

Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

MATHIAS HANSEN
MASTER THESIS

AN ACCOUNTING TOOL FOR INNER SOURCE CONTRIBUTIONS

Submitted on March 19, 2018

Supervisor: Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Munich, March 19, 2018

License

This work is licensed under the Creative Commons Attribution 4.0 International license (CC BY 4.0), see <https://creativecommons.org/licenses/by/4.0/>

Abstract

Inner source (IS) is the application of Open Source principles to projects within organizational boundaries. Typically, this means that the artifacts and work items of inner source projects (ISPs) are accessible to everyone within the organization and cross-functional collaboration is encouraged. The Collaboration Management Suite (CMSuite) developed by the Open Source Research Group at the Friedrich-Alexander University Erlangen-Nürnberg supports the analysis and visualization of such collaborations. CMSuite lacks features to account for contributions to ISPs. To draw conclusions, a database engineer has to define complex series of SQL queries or rely on simple, singular value metrics. This thesis elaborates a REST-based architecture that can be used to retrieve accounting related data in the form of a timeline using Apache Lens and Java. The data sets are visualized in an Angular 2+ frontend where the user can navigate and filter them. Based on this visualization, a non-technical user is able to explore the historic data of ISPs without needing to understand the underlying data schema and without needing to define database queries. As a result, the added functionality will not only save time but also make the CMSuite accessible to a wider user group.

Contents

1	Accounting for inner source contributions	1
1.1	Previous work	1
1.1.1	REA model for accounting systems	1
1.1.2	Financial management using CMSuite	3
1.2	Purpose	4
1.3	Requirements	4
1.3.1	Stakeholders	4
1.3.2	Functional requirements	5
1.3.3	Non-functional requirements	6
1.3.4	Evaluation scheme	7
2	Architecture and design	8
2.1	Design alternatives	8
2.1.1	Native SQL or HQL	9
2.1.2	Reuse Pentaho Data Integration	10
2.1.3	OLAP approach	10
2.2	Apache Lens	12
3	Implementation	16
3.1	Database	18
3.1.1	Denormalization	20
3.1.2	Apache Lens OLAP cubes	25
3.2	REST services	35
3.2.1	Data serialization	35
3.2.2	Endpoints	40
3.3	Angular frontend	43
3.4	Deployment and testing	45
4	Evaluation	46
5	Future work	50
	Appendices	52

Introduction

Inner Source (IS) is the use of open source software development practices and the establishment of an open source-like culture within organizations (Capraro & Riehle, 2017). With IS becoming increasingly popular, further research is required, as noted by Riehle, Capraro, Kips, and Horn (2016). To analyze IS collaborations within organizations the Open Source Research Group at the Friedrich-Alexander University Erlangen-Nürnberg is developing the Collaboration Management Suite (CMSuite). The basis for the analysis is information extracted from source code repositories like Git. A crawler component is extracting metadata (such as author, date etc.) and storing it in a database together with organization context (for example, the functional structure of the organization). Additionally, metrics for the evaluation of the collected data can be defined and displayed.

The previous work on CMSuite focused on data extraction from repositories and the database design. It is currently not possible to explore the data collected without running SQL queries. The use of SQL as the sole means of access limits the user group to software and database developers. The objective of this thesis is to enable user groups without technical knowledge to explore CMSuite data, and thereby allow them to see and account for contributions from organizational units or individuals. For example, a project manager should be able to view a timeline of code contributions from the developers who fall under his or her responsibility.

The contributions of this thesis are:

- Service components for displaying, navigating and exploring inner source contributions.
- Client components for showing contribution journals of individuals, inner source projects and org. elements.
- Client-side filtering and navigating of the different journals and visualization of contributor/receiver relationships.
- A demo dataset “SampleCorp”, containing organizational units and individuals with descriptive names and two org. dimensions.
- Unit/Karma-Tests.
- A Docker development environment.

This document is structured as follows. [Chapter 1](#) provides an overview of financial and management accounting. It also elaborates whether and how accounting principles can be applied to inner source accounting. [Chapter 2](#) discusses design alternatives and highlights the architecture of the proposed solution. [Chapter 3](#) covers and relevant implementation details and discusses identified issues and solutions. [Chapter 4](#) concludes with an evaluation. Finally, areas of further work are suggested in [chapter 5](#).

1 Accounting for inner source contributions

CMSuite provides data to analyze code contributions within an organization. It extracts this information from source code repositories using a crawler mechanism. The crawler works in similar way to a website search engine index. As modern version control systems store metadata (such as the author's name and email address), this information can be used to link authors to their superior units within the organization. The author's email address as well as the affected software repository can be used to connect both to their respective organizational units. Now that the author (source) and the receiving project (sink) of a code contribution are identified, this information can be used to account for their activity over time.

1.1 Previous work

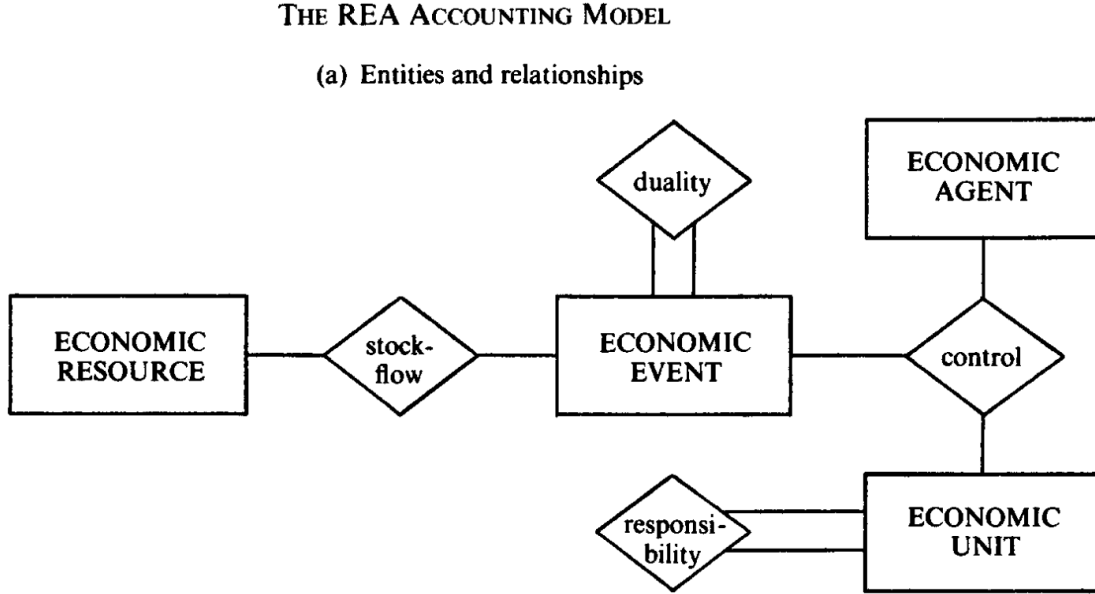
The following section provides an overview of the CMSuite database schema, and discusses inner source accounting in relation to the needs of financial management within organizations.

1.1.1 REA model for accounting systems

The Resources, Events, Agents (REA) model by McCarthy ([1982](#)) is a generalized accounting model, dropping double-entry bookkeeping paradigms on a database level. Double-entry bookkeeping is based on the idea of reducing and revealing errors during times when bookkeeping was based on paper-based ledgers. Modern database supported accounting no longer requires such means for error detection since account balances can be calculated when required.

Using generalization, McCarthy shows that accounting software can be modeled

Figure 1.1: REA accounting model by McCarthy (1982, p. 564)



using just four different entity types. See figure 1.1. The Duality Principle¹ as the basis for double-entry bookkeeping is evident. In database systems, McCarthy states it is sufficient to store the absolute² amount exactly once - as part of the economic event which induced the value change (i.e. a sale transaction). In CMSuite a code contribution represents a transaction between an individual and an inner source project. As a contribution does not directly carry a value amount, another means of quantification has to be used. Quantification and organizational aspects of inner source are subject to further research, as noted by Capraro and Riehle (2017) and are not in the scope of this thesis.

Parts of the CMSuite database schema are designed with the REA model in mind. The entities relevant to this thesis are:

- The *contributions* events log, storing the metadata of a contribution event (timestamp, receiver, contributor, etc.).
- The receiving *inner source project (ISP)* (sink).
- The contributing *individual* (source).
- The organizational hierarchy, defining the superior/subordinate reporting relationships between *organizational units*, ISPs and individuals.

¹Duality Principle: each transaction must involve exactly two accounts, classified as either debit or credit. An increment on one side leads to a decrement of the same amount on the other side.)

²Whether the amount is an increase or decrease depends on the viewer's perspective.

Within CMSuite the organizational hierarchy is logically segregated into different perspectives, called *organizational dimensions*. An organization can be logically structured in different ways at the same time, i.e. grouped by functional responsibilities (marketing, sales, etc.); by product lines (product A, product B, ...); by project teams; by geographical location and so on.

While the CMSuite database has been designed following the REA principles, CMSuite itself made no use of it. [Section 3.2.1](#) (p. 35) and [section 3.3](#) (p. 43) describe how the proposed solution takes advantage of the generalizations the REA model provides.

1.1.2 Financial management using CMSuite

Financial management distinguishes between three forms of accounting, namely *Tax Accounting*, *Financial Accounting* and *Managerial Accounting*. While the first two are dealing mostly with costs, the latter supports planning and decision making based on reporting and analytics. The CMSuite project is designed to support decision making and therefore focuses on the Managerial Accounting side.

Strategic decisions, however, are often based on reports that rely on financial information. Accounting, in general, has created various visualizations and reports to aid not only bookkeeping but also to present statistics. Because the underlying REA model originates in the accounting domain, some of these visualizations might apply to CMSuite. The proposed solution is showing contribution events in form of a "journal". The term was chosen deliberately as its original semantics from the bookkeeping domain can be transferred to inner source accounting.

In accounting and bookkeeping, a journal is a record of financial transactions in order by date. A journal is often defined as the book of original entry.

(Harold Averkamp, 2017)

It is important to note that CMSuite cannot provide any form of financial accounting. Financial accounting is based on a number of generally accepted principles. Most import the *Monetary Unit Assumption*. The assumption reads ‘financial accounting should record only those business events/transactions which are expressed in money terms’ (Jain & Khan, 2009, section 2.33) and further states that ‘transactions which cannot be quantified in financial terms, irrespective of their importance, are outside the scope of accounting records’. A (source code) contribution within inner source requires a certain amount of effort ³. However, it

³Expenses incurred and resources spent creating the contribution.

is difficult to estimate the effort spent writing source code solely based on repository metadata. While many organizations track and assign time budgets on a project or even work item level, the results are not accurate. Even when tracked at this level, efforts cannot be compared for a multitude of reasons, among them different skill levels or varying complexity.

It is likely, however, that comparing and monitoring cross-functional collaboration will become increasingly relevant, i.e. for resource planning. For example, development teams who contribute to an ISP belonging to a different functional unit could be compensated with increased budget and resources. Other units which profit from outside contributions but do not contribute themselves could face adjustments in the opposite direction. Even when management decides not to take action based on finances or costs, it is reasonable to monitor ‘debt’ and ‘credit’ of the involved organizational units in terms of resource consumption.

1.2 Purpose

Before this thesis, a software developer had to write SQL queries in PostgreSQL dialect to extract results from the CMSuite database. After writing and running a query, the results had to be exported as comma-separated values (CSV) before being post-processed using spreadsheets. Visualizations had to be configured manually.

This thesis extends CMSuite with components to directly visualize contribution journals. Different journals exist for inner source project, individuals, and org. elements. The journals are designed to be used by managers and other IS stakeholders to gain insights about the flow of contributions within their organization.

1.3 Requirements

1.3.1 Stakeholders

The stakeholders for the accountancy module are:

- An Inner Source Stakeholder, interested in gaining insights about ISPs within an organization.
- A Developer, creating and improving CMSuite features.
- An Administrator or DevOps engineer, deploying the different CMSuite component to staging/production environments.

-
- A Database Engineer, adapting the database to fit requirements and extracting information in the form of metrics and aggregations.

Different types of inner source stakeholders exist, such as contributors, committers and so on. These are relevant to understand inner source holistically. However, for the scope of this thesis, the differentiation is not important. The accountancy module is designed to provide analytical insights relevant to all inner source stakeholders.

1.3.2 Functional requirements

Multiple workshops and discussions based on user interface mock-ups (see [appendix A, figure 1](#)) lead to the definition of relevant user stories for inner source accounting.

The stories are grouped into two different feature clusters:

- Navigation within the organizational hierarchy and between the different journals.
- Exploration of accounting data (journal) of the different organizational elements.

In the following section, the identified requirements are listed, grouped by the aforementioned feature clusters. Their acceptance criteria are omitted here and instead will be elaborated as part of the conclusion.

Organizational hierarchy view

Q.1 As a user, I want to select which organizational dimension is the context for my analysis, so that I can navigate the logical hierarchy of the organization and select a specific element.

Contribution journals

R.1 As a user, I want to see a journal of received and provided contributions of the selected org. element and its children elements. The journal consists of a table containing the following columns: date, project agent, receiver agent (org. element), number of contributions and direction (see [R.4](#)). The selection within the organizational hierarchy view defines the receiving/-contributing org. elements I am interested in so that I get an overview of the contributions received (and provided) by the selected org. element.

