JOHANNES KEPLER
UNIVERSITY LINZ

IMPLEMENTING QUERY OPERATIONS FOR KNOWLEDGE GRAPH OLAP IN APACHE SPARK

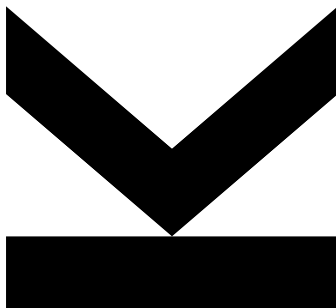
Author
Jennifer Klar, BSc

Submission
Institute of Business Informatics – Data & Knowledge Engineering

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

Assistant Thesis Supervisor
Bashar Ahmad, MSc

Month Year
February 2023



Master Thesis
to obtain the academic degree of

Master of Science
in the Master's Program

Business Informatics

JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at

KURZFASSUNG

Knowledge Graph OLAP kombiniert das Konzept von Knowledge Graphs (KG) mit einer multidimensionalen Sicht auf Daten, wie sie im Bereich des Online Analytical Processing (OLAP) angewandt wird. KG-OLAP-Würfel enthalten Wissen in Form von RDF-Tripel, welche durch hierarchisch strukturierte Dimensionen definiert werden und kontextabhängige Wissensgraphen bilden. Das Modell ermöglicht die Anwendung von kontextuellen und Graph-spezifischen Operationen auf den Daten für verschiedene Arten von Analysen. Eine SPARQL-basierte Implementierung erwies sich jedoch als nicht geeignet für große Datenmengen, was den Bedarf für eine skalierbare Implementierung unterstreicht. Ziel dieser Arbeit ist es daher, eine Implementierung bereitzustellen, die für große Datenmengen im Rahmen von KG-OLAP skalierbar ist und die erforderlichen Graph Operationen auf kontextualisiertem Wissen in Form von RDF-Daten durchführen kann. Folglich wird eine prototypische Implementierung unter Verwendung des Frameworks *Apache Spark* für verteilte Datenverarbeitung vorgeschlagen, die KG-OLAP-spezifische Graph Operationen auf RDF-Quadrupeln ausführt. Genauer gesagt wird das auf Spark aufbauende Graph Verarbeitungsframework *GraphX* verwendet. So werden RDF-Quadrupel auf die Graphendarstellung von Apache Spark GraphX abgebildet. Die Java-Implementierung ermöglicht dann die Konstruktion des Basisgraphen aus den RDF-Quelldaten, sowie die Durchführung folgender KG-OLAP-Graph Operationen auf dem Basisgraphen: *individual-generating abstraction, triple-generating abstraction, value-generating abstraction, reification* und *pivot.* Die Funktionalität und Anwendbarkeit des Spark-basierten Prototyps wird in Experimenten mit einem bereitgestellten großen Benchmark-Datensatz mit Daten aus dem Bereich Flugverkehrsmanagement (engl. air traffic management) demonstriert.

ABSTRACT

Knowledge Graph OLAP combines the concept of knowledge graphs (KG) and a multidimensional view on data as employed in online analytical processing (OLAP). KG-OLAP cubes contain knowledge in the form of RDF triples that are context-dependent, defined through hierarchically structured dimensions creating contextualized knowledge graphs. The model enables contextual and graph operations on the data for various kinds of analyses. A SPARQL-based implementation has proven not to be applicable for big volumes of data, accentuating the need for a scalable implementation. This thesis therefore aims at providing an implementation that is scalable for large amounts of data within the KG-OLAP setting and can perform the required graph operations on contextualized knowledge in the form of RDF data. Consequently, a prototypical implementation using the distributed processing framework *Apache Spark* is proposed that executes KG-OLAP graph operations on RDF quadruples. More specifically, the graph processing framework *GraphX* built on top of Spark is used. Thus, RDF quadruples are mapped to the Apache Spark GraphX graph representation. The Java implementation then allows for the construction of the initial graph from the RDF source data as well as for performing the following KG-OLAP graph operations on the base graph: *individual-generating abstraction, triple-generating abstraction, value-generating abstraction, reification* and *pivot*. The functionality and applicability of the Spark-based prototype is demonstrated in experiments on a provided large benchmark dataset containing data regarding air traffic management.

TABLE OF CONTENTS

1. Introduction 7
2. Background 9
2.1. Online Analytical Processing (OLAP) 9
2.2. (Contextualized) Knowledge Graphs 10
2.3. Knowledge Graph OLAP 11
2.4. RDF and SPARQL 12
2.5. Big Data Processing 13
2.6. Apache Spark 16
2.6.1. Spark Features 16
2.6.2. Spark Architecture 17
2.6.3. Further Spark Terminology and Concepts 19
2.6.4. Memory Structure 22
2.6.5. Lazy Evaluation 23
2.6.6. Performance 24
2.6.7. Apache Spark GraphX 29
3. Related Work 32
3.1. Graphs and Hadoop 32
3.2. Using Spark for RDF Processing 33
3.3. OLAP over Graphs and RDF Data 37
3.4. Summary of Current Literature 41
4. Data Model 42
4.1. Running Example 42
4.2. Technology Decisions 43
4.3. RDF and GraphX Mapping 46
4.3.1. Subjects and Objects 46
4.3.2. Predicates 46
4.3.3. Identifiers 47
4.3.4. Context 47
4.3.5. Type Property 48
4.3.6. Resulting RDF and GraphX Mapping 49
4.3.7. Example Graph Construction 50
5. Data Processing 51

5.1. GraphX Graph Construction 51
5.2. KG-OLAP Operations 53
5.2.1. Triple-Generating Abstraction 55
5.2.2. Individual-Generating Abstraction 59
5.2.3. Value-Generating Abstraction 63
5.2.4. Reification 70
5.2.5. Pivot 74
5.2.6. Slice-and-Dice 78
5.2.7. Merge 80
5.3. Implementation Details 82
5.3.1. Class Relation 82
5.3.2. Classes Vertex, Resource and Literal 82
5.3.3. Class GraphGenerator 82
5.3.4. Interface Transformation 85
5.3.5. Class Reification 86
5.3.6. Class Pivot 87
5.3.7. Class ValuegeneratingAbstraction – aggregate property values 88
5.3.8. Class IndividualGeneratingAbstraction – group by property 88
5.3.9. Class TripleGeneratingAbstraction – replace by grouping 89
5.3.10. Class Utils 89
5.3.11. Properties Files 90
5.4. Use of Program 91
6. Performance Experiments 93
6.1. Experiment Setup and Data 93
6.2. Experiment Results 95
7. Conclusion 97
8. References 99

LIST OF FIGURES

Figure 1: Spark Architecture adapted from (Zhang et al., 2018) 18
Figure 2: Narrow dependencies adapted from (Karau & Warren, 2017) 21
Figure 3: Wide dependencies adapted from (Karau & Warren, 2017)........ 22
Figure 4: Property graph example 29
Figure 5: Example vertex table........ 30
Figure 6: Example edge table........ 30
Figure 7: RDF – GraphX Mapping 49
Figure 8: Resulting edge collection........ 50
Figure 9: Resulting vertex collection 50
Figure 10: Base graph construction........ 51
Figure 11: Graph transformation 53
Figure 12: Triple-generating abstraction execution steps 55
Figure 13: Triple-generating abstraction example graph........ 56
Figure 14: Individual-generating abstraction execution steps 59
Figure 15: Individual-generating abstraction graph example........ 60
Figure 16: Value-generating abstraction execution steps 63
Figure 17: Value-generating abstraction graph example 64
Figure 18: Reification execution steps........ 70
Figure 19: Reification graph example 71
Figure 20: Pivot execution steps 74
Figure 21: Pivot graph example........ 75

LIST OF TABLES

Table 1: Spark performance settings by Nguyen et al. (2017) 25
Table 2: Triple-generating abstraction steps... 57
Table 3: Individual-generating abstraction steps.. 61
Table 4: Value-generating abstraction steps 65
Table 5: Reification steps 72
Table 6: Pivot steps.. 76
Table 7: Experiment results.. 95

1. Introduction

The concept of *Knowledge Graph OLAP (KG-OLAP)* was introduced by Schuetz et al. (2021). Their work combines both knowledge graphs and online analytical processing (OLAP) operations into a framework. Hereby, a multidimensional model – a cube – is constructed, containing hierarchically structured contexts comprising data in different levels of granularity. Each cell of the OLAP cube represents a context defined through certain dimensional levels. The knowledge contained in those cells is then represented in the form of RDF triples which can either be more general or more specific knowledge with respect to the scope depending on their context's granularity. With this system in place, operations like roll-up, abstraction, merge and slice-and-dice can then be performed to transform and analyse the data. Usually, the first choice for working with RDF data is using SPARQL as query language. Schuetz et al. (2021) also tested their concept on a SPARQL-based implementation which led to performance problems when having to deal with growing amounts of data.

Due to the shortcomings of the SPARQL-based implementation and the rising importance of big data as a topic to be aware of when designing data processing systems, there is a need for an implementation of the KG-OLAP concept that is able to process large volumes of data in acceptable timespans. With growing amounts of data traditionally used technologies and frameworks are often no longer capable to handle the workload (Jacobs, 2009). Therefore, new, dedicated big data frameworks are being developed and used more and more, relying on optimizations such as distributed processing to keep up with the demand of staying efficient.

A popular framework for big data processing is *Apache Spark* which was chosen for the implementation of the prototype described within this thesis. Spark is a distributed processing framework that aims to stay performant even when having to work with large amounts of data. KG-OLAP operations like pivot, reification, and abstraction were therefore implemented using Apache Spark and experiments conducted to determine Spark's applicability and efficiency for the KG-OLAP use case. The source code for the prototype and benchmark experiment results can be viewed at http://files.dke.uni-linz.ac.at/publications/mt2301/implementation.zip.

This thesis aims to investigate whether Apache Spark is suitable for processing contextualized knowledge graphs and performing KG-OLAP operations on the data. Within that, also the applicability of Spark for RDF data in general is investigated. Furthermore, the performance of the proposed KG-OLAP implementation is examined.

Based on the stated problem and resulting aims of this thesis, the following research questions were formulated to be answered:

- Can Apache Spark (GraphX) be used to process RDF data and perform KG-OLAP query operations?
- What is the performance of a KG-OLAP implementation using Apache Spark (GraphX)?

The first step towards answering the stated research questions was exploration of the state of the art of the field and related work. In particular, research on the basic concepts and technologies within this thesis relevant for the implementation of KG-OLAP in Apache Spark was conducted. Subsequently, a literature review was performed to find out whether there are already existing similar approaches that process RDF data or contextualized knowledge graphs with the help of Apache Spark or Apache Spark GraphX.

Following the research on the state of the art and related work, a prototype was implemented in Java using the Apache Spark GraphX framework. The implementation processes RDF data by performing KG-OLAP operations and transformations on the data using methods provided by Spark libraries. The resulting prototype was then used to transform the same data used in the evaluation of the SPARQL-based implementation by Schuetz et al. (2021). The results of those experiments allow for insights on whether the approach is applicable and feasible as an implementation of the KG-OLAP concept.

In creating the prototype, certain assumptions were made and the scope was constrained to certain aspects of the original concept by Schuetz et al. (2021). First, RDFS or OWL reasoning was not considered within the implementation described in this thesis. Reasoning would have to take place before loading the RDF triplets into the repository where graph operations are then performed. Another assumption made is that the data loaded is already cleaned, contains valid RDF and no inconsistencies with respect to any RDFS expressions are present.

The focus of the implementation was on graph operations whereas merging and slice-and-dice-operations were out of scope of the prototype described within this thesis. Merge and slice-and-dice operations may be performed before the data is then loaded into the Spark GraphX graph. Those operations could, for example, be executed using a data lakehouse for KG-OLAP like in Haunschmied (2022). The potential Spark implementation and algorithmic structure of the merge and slice-and-dice operations is, however, still sketched within this thesis. The resulting implementation and performance experiments consider the pivot, reification, triple-generating abstraction, individual-generating abstraction and value-generating abstraction operations.

The remainder of this thesis is structured as follows. Chapter 2 presents background information on the state of the art of relevant technologies and concepts. Chapter 3 presents related work using Apache Spark GraphX, RDF data, OLAP and distributed processing of data in general. Afterward, in Chapter 4, the data and data model used for the representation in Apache Spark GraphX within this thesis are explained. The data model especially focusses on the mapping between the source RDF data and the GraphX graph. Then, Chapter 5 presents all KG-OLAP query operations implemented. Next, this thesis' Java implementation is described in more technical terms before then in Chapter 6 the experimental setup and results of applying the implemented transformations and query operations described within this thesis are shown. Finally, in Chapter 7, findings of this thesis are discussed, and the research questions are answered. Furthermore, potential future work, regarding this thesis’ prototype and the KG-OLAP implementation is outlined.

2. Background

This section of the thesis deals with the fundamentals needed for implementing the Spark KG-OLAP prototype. Hereby the models and concepts that are used such as OLAP, knowledge graphs and so on are first described as well as KG-OLAP itself. Then, background information about parallel and big data processing in general is given as well as an introduction of the used framework Apache Spark.

2.1. Online Analytical Processing (OLAP)

The abbreviation OLAP stands for online analytical processing which is a paradigm especially designed for analytical queries. This stands in contrast to the concept of OLTP (online transaction processing) which is concerned with operational transactions rather than large analyses on data.

OLAP systems are usually comprised of dimensions and facts that define a multidimensional model – a data cube. Hereby dimensions are organized in hierarchies with certain numbers of levels. The dimensions and their levels control in which granularity or detail one can view the measures (the facts) that are contained within the cells of the cube. Those facts are represented by numeric values. Depending on the level that the data should be viewed at, the fact values are aggregated or rolled-up (for example by counting or summing) to a certain hierarchy level of each cube dimension. Additionally, operations such as slicing or dicing the cube can be done for reducing the amount data to be analysed beforehand. For further details and precise definitions of data warehousing and OLAP itself it is referred to (Vaisman & Zimányi, 2014).

A very simple use case for an OLAP system and operations could for example be sales data, which is also used in the work of Chaudhuri & Dayal (1997). Hereby a numeric measure representing the number of sales of a company could be used as a measure. Dimensions for this fact could then be the location (consisting of a hierarchy containing cities and countries) and the time period (consisting of a hierarchy of date, month and year). Now all dimensions together determine the measure, meaning that every combination of values of the dimension leads to a different aggregation of the number of sales. For example, there is one exact total number of sales for the 7th of August 2021 at Linz where the value for the time dimension is the 7th of August at the date level and the value for the location dimension is Linz at the city level. When then wanting to look at sales at higher levels there is for example also another exact total number of sales for the month of August in Austria and so on. This would be a roll-up from the city to country level and the date to month level. With this system in place, numerous different kinds of analysis are possible.

2.2. (Contextualized) Knowledge Graphs

The second fundamental concept used within KG-OLAP and necessary to understand for the implementation of this thesis' data processing are knowledge graphs.

A lot of research has been done in recent years on knowledge graphs and how to represent knowledge using them, especially in the context of the semantic web. This led to the formation of a multitude of different definitions and descriptions of the term.

Knowledge graphs are for example being described as networks containing entities and their types and properties as well as relations between those entities as in work by Krötzsch & Weikum (2016) or in other terms as "*a set of interconnected typed entities and their attributes*" (Gomez-Perez et al., 2017).

As stated by Hogan et al. (2021) the advantage of representing knowledge as a graph rather than a table in a relational database is that graphs provide an abstraction that can be used independently of the domain. This independence stems from the fact that graphs rely on simple vertices and connections (relationships) between those vertices, which is a natural way of describing data in a lot of different fields. Furthermore, there is no fixed schema needed which increases flexibility when the knowledge graph is extended.

Another key idea and extension to knowledge graphs used in KG-OLAP concerning RDF data representation is the contextualization of graphs. Contextualization here is described as additionally adding meta information to the knowledge graph. Such metadata could for example be time or location as it is used by Schuetz et al. (2021). This idea of contextualization is for instance used in the Contextualized Knowledge Repository (CKR) as described by Serafini & Homola (2012). In CKR, contexts are essentially considered a *box* that contain different statements which are valid for a certain collection of dimensions. Therefore, there are two parts that the knowledge can be split into. The first part is knowledge about objects (which are the statements within the box), the second part is the metadata or meta-knowledge about the box itself (which is represented by the dimensions). Since the dimensions themselves are hierarchically structured, this also means that some contexts contain statements that are more general, and others contain more specific knowledge creating coverage relations between the contexts and therefore between the knowledge they are comprised of.

When using GraphX it is also necessary to mention a specific form of graphs called *property graph*. When modelling such a graph, the nodes or vertices are regarded as entities, edges represent the relationship between those entities and properties are used to describe a certain feature of this relationship or the entities themselves. So it could be said that a property graph is *"a labelled multigraph where both vertices and edges may contain pairs of the form property–value"* (Angles, 2018). This means that all properties that are associated with the edges or the vertices are key–value pairs where values can be of any data type or structure. Moreover, all entities may have one or more labels associated with them as described by Tomaszuk (2016). In later sections concerning the implementation of this thesis' graph processing, the property

graph concept is explained again in more detail and specifically how it is implemented and used by Apache Spark and GraphX.

2.3. Knowledge Graph OLAP

Now combining the mentioned concepts of OLAP and contextualized knowledge graphs, the key term KG-OLAP of this thesis will be explained.

Knowledge Graph OLAP (KG-OLAP) as introduced by Schuetz et al. (2021) is a concept containing a multidimensional model as well as query operations that can be used to analyse data represented as knowledge graphs. The model – the KG-OLAP cubes – contains knowledge that is ordered into a hierarchy by using hierarchically structured contexts similar to the CKR concept by Serafini & Homola (2012). Hereby, every cell in this KG-OLAP cube represents one of those contexts. Within each context (the cell) data is contained in the form of RDF triples. Those triples replace the usually used numeric measures in traditional OLAP cubes in data warehousing. The hierarchical ordering makes it possible to include knowledge that is more specific but also knowledge that is more general.

There are two distinct kinds of operations and queries that can be performed on the KG-OLAP cube as stated in the framework definition. On the one hand, there are contextualized query operations, on the other hand, there are also graph operations. The most common examples for such query operations are *merge* and *abstract*. The merge operation, which is a contextualized operation, takes knowledge from different cells – or contexts – and merges them into one. An abstraction operation, which is a graph operation, then takes individuals from within the merged contexts of the cube and replaces them with more abstract ones generating a more general view on the knowledge represented. Here, there are different ways of doing so available which will be described later.

When compared to traditional OLAP, the key difference in KG-OLAP is that knowledge or facts are not described by using numeric values but rather RDF triples. Those triples and the knowledge they represent is then valid in the context that is characterized by the cell and its dimensional values.

For more details on KG-OLAP, formal definitions of the concepts and query operations it is referred to the publication by Schuetz et al. (2021).

2.4. RDF and SPARQL

As in this work and the KG-OLAP concept RDF is used to represent knowledge contained in the KG-OLAP cube cells, in this section, a brief overview of the Resource Description Framework (RDF) will be given. Furthermore, the mostly used query language for RDF data – SPARQL – will be described shortly, since the KG-OLAP implementation by Schuetz et al. (2021) is based on SPARQL.

As described in the work of Agathangelos et al. (2018), in RDF, data and therefore knowledge is represented as triples. Those triples consist of a subject, a predicate and an object. The predicate indicates the relation between the subject and the object. Hereby the basic elements within the data can be represented as URIs (or IRIs) or literals. Tomaszuk (2016) describes URIs or IRIs as identifiers that are unique and may be used globally to identify a certain resource. Literals are lexical values such as Strings or numbers.

In addition to RDF there is RDFS (Resource Description Framework Schema) which – according to the description used by Agathangelos et al. (2018) – adds semantics to the RDF data. The framework includes rules which help to generate new triples that are implicitly given through the already existing, explicitly given information. A collection of triples and the added semantics can then be represented as a directed graph. Nodes in this graph are the subjects and the objects. The labelled edges between the nodes represent the predicates – the relationships – between them as already explained.

Querying RDF data is also described by Agathangelos et al. (2018) in their work. Whenever RDF data is to be queried mostly the language SPARQL is used as recommended by the World Wide Web Consortium W3C (W3C, 2013). SPARQL is a graph matching language where a query consists of triple patterns. Those patterns are essentially RDF triples including subjects, objects and/or predicates that may be either a variable or a concrete literal. Simply put, when querying a certain graph or collection of graphs those patterns are compared against the triples in the dataset. Whenever a triple contained in the graph matches the pattern, those statements are returned as the query result.

More specifically, according to Agathangelos et al. (2018), the "*SPARQL query*" itself "*consists of three parts: pattern matching, solution modifiers and output*" (Agathangelos et al., 2018). Pattern matching may – additionally to the triple pattern itself – include optional parts, unions of multiple patterns, nesting or filtering for retrieving matching patterns in the graph data. Solution modifiers then may modify the computed query output by applying operations like *distinct*, *order*, *limit* and so on. As an output the query can either return a yes-or-no-answer, a certain number of values that match the pattern or possibly a calculation of them. A query may also result in the construction of new triples that were not present in the graph before, or it may return a description of a resource.

2.5. Big Data Processing

Since the implementation approach in this paper is focused on processing large amounts of data, the term *big data* and concepts within big data processing are now explained briefly in this section.

The term *big data* – as it is used nowadays – has first been mentioned in a published article in 1997 by Cox & Ellsworth where the term was used to described data that is large in size and challenges main memory, local and remote disk size. Therefore, big data is generally defined as data which is of such size that it cannot be processed in an efficient manner anymore by databases and technologies that are typically used (Jacobs, 2009).

Even as early as 2009 Jacobs already suggested using parallel processing when trying to work with large amounts of data. Whenever data is processed in parallel and a distributed way – meaning multiple compute nodes only process part of the whole dataset – it is important that communication required between the partitions or nodes is kept to a minimum. The more interaction between partitions needed, the worse the application may perform according to the author.

In the field of big data, often NoSQL databases are mentioned alongside parallel processing as discussed in a paper by Casado & Younas from 2015. As described by the authors, such databases are designed to offer horizontal scalability by being designed to be able to distribute and partition data and tasks. Some of those NoSQL concepts include for example "*key–value stores, document databases, column-oriented databases*" as well as "*graph databases*" (Casado & Younas, 2015). Concerning data processing, Casado & Younas (2015) also mention different approaches including *Hive*, *Pig*, *Cascading*, *Spark* and so on. Those are specially designed to handle big data.

All in all, literature around big data shows that there is a lot of research as well as implementations done in the field of distributed and parallel processing and different approaches might lead to favourable results. Some of those proposed frameworks and concepts are now described briefly.

Hadoop

In literature when researching specific solutions and technologies working with big data, one often comes across *Hadoop*. Apache Hadoop – as described by Bhosale & Gadekar (2014) – is a framework that allows data to be processed in a distributed way. It was originally built upon the *MapReduce* paradigm developed by Google. The Hadoop ecosystem contains four main parts with the help of which parallel and distributed data processing within a Hadoop cluster can be achieved (IBM Cloud Education, 2016). Those four parts are the *HDFS* (Hadoop Distributed File System), *YARN* (Yet Another Resource Negotiator), the previously mentioned *Hadoop MapReduce* and *Hadoop Core* containing necessary libraries that are needed.

As described by Bhosale & Gadekar (2014), *HDFS* is a storage system where data of large quantities can be stored in a fault tolerant way. As Hadoop works with distributed processes,

it creates clusters which the data is then allocated to after having been broken up into distinct parts. The data is also usually replicated among more than one node, thus ensuring fault tolerance. Whenever one part of the cluster fails, there are other machines within the cluster that still contain the lost data. Those nodes then can keep working and therefore no interruptions of the data processing task due to failures of other nodes in the cluster occur.

Another core concept within the Hadoop ecosystem already mentioned is *MapReduce* upon which it originally was built. MapReduce is an engine concerned with the processing of data that has been distributed or split up and can therefore be worked on in parallel as illustrated by Bhosale & Gadekar (2014). Simply put, the *Map* part of MapReduce transforms key–value pairs of data into another set of key–value pairs. The *Reduce* part is a task that then merges the generated mapped values that belong to the same key and applies a certain function. The output of MapReduce can then for example again be written back into a HDFS.

YARN, as another core component of the Hadoop ecosystem is used as a scheduler and manages resources across the used Hadoop cluster as described by IBM Cloud Education (2016). With the help of YARN, it can be assured that CPU and memory are allocated accordingly within the cluster between all nodes.

Hadoop vs. Spark

Closely linked to Hadoop one also encounters *Apache Spark* quite frequently in literature. Spark was developed to increase performance and speed of Hadoop. The framework is therefore suitable to work with Hadoop data and may be run on a Hadoop cluster using YARN. However, it is completely independent of the Hadoop system. This independence stems from the fact that Spark may also use its own cluster management and does not rely on any parts of Hadoop's ecosystem. Spark may use Hadoop's HDFS as a storage mechanism – if desired. However, when running Spark on a cluster using a different shared file system, Spark may be used in complete standalone mode as described in Apache Spark FAQ (2023).

