JOHANNES KEPLER
UNIVERSITY LINZ

Author
David Haunschmied
BSc

Submission
Department of Business Informatics – Data & Knowledge Engineering

Thesis Supervisor
Assoz.-Prof. Mag. Dr. Christoph Schütz

Assistant Thesis Supervisor
Bashar Ahmad MSc

September 2022

A Cloud-Native Data Lakehouse Architecture for Big Knowledge Graph OLAP

Master Thesis

to obtain the academic degree of

Master of Science

in the Master's Program

Business Informatics

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at

Kurzfassung

Wissensgraphen (engl. Knowledge Graphs, KGs) stellen reale Objekte und deren Beziehungen zueinander dar. KGs sind oft an bestimmte Kontexte gebunden. Dies ist im Flugverkehrsmanagement (engl. Air Traffic Management, ATM) der Fall, wo Wissen von Natur aus an einen Kontext gekoppelt ist, der aus Dimensionen wie Ort, Zeit oder Thema besteht. Diese Tatsache führte zur Entwicklung einer allgemein anwendbaren Technik namens KG-OLAP (Online Analytical Processing). KG-OLAP bietet eine mehrdimensionale Sicht auf kontextualisierte Wissensgraphen und ermöglicht kontextuelle und graphische Operationen auf dem resultierenden KG-OLAP-Würfel. Der auf GitHub veröffentlichte Proof-of-Concept-Prototyp mit GraphDB demonstriert die Funktionalität von KG-OLAP. Der Prototyp ist jedoch nicht für Big Data geeignet weshalb er nicht für datenintensive Anwendungen geeignet ist. Zum Beispiel werden alleine in Europa über zehn Milliarden RDF-Triples an ATM-Wissen jährlich generiert. Dabei ist nicht nur das Volumen ein Problem, sondern auch die Geschwindigkeit der Datengenerierung sowie die semi- und unstrukturierte Natur der ATM-Datentypen. Das Ziel dieser Arbeit ist es, eine generische Architektur vorzuschlagen und zu implementieren, die sowohl die Anforderungen von KG-OLAP als auch von Big Data erfüllt. Der erste Beitrag dieser Arbeit ist die Big KG-OLAP-Referenzarchitektur mit Prozessdefinitionen für die folgenden Hauptfunktionen: Datenaufnahme und kontextbezogene Operationen *Slice'n'Dice* und *Merge*. Der zweite Beitrag ist eine prototypische Cloud-Native-Implementierung der vorgeschlagenen Architektur, die auf Amazon Web Services bereitgestellt und anhand eines ATM Anwendungsfall demonstriert wird. Der dritte und letzte Beitrag ist eine Performanceevaluierung der Hauptfunktionen der Implementierung, um die Skalierbarkeit zu testen.

Abstract

Knowledge graphs (KGs) represent real objects and the relationships to each other. KGs are often bound to specific contexts. This is the case in air traffic management (ATM) where knowledge is inherently coupled to a context consisting of dimensions such as a location, time or topic. This fact led to the development of a generally applicable technique called KG-OLAP (online analytical processing). KG-OLAP provides a multidimensional view on contextualized knowledge graphs and enables contextual and graph operations on the resulting KG-OLAP cube. The proof-of-concept prototype with GraphDB published on GitHub demonstrates the functionality of KG-OLAP. However, the prototype is not feasible for big data which makes is unsuitable for data-intensive applications. For example, in Europe alone, over ten billion RDF triples of ATM knowledge are generated annually. With that not only the volume is a problem but also the speed of data generation as well as the semi- and unstructured nature of ATM data types. The goal of this thesis is to propose and implement a generic architecture that meets both KG-OLAP and big data requirements. The first contribution of this thesis is the Big KG-OLAP reference architecture including process definitions for the following main functionalities: data ingestion and contextual operations *Slice'n'Dice* and *Merge*. A prototypical cloud-native implementation of the proposed architecture deployed on Amazon Web Services and demonstrated using an ATM use case. The third and last contribution is a performance evaluation of the main functionalities testing their scalability.

Contents

List of Figures

Listings

1. Introduction

This thesis investigates the scalability of Knowledge Graph OLAP (KG-OLAP) - a conceptual framework for working with contextualized knowledge graphs - by proposing and implementing a cloud-native architecture. The contributed artifacts, the proposed reference architecture and the prototype implementation, lay the foundation for the applicability of KG-OLAP on big data sets. This section explains the motivation, contribution and the structure of the thesis.

1.1. Motivation

Knowledge graphs are gaining more and more popularity since Google published their search-engine enhancing knowledge graph (KG) back in 2012 [1]. KGs represent real objects and the relationships to each other. According to Serafini and Homola [2], knowledge is more and more inherently bound to specific contexts, such as to locations, times or topics. Taking search engines as an example, the current location and local time of the person who submits the search query define the results when querying for the search term *Restaurants*. Showing all results for *Restaurants* in the world without considering the context would inevitably lead to tremendous information overload and unusable results.

Context is also important in the aeronautics domain where pilots are being briefed with a flood of information without taking the flight context into account [3]. The same authors propose a technique to enrich data in air traffic management (ATM) with certain contexts to enable pilot briefings with information filtering. Based on this project, researches found out that different data in ATM is inherently context-dependent and that there is a need to better handle various data formats beyond notices to airmen (NOTAMs) [4].

Schuetz et al. developed a general applicable concept called KG-OLAP (Online Analytical Processing) which provides a multidimensional view on contextualized knowledge graphs [5]. KG-OLAP is similar to traditional OLAP but instead of numbers, the cells (=contexts) of a KG-OLAP cube contain knowledge in the form of RDF triples (Resource Description Framework). The proposed *contextual operations* (Slice'n'Dice, Merge) and *graph operations* (Abstraction, Pivoting, Reification) on the KG-OLAP cube enable a large range of different use cases. One use case are pilot briefings in ATM where the goal is to extract certain knowledge based on the context from the cube and manipulate it to provide pilots the least amount of information possible while of course not missing relevant pieces. The authors describe an implementation using a graph database (GraphDB[1]) and show its applicability for pilot briefings on small data sets. While the performance is good for small KG-OLAP cubes, the implementation is not feasible anymore once the data model exceeds about 5 000 contexts or 70 million RDF triples. The prototype is available online[2].

With the current implementation of KG-OLAP [5] it is not possible to support data intensive applications. First, data ingestion times are increasing non-linearly until a point is reached where the system can not handle new data anymore in a reasonable amount of time. As mentioned before, the maximum capabilities are less than 5 000 contexts or about 70 million RDF triples. Taking ATM as a use case scenario again, a goal might be to maintain the entire ATM knowledge of Europe in a single system. NASA researchers processed data for 1342 flights in a previous project which resulted in about 2.4 million RDF triples [6]. With over 30 000 flights per day on average handled by Eurocontrol [7], the current system would reach its limits within hours, if many new contexts are generated, or days at latest. Considering those numbers, the potential amount of ATM knowledge in Europe generated within a year is above 12 billion triples. Secondly, data is generated at a high speed with varying demand. In ATM for instance, more data is generated during holiday season and on the contrary there is less traffic during nights. Hence, a KG-OLAP system must provide constant insertion times regardless of the current load. Third, different data formats exist in the ATM domain, such as IWXXM [8] for weather data, FIXM for flight and flow information [9] or AIXM for aeronautical information services (e.g., NOTAMs) [10]. To enable a broad range application fields, it is essential that the KG-OLAP system is open for any data format. Only within ATM for instance, pilots are briefed with flight traffic, airport and weather information prior to the takeoff.

[1]https://graphdb.ontotext.com/
[2]http://kg-olap.dke.uni-linz.ac.at

The current implementation only accepts RDF format which requires other applications to transform data prior to ingestion.

To support the amount and complexity of current and future data sets, a KG-OLAP system must cope with big data. Big data is defined by three main components: *Volume*, *Velocity* and *Variety* [11]. *Volume* describes the previously unseen huge amount of data that is created in the big data era. The data is generated and ingested at high speed with spikes in both directions, hence systems need to be elastic to cope with increased *Velocity*. Finally, multiple analysts estimate that 80% to 90% percent of the data worldwide is unstructured [12]. This includes emails, presentations, videos, audio, pictures, text and more. The remaining percentage of the world's data is (semi-)structured. Self describing XML files are semi-structured while rows stored in RDMS (Relational Database Management Systems) count as structured data. Systems can not rely on structured data exclusively anymore, especially because the majority is not structured. The *Variety* component refers to this phenomenon and adds the third V to big data together with *Volume* and *Velocity*. The previously mentioned requirements for KG-OLAP combined with big data requirements led to the idea of Big KG-OLAP.

1.2. Contributions

The goal of this master thesis is to lay the foundation of Big KG-OLAP and continue from a theoretical concept proposal [13]. The authors shed light on the concept of distributed KG-OLAP cubes and updated operations from KG-OLAP that do not work for a distributed solution. The paper contains an approach for *contextual operations* (Slice'n'Dice and Merge) on distributed KG-OLAP cubes. *Graph operations* (Abstraction, Pivoting, Reification) manipulate the graph within single cells, hence the authors state KG-OLAP cube distribution do not affect the original concepts of those. Starting from these concepts and mentioned problems regarding the 3 Vs of big data, the focus of this thesis is to propose an architecture and implement a system that provides constant insertion times for big data while keeping reasonable response times for *contextual operations*. More precisely, the target is to store and maintain a large amount of basis data from which relatively small KG-OLAP cubes of interest can be queried. Those returned KG-OLAP cubes could be small enough to be processed efficiently by the current GraphDB implementation. *Graph operations* are not covered in this thesis as they are no subject to KG-OLAP cube distribution hence do not affect the resulting architecture. Scaling those is part of future research.

The current GraphDB implementation is not feasible for Big KG-OLAP. A different approach is necessary. A modern architecture to cope with big data workloads are data lakes [14]. In contrast to data warehouses, data lakes have no defined ETL-preprocesses, are cost-efficient for large amounts of data and store data in raw rather than in preprocessed format. Most of these characteristics fit well for Big KG-OLAP. However, *contextual operations* in KG-OLAP would not scale with raw data only as query complexity increases with every added data point. To support those, the system needs some preprocessing to find stored data points in the multidimensional model fast and return a result in RDF format. A hybrid solution for this use case is called data lakehouse, which combines advantages of data warehouses and lakes [15]. It adds an indexing, metadata and caching layer on top of a data lake, hence enables transaction management and fast, SQL-like queries. A data lakehouse approach was chosen as basis Big KG-OLAP architecture to ingest and store high loads of different data types while keeping a multidimensional OLAP model and caching on top.

The proposed data lakehouse architecture and implementation in this thesis follows cloud-native principles. The term cloud-native was first mentioned in literature in 2013 [16] and describes applications inherently designed for the cloud. According to a literature review study, a cloud-native application (CNA) is, among best-practices for application development such as version control and logging, characterized by its distribution, independent (micro-)services, horizontal scalability, operation on an elastic platform and a limited number of stateful services [17]. The goal of these architectural characteristics of CNAs is to build modern cloud applications that are easy to install, maintain, port, modify and scale while ensuring availability and fault-tolerance. Providing a modern application as foundation for Big KG-OLAP is important for future research efforts and the real-world applications.

1.3. Structure

This thesis is structured as follows. Section 2 covers the state of the art of applied concepts and technologies as well as related work. Afterwards, KG-OLAP is described in detail in

Section 3. Section 4 analyzes the requirements for Big KG-OLAP and identifies approaches to solve the discovered challenges. The next section explains and discusses the general reference architecture followed by the implemented prototype. Subsequently, a performance evaluation is conducted and the conclusion provides a summarized view on the work.

Section 2 covers all concepts and technologies that are important to understand this thesis. Knowledge graphs and data warehouses are explained to understand the KG-OLAP concept. Also, existing ideas that shape the design of the reference architecture are covered as well as cloud technologies that are used for prototype development. Moreover, the thesis is set into context of current knowledge by comparing it to related existing literature.

In Section 3, the KG-OLAP concept on which this thesis builds on is explained. It includes the multidimensional model of contextualized knowledge graphs, functional requirements and operations. The *contextual operations* are well described while an overview of the *graph operations* is given. Also, the limitations of the current implementation are part of this section.

A requirements analysis for a Big KG-OLAP system based on an air traffic management use case is done in Section 4. Furthermore, system design approaches are discussed to meet challenges especially to fulfill the 3 Vs of big data and functional requirements of KG-OLAP.

One contribution of this thesis, the reference architecture, is covered in Section 5. A detailed explanation of each system component and its interactions is given. Additionally, it describes the workflow of the two main functionalities, data ingestion and contextual operations.

Section 6 contains the architecture of the prototype and its implementation. The focus of this section is to explain which technologies or services, especially Amazon Web Services (AWS), are used and how they are integrated in the overall system. Moreover, a code-level overview of data ingestion and contextual operation is included.

The performance evaluation of the prototype implementation deployed on AWS is done in Section 7. It is structured in two subsections, one evaluating the data ingestion and the second one tests the contextual operations.

Finally, the conclusion in Section 8 summarizes the thesis and provides an outlook on further research directions.

2. Background

This section explains concepts and technologies used throughout the thesis. This state of the art review provides the reader a foundation to comprehend the following sections. Additionally, existing projects in the literature solving similar problems are reviewed.

2.1. Knowledge graphs and Resource Description Framework

Knowledge graphs (KGs) represent real world entities and their relationships. The term originated from the Semantic Web initiative and got popular since the last decade [18]. Semantic Web tries to bring structure to the internet, for example by applying schema.org[1]. KGs contain facts or assertional knowledge (ABox) and also terminological knowledge (TBox) [19]. For instance, the knowledge that *Runway 1 is closed* is ABox while *Runway XY has type Runway* is classified as TBox. The later ones are used to employ reasoning on the knowledge graph and derive new implicit knowledge. A knowledge graph example is illustrated in Figure 2.1.

RDF. In the Semantic Web community, the common standard to represent KGs is RDF (Resource Description Framework) [16]. The RDF model is a standard to exchange data on the web [18]. It is a semi-structured model that allows schema evolution under the open-world assumption. Entities and relationships are logically in the form of two nodes with a link while using the URI structure for naming. The format to represent knowledge is called *triple*. A triple consists of a subject, a predicate and an object and is read from left to right or from subject to object. Triples can be extended with a graph identifier resulting in a *quad*. Quads are the basis for contextualized knowledge graphs and KG-OLAP. Different RDF formats exists and are suitable for different use cases. In this thesis, the N-Quads format is used. This format is line based and best used for streaming, since the line order

[1]https://www.schema.org/

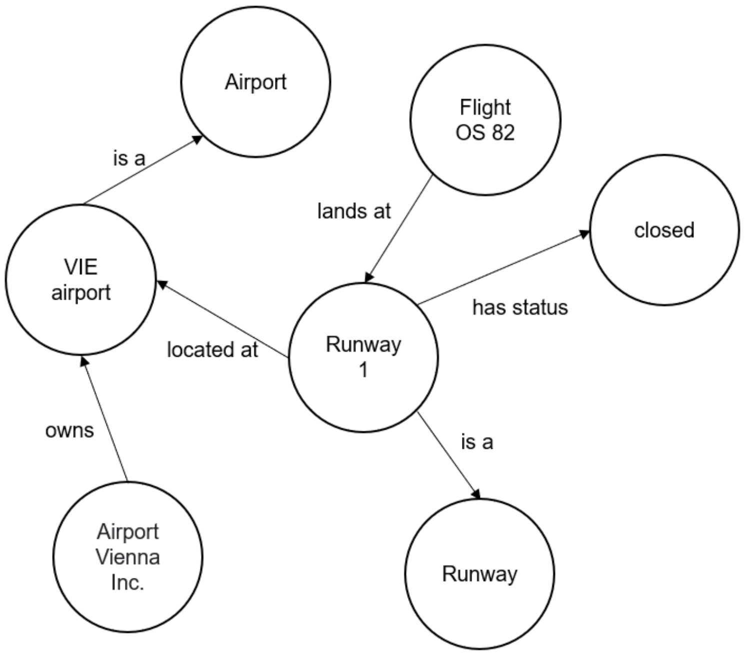


Figure 2.1.: Knowledge graph

is not important and data can be compressed [19]. The following two statements illustrate RDF and the N-Quads format in general form "<subject> <predicate> <object> <graph> ." and an example "<Runway 1> <has status> <closed> <vienna-2022-07-graph> .".

2.2. Data exchange in Air Traffic Management

AIXM [10] is the abbreviation for aeronautical information exchange model and is the standard format used in aeronautical information services such as for airports or routes. There is a UML (Unified Modeling Language) model available as well as a XSD (XML schema definition) schema with the current version *5.1.1*. Its main application are digital notices to airmen (Digital NOTAMs). Previously, NOTAMs were released in plain text that were hard to read for humans and also difficult to parse automatically. With the growing amount of NOTAMs worldwide (1 million a year), it got hard to extract information relevant for certain situations. Digital NOTAMs try to solve this issue. DNOTAMs are in XML (extensible markup language) format and use the AIXM XSD schema definition. The format is based on GML (Geography Markup Language) hence fits naturally with RDF [20]. Besides the fact that message of digital NOTAMs can be processed and transformed into RDF knowledge, it is also possible to extract the context of it. This enabled initiatives such as KG-OLAP. Listing 2.1 shows a digital NOTAM. Some elements and attributes were removed to increase the readability. The DNOTAM basically states that the airport EDDF is closed on Feb 1, 2018 from 00:00:00 until 23:59:59. The *AIXMBasicMessage* contains two members, the GML event and the notification content according to the AIXM 5.1.1 specification. It is important to mention that the structure of the DNOTAM works as follows. An element representing an *Entity* (upper case) is always followed by an element representing a *Predicate* (lower case) and vice versa. Moreover, each *Entity* has a unique GML identifier. For instance, the *AIXMBasicMessage* with the ID *9b118672 has a member* from type *Airport Heliport* with the ID *30fdd110*. Because of that it is a natural fit to transform a DNOTAM into RDF. Transformed into an RDF triple in N-Triple format, the given example would turn into: "<AIXMBasicMessage 9b118672> <hasMember> <AirportHeliport 30fdd110> .".

Listing 2.1: NOTAM example

```xml
<?xml version="1.0" encoding="UTF-8"?>
<message:AIXMBasicMessage xmlns:aixm="http://www.aixm.aero/schema/5.1.1"
    gml:id="9b118672">
    <message:hasMember>
        <event:Event gml:id="4e4d14a2">
            <event:timeSlice>
                <event:EventTimeSlice gml:id="db939cfa">
                    <gml:TimePeriod gml:id="84906336">
                        <gml:beginPosition>2018-02-01T00:00:00Z</gml:
                            beginPosition>
                        <gml:endPosition>2018-02-01T23:59:59Z</gml:
                            endPosition>
                    </gml:TimePeriod>
                    <event:textNOTAM>
                        <event:NOTAM gml:id="f571ed3b">
                            <event:affectedFIR>EDGG</event:affectedFIR>
                            <event:location>EDDF</event:location>
                            <event:text>Airport is closed</event:text>
                        </event:NOTAM>
                    </event:textNOTAM>
                </event:EventTimeSlice>
            </event:timeSlice>
        </event:Event>
    </message:hasMember>
    <message:hasMember>
        <aixm:AirportHeliport gml:id="30fdd110">
            <aixm:timeSlice>
                <aixm:AirportHeliportTimeSlice gml:id="63775a40">
                    <gml:TimePeriod gml:id="22fb4faa">
                        <gml:beginPosition>2018-02-01T00:00:00Z</gml:beginPosition>
                        <gml:endPosition>2018-02-01T23:59:59Z</gml:endPosition>
                    </gml:TimePeriod>
                    <aixm:availability gml:id="dc535683">
                        <aixm:ServiceOperationalStatus gml:id="df270f0c">
                            <aixm:operationalStatus>CLOSED</aixm:operationalStatus>
                        </aixm:ServiceOperationalStatus>
                    </aixm:availability>
                </aixm:AirportHeliportTimeSlice>
            </aixm:timeSlice>
        </aixm:AirportHeliport>
    </message:hasMember>
</message:AIXMBasicMessage>
```

Besides AIXM, there exist different data exchange models in air traffic management such as IWXXM for weather data or FIXM for flight and flow data. Those models are not discussed further as the prototype in this thesis is shown and tested with digital NOTAMs in AIXM format.

2.3. OLAP, Data Warehouses, Data Lakes and Data Lakehouses

Data warehouses and Online Analytical Processing (OLAP) are essential for data driven decisions for over 20 years [21]. Data warehouses are central databases that extract, transform and load (ETL) data from various sources and different structures. The central database is then optimized for analytical queries (OLAP) rather than for fast (Online) transaction processing (OLTP). On top of the architecture there are data mining, reporting and analysis tools which use the data warehouse as source. One major difference between operational database and data warehouses is the multidimensional view on the data. Figure 2.2 shows an OLAP cube for sales including cells that are separated along three dimensions: products, time and geography. Analytical queries are able to *Slice'n'Dice* the cube e.g. to select the number of grocery sales in Austria in Q1. Additional OLAP operations are *Roll-Up* for less detailed queries or *Drill-Down* to increase the granularity. Selecting the total number of TV sales regardless of the region or time is a *Roll-Up* example.

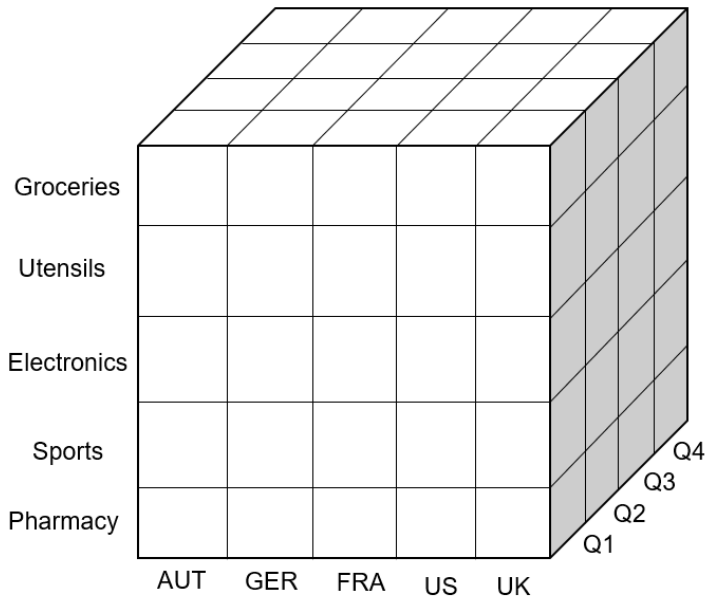


Figure 2.2.: OLAP cube

Important terms in OLAP besides *Dimension* are *Level*, *Member* and *Hierarchy*. A level specifies a granularity in a dimension. *Day*, *Month*, *Quarter* and *Year* are four levels of the *Time* dimensions while *Q1* is an exact member. A hierarchy is a complete chain of members, one for each level *Feb 1, 2022* → *Feb 2022* → *Q1* → *2022* is a valid hierarchy.

These multidimensional models are usually logically implemented in traditional RDMS databases and mapped into relations. The star schema is an example for a logical data warehouse design, where one fact table includes the numbers as well as a reference to each dimension table. An alternative is the snowflake schema with the difference that there is not a single table per dimension but multiple ones for each level. SQL (Structured Query Language) is used for ROLAP (relational OLAP) queries and also supports concepts such as *Roll-Up*. An alternative approach are MOLAP (Multidimensional OLAP) query languages.

In KG-OLAP [5], the case is similar but instead of numbers, the cells contain knowledge in the form of RDF triples. Also, OLAP is rather used to describe the multidimensional view on the contextualized knowledge graph than on the analytical focus as the concept also aims to support operational use cases. KG-OLAP supports *Slice'n'Dice* and *Roll-Up* (or *Merge*) while *Drill-Down* is the not explicitly mentioned as it is the default format. Queries that do not explicitly use *Roll-Up* return the result on the most specific granularity hence correspond to a *Drill-Down* operation. The system implemented within this thesis contains a domain specific language (DSL) called KGOQL (KG-OLAP Query Language) which is leaned on SQL but simplified and tailored for the contextual operations *Slice'n'Dice* and *Merge*. As mentioned before, the query output is a contextualized graph in RDF N-Quads format.

Data lakes were born back in 2010 [22] with the idea to store data in a single source called "lake" in its original format. The goal is to keep maintenance costs low and data openness high. To avoid data swamps there should be a set of functions, transformation engines and cleaners around the lake to ensure a minimal data quality standard and data interoperability. Data lakes try to tear down data silos and tackle big data challenges such as *Volume*, *Velocity* and *Variety* [14]. However, there are open challenges that still prevent data lakes from becoming the universal solution. Analysis performance uncertainty, quality, security and reanalyzation are some of those. For KG-OLAP, a data lake is not practicable due to the lack of the multidimensional model. Without this preprocessed structure, the context of each data file has to be analyzed during every OLAP operation. This would result in a query complexity 0(n) hence not suitable for large amounts of files.

Data lakehouses. The combination of a data warehouse and a data lake is called data lakehouse [15]. The authors define it as "a new generation of open platforms that unify data warehousing and advanced analytics". It combines the cost efficiency and data openness of data lakes as well as OLAP features such as ACID transactions, data versioning, indexing, caching and query optimization. The data is not completely ETL'd on insertion, but rather lazily transformed on querying. Figure 2.3 shows the proposed general architecture. Its basis is a data lake where any kind of data is stored in an open format (e.g. Parquet). On top of the data lake, there is a layer that extracts metadata, indexes the data and performs caching. This layer requires ETL processes that also ensure data quality simultaneously. Application fields such as business intelligence, reports, data science and machine learning communicate with this layer usually rather than querying files directly from the data lake (unless it is required). The authors especially emphasize the optimization for data science and machine learning and also propose special APIs on top of the Delta Lake implementation. While details of the proposed data lakehouse architecture in the referenced paper are not relevant, the basic idea to store data as it is with a data warehousing layer on top is a good fit for Big KG-OLAP.

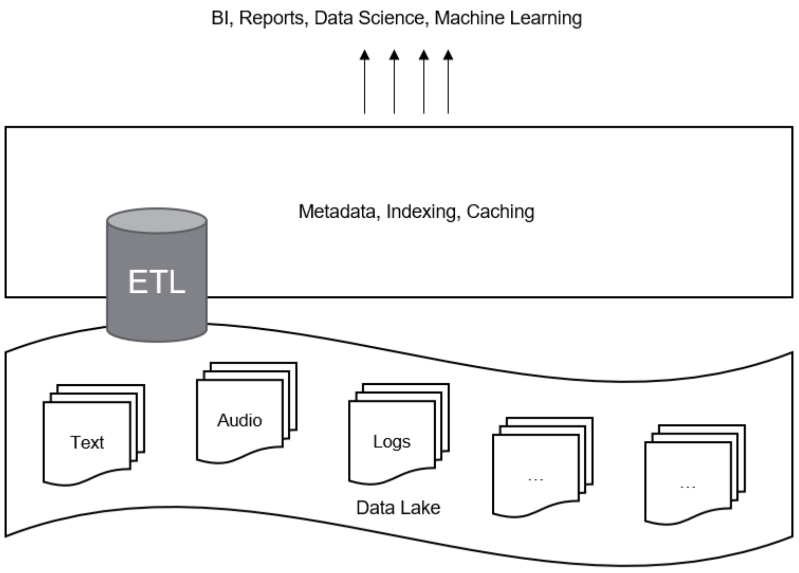

BI, Reports, Data Science, Machine Learning
Metadata, Indexing, Caching
ETL
Text
Audio
Logs
...
...
Data Lake

Figure 2.3.: Data Lakehouse (Source: Own drawing based on existing figure [15])

2.4. Cloud technologies

REST or REpresentational State Transfer is a web architecture for distributed systems [23]. REST is used for stateless client-server communication over the network. Each request is an independent entity which enhances scalability and reliability as no state has to be stored between requests. The main advancement of REST over other network communication architectures is its uniform interface to represent resources. Instead of methods, the main element of REST is the resource. Endpoints are built based on resource names and identifiers (Uniform Resource Locators or URLs) and every endpoint follows a similar pattern. In addition, the HTTP method describes the operation on the resource. For instance, a HTTP GET request on a specific resource with a given identifier always returns the same resource and will never modify it. REST allows transparent and standardized communication in the internet era and is a widely accepted architecture for cloud applications.

gRPC stands for gRPC Remote Procedure Calls is a framework for high performance network communication [24]. It is a novel alternate technology for REST to connect services in their environments. Instead of striving to standardize resource access and modification, gRPC relies on a shared protocol buffer message and service definitions. The protocol buffer format provides binary serialization for arbitrary data packages that do not exceed a few megabytes [25]. Creating and parsing protocol buffer messages is more efficient than transmitting human-readable formats such as JSON. Protocol buffer definitions need to be complied for each technology separately. Currently, there are 11 different languages supported. gRPC is based on HTTP/2 which allows multiplexing and connection reusage. Together with an efficient binary packaging, gRPC is faster than REST [26].

Container and Docker. The goal of a standardized container is to put a software component, including all its dependencies, into a self-explanatory and portable format. A container can be executed, regardless of its content, by any container runtime environment without additional dependencies and without knowledge of the underlying infrastructure [27]. Containers leverage basic modern Linux kernel features such as control groups (cgroups) and namespaces to isolate functionality and resources, similar to virtual machines. However, a container has less overhead than a virtual machine and is easier to port [28]. Containers play a big role in cloud-native development as they enable a self-contained and independent deployment of services [17]. A concrete container technology is Docker [29]. Docker is widely used and provides a set of tools to create container images, distribute and manage containers. Its container runtime is also supported by Kubernetes.

