 27, 'emma@example.com', '987 Walnut St')	insert into table2 (id, name, age, email, address) values (1, 'Emma', 27, 'emma@example.com', '987 Walnut St')
INSERT INTO table2 (id, name, age, email, address) VALUES (2, 'David', 32, 'david@example.com', '741 Birch St')	insert into table2 (id, name, age, email, address) values (2, 'David', 32, 'david@example.com', '741 Birch St')
INSERT INTO table3 (id, name, age, email, address) VALUES (1, 'Michael', 33, 'michael@example.com', '369 Cedar St')	insert into table3 (id, name, age, email, address) values (1, 'Michael', 33, 'michael@example.com', '369 Cedar St')
INSERT INTO table3 (id, name, age, email, address) VALUES (2, 'Olivia', 26, 'olivia@example.com', '258 Pine St')	insert into table3 (id, name, age, email, address) values (2, 'Olivia', 26, 'olivia@example.com', '258 Pine St')
INSERT INTO table3 (id, name, age, email, address) VALUES (3, 'Daniel', 23, 'daniel@example.com', '147 Walnut St')	insert into table3 (id, name, age, email, address) values (3, 'Daniel', 23, 'daniel@example.com', '147 Walnut St')
INSERT INTO table4 (id, name, age, email, address) VALUES (2, 'Matthew', 25, 'matthew@example.com', '369 Cedar St')	insert into table4 (id, name, age, email, address) values (2, 'Matthew', 25, 'matthew@example.com', '369 Cedar St')
INSERT INTO table4 (id, name, age, email, address) VALUES (3, 'Amelia', 30, 'amelia@example.com', '258 Pine St')	insert into table4 (id, name, age, email, address) values (3, 'Amelia', 30, 'amelia@example.com', '258 Pine St')
UPDATE table3 SET name = 'Updated Name' WHERE id = 2	update table3 set name = 'Updated Name' where id = 2
UPDATE table4 SET age = 29 WHERE id = 1	update table4 set age = 29 where id = 1
SELECT * FROM table1	select * from table1
SELECT id, name, age FROM table2 WHERE age > 30	select id, name, age from table2 where age > 30
SELECT * FROM table3 ORDER BY name ASC LIMIT 5	select * from table3 order by name asc fetch next 5 rows only
SELECT COUNT(*) FROM table4 WHERE name LIKE 'J%'	select count(*) from table4 where cast(name as varchar) like 'J%'
SELECT name, address FROM table5 WHERE id IN (1, 3, 5)	select name, address from table5 where id in (1, 3, 5)
SELECT AVG(age) FROM table1	select avg(age) from table1
SELECT DISTINCT age FROM table2	select distinct age from table2
SELECT name, COUNT(*) FROM table3 GROUP BY name	select name, count(*) from table3 group by name
SELECT * FROM table4 WHERE address IS NULL	select * from table4 where address is null
SELECT id, name FROM table5 WHERE name LIKE '%son%'	select id, name from table5 where cast(name as varchar) like '%son%'
SELECT * FROM table1 WHERE age BETWEEN 25 AND 35	select * from table1 where age between 25 and 35
SELECT SUM(age) FROM table2	select sum(age) from table2
SELECT name, age FROM table3 WHERE age = (SELECT MAX(age) FROM table3)	select name, age from table3 where age = (select max(age) from table3)
SELECT MIN(age) FROM table	select min(age) from table4
SELECT id, AVG(age) FROM table5 GROUP BY id HAVING AVG(age) > 30	select ID, avg(AGE) from TABLE5 group by ID having avg(AGE) > 30

Figure B.1: Result of jOOQ translations of MySQL queries to Postgres