Apache Spark was developed to be able to work with various kinds of operations and tasks including for example batch processing, iterative or interactive algorithms and queries or also real-time and streaming data or graph processing. Using Spark therefore may potentially solve problems that Hadoop faces with iterative and interactive processes according to Yang et al. (2016) since Hadoop is not optimized especially for iterative tasks.

A key difference between the two frameworks – Hadoop and Spark – as illustrated by Hong et al. (2017) is that Spark uses RDDs as its data abstraction model which will be explained in more detail later. In short, the data stored in RDDs can be kept in memory by Spark and directly reused. There is no need to store intermediate results which may improve overall performance. Hadoop and other systems that are based on MapReduce need to store data to disk after every processing step thereby increasing in- and output operations and thus runtime. Especially whenever the same data is used multiple times within a process and tasks are memory-bound, those performance advantages are prevalent according to Hong et al. (2017).

Another paper by Hazarika et al. (2017) also conducted experiments to look into performance differences between Hadoop and Spark. They compared Spark and Hadoop on both iterative and normal queries (logistic regression and word count) where they observed a significant difference between Spark and Hadoop when looking at performance outcome. Spark seems to be performing better in both kinds of algorithms within their experiments. One reason that these results might have been observed – according to the authors – is again Spark's in-memory storage feature. However, since memory is of course restricted to a certain amount, performance might stagnate or deteriorate when the number of iterations within a process is increased. Then Spark's memory-related benefits may be mitigated. All in all, the authors conclude that especially for a small number of iterations within an algorithm Spark outperforms Hadoop which, however, might vary depending on tasks and applications.

Parallel and Distributed RDF Systems

As it is the main goal to use a parallel, distributed system to process large amounts of data in general but especially for RDF data, distributed systems that are particularly concerned with RDF were researched in literature and are being elaborated on briefly in the following section.

Schätzle, Przyjaciel-Zablocki, Skilevic, et al. (2016) conducted research in the field of distributed RDF systems and based their own attempt to process RDF data on their findings. They first listed several standalone approaches for distributed systems which include for example *Virtuoso Cluster, 4store, YARS2* and *Clustered TDB* which all extend different centralized systems into distributed ones. Another system mentioned is *TriAD* which uses asynchronous message passing for RDF graphs and summarisation while distributing RDF triples on different nodes within a cluster by horizontally partitioning them. *METIS* partitions input RDF data and similarly creates a summarized graph. However, the authors argue that such centralized partitioning strategies used in the stated examples are often not as scalable for large amounts of data.

Additionally, the authors Schätzle, Przyjaciel-Zablocki, Skilevic, et al. (2016) analysed some federation attempts of distributed systems using RDF stores deployed on multiple nodes. Here, the main idea is that most of the data processing is performed on the specific nodes where the data is distributed to. Then, only if required, the complete dataset is brought back together and merged again, thus combining the nodes. Systems following this approach mentioned are for example *Partout*, *SemStore* or *DREAM*.

The third part of the work by Schätzle, Przyjaciel-Zablocki, Skilevic, et al. (2016) is an analysis of implementations within cloud infrastructures. Cloud systems often use resources that are designed specifically for big data. Hadoop and especially the already described Hadoop Distributed File System is used on those cloud platforms frequently. Systems that make use of the scalability in the cloud are for example *SHARD*, *PigSPARQL*, *Rya*, *H2RDF+* or *Sempala*.

This short summarization regarding distributed RDF data system shows that there are many projects that aim to process large volumes of data and also specifically big RDF data in parallel using decentralized storage systems to try and increase performance. They however often focus primarily on storage systems. Since in this thesis the main focus is on the processing of

big RDF data, however, and not necessarily on storing the data, it was further aimed to look at processing technologies and on how they work with big RDF graphs. Therefore, the processing framework Apache Spark was considered and analysed in even more detail in order to find out whether it is suitable for the tasks included in KG-OLAP query processing.

2.6. Apache Spark

Apache Spark was developed at the UC Berkeley AMPLab in 2009 by Matei Zaharia (Zaharia et al., 2010). The research project was then open sourced in 2010 with increasing interest illustrated by several papers and an increasing community. Spark was then donated to the Apache software foundation which took place in 2013 as stated in Apache Spark History (2023).

Spark is a processing engine that enables a user to perform data operations in a distributed manner. It is – according to Yang et al. (2016) – especially designed for iterative and interactive algorithms but can nevertheless be used for other use cases such as processing of graph data. The authors claim that Spark is often the chosen framework when distributed computing of big data is required in different fields. Since Spark was built upon Hadoop it also includes any advantages that might come with that. Further features of Spark will now be briefly illustrated in the next section.

2.6.1. Spark Features

The first feature and advantage of Spark often mentioned is its speed. IBM (IBM Cloud Education, 2016) for example claim that Spark runs one hundred times faster than Hadoop when it is run in memory. The increase in speed is made possible through the reduction in read and write operations to and from the disk that are necessary for the job execution as already mentioned. The speed advantage hereby depends, however, on the task at hand.

The described reduction of read and write operations is also linked to Spark making use of *RDDs – Resilient Distributed Datasets* – as its way of storing and also processing data. In short, Yang et al. (2016) describe RDDs as objects that can be viewed as a "*read-only collection of data*" that is partitioned in a certain way. When transforming RDDs the transformation's output – also referred to as intermediate result – can be cached by Spark which leads to less write operations to disk.

Another important feature of Spark is that in order to ensure fault tolerance, Spark does not replicate data on multiple nodes on the cluster as a lot of other frameworks might do according to Yang et al. (2016). Instead, Spark simply saves the complete data lineage – the steps that led from one RDD to another – making it possible to recompute RDDs if they get lost in the process or if a task fails.

As already mentioned, Spark provides methods for different types of tasks including for example batch or streaming data and SQL querying but even offers libraries and algorithms for machine learning and graph processing operations (IBM Cloud Education, 2016). This makes Spark a broadly usable framework that can be used in a variety of settings.

Another advantage of Spark is its flexibility. First, Spark code can be written in different programming languages including Java, Scala, Python and R where APIs are provided. Second, Spark may be deployed as a standalone application, but may also be run on Amazon EC2 or even locally as described in Apache Spark FAQ (2016).

With all those characteristics illustrated, Spark has the potential of performing well with the RDF data that must be processed in the KG-OLAP setting. For the prototype described within this thesis especially the advantage of Spark's speed was hereby considered when choosing it. Moreover, the easy-to-use library that was available in Java and the fact that the setup of the system to run the experiments was very easy were arguments in favour of Spark for the tasks involved in KG-OLAP querying and transformations.

2.6.2. Spark Architecture

Since it has now been established that Apache Spark can be considered a promising framework for implementing KG-OLAP query operations, some basic understanding of how Spark works is needed. The structure and certain specific Spark terminology is therefore illustrated briefly in the following section. The explanations are mainly based on Karau & Warren (2017).

Every Spark program in general consists of a hierarchy of five different layers. Within this hierarchy there is the application itself, jobs, stages, tasks and operations. The application is the entry point consisting of one or more jobs. Whenever there is a *SparkContext* that is created within Spark code and run in a program, the application is launched. This SparkContext can be thought of as several Spark configurations that determine different characteristics of the associated driver and executors that are involved in running the program. More details about drivers, executors and Spark scheduling will not be discussed here but can be found in Karau & Warren (2017).

The number of jobs within an application depends on the actions that should be performed during the processing of data. Whenever jobs are triggered by their corresponding actions, Spark returns an intermediate or end result of a process to the driver. Then a subsequent job might be triggered by the next action. The jobs themselves may then be divided into stages. Those stages consist of tasks requiring a number of operations to be executed as described and illustrated by Zhang et al. (2018).

The different levels and Spark structure are shown in Figure 1 and are then explained in more detail in the next sections.

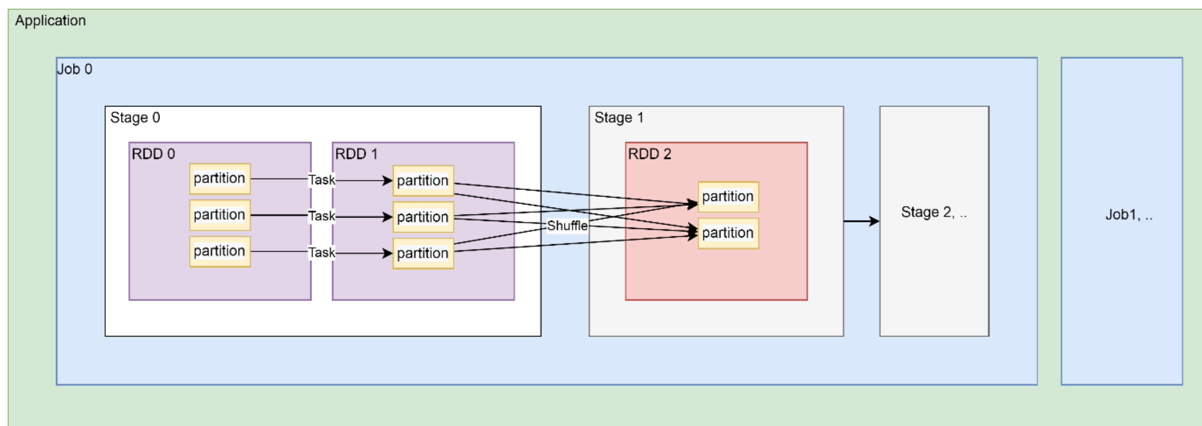


Figure 1: Spark Architecture adapted from (Zhang et al., 2018)

Jobs and Stages

In Spark, every job performed relates to an action. Actions are called by the driver of an application implemented with Spark. Karau & Warren (2017) describe a job as *"something that brings data out of the RDD world of Spark into some other storage system"*. This means a job ends (and another one might start) whenever data is brought back to the driver or persistently stored somewhere. Every Spark job might then consist of one or more stages.

Stages within a job are parts of work performed on data, which can be done without having to communicate with the driver. Therefore, a stage is something that can be executed without having to transfer data from one data partition to another. Whenever data needs be moved and hence some type of communication within the network is required, a new stage is created. The creation of a new subsequent stage for example takes place when operations like shuffling are to be executed. This is why such operations are also referred to as a *shuffle dependency* or a *wide transformation.* All created stages are then processed sequentially and may contain several different narrow tasks or *narrow transformations*. The difference and examples for narrow and wide transformations and their impact on Spark's performance will be discussed later in some more detail.

Tasks

Another component also described by Karau & Warren (2017) is the *task*. Tasks are the smallest units contained in a stage within the Spark execution environment – apart from operations. All tasks that belong to the same stage execute the exact same code at the same time. However, each task does so with a different, independent part of the complete data set that needs to be transformed. Those parts of data are called partitions.

Hereby, one task can only be performed on one executor. Every executor, however, has multiple slots where it can run tasks, thus making it possible to run multiple tasks or all at the same time. The number of slots of an executor is not static and changes depending on the current

task that needs to be executed. Here, the number of partitions of the resulting output RDD matches the number of tasks that can be executed at the same time in one stage – as stated by Karau & Warren (2017).

A cluster might not be able to run all tasks in parallel in each stage, since there may be limitation to the number of cores per executor when configuring the Spark application. It is therefore not possible to run more tasks as there are executor cores available for the Spark application. In order to calculate the number of tasks that may be run at the same time (in parallel) the following simple formula can be used (Karau & Warren, 2017):

*“total number of executor cores = number of cores per executor * the number of executors”*

Whenever there are more partitions and therefore more tasks than number of total executor slots, all additional tasks are processed only after the first ones have been finished and slots (resources) are free for them to be used. Usually, tasks within one stage need to be finished first before the next stage can be started and its tasks can be processed.

This allocation and distribution of tasks is handled by the *TaskScheduler* which might use fair scheduling or a first-in-first-out schedule. For the scheduling and execution of the tasks a *DAG* (Directed Acyclic Graph) – as described by Yang et al. (2016) – of the stages is built which is based on the lineage of the RDDs creating an execution pipeline.

2.6.3. Further Spark Terminology and Concepts

In this section some further Spark specific concepts that were looked at and used when implementing the prototype described within this thesis are briefly explained in order to generate a general understanding of the capabilities of Spark.

RDD

As already mentioned, Apache Spark uses *RDDs – Resilient Distributed Datasets* – as its main way of storing data.

RDDs were first introduced in 2012 by Zaharia et al. describing them as a new abstraction which can be used to efficiently reuse data implemented within the Spark system. Those RDDs are also created in a way to provide capabilities to operate on data in parallel as well as to assure fault tolerance in Spark applications.

As described by the authors Zaharia et al. (2012), usually in systems that use in-memory cluster storage, fault tolerance can only be assured by duplicating data across multiple nodes or by logging every update across all of them. Either way, such methods of providing fault tolerance lead to increasing amounts of data that needs to be sent across the computing cluster and thereby generating storage overhead. By using RDDs instead, Spark on the other hand does not need to store the same data multiple times but it saves the operations and their sequence in which they were performed on the data – the lineage. In doing so, Spark makes sure that whenever parts of the data are lost, the system still has enough knowledge to recreate the state of the data that it was in before the failure. The lineage is used to recompute this

previous state, thus guaranteeing fault tolerance without the need to store more data than necessary.

According to Zaharia et al. (2012) in short, RDDs are therefore an abstract form of read-only data which is partitioned across nodes. They can only be created in two ways: from other RDDs or from any kind of stable data source. Moreover, persisting and partitioning of the data contained in RDDs can be adjusted by the user depending on their storage needs and capabilities. Hereby, for example a key can be defined to partition the data across machines as desired. When Spark uses RDDs they are usually kept in memory per default, but they can also be persisted onto either one disk or all machines available in the cluster.

Transformation vs. Action

According to Karau & Warren (2017) Spark differentiates between transformations and actions as already mentioned. Transformations are performed on RDDs, altering them, and returning RDDs as output. In contrast, actions do not return a new RDD, but rather any different kind of data structure as their output. Actions are either used to collect information and bring back data to the driver for certain processing steps that require the whole dataset or to write to a stable storage medium. Such actions that bring back data to the driver are for example: *collect*, *collectAsMap*, *sample*, *reduce* or *take*. Hereby actions like *count* or *reduce* bring back a certain, predetermined amount of data to the driver. In contrast, for actions like *collect* or *sample* the amount of data that the transformation might result in is not known beforehand. The second category of actions – actions that write to storage – include for example methods like: *saveAsTextFile*, *saveAsObejctFile* and *saveAsSequenceFile*. There are also void functions that do not return any data like *foreach*. Those void methods are nevertheless classed and treated as actions which means that Spark is forced to execute a job. In other words, they more or less force the execution of the Spark program.

In contrast to actions, there are transformations as described by Karau & Warren (2017). Transformations are used to perform certain tasks like sorting, grouping, filtering, or mapping the data. Contrary to actions, transformations do not force Spark programs to be executed. This characteristic of Spark programs not executing immediately is called *lazy evaluation* and will be explained in more detail later.

Wide vs. Narrow Transformations

When looking at transformations on Spark RDDs, there are two different kinds to differentiate as mentioned by Karau & Warren (2017). First, there are transformations with narrow dependencies and then second, transformations with wide dependencies. This differentiation is especially important to consider when thinking about the performance and optimization of a Spark application.

Transformations with narrow dependencies are ones where the child RDD depends directly and finitely on the partitions of its parent RDD. This means that those dependencies can already be evaluated by Spark at the time the application and code is designed. Additionally, narrow transformations are only possible if each parent partition does not have more than one

child. As a result, the child partition depends either on just one parent partition or on a subset of them. This subset however also needs to be unique for this child only. Those characteristics makes it possible for narrow dependent transformations to be performed on a partition or set of partitions without having to know anything about other partitions of the RDD – therefore independently without the need to communicate.

The following illustration in Figure 2 shows how narrow transformations can look like in Spark's dependency graph:

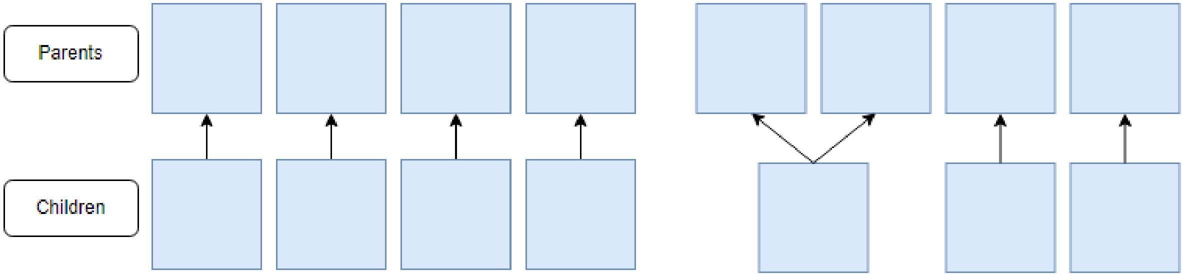


Figure 2: Narrow dependencies adapted from (Karau & Warren, 2017)

As is can be seen from Figure 2, there are basically two possibilities for children to depend on their parents. On the left part of the illustration each child partition only depends on one parent partition which means they can be processed easily at the same time since they do not need any information about the other parents or other children present. On the right side it can be seen that children may also depend on two (or even more) parents, but then no other child depends on the same set of parents. Therefore again, they can be processed independently.

Transformations with wide dependencies, on the other hand, need the data to be split and allocated in a certain way to have all information needed available. Such transformations can therefore not be performed independently on parts of the data. Specific partitioning is for example needed when performing *reduceByKey* or *groupByKey* transformations, when executing joins or when repartitioning the data in general. Most of the time, therefore, wide transformations need to shuffle the data first in order to reach the required partitioning state. Only if this certain partitioning of the data is already ensured before such a wide transformation is executed, it might be possible to eliminate the need for Spark to shuffle data. Otherwise, Spark needs to do such a rather expensive repartitioning step to execute the next steps in the application. Whenever data does need to be shuffled, Spark adds a *ShuffeledDependecy* to the dependency list of the RDD.

In Figure 3 it is shown how a dependency graph might look like with wide transformations in contrast to narrow transformations:

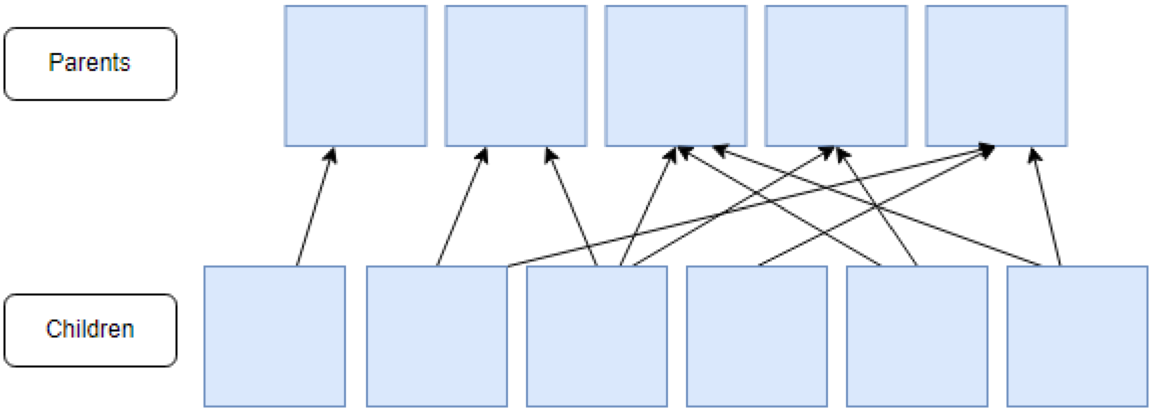


Figure 3: Wide dependencies adapted from (Karau & Warren, 2017)

Figure 3 shows how some children in wide dependencies may depend on just one parent similar to narrow transformations. However, they may also depend on more parents as it can be seen. The actual difference to narrow dependencies however are the child partitions that depend on parents that also other child partitions are dependent on. In essence, those children are sharing parents and therefore, there might be the need of shuffling to ensure that the data is partitioned in a way that children processes can be executed.

All in all, as discussed shuffling is a performance expensive task which should be avoided. Therefore, using narrow transformations instead of wide transformations – if possible – should be preferred when trying to optimize performance of the Spark application according to Karau & Warren (2017).

2.6.4. Memory Structure

An important advantage of Spark regarding performance – as stated by Karau & Warren (2017) – is the way it manages memory. The framework persists data without writing to disk but instead keeps the data on the executors in memory where it then is accessible when needed. The way data is stored can also be configured by the *persist* function. Here one can specify whether RDDs are stored in-memory for example as deserialized Java objects, as serialized Java objects or on disk. This might for example be necessary when RDDs are too large for RAM. Those ways of storage can be chosen depending on the needs of the Spark application and environment.

In general, memory in Spark is separated into three parts mentioned by Zhang et al. (2018) in their work. Those are there to either "*manage user memory, storage memory or execution memory*" (Zhang et al., 2018). Hereby, user memory is dependent on the function that the user specifies. Depending on the way the code is written and what functions are used by the user,

the memory usage changes. Storage memory is the part of memory that Spark uses for caching and broadcasting data which helps in reusing data and sharing variables between workers. Execution memory is the memory that is used whenever there is a shuffling operation necessary for any parts of the data processing pipeline. In Spark, there is no fixed cut between storage memory and execution memory. Therefore, both may use each other's space when they themselves do not have enough memory available for a certain operation. Overall, decisions and best practises regarding memory are difficult to generalise, since there are a lot of different factors that influence how memory is used and it also strongly depends on the specific use case, as stated by the authors Zhang et al. (2018).

2.6.5. Lazy Evaluation

Another key feature of Spark and its RDDs mentioned in Karau & Warren (2017) and Hong et al. (2017) is the lazy evaluation principle. Lazy evaluation means that whenever no wide transformations (as explained in Chapter *2.6.3 Further Spark Terminology and Concepts)* need to be performed within the Spak application, only a pipeline of RDDs is created without executing those tasks immediately. All the pipelined steps are then executed at the same time later whenever an action needs to be performed which forces the actual execution. An action is an operation that results in a data structure that is not necessarily an RDD and where data needs to be sent back to the driver or writes to persistent storage.

The lazy evaluation technique helps to decrease the number of in- and output overhead and thereby improves performance according to Hong et al. (2017). Since hereby only a directed acyclic graph (DAG) is created – or in other words a pipeline to represent the lineage of RDDs – it makes it possible for iterative algorithms to pass a job's result directly to the following one without having to persist data in between operations as described by Zheng et al. (2016).

After the DAG containing the plan of dependencies between the RDDs has been created and an action was obtained by Spark, Spark basically starts at the end of the graph – at the output dataset – and works its way back according to the provided lineage.

The advantage of this lazy evaluation characteristic can also be illustrated with the following example used by the authors Karau & Warren (2017). When a filter function and a map function must be performed on an RDD, usually that would mean each partition of the data needs to be accessed twice – once for the filtering and once for the mapping. Due to Spark's lazy evaluation, however, the executor gets the information that the steps need to be performed at the same time and therefore accesses each partition only once where it executes both the filtering and the mapping step at once.

Another advantage according to Karau & Warren (2017) is the fault tolerance that comes along with lazy evaluation. Since each partition of the data in a Spark application has the information of exactly which dependencies there are between parent and child partitions, any data that might be lost can easily be recomputed using the known lineage. This diminishes the need for logging or duplicating data, thus making Spark faster and perform even better.

The only disadvantage of lazy evaluation becomes apparent when trying to debug an application and find out exactly where Spark generates errors or fails. When there is for example an error already in an input step where data is read, the failing of the job will only be visible for the developer when an action – like for example when counting – is done in a subsequential step once data is sent back to the driver. This means that often the occurrence of an error in logs does not actually represent the moment or point in the code where it actually fails but aways at the same point at which an action is called.

Since literature suggests that lazy evaluation and memory management are important for improving performance of Spark, it was investigated in more detail how different settings that can be chosen when creating a *SparkContext* could have a further positive impact on the KG-OLAP implementation. This was done in order to find out whether there are any specific settings that should be used or set to a certain value. Those settings will be discussed now in the next section.

2.6.6. Performance

Since performance experiments and optimization are part of the research question and objective of this work, it was tried to find out, whether there are any advantages in changing the Spark configurations and settings to ensure best performance when creating the SparkContext used within the application. In general, there are a lot of parameters that one can set when executing a Spark job. In their paper, Nguyen et al. (2017) tried to identify relevant configuration settings and whether or how they might influence performance of Spark. They focused on the subset of settings that is meant to be important for performance tuning and separated them into two categories:

- application settings and
- settings for individual parts of the Spark application

For even more details on performance and how it was assessed it is referred to Nguyen et al. (2017) since only the main findings are presented within this thesis.