-
- R.2** As a user, I want to be able to select the time range to display events for and to be able to aggregate the journal of contribution by time, so that I can choose the time range and granularity for the events I am interested in.
 - R.3** As a user, I want to aggregate the journal of contributions by date and filter by direction, so that I select the time granularity I'm interested in and also whether I want cross-boundary contributions to be included or not.
 - R.4** As a user, I want to see whether contributions cross the boundaries of the selected org. element⁴ or not, so that I can quickly identify them.
 - R.5** As a user, I want to select the hierarchy level to be shown for the receiver and the contributor, so that I can choose the level of detail of the involved org. elements.
 - R.6** As a user, I want to filter whether shown contributions crossed boundaries of the selected org. element or not, so that I can quickly identify them.
 - R.7** As a user, I want to see whether a contribution went to a parent (ancestor) of the selected org. element, so that I can quickly identify them.
 - R.8** As a user, I want to aggregate the contributor in a fish-eye fashion (those closer to me with more detail), so that I see org. elements I am familiar with in greater detail than unfamiliar ones.

1.3.3 Non-functional requirements

- P.1** CMSuite is primarily a technology demonstrator, so the proposed solution does not have to be 'production-ready'. For example, usability concerns are not a priority for either the development process or the end-user experience, as long as any proposal for enhancement outlines how they can be resolved with reasonable effort.
- P.2** Alterations to the current data schema should be avoided. Existing tables and column definitions should remain unchanged whenever possible while the definition of additional tables, columns and views is permitted.
- P.3** The response time of the accounting module should stay within one to five seconds.
- P.4** The exploration of the historical data should be possible without database or SQL knowledge.
- P.5** The analytics component should not be written from scratch. Instead, existing solutions with a permissive license like the MIT or Apache license

⁴Also referred to as the *direction* of a contribution.

should be used wherever possible and reasonable.

1.3.4 Evaluation scheme

The mentioned requirements are evaluated using the following principles.

- A user story is considered to be fully implemented when the acceptance criteria are met (listed as part of the conclusion).
- The total number of the implemented user stories are put into relation of the total number of stories collected (realization rate).

2 Architecture and design

Prior to this thesis, the CMSuite project consisted of a database schema designed to hold historical data extracted from version control systems using a crawler mechanism. Additionally, it was possible to define metrics (singular value results and tuple based time series) using Pentaho Kettle. A simple CRUD¹ based persistence layer combined with a REST² service allowed for the creation and ID-based querying of most entity types. There was a simple frontend based on the Angular 2+ web application platform consisting mainly out of a metrics dashboard (see Daeubler (2017)). The frontend also contained a stub for the accounting module, listing contributions in a table. Navigation of the organizational hierarchy was implemented only partially, i.e. organizational dimensions were ignored. Little of the existing code of the accountancy module was re-used for this thesis. The implementation [chapter 3](#) will elaborate why.

The fundamental design question is how the information required can be extracted from the existing database. Most user stories listed in [section 1.3](#) suggest a list or table-style visualization. The remaining stories are defining means of aggregation and filtering - typical analytical use cases. Multiple possible alternatives are discussed in the following section.

For better readability, database related terms are formatted [LIKE THIS](#).

2.1 Design alternatives

The following sections discuss the design alternatives for realizing the collected requirements.

¹create, read, update, and delete

²REpresentational State Transfer (REST) web services provide interoperability between computer systems on the Internet (Wikipedia, [2018b](#))

2.1.1 Native SQL or HQL

One obvious solution for exploring data stored in a relational database is using SQL. While the existing requirements do not permit the use of SQL specified during runtime, it seems feasible to use SQL queries during design-time which then have to be parameterized. There are, however, a number of arguments against this approach.

The current database schema is normalized to reduce duplication and ensure referential integrity. This has the effect that `INSERTs`, `UPDATEs` and `DELETEs` are usually simple. On the other hand `SELECT` queries become more complex as they have to span multiple tables using `JOINS`. Additionally, the structure of an organization is stored as a finite graph using parent/child IDs. As an organization is possibly structured in more than one way, for example i.e. in the case of a matrix organization, multiples of the aforementioned finite graphs exist. These points combined result in a difficult to query database schema that is difficult to query. These issues are further elaborated in the implementation section (see [section 3.1](#)).

As the CMSuite persistence layer is based on Hibernate, the named query feature could be used to execute and fetch the result from the database. Usually named queries are stored using annotations on the database entities itself (see [listing 2.1](#) for an example).

Listing 2.1: JPA `@NamedQuery` example

```
1 // package/imports omitted
2
3 @NamedQueries({
4     @NamedQuery(
5         name = "findAllAgentsOrderedByName",
6         query = "SELECT DISTINCT a FROM Agent AS a ORDER BY a.name")
7 })
8 @Entity
9 public class Agent {
10     // class body omitted
11 }
```

The use of named queries has the disadvantage that the application would have to be re-compiled to change queries. While the Java Persistence API (JPA) 2.1+ added support for named queries defined during runtime, this leaves the problem of where to store them. One option would be to store them in the database itself, but this solution would make editing more difficult. Storing them within XML files which are loaded during runtime would be preferable, however this would

require additional effort for specifying a XML schema and the logic to read them. Additionally, such a solution might be limited in terms of scalability and usability.

2.1.2 Reuse Pentaho Data Integration

Daeubler (2017) introduced Pentaho Data Integration ETL (previously named Kettle) to the CMSuite project as part of his work of creating a metrics dashboard. Re-evaluation for the use-cases of this thesis revealed a number of problems, the most significant being the batch-oriented processing of Pentaho. As the full name implies, it is a ETL³ engine which works based on a three-step approach. First the required data is extracted from a source (in this case the CMSuite database), then it is transformed by the Pentaho engine (i.e. aggregates are built). Finally, the result is loaded back into the database. During the ETL process, Pentaho has to execute multiple queries for each transformation, each of which increases the response time. While the ETL approach is valid for computing metrics and time series⁴, it does not work well with data that is logically segregated (by the org. dimensions). Pre-computing all possible combinations would be possible in theory, but hardly justifiable given the expected CMSuite database size.

Pentaho Data Integration stores transformations as XML files using a proprietary schema. They are designed to be edited using the supplied GUI application called ‘Spoon’. Manual editing of the XML files using a text editor is not feasible. Relying on an external GUI application for making changes on the other side is also adding complexity.

2.1.3 OLAP approach

A recurring requirement of the many of the user stories is the ability to both query the contributions database and aggregate it on different levels of detail. Most prominently, the stories require the data to be aggregated by time, receiver and contributor. Together, these requirements would be best served by a data structure like a data cube. The online analytical processing (OLAP) domain, for example, deals with this problem.

Often OLAP systems are compared with online transaction processing (OLTP) systems. OLTP systems are a popular choice for operational data processing, where their transactional implementation provides guarantees for data consistency (ACID⁵). OLTP systems are optimized for concurrent [INSERT](#), [UPDATE](#) sce-

³Extract, transform, load.

⁴This was the problem Daeubler was solving.

⁵Atomicity, Consistency, Isolation, Durability.

narios usually involving negligible data volumes per transaction. OLAP systems however are optimized for READ, covering considerable amounts of data. There are different perceptions on whether a transaction in the OLTP realm is limited to a technical level (database transaction) or on a business level (business transaction). Regardless of which one is true, the comparison becomes increasingly irrelevant when distributed systems are relying on eventual consistency and sensor data from the Internet of Things (IoT) sphere being fed into OLAP systems.

OLAP systems are typically batch-oriented (ETL principle). Data is read from operation databases in intervals, transformed to optimize it for analytical queries, and then stored again. This has the disadvantage that results are not available until the transformation is finished. Additionally, the data is duplicated within the operational systems and the OLAP system. Lastly, the data is also stored using at least two different schemata, one for the operational system and another one for the OLAP system. In practice, the relation is closer to $m:1$ because there are usually many operational systems within an organization that converge their data into a single OLAP system⁶.

In recent years, OLAP engines are evolving towards real-time data processing. The aim is to close the gap between operational and analytical systems. For example, Apache Kylin, an analytics engine, supports real-time data streams provided by a stream processing software such as Apache Kafka.

Nevertheless, there are arguments against the integration of a full-blown OLAP system into the CMSuite project. CMSuite is a technology demonstrator and still in early development which means the architecture is likely evolving. Changes to the persistence layer would almost always require the ETL jobs to be migrated. This migration is already an issue today as the patch crawler and the Pentaho integration depend on the database schema. More importantly, the current data sets are small and do not justify the deployment of an OLAP system. It is also unclear how the analytics use cases will continue to evolve, making it difficult to determine whether an OLAP system will be a good fit in the wake of these future changes. The additional effort of maintaining an OLAP data schema and the learning curve required to deploy and operate it would strain the already limited development resources.

This thesis outlines an alternative solution to a full-blown OLAP system. The implementation discussed in this thesis is based on Apache Lens as it fulfills the requirements and provides additional short and long-term benefits for the CMSuite project's scalability, migration and learning curve. The following chapter will explain why Apache Lens was selected and how it is used.

⁶Of course it is also feasible that an organization operates multiple OLAP systems.

2.2 Apache Lens

Apache Lens acts as a ‘Unified Analytics Interface’; it provides an abstraction layer to bridge the (transitional) gap between operational live data and ETL processing. As such, it does not aim to provide OLAP functionality itself, but rather provides integration with common components used in data warehouse (DWH) systems.

Apache Lens provides a metadata layer that acts as an abstraction for the underlying execution engines and data sources. Lens can be queried using an SQL dialect. Lens decides how to answer the query based on configuration and data availability. A data source can be everything from a CSV file, a connection provided using JDBC⁷, or any data storage supported by Apache Hive.

The metadata used to answer Lens queries is stored using the Apache Hive⁸ metastore. In this case, the metadata are the table definitions consisting of columns names, data types, and comments additional to the data source. As shown in [figure 2.1](#), Apache Lens consists of the following key components:

- The Lens clients, i.e. the Java client or the command-line interface (CLI). The clients communicate with the Lens server using a REST API.
- The Lens server, providing the mentioned REST service allowing to define OLAP cubes (which are stored in the Hive metastore) and query them using CubeQL.
- Lens internally provides a driver API and a set of implementations for common execution engines, i.e. Hive Map Reduce or JDBC.

Lens itself is very lightweight as it builds on existing solutions with a proven track record inside the data warehousing domain. Hive itself is the part of a DWH ecosystem. Many widely used components like the cluster computing framework Spark and the data analyzation tool Pig are compatible with Hive. Also, a component called Hiveserver2 can be used to visualize the metadata using tools like Tableau or PowerBi.

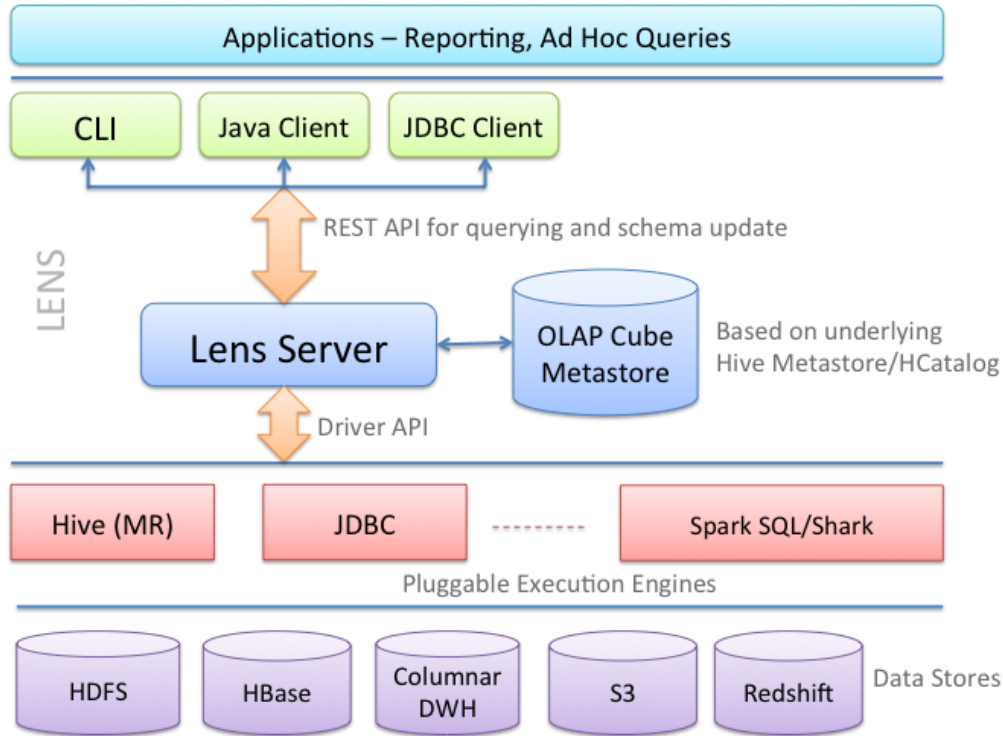
Why Apache Lens is used

Common OLAP solutions are designed to operate based on de-normalized data schemas like the star schema or the snowflake schema. Data has to be extracted

⁷Java Database Connectivity.