Kubernetes is an open source container orchestration software maintained by the Cloud Native Computing Foundation (CNCF) [30]. It is the industry standard to develop reliable, scalable distributed cloud-native systems. Apart from automated scaling mechanism for deployments it provides failover capabilities in case of system errors. Many cloud providers including AWS provide managed Kubernetes services such as Amazon EKS which is used in this thesis to run the Big KG-OLAP lakehouse. Along with the operational benefits of Kubernetes it enables abstracting the underlying infrastructure. There exist many concepts in Kubernetes for different kind of abstractions [31]. The declarative *Deployment* concept defines the configuration of *Pods* and *ReplicaSets*. A key concept in Kubernetes is a *Pod*, which consists of one or more (Docker) containers and represents the smallest logical unit in the ecosystem. A *ReplicaSet* on top ensures that the desired numbers of *Pods* are actually running. Apart from concepts for workload resources there are others, such as *Service* and *Ingress* for *Pod* network access and load balancing. The declaration of the concepts used for the deployment of the Big KG-OLAP lakehouse is added in appendix A.2.

Service mesh and Linkerd. A service mesh [32] sits right between services and manages and observes the traffic between them. The main purpose of using service meshes is reliability, observability and security. Most service meshes (including Linkerd) operate on ISO OSI layer 7 where its main focus is on HTTP including HTTP/2. Service meshes can retry failed idempotent HTTP requests, block unauthorized ones and collect communication metrics such as success rates. Linkerd is an open source service mesh for Kubernetes [25]. It is a lightweight proxy that runs as a sidecar container in every "meshed" *Pod* in the Kubernetes cluster. Because it is able to manage HTTP/2 traffic and therefore gRPC calls, it is used in the Big KG-OLAP deployment to enable load balancing for multiplexed and reused gRPC connections between the *Surface* and *Bed* services in Kubernetes.

NoSQL databases are an emerging trend that emerged due to scalability issues with traditional relational databases [33]. NoSQL or "Not Only SQL" refers to distributed databases of different subtypes that prefer availability over consistency according to the CAP theorem [34]. These subtypes include document databases, key-value caches, key-value stores, column stores or graph databases. Each type tries to solve a different issue while the common goal is to provide schema flexibility and big data support. NoSQL databases are not designed to meet the strict ACID properties of traditional databases but are rather optimized for availability, scalability and performance. On read, NoSQL databases guarantee that a non-error response is returned but not that it contains the latest update. This might not be suited for crucial applications that rely on strict transactions but is suited well for large distributed systems.

Message queues enable asynchronous communication in distributed systems [35]. A queue is a stateful message store that keeps a message until it is processed and deleted. This architectural pattern allows decoupling into small stateless services that require reliable asynchronous communication. Different forms of message queues such as point-to-point messaging or publish/subscribe exist and are suitable for various use cases [36]. In point-to-point, the queue is populated by a producer, a consumer periodically checks for new messages and one message is eventually processed by exactly one consumer. In contrast to that the latter one relies on the consumer to subscribe on topics of interest without the producer actively sending messages anywhere upfront.

Object stores store *objects* instead of *files* or *blocks* [37]. Objects can be any size or type, including files, database tables or multimedia. They combine the advantages of both files and blocks. Similar to blocks, objects can be directly accessed via an identifier without a storage device in front (e.g. a server) while providing a convenient file-like access interface. Additional metadata natively attached to the object enables features such as different per-object access policies [38]. Object storages work well for unstructured append-only data where in fact it provides infinite scale hence it is a good fit for long-term storage in data lakehouses [39]. They do not support essential file update mechanisms locking or sharing hence object storages should not be used for transactional data.

2.5. Related work

This section covers existing projects in the literature that are related to this work. Besides a short overview of every selected research it discusses overlaps and differences to the architecture and implementation presented in this thesis. Also, similar papers are grouped for better comprehension of this section aiming to place this work into the context of existing knowledge.

Data lakehouses. The term data lakehouse was introduced in 2021 [15]. The authors present the concept behind it, its motivations and requirements. They also compare this concept with the data lake and data warehouse and that a data lakehouse combines the advantages of both of them. In addition to the general architecture, the authors present an implementation based on the data lakehouse framework *Delta Lake*. The proposed implementation is based on Amazon S3 storage in Apache Parquet format with the transactional processing engine Delta Lake on top and with metadata, caching and indexing in between. SQL APIs and declarative DataFrame APIs enable OLAP queries and machine learning applications on the data lakehouse. The system is flexible and configurable. It can be explored as a potential data lakehouse for a Big KG-OLAP implementation in a future research. However, there are open questions including RDF mapping, real time querying and knowledge propagation which were the main reasons that an own data lakehouse prototype was built instead.

Similar to this thesis the authors of a prototypical research study also built a cloud-native data lakehouse based on open source tools such as Apache Spark or PostgreSQL for network telemetry and analytics [40]. The authors also chose to build on cloud-native principles to enable operation on modern private, public and hybrid clouds. The architecture is designed to work with (relatively small) row-based data that is temporarily stored in a relational PostgreSQL database which can be analyzed using SQL later. Hence, the current system would not work for large unstructured data objects, real time queries, or RDF based output.

Another study proposed a data lakehouse architecture for health data analysis which focuses especially on fine granular access control and permission management due to the sensitive nature of the data [41]. The system accepts structured, semi-structured and unstructured data. It does pre-processing, intensive processing and ongoing re-processing to keep data quality high and ready for analysis. These preparation steps are called pre-processing, data cataloging and data placement. In contrast to this work and the other lakehouse architectures, the authors do not consider those steps as part of the lakehouse. They see the lakehouse as a storage engine consisting of a metadata management layer, a data management layer and three storage levels namely caching layer, high performance storage layer and long term storage. The stored file format is Parquet and the tests show that they could reduce the query times as well as the used storage significantly. The authors do not state which technologies they used to implement the preparation steps or the lakehouse itself, hence it is hard to argue its applicability for Big KG-OLAP. From a high-level view it is a similar architecture without on-demand (pre-)processing and without the possibility to output RDF data.

OLAP over big data. An essential part of this thesis is the scalable index store which implemented the data warehouse star schema. Instead of a traditional relational database, the NoSQL technology Apache Cassandra is employed which is distributed per design hence horizontally scalable. Instead of facts, the "fact table" contains file locations referencing data in the Big KG-OLAP lakehouse. An existing paper follows a similar idea as the authors implemented a NoSQL based OLAP cube system with Apache HBase and Hadoop [42]. Its central contribution is the MC-CUBE (MapReduce Columnar CUBE) operator which is able to perform OLAP operations on non-relational and distributed data warehouses. This research shows that NoSQL database can lead to increased performance in (big) data warehousing.

Chen et al. [43] describe a distributed data warehouse applicable for large data sets. Its four conceptual modules - data acquisition, data storage, OLAP analysis and data visualization - are entirely based on Apache open source software projects such as Fluma, Kylin and Kafka except one tool called Saiku, which is used for visualization. The scalable system supports relational data sources as well as streaming data and includes two OLAP engines for different query requirements. Especially for MOLAP (multidimensional OLAP) which is performed by the Kylin engine, the system tests show very low operation times. The data warehouse is tailored to support traditional numbers-based-OLAP hence can not fulfill special KG-OLAP requirements such as schema variability or knowledge propagation.

Big data in ATM. A recent advance in ATM is the system-wide information management (SWIM) and with it the ATM community recognized the need to work with big data [44]. The goal of SWIM is to integrate and standardize various sources and data formats (e.g. AIXM or FIXM) in the ATM domain hence it is similar to the scope of the Big KG-OLAP lakehouse trying to merge different sources into a single virtual knowledge graph. The referenced paper extends the current SWIM architecture with frameworks that enable big data processing including data acquisition, data filtering and data lifecycle management. The proposed solution for that is a data lake extension based on big data technologies such as Hadoop or Apache Storm. In addition, a machine learning and analytics layer should create the possibility to gain deep insights in the data. For that, the authors propose the use of Apache Spark for machine learning, which is a widely used framework for big data processing. The SWIM initiative with big data support tries to solve a similar issue as Big KG-OLAP does for the ATM domain, providing a system for ATM big data. However, KG-OLAP is a generic approach for multidimensional operations on contextualized knowledge graphs while SWIM targets specific requirements in ATM.

Another approach to handle big data in ATM is described by [45]. It contains a complete architecture for handling large amounts of heterogeneous data in ATM. It is based around the data lake concept and includes "Ingest Adapters" where one ingest adapter is required per file type. The ingest adapters process incoming files at ingestion time before they are stored in the AWS hosted data lake which contains three different zones. The raw data processing zone archives data as it is in a file system such as HDFS using a "file type specific archiver". Only a subset of this data is stored in the processed data zone, which requires processors for each file type. Special data preparations are done in the refined data processing zone containing special analyzers. The processed and refined data is kept in Elasticsearch/PostgreSQL. Analytics & Visualization applications such as SQL, Kibana or Tableau enable insights on top of the data lake. The approach to treat different file types differently is similar to the *Engine* concept in this thesis. However, it lacks indexing support as it does not support extracting contexts out of ingested files.

Big data platform. An interesting pioneering work from 2011 presents ASTERIX, a scalable "non-Hadoop" data platform for evolving-world models [46]. Nowadays, we would refer to the system using keywords such as "big data" or "knowledge graph". Although it solves a different problem, this project follows similar approaches as the architecture presented in this thesis such as MapReduce computation and schema-variability. Also, ASTERIX comes with its own semi-structured JSON-like data model and XQuery inspired query language AQL. Instead of a central object storage and index store, the system is distributed across metadata nodes and compute nodes connected via a high-speed network, each using its own local disks for storage. This is not surprising as network accessible distributed storage mechanisms were not as mature back then. The project is known as Apache AsterixDB today[2].

[2]https://asterix.ics.uci.edu/

3. Knowledge Graph OLAP

Knowledge graph (KG) online analytical processing (OLAP) is a concept invented by Schuetz et al. [5] built on the fact that knowledge graphs are often context-dependent. This section gives an overview of KG-OLAP with a focus on concepts important for this thesis.

KG-OLAP provides a multidimensional view on knowledge graphs and extends contextualized knowledge graphs with OLAP and graph operations. Figure 3.1 shows a KG-OLAP cube with the three dimensions *time*, *geography* and *importance*. The single cells represent the contexts or the distinct graphs, each containing a set of RDF triples. Contextual operations can be used for queries such as *"obtain the knowledge for Feb 12 in LOVV-2 classified as critical"*.

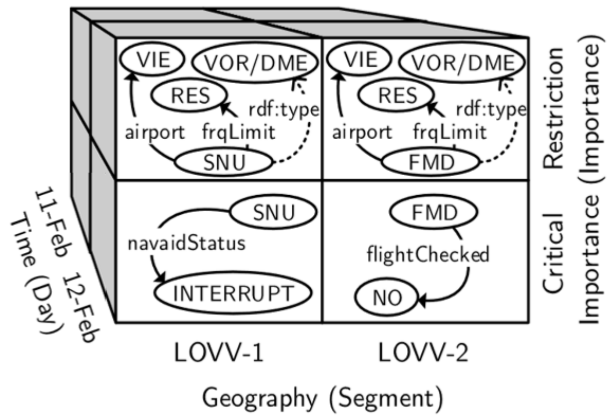


Figure 3.1.: KG-OLAP cube (Source: Own drawing based on existing figure [5])

The authors developed the KG-OLAP concept by focusing on two ATM use cases. The first use case is *pilot briefings*. Prior to departures, pilots are briefed with different types of data such as NOTAMs or weather data. Without context-based filtering, pilots drown in information. KG-OLAP should not only enable granular information selection based on the context but also knowledge abstraction to further improve information quality. In Use Case 2, air traffic flow and capacity management (ATFCM) employs post operational analysis where large amounts of ATM knowledge is mined to learn from past operational events. NOTAMs, weather data and flight data are used to analyze patterns such as the main reason for flight delays in a specific region. The first use case is further described in the next section while additional information for the second use case can be found in Schuetz et al. [5].

The requirements analysis of this use cases led to seven *functional requirements* of KG-OLAP. Next, a summary of those are given.

1: Heterogeneity. The system has no fixed schema. It allows multiple types of entities, relationships and employs a variable schema.

2: Ontological knowledge. The inclusion of domain specific information or "schema knowledge" in the KG is necessary.

3: Self-describing data. No external schema that defines the structure of the data is present. Instead, the data contains metadata that describes it.

4: Modularization. Knowledge that is true in different scopes must be split into multiple modules. In other words, attaching knowledge to a context (=module) must be supported.

5: General and specific knowledge. It must be possible to store knowledge on different granularity levels. For instance, knowledge can be true for the entire country of Austria not only for the Vienna airport. Querying for *relevant knowledge* of Vienna, *general knowledge* valid for Austria has to be included in the result. This concept is called *knowledge propagation*. Figure 3.2 illustrates how knowledge from more general contexts are implicitly inherited to more specific ones. The boxes are the different modules or contexts annotated with dimension members on different levels. The arrows indicate the inheritance. *C1 - C3* represent the knowledge of the three contexts. Knowledge with a solid line is bound to the context while knowledge with a dotted line is inherited from a parent context via knowledge propagation.

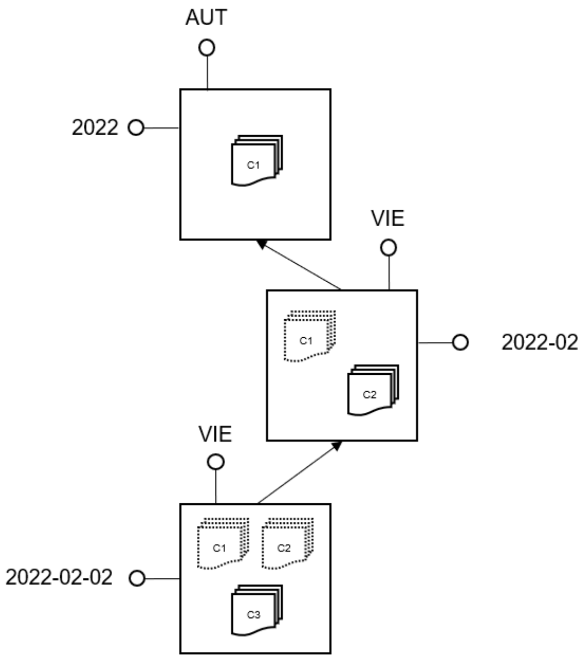


Figure 3.2.: Knowledge propagation concept

6: Knowledge selection and combination. The system must provide query operations for selecting and combining different modules (=contexts). In the paper, those are called *contextual operations* and consist of the OLAP operations *Slice'n'Dice* and *Merge* (*=Roll-Up*). Figure 3.3 shows the transformation of a KG-OLAP cube with *Slice'n'Dice* and *Merge*. For better comprehension, the example contains the same initial KG-OLAP cube as illustrated in the previous figure. The contextual operation *Slice'n'Dice* cuts certain cells out of the cube to select a subcube of it. The input for this operation are members, or level to value pairs. In the example, the members *VIE: location* and *2022-02: month* are given. These members then locate a specific coordinate in the cube. Every context at or below the coordinate is considered as *relevant context*. Every *relevant context* then results in a contextualized graph containing its own as well as inherited knowledge. The *Slice'n'Dice* operation is defined slightly different in the referenced KG-OLAP paper where contexts above the specified coordinate are also considered as *relevant contexts*. The *Merge* operation is used to reduce the granularity of the cube. Mostly, *Merge* is used after *Slice'n'Dice* to combine knowledge into a single resulting graph. The input for *Merge* are levels, where each context on a more specific granularity is rolled up to the given level. In the provided example, the context *VIE: location - 2022-02-02: day* is rolled up to *VIE: location - 2022-02: month* as *month* is given. The other context *VIE: location - 2022-02: month* does not have to be rolled up to *month* as the time dimension is already on *month* level. Finally, equal resulting contexts are merged into the same graph and duplicated knowledge is dropped. The result is a single graph containing knowledge from *C1*, *C2* and *C3*.

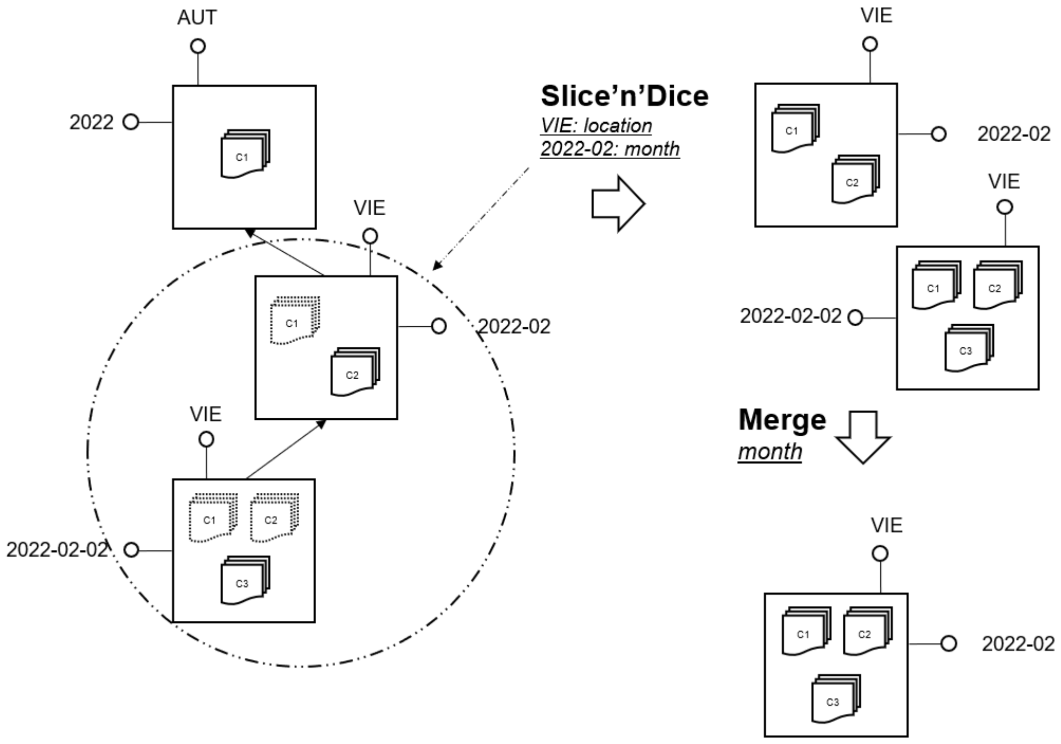


Figure 3.3.: KG-OLAP contextual operations

7: Knowledge abstraction. Query operations allow to obtain an abstract view of the knowledge. The paper proposes three *graph operations* called *Abstraction* allows to re, *Reification* and *Pivoting*. *Abstraction* replaces knowledge with more abstract entities. *Reification* and *Pivoting* are special operations to preserve contextual or assertional knowledge after a merge. These *graph operations* are not implemented in this thesis as they are no subject to KG-OLAP cube distribution hence do not affect the resulting architecture. Further information can be found in the KG-OLAP paper.

Limitations of the current GraphDB implementation described by Schuetz et al. [5] are mainly caused by the architecture and the resulting knowledge propagation on data ingestion. GraphDB[1] is a graph database or RDF triplestore hence the multidimensional model had to be implemented to fit the graph's data structure. Knowledge propagation and reasoning was realized via materialization in dedicated named graphs storing inferred knowledge for each context. Looking at Figure 3.2, the dotted context knowledge is not logically derived from the hierarchy but physically inherited at ingestion. This led to a non-linear increase in insertion time and a limit of less than 5 000 contexts or about 70 million triples in the test environment before it got unfeasible to maintain or to ingest more data. Apart from performance limitations, the GraphDB only allows RDF triple insertions but not raw data. To ingest NOTAMs in AIXM format or weather data in IWXXM format, external preprocessing is necessary.

[1]https://graphdb.ontotext.com/

4. Requirements analysis and solution approaches

This section broadly analyzes requirements and its solution approaches for a Big KG-OLAP system. The requirements can be separated into KG-OLAP, big data and other requirements. The KG-OLAP requirements include the seven functional requirements of KG-OLAP defined in section 3. Big KG-OLAP extends KG-OLAP with the ability to maintain big data workloads. Hence, it is essential for the system to address the V's of big data. The second section analyzes these non-functional requirements. In addition, other requirements are examined. Based on the analysis, solution approaches for each requirement are discussed which form the basis for the resulting architecture.

4.1. KG-OLAP requirements

KG-OLAP functional requirements defined in the referenced paper [5] are explained in section 3. These seven bullet points derive from use cases in ATM and previous initiatives and got generalized for the domain independent KG-OLAP concept. Big KG-OLAP has to fulfill every requirement of KG-OLAP as well. However, the last requirement *7: Knowledge abstraction* does not affect architectural scalability issues [13]. Therefore, it is not implemented in this thesis but part of further research to integrate it. This thesis focuses on the first six KG-OLAP requirements.

1: Heterogeneity. This refers to the requirement of an open world assumption. To fulfill this requirement, the authors of the KG-OLAP paper propose to use RDF. The format is used to represent interconnect data on the web and has no fixed schema [47]. It can be extended with any type of entity and relationship hence is not limited to structured homogenous data. Therefore, RDF is a good fit to fulfill KG-OLAP requirement 1. In this thesis, RDF is also used as output format for *contextual operations*.

2: Ontological knowledge. Any data store needs some type of structure. In the semantic web and knowledge graphs, this is called ontological knowledge. For instance, the triple *"VIE has status closed"* is valid while *"closed has status VIE"* makes no sense. The web ontology language (OWL) is a extensible format to describe such constraints. OWL can be represented in RDF format, hence the use of RDF as data format for (Big) KG-OLAP fulfills this requirement as well.

3: Self-describing data. The third requirement of KG-OLAP also targets the application of knowledge graph technologies similar to the first and second one. In ATM, there are different data formats employed and combined with a large amount of entities and relationships, a fixed relational schema would soon reach its limits. Hence, it is important to include schema knowledge in the data itself. This is possible in RDF. Besides assertional knowledge (ABox), RDF allows to include terminological knowledge (TBox). Just like usual graph knowledge such as *"VIE has status closed"* is it possible to add TBox knowledge describing triples such as *VIE is type Airport* to the graph.

4: Modularization. The main contribution of KG-OLAP is the extension of modularized operations to contextualized knowledge graphs. In the KG-OLAP paper, the authors implement the modularization with GraphDB and SPARQL. This is a good fit as both technologies work with contextualized knowledge graphs of RDF quads, but there are limitations as discussed in section 3. In this thesis, the approach is different. Instead of the obvious choice of graph technologies, an index store similar to the star schema in traditional OLAP was chosen to maintain the multidimensional model. This allows to insert knowledge independent of each other hence makes it possible to maintain larger amounts of data.

5: General and specific knowledge. One requirement in ATM is that knowledge that is valid for an entire region is also valid for every single route or airport within that region. The system must allow on the one hand to insert data on different granularity levels and on the other hand to inherit knowledge from higher levels (*knowledge propagation*). To solve the first part, the implementation must accept hierarchies on each level and validate each of them. For instance, the time hierarchy *"2022:year → 2022-02:month → [EMPTY]:day"* is valid while *"2022:year → [EMPTY]:month → 2022-02-01:day"* is not. Knowledge propagation is done on querying time rather than on insertion time. This enables constant insertion times while it only has a limited impact on queries as the KG-OLAP cubes queried are relatively smaller than the entire KG-OLAP cube stored. The knowledge propagation is implemented as an iterative process where general contexts are bound to specific ones. Both parts are realized on code-level and explained later in the thesis.

6: Knowledge selection and combination. In KG-OLAP it is important to retrieve a relatively small cube of the entire cube. For instance, for efficient and effective pilot briefings it must be possible to select data for a specific flight or at a specific location at a given time and maybe other dimensions. The *contextual operations* in KG-OLAP are *Slice'n'Dice* and *Merge*. The current GraphDB implementation utilizes SPARQL to perform these operations. In this thesis, a different technical approach is taken. For contextual operations, an own DSL named KGOQL is proposed. This language is based on SQL and allows to specify *Slice'n'Dice* and/or *Merge* queries. The *Slice'n'Dice* part of a given query is used to select contexts from the index store. The later one defines the resulting graph granularity returned by the query.

4.2. Big Data requirements

Big Data can be defined by 3 Vs according to [11]. There exist further definitions in literature and in practice adding more Vs to big data, but the referenced paper is the most accepted basis definition and fits for this thesis. The three main components of big data compared to traditional data workloads are *Volume*, *Velocity* and *Variety*.

Volume. The current KG-OLAP implementation has a limit of less than 5000 contexts or tens of millions of triples. The goal of Big KG-OLAP is to get rid of these limitations. Taking ATM as an example, NASA researchers processed data for 1342 flights which resulted in about 2.4 million RDF triples [6]. With over 30000 flights per day on average handled by Eurocontrol [7], the current system would reach its limits within hours, if many new contexts are generated, or days at latest. Considering those numbers, the potential amount of ATM knowledge in Europe generated within a year is above 12 billion triples. To address the problem of volume, the GraphDB implementation is replaced by a data lakehouse approach. Rather than storing triples centralized in a graph database, the idea is to store data in raw format, with metadata, indexing and caching above to speed up queries. Figure 4.1 shows the high level architecture of the Big KG-OLAP lakehouse based on the data lakehouse concept. Ingested data objects of any format are stored in raw format in a cheap object store rather than in converted RDF format in a graph database to keep operating costs low and scalability high. The layer on top of that shadows the raw physical storage. Newly ingested data is categorized, its context gets extracted and eventually the resulting indices are stored. Besides context extraction, the ETL-Engine also does the mapping from the raw data into RDF. This is explained in detail in the *Variety* paragraph. Also, recently used data gets cached in an RDF cache as it is considered as more relevant.

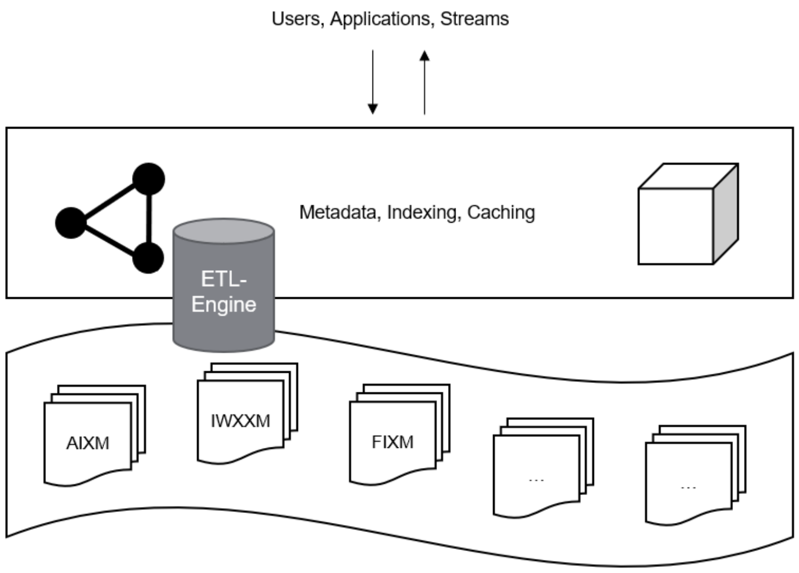


Figure 4.1.: Big KG-OLAP Lakehouse (Source: Own drawing based on existing figure [15])

Velocity. Currently, the KG-OLAP implementation is monolithic and the data ingestion rates rise with the amount of triples stored in the system. Apart from the amount of data, there is a limit on how many triples can be inserted in a given time frame. Hence, it is not scalable. This can be a severe problem for certain applications. For example, the amount of data generated in ATM in Europe is approximately 30 million triples per day (approximation based on numbers from two sources [6] [7]). Of course, the insertion rate is not constant, but varies from day to day or even within hours. Because of that, a Big KG-OLAP system has to be elastic. This means that it must be possible to scale the system up during times with large ingestion rates (e.g., holiday season) and down during more quiet times (e.g., night). To achieve that, the system architecture has to be distributed and horizontally scalable by design. Such service oriented architecture (SOA) is shown in Figure 4.2. Services are split by functionality and are independently replicable. There can be six instances of *Service A* while there is a single instance of *Service B*. These services communicate over standard network interfaces such as HTTP. The horizontal scalability of each service is essential to avoid bottlenecks and balance demands precisely. For example, a lot of *Service A* instances are required during high ingestion rates, while no *Service C* instance is required at all for this functionality. Inside the system, programming paradigms such as asynchronous processing for data ingestion and MapReduce for contextual operation processing enable the scalability on the code level. Each of these services is operated on an elastic platform that allows automatic scale-up and scale-down.

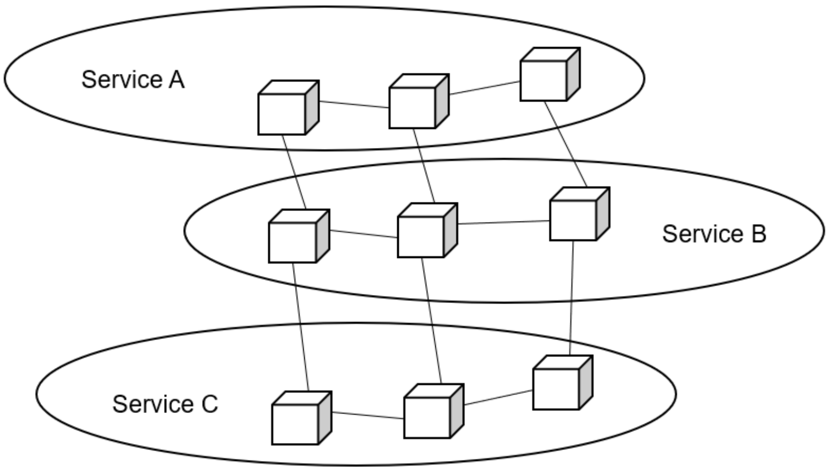


Figure 4.2.: Distributed and horizontally scalable system concept

Variety. A key requirement for Big KG-OLAP is its support for different data formats. Currently, data is preprocessed and transformed into RDF before it is inserted into the KG-OLAP GraphDB system. This is okay for small use cases but does not fit the idea of big data well. Instead, the system should be standalone and independent of other ETL processes, at least from an ingestion perspective. As mentioned beforehand, the data lakehouse approach is a good architectural fit as it provides scalable storage and data warehousing features. A key part of a data lakehouse are the integrated ETL processes. In

Big KG-OLAP, the ETL part is called *Engine* and its concept is shown in Figure 4.3. The Big KG-OLAP lakehouse basically accepts any data format. The only prerequisite is that there is an implemented *Engine* for the given format. In the ATM example, there could be two *Engines*, namely *AIXM Engine* and *IWXXM Engine*. A Big KG-OLAP lakehouse with those two *Engines* will then accept exactly those two formats. The amount of *Engines* per Big KG-OLAP lakehouse is unlimited. Basically, the *Engine* ensures data quality and transforms the (unstructured) data into semi-structured data. On data insertion, the metadata is extracted to categorize the data and choose the correct *Engine*. The *Engine* is then used to extract the data's context, resulting in one or more indices because one data point can belong to multiple contexts, for instance many days. The sum of all contexts form the multidimensional view on the virtual KG-OLAP cube. On querying, the data is lazily mapped into RDF using the *Engine*.