Table 1 shows the identified settings, their type, default value and what resource the setting may influence:

Table 1: Spark performance settings by Nguyen et al. (2017)

Setting	Default Value	Type	Resource
spark.driver.cores	1	Application	CPU
spark.driver.memory	1g	Application	Memory
spark.executor.memory	1g	Application	Memory
spark.executor.cores	-	Execution	CPU
spark.task.cpus	1	Scheduling	CPU
spark.default.parallelism	-	Execution	CPU
spark.memory.fraction	0.6	Memory Management	Memory
spark.reducer.maxSizeInFlight	48m	Shuffle	Memory
spark.shuffle.compress	True	Shuffle	Memory
spark.shuffle.spill.compress	True	Shuffle	Memory
spark.speculation	False	Scheduling	-

In Table 1 above, the hyphen (‘-‘) in the default value column indicates that the value for this specific setting is set by Spark itself depending on the environment the application is run in. In the *resource* column, the hyphen suggests that the setting influences more than one resource.

As it can be seen from Table 1, settings like *spark.driver.cores*, *spark.driver.memory* and *spark.executor.memory* are used to influence the performance of the driver and executors. All other settings listed can be adjusted for performance tuning at the task level. In their experiments Nguyen et al. (2017) set the first three settings mentioned for drivers and executors to values that made their experiments finish acceptably fast and left them unchanged for all further experiments. This decision was made since they found that leaving *spark.driver.cores*, *spark.driver.memory* and *spark.executor.memory* to their default values, made their processes very slow otherwise.

After analysing different metrics with different settings, the results by the authors Nguyen et al. (2017) included the following conclusions about performance impact of the identified configuration options:

Spark.executor.cores:

This setting describes the number of cores that each executor process is able to use. Changing this setting's value determines whether an executor can perform multiple tasks in a parallel manner. In general, executing tasks in parallel should decrease the overall time it takes to finish them altogether. In their experiments, the authors found that changing the executor cores from one to four improved performance significantly. However, when further increasing this number, no notable changes were observed. They also discovered that the number of tasks correlates with the runtime, meaning that jobs with a large number of tasks profit more from increasing the number of cores.

Spark.task.cpus:

The number of CPU cores available for executing tasks determines the number of tasks that can be processed in parallel within the system. The number of parallel tasks then in turn leads to a different number of "*waves of tasks"* Nguyen et al., 2017). The less waves, the faster the job finishes. This is why the analysis result of Nguyen et al. (2017) showed again that the number of tasks correlated with the time the program needs for its execution. Shuffling also impacts performance, meaning that the more tasks that require shuffling, the worse the performance. All in all, the authors concluded that increasing the number of the CPUs might only be effective when there are tasks to be completed that generally take a rather long time to be finished in relation to others.

Spark.default.parallelism:

Setting and optimizing the *default parallelism* configuration makes it possible for the user to define how many partitions are created after a shuffling task was performed. The number of partitions in turn defines the number of tasks in all stages that may be performed at the same time. After conducting different experiments with different settings and Spark programs, the authors Nguyen et al. (2017) concluded that setting parallelism to a higher value can be effective when trying to execute steps that include a *ShuffleMap* task which usually takes a rather long time to finish. One example would be the word count algorithm. For other tasks, changing spark default parallelism might not have any benefit when being increased.

Spark.memory.fraction:

Spark memory fraction describes how much memory Spark is allowed to use for storing data and executing processes. According to Nguyen et al. (2017), after testing different settings, they found that the amount of data that is inputted and the time for shuffling are positively correlated with the execution performance. Again, depending on the task, the size of memory fraction has more or less influence on performance. Especially whenever there is only a small shuffle time or not a lot of data to be processed, memory fraction is not affecting performance as much.

Spark.reducer.maxSizeInFlight:

MaxSizeInFlight is Spark's configuration that lets the user change the amount of data that Spark is able to fetch from outputs of mapping operations by each reducer. Again, shuffle time seems to correlate with this setting and execution duration. When setting *maxSizeInFlight* to a rather small number, this might impact a system in a positive way if there is not a lot of memory available and therefore a lot of transactions between the machines are necessary. With systems that have more memory available, changing the value does not improve performance in a notable way according to the authors' obtained results.

Spark.speculation:

Speculation is related to Spark's possibility to use speculative execution which can be allowed or denied by this setting. The value can therefore either be set to *true* or *false*. Per default it is set to *false* according to the specification by Yang et al. (2016). What Spark does when speculation is set to true, is restart any tasks that are considered slow and might therefore slow down the processing. Checking for those bottlenecks is done periodically by examining whether a certain task might take longer than a particular threshold.

Even though this might sound helpful in general, in certain scenarios setting the speculation to *true* might also impact performance negatively as described by Nguyen et al. (2017). When the setting is set to *true*, it might occur that the number of tasks that need to be performed is increased which in turn means that the run time of the Spark application might increase as well. Therefore, it again depends a lot on the application and general conditions.

Spark.shuffle.compress and Spark.shuffle.spill.compress:

Shuffle compress and *shuffle spill compress* can both be used to tell Spark whether to compress map output data and whether to spill data after shuffle operations. These settings' impact correlates with the time the executors take to de-serialize data in the processing and shuffle time. Nguyen et al. (2017) suggest that the configuration should be set after having weighted out CPU and network/disk performance. When compression takes place, data size is decreased meaning less network traffic. However, this might then cost CPU resources to compress the data in the first place. Compression might not help (and even increase runtime) when only small amounts of data need to be written and read by shuffle operations.

As with all the setting parameters, it is again very much task- and application-depended and factors outside of Spark need to be considered.

All in all, as described, when starting a Spark application, the user can choose different settings and parameter values – some of which were already discussed relating to Spark performance in previous sections but also a lot more that are out of the scope of this thesis. It can therefore be decided on the number of cores per executor, the memory size per executor and of course the number of executors itself and so on. A user can also specify how many cores every task gets assigned. As mentioned by Zhang et al. (2018), memory size per task is determined by this setting. If tasks do not receive enough resources (tasks of the same executor share the executor's memory) then an out-of-memory error may occur.

Of course, there are also a lot of other performance tuning options on different levels and layers or components (CPU, network bandwidth, memory, storage of data) of a Spark application. However, they were not further investigated as part of this thesis but are briefly stated, nevertheless. According to the Apache Spark documentation (Tuning Spark, 2022) the main tuning capabilities are provided through serialization and memory tuning. For serialization usually Java serialization is used per default. Serialization can, however, also be changed to *Kyro* serialization which is said to be more compact than Java serialization. Here it needs to be noted that not all serializable Java types are supported out of the box. In memory tuning, one can tune aspects like garbage collection, access of data and how much memory is used in general. There are also even more options to try and improve performance by for example changing parallelism levels or using broadcasted variables. For more details it is referred to (Tuning Spark, 2022).

Again, to summarize, those configurations and considerations need to be optimized depending on the problem, application, scenario, and a lot of other external factors. Therefore, it is not easy to say which settings are optimal in general. Thus, before conducting the actual experiments on this thesis implemented query operations, some of the settings mentioned were trialled to try and find an optimum. However, no changes of the settings that were tried out improved performance drastically. Therefore, for optimizing performance the implementation and Spark code itself was tried to be optimized. The Spark settings when creating the used SparkContext for the experiments were set to fixed values and kept throughout all further tests. Performance tuning with the stated settings may be part of future work.

2.6.7. Apache Spark GraphX

Since in this thesis not only Spark itself plays an important role but also Spark GraphX was used, the framework will be explained in theory in the next sections. The main reason that Spark GraphX was considered was that it is intended to be optimized for working with graph data which is the basis of the KG-OLAP concept as RDF data is used to represent knowledge.

Apache Spark GraphX was released as a component of the Apache Spark open-source project in the 0.9.0 version. It was first described in 2014 by Gonzalez et al. in their paper about "*Graph Processing in a Distributed Dataflow Framework*". The framework they proposed is an *"embedded graph processing framework"* (Gonzalez et al., 2014) which was built on the basis of Apache Spark. GraphX provides a graph abstraction which makes typical graph operations possible. However, in GraphX, also a view of the data as a collection is retained which allows other intuitive and natural data manipulation in addition to iterative graph algorithms and graph-parallel processing. To represent graph data, GraphX uses a property graph model which was already briefly described in Chapter *2.2: (Contextualized) Knowledge Graphs*.

The basic property graph in GraphX can be seen in the following example illustration in Figure 4 that models different nodes and their relationships:

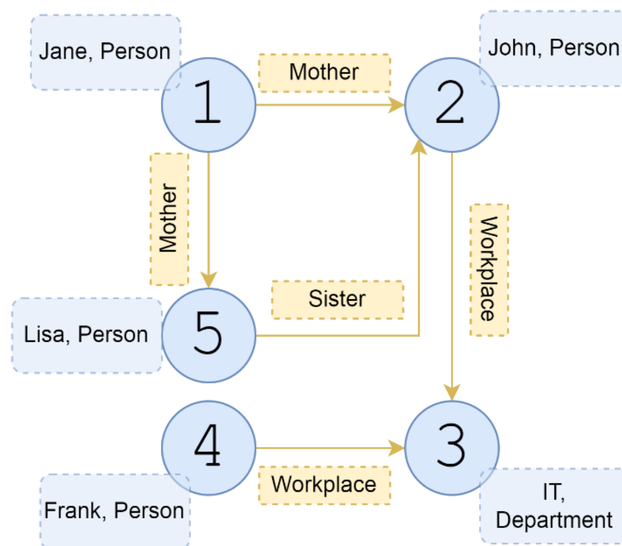


Figure 4: Property graph example

Figure 4 shows a property graph where the nodes (vertices) represent people and departments. The edges between the nodes are either there to represent personal relationships such as Jane being the mother of John or Lisa being the sister of John or other types of relationships like Frank having the IT-department as his workplace. It can be seen from the graph that the vertices themselves are represented and identified by their ID (the numbers within the circles) and can have arbitrary labels or values (e.g.: "Jane", "Department") that contain further information about the node. The edges are directed relationships between the nodes and can also have different values and labels associated with them. Here those are for example *sister* representing that Lisa is the sister of John. Since edges are directed, the relationship only is valid

in the specific direction indicated by the arrow, meaning that Lisa is the sister of John, but John is not the sister of Lisa.

This model can also be described as a combination of two collections: first, the vertex collection, second the edge collection. The vertex collection consists of the vertices (their properties) as well as their identifier. In GraphX the identifiers need to be 64-bit Integer values that are unique within the graph (Gonzalez et al., 2014). The edge collection, on the other hand, consists of edge properties as well as the two vertices that are connected through the edge. The two vertices are thereby only referenced via their ID in the edge collection. The vertices' attributes are not stored with the edges directly.

The following tables in Figure 5 and Figure 6 show how the collections could be thought of representing the graph from Figure 4:

ID	Property
1	Jane, Person
2	John, Person
3	IT, Department
4	Frank, Person
5	Lisa, Person

Figure 5: Example vertex table

SrcID	DstID	Property
1	2	Mother
1	5	Mother
5	2	Sister
2	3	Workplace
4	3	Workplace

Figure 6: Example edge table

The vertex table here in Figure 5 does contain all the properties in one column. Those could also be represented as two separate columns each containing only one of the Strings (e.g.: "Jane" and "Person"). Both tables could also contain more than just one or two properties.

In GraphX – according to the authors (Gonzalez et al., 2014) – a graph can be constructed by simply binding a vertex and edge property where the framework takes care of making sure that only unique vertices and edges are contained in the graph and there are no edges that reference a missing vertex. In the constructed graph one can access the data as edge triplets (which are edges that also contain the vertex attributes not only their IDs), as just edges or as just vertices. In any case, the data is then available as a collection in the form of an RDD that can be operated on.

As already mentioned, in GraphX (and Spark as its base) graphs are a distributed system. The collections of vertices and edges are partitioned. Hereby, the vertex collection is automatically hash-partitioned by the vertex-IDs where the indices are stored in a hash index to make it possible to join all the partitioned collections at any time as described in (Gonzalez et al., 2014).

The collection holding the edge data is partitioned horizontally in a way that can be defined by the user. It is also possible to partition the edges with a vertex-cut which is especially useful in networks and web graphs. Another possibility is the distribution based on the partitioning of the input collection, so for example the original placement of the data in de HDFS. Another

possible approach would be 2D hash partitioning or other functions that can be selected as described in the work of Gonzalez et al. (2014).

All in all, according to the authors Gonzalez et al. (2014) – who also compared GraphX to other graph systems – the way GraphX allows to use the stored data makes it comparably efficient to dedicated graph systems that only offer a “graph view” of the data. They also state that with the way GraphX is designed, it facilitates slicing, transforming and working with big graphs in general. This is especially important for the objective of this thesis which is why GraphX was used in the later described implementation.

3. Related Work

In the next sections, already existing concepts and implementations that are concerned with either Spark, GraphX, knowledge graphs, RDF or OLAP in any combination are described as they are found in literature. It must be noted that the following list of implemented work and concepts is not complete and that only selected ones are described as there are many different variations of related implementations and contents. Nevertheless, it shows in how far the topics covered in this thesis are relevant and have been and still are concerns in current research.

The following list shows the combinations of technologies and concepts that were looked at and are then described:

- Graphs and Hadoop
- RDF & SPARQL
- RDF & OLAP
- RDF and Spark/GraphX
- OLAP & Spark/GraphX

3.1. Graphs and Hadoop

One of the widely known frameworks using Hadoop is *Gradoop*. Gradoop was introduced in the work by Junghanns et al. (2015) and is an open-source analysis system and framework for graph management. It is based on the Hadoop environment and uses an Extended Property Graph Data Model. This graph model extends the usual property graph model, does not enforce any schemas and all graph elements can have different kinds of attributes. There are vertices, edges and logical graphs with various types and attributes that can be defined. It is noteworthy that the framework allows for more than one graph to be processed and queried together. *HBase* – which is built upon *BigTable* by Google and runs on HDFS – is used to store the graph data in a distributed form. Data processing is implemented using operators in the authors' developed domain specific language which is called *GrALa*. Gradoop was evaluated on social network and business intelligence data. The developers' proof of concept uses *Apache Flink* as the distributed execution engine.

Another framework built upon Hadoop was described by Farhan Husain et al. (2009) where they used HDFS as storage for RDF data. The data is first converted into N-triples using *Jena* (Apache Jena, 2022). Then, those triples are split into files that are put into HDFS. Queries on the data can be written by users in SPARQL, however, they are then translated into MapReduce jobs that are passed to Hadoop when being answered. The authors conclude that this approach was, in general, scalable for large RDF data sets according to their evaluation.

There are also multiple other implementations using Hadoop as their base like *HaLoop*, *Giraph* (Ammar & Ozsu, 2018) or *Hama* which often perform better than Hadoop when looking at execution time according to Elser & Montresor (2013).

3.2. Using Spark for RDF Processing

When looking at current literature on frameworks and implementations regarding querying RDF data with Apache Spark, most approaches that can be found are based on the query language SPARQL since it is the recommended standard for querying RDF data (W3C, 2013) as previously explained in Chapter *2.4: RDF and SPARQL*. Often such SPARQL queries are simply translated into other languages that are then used for the actual query answering within the frameworks and implementations according to Ali et al. (2020). One of such implementations is called *Sparklify* introduced by Sejdiu (2019) where SPARQL queries are translated into code that can then be executed by Spark.

There are also a number of experiments and different settings tried out for SPARQL queries on Apache Spark and GraphX trying to test whether distributed SPARQL query answering would lead to performance enhancements, for example by using the Spark SQL query engine as it was done by Ragab et al. (2020).

In the next section some selected implementations that use SPARQL or Spark SQL on RDF data are discussed in a bit more detail. It, however, must be noted that there are possibly more approaches and implementations present in current literature than there are included in this thesis.

HAQWA

One implementation that uses Spark for RDF data is *HAQWA* (Hash-based and Query Workload Aware) by Curé et al. (2015) which was an attempt of a system that uses Spark for realising parallelism in RDF data processing. Performance enhancements are achieved through two aspects: for partitioning the data a hah-based approach is used where the subject of the RDF-triples acts as the key for the hashing (*hash-based*). Another design decision was their allocation approach. Based on the frequency of certain queries on the graph data (*query-aware*), the data is distributed accordingly and sometimes replicated depending on computed costs for certain operations. Furthermore, a data encoding system is used to reduce memory usage by storing not Strings but Integers instead. Other than that, more optimizations that reduce the need for materialization steps as well as the amount of query reformulation necessary are implemented. For then querying the data they rely on SPARQL.

SPARQLGX

Another system developed by Graux et al. (2016) uses Spark and SPARQL queries on RDF data as well. *SPARQLGX* is a distributed RDF triple store where SPARQL queries are transformed into Spark (Scala) code that uses different evaluation strategies depending on the selected storage method and statistical information about the data. The approach is said to scale to large amounts of data using the distributed and parallel nature of Spark, as well as through choosing the right approach depending on how the data is stored and on how many subjects, objects and distinct predicates there are present.

S2RDF

S2RDF was developed on top of Spark by Schätzle, Przyjaciel-Zablocki, Skilevic, et al. (2016), using its relational interface. Here again queries can be written in SPARQL code. The data is partitioned using *ExtVP*. *ExtVP* is an approach extending vertical partitioning based on semi-joins to increase performance. They extend the following idea: Instead of for example storing the RDF triples in the form of *<subject> <predicate> <object>* the data is stored in a two-column data structure for each of the predicates that exists. For example, for the predicate *p* there is a table with all subject (*s)* and object (*o)* combinations that are connected via the specified predicate (*p*). The authors claim that this approach can make processing more efficient for large datasets, however, depending on the data structure it could also lead to problems since tables can be very different from each other in size depending on the number of triples that contain the same predicate. This means that for relatively large tables there could still be performance issues. Therefore, to mitigate such effects, the developers use an approach that precomputes possible joins between the tables.

SparkRDF

A hybrid approach was constructed by Chen et al. (2015) called *SparkRDF*. The system is an RDF data processing concept that uses distributed memory based on the Spark framework. *SparkRDF,* however, does not use the *<subject> <predicate> <object>* representation of the RDF data to process it. Instead, they use *a multi-layer Elastic Subgraph* which is based on five different indices over class and relation subgraphs and their possible combinations (C, R, CR, RC, CRC; where c = class and r = resource).

The input data is first converted to N-triples which are then split into the elastic discreted subgraphs. Thereby, for all triples of the RDF graph that do not contain the *rdf:type* predicate, their subjects and objects are extracted into the corresponding index file where the predicate is the file name, therefore grouping triples with the same relationship together. The same is done for the triples with the *rdf:type* predicate as well. Those are distributed into smaller class files with the object as the file name, thereby reducing storage needed since the file only contains the corresponding relevant subjects of a certain type or class.

In their system indices are stored in HDFS. Depending on the index information that is needed for a user query, the appropriate indices and files are then loaded and joined. The joining results are returned after executing the query plan. The query is processed by a *RDSG*-based (*Resilient Discreted SubGraph*) iterative method, meaning that subgraphs are modelled as a resilient discrete semantic subgraph which is an abstraction that uses distributed memory as already mentioned.

RDFSpark

Another implementation called *RDFSpark* by Banane & Belangour (2019) queries RDF data by using Spark instead of MapReduce to achieve scalability and performance wins. Thereby SPARQL queries are parsed, compiled, and optimized by applying different steps explained in more detail in their paper. Then, the queries are translated into a Spark program. In doing so, a layer between the original SPARQL query and Hadoop's MapReduce is created since there are Spark Jobs executing the query in between. The authors found that their approach performed better than their tested other implementations with *Jena HBase*, *H2RDF* or *CumulusRDF*.

Spar(k)ql

Similarly, Naacke et al. (2017) use SPARQL in combination with Spark to analyse RDF data. In their work *"Spar(k)ql – SPARQL Graph Pattern Processing with Apache Spark"* GraphX was not considered for their evaluation since its processing model is not applicable for their "*set-oriented pattern matching*" according to the authors. In their study they instead consider four approaches: *SPARQL SQL*, *SPARQL RDD, SPARQL DF* and *SPARQL Hybrid*.

SPARQL SQL rewrites a SPARQL query to SQL code and then lets the Spark SQL engine evaluated the query whereby the execution plan is performed by the Catalysts optimizer. SPARQL RDD uses Spark RDDs in combination with filter and join methods to evaluate SPARQL queries. SPARQL DF (DataFrame) uses Sparks *DataFrame* abstraction which can be considered a relational system. SPARQL Hybrid tries to combine the advantages of the three prior approaches.

GraphFrames

In contrast to approaches discussed as of now which focus mainly on Spark SQL, Spark's RDDs or GraphX, a paper by Bahrami et al. (2017) explored Spark's *GraphFrame* abstraction instead. Their decision was based on the fact that GraphX had already been used a lot more in literature as it was developed before the *GraphFrames* API. They, however, nevertheless also present an initial study of the system processing SPARQL queries. The authors not only implemented the query processing itself but also a search space pruning and query clause ranking to further improve performance. When creating the graph, two CSV files (one for the vertices and one for the edges) are created from the source RDF data. Data is again queried by subgraph matching similar to the discussed approaches before. Due to the reduction of triples through pruning as well as the efficient query clause ordering approach, the authors reduced the search time for query matching, thus making their method applicable for big datasets according to them.

Since now a number of concepts that use Spark and RDDs for representing and working with graph data were discussed, more specific use of GraphX in concepts, frameworks and implementations that can be found in literature were analysed. Some of these approaches and concepts found are now presented in the next sections.

S2X

In their paper Schätzle, Przyjaciel-Zablocki, Berberich, et al. (2016) introduce *S2X* (*SPARQL on Spark with GraphX*) which is said to "*combine the graph-parallel abstraction of GraphX to implement the graph pattern matching part of SPARQL*" and the "*data-parallel computation of Spark to build the results of the SPARQL operators*" (Schätzle, Przyjaciel-Zablocki, Berberich, et al., 2016). Essentially, the authors want to use SPARQL to query RDF but within a framework for parallel graph processing.

In their work, the authors define a mapping of RDF data to the representation that GraphX uses for vertices and edges. Therefore, subjects and objects contained in RDF structures are mapped to vertices in GraphX. RDF predicates then are represented as edges in between the vertices in the GraphX graph. The RDF IRIs may not directly be used as identifiers of the vertices in GraphX, since it is required that those are 64-bit Integer values as already described in Chapter *2.6.7: Apache Spark GraphX*. This is why, in S2X, IRIs are used as labels of the vertices rather than identifiers. The IDs are generated by the *zipWithUniqueID* method of the Spark library.

In *S2X*, it is taken advantage of the fact that GraphX is built on top of Spark and does not come as a standalone system. Basic Graph Pattern (BGP) matching is realized using GraphX in a graph-parallel way. Further SPARQL operators and modifies that may be part of a SPARQL query (e.g.: optional, limit, offset,.) are used in a more data-parallel way by using the basic API of Spark itself to implement them. This approach makes it possible to combine both ways of processing in a simple way.

There are also some shortcomings of the proposed system – as described in the paper. One of them is for example that above a certain threshold of data size (here stated in the total number of vertices), the runtime grows disproportionately depending on the query that is performed. Some queries also seem to be a burden on Java garbage collector which again increases runtime. Another problem is that – at least in the authors' experiments – SPARQL queries most of the time only use a small subgraph pattern for matching. Those small patterns do not seem to be ideal for the way GraphX is optimized. Usually, GraphX works with algorithms that take the whole graph into account (e.g.: *PageRank*), which is why it may lead to unbalanced workloads when subgraph pattern matching is performed on a small portion of the graph. The authors therefore propose that partitioning the task might be a possible solution to mitigate the stated weaknesses.

GraphX SPARQL Subgraph Matching

Another approach that also tries to use subgraph matching with SPARQL as well as the GraphX libraries in order to process RDF data was described in a paper by Kassaie (2017). Kassaie also claims that GraphX is one hundred times faster than Hadoop MapReduce. Moreover, GraphX was chosen since it combines data views as collections as well as the graph view. In their approach, the graph view is used for all queries that require pattern matching. The collection view then helps to combine the results of the pattern matching algorithm.

In order to do so, in their work, first data is loaded into RDDs using Spark. Then, subgraph matching is performed with the help of GraphX methods before then results are merged again using Spark RDDs. For the pattern matching, different operations available in the GraphX library like *sendMsg*, *mergeMsg*, *aggregateMessage* and so on are used.

The author Kassaie (2017) concludes that it is complicated to tune Spark because of the high number of parameters that one can adjust when setting up the application. Furthermore, it needs to be chosen appropriately whether and what RDDs to persist avoiding out of memory errors as well as simultaneously keeping performance acceptable. Moreover, decisions on types of variables to be used (broadcast variables or accumulators) have to be made depending on the task at hand.