⁸Apache Hive is a data warehouse software by itself, however, Lens mostly makes use of its metastore component.

Figure 2.1: Apache Lens Architecture (Apache Lens project, 2017)



from operational systems and transformed to fit the OLAP schema. This process is known as ETL. Apache Lens, however, allows the definition of a schema to specify OLAP metadata and at the same time use JDBC data sources and therefore RDBMS⁹ instead of a data store. As a result, ETL becomes optional. In a case where there are no precomputed aggregates available, Lens will fallback to the live database.

Lens distinguishes between database tables on a conceptional and logical level.

Conceptional tables are a set of fields¹⁰. A field can either be a simple attribute (i.e. referring to a database column), an expression like *NOW()+10 days* or an aggregate measure like *SUM()* or *AVG()*. A conceptional table also can have one or more JOIN chains, whereas a join chain connecting two conceptional tables by specifying a list of join fields i.e. *table1.field1=table2.field2*. Two different types of conceptional tables exist, dimension and cube. A cube can contain all field types mentioned above including join chains, while a dimension table cannot contain measures.

⁹Relational database management systems.

¹⁰A field consists of a name, a type and optionally a comment.

On a higher level, logical tables consist of columns that are referring to fields of the underlying conceptional table. A column, however, only consists of a name and a data type; the distinction between the different fields is discarded. There are also two different types of logical tables, dimension tables, and fact tables. Fact tables contain business facts – in case of CMSuite, the list of contributions. Dimension tables contain descriptive information used to constrain a query result, so its columns are typically used as part of the `WHERE` condition. Fact data is typically aggregated by time (i.e. by year, month, ...) and contains columns used to join the actual dimension tables. For further information refer to the official Apache Lens website <https://lens.apache.org/>.

After defining the OLAP data cubes, they can be queried using CubeQL (see [listing 2.2](#)), which is a subset of HiveQL.

```

1  [CUBE] SELECT [DISTINCT] select_expr , select_expr , ...
2  FROM cube_table_reference
3  [WHERE [where_condition AND] [TIME_RANGE_IN(colName, from , to)]]
4  [GROUP BY col_list]
5  [HAVING having_expr]
6  [ORDER BY colList]
7  [LIMIT number]
8
9  cube_table_reference:
10 cube_table_factor
11 | join_table
12 join_table:
13 cube_table_reference JOIN cube_table_factor [join_condition]
14 | cube_table_reference {LEFT|RIGHT|FULL} [OUTER] JOIN
   cube_table_reference [join_condition]
15 cube_table_factor:
16 cube_name or dimension_name [alias]
17 | ( cube_table_reference )
18 join_condition:
19 ON equality_expression ( AND equality_expression ) *
20 equality_expression:
21 expression = expression
22 colOrder: ( ASC | DESC )
23 colList : colName colOrder? ( ',' colName colOrder? ) *
24
25 TIME_RANGE_IN(colName, from , to) : The time range inclusive of
   'from' and exclusive of 'to'.
26 time range clause is applicable only if cube_table_reference has
   cube_name.
27 Format of the time range is <yyyy-MM-dd-HH:mm:ss,SSS>

```

Listing 2.2: CubeQL grammar

CubeQL is very similar to SQL. Developers familiar with SQL should be able to

understand and write queries based on CubeQL immediately. On account of the Lens abstraction layer, queries become very simple. In the case of CMSuite, no precomputed aggregates exist so Lens will automatically translate the CubeQL queries into a single native SQL query.

One of the main features of Lens is the ability to automatically determine which [JOINS](#) are necessary to compute the result. This is possible because the join chains are declared in the Lens configuration. In a case where there are multiple join chains available, Lens will determine the best chain to use. This is typically the shortest chain or the one with the smallest number of join tables. This also works for bridge tables, join tables holding 1:n relationships (i.e. like PersonLink). Bridge tables can be configured to be transparent to the user, using bridge table flattening. Additionally [GROUP BY](#) clauses are automatically inserted by Lens i.e. in the case a [WHERE](#) clause is referring to a many-column as part of a one-to-many cardinality.

As Lens is able to automatically insert join conditions and [GROUP BY](#) Clauses, they do not have to be specified as part of the CubeQL query. Queries therefore become very concise. Refer to [section 3.1.2](#) for a comparison between a CubeQL query and the PostgreSQL equivalent.

As mentioned before, it is unclear whether the development of the CMSuite project will make greater use of OLAP features or not. Apache Lens is a good fit for such scenarios as it scales very well. For example, precomputed aggregates can be integrated seamlessly without the need to change either the database or the Lens schema. Lens queries by themselves can be used to build the aggregates. Queries can be scheduled to run at fixed intervals to build aggregates and store them (in CSV files for example), improving query response times.

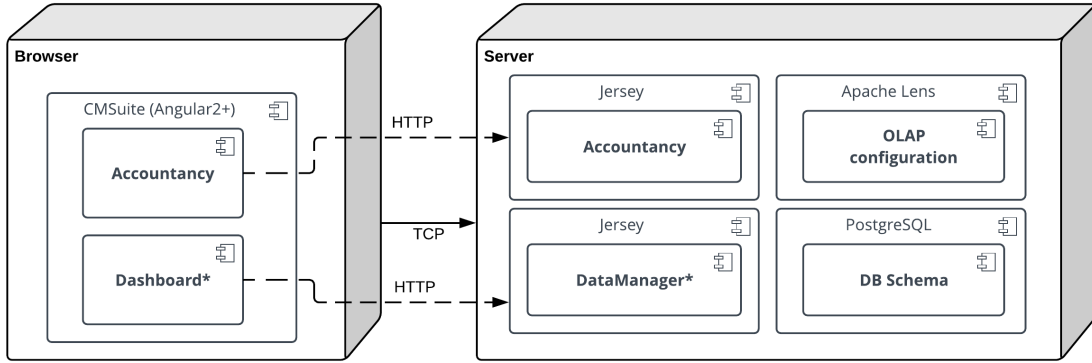
Exit strategy

It is the author's belief introducing an additional dependency to a software project should also include a strategy of how to remove it again. In the case of Apache Lens, this is possible because Lens is generating SQL queries to fetch the data from the live database. This means that Lens can be used a tool to build such SQL queries by running a CubeQL query and observing the SQL query fired against the database. While the resulting queries are complex, it is possible to convert them to named queries (see [listing 2.1](#) (p. 9)) and to parameterize them. This provides flexibility if the CMSuite team decides either that there will not be a need for additional queries or that Lens does not need to run in production environments. Lens could then solely be used to ease development. Of course, this option would still come with the disadvantages as outlined in [section 2.1.1](#).

3 Implementation

The CMSuite software is split into a server and a client part. For an overview of the involved components, refer to [figure 3.1](#).

Figure 3.1: CMSuite deployment diagram¹



The relevant components of CMSuite are:

- An Angular 2+ frontend consisting of an accountancy module (the dashboard module existed prior to this thesis and is only shown for completeness).
- The accountancy services, deployed using a Jersey server which provides the REST functionality.
- The Apache Lens OLAP configuration.
- The views for denormalization of the org. hierarchy have been added to the existing CMSuite database schema.

During the following sections, the components contributed or extended by this thesis are described in a bottom-up order. The elaboration starts at the database layer, continuing with the OLAP configuration, the REST services, and finally the

¹Some components like the PatchCrawler were omitted. Components marked with an asterisk existed prior to this thesis.

Angular user interface. Please note that [figure 3.1](#) omits certain components, such as the crawler mechanism, as they are not relevant for this thesis. Components that existed before (see [section 1.1](#)) like the Dashboard, the Datamanager services, and the Database schema are marked with an asterisk. The subsequent sections will outline whenever the solution built on previous work and to what extent.

Note that none of the previously existing accountancy services were re-used as they were written based on a CRUD approach. Additionally, the REST services were incomplete and many features are broken or stubs. However, some parts of the basic Angular 2+ frontend components could be re-used.

3.1 Database

The CMSuite database schema is difficult to query using SQL for two main reasons:

- It contains finite graphs stored as adjacency lists (parent/child relations).
- It is optimized for [INSERT](#)/[UPDATE](#)/[DELETE](#) consistency. In other words, the schema is normalized.

The following section will elaborate why these points are problematic for analytic use-cases.

There are multiple levels of database normalization, differing in their level of strictness. For example, the first normal form (1NF) states that there should be no columns storing duplicated data. The second normal form (2NF), among other things, moves data that applies to multiple rows to dedicated tables. The CMSuite is using the 3NF where no transitive functional dependencies are permitted between the columns of table.

While normalization provides many benefits like the prevention of insert and deletion anomalies, the request often lacks relativization. As Kimball and Ross (2013, p. 107) state “[...] designers must resist the normalization urges caused by years of operational database designs [...]”. While the statement is originally addressing designers of dimensional databases and OLAP systems, it also applies for general database design, including greenfield projects. Databases should not be normalized without considering the use cases of the software which builds on top.

A set of guide lines for OLAP systems can be derived from Kimball and Ross (2013, p. 107):

- Simplicity should be the primary objective of a database model. A high number of tables will cause users (including developers) to struggle with complexity.
- Normalization of dimension tables should be avoided, as this will make constraints on a dimension table based on dimensional attributes more difficult.
- Numerous tables inevitably lead to a high number of joins, reducing query performance. While joins are typically optimized by the database engine, increasing complexity might lead to the optimizer choosing a poor strategy.
- The space used up by duplicated data is usually insignificant².

²So is the additional effort of maintaining INSERT/UPDATE integrity when those are rare.

Many modern software projects, including CMSuite, are built with the separation of concerns (SoC) principle in mind, where among other things the used data storage mechanism is abstracted using a *persistence layer*³. In theory, this should allow the persistence components to evolve independently of the layers building on top of it. In practice, however, it is very difficult to achieve true separation. Spolsky (2002) states for example that all non-trivial abstractions are “leaky” to some degree, and therefore problematic. Even when the database layer is carefully abstracted, it is likely that certain implications (such as the availability or navigability of result sets) will eventually leak to higher levels.

For this reason, some engineers claim that it might be better to get rid of this abstraction. The underlying question is if a persistence layer abstraction almost inevitably leaks internals, why pursue it at all? The fact is that very few projects will switch persistence mechanisms during their lifetime. Even the ones that do probably are not doing so without extensive effort. A field where the paradigm of eliminating persistence abstraction is more widespread is the microservices domain. Software is structured as fine-grained independent services that are deployed independently and communicate using lightweight mechanisms. While there are various implementations of this pattern conceivable, the most radical do not even share a common database. Instead every service maintains its state in a form that suits its purpose best. Persistence abstractions consequently become irrelevant in such cases⁴.

As previously stated, Apache Lens is a ‘Unified Analytics Interface’. This means that Lens is optimized for data sources providing fact and dimension tables. This is also why Lens does not require an ETL setup; it builds on the assumption that the *extract* and *transform* steps have already happened.

The CMSuite data schema, however, is not designed for analytical purposes. Changing the data schema would break existing code and tools developed. Additionally, Requirement P.2 does not permit extensive schema modifications. As changing the database schema is not a viable option, this thesis uses database views to mitigate some of the problems.

³This is elaborated in more detail in the following [section 3.2.1](#) (p. 35).

⁴Some of the concerns might be shifted to the inter-service communication level, however.

3.1.1 Denormalization

Within CMSuite the hierarchy of an organization is modeled using three different types of elements:

- Organizational units (*orgelement* table)
- Individuals (*person* table)
- Inner source projects (*innersourceproject* table)

Refer to the [figure 2](#) (p. 53) for the relevant schema elements.

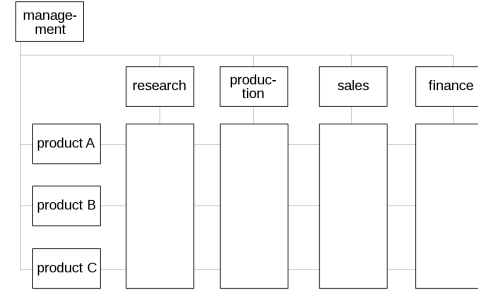


Figure 3.2: Matrix organization scheme (Wikipedia, 2018a)

Together the three element types are used to store a org. hierarchy in form a finite graph (tree). However, an organization is often logically structured in more than one way. Within CMSuite we call each distinct structure an organizational dimension.

For example, a matrix organization has at least two different organizational dimensions with different reporting relationships. [Figure 3.2](#) shows a functional dimension and a product-based dimension. An individual has multiple organizational units it formally belongs to; for example, an engineer can be assigned to the “Research” division while at the same time belonging to “Product Line A”.

The generalizations provided by the REA model ([figure 1.1](#) (p. 2)) can help clarify the reasoning for such structures. In REA terms individuals are classified as *Economic Agents*. Technically, organizational units (such as a “Marketing Department”) and inner source projects are classified as *Economic Units* within REA because they are means to declare and control responsibility but do not act on themselves. Instead, they have to rely on their subordinate individuals. For the sake of simplicity, and for the reason that there are no immediate benefits for CMSuite of splitting them further, this thesis treats all three as Economic Agents.