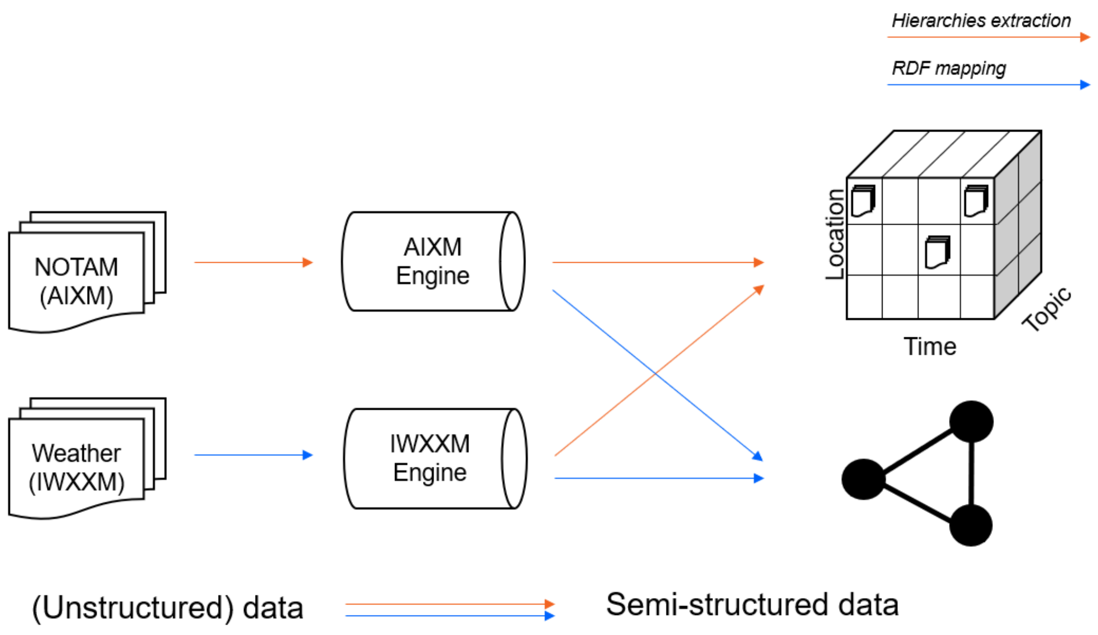


Figure 4.3.: Big KG-OLAP Engine concept

4.3. Other requirements

Apart from functional KG-OLAP requirements and non-functional big data characteristics, there are additional requirements that the Big KG-OLAP lakehouse implementation must fulfill.

Cube schema. The key configuration of a Big KG-OLAP lakehouse is the cube schema. It defines the set of dimensions and levels of the multidimensional model as well as the available hierarchies. This cube schema can not be hard-coded, otherwise the solution is not generally applicable. It should rather be configurable and considered as slowly-changing. Although the focus of this thesis is not to provide a production-ready solution, there should still be no dimensions or hierarchies in the code to provide a strong foundation for future efforts. The idea is to provide a configuration file with a definition of the dimensions together with the levels. The Big KG-OLAP lakehouse system must be able to interpret this file and dynamically adapt its behavior. An example of a dimension is *Location*. The levels of the *Location* dimension in hierarchical order are *Territory*, *Flight Information Region (FIR)* and *Location*. A *Territory* has many *FIRs* and a *FIR* in turn has many *Locations*. In addition, the file must contain the available hierarchies which could be stored in a database as well for more flexibility. Hierarchy definitions are necessary to create hierarchies out of single members. For instance, it would not be possible to infer the complete hierarchy from the member *LOWW:Location*. The hierarchy definition *Austria:Territory* → *LOVV:FIR* → *LOWW:Location* is required to know that the location *LOWW:Location* belongs to the fight information region *LOVV* and the territory *Austria*.

Cloud-native principles. As defined by a literature review study [17], a cloud-native application (CNA) is a modern approach to designing an application specifically to run in the cloud. It is characterized by its distribution, independent (micro-)services, horizontal scalability, operation on an elastic platform and a limited number of stateful services. In order to provide a usable foundation for future research efforts and production deployments, the Big KG-OLAP lakehouse architecture and the prototype implementation should follow those principles. Most of these requirements inherently come with the need to support big data but cloud-native requirements are more specific. According to them, the Big KG-OLAP lakehouse must be split into independent (micro-)services that are horizontally scalable while the number of stateful services is kept to a minimum. Moreover, operation on an elastic platform such as Kubernetes or selected Amazon Web Services is required. The approach to designing the Big KG-OLAP lakehouse as a cloud-native application is to split it into stateless services that contain the business logic surrounded by shared stateful services such as database or storage.

Application requirements for Big KG-OLAP system evaluation. The implemented Big KG-OLAP lakehouse in this master thesis is a generic solution that can be tailored to any use case. As mentioned previously, the only use-case-specific parts are the *Engine(s)* and the *Cube schema*. Although the architecture and also the implementation can be applied to any use case, it is essential to demonstrate its capabilities in a concrete one. In this thesis, the implementation is demonstrated with the ATM use case: pilot briefings with NOTAMs. It fits well with Big KG-OLAP as pilots always want to see a subset of the entire information during a single briefing. Hence, pilot briefings require to select a relatively small cube with information of interest out of the entire Big KG-OLAP cube, which is exactly the design goal of this Big KG-OLAP implementation. Pilot briefings are conducted with different data in the ATM domain. For simplicity, this thesis focuses on NOTAMs in AIXM format. The Big KG-OLAP lakehouse should be able to extract the context of NOTAMs in AIXM format and map them into RDF format. To solve that, an *AIXM Engine* for NOTAMs has to be implemented. An engine consists of an *Analyzer* and a *Mapper*. The *Analyzer* extracts the context out of a NOTAM while the *Mapper* transforms it into RDF statements. Also, the cube schema must be defined accordingly to support the pilot briefings use case.

5. Big KG-OLAP lakehouse reference architecture

The contributions of this master thesis are a reference architecture for a Big KG-OLAP lakehouse solution, a prototype implementation of this architecture and its performance evaluation. In this section, the reference architecture and its components are explained in detail as well as their interactions with each other. Afterwards, the two main functionalities *data ingestion* and *contextual operations* and their processing in the Big KG-OLAP lakehouse are described.

5.1. Architecture

Figure 5.1 shows the architecture of the Big KG-OLAP lakehouse. It consists of multiple components, each fulfilling a specific functionality. These components form a standalone system that is independent of external services. The cuboids represent the self-implemented stateless services *Surface*, *Circulator* and *Bed* that contain the business logic. The *Surface* is the connection point to external components, the *Circulator* performs background processing tasks and the *Bed* service constructs RDF graphs. These services require different stateful persistence services to store and manage data. A *Job queue*, an *Object storage*, a *Graph cache* and an *Index store* are used for that purpose. Each of the mentioned services must be horizontally scalable to support big data workloads. Apart from services, the Big KG-OLAP lakehouse requires the *Engines* to be installed in the *Circulator* and the *Bed* service and a *Cube schema* that defines the multidimensional model. On the top there are *Users, Applications and Streams* that interact with the system via the *Surface*. The arrows in the architecture represent the information flows.

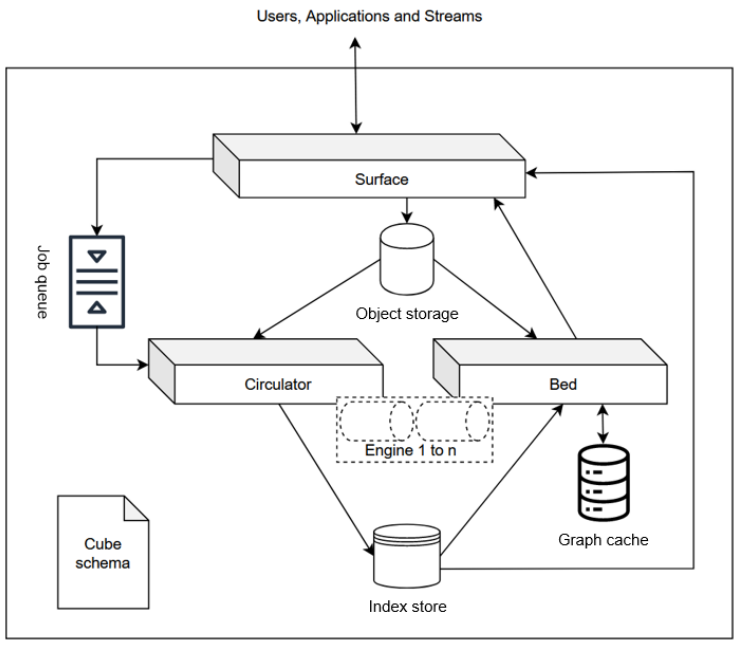


Figure 5.1.: Big KG-OLAP Lakehouse reference architecture

Surface. The *Surface* service is the single gateway to external services, users, applications, streams, etc. All request from outside the Big KG-OLAP lakehouse including *data ingestion* and *Context operation* requests arrive at the *Surface* service. Hence, it must provide an interface that offers both these main functionalities and other helpful services (e.g. *Cube schema* information) that can be used for different scenarios. An authentication protects the endpoint from unauthorized access. Requests are interpreted, handled or passed to other services, respectively and answered eventually. When the *Surface* receives new data objects *(data ingestion)* it stores the ingested data object in the *Object storage* with a generated unique identifier, reads metadata such as the data type from the requests and adds a new *Job queue* entry before it completes the requests. To answer *contextual operation* requests, the *Surface* parses the query, selects contexts from the *Index store* and requests the graph for each context from the *Bed* service before it aggregates the result and returns it to the requester.

Circulator. The *Circulator* service performs background tasks. Its main functionality is to periodically read new *data ingestion* jobs from the queue and process them. For each job, the *Circulator* reads the corresponding file from the *Object storage*, selects an installed engine based on the file metadata and uses it to extract the hierarchies out of it. Afterwards, it calculates the context of the hierarchies and stores the resulting contexts and indices in the *Index store*. Finally, the *Circulator* deletes the job from the queue and evicts the cache for the calculated contexts. The *Circulator* service is also the component to include future background processes such as data retention for instance.

Bed. The *Bed* service returns the graph for a given context. Similar to the *Circulator*, this service is hidden from the user and requires engines to be installed. However, the *Bed* service does not run as an asynchronous background process but rather as a synchronous one that is required to perform *Context operations*. It receives graph queries from the *Surface* service for a given context. The *Bed* service checks the *Graph cache* if the graph is cached and returns it if so. Otherwise, it selects the corresponding data object identifiers from the *Index store* and uses compatible engines to transform the data into RDF. Once every data object is transformed, it returns the resulting graph back to the *Surface*.

Index store. The *Index store* basically memorizes the contexts, data object metadata and its relationships. The relationship between *Data object* and *Context* is *M:N*. A single relation between a data object and a context is called index. The *Index store* acts as the data warehouse part in the Big KG-OLAP lakehouse architecture. Its logical design is multidimensional and consists of dimension tables and a "fact" table. But instead of facts such as numbers, the fact table contains data object identifiers that are used to find the data objects in the *Object storage*.

Object storage. The *Object storage* stores the data objects in raw format and form the data lake part of the system. The *Surface* receives new data objects and puts them into the *Object storage* with a generated unique identifier. Once stored, the data object is not changed anymore. The *Circulator* and the *Bed* service then read the data object from the *Object storage* on demand and process it using the compatible installed engine.

Job queue. The *Job queue* records new data ingestion events. When a new data object is ingested, the *Surface* service posts an event into the *Job queue*. This event or job is read by the *Circulator* service. The event is retained in the queue until it is explicitly removed by the *Circulator*. This ensures that no job is lost due to system failures.

Graph cache. The *Graph cache* temporarily stores graphs in RDF format for certain contexts. It receives new entries from the *Bed* service and provides cached ones to it. The purpose of the *Graph cache* is to provide a ready graph for a given context to the *Bed* so that it can answer its queries faster. If there is no graph cached for a given context, the *Bed* service has to read data objects from the *Object storage* and transform them into an RDF graph.

Engine 1 to n. The *Engines* are the ETL-adapters of the Big KG-OLAP lakehouse. The number of *Engines* installed in one Big KG-OLAP lakehouse is not limited logically. Each engine must be part of the *Circulator* and the *Bed* service. The *Circulator* uses the *Engines* to extract hierarchies out of data objects while the *Bed* service requires *Engines* to map data objects into RDF graphs. Hence, an *Engine* consists of two parts. A hierarchies extraction and an RDF mapping part. Moreover, each engine must have an associated data type so it can be linked to data objects.

Cube schema. The *Cube schema* defines the dimension, levels and hierarchies available in a Big KG-OLAP lakehouse deployment. Based on this configuration, the system can interpret *contextual operations* and infer hierarchies from members. The cube schema can be static or dynamic, while it is recommended to keep dimensions and levels static and hierarchies dynamic. If dimensions or levels are modified, the Big KG-OLAP lakehouse needs to be re-indexed. On the other hand, adding hierarchies requires no additional maintenance, while updating/deleting hierarchies results in minor migration steps.

Users, Applications and Streams. Any service or user that is capable and authorized to communicate with the interface provided by the *Surface* can interact with the Big KG-OLAP lakehouse. A widely adopted standard suitable for this interface is HTTP REST. Usually, data generation streams send *data ingestion* requests constantly with varying speeds. Applications and users employ *contextual operation* requests to gain interesting information.

5.2. Data ingestion

data ingestion is one of the two main functionalities of the Big KG-OLAP lakehouse. The system receives data objects from outside and ingests them. The ingestion includes storage and indexing. The indexing is required to build the multidimensional model that eventually enables *Context operations*. Figure 5.2 shows the *data ingestion* process in BPMN (Business Process Model and Notation) notation. Apart from the two services *Surface* and *Circulator* this process includes the *Job queue*, the *Index store* and the *Object storage*.

The process starts once the *Surface* receives a new data object through a designed interface (e.g., HTTP REST). Next, it reads the metadata and checks if there is an *Engine* installed for the given data type. If not, the process ends and the requester receives an error message. Otherwise, the *Surface* generates a unique identifier and uses it to store the data object in the *Object storage* as every data object in the *Object storage* requires a unique identifier. As a last task, the *Surface* registers the unique identifier of the data object as a new job in the *Job queue* and completes the (HTTP) request. The following processing steps happen asynchronously to the *data ingestion* request.

The *Circulator* periodically checks the *Job queue* for new jobs. Once it finds a new job, it reads the job information. From that point on, the job must be invisible to other *Circulator* instances to avoid duplicate processing. Based on the job information, the corresponding data object is read from the *Object storage*. Next, the *Circulator* selects the compatible *Engine* for the given data object and utilizes it to extract the hierarchies. A valid *Hierarchy* is given when there is no missing value for any parent level. For instance, the hierarchy *"2022:year → 2022-02:month → [EMPTY]:day"* is valid while *"2022:year → [EMPTY]:month → 2022-02-01:day"* is not. One data object may contain multiple hierarchies per dimension. For example, one NOTAM can be valid for multiple days or across multiple locations. After hierarchy extraction, the Cartesian product is calculated according to Figure 5.3, where each hierarchy of one dimension is combined with each hierarchy of every other dimension. The example shown in the figure contains three hierarchies for the *Location* dimension and also three hierarchies for the *Time* dimension. In total, this results in nine different contexts which are stored in the *Index store* right after together with the indices. An index is the relationship between a data object and a single context. Subsequently, the cached graphs from the *Graph cache* for every context found in the data object is evicted as the graph misses the information from the new data object hence is not valid anymore. At the end of the *data ingestion* process, the *Circulator* removes the job from the *Job queue*. If the process is not successful or takes too long, the job gets visible again to other *Circulator* instances. This concept ensures job durability and fault tolerance.

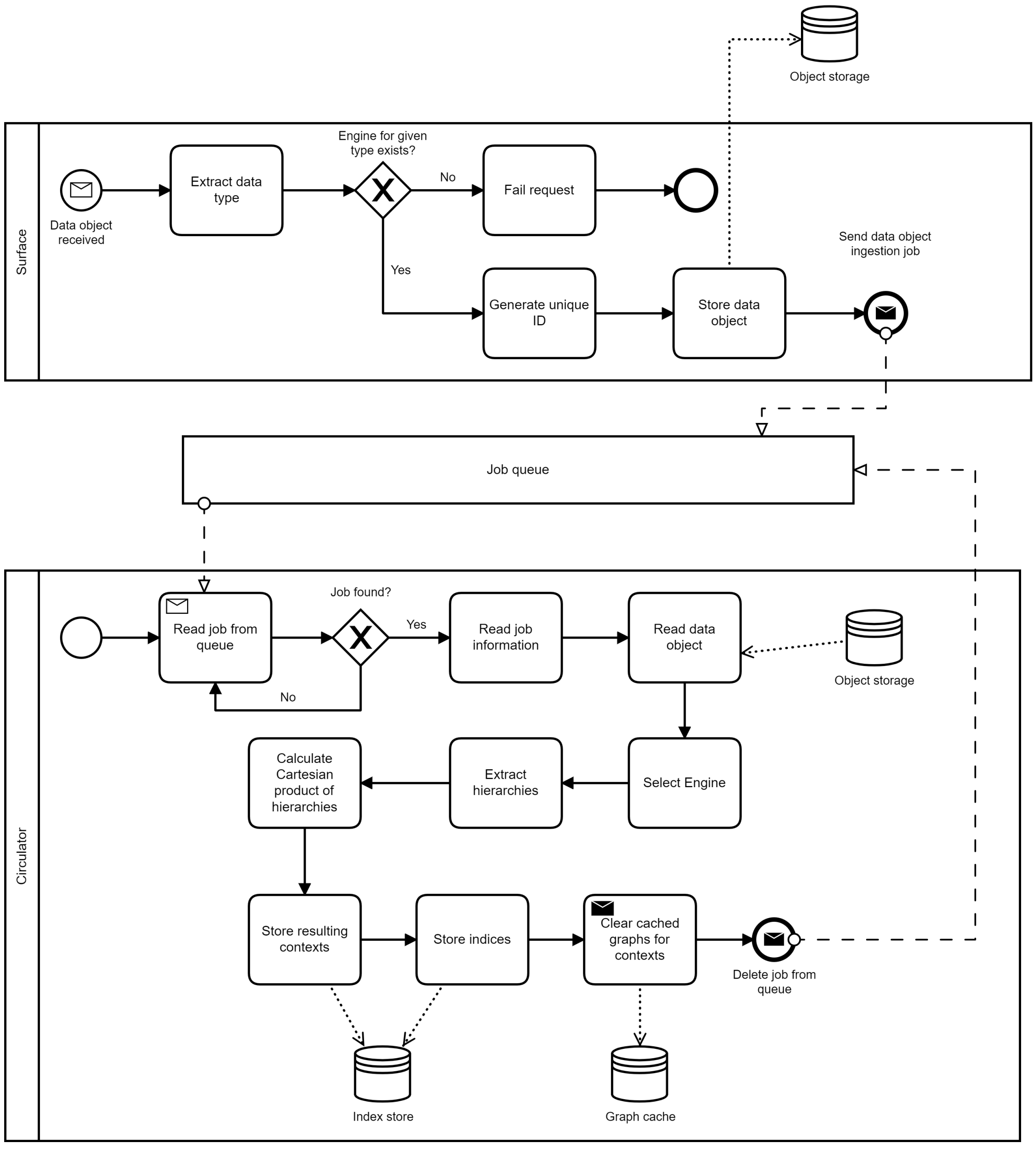


Figure 5.2.: Data ingestion process

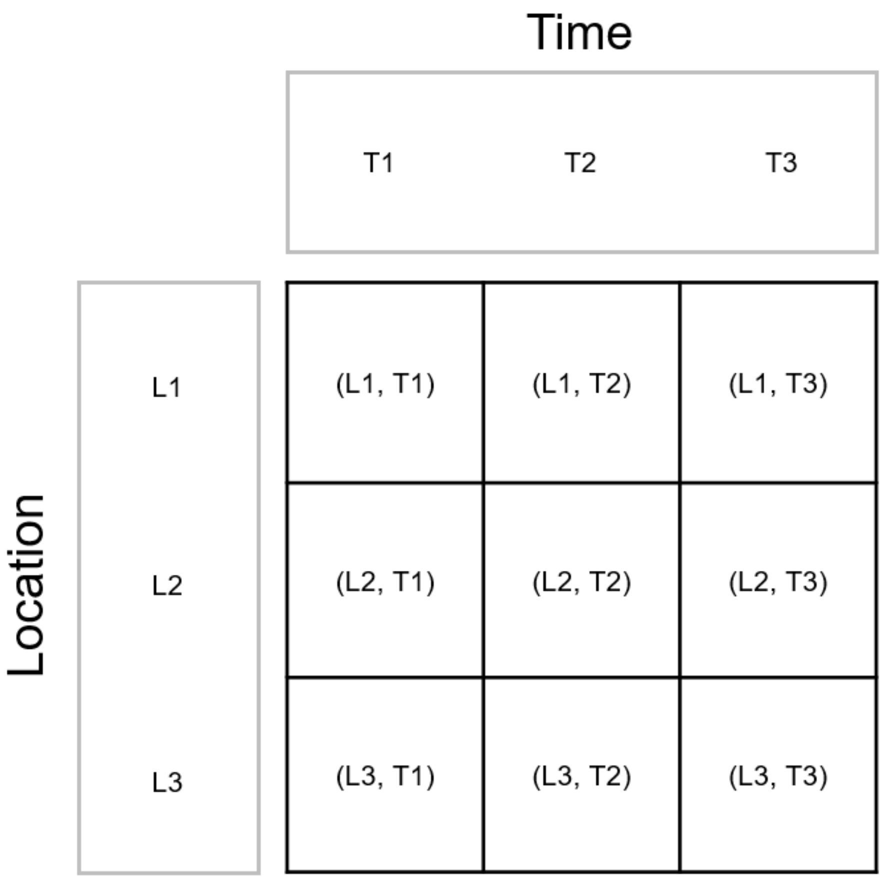


Figure 5.3.: Cartesian product of *Location* and *Time* hierarchies

5.3. Contextual operations

Interpreting and answering *contextual operation* queries is the second main functionality of the Big KG-OLAP lakehouse. The *contextual operations* are *Slice'n'Dice* and *Merge* while both operations are part of the same KGOQL query. Similar to SQL, the *Slice'n'Dice* would be the "WHERE" clause and *Merge* corresponds to the "GROUP BY" clause, respectively. Also, as already described in section 3, *Merge* is performed after *Slice'n'Dice*. Figure 5.4 demonstrates the *contextual operation* querying process from the reception of the KGOQL query until the return of the resulting RDF cube. A consistent example is illustrated throughout this section for a better comprehension.

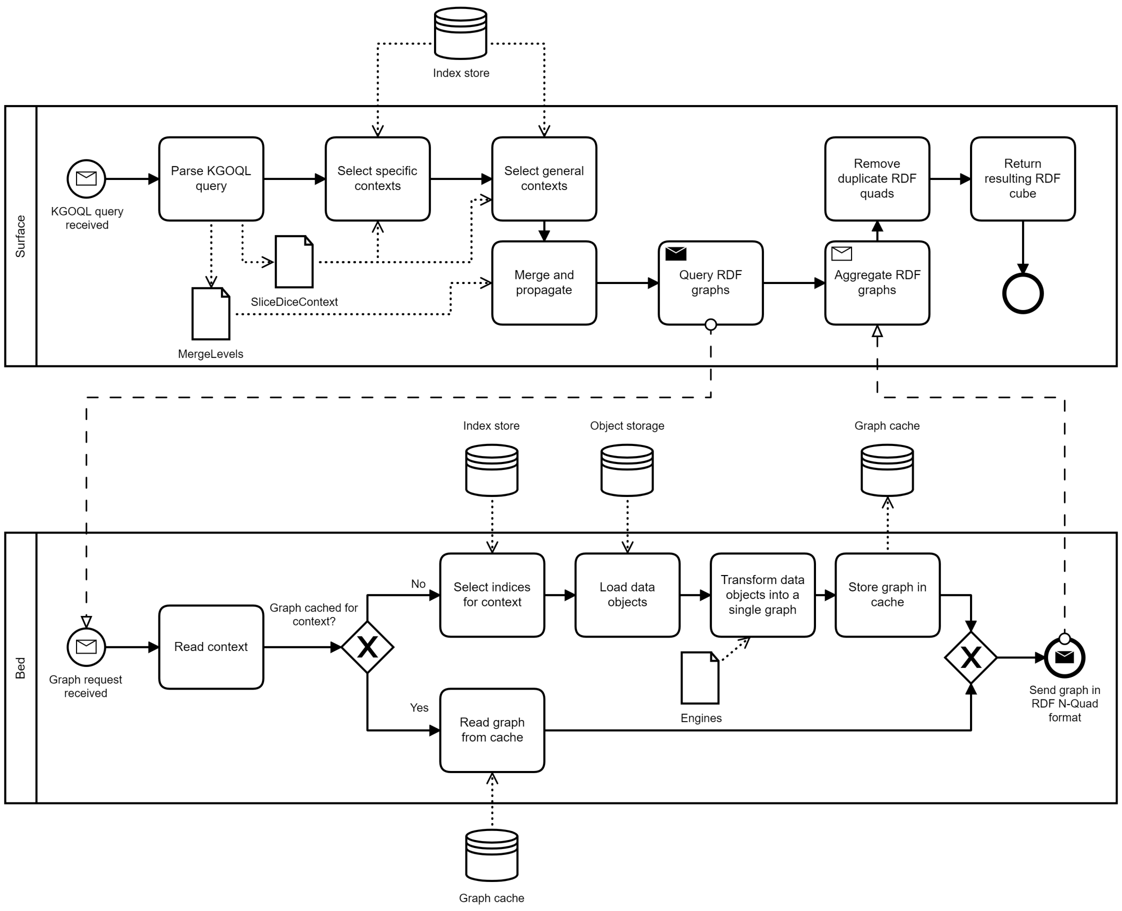


Figure 5.4.: Contextual operations process

The goal of *contextual operations* is to query an RDF cube of the Big KG-OLAP lakehouse. For that, the system must contain data and in this example, the stored model consists of five different contexts and data objects associated to them as shown in Figure 5.5. Contexts as well as the data objects (can be one or many) associated to it are labeled as *C1* to *C5*. The multidimensional model has two dimensions namely *Location* and *Time* at different levels. The *Location* dimension has three levels *Country*, *City* and *Organization* while the *Time* dimension includes *Year*, *Month* and *Day*. The arrows in the figure represent the hierarchical relationships between the contexts. For instance, *C2* inherits from *C1* because both of its members *"2022-02:month"* - *"LNZ:city"* roll up to the members of *C1*. *"2022-02:month"* rolls up to *"2022:year"* and *"LNZ:city"* rolls up to *"AUT:country"*. Another example is that *C5* inherits from *C2*, because *"JKU:organization"* rolls up to *"LNZ:city"* while the time dimension is on the same level with the same value *"2022-02:month"*. On the contrary, *C5* does not derive from *C3* because *"LNZ:city"* does not roll up to *"SBG:city"*.

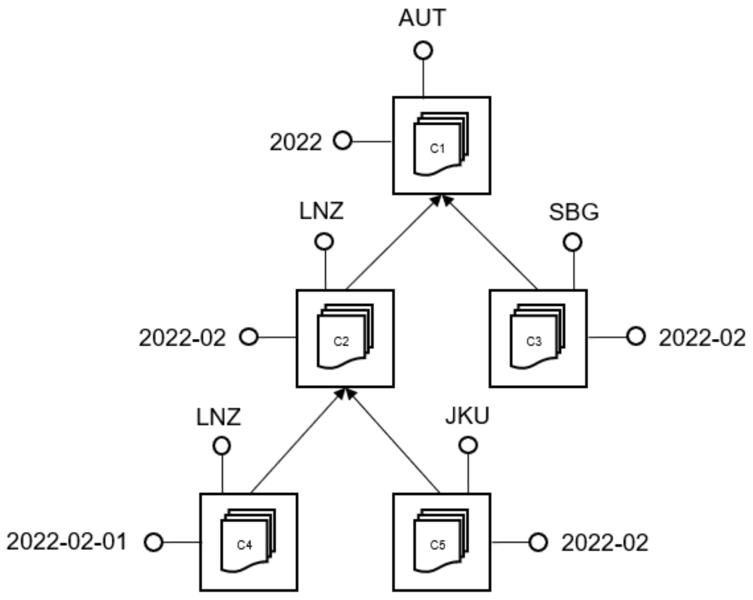


Figure 5.5.: Stored cube example

The process starts at the *Surface*. The service receives a KGOQL query from an external service, user, application or similar. The query used in this example is shown is listing 5.1.

Listing 5.1: KGOQL query

```
SELECT time_month=2022-02 AND location_city=LNZ ROLLUP ON time_month
```

Parse KGOQL query. At first, the query parser extracts the *SliceDiceContext* and the *MergeLevels* out of the query. Both are required for further steps. The *SliceDiceContext* defines the exact coordinate to select the subcube from. In the given example, it is *"2022-02:month" - "LNZ:city"*. The *MergeLevels* are a map from dimensions to levels, in this example *time:month*.

Select specific contexts. After query parsing the next step in the process is to select the specific contexts from the *Index store*. For that, the *Surface* constructs a query that can be understood by the *Index store* based on the *SliceDiceContext*. Figure 5.6 shows which contexts are then returned by the query. Every context at or below the *SliceDiceContext* is returned.