3.3. OLAP over Graphs and RDF Data

As in this thesis the concept of KG-OLAP cubes is used and worked with, it is necessary to analyse whether there are already approaches, frameworks or implementations that combine OLAP cubes with knowledge graphs or perform typical OLAP operations on RDF data in some way.

Literature shows that some work has been done on the combination of OLAP operations with graphs and also RDF analytics. Moreover, there are also approaches using Spark (GraphX) for OLAP operations. Some of those approaches will now be described briefly in the next sections.

OLAP on Graphs

The *graph OLAP* framework by Chen et al. (2008) aims to make it possible to use multi-dimensional and multi-level OLAP operations on graph data and provide capabilities to analyse the graphs. Thereby, the framework is divided into two parts: *informational OLAP* and *topological OLAP*. However, it is then focused on informational OLAP. The authors' work then also defines dimensions and measures specific for graph OLAP. Regarding the OLAP operations themselves, the system aims to materialize the graph either fully or partially by creating aggregated graphs at different levels. They thereby define roll-up, drill-down and slice-and-dice operations.

Likewise, G'omez et al. (2020) constructed a multidimensional *graphoid*-model on graphs where OLAP-like operations can be performed. The difference here is that they construct their graph as a hypergraph meaning that there might be n-ary or even duplicated edges between nodes. The authors claim that this is more appropriate for OLAP-like settings.

Another concept in addition to graph OLAP is called *Graph Cube*. The *Graph Cube model* was first introduced in a paper by Zhao et al. (2011) as a *"new data warehousing model"* that is designed to aid in OLAP query-processing in an efficient way. It extends the basic data cube by including aggregation and structure summarization of the data. Usually in OLAP systems, measures that are used to describe the data are expressed as numeric values that can be aggregated in a quite straightforward way. In *Graph Cube*, the measures, however, are represented as aggregated networks (aggregated from the base data graph) which are basically the

graph itself in an aggregated form. To implement such a graph cube, the authors mention three ways of doing so: fully materializing the whole graph cube (every aggregation possible is calculated and stored), no materialisation (every cuboid is computed on demand) or partial materialization (a selection of aggregations is precomputed). The right choice is a trade-off between query performance and storage needs according to the authors.

An actual implementation of a framework that was developed to analyse graph cubes using the multidimensional approach is called *TopoGraph*. The work done by Ghrab et al. (2021) defines three different types of graph cubes: *property graph cubes, topological graph cubes* and *graph-structured cubes*. The property graph cube is a combination of multiple aggregated graph cuboids. The authors especially mention the contained *base graph cuboid* where the graph is at its base level. Another special cuboid is the *apex cuboid* where the data is aggregated at the highest level. Topological graph cubes are graph cubes that contain topological information about the graphs which are represented in numbers. Graph-structured cubes are graph cubes that represent the concepts of the dimension with graphs containing measures and/or dimensions.

The graph cubes are used to represent the input graph – a multi-graph that is directed and contains attributes, nodes and relationships between them. Dimensions in the graph are possible perspectives of the content and topology. Those are structured in a certain hierarchy. The graph information can be aggregated by its dimensions and then the corresponding values can be calculated as it is known for OLAP operations.

In their paper the authors Ghrab et al. (2021) highlight that in order to increase performance of their approach, a distributed graph engine would, however, have to be used.

OLAP on RDF

More specific work in which OLAP cubes are described as RDF graphs were for instance done by Colazzo et al. (2014), Azirani et al. (2015) and Ibragimov et al. (2015).

Colazzo et al. (2014) introduce a formal framework for analytical (or warehouse) processing of RDF data. They define a multidimensional analytics model which is optimized to work with heterogenous RDF data in the form of graphs. Both the base data and the warehouse are RDF graphs. Their *Analytical Schemas (AnS*) are represented as graphs which contain facts that can be analysed by using dimensions and measures. In their definitions of the OLAP operations on RDF data they use a rewrite-approach on extended *AnQs* (*Analytical Queries*) since in their work they define cubes as being *AnQs* themselves. For their OLAP implementation they focused on two basic operations: slice-and-dice and drill-in and drill-out. Their work resulted in the conclusion that there is a feasible way of constructing an RDF warehouse and also that data can be queried. However, future work will focus on finding methods to deploy *AnS* and *AnQ* more efficiently by for example using parallel MapReduce.

Azirani et al. (2015) based their work on the model by Colazzo et al. (2014) and tried to optimize it by using materialized results of queries which are represented as a cube to analyse

another cube. They basically created algorithms for cube operations that use intermediate or final results of a previous query in order to answer another, new query.

Another paper that proposes a multidimensional RDF schema, which represents an OLAP cube, was written by Ibragimov et al. (2015). The schema is described in the QB4OLAP vocabulary where dimensions, levels, roll-up relations and so on can be expressed. The second part of the vocabulary that they use is specifically designed for RDF called *VoID*. *VoID* is used to aid in describing RDF metadata and how RDF data may be accessed. With the help of this schema, data can be queried, aggregated and a cube can be built.

OLAP with Spark/GraphX

Since those previous examples mentioned concerning OLAP operations on graphs do not necessarily make use of Spark, it was further analysed whether there are already implementations that include Spark and/or GraphX in the combination with OLAP cubes explicitly.

One of such a concept is *Grad* introduced by Ghrab et al. (2015) which was developed as a framework that aids in building OLAP cubes from graph data using property graphs – or rather an extension of them. The framework is designed to also support heterogenous graphs that have various kinds of edges and vertices with different attributes. This is supposed to be more in line with real-life scenarios. Again, the graph (the cells of the cube) contains numeric measures, which is different from KG-OLAP. There are content-based measures which are like the usual measures in OLAP systems. Graph-specific measures capture graph topology like *PageRank*, paths between pairs of nodes, centrality of nodes or number of cycles in the graph. Similarly, to already discussed approaches the graph cube is again defined as a collection of aggregated graphs where the base graph is transformed into all possible levels of the aggregations with adjusted measures.

Grad then is actually a graph database model which extends the property graph model. A prototype was developed with Neo4J acting as a graph database, HDFS for distributed processing and the GraphX library for graph-specific calculation of measures such as the mentioned PageRank. Results are again stored in a Neo4j instance.

Pagrol – as developed and described by Wang et al. (2014) – is a system enabling OLAP processing over attributed graphs with the help of a conceptual *Hyper Graph Cube* model. Queries and standard OLAP operations like slice-and-dice and roll-up are supported based on a MapReduce algorithm. The hyper graph model uses attributed graphs which means that attributes are added to vertices and edges. All vertices and edges are associated with respective dimensions (edge dimensions or vertex dimensions) which then can be used for aggregation. Moreover, materialisation, joining, batch processing and cost-based execution plan optimizations are proposed by the authors in order to increase performance.

There is also work on *distributed graph-based OLAP cubes* by Denis et al. (2013) which focuses on improving performance for cube aggregations and computations on the graph by using decentralized graph cubes. Through using a distributed system when for example performing an aggregation function, not only one machine is responsible for doing all the work,

but it is split to multiple nodes. Those node then perform the operation in parallel at the same time. This approach aims at reducing overall compute time. Their model also is similar to a MapReduce model since both aim at distributed computation.

For their implementation Denis et al. (2013) both considered Spark and Hadoop, whereby the authors decided on Spark for performance reasons. The operations of the cube queries are then executed by using HDFS and Spark together. It must be noted that the authors only conducted their experiments on homogenous graphs, heterogenous graphs would be part of future work. Moreover, only one materialization strategy was evaluated. For an overall conclusion it therefore would also need to be tested whether there are more beneficial ones.

Two other algorithms using Spark for OLAP cubes were proposed by Kang et al. (2020): *GraphNaiive* and *GraphTDC*. Both work with multidimensional graph tables, joint tables between vertices and edges as well as knowledge about the dimension values for the data.

GraphNaiive computes cuboids for all dimensions of the graph in a sequential way from the graph data which is distributed between nodes. Aggregations are calculated by the respective function to create the graph cube. Because of its sequential nature, the more data needs to be processed, the higher the number of computations.

To solve the problem of the computation increase, *GraphTDC* computes the same cuboids but only after having created an execution plan. The highest-order graph cuboid is computed first, afterwards the others are created according to the execution plan. As computation of descendant graphs use the aggregated values of their ancestor – using a top-down approach – faster and more efficient processing can be achieved.

The graph multidimensional tables are implemented using Spark RDDs. As a comparison, also the use of *DataFrames* was trialled by the authors Kang et al. (2020) which proved to perform worse than both the algorithms *GraphNaiive* and *GraphTDC.* Those findings show that RDDs might be more efficient than using *DataFrames*, at least in their specific context.

3.4. Summary of Current Literature

When looking at the available literature it is apparent that there is definitely room for further research especially concerning the combination of representing RDF data within OLAP cubes and processing the data with Spark and GraphX. However, there are already implementations and frameworks that consider some of the similar aspects that are subject of this thesis.

In general, when researching big data or parallel graph processing in literature, Hadoop and its components are often mentioned. An analysis of distributed graph systems by Ammar & Ozsu (2018) was done comparing *Hadoop*, *HaLoop*, *Vertica*, *Giraph*, *GraphLab*, *Blogel*, *Flink*, *Gelly* and *GraphX*. The authors concluded that all of them (apart from Vertica) read and write datasets from and to a distributed file system. *Hadoop*, *HaLoop* and *Giraph* make use of Hadoop's MapReduce for processing the data. *Flink* uses existing libraries from Java and Scala whereas *Blogel* as well as *GraphLab* are implemented using different libraries in C++. GraphX runs on top of Spark.

There is also work combining the concepts of graphs and OLAP, OLAP on RDF data and even combinations of Spark or GraphX and OLAP. Most of the approaches, however, differ on how either the RDF data, Spark or cubes are represented and used as they are in this thesis' described prototype. This makes the implementation described in this thesis relevant even though especially Spark is already widely used for big data (graph) processing in literature within concepts and implementations.

4. Data Model

This section describes in detail the implemented data model that is used within the prototype described in this thesis. First, the data structure of the KG-OLAP source data set that is used, and domain-specific examples used throughout this thesis are described. With the data structure in mind, considerations about why GraphX was chosen as a framework for the KG-OLAP implementation are argued. Then, the general design decisions concerning data representation in Spark and GraphX are illustrated.

4.1. Running Example

In this part of the thesis the data that was used while developing the prototype and also in the SPARQL-based prototype implemented by Schuetz et al. (2021) is briefly outlined. All further examples that are used in this thesis as well as the performance experiments are based upon this specific domain and use case. The implementation can, however, also be adapted to be used on datasets of different fields as long as the general KG-OLAP structure is adhered to.

In general, the data that is used for the experiments conducted within this thesis concerns the field of air traffic management (ATM). Within this data, messages, which are sent by air traffic control to inform affected people and departments about any occurrences or changes in the air traffic infrastructure are the main concern. The data needs to be distributed to those relevant people and departments to ensure safety and quality in the planning process but also for daily operations in air traffic management. Such messages may contain information about any events taking place as for example closures of runways due to contamination or any other problems that may affect airports, airplanes, pilots and their daily work. By merging, aggregating and abstracting this data and the contained knowledge, one can obtain different views on the data. There could be use cases where a more overview-like summary of the information is needed when for example informing pilots about the most important events more quickly. Other analyses might on the other hand need the data in a more detailed state. Depending on the information and granularity that is required to be distributed and the receiver of the message, the data needs to be on the appropriate level of aggregation.

The data contained in the hereby used datasets can for example include information about in how far a runway is approachable for airplanes depending on contamination (e.g.: snow) or other factors that might have an impact on being able to start and land an airplane. For further details about ATM data, it is referred to the paper by Schuetz et al. (2021).

As the data in KG-OLAP represents a multidimensional model, there are several dimensions with defined hierarchies for which the information contained in the RDF statements is relevant. Those hierarchies that are used to construct the KG-OLAP cube are in this case: the location, time, the importance and also the aircraft that the information is applicable for. All those four dimensions have different hierarchy levels:

- **Location**: segment, region
- **Time**: day, month, year
- **Importance**: importance, package
- **Aircraft**: model, type

Those hierarchies imply that for example the data on the most granular level of all four dimensions is relevant for one specific aircraft, on one specific day, at one specific location and with a certain importance. Then, there is also knowledge that is for example relevant at every location, at all times for all aircraft and concerns all importance levels. This would then be classified as the most general knowledge or context. The dimension value's (for example a specific date like the 5th of June 2022) dependence on each other is structured in a linear manner by roll-up-relationships represented as RDF-triples. Meaning that the 5th of June rolls up to the month of June. The month of June rolls up to the year 2022 and the year 2022 rolls up to the "All"-level which contains – as its name suggests – information relevant for all dimensions and all their levels. The same applies to all other dimensions present. For more details it is again referred to Schuetz et al. (2021).

4.2. Technology Decisions

In general, after researching big data processing frameworks and deciding on Spark to be used as a big data framework, one is presented with the decision to choose either GraphX or GraphFrames with them both being graph-specific abstractions built on top of Spark. Therefore, further literature was considered, to find out which of the two frameworks would be more appropriate for the tasks involved in KG-OLAP data processing and representation.

In their paper, Agathangelos et al. (2018) studied current usage of Apache Spark for graph processing showing the different aspects to consider as well as ideas and disadvantages of different concepts. As already described, Spark either uses abstractions called RDDs (Resilient Distributed Dataset) or DataFrames to represent data. Based on those two technologies, there are then the two mentioned graph-specific abstractions: GraphX and GraphFrames whereby GraphX builds upon RDDs and GraphFrames relies on DataFrames.

The authors therefore argue that the decision one must make when dealing with RDF data and wanting to use Spark, is to decide two things. First, one needs to choose the type of data model and second the type of Apache Spark abstraction and abstraction level one wants to use.

When using RDF, the possible data models are either *the triple model* or *the graph model* according to Agathangelos et al. (2018). In the triple model, RDF data is used in its natural structure – as a triple with subject, predicate and object. When deciding on the graph model,

RDF data, however, is stored and processed "*as a directed, labelled graph*" (Agathangelos et al., 2018). Therefore, an RDF triple is represented as an edge – which can be seen as the predicate – connecting two nodes: subject and object. The predicate itself is then represented as the label of the edge between the two nodes. This latter representation is mainly used in the graph API of Spark.

The second decision that needs to be made as proposed by Agathangelos et al. (2018) is the kind and level of abstraction and libraries that one wants to use. Here a choice must be made between "*RDDs, DataFrames, Spark SQL, GraphX and GraphFrames*" (Agathangelos et al., 2018). RDDs are considered to be a more low-level representation of data which enables the user to work in a very flexible way. They also for example provide more adjustable ways of partitioning the data. DataFrames on the other hand organize data in columns, similar to a table, which makes it easy to structure data and work with it since it is available in a familiar form.

As a more high-level abstraction, Spark SQL provides functionality to query structured data in a SQL-similar way. GraphX on the other hand is described as combining graph-parallel and data-parallel processing which increases performance and makes it even more flexible to work with. Moreover, widely used graph processing algorithms like PageRank or triangle counting can easily be implemented. GraphFrames puts another abstraction layer on top of the DataFrames abstraction which supports querying the data. Therefore, the decision here is two-folded. First, it needs to be decided whether to work with solely the more low-level structures like RDDs and DataFrames or whether to use the abstractions on top of either one of them. The second choice then becomes whether data should be represented in a table-like form or in a more graph and collection-based form.

With all the information gathered it was decided to use GraphX for the prototype described within this thesis since it combines a graph-view and a view on the data in the form of collections which are easy to use. Both views offer flexibility since they are a low-level representation of the data. Furthermore, the possibility of being able to analyse data with graph-specific algorithms in the future was thought of when making the decision. Since the main focus of the abstraction, pivot and reification implementations described within this thesis is transforming data rather than making ad-hoc queries, also GraphFrames advantage of making it possible to query the data easily did not change the decision for GraphX.

Another reason for not choosing GraphFrames (GraphFrames Overview, 2022) as of yet was that it is currently not part of the Apache Spark core. The developers list the following reasons: There are still changes to be made to the API of GraphFrames, some features that GraphX offers like partitioning are not implemented and also for development-cycle reasons. Especially the partitioning aspect was a consideration when choosing GraphX over GraphFrames.

This is not to say that GraphFrames could not also be used in future work. It would have to be determined whether GraphFrames could lead to further performance improvements or simpler transformations of the data but this was not tested for in the scope of this thesis.

Another decision that was made was not to use SPARQL querying. Even though when analysing already implemented similar approaches that use Spark to process RDF data, it was obvious that SPARQL queries are widely used in the implementations as the preferred query language. This possibly stems from the fact that SPARQL is the recommended way of querying RDF data. However, since in the KG-OLAP prototype implementation by Schuetz et al. (2021) SPARQL was used already and did not lead to sufficient performance, it was decided to try and only use low-level Spark methods for transforming the data. The decision was made in order to better be able to reap the benefits of Spark without having to rely on performance of SPARQL queries and without having to implement specific distributed querying either. Moreover, using RDDs can assure overall flexibility within the application for future additional aspects.

4.3. RDF and GraphX Mapping

In this section it is described how the GraphX specific data model was constructed out of the RDF source data. It was studied how to best represent RDF data in accordance with the way that Spark and GraphX work with data. Therefore, it was first analysed whether there were already any mappings from RDF to GraphX in literature that could be used. Then, a mapping was defined keeping the KG-OLAP query operations that should be implemented and performed in mind. All decisions were made based upon the objective of gaining performance advantages.

First, a decision needed to be made on how the mapping between the RDF data – in this case not only triples but quadruples – should be done in general. A similar mapping to the work from Schätzle, Przyjaciel-Zablocki, Berberich, et al. (2016) was chosen, which they use in their implementation of S2X. In short, this choice led to a model in which RDF objects and subjects are mapped to GraphX vertices and predicates are represented as edges between those vertices.

The following sections describe the decisions made for subjects, objects, relationships, named graphs and certain further information of the RDF-quads used within this thesis' described prototype in more detail.

4.3.1. Subjects and Objects

Per definition, subjects in RDF can only be resources and not literals, which means they are usually represented as IRIs. In this thesis' described prototype, RDF subjects are therefore read and stored as vertices, using the IRI as the property (data type String) and a 64-bit long Integer as the identifier, since those IDs are required when using GraphX. The identifier is created using a simple hash function which will be described later.

With RDF objects, there are two types that need to be handled differently: resources and literals. As with subjects, when an RDF statement's object is a resource, in GraphX the IRI is simply stored as the property of the vertex and again a generated ID is added. However, whenever there is a literal in the source data, its value is retrieved as a String and stored as the vertex property. For example, the literal *"hello world^^String*" is stored with the property *"hello world".* Again, an ID (64-bit long Integer) is added to uniquely identify the literal vertex. In order not to lose knowledge about the data type of the literal value, this information is stored in the edge as a property whenever this literal (object) is the destination vertex of the edge. This concept will be explained in the next sections about predicates in more detail.

4.3.2. Predicates

In GraphX, edges represent the relationship between two vertices. Therefore, every RDF predicate is mapped to a GraphX edge. This edge contains the predicate as a property, as well as the datatype of the destination vertex which can either be "*resource"* if the destination vertex is a resource or the specific datatype if it is a literal (for example: "Integer", "String", "Boolean" etc.). In doing so, information about datatypes is not lost. Whenever literal values stored as the

edge property must be parsed to be used for calculations or analyses, its datatype can therefore easily be retrieved. The edge then also references the vertices that it connects. Edges hereby do not contain the vertices themselves or their attributes, only their IDs are stored directly with the edge referencing the vertices.

4.3.3. Identifiers

Since – as already mentioned – GraphX demands all vertices to be identified by a 64-bit long ID, it was decided to use a hashing algorithm (in this case *MD5*) to generate this ID. Hereby, the actual value of the RDF literal (the value) or resource (the IRI) is used and hashed into a 64-bit long ID by using the following function call in Java:

UUID.nameUUIDFromBytes(vertexObject.toString().get-Bytes()).getMostSignificantBits()

This way the nodes are identified uniquely within the dataset. When creating IDs in this kind of way it also facilitates the process of generating and adding new vertices to the graph since their ID does only depend on the vertex value and can therefore be created independently of other vertices that are already present in the graph. It does not require storing information about already used vertex IDs or having to query the data when adding new vertices. Hashing also makes sure that vertices are not duplicated in the graph.

4.3.4. Context

Another important consideration regarding the mapping of RDF to GraphX is how to deal with named graphs, which in KG-OLAP represent the context of the RDF triples they are valid for. Since the source data consists of quadruples *(<subject> <object> <predicate> <graph>)* and not just triples (*<subject> <predicate> <object>*) and GraphX does not support operations on multiple graphs, another way of dealing with the contexts had to be found so that the information is not lost. Therefore, it was decided to store the context information – the name of the graph for which the RDF statement is valid – as a property of all edges that are contained within the specified cell. It was decided that the context should be stored with the edges and not the vertices, since in KG-OLAP RDF statements (combining an edge and two vertices) are only valid in a specific context, however, the vertices themselves may be used across multiple contexts. Therefore, it made sense to store context information with the edges rather than as a vertex property. This also potentially saves the number of vertices that need to be stored, since it reduces the total number of distinct vertices when the context is ignored while hashing the vertex values. In the data model, the resulting edge then therefore consists of the relationship predicate, the target datatype, and the context for which it is relevant. In terms of actual code, those three properties are stored as fields within a Java class called *Relation*. This will be explained in Chapter 5.3: *Implementation Details.*

4.3.5. Type Property

In RDF there can be numerous different types of predicates that describe a relationship between nodes. One of them is the special predicate *rdf:type* which associates a subject with a certain type. Even though statements containing the type property can also be regarded as a simple relationship and therefore mapped to a GraphX edge, it might also be interpreted as a direct property of the subject vertex since its type is described. Therefore, when constructing the GraphX data model there were two options to work with this specific predicate:

- Option 1: *<Subject-IRI> <rdf:type> <Object-IRI>*
    - The subject becomes a vertex with the IRI as its property
    - The object becomes a vertex with the IRI as its property
    - The predicate *rdf:type* is represented as an edge between subject and object with the relationship *rdf:type*
- Option 2: *<Subject-IRI, type=Object-IRI>*
    - The subject becomes a vertex with its *IRI* as its property and the object-IRI (the type of the subject) becomes an additional property of the subject itself
    - The object (denoting the subject's type) is not represented as an additional vertex
    - The predicate *rdf:type* is not represented as an additional edge

The reason why it was decided to go for option 2 in the case of this thesis has two components. First, some of the query operations and transformations that are performed on the data in KG-OLAP include a filter step which filters the data for certain types of vertices. Therefore, with option 2 when filtering the graph, only the vertices (and not all edges) need to be iterated through since the type-information needed is directly stored with them. The second reason was that by storing the type directly with the vertices the size of the data was reduced overall since less edges and vertices are created. This might lead to performance improvements since there is a smaller number of edge triplets in the graph to be processed and the type-objects are not present as vertices either. Nevertheless, it could be argued that also option 1 would be feasible and for other use cases might even be the better option especially since this would mean that the same type-object would only be present once (as a vertex) and not duplicated for multiple vertices as their property. However, in this specific case it was still decided on the second option.

4.3.6. Resulting RDF and GraphX Mapping

After the considerations illustrated in the sections before, the ultimate way of mapping RDF data to the Spark GraphX structure for this thesis' described prototype can be seen in Figure 7:

RDF	GraphX
subject	source vertex
predicate	edge
object	destination vertex
graph	context (edge property)
rdf:type	type (vertex property)

Figure 7: RDF – GraphX Mapping

The table in Figure 7 shows again that when mapping elements from RDF to GraphX, subjects and objects become vertices and predicates become edges. The graph name becomes the edge property *context* and the object of any type-relation becomes a direct property of the subject of the RDF statement (and therefore of the GraphX vertex).

4.3.7. Example Graph Construction

With the defined mapping the following example shows with a concrete small dataset how the intial base graph is constructed when transforming RDF triples into a GraphX graph. This constructed graph then constitutes the initial graph that KG-OLAP query operations may be performed on using Spark.

The example RDF data constists of quadruples indicating that the triples are context-dependend. Therefore, for example *Jane* is of type *Person* within *context 1* and *Frank* works at the *IT department* within *context 2*:

```
<Jane > <rdf:type> <Person> <context1>
<Jane> <hasMother> <Lisa> <context1>
<Jane> <hasAge> <24> <context1>
<Frank> <rdf:type> <Person> <context2>
<Frank> <hasWorkplace> <IT Department> <context2>
```

Now when the mapping as described above is applied to the dataset, the following collections are constructed which can be seen in Figure 8 and 9:

Src ID	Dst ID	Dst type	Predicate (relationship)	Graph (context)
1	2	Resource	hasMother	context1
1	3	Integer	hasAge	context1
4	5	Resource	hasWorkplace	context2
...	...	...	...	...

Figure 9: Resulting edge collection

ID	Value	Type
1	Jane	person
2	Lisa	
3	24	
4	Frank	person
5	IT Department	
...	...	