Treating the org. hierarchy as a tree of economic agents makes it easier to identify common features and reason about differences. The assumption is that all agents have an ID, a name, and potential children agents to form a finite graph. Note that in CMSuite individuals and inner source projects are always leaf nodes as they cannot have further children. As a contribution is inherently created by an individual and the receiver is always an inner source project, contributions affect only leaf nodes directly. Requirement [R.1](#) effectively states, however, that receivers and contributors are to be shown based on the selection of an arbitrary

hierarchy element.

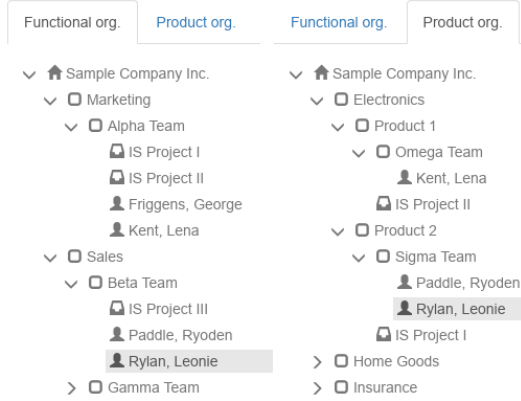


Figure 3.3: Organizational dimensions example

This means that *indirect* contributions need to be determined (see also [figure 3.3](#)). For example, when the user selects “Marketing Department” he wants to see all contributions either contributed by the individuals working for the marketing department or by any project belonging to it.

While the org. hierarchy graph is finite, it is not fixed in depth because an organization can have an arbitrary number of hierarchy levels. The list of agents is stored as an adjacency list, where each row resembles a parent-child relation (link). In case of CM-

Suite the (direct) links are stored using a join table for each agent type. To be able to answer queries about *indirect* reporting links like the mentioned marketing example, the adjacency list needs to be flattened. This flattening is a form of denormalization as some information will now be redundant.

```

1 cmsuite=# SELECT parent_orgelement_id, child_orgelement_id FROM
      orglink WHERE orgdimension_id=2;
2 parent_orgelement_id | child_orgelement_id
3 -----
4 0 | 80
5 0 | 81
6 0 | 82
7 80 | 900
8 80 | 901
9 81 | 902
10 82 | 903
11 900 | 800
12 901 | 801
13 902 | 802
14 902 | 803
15 (11 rows)

```

Listing 3.1: Sample query⁵

In the normalized form it is difficult to determine the indirect descendants for a given parent agent (see [listing 3.1](#)). To be able to query them efficiently using

⁵The following queries have been simplified for demonstration purposes. For example, the org. dimensions have been omitted.

Lens,⁶ this adjacency list has to be denormalized first. In SQL this problem can be solved using Common Table Expressions (CTEs). CTEs are used to realize recursive queries. See [listing 3.2](#) for a simplified query.

```
1 WITH RECURSIVE tree (child_orgelement_id , parents) AS (  
2     SELECT  
3         parent_orgelement_id ,  
4         ARRAY[]::integer []  
5     FROM orglink o  
6     WHERE parent_orgelement_id NOT IN  
7         (SELECT DISTINCT child_orgelement_id FROM orglink)  
8 UNION  
9     SELECT DISTINCT  
10        o.child_orgelement_id ,  
11        parents parent_orgelement_id  
12    FROM orglink o  
13    JOIN tree n ON n.child_orgelement_id = o.parent_orgelement_id  
14 )
```

Listing 3.2: CTE example⁷

The query works like this.

- **WITH RECURSIVE tree (child_orgelement_id , parents)**
Specifies what the result set of the query looks like, in this case it has two output columns, one listing a child ID and another containing all parent IDs.
- **SELECT (line 2).** Selects the initial set on which the recursion is based on. The query selects all root nodes exploiting the fact that they do not have any ancestors themselves.
- **UNION** discards duplicate rows, uses the remaining rows for the recursive expression.
- **SELECT (line 9).** The recursive expression, referring to the queries own output (tree). Concatenating to a list of parents.

The query is operating on a working table (queue) and repeats the recursion as long as its result is not empty. In other words, it completes after the recursive expression does not return any data. Technically, this is an iterative process, not recursion.

[Listing 3.3](#) shows the output of the previously specified CTE; the first column contains every node of the org. element graph, while the second column contains the list of ancestors. CTEs can be materialized as database views. Using a

⁶For example, to use the hierarchy as a dimension table.

⁷The final view for the denormalized org. hierarchy is shown in [appendix C \(p. 54\)](#), [listing 1](#).

materialized view, it is possible to query all direct and indirect successors with low effort ([listing 3.4](#)).

Note that the execution time of views is exactly the same as running the underlying SQL statement⁸.

```
1 cmsuite=# SELECT child_orgelement_id, parents FROM tree;
2 child_orgelement_id | parents
3 -----+-----
4                  0 | {}
5                  80 | {0}
6                  81 | {0}
7                  82 | {0}
8                 900 | {0,80}
9                 901 | {0,80}
10                902 | {0,81}
11                903 | {0,82}
12                800 | {0,80,900}
13                801 | {0,80,901}
14                802 | {0,81,902}
15                803 | {0,81,902}
16 (12 rows)
```

Listing 3.3: Query output using CTE and view

```
1 cmsuite=# SELECT child_orgelement_id, parents FROM tree WHERE 80 =
2 ANY(parents);
3 child_orgelement_id | parents
4 -----+-----
5                 900 | {0,80}
6                 901 | {0,80}
7                 800 | {0,80,900}
8                 801 | {0,80,901}
9 (4 rows)
```

Listing 3.4: Query all successors of a certain parent node

Each node of the org. graph can have further successors. Currently two different descendant node types are defined; ISPs and individuals. These are stored in two separate join tables similar to the orglink table (see [listing 3.5](#)).

⁸In PostgreSQL this can be verified using [EXPLAIN ANALYSE](#).

```

1 cmsuite=# SELECT orgelement_id , innersourceproject_id FROM
           innersourceprojectlink;
2  orgelement_id | innersourceproject_id
3  -----|-----
4             900 |             7001
5             901 |             7000
6             902 |             7002
7             902 |             7003
8             902 |             7009
9 (5 rows)

```

Listing 3.5: Query links between org. element and ISPs

Additional views have been defined to be able to query ISP/individual leaf nodes based on their ancestors. [Listing 3.6](#) shows the flattened hierarchy view for inner source projects. A similar view has been created for individuals.

```

1 SELECT DISTINCT
2     unnest(array_append(parents , child_orgelement_id)) AS
     orgelement_id ,
3     innersourceproject_id AS project_id
4 FROM innersourceprojectlink pl
5 JOIN orghierarchy
6 ON pl.orgelement_id=child_orgelement_id

```

Listing 3.6: ISP hierarchy view

[Listing 3.7](#) shows the result, a denormalized adjacency list containing every direct and indirect link between an ISP and and org. element⁹.

⁹The final version of this query has been extended to explicitly mark indirect links. Refer to [appendix C](#) (p. 54) for complete version.

1	orgelement_id	project_id
2		
3	0	7000
4	0	7001
5	0	7002
6	0	7003
7	0	7009
8	80	7000
9	80	7001
10	81	7002
11	81	7003
12	81	7009
13	900	7001
14	901	7000
15	902	7002
16	902	7003
17	902	7009
18	(15 rows)	

Listing 3.7: Query links between org. element and ISPs using view

With the introduced views the org. element hierarchy can now be easily queried, for example, to get a list of direct or indirect descendants for given agent. By using views instead of modifying the schema directly, all pre-existing code continues to work. The following section will outline how Apache Lens is used to build on top of this.

The metadata extracted by the patch crawler is stored in the “Patch” (contribution) table (see [appendix B](#) (p. 53)). In REA terms contributions are economic events. A contribution links two economic agents. The source of the contribution is called the ‘contributor’ and the sink of the patch is called the ‘receiver’. More specifically the contribution links an individual (contributor) to an inner source project (ISP), the receiver. Both are specific instances of an economic agent. They are also leaf nodes in the org. tree.

Currently, there is no economic resource (REA) modeled. Future extensions to CMSuite could introduce resources like ‘Capacity’ or ‘Cost’, however.

3.1.2 Apache Lens OLAP cubes

The solution proposed by this thesis relies on Apache Lens and its OLAP cube interface to fetch data from the underlying CMSuite database. Most of the effort for integrating Apache Lens into a software project is required for setting up the declarative configuration. This section will describe how Lens is configured to fit CMSuite requirements.

Lens can be configured using its command-line interface (CLI) which is started by executing “lens-client”. The CLI provides various commands which can be displayed by entering “help”. A description for each command including its arguments can be displayed with “help [command]” (see [listing 3.8](#)). Many commands require a *spec* file as an argument, which is an XML containing the actual configuration.

Listing 3.8: Help output for *create cube* command

```
lens-shell>help create cube
Keyword:                create cube
Description:            Create a new Cube, taking
                        spec from <path-to-cube-spec-file>
Keyword:                ** default **
Keyword:                path
Help:                   <path-to-cube-spec-file>
Mandatory:             true
Default if unspecified: '__NULL__'
```

Listing 3.9: CMSuite storage spec configuration

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<x_storage xmlns="uri:lens:cube:0.1" name="cmsuite"
  classname="org.apache.lens.storage.db.DBStorage">
  <properties>
    <property name="lens.storage.db.url"
      value="jdbc:postgresql://cmsuite-postgres-db/cmsuite"/>
  </properties>
</x_storage>
```

A minimal Lens configuration consists of a Lens database with at least one storage¹⁰. For CMSuite this configuration can be achieved by entering the command sequence shown in [listing 3.10](#) and a storage spec like in [listing 3.9](#). The sequence shown creates a new database called *cmsuite* and configures a JDBC storage specified in *cmsuite_storage.xml*.

¹⁰Lens uses the term “storage” for a data source i.e. a JDBC connection.

Listing 3.10: Command sequence to achieve a minimal Lens configuration

```
lens-shell>create database cmsuite  
lens-shell>use cmsuite  
lens-shell>create storage --path cmsuite_storage.xml
```

Apache Lens can be queried using OLAP CubeQL ([listing 2.2](#)), a subset of HiveQL which is based on SQL. CubeQL is designed for analytical purposes and as such is limited to [SELECT](#) queries. After configuring a storage, Lens will try to fall back to it whenever a query cannot be answered by any other means. In other words, CMSuite tables including the introduced views previously described in [section 3.1](#) can already be queried by typing and executing the SQL query within Lens CLI. As long as the queries do not make use of any special keywords or dialect, they do not have to be modified¹¹. For example, the query shown in [listing 3.7](#) will work unchanged. Please note, however, that in Lens CLI queries are not terminated using a semi-colon unlike in an interactive PostgreSQL shell.

¹¹The restriction to [SELECT](#) queries still applies.

Dimensional modeling

The process of creating OLAP data cubes is called dimensional modeling.

Dimensional modeling is widely accepted as the preferred technique for presenting analytic data because it addresses two simultaneous requirements:

- Delivering data that is understandable to the business users.
- Delivering fast query performance.

(Kimball & Ross, [2013](#))

Dimensional modeling treats data either as facts or dimensions. Usually, tables containing facts can easily be identified because they tend to grow by multiple orders of magnitude, compared to dimension tables. The CMSuite *patch* table is a fact table because it collects an ever-growing time series of contribution events. On the other hand, dimension tables are usually of descriptive nature and often represent hierarchical relationships, like the CMSuite org. hierarchy.

Dimensions are modeled by answering the question “how are the facts logically segregated?” or “what are the discriminators of the facts?”.

In CMSuite code contributions (patches) are directly segregated by

- The contribution date.
- The author of the contribution (contributing individual).
- The inner source project the contribution has been submitted to (receiving ISP).

As mentioned before, discriminators are usually part of hierarchy or turn into one when reducing or increasing their level of granularity. For example, the contribution date can be specified as:

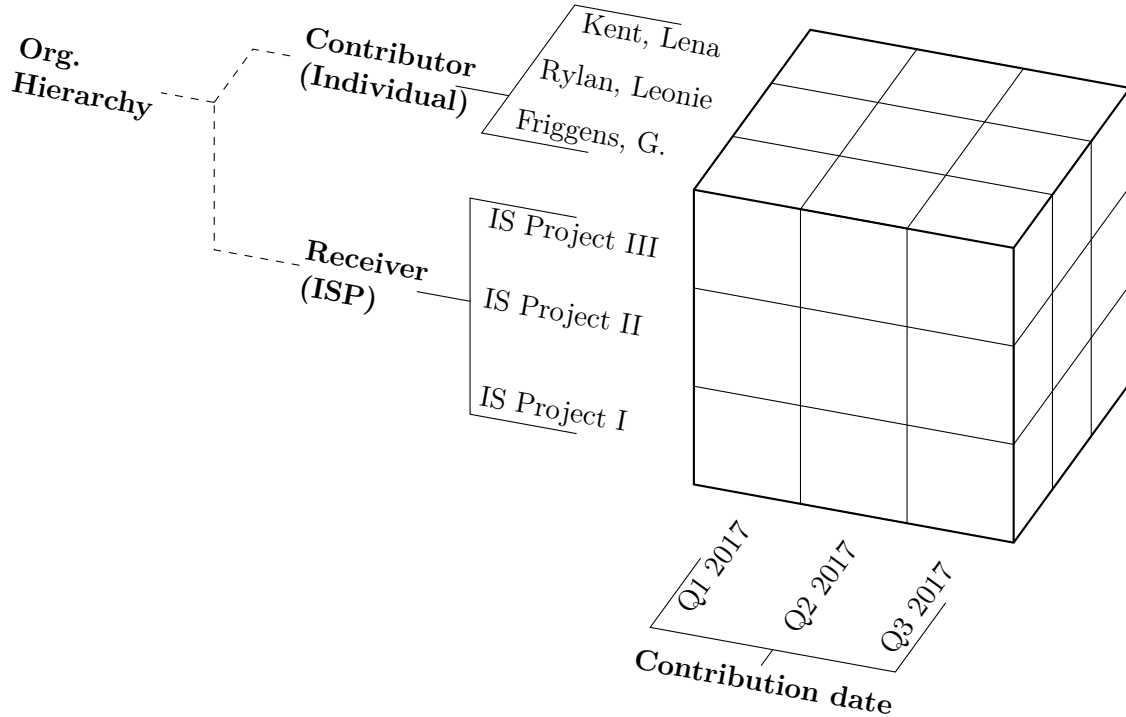
- The entire timestamp, i.e. *2018-01-01-14:36:58,421*
- The date only, *2018-01-01*
- The calendar week, *week 1*
- The month, *January*
- The quarter of the year, *1. Quarter*
- The year only, *2018*
- etc.

The org. element hierarchy has no direct influence on the segregation; it merely structures and groups authors and inner source projects. As such, it only indirectly segregates the patch table. Therefore the org. hierarchy can also be referred to as a hyper-dimension. Based on the previously mentioned observations, the following dimensions have been identified for CMSuite:

- The contributor dimension (author).
- The receiver dimension (inner source project).
- The org. hierarchy (hyperdimension for contributor/receiver).

Together they form the *journal* data cube. Refer to [figure 3.4](#) for a visual representation.

Figure 3.4: CMSuite *journal* cube



Based on the data cube abstraction, a number of operations are possible:

- The OLAP user can constrain the values of each dimension to restrict the returned dataset, which is called “dicing”.
- When the value for one dimension is restricted to a single value, effectively reducing the cube to a table, the operation is called “slicing”.
- The navigation of the receiver/contributor hierarchy is called “drill down/up”, depending on whether the user traverses from the entire organization down

to org. units or vice-versa.

- The fact dimension (labeled “Contribution date” in [figure 3.4](#)) is often aggregated (summarized), i.e. by year, which is called “roll-up”.

[Listing 3.11](#) shows the configuration for the *contributor* dimension (conceptional level). Note that despite the fact that Lens relies on XML, the spec files are well readable and can be edited manually using any XML editor, unlike Pentaho configurations. As mentioned in [section 2.2](#), the Lens configuration knows two different abstraction levels, each conceptional configuration (such as the contributor dimension) has at least one logical configuration. [Listing 3.12](#) shows the logical table of the contributor dimension. Note that join chains ([listing 3.11](#), [line 11](#)) are specified at the conceptional level whereas the storage configuration ([listing 3.12](#), [line 14](#)) takes place at the logical level.

Listing 3.11: CMSuite contributor dimension specification

