University of Tartu

Faculty of Science and Technology

Institute of Technology

Fred Peter Boldin

C-eraph: Towards continuous OpenCypher

Bachelors Thesis (12 ECTS)
Computer Engineering Curriculum

Supervisor:

Riccardo Tommasini Assistant Professor of Data Management

Tartu 2020

Abstract / Resümee

C-eraph: Towards continuous OpenCypher

With the surge of data caused by the creation of internet, internet of things, growth of the computing power and storage, there is a need to process data in real-time to benefit from it. We live in a world that is connected, all of the information around us is in relationship to one another. We can look at this information as a graph - nodes of objects connected with relationships. To take advantage of this data model of nodes and relationships a graph database can be used. Often the data that is collected needs to be used immediately and therefore streaming data processing can be used to use the dynamic data that is generated in real-time. In this thesis we have tried to find out if it is possible to create a system that is able to query streams of property graphs continuously. In this paper we try to provide the required background knowledge of the work and how the original system was changed to make it work with property graph data.

CERCS: P170 Computer science, numerical analysis, systems, control

Keywords: data, streaming, graphs, data processing, semantic web

C-eraph: Pidevaid andmevooge töötleva openCypher'i suunas

Tänu andmekoguse puhangule, mille on põhjustanud interneti teke, nutistu, arvutusvõimsuse ning andmekandjate mahu suurenemine, on tekkinud vajadus andmeid töödelda reaalajas. Me elame ühendatud maailmas, kogu meid ümbritsev informatsioon on omavahel seotud. Maailma võib vaadelda kui graafi, kus kogu informatsioon koosneb tippudest, mis on omavahel seotud servadega. Selleks, et graafi andmemudelit otstarbekalt rakendada saame kasutada graafi andmebaasi. Tihtipeale on tarvis tekkivaid andmeid koheselt kasutada. Seda aitab saavutada andmevoogude töötlemine, mis võimaldab dünaamiliste andmete kasutamist reaalajas. Käesoleva töö eesmärgiks on teha kindlaks, kas on võimalik ehitada süsteem, mis suudaks teha pidevaid päringuid andmevoogudele, mis hoiavad endas andmeid *property graph* mudeli kujul. Töö sisaldab endas vajalikku taustainfot selle kohta, kuidas töötas baassüsteem ning millised olid vajaminevad muudatused, mida tehti, selleks, et täide viia *property graph* mudeli lisamine sinna süsteemi.

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Märksõnad: andmed, andmevood, graafid, andmetöötlus, semantiline veeb

List of contents

List of figures

List of abbreviations

ASCII - American Standard Code for Information Interchange

RDF - Resource Description Framework

QL - Query Language

DBMS - Database Management System

RDBMS - Relational Database Management System

GDBMS - Graph Database Management System

RDBM - Relational Database Model

GDBM - Graph Database Model

W3C - World Wide Web Consortium

API - Application Programming Interface

1 Introduction

"Data is the new oil.", this quote by the British mathematician and data scientist Clive Humby [1] undeniably rings true in today's world. The abundance of data can be attributed to the birth of the Internet. The ability to transfer information over the network between electronic devices has permanently changed the way we look at information. The amount of data created is enormous, when looking at the info-graphic in Figure 1.1 in just one minute major companies produce complex data by the millions [2].

With the amount of data growing exponentially each year [3] the opportunity to take advantage of this information is remarkable. Indeed, collecting and analysing the data helps us make informed decisions with the ultimate goal of trying to predict the future to some extent or at least to react to changes fast enough [4].

1.1 Motivation

In these regards, data systems play a critical role for storing, managing and analysing data. To be able to do this we also need to store the data in some form and thanks to the high capacity of storage in modern computers and the variety of Database Management Systems (DBMS) we can store data in different ways.