Figure 8: Resulting vertex collection

The collection – represented as a table – on the right in Figure 9 represents all objects and subjects that are present in the RDF quadruples adding their IRI (here those are concrete Strings) as the value property and every rdf:type statement is added directly as a second property: *type*. The edge collection in Figure 8 contains only the IDs of subjects and objects pointing to the vertex collection, the data type of the destination vertex, the actual predicate in the relationship column and also the context for which the edge is valid.

It has to be noted here again that theoretically there would be the possibility for both the vertex and also the edge table to include multiple more columns with different additional properties. However, for the data and concept used within this thesis implementation, this was not included in the illustrations.

5. Data Processing

The following section first shows how the GraphX graph object representing the KG-OLAP cube is constructed in the prototype described within this thesis. Then, the query operations that can be executed on the graph are described as they are defined by Schuetz et al. (2021). Afterwards the underlying algorithms are broken down into steps and mapped to Spark-specific methods.

5.1. GraphX Graph Construction

For this thesis' described prototype to be able to process RDF source data, first it needs to be transformed into an initial base graph representing the KG-OLAP cube without any transformations performed on the data. Therefore, a generator for this graph object was implemented using Spark additionally to the query operations. This creation of the graph is not included in the later performance experiments as it is regarded to be a data preparation task and not an actual transformation or query within the KG-OLAP context, which is the main focus of this thesis. Nevertheless, it is briefly outlined how the graph construction works.

In this thesis it is assumed that the RDF source data is present as an N-Quad file. Simply speaking, in such files every line contains one RDF quadruple. There is also the possibility to easily transform other RDF formats like TriG and so on into N-Quads before then creating the actual GraphX graph.

The generator of the GraphX graph then uses this N-Quad file as an input and reads it line by line before applying the mapping definitions from Chapter 4.3: *RDF and GraphX Mapping* using a variety of Spark and Java methods. Figure 10 illustrates the steps of this process:

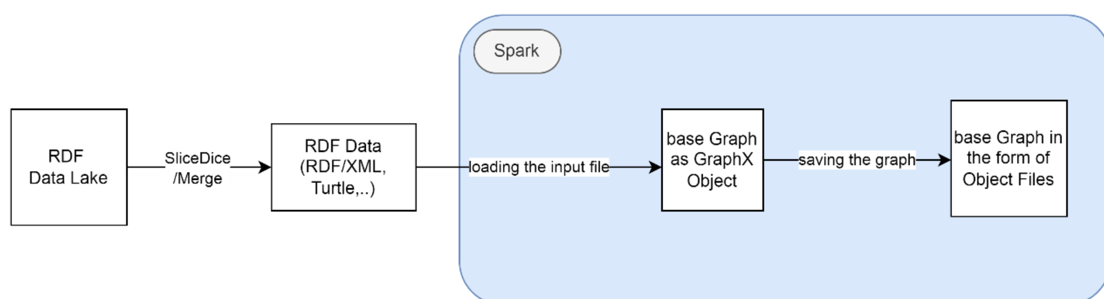


Figure 10: Base graph construction

As it can be seen from the simplified illustration in Figure 10, the steps in the blue box are part of this thesis and implemented in the described prototype using Spark. The steps before could vary and are out of the scope of this thesis. Therefore, data sources might be sliced and diced or merged beforehand. They may be stored for example in a Data Lake, Lakehouse or Data Warehouse. Then, the data is assumed to be present in any form of RDF notation or serialized format which can be transformed to N-Quads and read by the Spark application of this thesis.

After the data is loaded, a GraphX graph object is created and all RDF statements with their subjects, objects, predicates and names graphs are mapped accordingly. The constructed graph is stored as object files in persistent storage and forms the basis for all KG-OLAP query operations. Object files here are a serialized form of the data within the GraphX graph object. Those files can then easily – and very fast – be read and loaded again using Spark whenever transformations should be performed on the data which will be explained in the further sections.

5.2. KG-OLAP Operations

After the initial graph construction, transformations and query operations can be performed on the data contained in the base GraphX graph. Hereby in general the graph is first loaded from the object files that were created with the GraphGenerator using Spark methods. Then, the transformations and operation are executed, a new graph object is created and again persisted to storage as object files. This process is illustrated in Figure 11 in a simplified way:

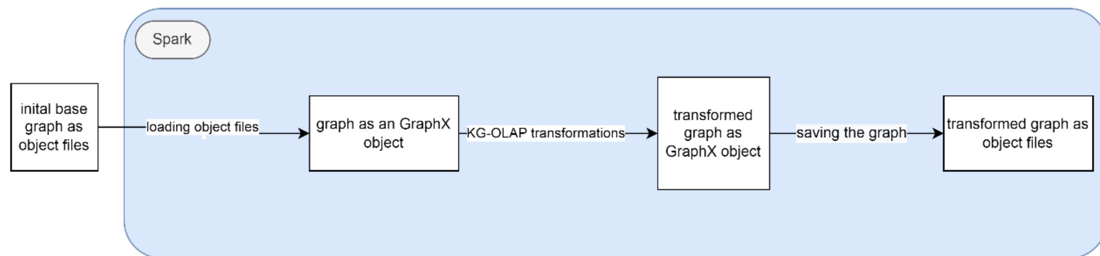


Figure 11: Graph transformation

The following KG-OLAP operations are implemented in this thesis' described prototype as defined by Schuetz et al. (2021):

- roll-up
- pivot
- reification

For the roll-up operation, usually two steps are needed: a merge operation and a type of an abstraction. The abstraction can be of three different kinds: triple-generating abstraction, individual-generating abstraction, or value-generating abstraction as defined by Schuetz et al. (2021). In this thesis' described prototype, it is assumed that merge and/or slice-and-dice operations have already taken place beforehand and therefore only the three categories of abstraction are implemented without the merging step. Moreover, reification and pivot operations are realized within this thesis' described prototype. All query operations are then executed on the before constructed graph object with the help of Spark and GraphX libraries.

To be able to use Spark and its parallelism optimally, first the algorithms needed in the KG-OLAP context were analysed in detail and described in a theoretical way step by step in the following sections. Hereby the concepts of Schuetz et al.(2021) are briefly summarized to understand what the different operations do and how they work in general. Then, those steps are translated into more technical terms and algorithmic structure in order to then match the steps needed for the operations to Spark functions and methods. This was done to also determine which of the steps could be optimized through Spark regarding performance by focussing on the use of narrow transformations whenever possible as described in Chapter 2.6.3: *Further Spark Terminology and Concepts*.

Even though the *slice-and-dice* and the *merge* operation were not included in the implementation in this thesis and are therefore not evaluated for their performance in the experiments,

their theoretical functionality and the steps involved were nevertheless described briefly in the following sections as well. In future work those operations could therefore also be implemented in Spark. For all other implemented operations that are the main focus of this thesis, a more detailed description and illustrations were constructed to better demonstrate how they work.

Use of Broadcast Variables

In terms of data processing there was also a decision on using broadcasted variables which are used within most of the operations and aim at improving Spark performance.

In KG-OLAP some of the operations require the graph's edge triplets to be filtered depending on a list of values or IDs of certain vertices. In the algorithms, those vertices are first extracted by filtering the graph on vertices or edges that fulfil a certain requirement. Such conditions may for example by a certain type-property of a vertex or a value of a relationship. The filtered vertices are then stored in a collection that is then later to be used as a lookup collection for the further steps in the algorithms.

In order to now efficiently use this generated list of vertices for further filtering of the graph, one needs to iterate through the list checking whether a certain vertex is contained. Using those collections, however, require them to be present as lists (and not RDDs anymore) which means that the data has to be collected with the Spark method *collect().* This function may be quite expensive to use as it brings all the data from partitioned RDDs back to the driver. Therefore, if the list is rather long this might lead to performance issues.

This is why using broadcast variables that are available within Spark was considered. Broadcast variables are designed so that their content (their value, which in this case is the list of vertices) is wrapped up by Spark and then sent and copied to all parallel-working nodes once. Since it is read-only, it therefore does not need to be done multiple times but only distributed to all workers once and thus decreasing data distribution across partitions and nodes.

5.2.1. Triple-Generating Abstraction

The *triple-generating abstraction* defined by Schuetz et al. (2021) transforms the graph by replacing certain individuals by their *grouping* object which is an abstraction of multiple individuals grouped together. The grouping-instance is specified through a triple's predicate value – a relationship between two nodes – where the triple's subject should be replaced by the triple's object. Then, all those subjects that are connected to the same object are grouped together since they are all replaced by the same grouping object. Therefore, the individuals are essentially merged together into a group which is a more abstract individual than the individuals each on their own.

This grouping is performed within a specific context or multiple contexts that may be defined. The data including the grouped structure again is then returned as a new graph. Thereby all triples with the predicate that is used for grouping at the beginning is retained in order to keep this information about the original – later replaced – individuals.

Figure 12 containing an illustration of the algorithmic steps of the triple-generating abstraction shows in a simplified way how the algorithm works when looking at specific triples or rather quadruples directly. Here the rectangles represent RDF quadruples or RDD collections containing edges or vertices. The arrows between the collections represent transformations performed on the data leading to intermediate and eventually the end result.

In this case, first the quadruples are filtered for ones that contain *predicate1* and a subject that is of *type1* which can be looked up in the vertex collection on the right of Figure 12. Then, in the filtered quadruples the subjects are replaced with their corresponding object. Therefore, at the end in all statements that contain *subject1* or *subject2*, those subjects are replaced by *object1*. This leads to a grouping of *subject1* and *subject2* together. The original edges containing *predicate1*, however, stay the same.

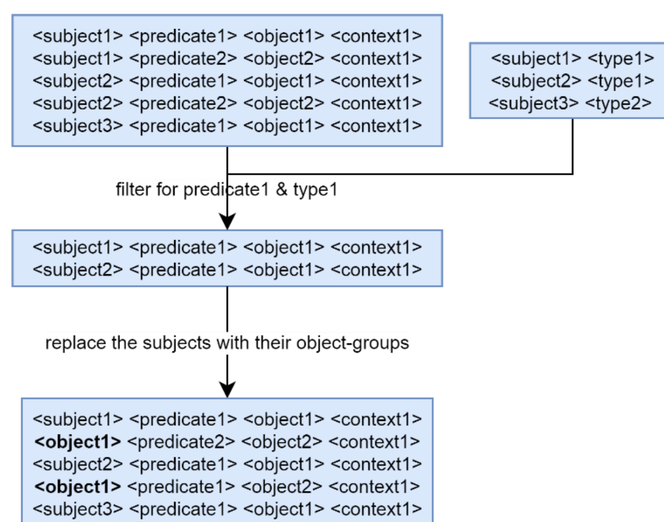


Figure 12: Triple-generating abstraction execution steps

The next illustration in Figure 13 shows a part of the initial KG-OLAP graph and the result after the triple-generating abstraction in a graph-oriented way. Here not all intermediate steps are shown but instead more specific ATM data is used to show the algorithm in a graph view as well. All circles therefore represent nodes or vertices in the graph, all arrows represent edges with a relationship attribute between them. The dotted arrows indicate that here the relationship is not represented as an edge but rather a direct attribute of the vertex that it is connected to, which in the case of this thesis is the *type* property. The type itself is therefore also not illustrated as a node (not a circle in the illustration) but an oval.

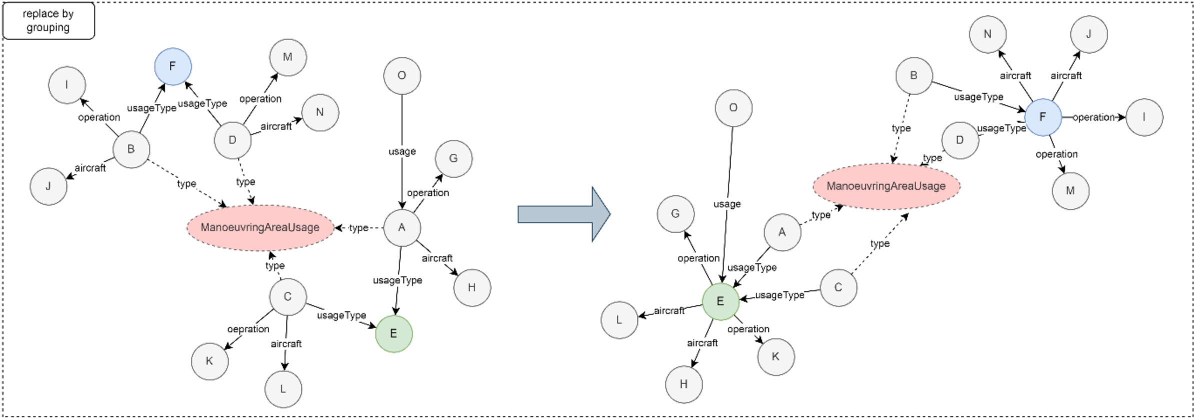


Figure 13: Triple-generating abstraction example graph

It can be seen that in this example graph, the nodes *A, B, C* and *D* are the filtered vertices that should be replaced by the nodes that they are connected to via the <*usageType*> predicate. This means that nodes *A* and *B* should be replaced with vertex *E* and nodes *C* and *D* should be replaced with vertex *F*. After the replacement is done, all other nodes that were connected to *A*, *B, C* or *D* are now connected to their replacements (*E* and *F*) instead. The only relationship remaining the same is the type-relation as well as the original <*usageType*> relationship that was used for the filtering at the beginning.

The following table shows the steps and which functions are used in the implementation of the triple-generating abstraction in this thesis with Spark. It is also stated whether those functions or methods can be categorized as a wide or a narrow transformation.

Table 2: Triple-generating abstraction steps

Spark Method	Description	Categorization
	The algorithm takes the following parameters as input: - *graph* - *replacementObject* - *groupingValue* - *(optional: list of contexts)*	
Triplets.toJavaRDD	*Get the triplet view of the graph data as a RDD*	*Narrow*
Filter	*Go through all those edge triplets within the graph and filter them for* - *edges where the subject is of a certain type that has been specified as the replacementObject* - *edges where the relationship (edge property) contains the supplied groupingValue* - *(and optionally edges of a certain context within the supplied list of contexts)*	*Narrow*
Map	*Map the filtered edge triplets to edges of the form:* - *<sourceID> < Relationship property> <destinationID>* *This results in edges where the source vertex is the individual that should be replaced, and the destination vertex is the individual that it should be replaced with*	*Narrow*
Collect	*Collect the mapped edges (bring them back to the driver) so that they can be put into a HashMap.* *The HashMap for those pairs acts as a lookup or mapping table.*	*Action*
ForEach	*Put all the collected edges into a HashMap with the following structure:* - *key: sourceID, value: destinationID*	*Not Spark*

	or they can also be imagined in the following way: - *(<vertexToBeReplaced>, <replacementVertex>)*	
Triplets.toJavaRDD	*Get the triplet view of the graph data as a RDD again*	*Narrow*
Map	*Go through the edge triplets again and filter all edges where either the subject or the object (or both) is contained in the mapping table as the <vertexToBeReplaced>, so the key of the key–value pairs* *Map all the triplets either to edges while replacing the subject and/or the object with their corresponding value within the HashMap (<replacementVertex>). All others are mapped to edges without changing any values. This leads to a complete RDD of all changed and old edges.* *Keep the original edges containing the groupingValue relationship the same in order to not lose information about their origin*	*Narrow*
Graph.apply	*Use graph apply to generate the new graph object with all changed and kept edges and vertices*	
	The output of the algorithm is then again a new GraphX graph object.	

5.2.2. Individual-Generating Abstraction

The *individual-generating abstraction* (Schuetz et al., 2021) is similar to the *triple-generating abstraction* since individuals (subjects and objects) are also replaced by a grouping-object. However, here the grouping-object is explicitly generated by adding a new individual that replaces all the vertices of a specific type. Then, a reference is created (an edge) from the original individuals to the grouping object so that the information which individuals belong to which group is not lost. This might increase the number of triples since the grouping edges are added. Again, the operation uses the grouping property and type of individual as well as the context(s) that the operation should be done on as input parameters. The data is returned as a new transformed graph.

Figure 14 shows in a simplified way that in the individual-generating abstraction first the data is filtered for *predicate1* and the objects are mapped to new *object-groups* stored as new vertices. Then, the grouping edges are generated with the *object-groups* as destination vertex. Next, all relevant subjects and objects are replaced by their *object-groups* and the new edges are combined with the old ones. Here again the rectangles represent RDF quadruples or RDD collections containing edges or vertices. The arrows between the collections represent transformations performed on the data leading to intermediate and eventually the end result.

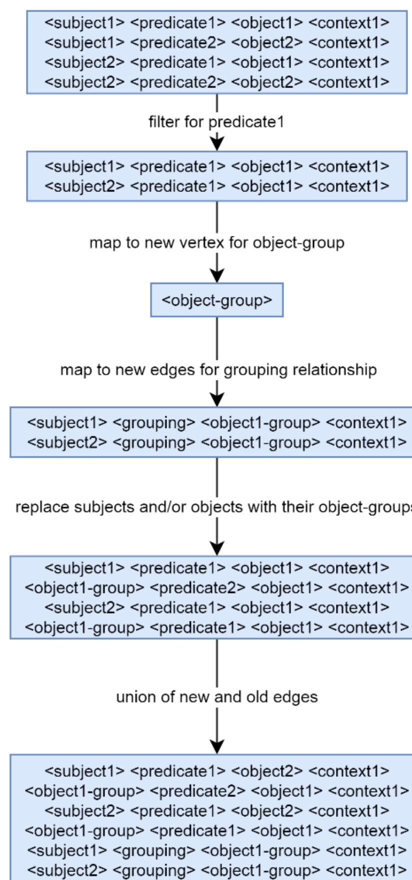


Figure 14: Individual-generating abstraction execution steps

The next illustration in Figure 15 shows a part of the initial KG-OLAP graph and the result after the individual-generating abstraction in a graph-oriented way. Here not all intermediate steps are shown but instead more specific ATM data is used to show the algorithm in a graph view as well. All circles therefore represent nodes or vertices in the graph, all arrows represent edges with a relationship attribute between them. The dotted arrows indicate that here the relationship is not represented as an edge but rather a direct attribute of the vertex that it is connected to, which in the case of this thesis is the *type* property. The type itself is therefore also not illustrated as a node (not a circle in the illustration) but an oval.

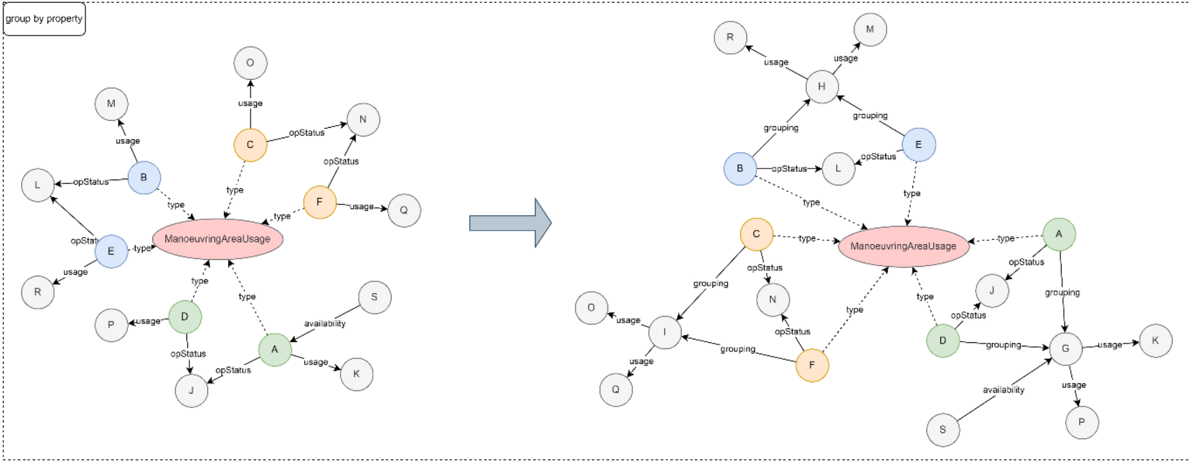


Figure 15: Individual-generating abstraction graph example

It can be seen that in this example graph the nodes *A, B, C, D, E* and *F* are the filtered vertices that should be replaced by new group vertices. Hereby for example *A* and *D* have to be replaced by the same object-group since they have the same <*opStatus*> which was defined as the *groupingProperty*. The same goes for *B* and *E* and *C* and *F*. This means that now new vertices *G*, *H* and *I* are constructed. The grouping relations <*grouping*> are added between the new grouping-vertices and the identified vertices to be replaced from before. Then, those vertices are replaced in all other edges by their groups. After the replacement was done, all remaining nodes that were connected to *A*, *B*, *C*, *D*, *E*, or *F* are now connected to their replacement. The only relationship remaining the same is again the type-relation as well as the original <*opStatus*> relationship that was used for the filtering at the beginning.

The following table shows the steps and which functions are used in the implementation of the individual-generating abstraction in this thesis with Spark. It is also stated whether those functions or methods can be categorized as a wide or a narrow transformation.

Table 3: Individual-generating abstraction steps

Spark Method	***Description***	***Categorization***
	The algorithm takes the following parameters as input: - *graph* - *groupingProperty* - *groupingPredicate* - *(optional: list of contexts)*	
Triplets.toJavaRDD	*Get the triplet view of the graph data as an RDD*	*Narrow*
Filter	*Go through of those edge triplets within the graph and filter them for* - *edges where the relationship (edge property) contains the supplied groupingProperty* - *(and optionally edges of a certain context within the supplied list of contexts)*	*Narrow*
Map	*Map the filtered edges to a Tuple2 class type object of the following structure by adding the String "group" to the original subject attribute and generating a new identifier for this new object:* - *(<SubjectGroupID> <SubjectAttribute + 'group'>)* *Those represent the grouping vertices that are later used to replace other subjects and therefore grouping the statements.*	*Narrow*
Collect	*Collect the tuples (bring them back to the driver) so that they can be put into a HashMap*	*Action*
ForEach	*Put all the tuples into a HashMap with the following structure:* - *(key: sourceID, value: group vertex ID)*	*Not Spark*
Map	*Map the filtered triplets from before to the form:* - *(sourceID, group vertex attribute, context)* *creating Tuple3 objects within an RDD*	*Narrow*

Map	*Map the generated Tuple3 to edges of the form:* - *(sourceID, group Vertex ID, relation)* *by generating the ID of the group vertex attribute and using the supplied groupingPredicate for the creation of the Relation.* *The Relation (edge attribute) then has the following structure:* - *Source Vertex: sourceID* - *Destination Vertex: Group vertex ID* - *Relationship: groupingPredicate* - *Context: context* - *Destination datatype: Resource* *This creates new statements that represent the link between the original subject and their grouping vertex which can be interpreted as: <Subject><belongsTo><Group>. So, the information about the original Subject and which group it belongs to does not get lost.*	*Narrow*
Triplets.toJavaRDD	*Get the triplet view of the graph data as a RDD again.*	*Narrow*
Map	*Go through the edge triplets again and filter all edges where either the subject or the object (or both) is contained in the mapping table as the <sourceVertex>, so the key of the key–value pairs* *Map all the triplets either to edges while replacing the subject and/or the object with their corresponding value within the HashMap (<subjectGroup>). All others are mapped to edges without changing any values. This leads to a complete RDD of all new and old edges.*	*Narrow*
Union	*Combine the edges where subject and or object were replaced or stayed the same with the new grouping edges.*	*Narrow*
Union	*Combine all old vertices with the new grouping vertices*	*Narrow*
Graph.apply	*Use graph apply to generate the new graph object with all changed and kept edges and vertices*	
	The output of the algorithm is then again a new GraphX graph object.	

5.2.3. Value-Generating Abstraction

The *value-generating abstraction* as defined by Schuetz et al. (2021) uses the graph and a predicate – for which the destination attributes (objects) should be aggregated – as its input. Then, an aggregation method needs to be chosen that should be performed on the data. The options here are SUM (sum), COUNT (number of items), AVG (average), MIN (minimum value) and MAX (maximum value). The data is first filtered for triples with the specified predicate, then for each individual that has for example two relationships of the specified predicate-type, the objects of those statements are used and the aggregation operation is performed. The aggregated data is again returned as a graph.

The following illustration in Figure 16 of the algorithmic steps of the value-generating abstraction shows in a simplified way that first the data is filtered for *predicate1* and the triples are mapped to pairs containing the values that should be aggregated, the RDF subject and the context for which the data is relevant. Then, for each subject all its values (for *subject1* this would be *123* and *2*) are aggregated (in this example they are summed up) and new vertices are created for the aggregated values (here *125* for *subject1*). New edges between the new calculated values and the original subjects are created containing the same predicate that was used for the aggregation. Then, the old edges are combined with the newly generated ones.

The notation in Figure 16 again is the same with rectangles representing RDF quadruples or RDDs containing edge triplets or edges or vertices. The arrows show transformations that lead to intermediate and eventually the end results.