```
1 <x_dimension name="contributor" xmlns="uri:lens:cube:0.1"
2 /* omitted */>
3   <attributes>
4     <dim_attribute name="orgelement_id" _type="INT"/>
5     <dim_attribute name="individual_id" _type="INT"/>
6     <dim_attribute name="direct" _type="BOOLEAN"/>
7     <dim_attribute name="contributor_depth" _type="INT"/>
8     <dim_attribute name="dimension_id" _type="INT"/>
9   </attributes>
10
11   <join_chains>
12     <join_chain name="journal_chain">
13       <paths>
14         <path>
15           <edges>
16             <edge>
17               <from table="contributor" column="individual_id" />
18               <to table="journal" column="individual_id" />
19             </edge>
20           </edges>
21         </path>
22       /* contributor/orgelement path omitted */
23     </paths>
24   </join_chain>
25 </join_chains>
26 </x_dimension>
```

Listing 3.12: CMSuite contributor dimension table specification

```
1 <x_dimension_table dimension_name="contributor"
   table_name="contributor_table" /* omitted */>
2
3 <columns>
4   <column name="orgelement_id" _type="INT"/>
5   <column name="individual_id" _type="INT"/>
6   <column name="direct" _type="BOOLEAN"/>
7   <column name="contributor_depth" comment="Hierarchy_level_of_
   contributor" _type="INT"/>
8 </columns>
9
10 <properties>
11   <property name="dimtable.contributor_table.part.cols"
   value="dimension_id"/>
12 </properties>
13
14 <storage_tables>
15   <storage_table>
16     <storage_name>cmsuite</storage_name>
17     <table_desc external="true" /* omitted */>
18       <part_cols>
19         <column comment="Org._Dimension"
   name="dimension_id" _type="STRING"/>
20       </part_cols>
21       <table_parameters>
22         <property name="lens.metastore.native.db.name"
   value="public"/>
23         <property name="lens.metastore.native.table.name"
   value="contributor"/>
24         <property
   name="lens.metastore.native.table.column.mapping"
   value="contributor_depth=depth"/>
25       </table_parameters>
26     </table_desc>
27   </storage_table>
28 </storage_tables>
29
30 </x_dimension_table>
```

The receiver dimension is similarly configured.

Listing 3.13: CMSuite contributor fact table specification

```
1 <x_fact_table cube_name="journal" name="contribution_fact"
2 /* omitted */>
3
4 <columns>
5   <column name="id" comment="ID" _type="INT"/>
6   <column name="commit_date" _type="TIMESTAMP"/>
7   <column name="commit_message" _type="STRING"/>
8   <column name="individual_id" _type="INT"/>
9   <column name="receiving_innersourceproject_id" _type="INT"/>
10 </columns>
11
12 <properties>
13   <property name="cube.fact.is.aggregated" value="false"/>
14   <property name="dimension.journal.timed.dimension" value="ct"/>
15 </properties>
16
17 <storage_tables>
18   <storage_table>
19     <update_periods>
20       <update_period>CONTINUOUS</update_period>
21     </update_periods>
22     <storage_name>cmsuite</storage_name>
23     <table_desc external="true" /* omitted */ >
24       <part_cols>
25         <column comment="Commit_time_partition"
26           name="ct" _type="STRING"/>
27       </part_cols>
28       <table_parameters>
29         <property name="lens.metastore.native.db.name"
30           value="public"/>
31         <property name="lens.metastore.native.table.name"
32           value="patch"/>
33         <property
34           name="lens.metastore.native.table.column.mapping"
35           value="individual_id=author_id"/>
36       </table_parameters>
37       <time_part_cols>ct</time_part_cols>
38     </table_desc>
39   </storage_table>
40 </storage_tables>
41 </x_fact_table>
```

As the determination of the “value” of a contribution is out of the scope of this thesis (see debt/credit discussion [section 1.1.2](#)), the number of contributions is used as a method to quantify them.

Writing CubeQL queries requires no learning effort from developers already familiar with SQL other than using `TIME_RANGE_IN()` for constraining the time dimension.

```

1 CUBE SELECT c_quarter AS period ,
2     receiver_details.orgelement_id AS receiver_orgelement_agent ,
3     contributor_details.orgelement_id AS
4     contributor_orgelement_agent ,
5     project_details.id AS project_agent ,
6     COUNT(*) AS amount
7 FROM journal
8 WHERE TIME_RANGE_IN(commit_date, '2017-01-01', '2018-01-01')
9     AND receiver_details.dimension_id = 2
10    AND contributor_details.dimension_id =
11    receiver_details.dimension_id
12    AND individual_details.id = 302
13    AND receiver_details.direct = true
14    AND contributor_details.direct = true

```

Listing 3.14: CubeQL sample query¹²

When comparing a Lens CubeQL query ([listing 3.14](#)) to its PostgreSQL equivalent ([listing 3.16](#)), it becomes clear Lens queries are much shorter and easier to understand. Also, note that the PostgreSQL query already makes use of the introduced database views, so the actual native query is about twice as long as shown. Both produce the same output as shown in [listing 3.15](#).

```

1 period      receiver_orgelement_agent      contributor_orgelement_agent
2      project_agent      amount
3
4 4         900         801         7001         11
5 4         901         801         7000         5
6 4         902         801         7002         23
7 3 rows processed in (0) seconds.

```

Listing 3.15: Lens shell output for [listing 3.14](#)

¹²Note that the CUBE keyword in [listing 3.14](#) (line 1) is optional.

```

1 SELECT CAST(EXTRACT(quarter FROM date(contribution.commit_date)) AS
   int) AS period ,
2     receiver_details.orgelement_id AS receiver_orgelement_agent ,
3     contributor_details.orgelement_id AS
   contributor_orgelement_agent ,
4     project_details.id AS project_agent ,
5     COUNT(*) AS amount
6 FROM patch contribution
7 INNER JOIN (SELECT project_id , orgelement_id , dimension_id FROM
   receiver) receiver_details
8     ON (contribution.receiving_innersourceproject_id =
   receiver_details.project_id)
9 INNER JOIN (SELECT id FROM person) individual
10    ON (contribution.author_id = individual.id)
11 INNER JOIN (SELECT individual_id , orgelement_id , direct , dimension_id
   FROM contributor) contributor_details
12    ON (contribution.author_id =
   contributor_details.individual_id)
13 INNER JOIN (SELECT id FROM innersourceproject) project_details
14    ON (contribution.receiving_innersourceproject_id =
   project_details.id)
15 WHERE (contribution.commit_date BETWEEN '2017-01-01_00:00:00 ' AND
   '2018-01-01_00:00:00 ')
16 AND receiver_details.dimension_id = 2
17 AND contributor_details.dimension_id =
   receiver_details.dimension_id)
18 AND individual.id = 302
19 AND receiver_details.direct = true
20 AND contributor_details.direct = true
21 GROUP BY CAST(EXTRACT(quarter FROM
   DATE((contribution.commit_date))) AS int) ,
22     receiver_details.orgelement_id ,
23     contributor_details.orgelement_id ,
24     project_details.id

```

Listing 3.16: PostgreSQL equivalent to [listing 3.14](#)

Running OLAP CubeQL from Java

Running a OLAP CubeQL query is simple. See [listing 3.17](#) for an example. The *result* will contain the Lens Query result set consisting of columns and metadata like the column names.

Listing 3.17: Running a lens CubeQL query from Java

```
1 LensClientConfig clientConfig = new LensClientConfig();
2 clientConfig.setLensDatabase("cmsuite");
3 LensClient lensClient = new LensClient(clientConfig);
4
5 String query = "SELECT ...";
6 LensStatement statement = lensClient.getStatement();
7 QueryHandleWithResultSet result = statement.executeQuery(query,
    name, QUERY_TIMEOUT);
```

3.2 REST services

3.2.1 Data serialization

Currently, the CMSuite project is using the data access object (DAO) pattern to provide a layer of abstraction from the used persistence mechanism. For example, an AgentService would provide a getAll() method, returning all known agents. This way the consumer of the AgentService does not need to know how the AgentService is storing the data or where it is getting it from. In the case of CMSuite the callers of AgentService do not know it is relying on Hibernate ORM for storing and loading data from a PostgreSQL database. This works especially well in CRUD scenarios where the service API always returns a model instance (i.e. one or more domain model instances¹³ of type “Agent”). As soon as queries have analytical aspects and involve aggregation, i.e. in form of MAX(), COUNT() in conjunction with GROUP, this pattern breaks. The service can no longer return an instance of a domain model as its interface does not fit the query. This leads to another pattern which is also used in the CMSuite project – the data transfer object (DTO) pattern. The DTO pattern introduces dedicated classes for each distinctive type to be transferred using REST. DTOs are also often used to return composite data which would otherwise require multiple calls to a service. Additionally, DTOs can be used to hide information when not all fields in a domain model instance should be leaked to the user of the DAO. For example, when returning an instance of a “User”, a DTO could contain its name and email from the domain model, but not the hash of the user’s password.

The main reasons for using DTOs can be summarized as:

1. Information hiding

¹³Domain model classes are designed to persist data, i.e. in a database.

-
- Hide the used persistence mechanism (Hibernate ORM) and the database schema structure.
 - Hide specific information of the domain model (i.e. password hashes).
2. Return data composed from different domain model instances.
 - For example, return a “UserInfo” DTO containing both an address and email, although they are stored in two different domain objects in the database.
 3. Provide different level of details.
 - For example, when returning a “Customer” DTO, the response could contain the number of open orders. If the number is zero, the caller of the service would not have to fetch the “CustomerWithOrders” DTO. This is also used for performance optimization when returning all orders is a costly call.

The main disadvantage of using the DTO pattern is that the created DTOs are leading to duplicated code. For each entity on the persistence layer, there has to be at least one DTO. In practice there is more than just one DTO; one for the entity and one for each distinct query involving aggregation or composition. Additionally, there might be more DTOs in case not all fields of the entity are desired, especially where it would be too costly to transfer all of them.

For analytic scenarios DTOs are not just duplicated code, they are also wasteful because they are used for unidirectional `SELECT` operations only, not CRUD. DTOs are designed to serialize results provided by a service, i.e. to transfer data using REST. DTOs are most useful for bidirectional communication, where they aid the deserialization¹⁴ of received data while also providing isolation from the persistence mechanism. This is the case when DTOs are modified by the caller and then passed back to the DAO layer for updating the domain model.

Java 7 in conjunction with JAX-RS¹⁵ can make DTOs obsolete, especially when combined with REST. REST services can directly return domain model instances or any other Java object which are serialized (and deserialized) transparently.

The proposed solution in this thesis does not involve using DTOs. Instead, custom serializers and *mix-ins* are used. The following sections describe how they work and the benefits they provide.

¹⁴Deserialization is sometimes also called unmarshalling.

¹⁵Java API for RESTful Web Services (JAX-RS).

Serialization of Lens queries

CMSuite previously provided a module called “datamanager”, which supports CRUD use cases based on model entities using both the DAO and DTO pattern. The module covered by this thesis is called “accountancy”, which addresses analytical scenarios. The distinctive use cases of CRUD vs. analytics (read) are visible already at an artifact level.

As has been established, DTOs are not a good fit for unidirectional (read-only) use cases such as REST endpoints solely returning analytical data. Instead the data can be serialized using other means, i.e. by using a serializer directly.

For analytics use cases, none of the mentioned reasons for using DTOs is a concern because:

1. Information hiding is happening explicitly as only columns to be returned to the caller are queried.
2. Composition is an inherent feature of SQL ([SELECT](#)/[JOIN](#)). The result set is therefore almost always a composite.
3. The same is true for information hiding in the form of aggregates (i.e. using [SUM\(\)](#) or [GROUP BY](#)).

The accountancy module mostly displays tables along with options to aggregate and filter the data displayed. As such, the data returned from the underlying REST service is very uniform. The service endpoints respond with a number of rows extracted from the database. Introducing DTOs to transform the database results would not provide additional benefits. Instead the accountancy REST endpoints can directly return the database result. A generic Lens query serializer implemented using Jackson is ensuring a standard format for the JSON output, acting as a layer of abstraction. [Listing 3.18](#) shows an example JSON string returned by the Journal REST endpoint. It consists of three main elements:

- The *type* property, providing the means to distinguish different result formats on the client side.
- The *rows* property, containing the result row data from the Lens query.
- The *meta* property, describing the columns configuration of the *rows* result set including the queried columns and their data type.

Using this technique, any Lens query can be serialized in a standardized form and query changes are reflected automatically. This approach provides all the benefits of a DTO approach, i.e. the abstraction from the data source, while not involving any of the disadvantages such as duplicated code. Additionally, it is possible to replace Apache Lens with a different product without breaking

the compatibility to the client side, by simply maintaining the same serialization format.

Listing 3.18: Uniform query result example