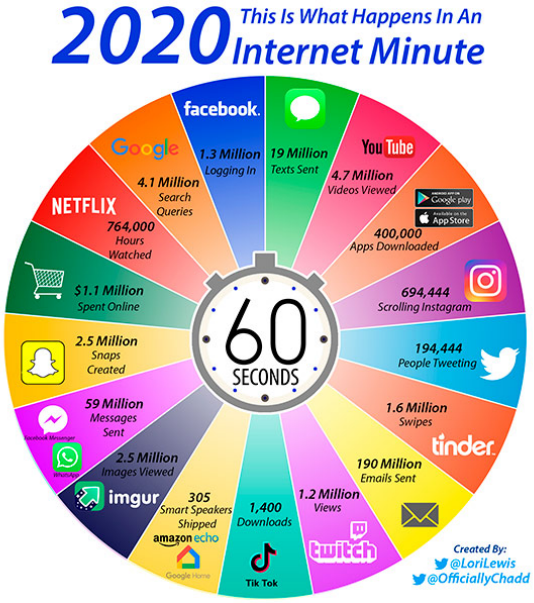

Figure 1.1: Infographic: what happens in an internet minute 2020

We can see from Figure 1.1 that data is very heterogeneous. The variety of information provides

us with an interesting problem, how do we store and manage this data coming from different sources in an integrated way. As of now making sense of data is still heavily reliant on humans. The aim is to create context for the diverse data so that computers can digest it [5].

Another thing that the info-graphic is trying to present is the data velocity representing the speed at which data are consumed and insights are produced. The enormous amount of data is creating a need for communicating it as fast as possible. If we do not take advantage of data in real-time it becomes a bottleneck for getting the maximum value from it. One of the ways that we are tackling the problem of data velocity is stream processing, analysing the data in a continuous flow rather than in batches [6].

Charles W. Bachman is the person to design the Integrated Database System in the early 1960s [7]. Ever since then the development of Data Base Management Systems(DBMS) has never stopped [8]. There are a plethora of different database models, query languages, and storage techniques in the DBMS literature. Undeniably, the most popular and used are the relational DBMS and SQL [9] where the records store as rows in tables. And although relational databases have been used they have drawbacks that, in such a data-intense world, may impact application design. In particular, despite the name, relations are not first-class in Relational Database Management Systems(RDBMS). Connections between data requires complex analytics, they need indexes and running complex queries with many joins operations. This degrades the response time of the databases making it inadequate when the results are needed in real time.

Therefore, as can be seen in Figure 1.1 in an interconnected world Graph Database Management Systems(GDBMS) are getting momentum. Graphs are an abstract data form that consists of both vertices and edges. Graph databases manage the data considering relationships between them as first-class citizens. In this form connections are easier to find and is far easier to analyze. Graph models are flexible and allow extensibility.

Since the graph database is based on graph theory and the relationships are stored as a first-class entities it is very suitable not just for querying single data points but also relationship networks. GDBMS uses index-free adjacency [10] to store the nodes in a way that maintains them as directly connected. This makes querying the data a special case of graph traversal. Moreover, the graph database query languages are more expressive.

There are countless scenarios where the data that we produce and aquire needs to be processed and analysed in real-time. Many modern companies take advantage of this to give them the ability to act upon the data immediately. Great examples of this are organizations such as Spotify, Netflix and Amazon, they all use data streaming to recommend their users new products that might interest them according to the data that they have gathered so far from analysing their choices. There are many other use cases for streaming such as: fraud detection, sensors, business analytics, location data etc. One could argue that any industry that deals with data can benefit from data streaming [11]. However, batch data processing has some major drawbacks, i.e. the time delay between the collection of the data and getting the result. Stream processing, on the other hand, analyses information on-the-fly without requiring to put data at rest.

In order to effectively use graph processing some requirements need to be met. As we can see from Figure 1.1 the amount of data is huge, therefore we need a system with scalable processing to keep up with the data as it grows. We need a powerful query language that gives us the ability to extract intricate information from that data. The system has to also have expressive ontology

in order to describe artifacts with different degrees of structure. To get the most from graph processing the system should utilize continuous semantics to combine it with stream processing so the data can be processed in real-time.