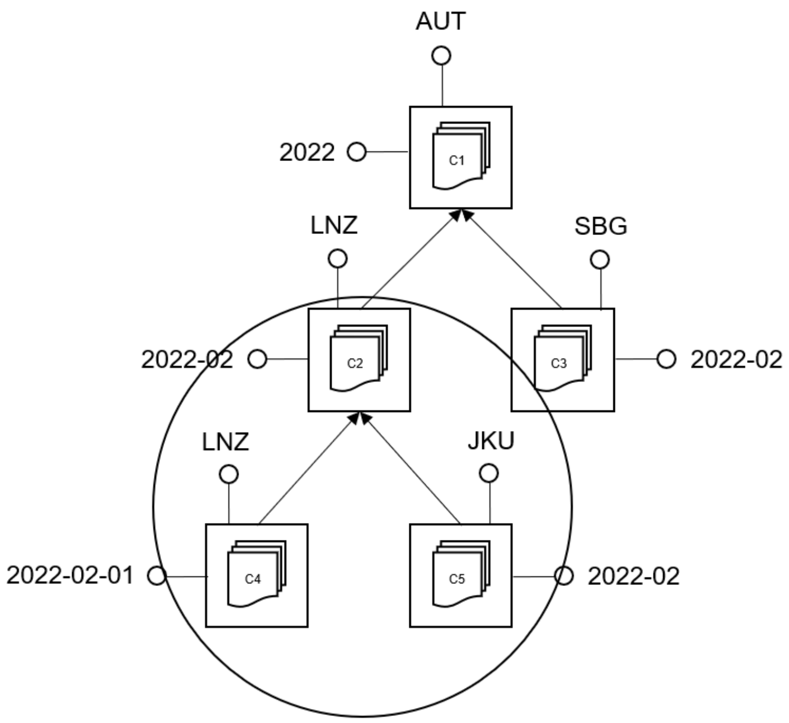


Figure 5.6.: Select specific contexts example

Select general contexts. Due to the functional requirement of knowledge propagation, it is necessary to include knowledge from parent contexts. To achieve that, general contexts have to be selected from the *Index store* similar to the selection of specific ones. The difference now is that only contexts above the *SliceDiceContext* are returned by the query. The selection is shown in Figure 5.7.

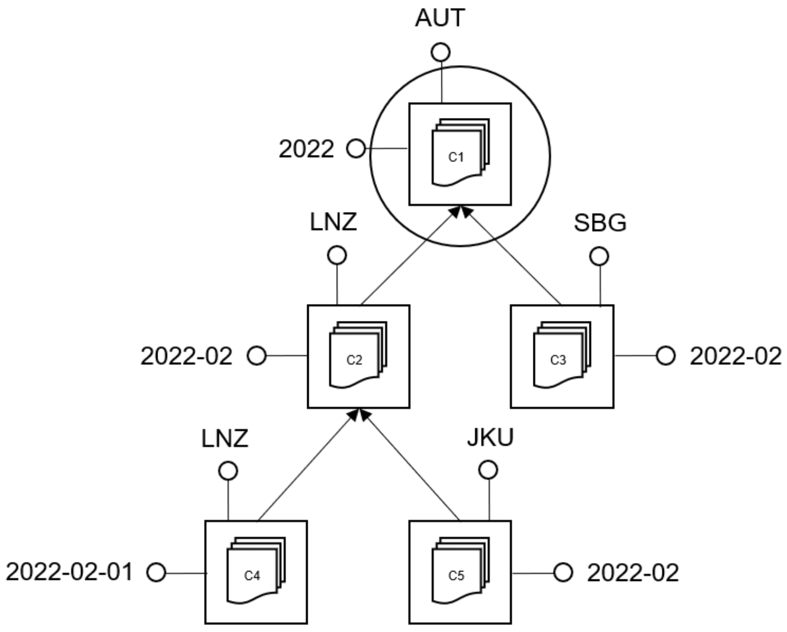


Figure 5.7.: Select general contexts example

Merge and propagate. Once every relevant context is selected from the *Index store*, the result structure can be calculated. This includes two steps within one task. These steps are *Knowledge propagation* and *Merge*. This results in a mapping from each relevant context to a final context. A final context or final graph is one cell in the resulting RDF cube. Both steps are drawn as a combined task as they happen together per context in an iterative process. For every specific context, every context that rolls up to it as well as itself is rolled up to the *MergeLevels* and registered as a final context. The result is a map of specific contexts to final contexts. After that, every general context is mapped to every final context as well because every specific context per definition inherits from every general context. Figure 5.8 shows the result of this phase for the given example. It is a mapping from existing contexts to final contexts.*C2* does not inherit knowledge from another specific context (*C4* or *C5*) and can not be rolled up. Hence, the existing context *C2* is mapped to the final context *"2022-02:month" - "LNZ:city"*. Next, *C4* does inherit knowledge from *C2* and can be rolled up to *"2022-02:month" - "LNZ:city"*. Therefore, *C4* and *C2* are mapped to the final context *"2022-02:month" - "LNZ:city"*. As this final context already contains *C2*, the final context is not added again. *C5* also inherits knowledge from *C2* and can be rolled up to *"2022-02:month" - "JKU:organization"*. Hence, *C5* and *C2* are mapped to the final context *"2022-02:month" - "JKU:organization"*. As a last step, the general context *C1* is mapped to both final contexts *"2022-02:month" - "LNZ:city"* and *"2022-02:month" - "JKU:organization"*.

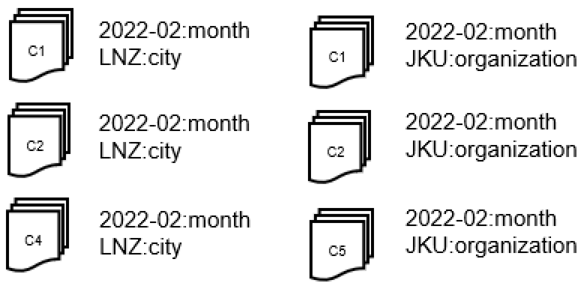


Figure 5.8.: Merge and propagate example

Query and aggregate RDF cube. For every specific and general context one RDF graph request is sent to the *Bed* service together with its associated final graphs. For every graph request, the *Bed* service reads the contexts and checks the *Graph cache* if an entry for the given context exists. If yes, it uses the cached graph to answer the query faster. Otherwise, the *Bed* service selects the indices associated to the contexts from the *Index store*, loads the data objects and transforms them into a single RDF graph. Every data object is transformed separately with the compatible *Engine* which is selected based on the data type. The resulting graph is stored in the *Graph cache* to temporarily answer subsequent queries for the given context faster. Now, regardless if the graph is read from the cache or constructed from the data objects, it is returned to the *Surface* in RDF N-Quads format. This format is line based and independent of any line order, which enables asynchronous parallelization hence improves performance. The *Surface* then simply aggregates all the lines it receives from the *Bed* services, drops duplicates and returns the resulting RDF cube in N-Quads format to the requestor.

In the given example, the *Surface* service sends four different requests (*C1*, *C2*, *C4*, *C5*) to the *Bed* service. The *Bed* service constructs the RDF triples or reads it from the cache and then creates a graph for every given final context. In case of *C1*, *C2* this results in two RDF graphs one named *"2022-02:month" - "LNZ:city"* and the other one is *"2022-02:month"*

- *"JKU:organization"*. For *C4* and *C5*, only one graph *"2022-02:month" - "LNZ:city"* and *"2022-02:month" - "JKU:organization"* respectively is created. The RDF triples and resulting graphs are used to construct RDF quads which are then sent back in N-Quads format to the *Surface* where they are combined, resulting in an RDF cube with two named graphs.

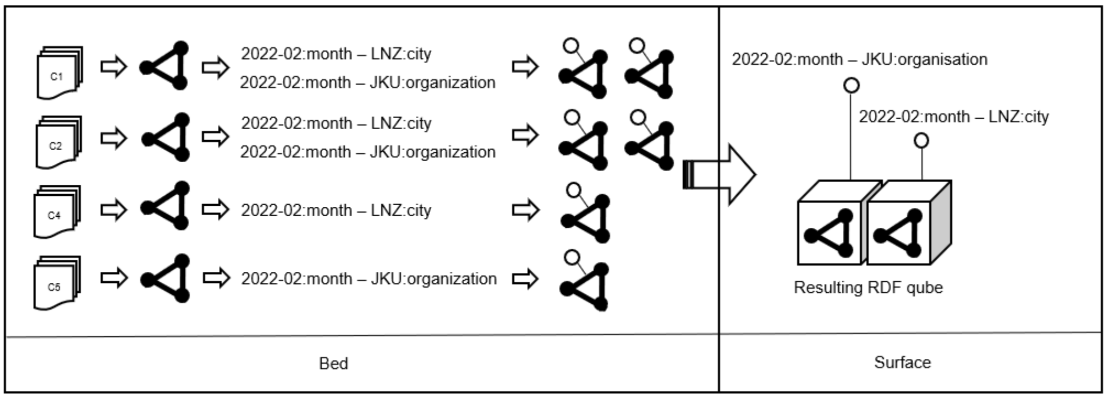


Figure 5.9.: Query and aggregate cube example

6. Cloud native prototype implementation

The second contribution of this thesis is the prototypical implementation of the proposed Big KG-OLAP lakehouse reference architecture. This section discusses the overall implementation architecture in detail. The implementation of the two main functionalities *data ingestion* and *contextual operations* is also part of this section. The resulting prototype is the basis for future Big KG-OLAP research efforts. Hence, it is important that the resulting artifact is maintainable, extensible and portable. For instance, external services must not be tightly coupled to the code but should rather be subject to configuration. In general, the implementation follows cloud-native application design principles. This modern approach to develop applications ensures portability and operation in state of the art cloud environments.

6.1. Architecture

Figure 6.1 shows the cloud-native prototype implementation architecture of the Big KG-OLAP lakehouse. Its structure is designed to match the reference architecture but with the obvious difference that concrete technologies replaced general components. The prototype is developed with and deployed on Amazon Web Services (AWS) infrastructure. The self-implemented core of the Big KG-OLAP lakehouse, the three services *Surface*, *Circulator* and *Bed* are deployed on the AWS managed Kubernetes service *Amazon EKS* (Elastic Kubernetes Service). All other required services are rented from AWS. The deployment on AWS infrastructure allows performance evaluation of the prototype without physical hardware and saves a lot of configuration and setup work. GitHub[1] is used as version control system to manage the source code of the services, engines, shared components, configuration (including the YAML *Cube schema*) and deployment specification. The

[1]https://github.com/

repository also contains an Angular UI (user interface) and two Java applications for performance testing.

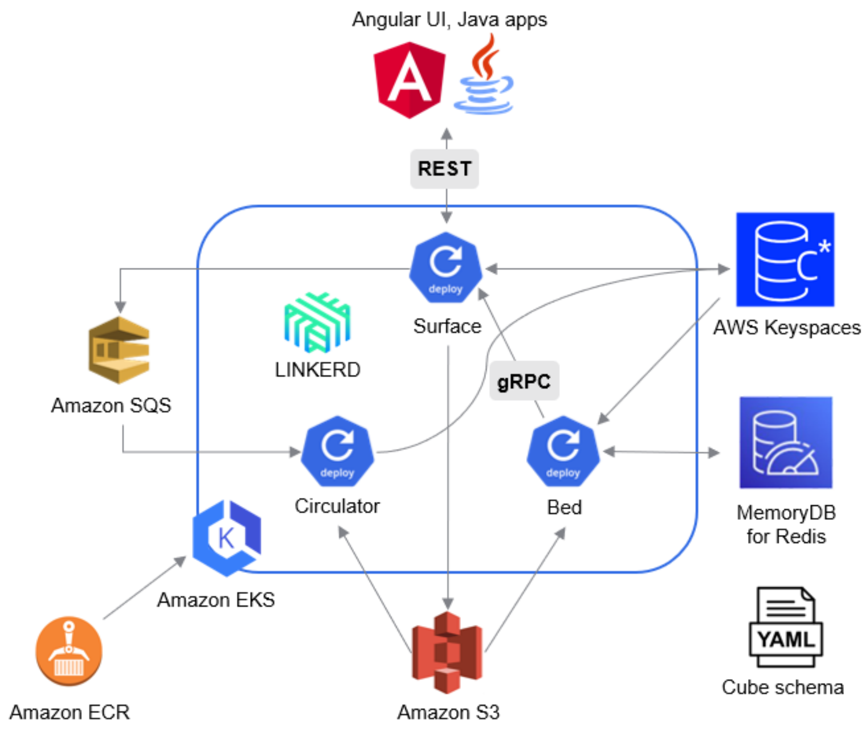


Figure 6.1.: Cloud-native implementation architecture

Amazon EKS and Linkerd. Amazon EKS is a managed Kubernetes service provided by Amazon Web Services. Kubernetes clusters can be created and configured via the AWS console. A cluster requires nodes to handle the workload. Based on computing demand, a cluster has different types and quantities of nodes. For the performance evaluation conducted in this thesis, a single node is used. This is further explained in section 7. Once the cluster is up and running, a command line interface (CLI) allows to manage Kubernetes resources remotely. Before deploying the services *Surface*, *Circulator* and *Bed* additional installations are required for load balancing. The *Surface* provides a REST endpoint while the *Bed* exposes a gRPC service. As both services must be horizontally scalable, load balancing is required. For instance, if two *Surface* instances are running, both processes should receive a similar amount of requests. The same behavior is required for the *Bed* service and the *Web UI*. The *Circulator* has no interface, hence no load balancing is required for this service. Load balancing HTTP/1.1 REST requests from outside the

EKS cluster requires an application load balancer (ALB) that routes traffic on the ISO-OSI model application layer. AWS provides a documentation how to install that. For cluster internal gRPC load balancing, a different approach is required. Kubernetes offers the so-called *Service* resource for automatic service discovery and load balancing by assigning a single DNS name to a set of pods. However, this is a connection level load balancing strategy that does not work for gRPC. The gRPC protocol is built on HTTP/2 which reuses connections unlike HTTP/1.1. This is solved by installing the service mesh *Linkerd* on the Kubernetes cluster which automatically performs application level (ISO-OSI level 7) load balancing. The Amazon EKS deployment configuration used to deploy the *Surface*, *Circulator*, *Bed* and *Web UI* is included and described in appendix B.2.

Surface. The *Surface* service is one of three self implemented Spring Boot based services. This Java application contains a web server and is the entry point for external requesters. A servlet offers HTTP REST endpoints for *data ingestion*, *contextual operations* and other functionalities. Figure 6.2 shows the class diagram of the essential parts of the *Surface* service as well as shared classes such as *Context* for better comprehension. The *LakehouseC-ontroller* is the servlet that contains the HTTP REST endpoint methods. For *data ingestion*, the servlet receives a *MultipartFile* object which contains the data and a *HttpServletRequest* object with metadata including the data type. Those are not part of the diagram as they are no self-implemented classes. The *MultipartFile* is directly stored in the object storage using the *StorageService* and a new data ingestion job is posted into the *MessagingService*. *StorageService* is the interface to access the *Object storage* while the *MessagingService* is the bridge to the *Job queue*. Also, file metadata such as the original name and its size are persisted using the *Context Service*. The *DatabaseService* is the interface to read from and write to the AWS Keyspaces database. Besides the functionalities of the *Index store* it is used for arbitrary features such as file metadata storage or logging. However, these features are no essential part of the reference architecture, hence are omitted in section 5. For *contextual operations*, the servlet receives a *CubeRequest* containing the KGOQL query as *String* and passes it directly to the *QueryService*. The *QueryService* has a *QueryParserService* which extracts the *SliceDiceContext* and *MergeLevels* out of the query. If the query is invalid, the *QueryParserService* throws an *InvalidQueryException* instead. The *SliceDiceContext* is used to query the specific and general *Context* objects from the *DatabaseService*. Together with the *MergeLevels* these contexts are the input for the *MergeAndPropagateService* which returns a *MergeAndPropagateResult*. For every entry in this result the *GraphQueryServiceClient* is called which is the client interface to query graphs from the *Bed* service. If a request to the

Bed takes longer than a configurable timeout for any reason, a *GraphQueryTimeoutException* is thrown. Finally, the *QueryService* creates a *CubeResult* which is returned from the servlet to the requester. Besides these classes there is a security configuration and a simple user management to protect the endpoints from unauthorized access.

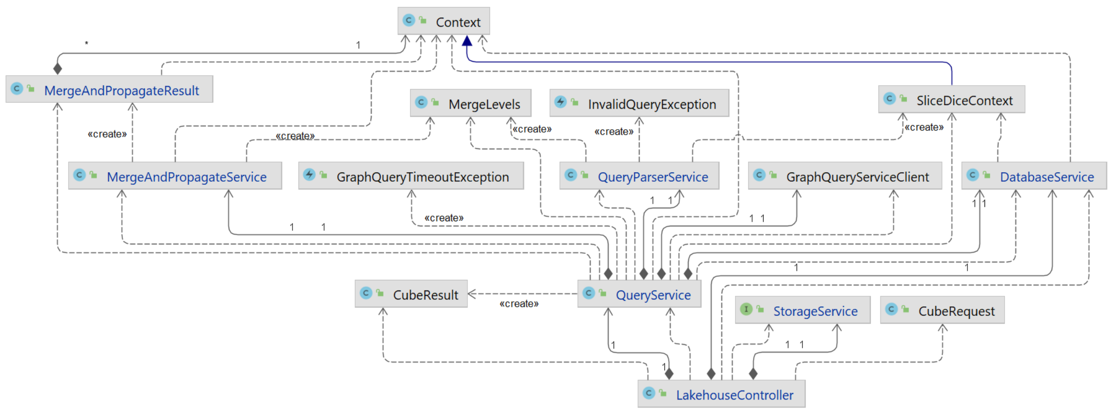


Figure 6.2.: Spring Boot service *Surface* class diagram

Circulator. The *Circulator* service is a Spring Boot services that performs background processing tasks. Its main purpose is to periodically check the *data ingestion* queue and process jobs. Not implemented background processes such as data retention would also fit best in the *Circulator* service. Figure 6.3 illustrates the class diagram of the service's implementation and tightly coupled shared classes such as *Engine* or *Context*. The main class is the *DataIngestionScheduler* which has a method that is scheduled to run periodically. Within every method run, the *MessagingService* is checked if new data ingestion jobs are available. If not, the method returns and runs again in a configurable interval. Otherwise, the job is processed and the method recursively called again, as it is assumed that the queue is not empty. To scale vertically, one method run does not only process one job but a configurable number *BATCH_SIZE* (5 currently). Every job is then processed simultaneously with multi-threading. Of course, the number of jobs that can be processed in parallel is limited by the number of available processors. The *ContextExtractionService* gets the file information as input, reads it from the *StorageService*, selects the compatible *Engine* and creates a set of *Context* objects. These objects together with its associated file indices are upserted in the *DatabaseService* and the cache entries for these contexts are evicted via the *GraphCache* interface. An *Upsert* is a combination of *Insert* and *Update* which enables parallel context insertion in this case. With a simple *Insert* it would not be possible that two simultaneously ingested files share the same context as one could not be persisted in the database.

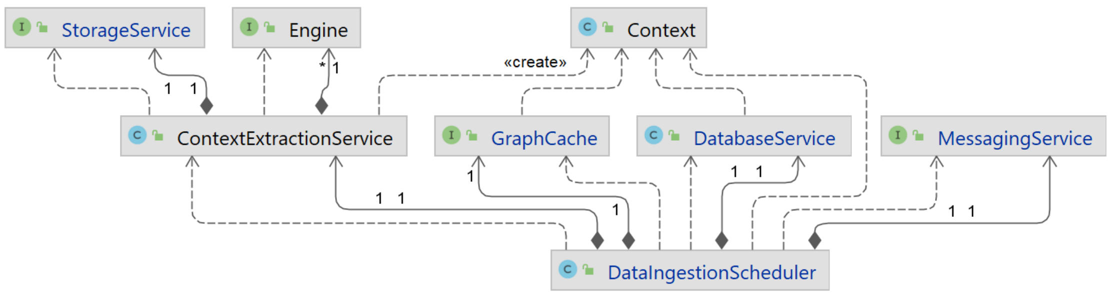


Figure 6.3.: Spring Boot service *Circulator* class diagram

Bed. The *Bed* service is the third Spring Boot service that returns RDF quads for a given context identifier and final graphs. For that, it provides a gRPC service that is consumed by the *Surface* service. The gRPC protocol is used over HTTP REST because it is more efficient for high throughput due to connection re-usage and lightweight protobuf messages. Also, it allows to compress transferred data which is especially useful for RDF quads that contains repeating characters. Listing 6.1 contains the shared protobuf definition that defines the service, the request and response format. The request message starts in line 6 and it contains the unique identifier of the current *contextual operation* for tracking purposes, the given context ID and a set of final graphs to which the resulting RDF triples are associated. The response starting in line 12 simply is a set of quads, where each quad is represented as an independent line in RDF N-Quads format. Line 16 to 18 finally defines the gRPC service. It is important to mention that it returns a *stream* of response objects, as the *Bed* service returns the resulting RDF quad lines in chunks to avoid network overload and keep the memory footprint low. The chunk size is configured based on gRPC best practices found during a web search.

Listing 6.1: gRPC GraphQueryService protobuf definition

```
1  syntax = "proto3";
2  option java_multiple_files = true;
3
4  package at.jku.dke.bigkgolap.shared.grpc;
5
6  message GraphQueryRequest {
7    string queryUuid = 1;
```

```
  string contextId = 2;
  repeated string graphs = 3;
}

message GraphQueryResponse {
  repeated string quads = 1;
}

service GraphQueryService {
  rpc queryGraph(GraphQueryRequest) returns (stream GraphQueryResponse);
}
```

Figure 6.3 shows the class diagram of the *Bed* implementation as well as important dependent classes. The *GraphQueryServiceImpl* class extends from a generated class based on the protobuf definition. An overridden method forms the entry point of the gRPC request hence the connection point of the *Surface* service. The *GraphService* checks the *GraphCache* if the connected *MemoryDB for Redis* has an RDF graph cached for the given context ID. If not, the *GraphService* continues and queries the *LakehouseFile* objects from the *DatabaseService*, which are the files associated to the context ID. These *LakehouseFile* objects are passed to the *FileLoaderService* which selects the compatible *Engine* for every single file object and uses it to transform the file content into an RDF graph. Finally, the *GraphQueryServiceImpl* iterates the triples of the resulting RDF graph, generates a quad for every triple and every final graph before it returns the final RDF quads back to the *Surface*.

AWS Keyspaces. This prototype implementation uses the Cassandra compatible service *AWS Keyspaces* as the *Index store*. Cassandra was chosen as an *Index store* implementation over relational databases such as PostgreSQL due to its distribution by design and its alignment for big data. *AWS Keyspaces* is a managed Cassandra services that requires a low amount of configuration hence it is a good fit for a proof of concept. The service employs a pay-per-use model and scales automatically in the background. The logical design of the database is similar to a star schema in data warehousing. However, the NoSQL database Cassandra does not support relations nor supports complex queries that do not include the primary key. Hence, the data model has to be designed differently while it is optimized for fast reads. Writes are always fast in Cassandra as it is an append-only database. Listing

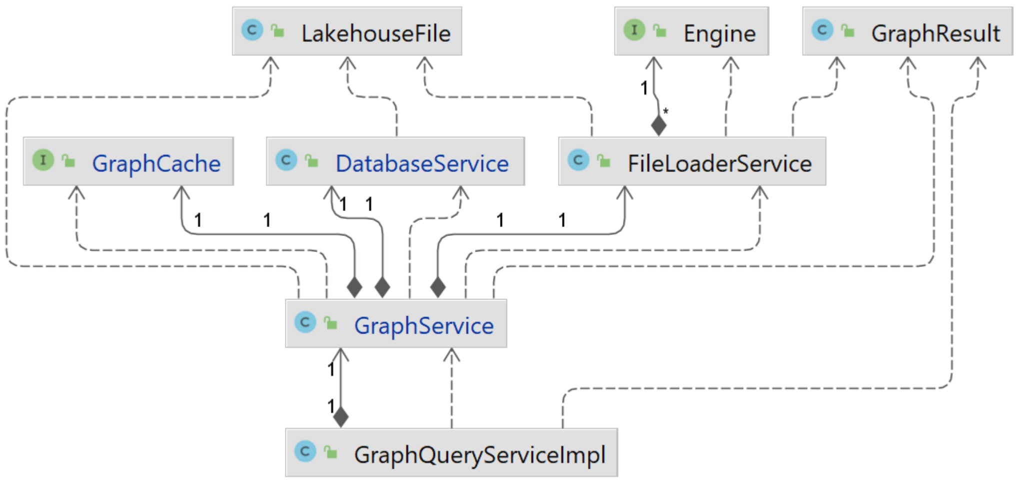


Figure 6.4.: Spring Boot service *Bed* class diagram

6.2 shows relevant parts of the logical data model in CQL (Cassandra query language). First, the keyspace *lakehouse* is created and replicated within a single AWS region. This is the default replication strategy. This keyspace contains a table for every dimension as well as two additional tables for storing the "facts", which are file identifiers (as stored names). The "fact" table is split up into two tables so that the *Surface* can upsert the file details in the database before the *context_hash* is known. Every insert or update in Cassandra results in an upsert and it is mandatory that the query includes every part of the primary key. The primary key consists of a *PARTITION_KEY* and a *CLUSTERING_KEY*. The first attribute of the primary key is the partition key while the remaining parts form the clustering key. Because the *context_hash* is part of the primary key in the *lakehouse.files* table, file details could not be inserted by the *Surface* before the *Circulator* successfully calculates the context and upserts the index (= combination of context and file). The data model employs hashes to support simultaneous upserts and reads. Every hierarchy (=entry in a dimension table) as well as every context has a unique hashed based on its values. A context hash is the hash of the associated hierarchies. If two *Circulators* ingest two independent files with the same context it would result in the same hierarchy and context hashes. The latest upsert is going to be the eventual value but because both upserts set the same values it does not matter. Only the two inserted indices in the *lakehouse.files* table will stay, both having the same context hash.

Each dimension table is designed similarly. The partition key is always the highest level of the dimension and Cassandra uses it to shard a table across multiple nodes. The remaining levels are used as clustering keys in hierarchical order. Cassandra sorts entries within one partition based on them. In case of *TIME* it makes sense to sort descending by *month* and *day* to find more recent entries faster. A special characteristic in this design is that there is no dedicated table necessary that stores the context. Each hierarchy rather knows to which contexts it is associated to and on querying, the intersection of the context hashes of each hierarchy is the set of existing context hashes. These context hashes are used later on to query files from the *lakehouse.files* table, which works because the context hash is the partition key in this table. It is worth to mention that both file tables contain the *engine_type* which is a designed redundancy to save requests.

Listing 6.2: AWS Keyspaces Cassandra data model excerpt in CQL

```
CREATE KEYSPACE IF NOT EXISTS lakehouse WITH REPLICATION = { 'class' : '
    SingleRegionStrategy' };

CREATE TABLE IF NOT EXISTS lakehouse.time
(
    year INT,
    month VARCHAR,
    day BIGINT,

    hash VARCHAR,
    context_hashes SET<VARCHAR>,

    PRIMARY KEY (year, month, day)
) WITH CLUSTERING ORDER BY (month DESC, day DESC);

CREATE TABLE IF NOT EXISTS lakehouse.location
(
    territory VARCHAR,
    fir VARCHAR,
    location VARCHAR,

    hash VARCHAR,
    context_hashes SET<VARCHAR>,

    PRIMARY KEY (territory, fir, location)
);

CREATE TABLE IF NOT EXISTS lakehouse.topic
(
    category VARCHAR,
    family VARCHAR,
    feature VARCHAR,

    hash VARCHAR,
    context_hashes SET<VARCHAR>,

    PRIMARY KEY (category, family, feature)
);

CREATE TABLE IF NOT EXISTS lakehouse.files
(
    context_hash VARCHAR,
    stored_name VARCHAR,

    engine_type VARCHAR,

    PRIMARY KEY (context_hash, stored_name)
);

CREATE TABLE IF NOT EXISTS lakehouse.file_details
(
    stored_name VARCHAR,

    engine_type VARCHAR,
    original_name VARCHAR,
    size_bytes bigint,
    PRIMARY KEY (stored_name)
);
```

Amazon S3. Amazons Simple Storage Service (S3) acts as the *Object storage* in the Big KG-OLAP lakehouse prototype implementation. It is highly scalable, cost-efficient and provides a simple API. *Amazon S3* mainly operates with the two concepts *Bucket* and *Key*. On data ingestion, a unique identifier is calculated for the file. The identifier is the *Key* under which the file is then stored in the configured *Bucket*. From a code-perspective shown in Figure 6.5, there is a *S3StorageService* class which implements the *StorageService* interface. The *LocalStorageService* was implemented for local development and uses the file system underneath and mocks an "object storage". Based on the active Spring profile, a different implementation is used. This approach also allows to replace the *Amazon S3* implementation with other *Object storage* solutions easily. In future efforts, a tiered storage service could be implemented where files are fetched from *Amazon S3* and temporarily stored on the local file system to increase read performance.

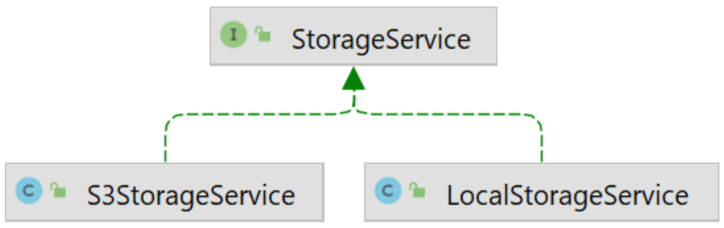


Figure 6.5.: *StorageService* hierarchy class diagram

Amazon SQS. The Amazon Simple Queue Service (SQS) is used as *Job queue* for *data ingestion* jobs. Similar to *Amazon S3* the *Amazon SQS* service is also scalable, cost-efficient and reliable using a similar API. The *Surface* sends events about new *data ingestion* jobs via HTTP to a configured *Amazon SQS* queue where the *Circulator* reads it from. A message contains the file *Key* as well as the type. The *Circulator* then simply requests the file with the given *Key* from the configured *Bucket*, selects the *Engine* based on the given type and performs the context extraction afterwards. The file type is part of the message to save database read requests.