```
1 {
2   "type" : "de.fau.cmsuite.accountancy.QueryResult",
3   "rows" : [
4     /* row data omitted */
5   ],
6   "meta" : [ {
7     "name" : "period",
8     "type" : "INT"
9   }, {
10    "name" : "receiver_orgelement_agent",
11    "type" : "INT"
12  }, {
13    "name" : "contributor_orgelement_agent",
14    "type" : "INT"
15  }, {
16    "name" : "project_agent",
17    "type" : "INT"
18  }, {
19    "name" : "direct",
20    "type" : "TINYINT"
21  }, {
22    "name" : "amount",
23    "type" : "BIGINT"
24  } ]
```

This approach works well for uni-directional serialization of uniform data. However, there is a supplementary solution for bi-directional use cases when a REST service is also used to update or save data.

Serialization of org. element hierarchy

Prior to this thesis, the *OrgElementResource* provided by the *datamanager* component would be used to return the serialized JSON org. element hierarchy. This hierarchy was used by the Angular UI to render a navigation tree.

However, the existing implementation had no support for dimensions; the output would contain elements from all dimensions. Also, it was using a *OrgElementDto*

containing links to its parent and children elements, members and inner source projects. Each link was stored in an individual LinkDto, i.e. the PersonLinkDto, which then referenced the instance it linked to, i.e a PersonDto.

There are multiple problems with this implementation:

- It provides no abstraction from the database schema as it closely resembles the entity structure, including the join tables. Structural database changes would almost certainly require changes in the DTO structure.
- There is no information hiding, all database columns are leaking into the DTOs.
- The DTO structure uses deep nesting. For example, an OrgElement would contain numerous LinkDTOs, which would then contain a number of PersonDTOs and so on.
- The nested DTO structure lead to nested JSON, which was then difficult to work with on the client side.
- Because of the nested DTOs, information would be serialized redundantly. Each root node of the org. element hierarchy would be serialized including all its ancestors. The same serialization, however, would be repeated for each ancestor node.

As mentioned in [section 1.1.1](#), the implementation of this thesis makes use of the generalization the REA model provides. Instead of using DTOs, the re-implementation uses Jackson mix-ins. Jackson serialization is usually customized by adding Jackson annotation to the class to be serialized. [Listing 3.19](#) shows a usage example where a property is renamed ([line 2](#)) and another property is ignored ([line 5](#)) during serialization. While this approach is flexible, it has the disadvantage of polluting the code with serialization information¹⁶.

Listing 3.19: Jackson annotation example (Github, 2018)

```
1 public class Name {  
2     @JsonProperty("firstName")  
3     public String _first_name;  
4  
5     @JsonIgnore  
6     public int internalValue;  
7 }
```

Starting with version 1.2 Jackson provides a feature called “Mix-in Annotations” which addresses this issue. Instead of having to annotate the class to be serialized,

¹⁶Technically the annotations also conflict with the separation of concerns principle.

a separate mix-in class is used to describe how a class (target) is to be serialized and deserialized.

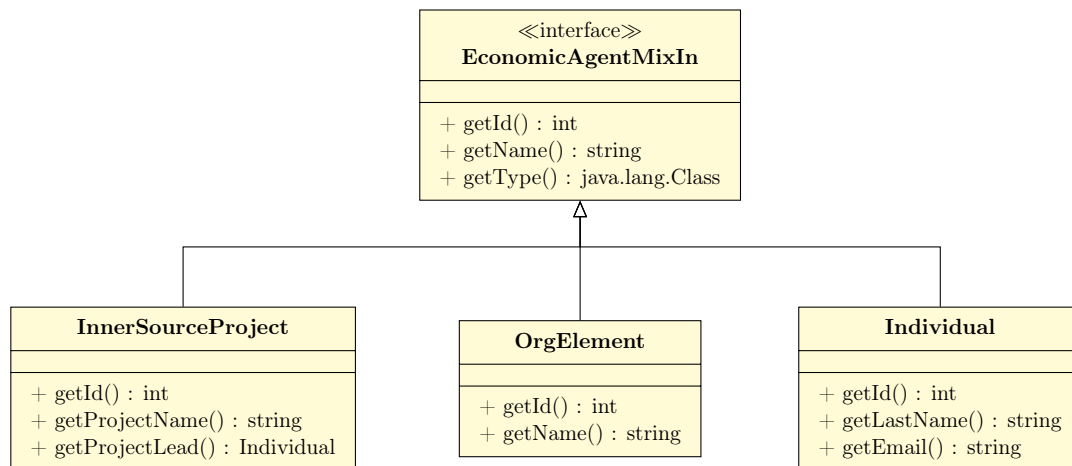
The mix-in class does not have to be a concrete class; it can be an abstract class or an interface instead. Additionally, the mix-in target does not have to extend or implement the mix-in. This is especially useful for serializing third-party classes that cannot be changed easily. However, it is still recommended that the target class extends or implements the mix-in, as this adds the benefits of compile-time checks (`@Override`) to ensure the mix-in does actually fit the target (FasterXML Wiki, 2018).

The benefits of using Jackson mix-ins are:

- Elimination of DTOs, instead mix-ins take their place. This leads to a substantial code reduction, as mix-ins can be abstract.
- Multiple mix-ins for the same target class can exist, and one mix-in can be used for multiple targets. This provides flexibility and eases information hiding.

For CMSuite, a `EconomicAgentMixIn` has been defined for the purpose of serializing the org. element hierarchy in a homogenous way. See figure 3.5 for an example class diagram. Similarly, a `EconomicReferenceMixIn` is used to serialize the different link instances. Using this approach, the finite graph is serialized as a flat list of JSON objects.

Figure 3.5: Jackson mix-in example class diagram



3.2.2 Endpoints

The URIs defined for the journal resource are shown in figure 3.6. All path parameters marked wrapped with curly braces are mandatory. The HTTP method

used for all following resources is GET. All endpoints return data in the JSON format.

Figure 3.6: URIs of journal REST resource

URI	Description
/journal/{dimensionId}/orgelementtypes	Returns all known org. element types i.e. <i>Department</i> , <i>Team</i> , etc.
/journal/{dimensionId}/orgelement/{orgelementId}	Returns the journal for the given org. element.
/journal/{dimensionId}/project/{projectId}	Returns the journal for the given ISP.
/journal/{dimensionId}/individual/{individualId}	Returns the journal for the given individual.

Additionally, each of the journals listed in [figure 3.6](#) can be further parameterized using query parameters. The query parameters and their valid values are listed in [figure 3.7](#).

The URIs defined for the agent resource are shown in [figure 3.8](#).

Figure 3.7: Query parameters for journal REST resource endpoints

URI	Type	Format/Values	Description
timeRangeFrom	string	yyyy-MM-dd-HH:mm:ss,SSS	Start of the time range.
timeRangeUntil	string	yyyy-MM-dd-HH:mm:ss,SSS	End of the time range.
timeAggregation	string	disabled, day, weekofyear, month, year, quarter, overall	Sets the aggregation of the time dimension.
receiverAncestors	int	0=indirect, 1=direct, -1=both	Whether direct or indirect receivers should be included.
receiverLevel	int	positive integer or -1=match selection	The receiver level to show.
contributorAncestors	int	0=indirect, 1=direct, -1=both	Whether direct or indirect contributors should be included.
contributorLevel	int	positive integer or -1=match selection	The contributor level to show.

Figure 3.8: URIs of agent REST resource

Parameter	Description
/agent/{dimensionId}/supported	Returns list of supported agents, i.e. <i>orgelement</i> , <i>project</i> , <i>individual</i> .
/agent/{dimensionId}/all	Returns all agents for the given dimension.
/agent/{dimensionId}/orgelement	Returns all org. elements for the given dimension.
/agent/{dimensionId}/project	Returns all ISPs for the given dimension.
/agent/{dimensionId}/individual	Returns all individuals for the given dimension.

3.3 Angular frontend

Figure 3.9: Screenshot of CMSuite Accountancy user interface¹⁷

The screenshot displays the CMSuite Accountancy user interface. On the left is a sidebar with a tree view of the organization hierarchy. The main area is divided into two sections: 'Analysis Parameters' on the left and a table titled 'Economic Events Journal - Alpha Team' on the right.

Analysis Parameters:

- Time range:** 01/01/2017 (format: mm/dd/yyyy)
- Time Aggregation:** per quarter
- Contributors:** direct & indirect, match selection
- Receivers:** direct, match selection
- Apply** button

Economic Events Journal - Alpha Team Table:

Period	Contributor	Receiver	Contributions	Project
4	Alpha Team	Sales	8	IS Project X
4	Sample Company Inc.	Alpha Team	81	IS Project I
4	Sample Company Inc.	Alpha Team	43	IS Project II
4	Marketing	Alpha Team	34	IS Project I
4	Marketing	Alpha Team	32	IS Project II
4	Sales	Alpha Team	47	IS Project I
4	Sales	Alpha Team	11	IS Project II
4	Alpha Team	Alpha Team	34	IS Project I
4	Alpha Team	Alpha Team	32	IS Project II
4	Beta Team	Alpha Team	19	IS Project I

Navigation: « Previous 1 2 Next »

Handling of economic agents

As mentioned in the previous section, the REST service that provides the EconomicAgents has been re-written to return a flat list of agents and their child-references instead of returning a tree. It is easier to find an agent by its ID in a list than a tree. The agent list is not only used for displaying the org. hierarchy but also to link result entries within the journals. A user can jump to the journal of a giving agent by clicking on its name within the journal.

Additionally, as the agent hierarchy now consists of generic agents instead of concrete instances of individuals, org. elements and ISPs there is less code needed to deal with the different agent types. The previously implemented navigation used classes for the individual subtypes, which is not necessary.

The generic agent handling also provides the benefit of being able to introduce new agent types without having to change the UI. The only thing left to do when adding a new agent type would be defining the icon to be displayed.

Previously org. elements would be serialized redundantly, making it impossible to provide convenience features like unfolding the org. element hierarchy to the currently selected node. Automatic unfolding is necessary because otherwise the

¹⁷Screenshot has been edited to fit the page.

development process would be very cumbersome. Typescript projects were typically re-compiled on file save, after which a browser reload is triggered to reflect the changes in the code. This process would cause the org. hierarchy to collapse every time a code change is performed. After introducing org. dimension support, the correct dimension also had to be selected again. The re-written implementation automatically selects the currently viewed dimension, and unfolds the org. hierarchy to the selected element by extracting this information from the URL.

Semantic URLs

While the user is navigating the accountancy module, its URL always reflects the currently selected element. See [Listing 3.20](#) for an example URL that contains the currently selected dimension (dim/[dimensionId]), and the viewed agent (agent/[agentType]/[agentId]). Also, semantic URLs allow the sending of links to certain journals to other CMSuite users.

Listing 3.20: Accountancy client URL example