1.1.1 Problem statement

The problems of data velocity and variety are not isolated. To be able to analyse complex data in real-time is one of the biggest obstacles in big data. As the data grows and gets more complex we need to process it in the most efficient way possible. For this we need to build systems that can handle both of these problems of variety and velocity at the same time. Using graphs to manage the varying data and streaming to communicate it quickly can help us pursue solving that problem.

Some existing solutions are tailored for interoperability and use the data model of Resource Description Framework(RDF) along with the SPARQL query language. The problem is that these solutions do not have fine-grained access to information because of the limitations of the SPARQL query language. They also do not have the ability to model complex application domains due to inexpressive ontology of RDF.

To satisfy the requirements of effective graph stream processing we have decided to adapt the existing solutions to use the data model of property graphs and Cypher query language, together they provide a powerful query langauge and a ontologically expressive data storage.

In this thesis we are trying to find out if it is possible to create a system that is able to continuously stream and query property graph data. Additionally we will try to determine if it is possible for this system to work in close to real-time and keep the temporary data that it processes in-memory as opposed to writing it on disk.

2 Background

In the world of graphs and streaming there have been some solutions made that integrate these two parts, but they have still remained separated. For example there is a solution for integrating Apache Kafka with Neo4j [12], but it is just a connection through a plugin between the two and not a unified system that works in-memory. There is no system or protocol built to continuously stream and query the property graph data using a stream processing engine while not writing to disk [13]. In this section I will go over some of the existing solutions that have been developed and are relevant in the area of graphs and stream processing.

2.1 Streaming

A stream is an unbouded sequence of data. Usually the streams are in the format of (d, t) where d is the data and t is the timestamp. Three specific characteristics differentiate streams from other data: unboundedness, data are ordered by time, data are shared in a push model as opposed to pull [14].

2.1.1 Kafka

Data streaming is widely used and there are various solutions for it, one of the most popular streaming platforms is Apache Kafka. Kafka communicates through a TCP-protocol in a way that is simple and highly efficient. It offers three key capabilities:

- Publishing and subscribing to streams of records that are similar to a message queue
- Processing streams of records as they occur
- Storing streams of records in a fault-tolerant way

Kafka is used as a way to build real-time streaming data pipelines and for real-time streaming applications that transform streams of data [15]. Kafka is run on a cluster of multiple servers that is highly scalable. Streams of records are stored in categories that are called topics. The records are contain a key, value and a timestamp. The two main parts of the Kafka that communicate the data through the cluster are producers and consumers [16]. Producers publish the data to chosen topics. The topics are managed by brokers that serve and receive the data. Consumers subscribe to topics, messages in a topic are delivered to consumers. Kafka provides fault tolerance guarantees for the user: topics are replicated across servers, the messages written in a topic partition are ordered and the consumer instances see the records in the order that they are stored in the topic. Kafka's model provides a very concise, reliable and easily managed way to manage streams of records.

2.1.2 CQL

Continuous Query Language(CQL) is a SQL-based query language for registering continuous queries against relations and streams. A *Stream* is a collection of elements (s, t) where s is a tuple belonging to the schema of the *Stream* and t is the timestamp of the element. *Relations* are mappings from each time instant to a finite collection of tuples belonging to the *Relation* schema. *Instantaneous relations* are the collection of tuples in a *relation* that are related to a time instant t [17].

CQL has an operational model that consists of the operators: Stream-to-Relation(S2R), Relation-to-Relation(R2R) and Relation-to-Stream(R2S). These operators define the semantics of a *Query* as a composition of operators and set of input streams and relations. The result of this *Query* is calculated at the time instant t.

The R2R operators handle the time-varying relations. The S2R operators are based on the concept of time windows over a stream. There are three options for dealing with the R2S operator which are:

- **R-Stream** - Used for streaming out all of the elements of an instantaneous relation at a particular moment
- **I-Stream** - Used for streaming out all new entries of an instantaneous relation the previous one
- **D-Stream** - Used for streaming out all deleted entries of an instantaneous relation

The model of CQL is illustrated in Figure 2.1.