Amazon MemoryDB for Redis. The key-value in-memory cache technology Redis[2] was chosen as *Graph cache*. Amazon offers a service called *Amazon MemoryDB* which is a

[2]https://redis.io/

managed Redis service with durable storage. The Java library *Jedis*[3] provides straight forward operations on a Redis cluster. Graphs are cached per context. The context identifier is the *key* in the key-value cache while the graph is the *value*. Graphs are stored in *RDF Thrift* format which is an efficient binary data encoding format for RDF [48]. Figure 6.6 contains the class diagram of the *GraphCache*. Depending on the active Spring profile, either a configured Redis endpoint is used in *RedisCache* as the graph cache or a placeholder implementation *NoCache* is activated that does not do anything. The *NoCache* class is used to turn off graph caching.

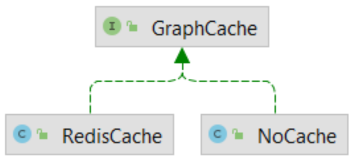


Figure 6.6.: *GraphCache* hierarchy class diagram

Engine architecture and AIXM implementation. The *Engines* are the ETL-Adapters of the Big KG-OLAP lakehouse. In this prototype implementation, *Engines* are Java libraries that are included as runtime dependencies in the Spring services. The Spring services then use the *java.util.ServiceLoader* class to load every *Engine* implementation on the classpath and instantiates them. All *Engine* objects are then stored in a map together with an identifier as the key. The key matches data types so that services can search the capable *Engine* in the map based on a given data type. Figure 6.7 shows the class diagram of the *Engine* architecture and the concrete *AixmEngine* implementation. The *Engine* interface has three methods, one to get the unique identifier to match it with data types, one to get the *Analyzer* and the third one to retrieve the *Mapper* instance. Generics ensure type safety. The *Analyzer* is the part of the *Engine* that extracts the context out of a data object. Its return type *AnalyzerResult* consists of a set of *Hierarchy* objects with which the Cartesian product is calculated in a later step to get the final set of contexts. The *Mapper* gets an intermediate Apache Jena[4] *Model* as input and it uses it to create and append RDF resources. Apache Jena is an open source Java framework to build graph based applications. While both services *Circulator* and *Bed* use the *Engine* architecturally, the *Circulator* only utilizes the *Analyzer* and the *Bed* makes use of the *Mapper*.

[3]https://github.com/redis/jedis
[4]https://jena.apache.org/

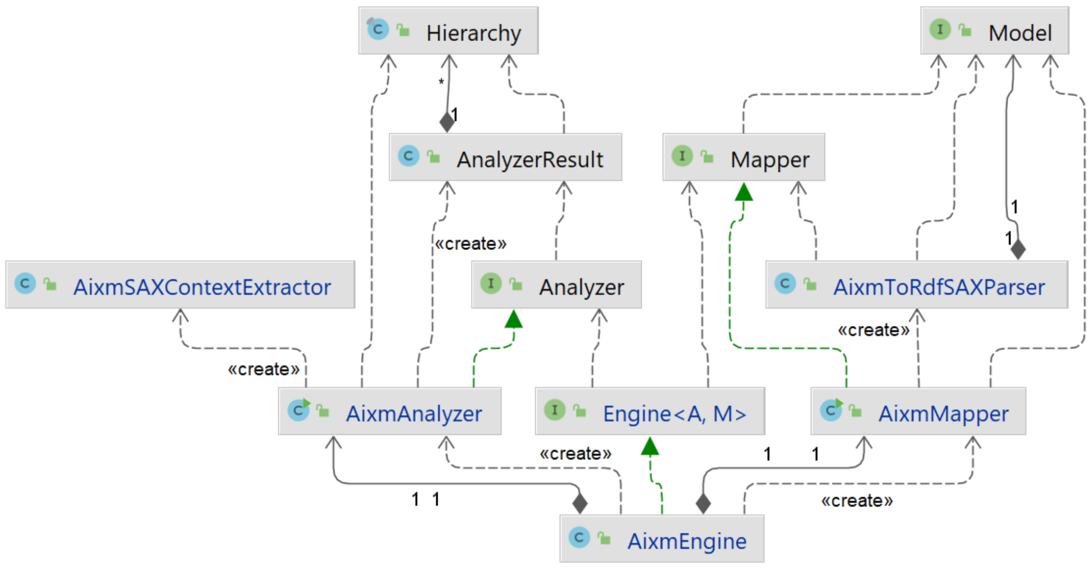


Figure 6.7.: *Engine* and *AixmEngine* class diagram

Ingested digital NOTAMs or DNOTAMs are processed by the *AixmEngine*. Both the *AixmAnalyzer* and the *AixmMapper* are based on a SAX parser (Simple API for XML). The *AixmSAXContextExtractor* used by the *AixmAnalyzer* contains the required domain knowledge to extract the *Location*, *Time* and *Topic* out of it with the SAX parser. The *AixmAnalyzer* then uses the *Cube schema* to build the hierarchies. Every element is checked line by line and relevant information gets extracted. A more complex task is to map the DNOTAM into RDF which is done by the *AixmMapper* or *AixmToRdfSAXParser*. The prototypical mapper supports three concepts *"aixm:availability"*, *"aixm:activation"*, *"aixm:annotation"*. The SAX parser implementation creates RDF resources for every child element of a supported concept recursively as well as for important parent elements. Due to the structure of a DNOTAM that the direct children of "Subject" element always are "Predicates" in lower case and vice versa, RDF triples generation fits naturally.

Cube schema. The *Cube schema* defines the multidimensional model of the Big KG-OLAP lakehouse. For portability and to support different use cases, it should be configurable. As a first step, the *Cube schema* is defined in a YAML file and interpreted automatically by the Spring Java services. The logical design of the Cassandra database is not automatically derived from it yet but due to its design it is easily possible to automatically create dimension tables based on the *Cube schema* in a future effort. An excerpt of the current

Cube schema is included in listing 6.3. The schema includes the dimensions with its corresponding levels which in turn must contain its parent level and Java data type. Also, all hierarchies must be specified. Because those are dynamic and subject to change they should be moved into a database in the future. The *Time* dimension is a special case. It is not necessary to configure every day in history as a hierarchy but the application code rather recognizes it as *Time* dimension and automatically infers hierarchies.

Listing 6.3: YAML Cube schema excerpt for ATM use case

```yaml
schema:
  dimensions:
    location: # dimension
      territory: # level
        rollUpLevel: ALL
        javaDataType: String
      fir:
        rollUpLevel: territory
        javaDataType: String
      location:
        rollUpLevel: fir
        javaDataType: String
    topic:
      category:
        rollUpLevel: ALL
        javaDataType: String
      family:
        rollUpLevel: category
        javaDataType: String
      feature:
        rollUpLevel: family
        javaDataType: String

    time:
      year:
        rollUpLevel: ALL
        javaDataType: Year
      month:
        rollUpLevel: year
        javaDataType: YearMonth
      day:
        rollUpLevel: month
        javaDataType: LocalDate

hierarchies:
  location:
    - territory: Austria
      fir: LOVV
      location: LOWW
    - territory: Germany
      fir: EDGG
      location: EDGP
  ...

  topic:
    - category: AirportHeliport
      family: AirportHeliport
      feature: NonMovementArea
    - category: Routes
      family: EnRoute
      feature: RouteSegment
  ...
```

This *Cube schema* in YAML format is loaded by the *CubeSchemaReader*. The reader class parses it into a Java *RawCubeSchema* which in turn serves as input for the *CubeSchema*. A *CubeSchema* consists of a set of *Level* objects. These classes and all other classes that are important in this context are shown in the class diagram in Figure 6.9. A *Level* knows its dimension, identifier, data type, depth and to which *Level* it rolls up to. The depth is important to order levels within a dimension for database queries. *Member* objects represent level to value pairs. A *Hierarchy* is a set of *Member* objects but it is only valid if it contains members of the same dimensions, only one member per dimensions and misses no members that another rolls up to. If one of these constraints is not satisfied, a *HierarchyInvalidException* is thrown. For simplicity, this class is not included in the diagram. A combination of one *Hierarchy* object per dimension is called *Context*. A context represents a coordinate in the virtual KG-OLAP cube. The *SliceDiceContext* is a special case where missing hierarchies are not substituted with a hierarchy on the *ALL* level. For instance, a *Slice'n'Dice* query that only slices the *Time* dimension should return all the data for the given time regardless of the *Topic* and *Location*. If missing hierarchies would be substituted, only contexts on the *ALL* level in the *Topic* and *Location* dimension would be returned for the same query. A *StoredHierarchy* is similar to a *Hierarchy* but with the difference that it knows to which contexts it is associated. The usage of this is class further explained in section 6.3. A *MergeLevels* object consists of a map of dimension to level entries which is used for *contextual operations*. Finally, the *RollUpFun* holds the knowledge to roll up a given *Member* to its parent one. For that, it uses the hierarchies defined in the *Cube schema* and an input *Member* to construct the parent *Member* object. This class also holds the special handling of *Time* members which do not require explicitly defined hierarchies to roll up a member.

Angular UI and Java apps. An *Angular UI*, a *Java file uploader* and a *Java context operation performance test* were implemented to test the usage of the prototype and run performance evaluation tests. The *Angular UI* is a web interface that allows the user to upload files (for *data ingestion*) and enter KGOQL queries (for *contextual operations*). Currently, the only data type that can be selected is *AIXM*. Moreover, there is a query log on the right side that shows interesting parameters of recently processed *contextual operations*, two graphs on the bottom that plots *data ingestion* times and general statistics about the Big KG-OLAP lakehouse below the headline. Besides functionality, there is a dynamic syntax documentation how to use the KGOQL based on the underlying *Cube schema* collapsed under the text area together with a few static examples. The screenshot shows the overview of the *Angular UI* with the mentioned functionalities. Underneath the *Query cube* button appears the query result in plain RDF N-Quads format (stripped to 1000 lines to avoid UI overload) together with the result size in quads and mebibytes.

Two Java applications *Java file uploader* and *Java context operation performance test* support the process to evaluate the performance of the cloud-native Big KG-OLAP lakehouse. The first one is a single class Java application that uploads files to a configured *Surface* endpoint. During the performance test in section 7, multiple of those instances are started to test the limits of the REST service. The latter one is a complete automatic script that autonomously

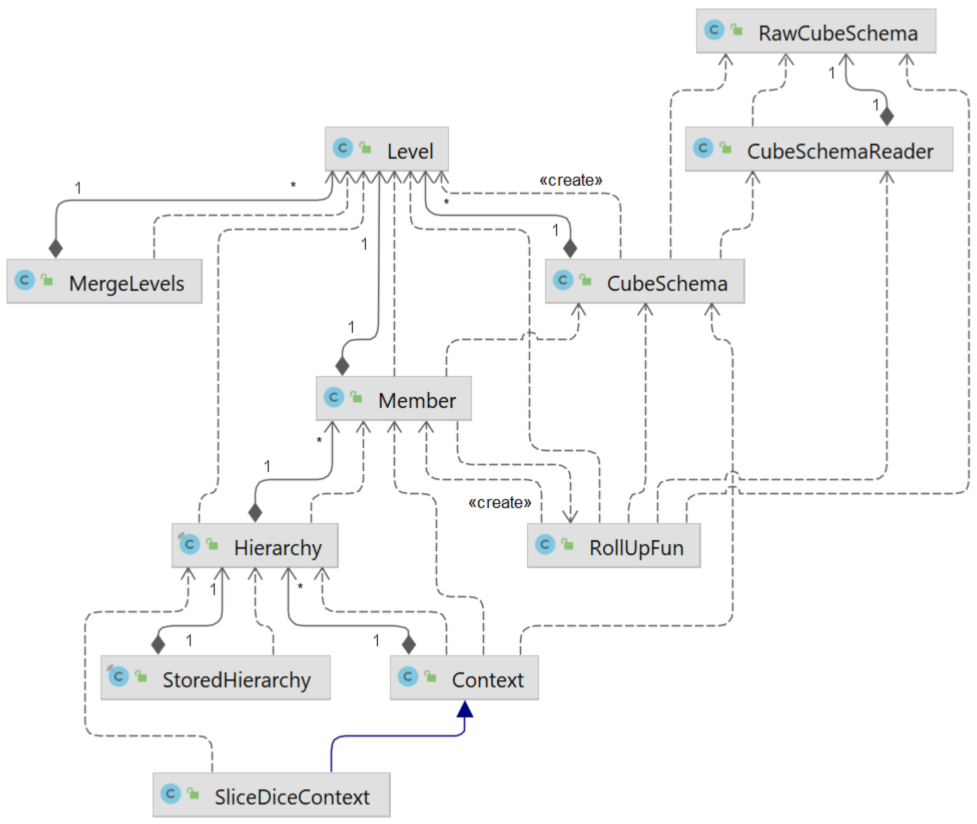
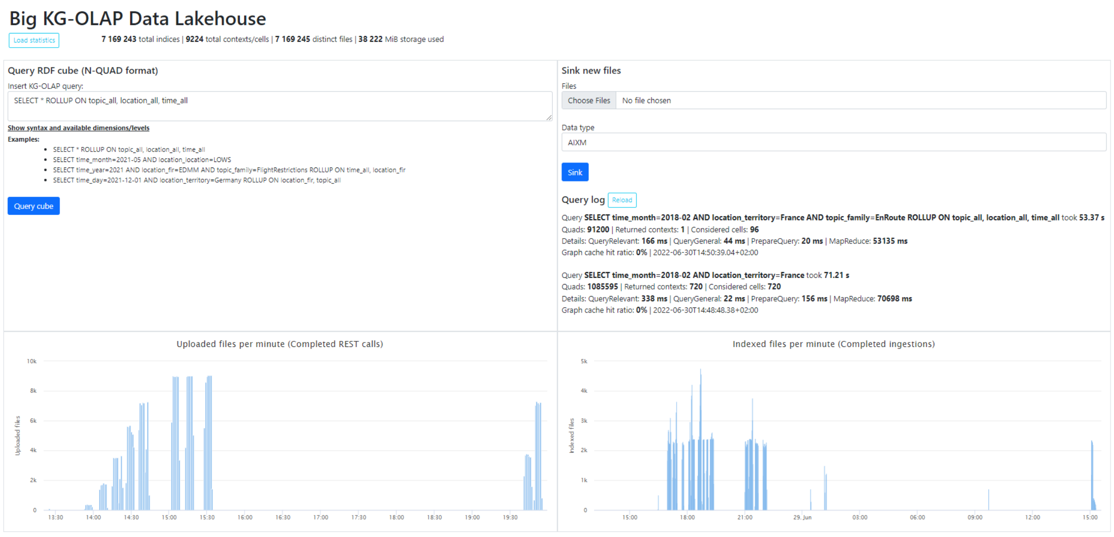


Figure 6.8.: *Cube schema* class diagram

runs *contextual operation* tests. It can communicate with the Kubernetes cluster and the *Surface* services. During the test run it restarts running Kubernetes instances, scales services, evicts the *Graph cache* and sends *contextual operation* queries to the configured *Surface* endpoint without manual interaction.


Load statistics
Choose Files
Sink
Reload
Query cube
Uploaded files per minute (Completed REST calls)
Uploaded files
Indexed files per minute (Completed ingestions)
Indexed files

Big KG-OLAP Data Lakehouse

7 169 243 total indices | **9224** total contexts/cells | **7 169 245** distinct files | **38 222** MiB storage used

Query RDF cube (N-QUAD format)

Insert KG-OLAP query:

SELECT * ROLLUP ON topic_all, location_all, time_all

Show syntax and available dimensions/levels

Examples:

- SELECT * ROLLUP ON topic_all, location_all, time_all
- SELECT time_month=2021-05 AND location_location=LOWS
- SELECT time_year=2021 AND location_fir=EDMM AND topic_family=FlightRestrictions ROLLUP ON time_all, location_fir
- SELECT time_day=2021-12-01 AND location_territory=Germany ROLLUP ON location_fir, topic_all

Sink new files

Files

No file chosen

Data type

AIXM

Query log

Query **SELECT time_month=2018-02 AND location_territory=France AND topic_family=EnRoute ROLLUP ON topic_all, location_all, time_all** took **53.37 s**
Quads: **91200** | Returned contexts: **1** | Considered cells: **96**
Details: QueryRelevant: **166 ms** | QueryGeneral: **44 ms** | PrepareQuery: **20 ms** | MapReduce: **53135 ms**
Graph cache hit ratio: **0%** | 2022-06-30T14:50:39.04+02:00

Query **SELECT time_month=2018-02 AND location_territory=France** took **71.21 s**
Quads: **1085595** | Returned contexts: **720** | Considered cells: **720**
Details: QueryRelevant: **338 ms** | QueryGeneral: **22 ms** | PrepareQuery: **156 ms** | MapReduce: **70698 ms**
Graph cache hit ratio: **0%** | 2022-06-30T14:48:48.38+02:00

Figure 6.9.: Big KG-OLAP lakehouse Angular UI

6.2. Data ingestion

The *Surface* provides an HTTP POST REST endpoint to upload data object as files. The HTTP POST file upload request also must contain a parameter called *type*. This parameter specifies the data type hence which *Engine* is used to analyze or map the file, respectively. The *Surface* also checks immediately if there is an *Engine* installed for the given *type* and fails the request if not. Otherwise, the file is stored on *Amazon S3* under a generated *Key* and its details such as file size and original name are stored in the *Amazon Keyspaces* database. Finally, the file key and the type are sent as a single message to the *Amazon SQS* queue and the request returns the successful status code 200.

The *Circulator* service class *DataIngestionScheduler* has a method called *ingest* with the Spring *Scheduled* annotation that is scheduled to run every 100 milliseconds after the last method run finishes. Figure 6.10 shows the important method calls that happen during a single *data ingestion* cycle in the *Circulator* service. The first statement in the *ingest* method is to poll a configurable number of new files (currently the *BATCH_SIZE* is 5) from the *SqsMessagingService*. The polled files are then processed in parallel using the Java parallel stream functionality. Every file is passed to the *ContextExtractionServiceanalyze* method. This method loads the file from the *S3StorageService* and the compatible *Engine* (currently *AixmEngine* is the only one) and uses it to extract the relevant *Member* objects out of the file. These *Member* objects are transformed into complete *Hierarchy* objects using the *Cube schema* knowledge. The Cartesian product of all *Hierarchies* result in a set of *Contexts* which is the output of the *ContextExtractionServiceanalyze* method. Every *Hierarchy* has a unique hash based on its *Member* values. The hash function used to calculate the hash it not the default Java *hashCode* function but the 160-bit SHA-1 function because the default 32-bit *hashCode* implementation is not collision-save enough to be used as a unique ID while SHA-1 practically is [49]. A *Context* in turn is also identified by a hash based on the hashes of the *Hierarchies* it consists of. The use of hashes as IDs enable parallel *upserts* of equal contexts by multiple *Circulator* instances. Once the *ContextExtractionServiceanalyze* method returns, the *DatabaseServiceupsertContext* method is called. Every *Hierarchy* is stored independently using the *Member* values as primary key. The database entry also contains the *Hierarchy* hash as well as a set of *Context* hashes to which it is associated to. This is important for later *Context operations*. Also, the *Context* hash is stored in an own table for statistical purposes. Afterwards, the relationships between every context and the file identifier are stored (=indices) using the *DatabaseServiceupsertFile* method. If every operation is successful, cached graphs for the contexts are evicted with *RedisCachedeleteCachedGraphs* and the job is deleted from the queue with *SqsMessagingServicedeleteMessage*. Otherwise, if at least one operation fails, the cache is not touched and the message is not deleted from the queue. The message is then processed again after a configurable visibility timeout by the same or another *Circulator* instances. Due to the hash-based *upsert* strategy it is no problem if a subset of the database requests were successful because the overriding requests have the same hashes hence the database records are equal to the previous ones. Finally, if the number of polled files matches the *BATCH_SIZE*, the *DataIngestionScheduleringest* method calls itself recursively as it is assumed the queue is not empty. This increases the ingestion speed. If not, the method is scheduled to run again 100 milliseconds later.

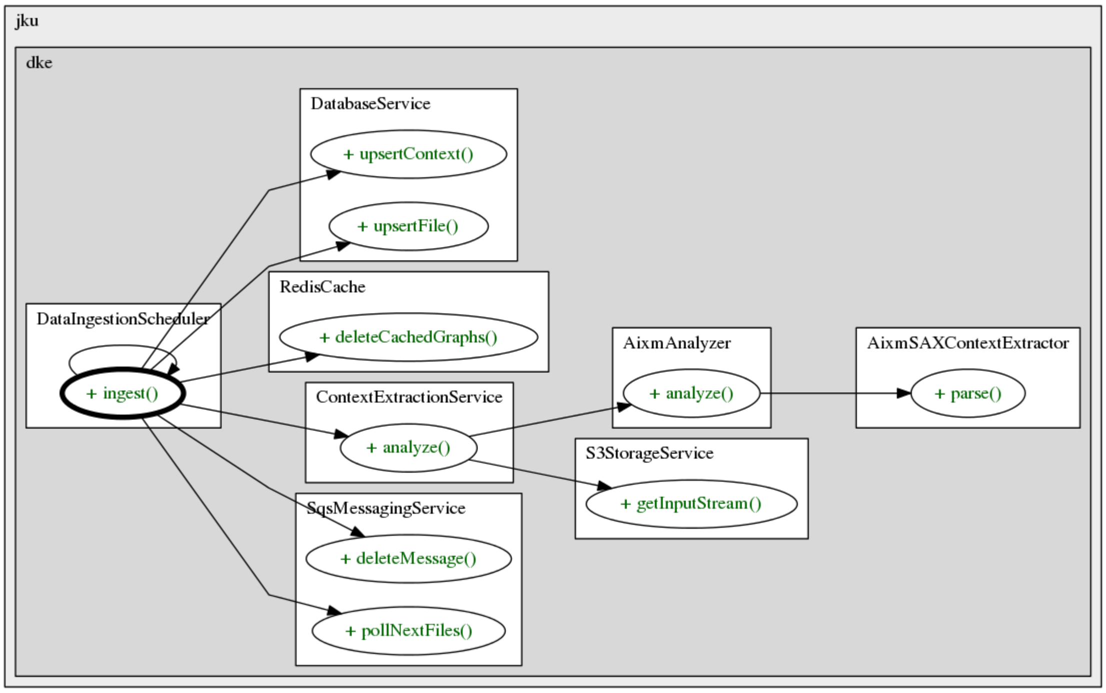


Figure 6.10.: *DataIngestionScheduler* method calls

6.3. Contextual operations

The second main functionality of the Big KG-OLAP lakehouse prototype implementation is the processing of *contextual operations*. The *Surface* provides an HTTP POST REST endpoint that accepts a JSON body containing a KGOQL query. It returns RDF quads in N-Quads format as plain text. This subsection explains the implementation of the *contextual operations*. To better comprehend it the same continuous example as in section 5 is used throughout this subsection. The KGOQL query is shown again in listing 6.4. For reference, the KGOQL grammar is described in appendix A.

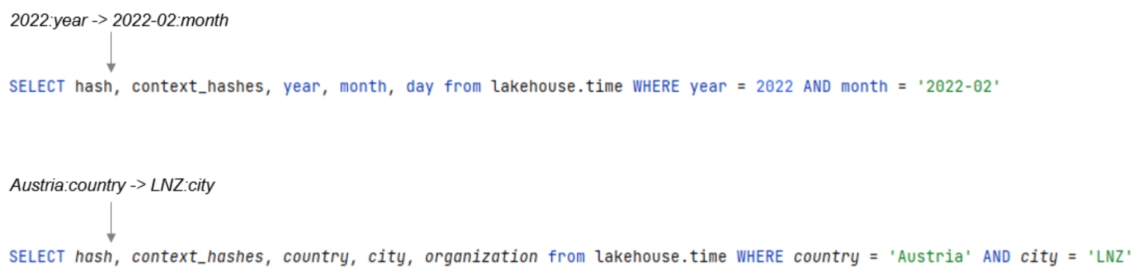
Listing 6.4: KGOQL query

```
SELECT time_month=2022-02 AND location_city=LNZ ROLLUP ON time_month
```

Parse KGOQL query. The query can be a simple *Slice'n'Dice* operations or contain a roll up clause to add the *Merge* operation. The *Surface* service contains an implementation to interpret KGOQL queries in the *QueryParserService* class. It extracts and constructs the

SliceDiceContext and *MergeLevels*. In the given example, the constructed *SliceDiceContext* is *"2022:year -> 2022-02:month" - "Austria:country -> LNZ:city"*. The *QueryParserService* utilizes the *CubeSchema* to infer the complete *Hierarchies* from the extracted *Members*. Without the complete *Hierarchies* it would not be possible to generate the necessary database statements to select the corresponding contexts. The *MergeLevels* are a map from dimensions to levels, in this example *time:month*. The *SliceDiceContext* is the input to query the *specific* and *general* contexts from the *Amazon Keyspaces* database.

Select specific contexts. The goal of this step is to find any stored context that is located at or underneath the given *SliceDiceContext* coordinate. Because contexts are stored in distributed dimension tables and the NoSQL database does not support joins, one query per dimension is necessary. For every dimension that is present in the *SliceDiceContext*, queries are mapped according to the *Hierarchies* it contains. The query mapping from the *Hierarchies* to the CQL queries is illustrated in Figure 6.11. For every dimension in the *SliceDiceContext*, the *Hierarchies* at or below the coordinate together with its hash and associated *context_hashes* are selected. A *Hierarchy* knowing its associated *context_hashes* is represented as a *StoredHierarchy* object in the code. A special case are absent dimensions (e.g., if the query would not contain the clause *location_city=LNZ* the *LOCATION* dimension would be absent) where every existing *Hierarchy* is selected from the database. Knowing the *Hierarchy* of absent dimensions is required later for RDF cube construction. Now that every relevant *Hierarchy* is known, the Cartesian product is calculated. This results in a set of potential existing *Contexts*. Finally, the intersection set of the *context_hashes* of the potential *Contexts* is calculated which results in a subset of complete *Context* objects that actually exist in the Big KG-OLAP lakehouse.



```
SELECT hash, context_hashes, year, month, day from lakehouse.time WHERE year = 2022 AND month = '2022-02'
```

```
SELECT hash, context_hashes, country, city, organization from lakehouse.time WHERE country = 'Austria' AND city = 'LNZ'
```

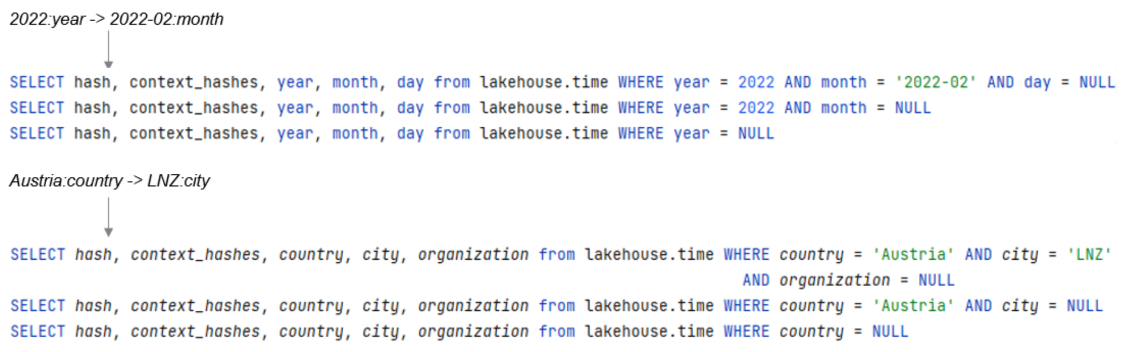
Figure 6.11.: Select specific contexts CQL queries based on SliceDiceContext

The three specific contexts returned by this step are shown in listing 6.5.

Listing 6.5: Selected specific contexts

```
C2: "2022:year -> 2022-02:month" - "Austria:country -> LNZ:city"
C4: "2022:year -> 2022-02:month -> 2022-02-01:day" - "Austria:country ->
    LNZ:city"
C5: "2022:year -> 2022-02:month" - "Austria:country -> LNZ:city -> JKU:
    organization"
```

Select general contexts. The difference between the previous step and this one is that the *general* contexts refer to any context above the *SliceDiceContext* (instead of at or below). For that, one or more queries are built per dimension present in the *SliceDiceContext* to retrieve the required hierarchies from the database. Figure 6.12 shows the CQL queries that are generated based on the given *Hierarchies* in the *SliceDiceContext*. The first query selects the *Hierarchy* at the *SliceDiceContext* while the following ones target each levels above. Selecting dimensions that are absent in the *SliceDiceContext* is not required in this case because only the *context_hashes* are required later on. Once again, the Cartesian product of the *Hierarchies* per dimension is constructed to know the potential general *context_hashes*. Because *Hierarchies* can be absent in the Cartesian product, *Context* objects can not be created as a *Context* is only valid if it contains one *Hierarchy* per dimension. This is no problem as for general context, only the *context_hashes* are required in a later step. The Cartesian product is the input for the intersection of the *context_hashes* which results in a subset of them. This subset of *context_hashes* represent the IDs of the *general* context for the given *SliceDiceContext*.



```
SELECT hash, context_hashes, year, month, day from lakehouse.time WHERE year = 2022 AND month = '2022-02' AND day = NULL
SELECT hash, context_hashes, year, month, day from lakehouse.time WHERE year = 2022 AND month = NULL
SELECT hash, context_hashes, year, month, day from lakehouse.time WHERE year = NULL
```

```
SELECT hash, context_hashes, country, city, organization from lakehouse.time WHERE country = 'Austria' AND city = 'LNZ'
                                                                                AND organization = NULL
SELECT hash, context_hashes, country, city, organization from lakehouse.time WHERE country = 'Austria' AND city = NULL
SELECT hash, context_hashes, country, city, organization from lakehouse.time WHERE country = NULL
```

Figure 6.12.: Select general contexts CQL queries based on SliceDiceContext

The single general context returned by this step are shown in listing 6.6.