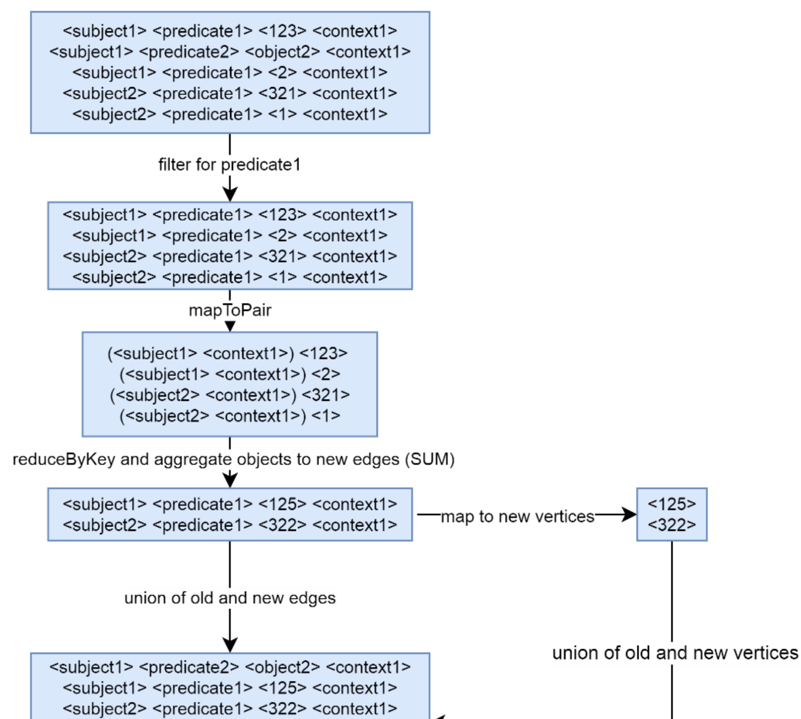


Figure 16: Value-generating abstraction execution steps

Figure 17 shows a part of the initial KG-OLAP graph and the result after the value-generating abstraction in a graph-oriented way. Here not all intermediate steps are shown but instead more specific ATM data is used to show the algorithm in a graph view as well. All circles therefore represent nodes or vertices in the graph, all arrows represent edges with a relationship attribute between them. The dotted arrows indicate that here the relationship is not represented as an edge but rather a direct attribute of the vertex that it is connected to, which in the case of this thesis is the *type* property. The type itself is therefore also not illustrated as a node (not a circle in the illustration) but an oval. Literals are here also represented as oval since they are in fact vertices but different in their type. All other nodes are considered to be resources identified by a IRI.

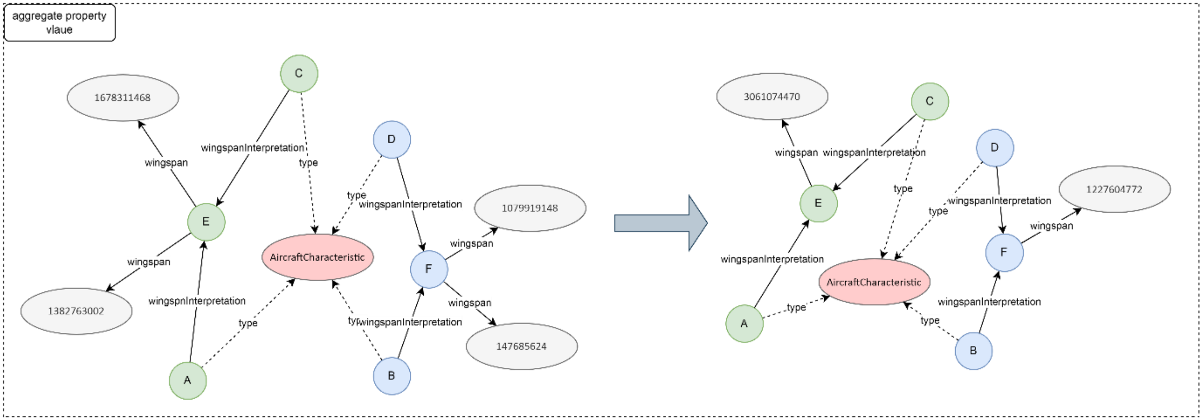


Figure 17: Value-generating abstraction graph example

In Figure 17 it has to be noted that prior to the value-generating abstraction, a triple- or individual-generating abstraction step was already performed which can be seen by the connection between the nodes *D* and *B* to their replacements *F* and *C* and *A* to their replacement *E*. Then, the value-generating abstraction can aggregate the <*wingspans*> of the *groups E* and *F.*

The illustration shows that in this example graph the nodes *E* and *F* are the filtered vertices that should be used to aggregate a certain value they are connected to by a certain edge property. In this case this is the <*wingspan*> predicate. This means that for each filtered individual, all their found wingspans are summed up, creating an aggregated wingspan sum as a new value.

Therefore, the wingspan values for each node (*E* and *F*) are summed up and added as a new node which can be seen in the transformed graph. All other edges between the original nodes stay the same. Also the <*type*> property remains the same as well.

The following table shows the steps and which functions are used in the implementation of the value-generating abstraction in this thesis with Spark. It is also stated whether those functions or methods can be categorized as a wide or a narrow transformation.

Table 4: Value-generating abstraction steps

Spark Method	***Description***	***Categorization***
	The algorithm takes the following parameters as input: - *graph* - *aggregateProperty* - *aggregateType* - *(optional: list of contexts)*	
Triplets.toJavaRDD	*Get the edge triplet view of the graph data as an RDD*	*Narrow*
Filter	*Go through all those edge triplets and filter them for:* - *edges that have the aggregateProperty as predicate (as edge property relationship)* - *(and optionally only the edges with a certain context within the list of contexts)*	*Narrow*
MapToPair	*For all those filtered edges, create key–value pairs containing the source vertex of the edge (plus the context of the edge) and the destination vertex attribute. Hereby the combination of the vertex and the edge context acts as a key and the destination vertex as the value of the key–value pair in the following form:* - *((sourceID, context), destinationAttribute)*	*Narrow*
	The next steps are dependent on the aggregation type	

	AVG	
MapValues	*For both edges and vertices map the values of the pairs to new tuples leading to the following structure:* - *((sourceID, context), (destinationAttribute, 1))* *This represents two tuples within a tuple, the first one being the source vertex and its context from before and the second representing the values that should be aggregated and a "count" which is 1 for each value. The counter is then later used to calculate the average.*	*Narrow*
Reduce-ByKey	*For both the new edges and vertices reduce the pairs by the key (source vertex + context) to bring the values for each key into the form* - *(destinationValue + destinationValue, 1 + 1)* *This step aggregates for each distinct pair the sums of the destination values and the sums of the counts.*	*Wide*
MapVaules	*Map the values of the pair again by dividing the sums by the counts for each key (source vertex + context) to get the average per key*	*Narrow*
Map	*Map the pairs to new edges containing the aggregated values as destination vertex, the source vertex as source vertex and the aggregateProperty as relationship attribute.* *Also map them to new vertices containing the aggregated value as vertex value and a generated ID.*	*Narrow*

	COUNT	
MapValues	*Map the values to 1 to count each occurrence of the pair*	*Narrow*
Reduce-ByKey	*Reduce the key–value pairs to new pairs containing the key (source vertex + context) and the number of the values (counts) to generate the new aggregated values for each key.*	*Wide*
Map	*Map the pairs to new edges containing the aggregated values as destination vertex, the source vertex as source vertex and the aggregateProperty as relationship attribute.* *Also map them to new vertices containing the aggregated value as vertex value and a generated ID.*	*Narrow*

SUM:		
Reduce-ByKey	*Directly reduce the pairs by key to sum up the destination values to generate new aggregated values for each key (source vertex + context)*	*Wide*
Map	*Map the pairs to new edges containing the aggregated values as destination vertex, the source vertex as source vertex and the aggregateProperty as relationship attribute.* *Also map them to new vertices containing the aggregated value as vertex value and a generated ID.*	*Narrow*

MAX:		
Reduce-ByKey	*Directly reduce the pairs by key to find the maximum of the destination values for each pair with a mathematical Java function and generate new values from the found minimum for each key.*	*Wide*
Map	*Map the pairs to new edges containing the aggregated values as destination vertex, the source vertex as source vertex and the aggregateProperty as relationship attribute.* *Also map them to new vertices containing the aggregated value as vertex value and a generated ID.*	*Narrow*

MIN:		
Reduce-ByKey	*Directly reduce the pairs by key to find the minimum of the destination values for each pair with a mathematical Java function and generate new values from the found minimum for each key.*	*Wide*
Map	*Map the pairs to new edges containing the aggregated values as destination vertex, the source vertex as source vertex and the aggregateProperty as relationship attribute.* *Also map them to new vertices containing the aggregated value as vertex value and a generated ID.*	*Narrow*

	The next steps apply to all aggregation types	
Triplets.toJa-vaRDD	*Get the edge triplet view of the graph data as a RDD again.*	*Narrow*
Filter	*Filter the triplets that do need have to be changed which are:* - *all edges that have to be kept as they are i.e., the ones that do not contain the aggregateProperty as edge property (and optionally that are not within the list of contexts), all others received new destination vertices in the aggregation steps before*	*Narrow*
Map	*Map the triples to edges where nothing was changed (keeping all their attributes and source and destination values the same)*	*Narrow*
Union	*Combine the newly generated edges with the edges that were kept the same*	*Narrow*
Union	*Combine the newly generated new vertices with the already existing vertices*	*Narrow*
Graph.apply	*Use graph apply to generate the new graph object with all changed and kept edges and vertices*	
	The output of the algorithm is then again a new GraphX graph object.	

5.2.4. Reification

The *reification* operation (Schuetz et al., 2021) transforms the graph by creating a new vertex that represents a whole triple. The new vertex then gets connected to the original subject, the original object and a new type vertex via supplied predicates. In order to specify the statements that should be reified, a predicate used to find them needs to be provided. Then, for each edge that contains this predicate, a new generated individual, a new type-individual and three edges from this new statement-vertex connecting it to the original subject, the object of the statement and the type are added. The result is again a new graph.

The following illustration in Figure 18 of the algorithmic steps of the reification shows in a simplified way that first the data is filtered for *predicate1* and a new vertex is created to represent the whole RDF statement as one. Furthermore, a vertex for the new *type* individual is created. Then, three edges are created referencing the original *subject1*, *object1* and the *type* of the statement and connecting them to the new statement vertex via supplied predicates which are in this case *<hasSubject>, <hasObject>* and *<hasType>.* Next, all old and new edges are merged. Again, in this illustration the rectangles represent RDF statements or RDDs with edge triplets or vertices and the arrows between them show transformations leading to intermediate or eventually the end result.

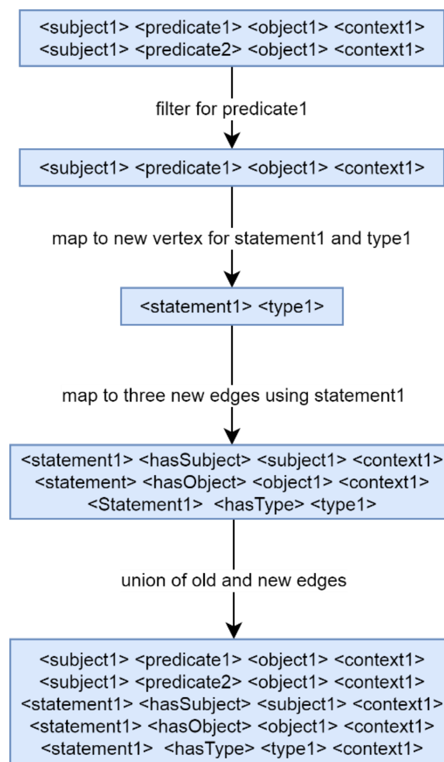


Figure 18: Reification execution steps

The next illustration shows a part of the initial KG-OLAP graph and the result after the reification operation in a graph-oriented way. Here not all intermediate steps are shown but instead more specific ATM data is used to show the algorithm in a graph view as well. All circles therefore represent nodes or vertices in the graph, all arrows represent edges with a relationship attribute between them. The dotted arrows indicate that here the relationship is not represented as an edge but rather a direct attribute of the vertex that it is connected to, which in the case of this thesis is the *type* property. The type itself is therefore also not a node (not a circle in the illustration) but an oval.

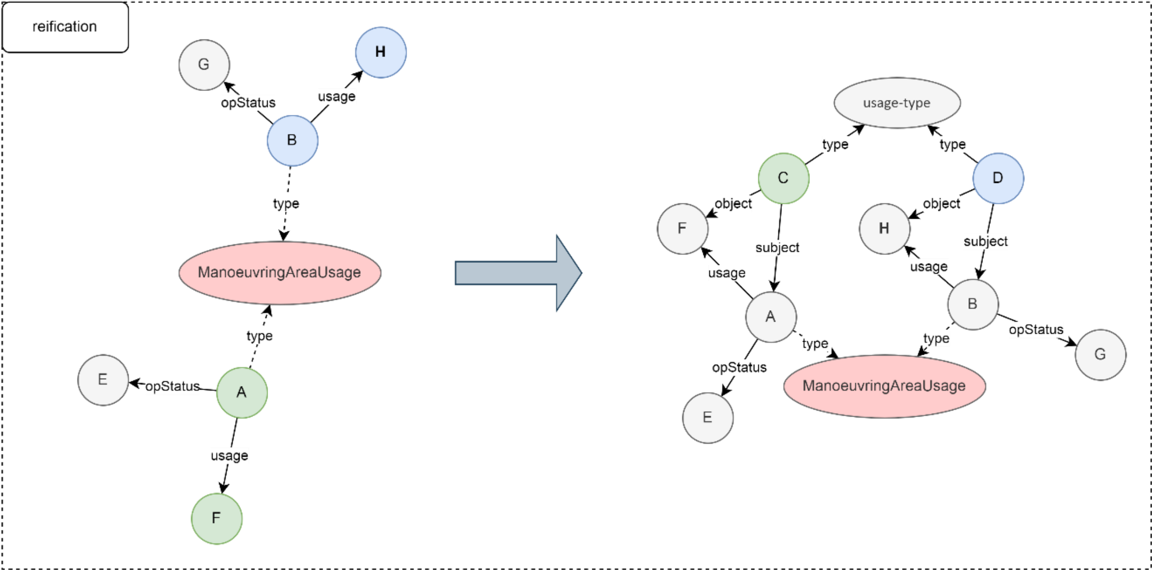


Figure 19: Reification graph example

From Figure 19 above it can be seen that in this example graph, the statements *<A><usage><F>* and *<B><usage><H>* are chosen to be reified by filtering them through the *<usage>* predicate. Then, for both of those statements a new vertex representing the whole statement (including the relationship, subject and object) is created, which in this example is node *C* representing the statement *<A><usage><F>* and *D* representing *<B><usage><H>.*

Then, the subjects of the original statements (*A* and *B*) are referenced and connected to the new statement vertices (*C* and *D*) via the *<subject>* relation indicating that A and B are the subjects of the statements *C* and *D*. The objects of the original statements (*F* and *H*) are referenced and connected to the statement vertices with the *<object>* relation. Also, a new vertex – the *type* vertex – is created and connected to the statements via the *<type>* relation (not to be confused with the dotted line saying *<type>* as well; this is the vertex property and not an actual edge). The relation that the original statements were filtered by (*<usage>*) and also the type-property stay the same in the transformed graph.

The following table shows the steps and which functions are used in the implementation of the reification operation in this thesis with Spark. It is also stated whether those functions or methods can be categorized as a wide or a narrow transformation.

Table 5: Reification steps

Spark Method	***Description***	***Categorization***
	The algorithm takes the following parameters as input: - *graph* - *reificationPredicate* - *type vertex IRI* - *objectRelation* - *subjectRelation* - *(optional: list of contexts)*	
Triplets.toJavaRDD	*Get the edge triplet view of the graph data as a RDD*	*Narrow*
Filter	*Go through all those triplets and filter them for:* - *edges that contain the reificationPredicate as the predicate (relation property)* - *(and optionally that are contained within the specified contexts)*	*Narrow*
New Vertex	*Create a vertex object with the supplied type vertex IRI as attribute and generated ID to identify the vertex*	*Not Spark*
Parallelize	*Use parallelize to create an RDD from the type vertex object to be able to union it later with the already existing vertices in another RDD*	*Narrow*
Map	*Map the filtered triples from before to instances of the type Tuple4 of the form: within an RDD:* - *sourceID* - *destinationID* - *new statement-vertex attribute (IRI)* - *context*	*Narrow*

Map	*Map the Tuple4 objects to Tuple5 objects within the RDD to generate the ID from the new statement vertex leading to the form:* - *sourceID* - *destinationID* - *new statement-vertex attribute (IRI)* - *new statement-vertex ID* - *context*	*Narrow*
Map	*Map the Tuple5 RDD to new vertices in the form of to represent the new statement vertices by themselves:* - *new statement-vertex ID* - *new statement-vertex attribute (IRI)*	*Narrow*
Map	*From the Tuple5 RDD also map the RDD to edges of the form: to represent the link from the new statement vertex to the original subject:* - *statement vertex ID* - *sourceID* - *subjectRelation (“hasSubject”)*	*Narrow*
Map	*From the Tuple5 RDD also map the RDD to edges of the form: to represent the link from the new statement vertex to the original object:* - *statement vertex ID* - *destinationID* - *objectRelation („hasObject“)*	*Narrow*
Map	*From the Tuple5 RDD also map the RDD to edges of the form: to represent the link from the new statement vertex to the new type vertex:* - *statement vertex ID* - *type vertex ID* - *type-relation (“type”)*	*Narrow*
Union	*Combine the old edges with the newly generated edges*	*Narrow*
Union	*Combine the old vertices with the new vertices*	*Narrow*
Graph.apply	*Use graph apply to generate the new graph object with all changed and kept edges and vertices*	

	The output of the algorithm is then again a new GraphX graph object.	

5.2.5. Pivot

The *pivot* operation (Schuetz et al., 2021) adds dimensional values – that define a KG-OLAP cell or context – to certain individuals within that context. Therefore, the dimension that should be used for the pivot operation and which of the individuals should receive the connection to this dimension value needs to be specified. Furthermore, the context(s) for which the operation should be performed may be defined. Then, the graph is filtered for the specified dimension predicate (for example <*hasLocation*>) in the global named graph. Then, for the specified individuals (for example vertices that are of the type <*ManouevringAreaUsage*>) a new edge with the specified predicate and the same object referenced via the original predicate (the <*hasLocation*> relationship) is added to the graph. The result of the operation is a new graph.

The following illustration in Figure 20 of the algorithmic steps of the pivot operation shows in a simplified way that first the data is filtered for *predicate1* which represents the dimension predicate in the global context. The data is also filtered for *predicate2* which contains the individuals for pivoting. It is then checked whether the individuals found are within the correct modules corresponding to the supplied context. If that is the case, they are mapped to a new edge with the *pivotPredicate* (*predicate4*). New and old edges are combined. Hereby again rectangles illustrate RDF statements or RDDs containing edges or vertices, the arrows connecting them show the transformational steps leading to intermediate or end results.

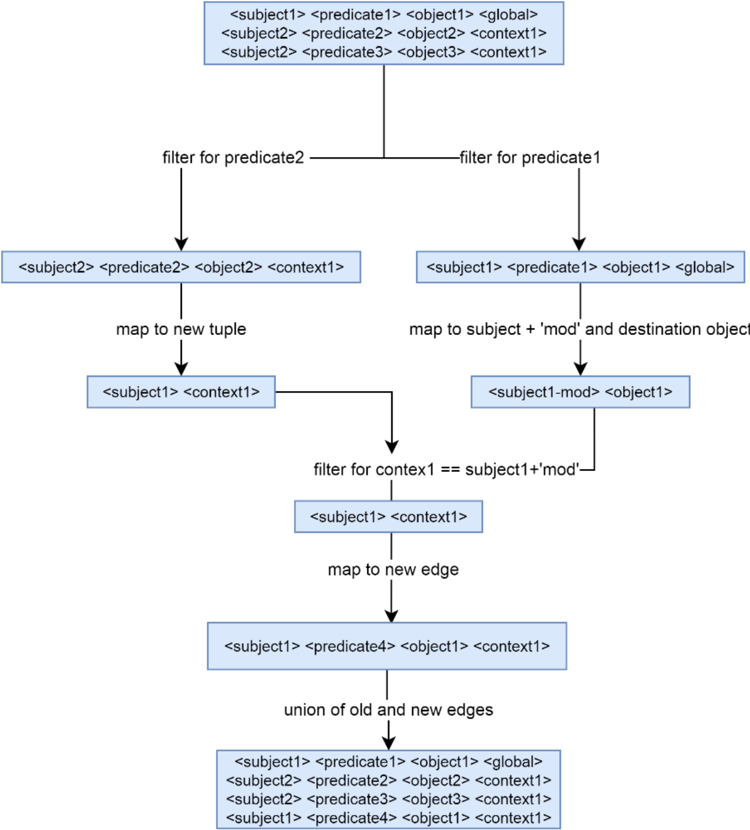


Figure 20: Pivot execution steps

The next illustration in Figure 21 shows a part of the initial KG-OLAP graph and the result after the pivot operation in a graph-oriented way. Here not all intermediate steps are shown but instead more specific ATM data is used to show the algorithm in a graph view as well. All circles therefore represent nodes or vertices in the graph, all arrows represent edges with a relationship attribute between them. The dotted arrows indicate that here the relationship is not represented as an edge but rather a direct attribute of the vertex that it is connected to, which in the case of this thesis is the *type* property. The type itself is therefore also not a node (not a circle in the illustration) but an oval. Literals are represented as ovals even though there are technically also normal nodes in the graph. Contexts are represented as dotted boxes indicating that the data the box contains is valid for this specific context.

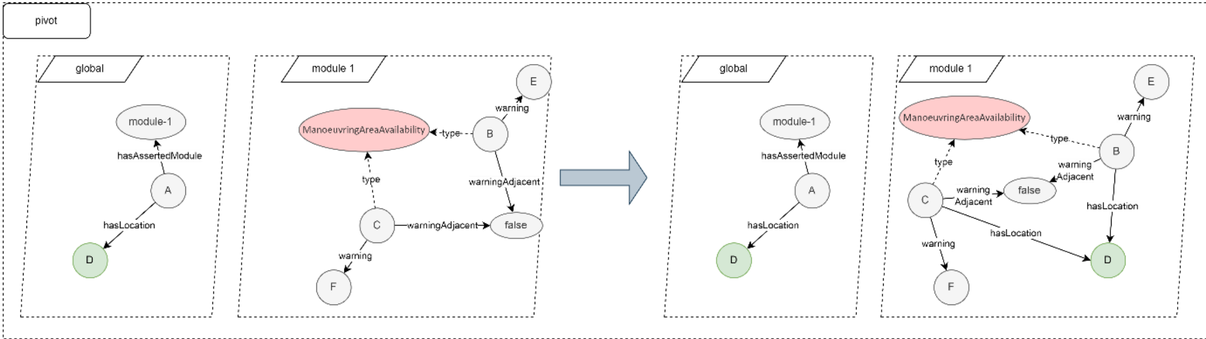


Figure 21: Pivot graph example

It can be seen that in this example graph in Figure 21, first the modules that have the correct value at their <*hasLocation*> relationship (here node *D*) are filtered within the *global* context which results in the vertex *A.* Then, the corresponding modules of cell *A* (here <*module-1*>) are used for the pivoting. Within the <*module-1*> context, triplets with the predicate <*warning*> – supplied as the *selectionCondition* – are filtered. The subjects of those statements (here nodes *B* and *C*) are the ones that should receive the pivoting relation to the dimension value vertex *D*. Those required edges are then created within *module-1* by using the pivot predicate <*hasLocation*> which results in the edges <*B*><*hasLocation*><*D*> *and* <*C*><*hasLocation*><*D*>*.* The data of the *global* context stays the same as well as all other edges that were present before and the type properties are kept in *module1*.

The following table shows the steps and which functions are used in the implementation of the pivot operation in this thesis with Spark. It is also stated whether those functions or methods can be categorized as a wide or a narrow transformation.