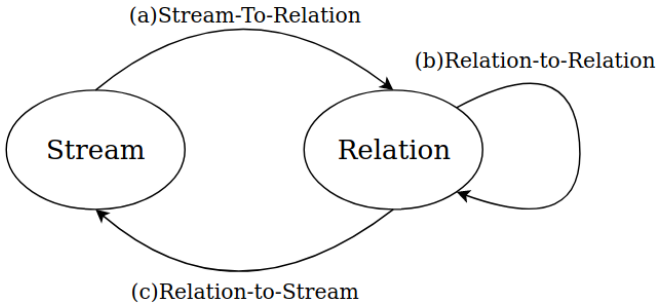

Figure 2.1: Operational model of CQL from

2.1.3 Esper

Streams are unbounded sequences of data. To aggregate large amounts of different information about the relationships of events in real time a tool is needed. Out of this necessity Esper was created, a language, compiler and runtime for processing complex events available to use in Java. The language of Esper is called Event Processing Language and it implements and extends the standard of SQL while enabling rich expressions over events and time [18].

An integral part of Esper is windowing. A window is a time interval that repeats and is used

to separate the data into groups. This enables the limitation of events to be queried and aggregated. For example when creating a simple fire detection mechanism measuring the average temperature of a room combined with the sensory data that shows the percentage of CO_2 in the air we can take the average of that data from the past 15 seconds and analyse it. If the combined information is above the allowed threshold a *FIRE* event can be created which can automatically call the rescue services or activate the emergency water sprinklers [19]. This is just one of many possible use cases for streaming engines such as Esper. In the model of CQL Esper can serve the part of both the Window Operator and Streaming Operator.

Esper event processing language(EPL) that extends SQL, it merges together event stream processing and complex event processing into one language. It includes causality patterns and event windows as first-class citizens. EPL allows using complex matching patterns, filtering and sorting. Thanks to this we can use EPL to analyze series of events with regard to time and derive conclusions from them [20]. Using EPL we can register statements in the runtime using Java objects(JavaBean or plain-old-java-objects) to represent events. A listener class will be called by the runtime when the EPL condition is matched as events arrive [21].

As a simple example of Esper we can look at this EPL query:

```
SELECT rstream avg(value) AS value FROM temperature#time(10 seconds)
```

In this query the *Select* clause specifies the events to retrieve, here it is *avg(value)*. The *rstream* tells the engine to deliver only the remove stream which denotes the result of events that are leaving the data window. Following the *from* clause is the name of the stream which is *temperature*. The listener then recieves only the events of a 10 second time window as specified by *time(10 seconds)*.

Using the Java API we can easily create the EPL statements using the administrative interface *EPAdministrator* like so:

```
EPServiceProvider epService = EPServiceProviderManager.getDefaultProvider()
    ;
EPAdministrator admin = epService.getEPAdministrator();

EPStatement avgStmt = admin.createEPL(
  "Select rstream avg(value) as value from temperature#time(10 seconds)");
```

After that we can subscribe to updates posted by this statement using listeners.

```
UpdateListener myListener = new MyUpdateListener();
avgStmt.addListener(myListener);
```

The EPL statements publish both new and old data to the *UpdateListener*. New data represents new events and old data represents prior event values.

2.2 Graphs

2.2.1 Graph Processing and Neo4j

Assuming a certain familiarity with relational data model, we present GDBMS in comparison with RDBMS.

In Figures 2.3 and 2.2 the Northwind data set is used here to demonstrate the differences between these two data models [22].