Listing 6.6: Selected general context

```
C1: "2022:year" - "Austria:country"
```

Merge and propagate. This step includes two tasks, merging contexts up to the given *MergeLevels* and propagate knowledge from generic contexts to specific ones. Listing 6.7 shows the *Merge and propagate* algorithm in pseudocode. The first outer loop goes through every *specific* context and creates a map entry in the *result* for it. The inner loop also iterates every *specific* context and checks if the current one rolls up to or is equal to the outer one. If yes, the inner context is rolled up to the given *MergeLevels* and registered as *special* context. A *special* context describes a context that derives knowledge from a *generic* one. The distinct resulting *special* contexts are then mapped to the outer context. The intermediate result is a mapping from every *specific* context (=context at or under the *SliceDiceContext*) to itself and any other *specific* context underneath it to which it propagates its knowledge taking the *Merge* step into consideration. The intermediate result is added as comment (lines starting with a hashtag) in listing 6.7 line 13 - 17. The context *C2* propagates its knowledge to two graphs whereas *C1* and *C2* to one, respectively. In the final step, one *result* mapping from every *general* context to all existing graphs is added because the general contexts are always more generic than the previous created graphs. The final result looks as shown in the same listing line 24 - 29. Both *C1* and *C2* propagate their knowledge to the final graphs *"2022:year -> 2022-02:month" - "Austria:country -> LNZ:city"* and *"2022:year -> 2022-02:month" - "Austria:country -> LNZ:city -> JKU:organization"*. Knowledge of *C4* is added to *"2022:year -> 2022-02:month" - "Austria:country -> LNZ:city"* while *C5* is linked to *"2022-02:month" - "Austria:country -> LNZ:city -> JKU:organization"*. This mapping is returned as *MergeAndPropagateResult*.

Listing 6.7: Merge and propagate pseudo code

```
1  result = [:]
2
3  # merge and propagate specific knowledge
4  For Each ctx in specificContexts
5      specialCtxs = []
6      For Each potentialSpecialCtx in specificContexts
7          If (ctx == potentialSpecialCtx Or potentialSpecialCtx.rollsUpTo(ctx)
               ) {
8              potentialSpecialCtx.rollUpTo(mergeLevels)
```

```
            specialCtxs.Add(potentialSpecialCtx)
        }
    result.put(ctx, specialCtxs)

# intermediate result mapping
# [C2:"2022:year -> 2022-02:month" - "Austria:country -> LNZ:city"]
# [C2:"2022:year -> 2022-02:month" - "Austria:country -> LNZ:city -> JKU:
    organization"]
# [C4:"2022:year -> 2022-02:month" - "Austria:country -> LNZ:city"]
# [C5:"2022:year -> 2022-02:month" - "Austria:country -> LNZ:city -> JKU:
    organization"]

# propagate general knowledge
For Each ctx in generalContexts
    result.put(ctx, result.values())

# final result mapping
# [C1:"2022:year -> 2022-02:month" - "Austria:country -> LNZ:city"]
# [C1:"2022:year -> 2022-02:month" - "Austria:country -> LNZ:city -> JKU:
    organization"]
# [C2:"2022:year -> 2022-02:month" - "Austria:country -> LNZ:city"]
# [C2:"2022:year -> 2022-02:month" - "Austria:country -> LNZ:city -> JKU:
    organization"]
# [C4:"2022:year -> 2022-02:month" - "Austria:country -> LNZ:city"]
# [C5:"2022:year -> 2022-02:month" - "Austria:country -> LNZ:city -> JKU:
    organization"]

Return result
```

Query and aggregate RDF cube. The previous result is the input for this *MapReduce* step. For every distinct context in the *MergeAndPropagateResult*, one gRPC request is sent to the *Bed* service. Those requests are sent asynchronously and in parallel to utilize the horizontal scalability of the *Bed*. Thread-safe *java.util.concurrent.CountDownLatch* objects ensure on the *Surface* side that the main thread waits until every request is completed. The *Bed* service returns chunks of RDF quads in N-Quads format until it completes. This process is described in the following paragraph. These RDF quads are added to a thread-safe set implementation which per definition automatically drops duplicate quads. In addition to this context-dependent knowledge, context information about the graphs themselves is added to the default graph. For instance, the default graph then contains the knowledge that the graph *"2022:year -> 2022-02:month" - "Austria:country -> LNZ:city"* contains the member *"2022:year"*. Finally, the *Surface* streams the resulting RDF cube in the line-based RDF N-Quads format to the requesting *Surface* instance. An example result is presented in listing 6.8. The first 9 lines show the metadata of the graphs in the default graph (no named graph at the end of the line) followed by context-dependent knowledge in form of four RDF quads associated to either one of the two named graphs. The dot at the end of each line completes the RDF statement.

Listing 6.8: Example RDF cube in RDF N-Quads format

```
 1 <uri:bigkgolap/context/context1> <uri:bigkgolap/time_year> "2022" .
 2 <uri:bigkgolap/context/context1> <uri:bigkgolap/time_month> "2022-02" .
 3 <uri:bigkgolap/context/context1> <uri:bigkgolap/location_territory> "
       Austria" .
 4 <uri:bigkgolap/context/context1> <uri:bigkgolap/location_city> "LNZ" .
 5 <uri:bigkgolap/context/context2> <uri:bigkgolap/time_year> "2022" .
 6 <uri:bigkgolap/context/context2> <uri:bigkgolap/time_month> "2022-02" .
 7 <uri:bigkgolap/context/context2> <uri:bigkgolap/location_territory> "
       Austria" .
 8 <uri:bigkgolap/context/context2> <uri:bigkgolap/location_city> "LNZ" .
 9 <uri:bigkgolap/context/context2> <uri:bigkgolap/location_organization> "JKU
       " .
10 <uri:bigkgolap/entity#ID1> <http://www.w3.org/1999/02/22-rID2-syntax-ns#
       type> "Inhabitants" <uri:bigkgolap/context/context1> .
11 <uri:bigkgolap/entity#ID1> <uri:bigkgolap/entity#number> 200000 <uri:
       bigkgolap/context/context1> .
12 <uri:bigkgolap/entity#ID2> <http://www.w3.org/1999/02/22-rID2-syntax-ns#
       type> "Established" <uri:bigkgolap/context/context2> .
13 <uri:bigkgolap/entity#ID2> <uri:bigkgolap/entity#number> 1966 <uri:
       bigkgolap/context/context2> .
```

The *Bed* service operates in the background. It utilizes the *Engines* to transform files associated to a given context into RDF models. Next, it iterates over every triple in the RDF model and constructs quads for the given final graphs. These RDF quads are written in N-Quads format and sent back to the *Surface* service in chunks of approximately 40 KiB because the optimal size for streaming large payloads appears to be 16 - 64 KiB [50]. However, due to code specifics it is not possible to efficiently track the real size of chunks. Instead, the lines per chunk are tracked. With two bytes per character and roughly 300 characters per line this results in about 70 lines per chunk. Additionally, the chunks are compressed with Gzip because it can obtain significant compression ratios of x8-x10 on RDF N-Quads [48].

7. Performance evaluation

An essential part of this thesis and the third contribution is the performance evaluation of the cloud-native Big KG-OLAP lakehouse prototype implementation. More precisely, the scalability of the two main functionalities *data ingestion* and *contextual operations* is tested. For that, some features were added to the prototype such as additional logging or REST endpoints for test automation. The tests were conducted exclusively with AWS resources and the setups are described before each test.

7.1. Data ingestion

This section covers the scalability of the *data ingestion* functionality. To evaluate the capability to ingest a large amount of data, it is necessary to reveal the limits of involved components. Those are the *Surface* and the *Circulator*. Hence, the evaluation is split into two independent tests. The first test focuses on the *Surface* while the second one targets the *Circulator*. Logically, there is the *Amazon SQS* job queue between both services in the *data ingestion* process. To ensure an independent evaluation, the *Circulator* services is scaled down to zero instances during the *Surface* tests, which means the *Amazon SQS* job queue fills up steadily with every ingested file. The persisted jobs are eventually processed during the *Circulator* tests when no file is uploaded anymore at the time of testing. Subsection 1 evaluates the scalability of the *Surface* while subsection 2 covers the *Circulator*.

7.1.1. File upload (Surface)

In this test, files are uploaded to the *Surface*'s HTTP POST REST endpoint for *data ingestion* using the *Java file uploader*. The *Surface* simply processes the requests, stores the file in the configured *Amazon S3* bucket and registers an event in the *Amazon SQS* job queue before it completes the request. The goal of this test is to reveal how many files can be uploaded successfully to one or more *Surface* instances on a single machine. For that, a self implemented *Java file uploader* is employed. It is a simple Java application that loads files from a configured source and sends HTTP POST requests to a configurable endpoint. The requests are sent asynchronously to simulate a high throughput. The test strategy is to upload files for five total minutes and calculate the average and median throughput per minute afterwards. Then, change the test setup, e.g. increase the number of *Java file uploader* instances. The starting test setup looks as follows.

Big KG-OLAP lakehouse:

- Deployment according to appendix B.2 and section 6
- One *Amazon EKS* cluster node *m5.4xlarge*: 16 vCPU, 64GiB memory, Up to 10 Gigabit (Amazon Linux 2 x86_64 OS)
- 1 Surface instance
- 0 Bed instances
- 0 Circulator instances
- 1 Angular web UI instance

Java file uploader:

- EC2 (Elastic Compute Cloud) instance *c5.4xlarge*: 16 vCPU, 64GiB memory, Up to 10 Gigabit (Amazon Linux 2 x86_64 OS)
- 1 file uploader instance

Test 1

Purpose. Test how many files one *Java file uploader* can send per minute.
Test time frame. 13:53 – 13:59 The test time frame includes setup at the beginning and tear down at the end. This is the same for every test.
Observation time frame. 13:54 – 13:58 The actual observation time between setup and

tear down. This is the same for every test.
Uploads per minute within observation time frame. (visualized in Figure 7.1)

- Total: 358, 374, 359, 366, 361
- Average: 363.6
- Median: 361

Evaluation. About 364 uploads per minute are possible with one surface and one *Java file uploader* instance running on the two separate nodes.

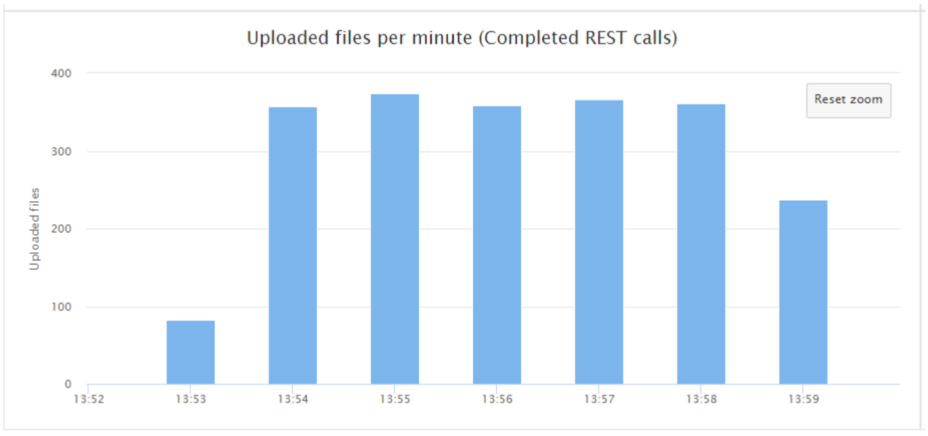


Figure 7.1.: Surface performance test 1

Test 2

Purpose. Test how many files one *Surface* instance can ingest on one machine.
Java file uploader setup change 5 *Java file uploader* instances
Test time frame. 13:53 – 13:59
Observation time frame. 13:54 – 13:58
Uploads per minute within observation time frame. (visualized in Figure 7.2)

- Total: 1679, 1698, 1816, 1730, 1744
- Average: 1733.4
- Median: 1730

Evaluation. Five *Java file uploader* instances are able to upload approximately five times as many files to one *Surface* instance than a single instance. This indicates that the upload limits of the *Surface* instance are not reached yet.

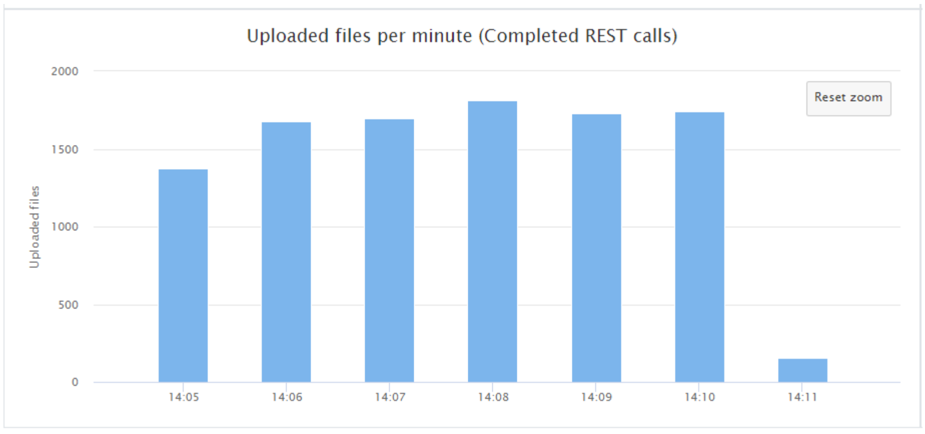


Figure 7.2.: Surface performance test 2

Test 3

Purpose. Test how many files one *Surface* instance can ingest on one machine.
Java file uploader setup change 10 *Java file uploader* instances
Test time frame. 19:41 – 19:47
Observation time frame. 19:42 – 19:46
Uploads per minute within observation time frame. (visualized in Figure 7.3)

- Total: 3691, 3787, 3742, 3598, 2339
- Average: 3431.4
- Median: 3691

Evaluation. Ten *Java file uploader* instances are able to upload approximately ten times as many files to one *Surface* instance than a single instance. This indicates that the upload limits of the *Surface* instance are not reached yet.

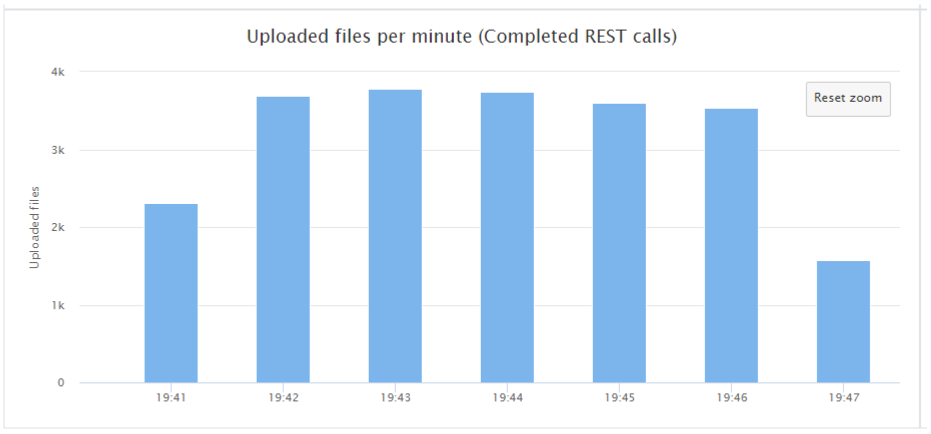


Figure 7.3.: Surface performance test 3

Test 4

Purpose. Test how many files one *Surface* instance can ingest on one machine.
Java file uploader setup change 15 *Java file uploader* instances
Test time frame. 14:26 – 14:32
Observation time frame. 14:27 – 14:31
Uploads per minute within observation time frame. (visualized in Figure 7.4)

- Total: 5612, 5579, 5686, 5251, 5106
- Average: 5446.8
- Median: 5579

Evaluation. 15 *Java file uploader* instances are able to upload approximately 15 times as many files to one *Surface* instance than a single instance. This indicates that the upload limits of the *Surface* instance are not reached yet.

Test 5

Purpose. Test how many files one *Surface* instance can ingest on one machine.
Java file uploader setup change 20 *Java file uploader* instances
Test time frame. 19:49 – 19:55

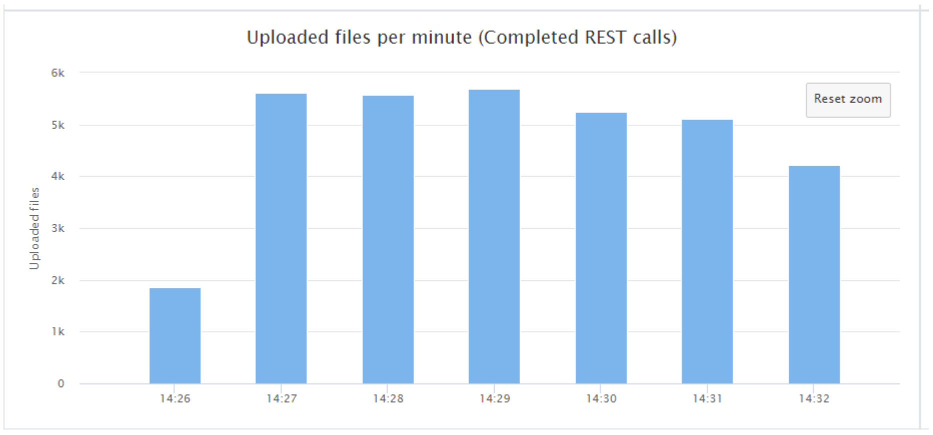


Figure 7.4.: Surface performance test 4

Observation time frame. 19:50 – 19:54
Uploads per minute within observation time frame. (visualized in Figure 7.5)

- Total: 7022, 7297, 7199, 7109, 7228
- Average: 7171
- Median: 7199

Evaluation. 20 *Java file uploader* instances are able to upload approximately 20 times as many files to one *Surface* instance than a single instance. This indicates that the upload limits of the *Surface* instance are not reached yet.

Test 6

Purpose. Test how many files one *Surface* instance can ingest on one machine.
Java file uploader setup change 30 *Java file uploader* instances
Test time frame. 15:02 – 15:08
Observation time frame. 15:03 – 15:07
Uploads per minute within observation time frame. (visualized in Figure 7.6)

- Total: 8988, 8983, 8952, 8987, 8955
- Average: 8973

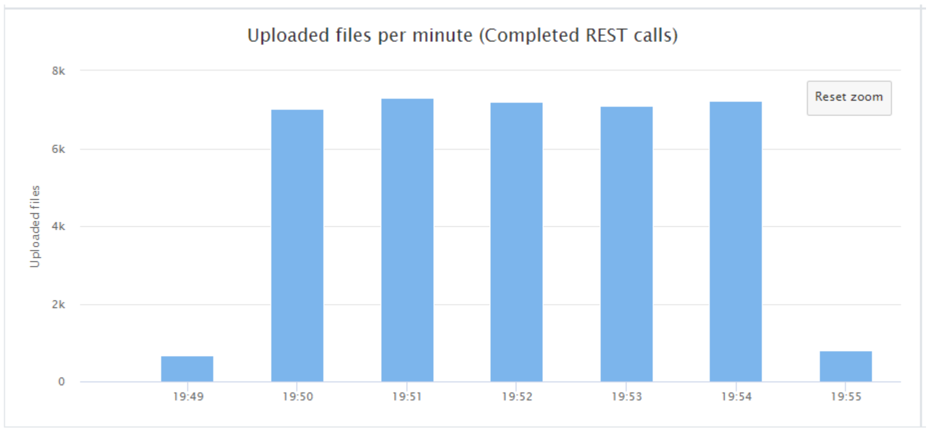


Figure 7.5.: Surface performance test 5

- Median: 8983

Evaluation. 30 *Java file uploader* instances are not able to upload approximately 30 times as many files to one *Surface* instance than a single instance. This indicates that the upload limits of the *Surface* instance reached.

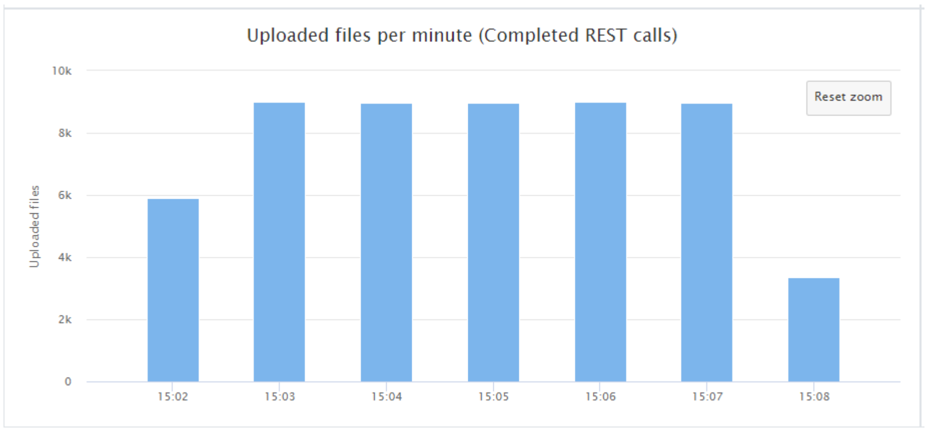


Figure 7.6.: Surface performance test 6

Test 7

Purpose. Verify that the upload limits are reached.
Java file uploader setup change 40 *Java file uploader* instances

Test time frame. 15:13 – 15:19
Observation time frame. 15:14 – 15:18
Uploads per minute within observation time frame. (visualized in Figure 7.7)

- Total: 8977, 8985, 9008, 8993, 8994
- Average: 8991.4
- Median: 8993

Evaluation. The current setup with 40 *Java file uploader* did not lead to a signification upload rate increase. This verifies that the limits are reached.

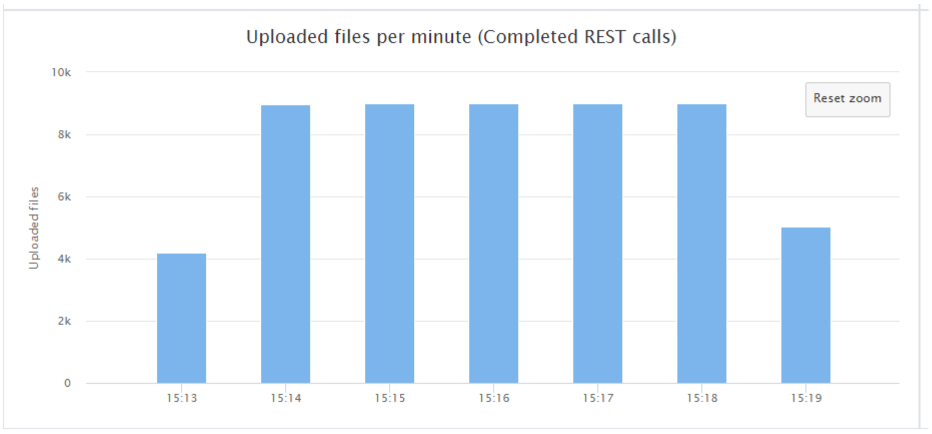


Figure 7.7.: Surface performance test 7

Test 8

Purpose. Test if two *Surface* instances can ingest more files than one.
Surface instances change 2 *Surface* instances
Test time frame. 15:28 – 15:34
Observation time frame. 15:29 – 15:33
Uploads per minute within observation time frame. (visualized in Figure 7.8)

- Total: 8589, 8970, 9026, 9042, 9042
- Average: 8933.8

- Median: 9026

Evaluation. Two *Surface* instances can not accept more uploads than one *Surface* on the used machine. This indicates that the machine or network limits are reached.

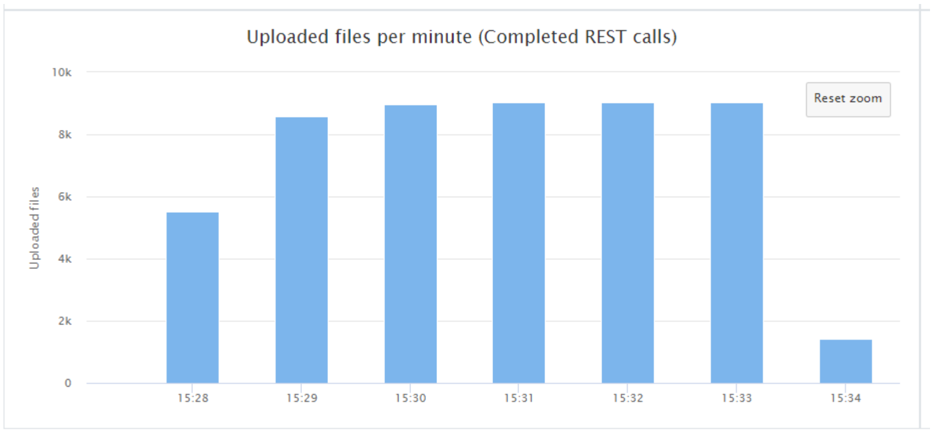


Figure 7.8.: Surface performance test 8

Conclusion. Figure 7.9 shows the average numbers of uploads per minute for the different test setups. There is a linear increase in uploads per minute up to 20 *Java file uploader* instances. From 20 to 30 *Java file uploader* instances the increase is less than linear and with 30, the maximum number of uploads is reached. Increasing the number of *Java file uploader* or *Surface* instances did not increase the number further. This indicates that with the current AWS machine setup utilized to run the *Surface* and *Java file uploader* instances, the maximum is about 9000 uploads per minute.

7.1.2. File indexing (Circulator)

The second part of the *data ingestion* performance evaluation is to test the amount of files that can be processed by one or more *Circulator* instances on one machine. Because the file upload test was conducted before this one, the *Amazon SQS* job queue contains a large amount of data ingestion jobs ready to be processed. The initial setup for this test looks as follows.

Big KG-OLAP lakehouse:

- Deployment according to appendix B.2 and section 6

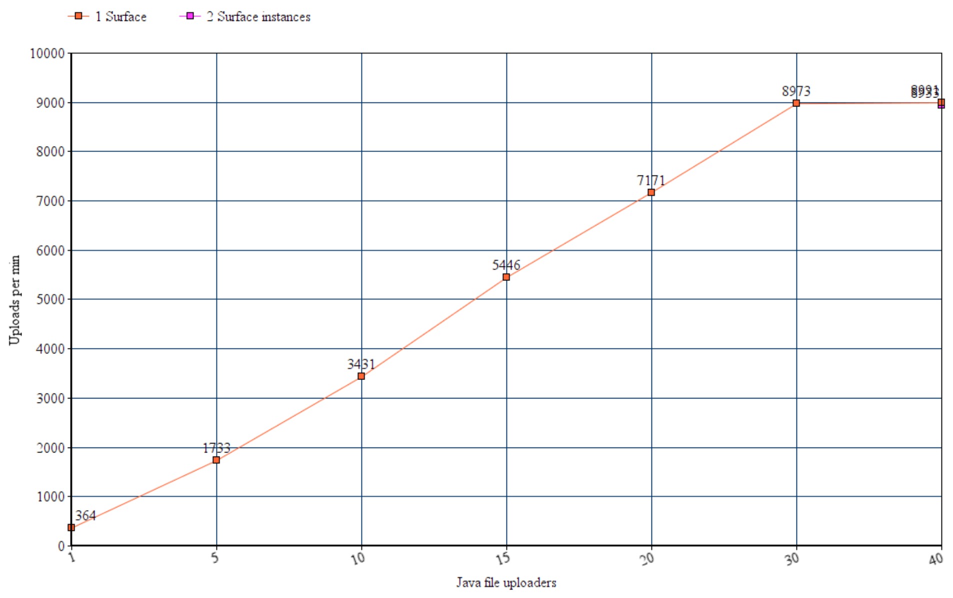


Figure 7.9.: Surface performance test overview

- One *Amazon EKS* cluster node *m5.4xlarge*: 16 vCPU, 64GiB memory, Up to 10 Gigabit (Amazon Linux 2 x86_64 OS)
- 0 Surface instances
- 0 Bed instances
- 1 Circulator instance
- 1 Angular web UI instance

Test 1

Purpose. Test the limits of one *Circulator* instance on the machine.
Test time frame. 17:14 – 17:20
Observation time frame. 17:15 – 17:19
Uploads per minute within observation time frame. (visualized in Figure 7.10)

- Total: 2217, 2310, 2279, 2266, 2269
- Average: 2268.2

- Median: 2269

Evaluation. One *Circulator* instance can process a steady amount of about 2268 files per minute on the single node.

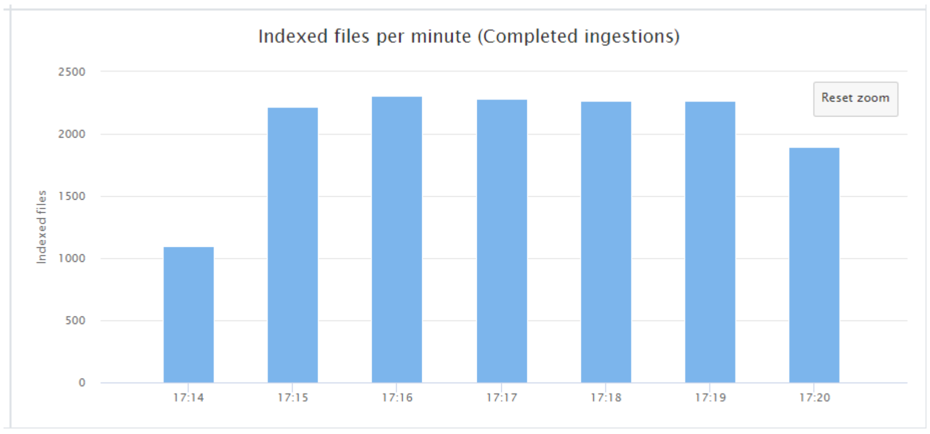


Figure 7.10.: Circulator performance test 1

Test 2

Purpose. Test how one *Circulator* instance performs over a longer test time.
Test time frame. 21:56 – 22:08
Observation time frame. 21:57 – 22:07
Uploads per minute within observation time frame. (visualized in Figure 7.11)

- Total: 2373, 2165, 2233, 2114, 2113, 2250, 2039, 2161, 2122, 2195, 2282
- Average: 2186.1
- Median: 2165

Evaluation. The indexing rate did not change significantly over a longer test period in comparison to the first test.