```
http://cmsuite.com/#/accountancy/dim/1/agent/orgelement/100
```

Filtering and aggregation options







Each accountancy journal can be aggregated by time, as outlined in [section 3.1.2](#). Additionally, the two sides of a contribution – its receiver and its contributor – can be filtered. The user can choose whether only direct or indirect contributors/receivers are to be shown (or both). Moreover, the user can choose whether displayed org. element level of the receiver/contributor matches the selection or if it should be fixed, for example, to show teams only.

Journal icon legend

The contribution journal displays a number of icons next to the contribution count to provide information about the receiver/contributor relationship of the selected element. See [figure 3.10](#) for a description of the meaning of each icon. Additionally, contributions crossing org. units are marked in orange, and indirect contributions use an italic font type.

¹⁸Icons provided by Glyphicons.com (Glyphicons project, [2017](#)).

Figure 3.10: Journal icon¹⁸legend

Icon	Description
	Contribution is self-contained i.e. a team contributing to itself.
	Contribution to a ancestor org. element.
	Contribution to a descendant org. element.
	Contribution to a sibling, an org. element on the same hierarchy level.
	Contribution is indirect.
	Contribution is crossing org. units. The contribution receiver and contributor share no common ancestors.

3.4 Deployment and testing

To ease the development process and also to provide documentation of how Apache Lens is configured, a Docker environment has been defined. CMSuite previously made use of Docker before, so adding another setup was the logical choice. See [appendix D](#) (p. 56) for the *Dockerfile*.

Most of the functionality of this thesis can be traced to the database and Apache Lens. Considering that CMSuite is primarily a technology demonstrator a smoke test has been added to verify that all server components work together as expected. An integration test verifies the result of the introduced REST API fetched by Lens.

As the most complex client component, the AgentService was developed using a test-driven approach. Its Karma test has a statement, branch, and line coverage between 90-97%.

4 Evaluation

This section evaluates whether the requirements defined at beginning of this thesis have been met. Together with the requirements described in [section 1.3](#), acceptance criteria have been defined. The criteria are listed below, marked with a ☒ when fulfilled, and with ☐ otherwise. Features which exceed the specified requirements are marked with \ast .

Functional requirements

Requirement [Q.1](#) asks for the possibility of selecting the org. dimension as the basis of the exploration. This was achieved by introducing a tab group containing the org. hierarchy navigation tree. However, the entire navigation code has been rewritten to address other concerns, like the linking of agents in other parts of the accountancy module.

- ☒ An org. dimension can be selected.
- ☒ Upon selection, the navigation tree shows the specific org. dimension.
- \ast The navigation tree automatically unfolds and scrolls to the selected item. If the user follows links between agents within the journal view, the navigation tree will update its selection as well.

The organizational journals introduced for all current agent types (ISP, individual and org. element) fulfill requirement [R.1](#).

- ☒ Based on the selection within the org. hierarchy ([Q.1](#)), a journal of contributions is shown.
- \ast All agents displayed in a journal are links that can be used to jump to the journal of the specific agent.

Several filtering and aggregation options have been implemented so the user can select the entries displayed and their level of detail within the journal view. Together the implemented functionality satisfy the requirements [R.2](#), [R.3](#), and [R.5](#).

-
- ☑ The time range of the journal and time aggregations (per day, week, month, ...) can be selected.
 - ☑ The hierarchy level to be shown for the contributor/receiver side can be selected independently.
 - ☐ Contributions that cross org. unit boundaries can be filtered.
 - ☑ An icon is shown when a contribution goes to an ancestor or a descendant.
 - ☑ Contributions that cross boundaries of org. units are marked by a specific icon.
 - ☑ The entries of the journal are updated when filter options are changed and applied.
 - * Indirect contributions can be filtered and are marked with an icon.

Non-Functional requirements

P.1 states that CMSuite is a technology demonstrator and the solution does not have to be production ready. The concern behind this requirement is that effort spent in fine-tuning might be lost when the solution is heavily altered or even discarded later on. The requested features have been implemented with these concerns in mind. Excess features like the automatic unfolding of the org. hierarchy were developed to ease the development process, rather than providing a production-ready solution. The available filter options are also not optimized yet. Some filter combinations or options probably do not make sense in a production version, however, they help the user gain a better understanding of the data. In the beginning of a software project, it is often unclear what functionality is actually needed to provide a service. A technology demonstrator or prototype can help identify those needs.

As there are no database schema changes required, other than introducing an insignificant number of views, requirement P.2 is fulfilled.

Requirement P.3 states that the response time of the accountancy module should stay within a few seconds. This is the case as the solution proposed by this thesis has a typical response time around 250ms on an average dual-core development machine, without any optimizations.

The exploration of the CMSuite data is now possible without database or SQL knowledge. Thereby P.4 is fulfilled as well.

The architecture and implementation is built based on Apache Lens. Lens is released under the Apache Software License, Version 2.0. P.5 asks the developer

to consider and build on existing solutions with a permissive license and is thereby also fulfilled.

The fisheye based aggregation [R.8](#) was not implemented due to the lack of time. Refer to [chapter 5](#) for a possible solution.

Conclusion

From the nine functional requirements formalized, seven have been implemented completely, one partially ([item R.4](#)) and one not at all ([item R.8](#)). All non-functional requirements have been met. Ultimately, the goal for this thesis has been achieved. Non-technical users are now able to explore and account for inner source contributions within an organization. The introduced web-interface and the journals provided help them to gain a better understanding of inner source contributions within an organization.

During the implementation, a number of issues (see DAO/DTO and serialization discussion) were identified and solutions were found. However, the non-functional requirements required possible solutions to be compatible with the existing CMSuite database, thereby restricting their design.

On an architectural level, it is not known whether the current CMSuite database design provides a good fit for analytics use cases. As mentioned before in [section 3.1](#), the schema is optimized (denormalized) to fit CRUD use cases, although the requirements collected by this thesis suggest CMSuite might be used primarily for analytics. In this case, [INSERT](#) and [UPDATE](#) operations are rare in comparison to select operations and therefore optimization of insert/update do not seem to be justified. It is reasonable to sacrifice ease of insert and update operations¹ when [SELECT](#) operations become much simpler in return. This might have further implications, for example, if [INSERT](#) and [UPDATES](#) are rare, the use of a OR-mapper like Hibernate might not be necessary.

The proposed solution relies on Apache Lens to provide the analytics based on OLAP data cubes. It is not known how big the community behind Apache Lens is. A number of related companies rely on it, however, it is uncertain what will happen if they decide to abandon the project. There are alternatives to Lens, although each of these comes with their own drawbacks. For example, some of them are not written in Java, while others do not support relational queries. At the time of the writing, only Apache Kylin seemed a viable alternative, as it introduced support for relational data sources with version 2.1², released in October 2017. To date, however, this feature is not documented and lacks testing.

¹For example consistency might be more difficult to achieve.

²<https://issues.apache.org/jira/browse/KYLIN-1351>.

The proposed solution has different impacts on the previously collected stakeholders ([section 1.3.1](#)):

- Developers of CMSuite have to learn about the Apache Lens OLAP cube configuration when changes to the data cubes are required. The core architecture has been documented in [section 2.2](#) and [section 3.1.2](#). However, the introduced Jackson mix-ins potentially save substantial effort compared to the DTO approach.
- As the usage of Apache Lens introduces an additional run-time dependency, Administrators and DevOps engineers have to learn how to install and configure Apache Lens. The created Dockerfile (see [appendix D](#) (p. 56)) provides the necessary documentation.
- The introduced database views make the Lens cube configuration more robust as they provide an additional abstraction. However, database changes might break the view queries. The used common table expression (CTEs) have been elaborated in [section 3.1](#), as they might be difficult to understand. This thesis also provides guidelines for a database schema re-design should the developers come to the conclusion that analytics are indeed the main use-case for CMSuite.
- The remaining inner source stakeholders profit from the new accountancy UI, which can be used intuitively. It hides the complexity of the data schema and enables the exploration of the data collected by the patch crawler.

The implementation elaborated in this thesis also introduces technical debt. For example, the CubeQL queries are currently embedded as strings in data access objects³. While this is not ideal, it is justifiable given the technology demonstrator character of CMSuite. Lens supports prepared queries, however, they currently cannot be parameterized. As the queries can require up to 10 parameters, the method signatures of the REST resources can be confusing. To mitigate this problem, the parameters have been grouped logically and the groups are commented.

While the Lens CLI proved useful during development by allowing new queries to be easily tested, it also came with its quirks. For example, error messages are sometimes not useful and the cause of a problem often has to be traced using the log file of the Lens server. Also, queries have to be executed using “select” in lowercase letters, as this is the name of the shell command. No uppercase alias exists; this is why “SELECT” fails.

³The implications and possible solutions have been elaborated in [section 2.1.1](#).

5 Future work

As mentioned before, the user story [R.8](#) has not yet been implemented. The fisheye view shows and aggregates org. elements based on their distance to the selected element in the org. element graph. The assumption is that a user tends to select org. elements he is familiar with (and therefore most interested in). At the same time the user's knowledge about other organizational units typically decreases with their distance to the org. element he or she is assigned to.

The architecture proposed by this thesis can be used to design a journal aggregating entries based on distance. To implement the architecture, the graph distance for each receiver/contributor must be known. This can be achieved by introducing an CTE view (see [section 3.1](#)) which consists of two columns. The first column would contain every existing agent ID, whereas the second column would contain every agent within distance one¹. The solution assumes that indirect ancestors are not shown. The created view can then be used as a further dimension table. Using this approach, journal records can be aggregated by graph distance.

Contributions which cross org. unit boundaries cannot be filtered ([R.6](#)). However, an icon is shown in the journal UI whenever a contribution crosses boundaries (see [figure 3.10](#) (p. 45)). To be able to filter the corresponding journal, another view has to be introduced. This view requires three columns. The first two together list every possible contributor/receiver combination, while the third column states whether receiver and contributor share a common ancestor other than the root node (boolean). This view can then also be used as a dimension table, filtering and aggregating journal entries based on the value of column three.

Org. hierarchy branch validity

The current CMSuite database schema contains *valid_since* and *valid_until* columns for each link type (see [appendix B](#)). These columns were introduced

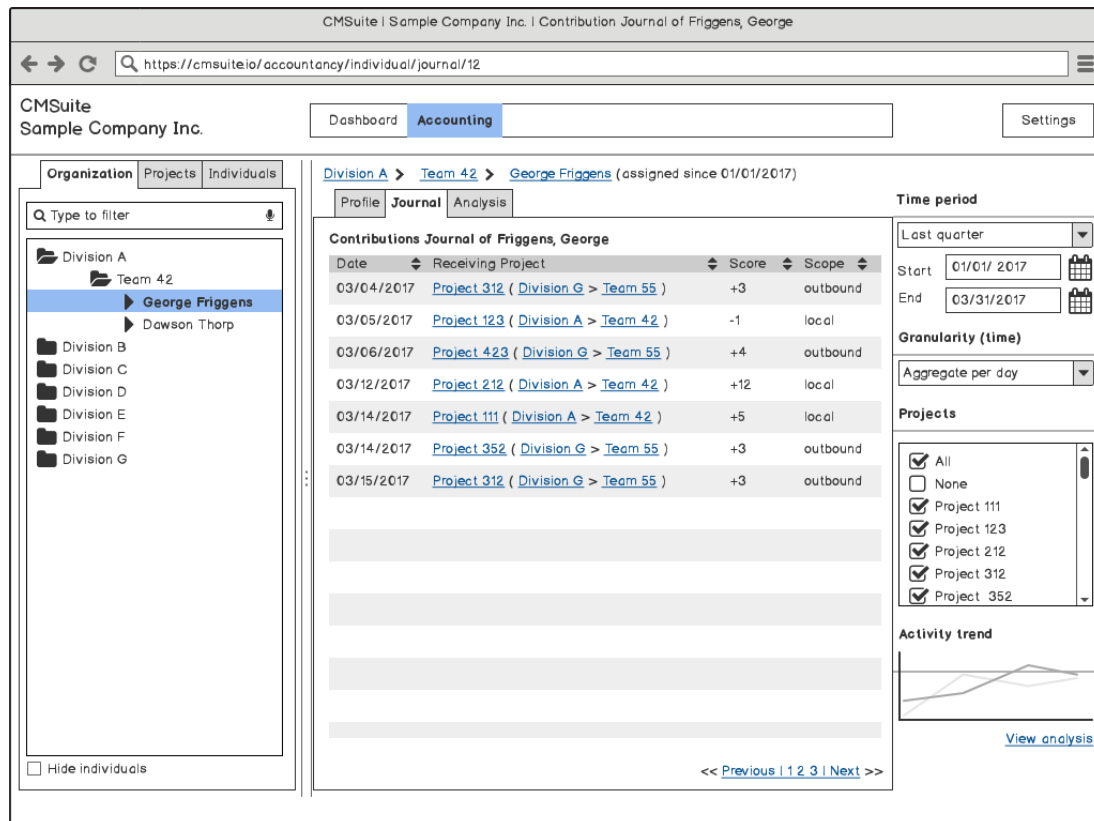
¹This solution could also be extended to contain the distance of all nodes. However, it would require a third column containing the distance.

to be able to model structural changes of an organization over time. For example, individuals might move from one org. unit to another (i.e. in case of a promotion) or management might re-structure the organization. While there were no user stories specified to address this issue, the proposed solution using Apache Lens can be used to solve it.

On a database level, the validity timestamps need to be included in the denormalized view, marked as a time partition like in [listing 3.13](#) (p. 32), [line 32](#). The time partition can then be used within a [TIME_RANGE_IN\(\)](#) clause. On the UI level, both the org. hierarchy navigation and the journal are affected. A change in the time range should also update the org. tree. For this Jackson EconomicAgentReference mix-ins need to be extended to include the validity field. Afterwards, the tree and journal should work as expected.

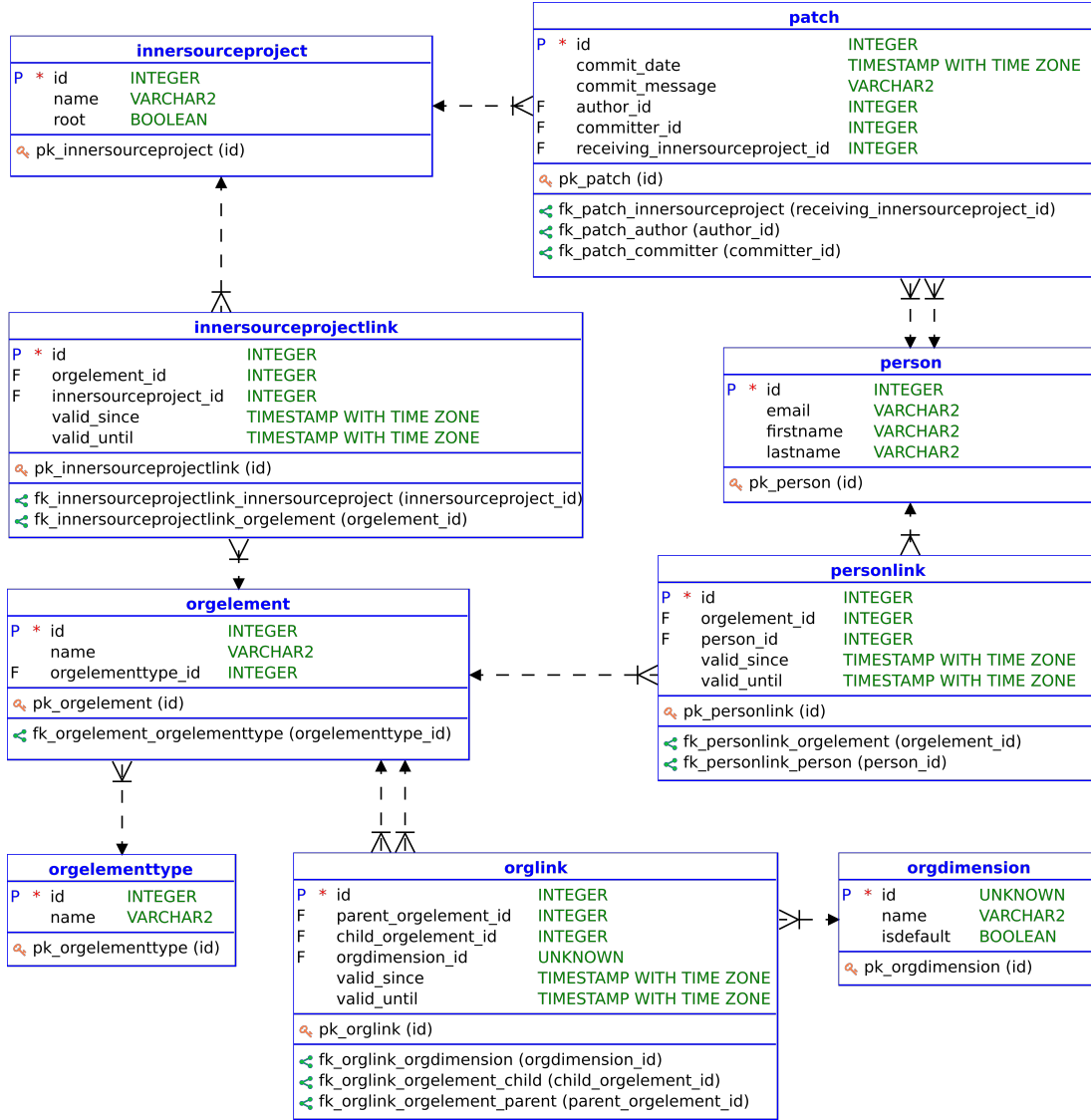
Appendix A User interface mockup

Figure 1: Early accountancy mockup for individual contributions



Appendix B CMSuite database schema

Figure 2: CMSuite database schema²



²Schema has been modified to exclude entities and columns not relevant for this thesis.

Appendix C Full database views