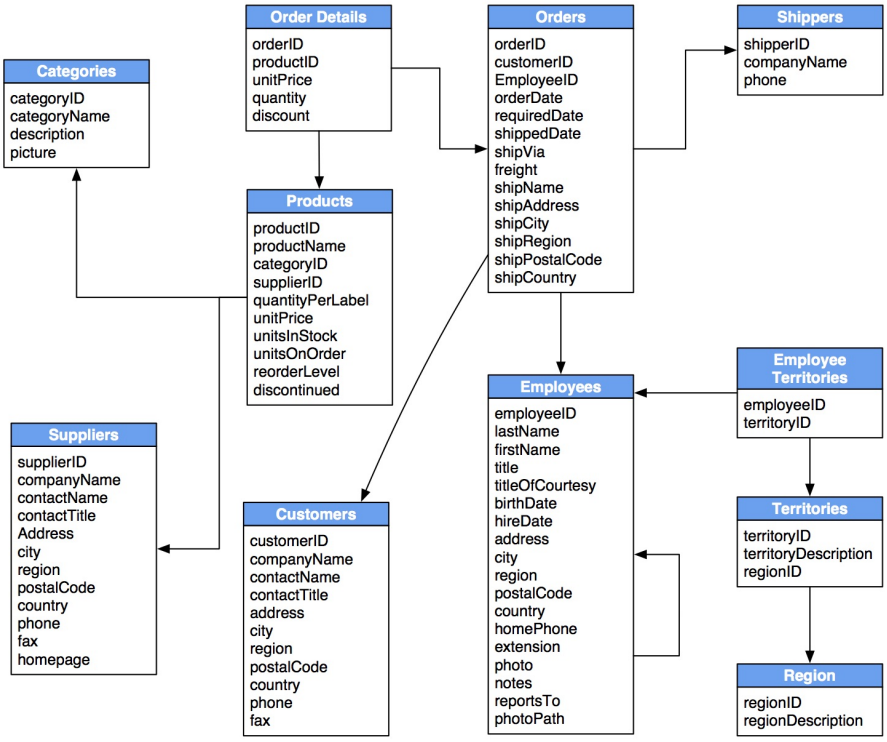

Figure 2.2: Northwind data set represented as a relational model

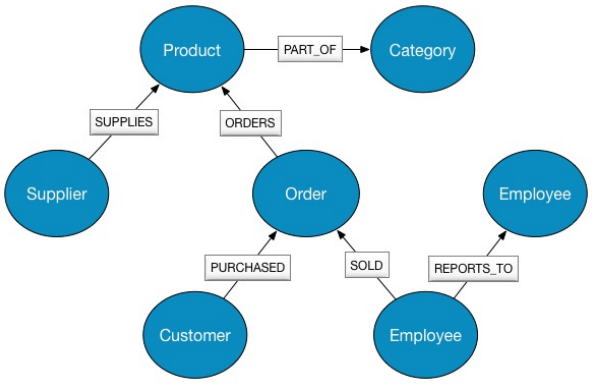

Figure 2.3: Northwind data set represented as a graph model

It is clear to see that a graph model is more natural at representing real life complex relationships and is easier to read than the relational model [23].

2.2.2 Property graph data model & Cypher

The graph model used in Neo4J is the property graph, these are the components which make up a property graph [23]:

- Nodes and relationships
- Nodes contain properties
- Nodes can be labeled with one or more labels
- Relationships are named and directed, and always have a start and end node
- Relationships can also have properties

A visual representation of a property graph would look something like Figure 2.4 [24].