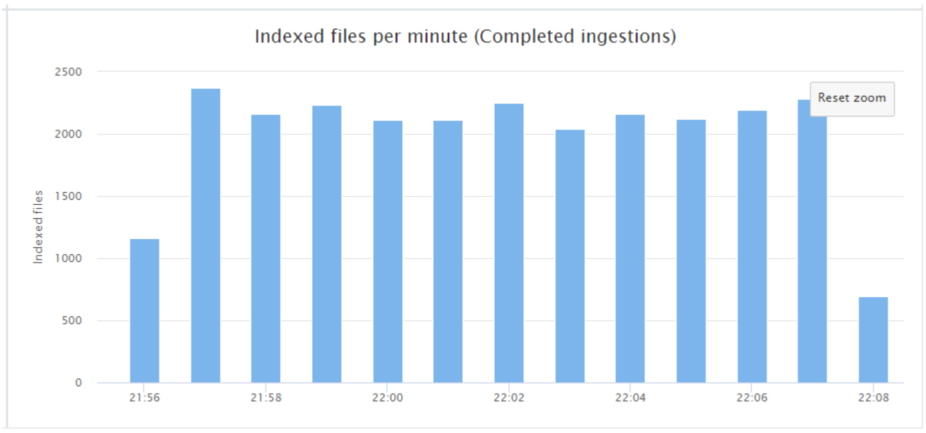


Figure 7.11.: Circulator performance test 2

Test 3

Purpose. Test if two *Circulator* instances are can process more files on a single machine than one.
Circulator instances change 2 *Circulator* instances
Test time frame. 17:22 – 17:28
Observation time frame. 17:23 – 17:27
Uploads per minute within observation time frame. (visualized in Figure 7.12)

- Total: 2363, 2398, 3289, 3643, 2260
- Average: 2790.6
- Median: 2398

Evaluation. The indexing rates per minute varies a lot. One suspicious phenomenon observed in the log was that a lot of write requests to the *Amazon Keyspaces* database failed. The data used to measure the performance is based on data stored in the Cassandra database. Hence, it is likely that the spikes in performance are caused by the *Amazon Keyspaces* service. The next test checks if a longer test period leads to an increased average indexing time per minute.

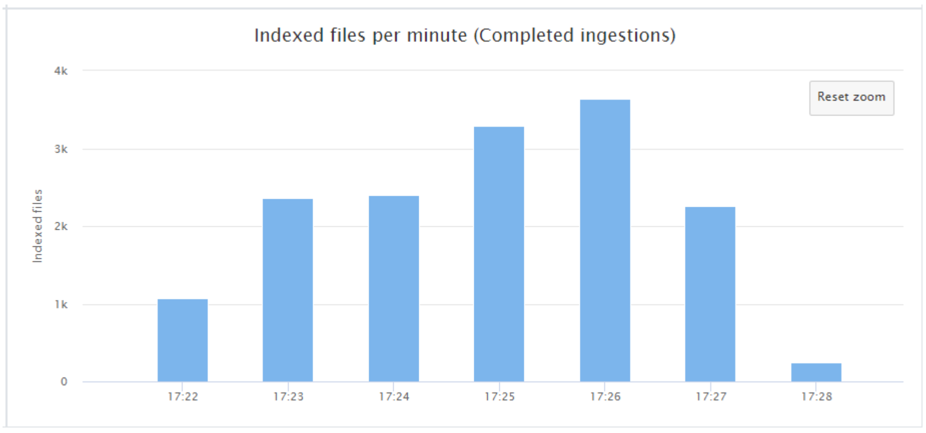


Figure 7.12.: Circulator performance test 3

Test 4

Purpose. Test how two *Circulator* instances perform over a longer test time.
Test time frame. 18:11 – 18:23
Observation time frame. 18:12 – 18:22
Uploads per minute within observation time frame. (visualized in Figure 7.13)

- Total: 3733, 3862, 4204, 2514, 2371, 2375, 2384, 2378, 2383, 2390, 2374
- Average: 2815.3
- Median: 2384

Evaluation. The ingestion rate per minute did not change significantly with a longer test time. However, the low variance from 18:15 – 18:22 is remarkable. This could not be observed in further tests. It is likely that the *Amazon Keyspaces* service stabilized its maximum throughput at this time and did not scale up or down.

Test 5

Purpose. Test if three *Circulator* instances are can process more files on a single machine than two.
Circulator instances change 3 *Circulator* instances

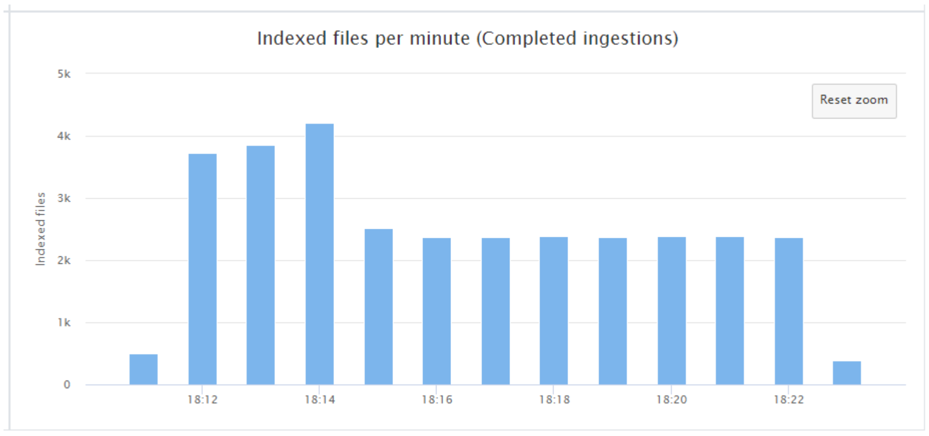


Figure 7.13.: Circulator performance test 4

Test time frame. 18:03 – 18:09
Observation time frame. 18:04 – 18:08
Uploads per minute within observation time frame. (visualized in Figure 7.14)

- Total: 2330, 2311, 2315, 2375, 2971
- Average: 2460.4
- Median: 2330

Evaluation. With three *Circulator* instances, the average ingestion rate decreased slightly. The reason could be that more load on *Amazon Keyspaces* led to an increased amount of failed database writes.

Test 6

Purpose. Test how three *Circulator* instances perform over a longer test time.
Test time frame. 18:32 – 18:44
Observation time frame. 18:33 – 18:43
Uploads per minute within observation time frame. (visualized in Figure 7.15)

- Total: 2372, 2378, 2374, 2374, 2488, 2989, 3523, 4219, 4753, 4561, 3375
- Average: 3218.73

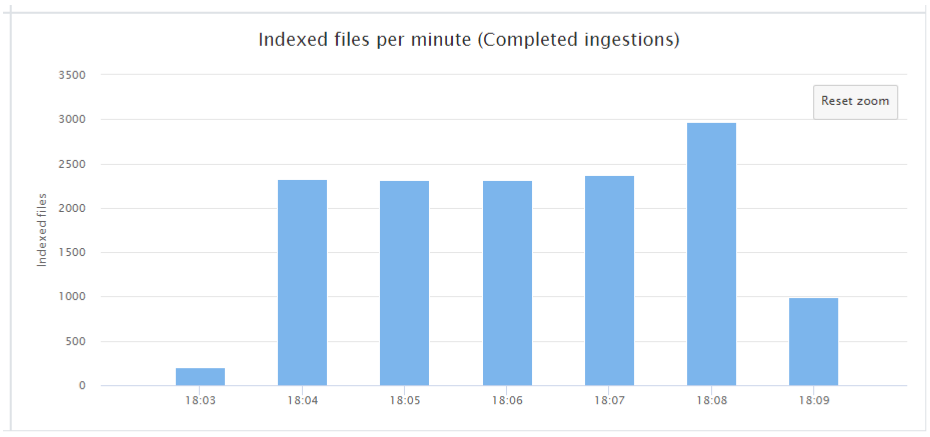


Figure 7.14.: Circulator performance test 5

- Median: 2989

Evaluation. The average number of indexed files per minute increased with a test runtime of 11 minutes over 5 minutes. Together with the fact that the numbers increase throughout the test period, it is most likely that *Amazon Keyspaces*, which scales automatically, is the bottleneck in the current setup.

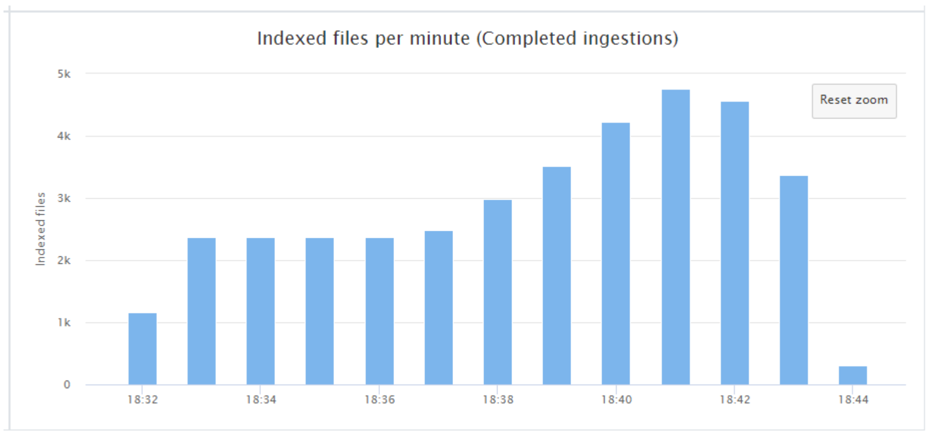


Figure 7.15.: Circulator performance test 6

Test 7

Purpose. Test if four *Circulator* instances are can process more files on a single machine than three.
Circulator instances change 4 *Circulator* instances
Test time frame. 18:47 – 18:53
Observation time frame. 18:48 – 18:52
Uploads per minute within observation time frame. (visualized in Figure 7.16)

- Total: 2375, 2363, 2376, 2365, 2718
- Average: 2439.4
- Median: 2375

Evaluation. With four *Circulator* instances, the average ingestion rate did not increase significantly. The numbers are lower than three *Circulator* instances process over an extended test time. This is most likely caused by the *Amazon Keyspaces* write limits at this time.

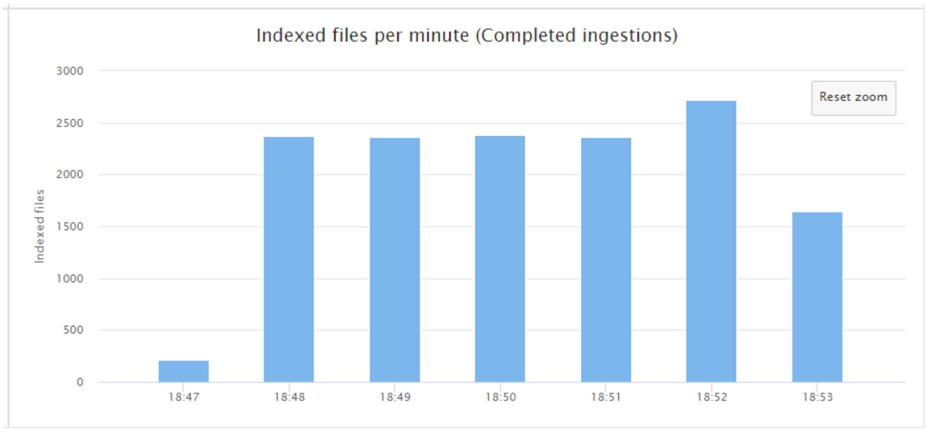


Figure 7.16.: Circulator performance test 7

Test 8

Purpose. Test how four *Circulator* instances perform over a longer test time.
Test time frame. 21:13 – 21:25

Observation time frame. 21:14 – 21:24
Uploads per minute within observation time frame. (visualized in Figure 7.17)

- Total: 2397, 3405, 5147, 3813, 4257, 4756, 4445, 4128, 4989, 5031, 5868
- Average: 4385.1
- Median: 4445

Evaluation. The average number of indexed files per minute increased with a test runtime of 11 minutes over 5 minutes. As already observed in test 6, the numbers increase over time in this test as well. This is caused by the automatic scaling mechanism of *Amazon Keyspaces*, which is clearly the limiting factor at this time.

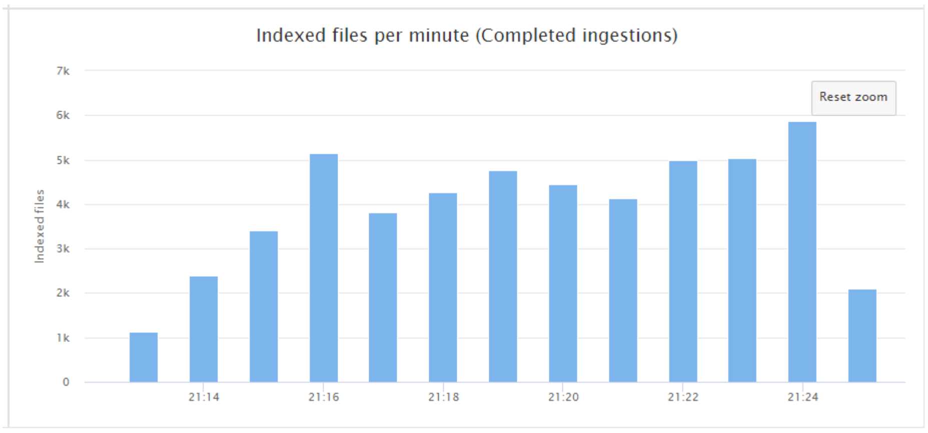


Figure 7.17.: Circulator performance test 8

Test 9

Purpose. Find the maximum number of *Circulator* instances that can run in parallel and index more files per minute than fewer instances.
Circulator instances change 5 *Circulator* instances
Test time frame. 18:59 - 19:05
Observation time frame. 19:00 – 19:04
Uploads per minute within observation time frame. (visualized in Figure 7.18)

- Total: 4511, 2515, 7583, 5423, 5326

- Average: 5071.6
- Median: 5326

Evaluation. The average number of indexed files per minute is higher with five *Circulator* instances compared to four instances. This indicates that the maximum number of possible running *Circulator* instances per machine is not reached. However, a large amount of write requests to the *Amazon Keyspaces* database fail and the scaling mechanism of *Amazon Keyspaces* in the background is opaque. Hence, it is impossible to test the limits of the *Circulator* service with the *Amazon Keyspaces* service being a bottleneck underneath.

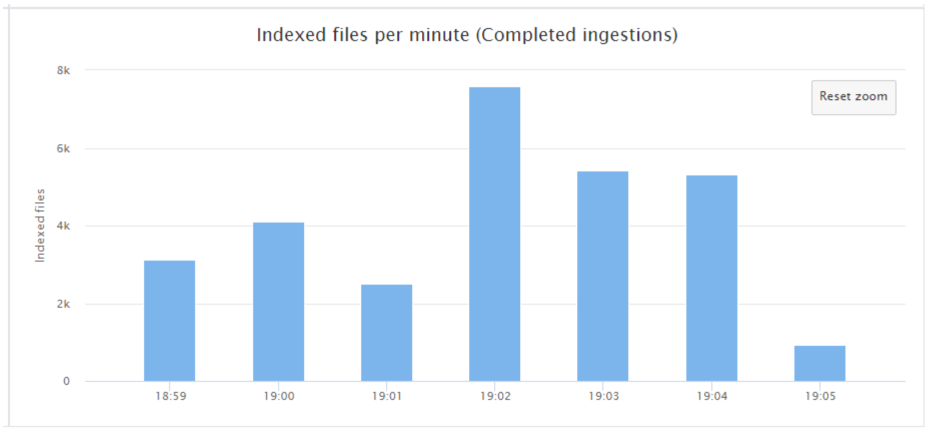


Figure 7.18.: Circulator performance test 9

Test 10

Purpose. Test how five *Circulator* instances perform over a longer test time.
Test time frame. 19:10 – 19:22
Observation time frame. 19:11 – 19:21
Uploads per minute within observation time frame. (visualized in Figure 7.19)

- Total: 5962, 3604, 6596, 7619, 3953, 5953, 6719, 4849, 6269, 6308, 11972
- Average: 6345.82
- Median: 6269

Evaluation. The average number of indexed files per minute increased with a test runtime of 11 minutes over 5 minutes. This indicates that *Amazon Keyspaces* scaled automatically in the background. The high variance between the minutes indicates a huge number of failed insertions, comparing the maximum (12000) to the minimum (3600).

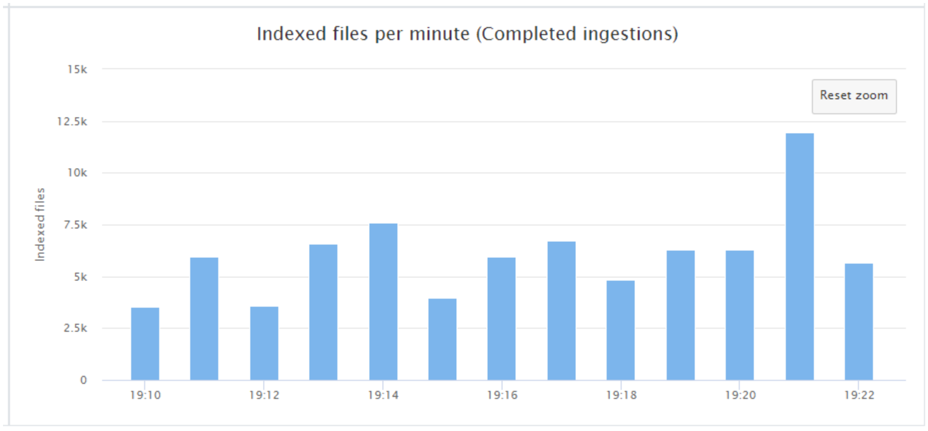


Figure 7.19.: Circulator performance test 10

Conclusion. Figure 7.20 shows the average numbers of files indexed per minute for the different test setups. One *Circulator* instance is able to process about 2000 files per minute on the utilized machine. The graph clearly shows that more *Circulator* instances can get more ingestions done. However, the clear bottleneck in the current prototype implementation is the *Amazon Keyspaces* managed Cassandra database service. With more than one *Circulator* instances, write requests start to fail hence files are processed multiple times until the requests succeed. With longer testing times, the automatic scaling mechanism of *Amazon Keyspaces* shows its effect. Because of that, the average file indexing numbers are higher with increased observation time. This phenomenon starts at three *Circulator* instances which is likely caused that the higher number of failed requests lead to increased database scaling. Due to the *Amazon Keyspaces* bottleneck, it is not possible to reveal the real limits of how many files can be indexed on the current Big KG-OLAP lakehouse setup.

Data ingestion performance evaluation conclusion. Unfortunately, the performance evaluation does not result in a single number stating how many files can be ingested per minute due to underlying constraints and opaque scaling mechanisms. However, with two *m5.4xlarge* nodes on the *Amazon EKS* cluster, one running a *Surface* instance and the

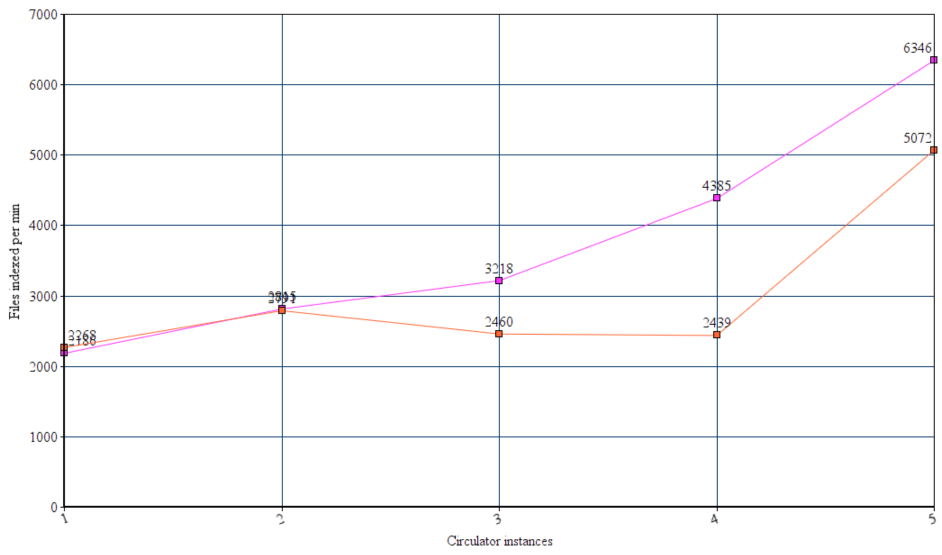


Figure 7.20.: Circulator performance test overview

other one operating five *Circulator* instances, a throughput between 6000 and 9000 files per minute can be achieved once the *Amazon Keyspaces* services scaled itself up. The lower limit of 6000 is caused by the *Circulator* while the maximum number of 9000 is the upload limit of the *Surface*. Because both services can be scaled horizontally over multiple machines, there is no theoretical throughput limit provided that *Amazon S3* and the *Amazon Keyspaces* services scale as well. For example, with a third node running another *Surface* instance, the limiting factor of the *data ingestion* would be solely the *Amazon Keyspaces* service. In further tests, *Amazon Keyspaces* can be replaced with a bigger *Apache Cassandra* deployment to remove this bottleneck.

7.2. Contextual operations

The performance evaluation of the *contextual operations* is done with an automated script. The *Java context operation performance test* application runs configured *contextual operations* five times without cache and five times with cache and calculates the average response times for both scenarios before changing the setup. The test setup use 1, 2, 4 and 8 *Bed* instances to test the horizontal scalability of the RDF cube construction. The test setup looks as follows.

Big KG-OLAP lakehouse:

- Deployment according to appendix B.2 and section 6
- One *Amazon EKS* cluster node *m5.4xlarge*: 16 vCPU, 64GiB memory, Up to 10 Gigabit (Amazon Linux 2 x86_64 OS)
- 1 Surface instance
- 1 / 2 / 4 / 8 Bed instances
- 0 Circulator instances
- 0 Angular web UI instances
- 0 Angular web UI instances
- Total contexts stored: 9,224
- Total files stored: 7,169,243
- Total indices (files to context relations): 7,169,243
- Total *Object storage* size in MiB: 38,222

The *Context operations* used to evaluate the performance are shown in listing 7.1. They include four *Slice'n'Dice* operations as well as the same four operations but with a *Merge* clause. Each of the four different *Slice'n'Dice* operation targets a number of contexts on a different order of magnitude. This also implies that the number of resulting quads and size in bytes increases on the same order. *Q1/Q5* target 2 contexts and results in 2209 quads or 0.6 MiB. *Q2/Q6* select 12 contexts and return 17,604 quads in 5 MiB. *Q3/Q7* extract 96 contexts which contain 152,832 quads and a total size of 43 MiB. The most complex queries are *Q4/Q8* which include 720 contexts and 1,085,595 quads in 319 MiB.

Listing 7.1: Context operations used for performance evaluation

```
Q1: SELECT time_day=2019-01-01 AND location_location=LOWS AND topic_feature
    =AircraftStand
Q2: SELECT time_day=2019-01-01 AND location_fir=LOVV AND topic_category=
    Routes
Q3: SELECT time_month=2018-02 AND location_territory=France AND
    topic_family=EnRoute
```

```
Q4: SELECT time_month=2018-02 AND location_territory=France
Q5: SELECT time_day=2019-01-01 AND location_location=LOWS AND topic_feature
    =AircraftStand ROLLUP ON topic_all, location_all, time_all
Q6: SELECT time_day=2019-01-01 AND location_fir=LOVV AND topic_category=
    Routes ROLLUP ON topic_all, location_all, time_all
Q7: SELECT time_month=2018-02 AND location_territory=France AND
    topic_family=EnRoute ROLLUP ON topic_all, location_all, time_all
Q8: SELECT time_month=2018-02 AND location_territory=France ROLLUP ON
    topic_all, location_all, time_all
```

The tests were conducted on a single machine. The *Surface* service as well as the *Bed* instances (1, 2, 4 or 8) ran on the same host. Every query was run 5 times (with cache and without cache) and the charts in Figures 7.21, 7.22, 7.23 and 7.24 show the average runtime of these 5 test runs.

In general, the charts clearly show that the larger the query gets in terms of considered contexts, the longer it takes to process them. However, there are significant runtime differences for large queries depending on the test setups.

The tests conducted without cached contexts show that 4 *Bed* instances clearly performed best, with less than a linear increase of runtime. On the opposite, the runtimes for tests with a single *Bed* instance increased far more than linearly. Tests conduced with 2 and 8 *Bed* instances are located somewhere in the middle on an almost linear scale, but rather close to the performance of 4 *Bed* instances.

Interestingly, in case of cached contexts, the previously observed conclusions are not true anymore. In both tests, Slice'n'Dice and Merge, the runtimes for the queries increased with the number of *Bed* instances. Especially for the largest queries Q4 and Q8, the difference is relatively significant because it almost doubled.

Scaling *Bed* instances is necessary to answer large queries (> 100 contexts) fast because file mapping into RDF is parallelized. For cached contexts, scaling merely leads to an increase in runtime, which is remarkable in relative numbers but less significant in absolute numbers. This indicates that simply for transferring data from the cache via *Bed* instances to the surface service, no scaling is necessary for this amount of load. Further tests are necessary to show the performance for queries with more than a thousand contexts and also how runtimes behave if load is spread across multiple hosts but not parallelized on a single machine.

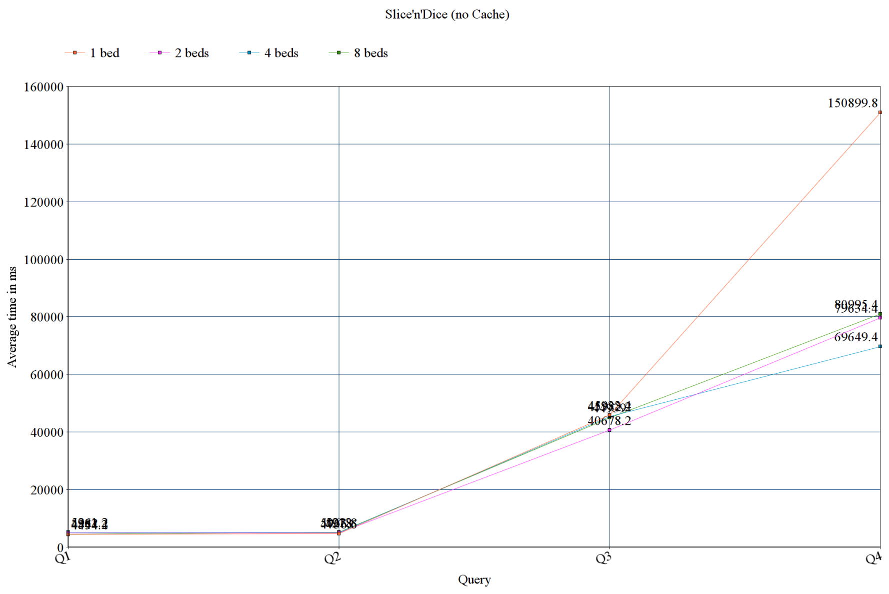


Figure 7.21.: *Slice'n'Dice* queries Q1-Q4 performance test overview without graph caching

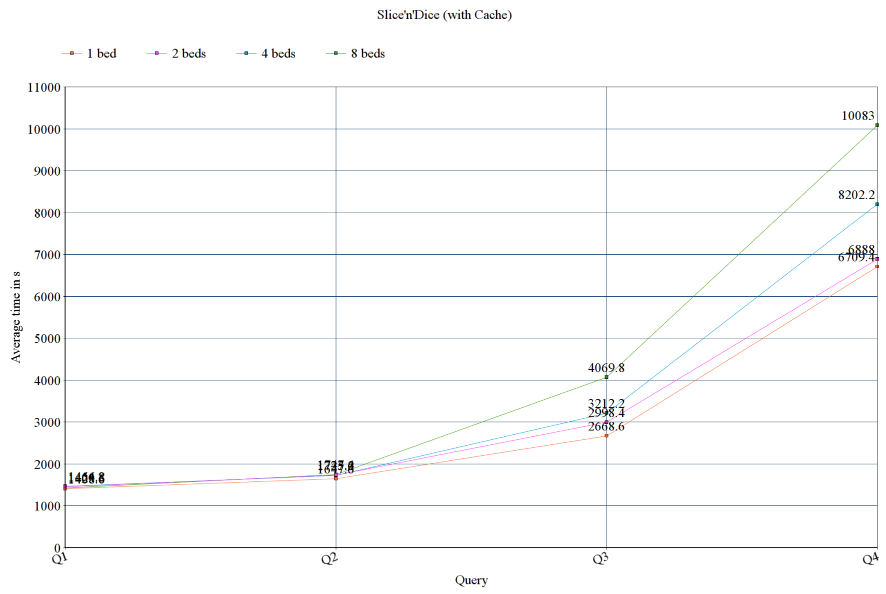


Figure 7.22.: *Slice'n'Dice* queries Q1-Q4 performance test overview with graph caching

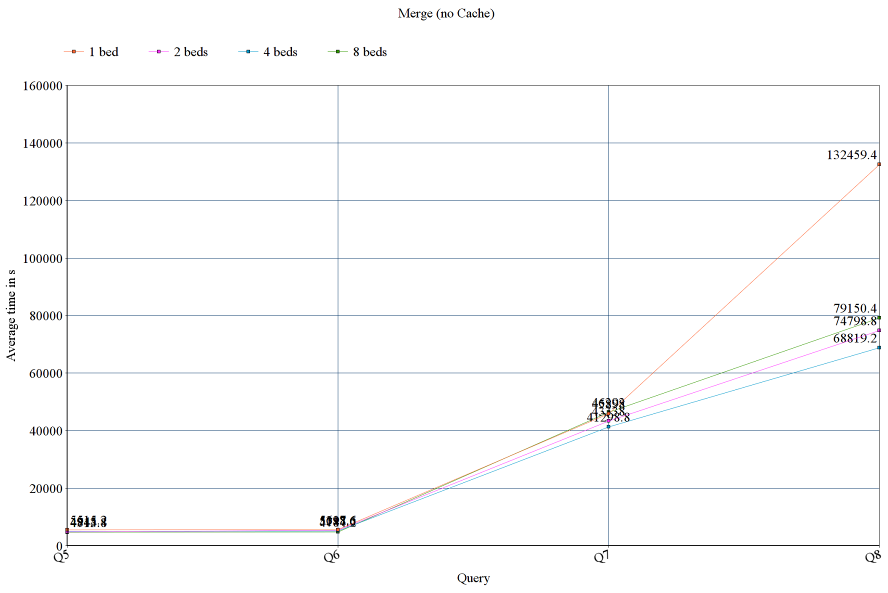


Figure 7.23.: *Merge* queries Q5-Q8 performance test overview without graph caching

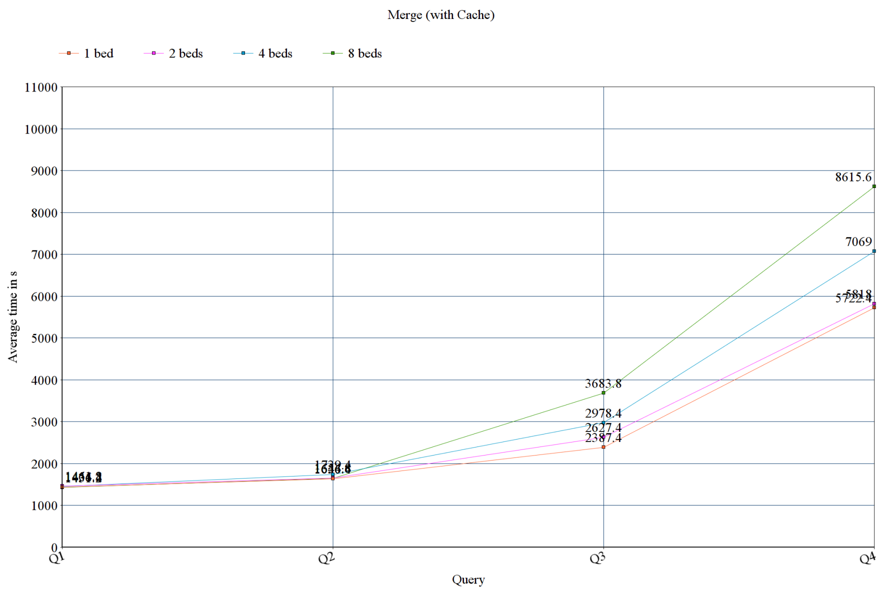


Figure 7.24.: *Merge* queries Q5-Q8 performance test overview with graph caching

8. Conclusion

The goal of this thesis was to identify big data requirements for KG-OLAP and develop an architecture that fulfills those requirements as well as KG-OLAP functionalities. In addition to that, a system is implemented that can be operated on modern cloud environments to lay the foundation for future research efforts in this direction. The resulting architecture and implementation is a cloud-native data lakehouse tailored to meet KG-OLAP requirements. As opposed to a central graph database, the Big KG-OLAP lakehouse is distributed and scalable by design. A key prerequisite to avoid a too large physical graph is the *Engine* concept which allows to index data on ingestion and construct the graph on-demand. This design enables scalable data ingestion and contextual operations.