```

1 CREATE RECURSIVE VIEW orghierarchy(dimension_id ,
  child_orgelement_id , parents , depth , orgelementtype_id) AS (
2   SELECT DISTINCT orgdimension_id ,
3     parent_orgelement_id ,
4     ARRAY[]::integer [] ,
5     0 ,
6     e.orgelementtype_id
7 FROM orglink o
8 JOIN orgelement e ON e.id = o.parent_orgelement_id
9 WHERE parent_orgelement_id NOT IN
10    (SELECT DISTINCT child_orgelement_id FROM orglink)
11
12 UNION
13 -- recursive expression --
14   SELECT DISTINCT orgdimension_id ,
15     o.child_orgelement_id ,
16     parents || parent_orgelement_id ,
17     depth + 1 ,
18     e.orgelementtype_id
19 FROM orglink o
20 JOIN orgelement e ON e.id = o.child_orgelement_id
21 JOIN orghierarchy n ON n.child_orgelement_id =
  o.parent_orgelement_id
22 );

```

Listing 1: View for org. element hierarchy

Listing 1 shows the final view for the denormalized org. hierarchy, containing the depth (hierarchy level) and the type of the org. element from the *orgelement* table.

```

1 CREATE VIEW orgelementtypehierarchy(dimension_id ,
  orgelementtype_id , name, depth) AS (
2   SELECT DISTINCT dimension_id , orgelementtype_id , name , depth
  from orghierarchy
3   JOIN orgelementtype
4     ON orgelementtype_id=id
5   ORDER BY dimension_id , depth
6 );

```

Listing 2: View for org. element type hierarchy

Listing 2 shows the view for the org. element type hierarchy (like Organization → Product line → Team).

```

1 CREATE VIEW receiver AS (
2     SELECT DISTINCT
3         a.elem AS orgelement_id,
4         innersourceproject_id AS project_id,
5         a.elem = pl.orgelement_id AS direct,
6         a.index AS depth,
7         dimension_id
8     FROM innersourceprojectlink pl
9     JOIN orghierarchy
10         ON pl.orgelement_id=child_orgelement_id
11     JOIN LATERAL unnest(array_append(parents, child_orgelement_id))
12         WITH ORDINALITY AS a(elem, index) ON TRUE
13     ORDER BY dimension_id, orgelement_id, project_id
14 );

```

Listing 3: View for receiver hierarchy (inner source projects)

Listing 3 shows the entire receiver hierarchy view elaborated in [listing 3.6](#), including the receivers' depth within the org. hierarchy and its dimension. Refer to [listing 4](#) for the contributors' side equivalent.

```

1 CREATE VIEW contributor AS (
2     SELECT DISTINCT
3         a.elem AS orgelement_id,
4         person_id as individual_id,
5         a.elem = pl.orgelement_id AS direct,
6         a.index AS depth,
7         dimension_id
8     FROM personlink pl
9     JOIN orghierarchy
10         ON pl.orgelement_id=child_orgelement_id
11     JOIN LATERAL unnest(array_append(parents, child_orgelement_id))
12         WITH ORDINALITY AS a(elem, index) ON TRUE
13     ORDER BY dimension_id, orgelement_id, person_id
14 );

```

Listing 4: View for contributor hierarchy (individuals)

Appendix D Docker environment

Listing 5: Self-contained Lens environment

```
FROM mojo-docker.cs.fau.de/osrg/cmsuite-java8:1.0
RUN apt-get update && apt-get install -y wget tar vim
    postgresql-client net-tools && apt-get clean

ENV LENS_USER=lens

ENV HADOOP_VERSION=2.6.0
ENV HIVE_VERSION=2.1.0
ENV LENS_VERSION=2.6.1

ENV HADOOP_HOME=/opt/hadoop-$HADOOP_VERSION
ENV HIVE_HOME=/opt/hive-$HIVE_VERSION
ENV LENS_HOME=/opt/lens-$LENS_VERSION

ENV APACHE_DIST=https://archive.apache.org/dist
RUN mkdir $HADOOP_HOME \
    && wget -qO - $APACHE_DIST/hadoop/common/hadoop-
        $HADOOP_VERSION/hadoop-$HADOOP_VERSION.tar.gz \
    | tar xz --strip 1 -C $HADOOP_HOME
RUN mkdir $HIVE_HOME && \
    wget -qO - $APACHE_DIST/hive/hive-2.1.0/apache-hive
        -2.1.0-bin.tar.gz \
    | tar xz --strip 1 -C $HIVE_HOME
RUN mkdir $LENS_HOME \
    && wget -qO - $APACHE_DIST/lens/2.6.1/apache-lens
        -2.6.1-bin.tar.gz \
    | tar xz --strip 1 -C $LENS_HOME

# configure Hive
RUN mv $HIVE_HOME/conf/hive-env.sh.template $HIVE_HOME/
    conf/hive-env.sh && \
    mkdir -p /var/log/hive/query mkdir -p /tmp/hive &&
    chmod 1777 /var/log/hive /tmp/hive
ADD assets/hive/hive-site.xml $HIVE_HOME/conf/
RUN wget -q https://jdbc.postgresql.org/download/
    postgresql-42.2.0.jar -P $HIVE_HOME/lib/
```

```
# configure Lens
ADD assets/lens/lens-site.xml $LENS_HOME/server/conf/
ADD assets/lens/lens-client-site.xml $LENS_HOME/client/
  conf/
RUN mkdir $LENS_HOME/server/conf/drivers/jdbc/jdbc2 && \
  mkdir /var/log/lens && chmod 1777 /var/log/lens && \
  ln -s $LENS_HOME/client/bin/lens-cli.sh /usr/bin/lens
  -client
ADD assets/lens/jdbcdriver-site.xml $LENS_HOME/server/
  conf/drivers/jdbc/jdbc2/

# set entrypoint
RUN mkdir /opt/util
ADD assets/util/* /opt/util/
ADD assets/entrypoint.sh /
RUN rm -fR /entrypoint && chmod +x /entrypoint.sh
CMD /entrypoint.sh && /bin/bash
ENV PATH="$HADOOP_HOME/bin:$HADOOP_HOME/sbin:$HIVE_HOME
  /bin:${PATH}"

# add lens user
RUN useradd -m -d /home/$LENS_USER -s /bin/bash
  $LENS_USER && \
  chown -R $LENS_USER:$LENS_USER /var/log/hive /var/log
  /lens /tmp/hive $LENS_HOME/server/webapp/
ENV HOME /home/$LENS_USER
WORKDIR $HOME
USER $LENS_USER
```


References

- Apache Lens project. (2017). Apache Lens website. Retrieved November 6, 2017, from <https://lens.apache.org/>
- Capraro, M. & Riehle, D. (2017). Inner source definition, benefits, and challenges. *ACM Computing Surveys*, 49(4).
- Daeubler, A. (2017). *Design and implementation of an adaptable metrics dashboard* (Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg).
- FasterXML Wiki. (2018). Jackson mix-in annotations. Retrieved March 2, 2018, from <https://github.com/FasterXML/jackson-docs/wiki/JacksonMixInAnnotations>
- Github, F. J. (2018). Jackson usage examples. Retrieved March 2, 2018, from <https://github.com/FasterXML/jackson-annotations>
- Glyphicons project. (2017). Glyphicons. Retrieved December 22, 2017, from <http://glyphicons.com>
- Harold Averkamp, L., AccountingCoach. (2017). What is a journal? Retrieved December 3, 2017, from <https://www.accountingcoach.com/blog/what-is-a-journal>
- Jain, P. K. & Khan, M. (2009). *Management accounting* (5th ed.). McGraw-Hill Education - Europe.
- Kimball, R. & Ross, M. (2013). *The data warehouse lifecycle toolkit* (3rd ed.). John Wiley & Sons, Inc.
- McCarthy, E. W. (1982). The REA accounting model: A generalized framework for accounting systems in a shared data environment. *The Accounting Review*, 554–78.
- Riehle, D., Capraro, M., Kips, D., & Horn, L. (2016). Inner source in platform-based product engineering. *ACM Computing Surveys*, 42(12).
- Spolsky, J. (2002). The law of leaky abstractions. *Private Blog*. Retrieved February 6, 2018, from <http://www.joelonsoftware.com/articles/LeakyAbstractions.html>
- Wikipedia. (2018a). Matrix management. Retrieved February 19, 2018, from https://en.wikipedia.org/wiki/Matrix_management
- Wikipedia. (2018b). Representational state transfer. Retrieved February 19, 2018, from https://en.wikipedia.org/wiki/Representational_state_transfer