Table 6: Pivot steps

Spark Method	Description	Categoriza-tion
	The algorithm takes the following parameters as input: - *graph* - *dimensionPropert* - *pivotProperty* - *type* - *selectionCondition* - *(optional: list of contexts)*	
Triplets.toJavaRDD	*Get the edge triplet view of the graph data as a RDD*	*Narrow*
Filter	*Go through all those edge triplets and filter them for:* - *edges that contain the dimensionProperty as relation attribute* - *this could for example be the “hasLocation” property*	*Narrow*
Map	*Map the filtered edges to new tuples and adding the String “mod” leading to tuples of the form:* - *(<sourceAttribute + “mod”>, <destinationID>)* *By adding “mod” automatically the correct contexts are referenced as by the definition of the KG-OLAP data.*	*Narrow*
Collect	*Collect the created tuples (bring them back to the driver) so that they can be put into a HashMap*	*Action*
forEach	*Put all the tuples in a HashMap with the following structure:* - *key: sourceAttribute + “mod”* - *value: destinationID*	*Not Spark*
Triplets.toJavaRDD	*Get the triplet view of the graph data as a RDD again.*	*Narrow*

Filter	*Go through the whole graph again and filter all triples that contain a subject of the type specified as the selection-Condition (and optionally that are contained within the specified contexts)*	*Narrow*
Map	*Map filtered triples to new tuples of the form:* - *(sourceID, context)*	*Narrow*
Map	*Map the tuples to new edges using the lookup HashMap of the form:* - *sourceID, as sourceID* - *looked up ID as destinationID* - *pivotProperty as relationship attribute*	*Narrow*
Union	*Combine all the old edges with the new edges*	*Narrow*
Graph.apply	*Use graph apply to generate the new graph object with all changed and kept edges and vertices*	
	The output of the algorithm is then again a new GraphX graph object.	

Since also the operations slice-and-dice and merging are part of the KG-OLAP concept, those two operations are described now in more detail. However, the steps included in the algorithm were not mapped to specific Spark methods and functions, since the operations are not part of the practical implementation described within this thesis.

5.2.6. Slice-and-Dice

As described by Schuetz et al. (2021) the *slice-and-dice* operation simply creates a sub-cube of the original knowledge graph cube by defining dimension attribute values that should be included in the subset of knowledge. Thereby the needed cells and their modules are cut out of the initial cube as well as their covered cells as defined through the dimension hierarchies.

In terms of programming, this means that the slice-and-dice operation takes the whole graph object as an input and reduces it to a certain slice or dice. In a graph-oriented view, the operation leads to only certain vertices and edges with certain contexts being included in the resulting subgraph.

Therefore, values for each dimension of the cube need to be specified that should be extracted which can for example be a certain location, a certain importance and a certain date in the ATM setting. Then, all cells (contexts) within the cube with those values as their dimensions are filtered. In the GraphX KG-OLAP data model that would mean that first all contexts defined through the specifies dimension values must be extracted. Then, all edges with one of those certain filtered contexts are included in the subgraph, as well as all the vertices that are connected by those edges.

For the selected cells or modules, all other cells that are covered by them (due to their roll-up relationships) are also included in the subgraph using the <*coverage*> RDF predicate. The data of those cells – that is contained in their corresponding modules – is returned as a new graph which may contain equally or less triples than the original graph object.

The basic steps in this operation are the following:

Input variables: graph, values for all dimensions at which should be sliced or diced

1. *For each dimension*
    1.1. *Go through the whole graph and filter the edge triplets that have an edge with the property <hasDimension> for the specific dimension (for example <hasLocation>) and the specific value as destination vertex (for example <Cell> <hasLocation> <Austria> where <Austria> is the specified dimension value that should be used for the slice-and-dice operation)*

    1.2. *Also include all edges that have dimension values that are covered by the specified value (for example <Linz> is also covered by <Austria>) which are represented as a rollup/coverage relationship in the graph.*

    1.3. *Create a list of those subjects (which are the cells with the specified hierarchy values) with the correct dimension values, those are the contexts that should be included in the sliced or diced subgraph.*

2. *Intersect those resulting lists of those subjects (cells) to find the ones that satisfy all values of all four dimensions.*
3. *Create a list of all those subjects (contexts)*
4. *Go through the whole graph again and now filter all the edges which's context is contained in the list or that belong to the <global> context since this data should also be included as the contained knowledge is applicable for all contexts and therefore always part of any subgraph.*
5. *Create a new subgraph with the filtered edges and vertices.*

Output: new GraphX Graph object

5.2.7. Merge

The *merge* operation as defined by Schuetz et al. (2021) takes the graph and merges knowledge of certain cells at particular levels of dimensions with their covered, “lower” level cell knowledge. Therefore, one needs to specify the hierarchy level – for each dimension that exist in the KG-OLAP cube – that the cells should be merged to. Then, for each of those cells at the particular dimension levels, all covered cells and their modules are retrieved. The context of all edges within those modules is then overwritten, thus merging them to the cell in the specified dimension hierarchy. This leads to all the knowledge from dimension hierarchies from the lower levels now being contained within the same, higher context and essentially merging them together into one.

The basic steps in this operation are the following:

Input variables: graph, level of all dimensions at which should be merged

1. *For each dimension:*
    1.1. *Go through the whole graph and filter the edge triplets that have an edge with the property <atLevel> and the specified level as destination vertex (for example if the value “Level_Date_Month” is specified for the date dimension, all edges that are of the structure <subject> <atLevel> <Level_Date_Month> are retrieved)*
    1.2. *Create a list of those subjects (those represent all the members of the specified dimension hierarchy level, for example <October> <atLevel> <Level_Date_Month> then the subject <October> is put into the list)*
    1.3. *Go through the whole graph again and find edges that have one of those values from the list as destination vertex in the form of for example <Cell> <hasDate> <October>*
2. *Intersect those resulting lists of the found subjects (cells) to find the ones that satisfy all levels of all four dimensions*
3. *For the resulting cells, get all the other cells that are covered by them as well (for example the Month of October also covers the date 4th of October)*
4. *Create a list of all those covered subjects (contexts)*
5. *Go through the whole graph again and find all statements with a context present in this list*
6. *Replace their context with their corresponding covering context (which is from a hierarchy level above) which merges lower hierarchy level data to contexts at levels above*
7. *Create a new graph with the merged data*

8. *Output: new GraphX graph object*

When looking at the description and the details on the tables of each of the operations that were implemented, it can be seen that it was tried to stick to as many narrow transformations as possible. This was done in order to make sure that performance is as optimal as possible regarding Spark. Whenever a list needed to be created to be iterated through, this list was converted to a broadcast variable to make sure that every node can use the list in parallel and the list is sent to the workers only once.

5.3. Implementation Details

The following section contains detailed information about the actual Java program that was written when implementing the prototype for KG-OLAP using Spark (GraphX). The code includes both the construction of the base graph from N-Quads as well as the specific KG-OLAP operations on the cube, generating a new transformed graph as an output.

5.3.1. Class Relation

In the technical data model established within this thesis' described prototype, edges between vertices in the GraphX graph are of the type *Relation* which is a Java class that represents the relation between two vertices with certain attributes describing this relationship.

The Relation type therefore consists of the multiple fields. The field *relationship* is used to store the original predicate value of the RDF triple that the edge is created from. Here simply the String value of the predicate is used. The field *context* contains the name of the graph of the original dataset that was supplied to the *GraphGenerator* when creating the initial GraphX graph. The field *targetDatatype* stores either the type of the literal (for example *String, Integer, Boolean*) of the original RDF object (the destination vertex of the GraphX edge) as a String or the String "*Resource"* when the destination vertex is a RDF resource and not a literal.

5.3.2. Classes Vertex, Resource and Literal

The *Vertex* interface is required in order to be able to create the initial graph object (which is done with the help of the *GraphGenerator* class) where there is a function accepting Resources and Literals and storing them in the same collection. Therefore, an interface was created to be able to abstract from the more specific classes *Resource* and *Literal*.

The classes *Resource* and *Literal* therefore then implement the Interface. A *Resource* is used to represent any RDF resource (object or subject) that is not a literal and has the fields *value* and *type* to store the IRI of the RDF subject or object and the *rdf:type* property if there is one present in the source data. The *Literal* class can be used for any RDF literal as representation where only the value of the RDF literal is stored in the field *value*.

5.3.3. Class GraphGenerator

The *GraphGenerator* class represents functionality for creating a new initial base graph directly from the RDF data source. Therefore – in the case of this thesis – a N-Quad file is supplied as input which is a serialized form of RDF triples. Each row contained in the source file is iterated through and analysed for its contents. The class then creates vertices from all subjects and objects in the RDF statements as described in Chapter *4.3: RDF and GraphX Mapping*. Afterwards, the class creates edges connecting the already created vertices. Hereby, the RDF-predicate is stored as a field value of the edge which is of class type *Relation* describing the relationship between the two vertices. Moreover, the graph name within the RDF statements is stored in the edge as an additional field which serves as the context described in *4.3.4* Context.

It is important to note that in general every single distinct RDF statement is translated into one edge in the GraphX graph. However, all RDF statements that describe a *rdf:type*-relation (predicate = *<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>)* are not converted into edges in the graph. In this case the *type*-property is directly stored with the subject-vertex as an additional field.

The following methods contained in the class are involved in this process:

getJavaRDD

This method loads the N-Quad file from a specified path and reads every line of it separately It thereby also needs a supplied *JavaSparkContext* that must be created as parameter in order to do so.

The base data might look like this within the N-Quad File:

```
<A> <type> <x> <Graph1> .
<A> <hasParent> <B> <Graph1> .
<A> <hasChild> <C> <Graph1> .
<B> <hasParent> <D> <Graph2> .
<D> <hasAge> <5> <Graph2> .
```

Then, the method creates a quadruple (stored as instance of the class *Quad* which is generated with the help of the *RDFManager* of the Jena library) for each found valid line. A line is valid when it is not empty, not a comment and has length > 1.

Those quadruples are directly stored in a *JavaRDD<Quad>* which is then returned as the result of the method. A JavaRDD is the specific Java version of a Spark RDD.

generateGraph

This method first calls the *getJavaRDD* method to generate the *JavaRDD* containing the quadruples. Then, all RDF statements (each represented as a Quad) are filtered for ones that do not contain the type-relation as predicate. For each of those statements, an edge is created and put into a RDD of the form *JavaRDD<Edge<Relation>>.* The Edge class is part of Spark and represents edges with a specified type which in this case is the *Relation*-type. The Relation-class then contains fields that describe the edge which in the case of this implementation is the context, predicate and also the destination vertex datatype as already described.

All subjects of such statements are mapped to new vertices of type *Resource* since there can only be resources and no literals in the subject of an RDF statement. Resource again is a custom created class that contains a field for the vertex value and the *type* field.

All objects of those statements are also mapped to vertices, but it is first checked whether they are in fact literals or if they are also resources. In the example from above, subjects *A, B* and *D* are all resources. Object *C* is also a resource and object *5* is a literal:

```
A = resource
B = resource
C = resource
D = resource
5 = literal
```

In case of literals, the literal value is used as the vertex value. For resources, again their URI is stored as its value. A list of statements without type-relation is now generated that might look like this in an abstract way:

```
[A(no type), B(no type), C(no type), D(no type), 5(no type)]
```

Then, all statements that contain the type-relation are filtered and mapped to tuples with an ID and the vertex itself which again contains the URI of the subject and also the object of the statement which describes the subject's type.

In this example, subject *A* is the only subject where there is a type-relation present in the whole graph. All other statements and therefore all other subjects do not have a type.

```
<A> <type> <x> <Graph 1> .
<A> <hasParent> <B> <Graph 1> .
<A> <hasChild> <C> <Graph 1> .
<B> <hasParent> <D> <Graph 2> .
<D> <hasAge> <5> <Graph2> .
```

Therefore, only the vertex *A* is stored in the list of vertices with *a* type.

```
[A (type x)]
```

Then, the list of vertices without a type are compared with the list of vertices containing a type since both may contain the same vertices as all RDF statements are considered here. All vertices that are already contained in the list of vertices with a type are removed from the list of vertices without a type to remove duplicates.

```
[A(no type), B(no type), C(no type), D(no type), 5(no type)]
MINUS [A(type x)]
= [B(no type), C(no type), D(no type), 5(no type)]
```

Then, the rest of the vertices without a type that remained after the subtraction is combined with the vertices that do have a type to generate a distinct list of subject-vertices with as much (type) information as possible:

```
[B(no type), C(no type), D(no type), 5(no type)] + [A(type x)]
= [B(no type), C(no type), D(no type), 5(no type), A(type x)]
```

Each of those vertices is then mapped to a new instance of type *Tuple2* of with the structure *<Object, Object>*. Hereby the first object donates the ID of the vertex and the second one the actual instance of class Resource or class Literal. Depending on whether there is any type information about the vertex or not the type is stored in the type field of the Resource class. All other Resources receive the default value "no type" as type value. Now the graph is generated out of the edges and vertices and returned by the used method. Hereby the method *Graph.apply()* provided by GraphX is used.

The reason for this way of creating the vertices was to ensure that Spark can – as long as possible – work with RDDs in parallel without having to collect the data to the driver in between.

5.3.4. Interface Transformation

The *Transformation* interface contains the *transform*-method and is implemented by classes that represent KG-OLAP operations (pivot, reification, types of abstraction).

For each KG-OLAP operation there are two different transform-methods (which are included in the interface). One of them uses a list of String values (*ArrayList <String> contexts)* as an input parameter in addition to all other parameters that both use. Therefore, the transform operation is in this case only performed on all the contexts that are defined in this list. This means that before performing the transformations, the dataset is basically filtered by looking whether the edge belongs to a context that is contained in the list.

The second implementation of the transform-method ignores those contexts and executes the operation on the whole dataset without filtering it before doing so. Whenever the context list is used, it is transformed into a broadcast variable so that it can be used by all tasks at the same time when needed. The purpose of broadcasting information is described in Chapter *5.2 KG-OLAP Operations*.

Generally, all the transformations follow a similar logic which will be explained later in the classes' descriptions. In simple terms the steps are the following:

First, triplets are filtered for a certain relationship or subject type. The found triples are mapped to new vertices and/or new edges. This can be done by generating new values or replacing individuals with others. Then, edges or vertices are replaced with new ones, or newly generated ones are added to the graph. Lastly, a new graph is generated and returned.

The following classes implement the Transformation interface and follow the described process in some way.

5.3.5. Class Reification

This method reifies RDF statements that contain a certain predicate. First, a new vertex which is a tuple is created to represent the statement to be reified itself. Another vertex is created to represent the type of the statement. Three edges are created to describe the statement-vertex: a connection to the original subject-vertex, one to the original object-vertex and one to the generated type-vertex.

The steps in the code are therefore as follows. All triplets in the supplied graph that contain a certain relationship (value of the *relationship* field of the edge of class *Relation*) are filtered and then those found statements are mapped to a new *Tuple4* instance which contains:

- the ID of the subject-vertex of the statement
- the ID of the object-vertex of the statement
- a new resource-vertex that represents the statement as an individual (stored as the vertex field *value*)
- the context for which the statement is valid (field *context* of class *Relation*)

Then, the creation of a new type-vertex (the type-vertex value has to be specified by the user and the identifier is then generated with the *getIDOfObject* method) takes place. Next, the type-vertex is put into a list which is parallelized to transform it into a RDD (since it later has to be joined with the other vertices present as RDDs by a union operation).

Afterwards, the statements from before in the form of a list of type *Tuple4* are then further processed by adding a 5th attribute (making them a *Tuple5* type) which is an ID of the newly generated statement generated by the *getIDOfObject* method.

From the list of *Tuple5* objects, the new vertices are created which then results in objects of the following form: *(ID, Vertex (Statement-value)*) for each of the individuals contained in the list of *Tuple5* objects.

Furthermore, three edges for each of the *Tuple5*-vertices are needed:

- New edge with the statement as the source vertex (subject), the subject-relation as the predicate (field *value* of the Relation class) as well as the context (field *context* of the Relation class), and the original subject as the destination vertex of the Relation
- New edge with the statement as the source vertex (subject), the object-relation as the predicate (field *value* of the Relation class) as well as the context (field *context* of the Relation class), and the original object as the destination vertex of the Relation
- New edge with the statement as the source vertex (subject), the type-relation as the predicate (field *value* of the Relation class) as well as the context (field *context* of the Relation class), and the generated type-vertex as the destination vertex of the Relation

Then, all the already existing edges and vertices are joined with the new ones by using the Spark method *union()* which creates a new RDD for both edges and vertices. Eventually, again a new graph is created from the two RDDs.

5.3.6. Class Pivot

The class Pivot also implements the transformation interface. The steps in the code are as follows:

First of all, all the edge triplets of the graph that contain the *dimensionProperty* as a predicate (e.g.: *hasLocation*, *hasDate*, etc.) are filtered and mapped to new tuples which contain:

- The source attribute of the filtered triplets (this is the identifier of a cell in the KG-OLAP cube since statements with *dimensionProperties* contain subjects that represent cells) with the added String *-mod* (by adding '-mod' automatically the identifier for the cell's module is retrieved since modules in KG-OLAP are named like their corresponding cells plus the added String 'mod')
- The ID of the destination vertex (which is the dimension value e.g.: *Linz* or a date)

Those tuples are put into a HashMap since they will later be used to find the correct dimension value depending on the context for which it is applicable. The HashMap serves as a mapping and lookup table.

Then, the triples are filtered again by finding statements in which the subject is of a certain type (*Resource* field *type*) which is defined by the user's chosen *selectionCondition.* In the ATM data set one of those could for example be *ManoeuvringAreaAvailability*.

Those filtered statements are then mapped to tuples containing the ID of the subject as well as the relevant context extracted from the *Relation* (edge) individual. Then, Spark's *distinct* method is used since those subjects might be present more than once in the data set which would lead to duplicate edges being created later.

Then, the found tuples are mapped to new edges of type *Relation* with the subject as the source vertex, the applicable destination objects (e.g.: Linz) as destination vertex depending on the context of the subject (looked up in the HashMap) and a new *Relation* containing the *pivotProperty* which is also be supplied to the method. This can for example be *object-model#hasLocation.* Then, the new edges are combined with the RDD containing the old ones and a new transformed graph is created from edge and vertex RDDs.

In the graph all edges basically stay the same. The only change is that the subject-vertices that are of the type specified in the *selectionCondition* receive a new edge (within their context which is the KG-OLAP module) that points to the dimension value (e.g.: *Linz*). This is the same value that the cell the edges are contained in points to via the *dimensionProperty*.

5.3.7. Class ValuegeneratingAbstraction – aggregate property values

The next class implementing the Transformation interface it the *ValuegeneratingAbstraction*.

Here all edge triplets that contain the *aggregateProperty* as *Relation* field *relationship* of the edge are filtered and mapped to a *pairRDD*. A pairRDD is a special RDD containing tuples with key and value and offers some additional functionality to simple RDDs containing tuples.

Those key–value pairs consist of a tuple as the key which contains the source-vertex of the before filtered triple as well as the context of the triple. The value of the key–value pair is the destination-vertex value which here is assumed to be a numeric value, for example the *wing-span* as an Integer value.

The next steps in the algorithm are dependent on the type of aggregation that should be performed. Available aggregation types are *sum*, *count*, *average*, *max* and *min* which can be supplied to the *transform* method. The basic steps are, however, the same for each of them.

First, new edges are created by mapping the tuples of the previously generated *pairRDD* (that are of the form: (*Tuple2(Vertex(subject), "context"), Literal(numeric value)*) to edges where the source (subject) of the tuple within the tuple stays the subject and the numeric value is aggregated depending on the type (e.g. when counting, the numeric value is mapped to 1 and then summed up, when summing the actual values are summed up, etc.)

New vertices are then created by doing the same aggregation and calculation and then storing the calculated result as the value of the new *Vertex* instance. Additionally, a new ID is generated using the *getIDOfObject* method to identify the new numeric value contained in the vertex.

Last, the new edges and vertices are combined with to the original RDDs and the statements that were aggregated are removed from the graph. A new transformed graph is created and returned by the method.

5.3.8. Class IndividualGeneratingAbstraction – group by property

The next class implementing the Transformation interface is the *IndividualGeneratingAbstraction*. Here all edge triplets that contain the *groupingProperty* as *relationship* field value of the *Relation* are filtered. In such statements, the object is the individual by which the subjects of the triplets should later be grouped. Those are mapped to new tuples of the form: (Vertex(subject), Vertex(group)). The group is generated by simply adding the String *-group* to the already existing attribute value of the objects. Those are put into a HashMap to serve as a lookup and mapping table where the subjects with their corresponding groups can be found later.

For each filtered pair (subject and group), a new edge that represents the link between the subjects and their newly generated grouping vertex they belong to is created.

Then, all triplets of the graph are gone through again and statements are filtered that contain one of the vertices stored in the HashMap. Those are then replaced by the corresponding group individual according to the lookup HashMap and new edges are created using them. Here the source vertex, the destination vertex or both may be replaced, or stay the same.

At the end, a new graph from the union of new and old edges and vertices is created and returned by the method.

5.3.9. Class TripleGeneratingAbstraction – replace by grouping

The last class implementing the *Transformation* interface is the *TripleGeneratingAbstraction*. Here all edge triplets that contain the *groupingValue* as the predicate (edge *Relation* field *relationship*) and where the source vertex is of a certain type (*Vertex* field *type*) which is supplied to the method as the *replacementObject* are filtered and mapped to edges. Hereby, the subjects represent the vertices that should be replaced, and the destination vertices are the ones that should be used to replace them. Those pairs are then all put into a HashMap to be used as a lookup and mapping table later.

Then, the graph is searched again for all the statements that contain source or destination vertices that should be replaced according to the lookup table by checking whether they are contained in the HashMap. If so, those are replaced by their corresponding individual according to the HashMap and new edges are created accordingly. Here the source vertex, the destination vertex or both may be replaced, or stay the same.

Again, afterwards a new graph from the union of new edges and old and new vertices is created and returned by the method.

5.3.10. Class Utils

The class *Utils* contains different methods to be used in either the generation of graph objects, using Spark in general or when transforming data with the implemented KG-OLAP query operations.

Configs

This method loads the properties from the *config.properties* file and returns them.

sparkConf

This method creates a *sparkConf* (Spark Configuration) from the properties loaded by the configs method.

JavaSparkContex

This method creates a *JavaSparkContext* from the *sparkConf* created by the corresponding Spark method.

getIDOfObject

This method creates a UUID which is a hash of a vertex value. It is required since GraphX vertices must have a unique identifier additionally to the actual vertex value.

Therefore, a vertex basically is a tuple consisting of the value (the objects or subject IRI or literal value that is contained in the base dataset) and then the MD5 hash of this value as the identifier to be able to create vertices for subjects and objects of the following form: *(MD5(value), value)*

5.3.11. Properties Files

The Java program of the prototype currently also includes two property files that are used throughout the code: *config.properties* (graphx-kg-olap\src\main\resources\config.properties) and *params.properies* (graphx-kg-olap\src\main\resources\params.properties).

Both files contain hard-coded information that would have to be adapted before building the jar files and executing them by the user if changes are desired. Further work would include making it possible for the user to change the values that are contained in both of those files in an actual graphical interface or pass certain values as arguments when calling the jar-files. This is, however, not implemented in the scope of this prototype but would be possible due to the architecture of the code.

The *config.properties* file contains information about different performance related configurations and settings for the *SparkContext* that is used when executing any type of Spark task. They were already described in Chapter *2.6.6* about *Performance*. Those settings were kept the same for the whole performance experiments described in Chapter *6: Performance Experiments.*

For the purpose of the performance experiments, all the parameters that are passed especially for the transformation methods (e.g.: *groupingProperty*, *groupingPredicate*, etc.) are also hard-coded in the properties file (*params.properies*) and the ones that are needed for each transformation are loaded at the beginning in the corresponding *transform* method. This would have to be adapted in future work to make it possible for the user to choose their own properties more dynamically. With the current implementation as already mentioned, they would have, however, the possibility to change the properties file before building and executing the jar file. This makes the program usable for other use cases apart from ATM as well.

5.4. Use of Program

The program is written in Java and set up as a Maven project. In order to build the *GraphGenerator* jar-file and also the jar-file for the transformations, the following command has to be executed:

```
mvn clean package
```

The *GraphGenerator* then can then be used in the following way:

```
usage: graphx-kgolap-graphgenerator
 -d,--destinationPath <arg>  the destination path for the output files
 -f,--fileName <arg>     the name of the source nq-file
 -h,--help
 -s,--sourcePath <arg>    the source path to the nq-file
```

For the *GraphGenerator* to work, a destination path has to be added to specify where the output object files should be stored. The *fileName* determines the name of the source N-Quad file and the *sourcePath* specifies where this file is to be found. The following shows an example usage of the *graphGenerator*-jar-file:

Example:

```
java -jar target\graphx-kg-olap-1.0-graphGenerator.jar -d src\main\data\ -s
src\main\data\bd108186-5adb-4f00-b5fe-b24a8993560b.nq
```

All transformations can be executed in the following way

```
usage: graphx-kgolap-transformations
 -d,--destinationPath <arg>  the destination path for the output files
 -e,--edgesFolder <arg>    the folder containing the edges as spark
              object files
 -h,--help          print this message
 -t,--transformation <arg>  the kind of transformation that should be
              performed (reification, pivot, vga – value
              generating abstraction, tga – triple
              generating abstraction, iga – individual
              generating abstraction
 -v,--verticesFolder <arg>  the folder containing the vertices as spark
              object files
```

For the transformation jar-file to work, a destination path indicating where the output files should be stored has to be supplied. Then, the path to the folder where the edges (from the *GraphGenerator* output) files can be found as well as the vertices folder where the vertices are located need to be stated. The *transformation* option determines which transformation should be performed on the base graph. The user can decide between *reification, pivot, vga (value-generating abstraction), iga (individual-generating abstraction) and tga (triple-generating abstraction)*.