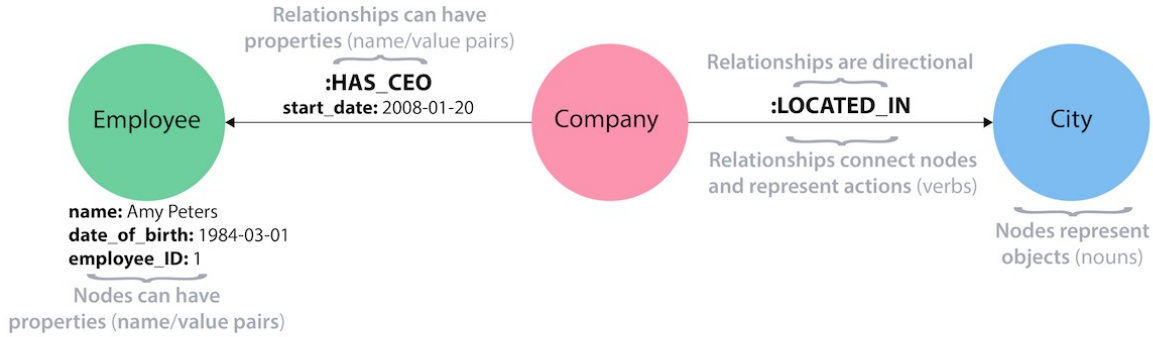

Figure 2.4: A visual representation of property graph and its components

In property graph nodes are entities that can hold any number of attributes, which are called properties. Nodes have different labels such as *Employee* or *Company* to represent different roles in the database.

Relationships are from one node to another, they have a type, name and just like nodes they too can have properties that give them a richer description. Because the entities in Neo4j have unique identifiers there can be many relationships of the same type between two nodes [24]. In order to query information from a Neo4J database it is required to use the Cypher. It is similar to other query languages but is distinct in quite a few ways. It resembles ASCII art and is quite compact and easy to grasp [23].

Here is an example of creating a few person nodes and connecting them with a friend relationship from the Neo4j Cypher manual [25].

```
CREATE (john:Person {name: 'John'})
CREATE (joe:Person {name: 'Joe'})
CREATE (steve:Person {name: 'Steve'})
CREATE (sara:Person {name: 'Sara'})
CREATE (maria:Person {name: 'Maria'})
CREATE (john)-[:FRIEND]->(joe)-[:FRIEND]->(steve)
CREATE (john)-[:FRIEND]->(sara)-[:FRIEND]->(maria)
```

The result of this query is a property graph that can be seen on Figure 2.5.

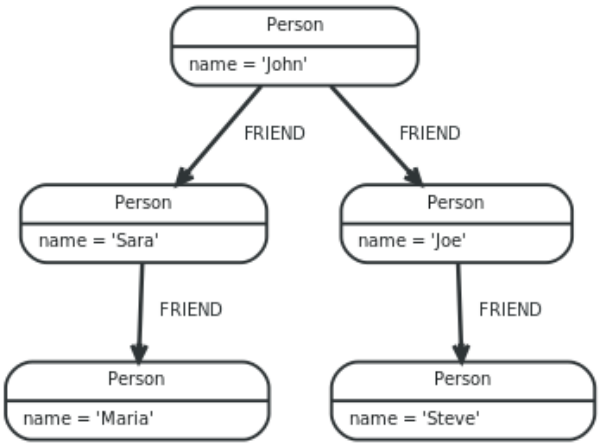

Figure 2.5: Result of the example query that utilizes the CREATE command to make nodes and relationships

As an example, in order to find all Johns non-direct friends, it is possible to use this query.

```
MATCH (john {name: 'John'})-[:FRIEND]->()-[:FRIEND]->(fof)
RETURN john.name, fof.name
```

The result of this query would look like this.

john.name & fof.name
"John" & "Maria"
"John" & "Steve"

In Cypher, just like in most query languages, clauses are the main part of the query. Just like in this example the most basic queries can consist of just the MATCH and RETURN clause. Any additional querying aspects can be constructed on top of this for creating more specific and complex queries. To guarantee quick and efficient querying Neo4j allows running major graph algorithms such as Dijkstra, A* and others [23].

When looking at other the available solutions for graph processing there are many out there. The most common ways to model graph data is with RDF or as Property Graphs. Apache Jena Fuseki is a Open Source SPARQL server [26], it serves as a triple store that can be ran in the background by an application located in a server. It can be run as a operating system service, as a server or a Java web application. Fuseki provides the SPARQL protocols for querying and updating and also the SPARQL Graph Store protocol. It offers a transactional persistent storage layer that can be used to provide the protocol engine for other RDF query and storages[27]. Another option for storing triples is OpenLink Virtuoso, is provides database functionality that handles RDF data [28]. Like Fuseki it also supports SPARQL query language. Users can utilize Virtuoso WebDAV repository for uploading the RDF data into the Virtuoso Quad store via the HTTP protocol [29]. However the most popular way to represent graph data is in the form of Property Graphs. Instead of the RDF model that uses URI's that provide no internal structure for the data, the Property Graph that uses Nodes and Relationships that do provide internal structure in the form of properties [30]. This makes the property graph data more rich and therefore more useful for many cases. There are different query languages to deal with Property Graphs for