The thesis gives a theoretical background on KG-OLAP and relevant concepts before Big KG-OLAP requirements are defined. Based on the defined requirements, the general architecture is proposed which forms the first contribution of the thesis. It is technology independent and lays the foundation for the following and future implementations. The second contribution is the concrete prototype implementation. Built on state-of-the-art technologies and AWS, the presented prototype demonstrates the feasibility of the architecture and its applicability for pilot briefings in ATM. A performance evaluation of the prototype's main functionalities data ingestion and contextual operations as the third contribution verified its scalability and rounds up the thesis.

Concluding, this thesis set a first step into the direction of Big KG-OLAP. Future work should focus on additional and more rigorous performance evaluations to reveal horizontal scalability limits of the system, especially due to the bottleneck experienced with AWS Keyspaces. Such tests will be the basis for further improvements of the reference architecture and prototype implementation.

Bibliography

[1] Amit Singhal. *Introducing the Knowledge Graph: things, not strings*. Google. May 16, 2012. URL: `https://blog.google/products/search/introducing-knowledge-graph-things-not/` (visited on 07/02/2022) (cit. on p. 1).

[2] Luciano Serafini and Martin Homola. "Contextualized knowledge repositories for the Semantic Web". In: *Journal of Web Semantics*. Reasoning with context in the Semantic Web 12-13 (Apr. 1, 2012), pp. 64–87. ISSN: 1570-8268. DOI: `10.1016/j.websem.2011.12.003` (cit. on p. 1).

[3] Dieter Steiner et al. "Semantic enrichment of DNOTAMs to reduce information overload in pilot briefings". In: *2016 Integrated Communications Navigation and Surveillance (ICNS)*. 2016 Integrated Communications Navigation and Surveillance (ICNS). Apr. 2016, 6B2–1–6B2–13. DOI: `10.1109/ICNSURV.2016.7486359` (cit. on p. 1).

[4] Christoph G Schuetz et al. "The Case for Contextualized Knowledge Graphs in Air Traffic Management". In: *CKGSemStats@ ISWC*. 2018, p. 8 (cit. on p. 1).

[5] Christoph G. Schuetz et al. "Knowledge Graph OLAP". In: *Semantic Web* 12.4 (Jan. 1, 2021). Publisher: IOS Press, pp. 649–683. ISSN: 1570-0844. DOI: `10.3233/SW-200419` (cit. on pp. 2, 12, 20, 21, 24, 25).

[6] Richard M. Keller et al. "Semantic representation and scale-up of integrated air traffic management data". In: *Proceedings of the International Workshop on Semantic Big Data - SBD '16*. the International Workshop. San Francisco, California: ACM Press, 2016, pp. 1–6. ISBN: 978-1-4503-4299-5. DOI: `10.1145/2928294.2928296` (cit. on pp. 2, 27, 28).

[7] Eamonn Brennan. *New traffic record set: 37,228 flights in one day*. July 2019. URL: `https://www.eurocontrol.int/news/new-traffic-record-set-37228-flights-one-day` (visited on 07/03/2022) (cit. on pp. 2, 27, 28).

[8] *Homepage of the EUR ICAO Data Management Group (DMG)*. URL: `https://eur-rodex.austrocontrol.at/iwxxm-intro.html` (visited on 07/05/2022) (cit. on p. 2).

[9] *FIXM*. URL: `https://www.fixm.aero/` (visited on 07/04/2022) (cit. on p. 2).

[10] *AIXM*. URL: `https://www.aixm.aero/` (visited on 07/04/2022) (cit. on pp. 2, 8).

[11] Seref Sagiroglu and Duygu Sinanc. "Big data: A review". In: *2013 International Conference on Collaboration Technologies and Systems (CTS)*. 2013 International Conference on Collaboration Technologies and Systems (CTS). May 2013, pp. 42–47. DOI: `10.1109/CTS.2013.6567202` (cit. on pp. 3, 27).

[12] Tam Harbert. *Tapping the power of unstructured data*. MIT Sloan. Feb. 1, 2021. URL: `https://mitsloan.mit.edu/ideas-made-to-matter/tapping-power-unstructured-data` (visited on 07/05/2022) (cit. on p. 3).

[13] Loris Bozzato and Christoph G Schuetz. "Towards Distributed Contextualized Knowledge Repositories for Analysis of Large-Scale Knowledge Graphs". In: *CILC*. 2020, p. 9 (cit. on pp. 3, 25).

[14] Pwint Phyu Khine and Zhao Shun Wang. "Data lake: a new ideology in big data era". In: *ITM Web of Conferences* 17 (2018). Publisher: EDP Sciences, p. 03025. ISSN: 2271-2097. DOI: `10.1051/itmconf/20181703025` (cit. on pp. 4, 12).

[15] Michael Armbrust et al. "Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics". In: *Proceedings of CIDR*. 2021, p. 8 (cit. on pp. 4, 13, 17, 28).

[16] Vasilios Andrikopoulos et al. "How to adapt applications for the Cloud environment". In: *Computing* 95.6 (June 1, 2013), pp. 493–535. ISSN: 1436-5057. DOI: `10.1007/s00607-012-0248-2` (cit. on p. 4).

[17] Nane Kratzke and Peter-Christian Quint. "Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study". In: *Journal of Systems and Software* 126 (Apr. 1, 2017), pp. 1–16. ISSN: 0164-1212. DOI: `10.1016/j.jss.2017.01.001` (cit. on pp. 4, 14, 31).

[18] Dieter Fensel et al. "Introduction: What Is a Knowledge Graph?" In: *Knowledge Graphs: Methodology, Tools and Selected Use Cases*. Ed. by Dieter Fensel et al. Cham: Springer International Publishing, 2020, pp. 1–10. ISBN: 978-3-030-37439-6. DOI: `10.1007/978-3-030-37439-6_1` (cit. on p. 7).

[19] Jeff Z. Pan et al. *Exploiting Linked Data and Knowledge Graphs in Large Organisations*. 2017. URL: https://link.springer.com/book/10.1007/978-3-319-45654-6 (visited on 07/06/2022) (cit. on p. 7).

[20] Linda van den Brink et al. "Linking spatial data: automated conversion of geo-information models and GML data to RDF". In: *International Journal of Spatial Data Infrastructures Research* 9.0 (Oct. 15, 2014). Number: 0, pp. 59–85. ISSN: 1725-0463 (cit. on p. 9).

[21] Surajit Chaudhuri and Umeshwar Dayal. "An overview of data warehousing and OLAP technology". In: *ACM SIGMOD Record* 26.1 (Mar. 1, 1997), pp. 65–74. ISSN: 0163-5808. DOI: 10.1145/248603.248616. URL: https://doi.org/10.1145/248603.248616 (visited on 07/06/2022) (cit. on p. 11).

[22] Christoph Quix and Rihan Hai. "Data Lake". In: *Encyclopedia of Big Data Technologies*. Ed. by Sherif Sakr and Albert Zomaya. Cham: Springer International Publishing, 2018, pp. 1–8. ISBN: 978-3-319-63962-8. DOI: 10.1007/978-3-319-63962-8_7-1 (cit. on p. 12).

[23] Roy Fielding. *Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST)*. 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm (visited on 08/05/2022) (cit. on p. 14).

[24] gRPC Authors. *gRPC*. gRPC. 2022. URL: https://grpc.io/ (visited on 08/05/2022) (cit. on p. 14).

[25] *Overview | Protocol Buffers*. Google Developers. July 2022. URL: https://developers.google.com/protocol-buffers/docs/overview (visited on 08/05/2022) (cit. on pp. 14, 15).

[26] Ruwan Fernando. *Evaluating Performance of REST vs. gRPC*. Medium. Apr. 7, 2019. URL: https://medium.com/@EmperorRXF/evaluating-performance-of-rest-vs-grpc-1b8bdf0b22da (visited on 08/05/2022) (cit. on p. 14).

[27] OCI. *Runtime Specification Principles*. June 9, 2020. URL: https://github.com/opencontainers/runtime-spec/blob/main/principles.md (visited on 06/15/2020) (cit. on p. 14).

[28] Matt Stine. *Migrating to Cloud-Native Application Architectures*. O'Reilly Media, 2015 (cit. on p. 14).

[29] Docker. *Home - Docker*. May 10, 2022. URL: https://www.docker.com/ (visited on 08/05/2022) (cit. on p. 15).

[30] Brendan Burns et al. *Kubernetes: Up and Running*. "O'Reilly Media, Inc.", Aug. 2, 2022. 327 pp. ISBN: 978-1-09-811015-4 (cit. on p. 15).

[31] *Concepts*. Kubernetes. June 22, 2020. URL: https://kubernetes.io/docs/concepts/ (visited on 08/15/2022) (cit. on p. 15).

[32] *The Service Mesh: What Every Engineer Needs to Know about the World's Most Over-Hyped Technology*. URL: https://buoyant.io/service-mesh-manifesto (visited on 08/15/2022) (cit. on p. 15).

[33] Ali Davoudian, Liu Chen, and Mengchi Liu. "A Survey on NoSQL Stores". In: *ACM Computing Surveys* 51.2 (Mar. 31, 2019), pp. 1–43. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3158661 (cit. on p. 15).

[34] Eric A Brewer. "Towards robust distributed systems". In: *PODC*. Vol. 7. 10.1145. Portland, OR. 2000, pp. 343477–343502 (cit. on p. 15).

[35] *What is a Message Queue?* Amazon Web Services, Inc. 2022. URL: https://aws.amazon.com/message-queue/ (visited on 08/15/2022) (cit. on p. 16).

[36] *IBM Docs*. July 24, 2022. URL: https://www.ibm.com/cloud/learn/message-queues?lc=de&mhsrc=ibmsearch_a&mhq=message%20queue (visited on 08/15/2022) (cit. on p. 16).

[37] M. Mesnier, G.R. Ganger, and E. Riedel. "Object-based storage". In: *IEEE Communications Magazine* 41.8 (Aug. 2003). Conference Name: IEEE Communications Magazine, pp. 84–90. ISSN: 1558-1896. DOI: 10.1109/MCOM.2003.1222722 (cit. on p. 16).

[38] Michael Factor et al. "Object storage: The future building block for storage systems". In: *In 2nd International IEEE Symposium on Mass Storage Systems and Technologies, Sardinia*. 2005, pp. 119–123 (cit. on p. 16).

[39] *Block Storage vs. Object Storage vs. File Storage: What's the Difference?* Qumulo. Feb. 1, 2022. URL: https://qumulo.com/blog/block-storage-vs-object-storage-vs-file-storage/ (visited on 08/21/2022) (cit. on p. 16).

[40] Daniel Tovarňák, Matúš Raček, and Petr Velan. "Cloud Native Data Platform for Network Telemetry and Analytics". In: *2021 17th International Conference on Network and Service Management (CNSM)*. 2021 17th International Conference on Network and Service Management (CNSM). ISSN: 2165-963X. Oct. 2021, pp. 394–396. DOI: `10.23919/CNSM52442.2021.9615568` (cit. on p. 17).

[41] Edmon Begoli, Ian Goethert, and Kathryn Knight. "A Lakehouse Architecture for the Management and Analysis of Heterogeneous Data for Biomedical Research and Mega-biobanks". In: *2021 IEEE International Conference on Big Data (Big Data)*. 2021 IEEE International Conference on Big Data (Big Data). Dec. 2021, pp. 4643–4651. DOI: `10.1109/BigData52589.2021.9671534` (cit. on p. 17).

[42] Khaled Dehdouh, Omar Boussaid, and Fadila Bentayeb. "Big Data Warehouse: Building Columnar NoSQL OLAP Cubes". In: *International Journal of Decision Support System Technology (IJDSST)* 12.1 (Jan. 1, 2020). Publisher: IGI Global, pp. 1–24. ISSN: 1941-6296. DOI: `10.4018/IJDSST.2020010101` (cit. on p. 18).

[43] Wenhao Chen et al. "An optimized distributed OLAP system for big data". In: *2017 2nd IEEE International Conference on Computational Intelligence and Applications (ICCIA)*. 2017 2nd IEEE International Conference on Computational Intelligence and Applications (ICCIA). Sept. 2017, pp. 36–40. DOI: `10.1109/CIAPP.2017.8167056` (cit. on p. 18).

[44] Alessandro Ferreira Leite et al. "Big data management and processing in the context of the system wide information management". In: *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*. 2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC). ISSN: 2153-0017. Oct. 2017, pp. 1–8. DOI: `10.1109/ITSC.2017.8317715` (cit. on p. 18).

[45] Ramakrishna Raju, Rohit Mital, and Daniel Finkelsztein. "Data Lake Architecture for Air Traffic Management". In: *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*. 2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC). ISSN: 2155-7209. Sept. 2018, pp. 1–6. DOI: `10.1109/DASC.2018.8569361` (cit. on p. 19).

[46] Alexander Behm et al. "ASTERIX: towards a scalable, semistructured data platform for evolving-world models". In: *Distributed and Parallel Databases* 29.3 (June 2011), pp. 185–216. ISSN: 0926-8782, 1573-7578. DOI: `10.1007/s10619-011-7082-y` (cit. on p. 19).

[47] W3C. *RDF 1.1 N-Quads*. Feb. 25, 2014. URL: `https://www.w3.org/TR/n-quads/` (visited on 07/06/2022) (cit. on p. 25).

[48] *Apache Jena - RDF Binary using Apache Thrift*. URL: `https://jena.apache.org/documentation/io/rdf-binary.html` (visited on 07/27/2022) (cit. on pp. 56, 70).

[49] Jeff Preshing. *Hash Collision Probabilities*. May 4, 2011. URL: `https://preshing.com/20110504/hash-collision-probabilities/` (visited on 12/27/2021) (cit. on p. 63).

[50] nathanielmanistaatgoogle. *Best message size for streaming large payloads? · Issue #371 · grpc/grpc.github.io*. GitHub. 2016. URL: `https://github.com/grpc/grpc.github.io/issues/371` (visited on 07/28/2022) (cit. on p. 70).

A. KGOQL Grammar

To perform *contextual operations* on the Big KG-OLAP lakehouse, a custom query language called KG-OLAP query language or KGOQL was defined. The section explains the KGOQL grammar and shows examples. Figure A.1 illustrates the KGOQL grammar definition and shows the possibilities. A KGOQL query must contain at least one *Slice'n'Dice* expression (sdExpr) while the *Merge* expression (mergeExpr) is optional. Multiple *Slice'n'Dice* expressions are connected with the *AND* keyword. In general, it is mandatory to put a space between a keywords such as *SELECT* or *AND* or expressions. *Merge* expressions are separated by a comma. Also, the grammar is not case sensitive.

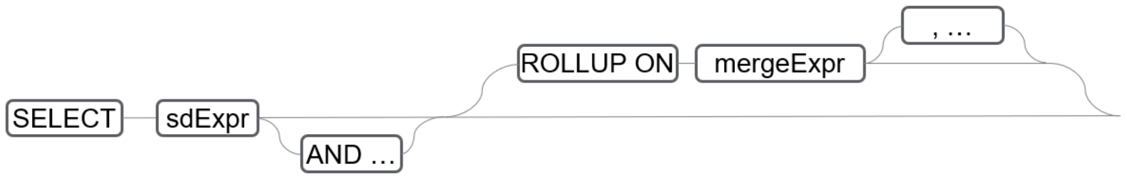


Figure A.1.: KGOQL grammar

Figure A.2 shows the grammar of a *Slice'n'Dice* expression. A * (asterisk) as *sdExpr* queries the entire cube and no more *sdExpr* can be concatenated with the *AND* keyword. The other possibility is to specify one or more *Members* to slice/dice at a certain cube coordinate. A *Member* consists of the complete identifier of a *Level* and a value. The complete identifier of a *Level* is the combination of the *Dimension* and the *Level* identifier separated by an underscore. Values must be specified in the correct format defined by the associated *Level*. Only one *Member* per dimension is allowed.

The *Merge* expression (mergeExpr) grammar is described in A.3. A *mergeExpr* is a complete identifier of a *Level* to which the query rolls up to. As described above, the complete identifier of a *Level* is the combination of the *Dimension* and the *Level* identifier separated by an underscore. It is also possible to use the *ALL* level in the *mergeExpr* to merge all

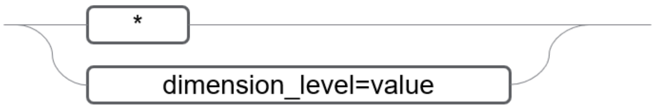


Figure A.2.: *Slice'n'Dice* expression

hierarchies of the respective dimension. Same as for the *sdExpr*, only one *Member* per dimension is allowed.

dimension_level

Figure A.3.: *Merge* expression

Listing A.1 contains four valid KGOQL queries that show the versatility of the query language. Those queries were used to test the Big KG-OLAP lakehouse prototype and demonstrate its applicability for the ATM pilot briefings use case. The available dimensions are *TIME*, *LOCATION* and *TOPIC*. The levels of the *TIME* dimensions are *YEAR*, *MONTH* and *DAY*. The *LOCATION* is split in the granularities *TERRITORY*, *Flight information region (FIR)* and *LOCATION*. A *TOPIC* can be drilled down from *CATEGORY* over *FAMILY* to *TOPIC*. As stated above, the *Levels* define the data formats for the given values such as *yyyy-MM-dd* for *TIME_DAY*.

Listing A.1: KGOQL query examples

```
SELECT * ROLLUP ON topic_all, location_all, time_all
SELECT time_month=2021-05 AND location_location=LOWS
SELECT time_year=2021 AND location_fir=EDMM AND topic_family=
    FlightRestrictions ROLLUP ON time_all, location_fir
SELECT time_day=2021-12-01 AND location_territory=Germany ROLLUP ON
    location_fir, topic_all
```

B. Installation

This appendix section includes installation documentation. It describes the structure of the GitHub repository[1] and the required steps to build and run the Big KG-OLAP lakehouse. In addition to that, the Kubernetes configuration for the deployment on AWS is included.

B.1. Repository structure, build and run

The Big KG-OLAP lakehouse project is hosted on GitHub[2] and uses *Gradle* for build, deploy and dependency management. The root level project is structured into the following sub-projects.

- *surface* : Frontend service that provides the public REST API and other functionalities
- *bed* : Backend service that is used internally to perform context operations
- *circulator* : Background processor that performs asynchronous jobs such as processing newly added files
- *svc-shared* : Shared code between *surface*, *bed* and *circulator*
- *shared-api* : API code that is used by every other module
- *aixm-engine* : Engine implementation for ‘NOTAM’ data
- *iwxxm-engine* : Engine implementation for ‘METAR’ data (not implemented yet)
- *web-ui* : Angular UI to query/ingest data

[1]https://github.com/davidHaunschmied/big-kgolap-lakehouse
[2]https://github.com/

- *file-uploader* : Script to fetch data from S3 and upload them to the surface REST endpoint
- *query-perf-test* : Script to make a complete contextual operations test run (handles scaling and restarts as well via "kubectl")

Prerequisites for local development.

- *JDK 11* or newer on *JAVA_HOME* environment variable
- *npm* version 12 or later for Angular UI build and development
- Apache Cassandra instance (Port needs to be configured, default is 9042)

  Sample docker command: *docker run -d -p 9042:9042 cassandra*
- Redis instance (Port needs to be configured, default is '6379')

  docker run –name redis-cache -d -p 6379:6379 redis

Configuration. To adjust the Big KG-OLAP lakehouse to your local environment, change variables such as the Apache Cassandra or Redis port accordingly in *svc-shared/src/main/resources/application-base.properties*. Most likely, the only parameter that needs to be adjusted is *lakehouse.storage-dir*. It defines the directory in which the files are stored.

Setup.

1. Run *gradlew build* in the root folder (Cassandra on port 9142 required for persistence tests, use Gradle option *-x test* to skip tests)
2. Run *npm install* in */web-ui*

Run.

1. Run *SurfaceApplication.java* and go to *http://localhost:8080/swagger-ui/index.html* to see the REST API
2. Run *BedApplication.java*
3. Run *CirculatorApplication.java*
4. Run *web-ui* by executing *npm start* in */web-ui* and go to *http://localhost:4200*

Deploy.

1. Make sure *docker* executable is on PATH and docker server is running
2. Run *gradle buildDockerImage*
3. Run services

   Surface: *docker run -p 8080:8080 jku-dke/big-kgolap-lakehouse/surface:latest*

   Bed: *docker run -p 9091:9091 jku-dke/big-kgolap-lakehouse/bed:latest*

   Circulator: *docker run jku-dke/big-kgolap-lakehouse/circulator:latest*

Pushing the docker images to Amazon ECR.

1. Make sure *aws* executable is on PATH
2. Log in with *aws ecr get-login-password –region us-east-2 | docker login –username AWS –password-stdin <acc>.dkr.ecr.us-east-2.amazonaws.com*
3. Run *gradle pushDockerImage*

B.2. Amazon EKS deployment configuration

This deployment configuration was used to deploy the three Spring services *Surface*, *Bed* and *Circulator* as well as the Angular web UI on an Amazon EKS cluster. The first resource creates is the *big-kgolap-lakehouse* namespace in which the following ones are logically included. The four deployments (three services plus web ui) follow a similar configuration. Basically, there is one replica (= one pod) per deployment. Further scaling is subject to the respective performance tests. The container images are stored on Amazon ECR (Elastic Container Registry) and pulled on every restart. This policy ensures that the latest version is running. The environment variable *SPRING_PROFILES_ACTIVE=aws* must be set on every Spring service to choose the correct Spring configuration. The *aws* configuration includes the correct endpoints to the required AWS services such as the *Amazon SQS* job queue, *Amazon S3* object storage or the *Amazon Keyspaces* Cassandra database. The *containerPorts* specified in the deployment configuration define the port on which the pods are accessible. A Kubernetes *Service* in front of the pods enables service discovery.

Every deployment is associated to a service expect the *Circulator* as it does not provide any interface but rather performs background processing tasks. For the two external accessible deployments *Surface* and *Web UI*, the service type *NodePort* is used. This allows to access the corresponding pods from outside the cluster. An internet-facing *Ingress* configured as ALB (Application Load Balancer) generates an external DNS name on top of the *NodePort* and provides ISO-OSI application layer load balancing. The*Bed* service must not be accessible from the outside hence *ClusterIP* is used as service type. The load balancing between *Bed* instances requires a service mesh such as *Linkerd* to be installed and that the client accesses the *Bed* via the service name rather than a pod IP. The environment variable *BED_HOST=big-kgolap-lakehouse-bed-svc* ensures that the *Surface* uses the service name (line 52). One pitfall that is worth to mention is that an ALB *Ingress* has a default idle timeout of one minute at the time of writing. This results in aborted requests for *contextual operations* that take longer than one minute of processing. This can be solved by configuring an increased idle timeout (line 182).

Listing B.1: Amazon EKS Big KG-OLAP lakehouse deployment configuration

```
 1 apiVersion: v1
 2 kind: Namespace
 3 metadata:
 4   name: big-kgolap-lakehouse
 5---
 6 apiVersion: apps/v1
 7 kind: Deployment
 8 metadata:
 9   name: big-kgolap-lakehouse-web-ui
10   namespace: big-kgolap-lakehouse
11   labels:
12     app: big-kgolap-lakehouse-web-ui
13 spec:
14   replicas: 1
15   selector:
16     matchLabels:
17       app: big-kgolap-lakehouse-web-ui
18   template:
19     metadata:
        labels:
          app: big-kgolap-lakehouse-web-ui
      spec:
        containers:
          - name: web-ui
            image: acc.dkr.ecr.us-east-2.amazonaws.com/big-kgolap-lakehouse/web-ui:0.2
            imagePullPolicy: Always
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: big-kgolap-lakehouse-surface
  namespace: big-kgolap-lakehouse
  labels:
    app: big-kgolap-lakehouse-surface
spec:
  replicas: 1
  selector:
    matchLabels:
      app: big-kgolap-lakehouse-surface
  template:
    metadata:
      labels:
        app: big-kgolap-lakehouse-surface
    spec:
      containers:
        - name: surface
          image: acc.dkr.ecr.us-east-2.amazonaws.com/big-kgolap-lakehouse/surface:0.2
          imagePullPolicy: Always
          env:
            - name: SPRING_PROFILES_ACTIVE
              value: "aws"
          - name: BED_HOST
            value: "big-kgolap-lakehouse-bed-svc"
        ports:
          - containerPort: 8080
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: big-kgolap-lakehouse-bed
  namespace: big-kgolap-lakehouse
  labels:
    app: big-kgolap-lakehouse-bed
spec:
  replicas: 1
  selector:
    matchLabels:
      app: big-kgolap-lakehouse-bed
  template:
    metadata:
      labels:
        app: big-kgolap-lakehouse-bed
    spec:
      containers:
        - name: bed
          image: acc.dkr.ecr.us-east-2.amazonaws.com/big-kgolap-lakehouse/bed
              :0.2
          imagePullPolicy: Always
          env:
            - name: SPRING_PROFILES_ACTIVE
              value: "aws"
            - name: BED_HOST
              value: "big-kgolap-lakehouse-bed-svc"
          ports:
            - containerPort: 9091
```

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: big-kgolap-lakehouse-circulator
  namespace: big-kgolap-lakehouse
  labels:
    app: big-kgolap-lakehouse-circulator
spec:
  replicas: 1
  selector:
    matchLabels:
      app: big-kgolap-lakehouse-circulator
  template:
    metadata:
      labels:
        app: big-kgolap-lakehouse-circulator
    spec:
      containers:
       - name: circulator
         image: acc.dkr.ecr.us-east-2.amazonaws.com/big-kgolap-lakehouse/
             circulator:0.2
         imagePullPolicy: Always
         env:
          - name: SPRING_PROFILES_ACTIVE
            value: "aws"
---
apiVersion: v1
kind: Service
metadata:
  name: big-kgolap-lakehouse-web-ui-svc
  namespace: big-kgolap-lakehouse
spec:
  selector:
```

```yaml
    app: big-kgolap-lakehouse-web-ui
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: NodePort
---
apiVersion: v1
kind: Service
metadata:
  name: big-kgolap-lakehouse-surface-svc
  namespace: big-kgolap-lakehouse
spec:
  selector:
    app: big-kgolap-lakehouse-surface
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
  type: NodePort
---
apiVersion: v1
kind: Service
metadata:
  name: big-kgolap-lakehouse-bed-svc
  namespace: big-kgolap-lakehouse
spec:
  selector:
    app: big-kgolap-lakehouse-bed
  ports:
    - protocol: TCP
      port: 9091
      targetPort: 9091
  type: ClusterIP
```

```
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  namespace: big−kgolap−lakehouse
  name: big−kgolap−lakehouse−web−ui−svc−ingress
  annotations:
    alb.ingress.kubernetes.io/scheme: internet−facing
    alb.ingress.kubernetes.io/target−type: ip
spec:
  ingressClassName: alb
  rules:
    - http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: big−kgolap−lakehouse−web−ui−svc
                port:
                  number: 80
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  namespace: big−kgolap−lakehouse
  name: big−kgolap−lakehouse−surface−svc−ingress
  annotations:
    alb.ingress.kubernetes.io/scheme: internet−facing
    alb.ingress.kubernetes.io/target−type: ip
    alb.ingress.kubernetes.io/load−balancer−attributes: idle_timeout.timeout_seconds
        =600
spec:
  ingressClassName: alb
```

```yaml
rules:
  - http:
      paths:
        - path: /
          pathType: Prefix
          backend:
            service:
              name: big-kgolap-lakehouse-surface-svc
              port:
                number: 8080
```