The following shows an example usage of the transformations.jar-file where *reification* is performed on data located in *scr\main\data* with an output destination and two folders that will be called *result_vertices* and *result_edges.*

Example:

```
Java-jar target\graphx-kg-olap-1.0-transformations.jar
-e src\main\data\edges -v src\main\data\vertices -t reification
-d src\main\data\result_
```

```
java -jar target\graphx-kg-olap-1.0-transformations.jar
-e src\main\data\edges -v src\main\data\vertices -t iga
-d src\main\data\result2_
```

6. Performance Experiments

To be able to determine the applicability of the implemented KG-OLAP query operations with Apache Spark, several experiments were conducted. Hereby the runtime for executing the query operations on a provided dataset was measured and analysed.

Prior to the actual experiments, the data set supplied in *TriG* format was first transformed into an N-Quad file which is a serialized form of RDF data. Using the implemented *GraphGenerator*, the resulting N-Quad file was then further transformed into a Spark GraphX graph object. All edges and vertices of the graph were stored as objects file which are a Java serialisation form for data within RDDs. Spark hereby transforms objects contained in an RDD in a way that makes it possible to easily store and load them again using Spark as described in (RDD Programming Guide, 2022).

From then on, the experiments were conducted whereby the duration of the query operations was measured. This measuring included the following parts of the processing pipeline:

- the duration of loading the created base object files into a GraphX graph object
- the duration of transforming the GraphX graph by applying one KG-OLAP query operation
- the duration of writing the transformed graph object back to storage in the form of object files (here one folder for edges and one folder for vertices is created)

6.1. Experiment Setup and Data

The used environment for the performance experiments was a virtual CentOS 6.8 machine with 128 gigabytes main memory. The machine used four cores of an Intel Xeon CPU E5-2640 v4 machine with 2.4 GHz as also used for the experiments conducted in Schuetz et al. (2021). The query operations were run on the Java Virtual Machine granting 80GB of heap space.

With the aim of measuring runtime, Spark event logging was enabled, and the duration of the query operations was calculated using the first and last timestamp of the generated log-files for all runs of each type of operation. The tests were run five times per query operation. Then, average and median run time was calculated and used to illustrate the achievable performance of the proposed implementation.

The data chosen for the experiments was a KG-OLAP dataset containing knowledge regarding air traffic management also used by Schuetz et al. (2021). The data is concerned with information about airports, runways, different warnings and events like closure of runways or taxiways. Included is also information about the aircraft themselves like their wingspan or information about runways like contamination of different types and severity. The KG-OLAP cube in this case therefore contains knowledge of different granularities with four dimensions (aircraft, location, date and importance). The contexts are hereby ordered into five different levels including the additionally root context which contains general knowledge relevant for all contexts.

For the experiments, one of the largest benchmark datasets from the datasets provided by Schuetz et al. (2021) was used. This dataset was chosen in order to determine the applicability of the Spark program to operate on large data sets and still perform in an acceptably fast way.

The data used within the experiments then consisted of 33 916 567 N-Quads (about 7 gigabytes file size for the N-Quad file). After being transformed into a GraphX graph object with the described structure of this paper, there were 20 498 972 edges and 14 992 159 vertices in the dataset the query operations were performed on. Hereby all contexts (cells of the KG-OLAP cube) within the dataset were used.

6.2. Experiment Results

Table 7 shows the runtimes for the different KG-OLAP query operations that were implemented within this thesis' described prototype. Additionally, the runtime was measured for only reading and writing the GraphX graph object from and to storage without transforming the data in between. This was done to generate an understanding about the duration that in- and output takes up from the total runtime of a transformation.

Table 7: Experiment results

Transformation	No	Duration	Mean	Median	SD (seconds)	SE (seconds)
Pivot	1	05:48	05:35	05:34	7.32	3.27
	2	05:26				
	3	05:34				
	4	05:34				
	5	05:31				
Reification	1	05:23	05:27	05:29	5.21	2.33
	2	05:31				
	3	05:29				
	4	05:19				
	5	05:33				
Individual-generating abstraction	1	05:20	05:27	05:24	6.51	2.91
	2	05:37				
	3	05:24				
	4	05:24				
	5	05:30				
Triple-generating abstraction	1	05:15	05:14	05:14	3.41	1.52
	2	05:10				
	3	05:10				
	4	05:19				
	5	05:14				
Value-generating abstraction (SUM)	1	04:59	04:58	05:14	3.74	1.67
	2	05:05				
	3	04:54				
	4	04:56				
	5	04:58				
Input/Output only	1	02:00	01:59	01:59	0.77	0.35
	2	01:58				
	3	01:59				
	4	02:00				
	5	01:59				

As it can be seen from the results in Table 7, all operations included in the implementation described in this thesis were executed five times on the same dataset with the same Spark settings and on the same environment. Then, the start and the end timestamps were extracted from the log files for each run to then calculate the duration between them. From those five runs for each operation, the average, median, standard deviation, and standard error were calculated which can be seen in the table.

A further experiment was done on simply loading and reading the object files into a GraphX graph and then – without performing any transformations – writing them back to new folders. The reason for including this test was to show how much time it approximately takes for Spark within this thesis' described setting to read and write the data. This way it can also be estimated how long the operations themselves might have really taken excluding the read and write times. It must be noted, however, that those results are just to provide a rough idea for the read and write times as well as the isolated query operation runtimes, since with the way Spark handles tasks, it cannot be determined exactly how long each part takes individually. It nevertheless serves as a good estimate of performance.

Generally, the resulting average and median runtimes show that the operations were finished in between five and six minutes. Those run times are considered acceptable for the amount of data that was processed in the KG-OLAP setting described within this thesis. The results obtained in the experiments show the general suitability of Apache Spark and the proposed implementation approach for the data structure of KG-OLAP. Further optimizations could possibly be achieved by experimenting with different Spark configurations, partitioning strategies for the data or even by adding more nodes to a computing cluster. However, those experiments are out of the scope of this thesis and would be part of future work.

When comparing the experiment results of this implementation with the performance results obtained with the SPARQL-based implementation by Schuetz et al. (2021), it has to be noted that a comparison between the two implementations cannot be made fairly. This is because there are slight differences in the setup and also the scope of the implementation and the steps included in the runtime measurement. In the prototype described within this thesis, for example, the runtime was calculated including input and output operations which was not done in the paper by Schuetz et al. (2021). Furthermore, the SPARQL-based implementation only constructs delta tables at first when applying query operations. Those tables represent the RDF statements that would need to be added or deleted from the original dataset first, in order to actually complete the operation to then obtain the final, transformed graph. Those insertions and deletions were not considered since they are not specific for the KG-OLAP concept (Schuetz et al., 2021). In the implementation described in this thesis, on the other hand, the result of the operations is always a full graph with all statements in place since Spark allows to do so easily. Therefore, a fair comparison cannot directly be made.

Nevertheless, still, the generated resulting run times of this thesis' experiments can be considered to be an indicator of good performance of the Spark code when using large datasets of KG-OLAP.

7. Conclusion

This thesis was aimed at constructing a prototype implementing the concept of KG-OLAP query operations with the help of a framework that is suitable for handling large amounts of data. Apache Spark was chosen as the preferred processing framework since it was developed to provide speed and scalability for handling big quantities of data. Therefore, Apache Spark's in-memory parallel processing capabilities were leveraged in the implementation of the KG-OLAP query operations which was shown in an experiment of the implemented operations.

First, key concepts concerning the implementation described in this thesis and the KG-OLAP framework itself were explained and illustrated to create a basis for the technologies used in the prototype. Then, current state of the art in literature regarding concepts and dealing with RDF data, big data frameworks and OLAP cube processing were analysed to argue the relevance of this thesis' work. Based on the findings, an appropriate data model including mapping from RDF data to the used Apache Spark GraphX graph representation was developed, as well as a strategy to implement graph operations needed for KG-OLAP. Hereby, performance was the main focus trying to reap the benefits of different characteristics and features of Spark. Lastly, several experiments were conducted using the implemented query operations of this thesis' described prototype and performing them on a provided KG-OLAP dataset. Hereby, a benchmark dataset supplied by Schuetz et al. (2021) was chosen for the experimental setup, previously also used in the evaluation of the SPARQL-based KG-OLAP implementation. The dataset was chosen in order to foster a better understanding of possible performance advantages when using Spark instead of SPARQL.

The successfully implemented prototype and analyses described within this thesis answer the research question on whether Apache Spark (GraphX) can be used to process RDF data and perform KG-OLAP query operations. Since Spark and RDDs are able to work at a low level with any kind of data through different methods provided by the libraries, there was no issue in implementing the desired operations resulting in aggregated and transformed graph data in the air traffic management setting where the resulting graphs can be used to provide a summarized view on data about aircraft, runways and so on. This aggregated data produced may then again be used by Spark or other systems for even further analysis. The prototype was implemented in a way that also makes it possible to be used for different domains other than ATM if the base data structure of KG-OLAP is adhered to in the source data.

The second research question regarding performance when using the stated KG-OLAP operations on the data used in this prototype can also be answered when looking at the experiment results. Since the implementation was tested on an RDF dataset containing 34 million of quadruples available from the KG-OLAP SPARQL implementation, it can be said that for such sizes of data, the prototype performs well, leveraging the benefits of Spark in-memory parallel processing capabilities. This can be said considering the used technical environment and data structure. For different setups or datasets, more experiments would be necessary. Furthermore, the comparison with the SPARQL-based implementation can only be done fairly if there were more experiments with a completely identical setup.

All in all, the work described within this thesis shows that Apache Spark is a promising approach for RDF data processing within the KG-OLAP framework. The work can therefore be used as a basis for further implementations with Apache Spark (GraphX) or other big data processing frameworks for the KG-OLAP concept.

As stated throughout the thesis, the prototype developed has limitations and only looked at a defined scope out of the whole KG-OLAP concept. Regarding additional operations like merge and slice-and-dice as well as RDF reasoning and knowledge propagation, there would have to be further development of the prototype implementing those as well using Apache Spark and GraphX. In general, it should be possible to also realize those additional components of the concepts with the help of the established data model and data processing pipelines as they all follow a similar pattern.

Furthermore, experiments with even larger amounts of data would have to be done in order to possibly find a maximum capacity at which Apache Spark is still able to handle the KG-OLAP operations or still performs better in comparison to other implementations using different technologies. Here especially all Spark configurations and optimization techniques should be used and selected for the tasks involved. For example, configurations for parallelization, partitioning strategies and possibly using a cluster computing approach could lead to further performance gains, especially when experimenting with larger datasets and doing more complex analysis or aggregations.

Another variation of the implementations and experiments would be to build similar processing pipelines and query operations with the help of Spark GraphFrames instead of the GraphX framework. It could then be found out whether a tabular representation of the data in Spark's DataFrames instead of RDDs would increase performance or bring other additional benefits. For this implementation there would also be the need to rethink the data model and if it can still be applied when using GraphFrames to represent the RDF knowledge graphs.

Since GraphX was also especially designed for graph-specific algorithms like PageRank, there would also be the possibility to implement further transformations and query operations even beyond the KG-OLAP operations. Additional analysis could be interesting in the ATM field but also other use cases that rely on a multidimensional model. Using graph-specific algorithms then would leverage the advantages of using GraphX on top of Spark even more and open up more opportunities to analyse big amounts of RDF data with adequate performance.

8. References

Agathangelos, G., Troullinou, H. Kondylakis, K. Stefanidis, & D. Plexousakis. (2018). RDF Query Answering Using Apache Spark: Review and Assessment. 2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW), 54–59. https://doi.org/10.1109/ICDEW.2018.00016

Ali, W., Saleem, M., Yao, B., Hogan, A., & Ngonga Ngomo, A.-C. (2020). *Storage, Indexing, Query Processing, and Benchmarking in Centralized and Distributed RDF Engines: A Survey* [Preprint]. MATHEMATICS & COMPUTER SCIENCE. https://doi.org/10.20944/preprints202005.0360.v3

Ammar, K., & Ozsu, T. (2018). *Experimental Analysis of Distributed Graph Systems*. https://doi.org/10.48550/ARXIV.1806.08082

Angles, R. (2018). The Property Graph Database Model. *Alberto Mendelzon Workshop on Foundations of Data Management*.

Apache Jena. 2022. URL: https://jena.apache.org/ (visited on 2022-11-20)

Apache Spark FAQ. 2023. URL: https://spark.apache.org/faq.html (visited on 2023-01-22)

Apache Spark History. 2023. URL: https://spark.apache.org/history.html (visited on 2023-01-21)

Azirani, E. A., Goasdoué, F., Manolescu, I., & Roatis, A. (2015). Efficient OLAP operations for RDF analytics. *2015 31st IEEE International Conference on Data Engineering Workshops*, 71–76.

Bahrami, R. A., Gulati, J., & Abulaish, M. (2017). Efficient processing of SPARQL queries over GraphFrames. *Proceedings of the International Conference on Web Intelligence*, 678–685. https://doi.org/10.1145/3106426.3106534

Banane, Mouad & Belangour, Abdessamad. (2019). RDFSpark: a new solution for querying massive RDF data using spark. International Journal of Engineering and Technology. 288-294. 10.14419/ijet.v8i3.23157.

Bhosale, H. S., & Gadekar, D. P. (2014). A review paper on big data and hadoop. *International Journal of Scientific and Research Publications*, *4*(10), 1-7.

Casado, R., & Younas, M. (2015). Emerging trends and technologies in big data processing: INCOS 2013 SI ‘ADVANCES ON CLOUD SERVICES AND CLOUD COMPUTING’. *Concurrency and Computation: Practice and Experience*, *27*(8), 2078–2091. https://doi.org/10.1002/cpe.3398

Chaudhuri, S., & Dayal, U. (1997). An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, *26*(1), 65–74. https://doi.org/10.1145/248603.248616

Chen, C., Yan, X., Zhu, F., Han, J., & Yu, P. S. (2008). Graph OLAP: Towards Online Analytical Processing on Graphs. *2008 Eighth IEEE International Conference on Data Mining*, 103–112. https://doi.org/10.1109/ICDM.2008.30

Chen, X., Chen, H., Zhang, N., & Zhang, S. (2015). SparkRDF: Elastic Discreted RDF Graph Processing Engine With Distributed Memory. *2015 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, *1*, 292–300. https://doi.org/10.1109/WI-IAT.2015.186

Colazzo, D., Goasdoué, F., Manolescu, I., & Roatiş, A. (2014). RDF analytics: Lenses over semantic graphs. *Proceedings of the 23rd International Conference on World Wide Web - WWW ’14*, 467–478. https://doi.org/10.1145/2566486.2567982

Cox, M., & Ellsworth, D. (1997). Application-controlled demand paging for out-of-core visualization. *Proceedings. Visualization ’97 (Cat. No. 97CB36155)*, 235-244,. https://doi.org/10.1109/VISUAL.1997.663888

Curé, O., Naacke, H., Baazizi, M., & Amann, B. (2015). HAQWA: a Hash-based and Query Workload Aware Distributed RDF Store. *International Semantic Web Conference*.

Denis, B., Ghrab, A., & Skhiri, S. (2013). A distributed approach for graph-oriented multidimensional analysis. *2013 IEEE International Conference on Big Data*, 9–16. https://doi.org/10.1109/BigData.2013.6691777

Elser, B., & Montresor, A. (2013). An evaluation study of BigData frameworks for graph processing. *2013 IEEE International Conference on Big Data*, 60–67. https://doi.org/10.1109/BigData.2013.6691555

Farhan Husain, M., Doshi, P., Khan, L., & Thuraisingham, B. (2009). Storage and Retrieval of Large RDF Graph Using Hadoop and MapReduce. In M. G. Jaatun, G. Zhao, & C. Rong (Hrsg.), *Cloud Computing* (Bd. 5931, S. 680–686). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-10665-1_72

Ghrab, A., Romero, O., Skhiri, S., Vaisman, A., & Zimányi, E. (2015). A Framework for Building OLAP Cubes on Graphs. In M. Tadeusz, P. Valduriez, & L. Bellatreche (Hrsg.), *Advances in Databases and Information Systems* (Bd. 9282, S. 92–105). Springer International Publishing. https://doi.org/10.1007/978-3-319-23135-8_7

Ghrab, A., Romero, O., Skhiri, S., & Zimányi, E. (2021). TopoGraph: An End-To-End Framework to Build and Analyze Graph Cubes. *Information Systems Frontiers*, *23*(1), 203–226. https://doi.org/10.1007/s10796-020-10000-z

G'omez, L., Kuijpers, B., & Vaisman, A. A. (2020). Online analytical processsing on graph data. *Intell. Data Anal.*, *24*, 515–541.

Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., & Stoica, I. (2014). Graphx: Graph processing in a distributed dataflow framework. *OSDI*, 599–613.

GraphFrames Overview. 2022. URL: https://graphframes.github.io/graphframes/docs/_site/index.html (visited on 2022-07-22)

Graux, D., Jachiet, L., Genevès, P., & Layaïda, N. (2016). SPARQLGX: Efficient Distributed Evaluation of SPARQL with Apache Spark. In P. Groth, E. Simperl, A. Gray, M. Sabou, M. Krötzsch, F. Lecue, F. Flöck, & Y. Gil (Hrsg.), *The Semantic Web – ISWC 2016* (S. 80–87). Springer International Publishing.

Haunschmied, D. (2022). *A Cloud-Native Data Lakehouse Architecture for Big Knowledge Graph OLAP* (Master's thesis). Universität Linz, Institut für Wirtschaftsinformatik - Data & Knowledge Engineering, Linz, Österreich.

Hazarika, A. V., Ram, G. J. S. R., & Jain, E. (2017). Performance comparision of Hadoop and spark engine. *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, 671–674. https://doi.org/10.1109/I-SMAC.2017.8058263

Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., Melo, G. D., Gutierrez, C., Kirrane, S., Gayo, J. E. L., Navigli, R., Neumaier, S., & others. (2021). Knowledge graphs. *ACM Computing Surveys (CSUR)*, *54*(4), 1–37.

Hong, S., Choi, W., & Jeong, W.-K. (2017). GPU in-Memory Processing Using Spark for Iterative Computation. *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 31–41. https://doi.org/10.1109/CCGRID.2017.41

IBM Cloud Education. (27 May 2016). Hadoop vs. Spark: What's the Difference? URL: https://www.ibm.com/cloud/blog/hadoop-vs-spark (vishadited on 2023-01-21)

Ibragimov, D., Hose, K., Pedersen, T. B., & Zimányi, E. (2015). Towards Exploratory OLAP Over Linked Open Data – A Case Study. In M. Castellanos, U. Dayal, T. B. Pedersen, & N. Tatbul (Hrsg.), *Enabling Real-Time Business Intelligence* (Bd. 206, S. 114–132). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-46839-5_8

Jacobs, A. (2009). The pathologies of big data. *Communications of the ACM*, *52*(8), 36–44. https://doi.org/10.1145/1536616.1536632

Junghanns, M., Petermann, A., Gómez, K., & Rahm, E. (2015). *GRADOOP: Scalable Graph Data Management and Analytics with Hadoop*.

Kang, S., Lee, S., & Kim, J. (2020). Distributed graph cube generation using Spark framework. *The Journal of Supercomputing*, *76*(10), 8118–8139. https://doi.org/10.1007/s11227-019-02746-4

Karau, H., & Warren, R. (2017). *High performance Spark: Best practices for scaling and optimizing Apache Spark* (First edition : June 2017). O'Reilly Media, Inc.

Kassaie, B. (2017). SPARQL over GraphX. *ArXiv, abs/1701.03091*.

Krötzsch, M., & Weikum, G. (2016). Editorial for special section on knowledge graphs, Journal of Web Semantics 37-38 (2016), 53–54. doi:10.1016/j.websem.2016.04.002

Naacke, H., Amann, B., & Curé, O. (2017). SPARQL Graph Pattern Processing with Apache Spark. Proceedings of the Fifth International Workshop on Graph Data-Management Experiences & Systems, 1–7. https://doi.org/10.1145/3078447.3078448

Nguyen, N., Khan, M. M. H., Albayram, Y., & Wang, K. (2017). Understanding the Influence of Configuration Settings: An Execution Model-Driven Framework for Apache Spark Platform. *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, 802–807. https://doi.org/10.1109/CLOUD.2017.119

Gomez-Perez, J. M., Pan, J. Z., Vetere, G., & Wu, H. (2017). Enterprise Knowledge Graph: An Introduction. In J. Z. Pan, G. Vetere, J. M. Gomez-Perez, & H. Wu (Hrsg.), Exploiting Linked Data and Knowledge Graphs in Large Organisations (S. 1–14). Springer International Publishing. https://doi.org/10.1007/978-3-319-45654-6_1

Ragab, M., Eyvazov, S., Tommasini, R., & Sakr, S. (o. J.). *Systematic Performance Analysis of Distributed SPARQL Query Answering Using Spark-SQL*.

RDD Programming Guide. 2022. URL: https://spark.apache.org/docs/latest/rdd-programming-guide.html (visited on 2022-07-22)

Schätzle, A., Przyjaciel-Zablocki, M., Berberich, T., & Lausen, G. (2016). S2X: Graph-Parallel Querying of RDF with GraphX. In F. Wang, G. Luo, C. Weng, A. Khan, P. Mitra, & C. Yu (Hrsg.), *Biomedical Data Management and Graph Online Querying* (Bd. 9579, S. 155–168). Springer International Publishing. https://doi.org/10.1007/978-3-319-41576-5_12

Schätzle, A., Przyjaciel-Zablocki, M., Skilevic, S., & Lausen, G. (2016). S2RDF: RDF querying with SPARQL on spark. *Proceedings of the VLDB Endowment*, *9*(10), 804–815. https://doi.org/10.14778/2977797.2977806

Schuetz, C. G., Bozzato, L., Neumayr, B., Schrefl, M., & Serafini, L. (2021). Knowledge Graph OLAP: A multidimensional model and query operations for contextualized knowledge graphs. *Semantic Web*, *12*(4), 649–683. https://doi.org/10.3233/SW-200419

Sejdiu, G. (2019). *Sparklify*. 0 Bytes. https://doi.org/10.6084/M9.FIGSHARE.7963193.V1

Serafini, L., & Homola, M. (2012). Contextualized knowledge repositories for the Semantic Web. *Journal of Web Semantics*, *12–13*, 64–87. https://doi.org/10.1016/j.websem.2011.12.003

Tomaszuk, D. (2016). RDF data in property graph model. *Research Conference on Metadata and Semantics Research*, 104–115.

Tuning Spark. 2022. URL: https://spark.apache.org/docs/latest/tuning.html (visited on 2022-07-22)

Vaisman, A., & Zimányi, E. (2014). *Data Warehouse Systems: Design and Implementation* (1st ed. 2014). Springer Berlin Heidelberg: Imprint: Springer. https://doi.org/10.1007/978-3-642-54655-6

Wang, Z., Fan, Q., Wang, H., Tan, K.-L., Agrawal, D., & El Abbadi, A. (2014). Pagrol: Parallel graph olap over large-scale attributed graphs. *2014 IEEE 30th International Conference on Data Engineering*, 496–507. https://doi.org/10.1109/ICDE.2014.6816676

W3C (World Wide Web Consortium) (2013) SPARQL 1.1 Overview, W3C Recommendation 21 March 2013, URL: https://www.w3.org/TR/sparql11-overview/ (visited on 2023-01-20)

Yang, H., Liu, X., Chen, S., Lei, Z., Du, H., & Zhu, C. (2016). Improving Spark performance with MPTE in heterogeneous environments. *2016 International Conference on Audio, Language and Image Processing (ICALIP)*, 28–33. https://doi.org/10.1109/ICALIP.2016.7846627

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M., Shenker, S., & Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (S. 2).

Zhang, H., Liu, Z., & Wang, L. (2018). Tuning Performance of Spark Programs. *2018 IEEE International Conference on Cloud Engineering (IC2E)*, 282–285. https://doi.org/10.1109/IC2E.2018.00057

Zhao, P., Li, X., Xin, D., & Han, J. (2011). Graph cube: On warehousing and OLAP multidimensional networks. *Proceedings of the 2011 International Conference on Management of Data - SIGMOD '11*, 853. https://doi.org/10.1145/1989323.1989413

Zheng, J., Ma, Q., & Zhou, W. (2016). Performance comparison of full-batch BP and mini-batch BP algorithm on Spark framework. *2016 8th International Conference on Wireless Communications & Signal Processing (WCSP)*, 1–5. https://doi.org/10.1109/WCSP.2016.7752505