example PGQL that is built on top of SQL and is used for graph pattern matching [31], but the most widely used on is Cypher because it is used alongside the most popular Graph Database Managment System Neo4j [32].

2.2.3 RDF and SPARQL

In this chapter we will go over the RDF and the SPARQL query language.

RDF that is a graph data model that fosters interoperability on the Web. It enables efficient integration of data from multiple sources in a way that separates the data from the schema. Because the RDF is schema agnostic, it can be used for data integration. RDF data are represented in as statements $< Subject, Predicate, Object >$ [33].

We can imagine that in a social network a person named Mark has listed some information about him, this information can be viewed as multiple RDF statements as shown in Figure 2.6.

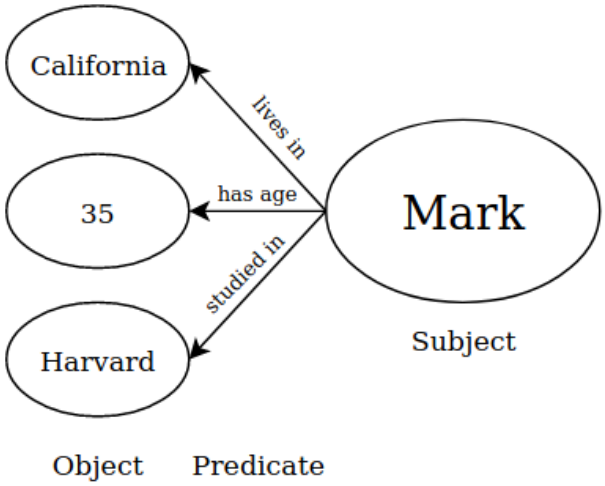

Figure 2.6: An example of RDF data statements

As an example of this we can imagine a graph where a person named Mark adds another person called Robert to his friends list. The result of this graph is illustrated in the Figure 2.7.

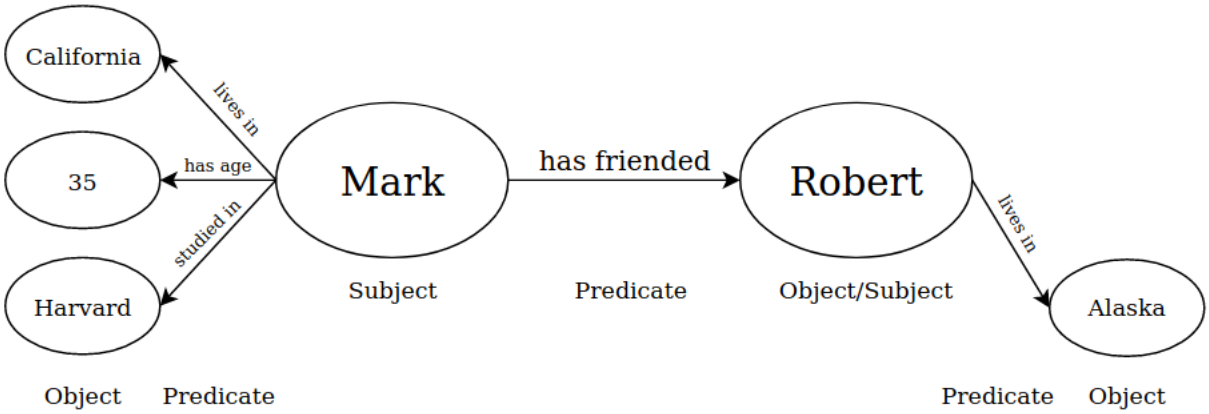

Figure 2.7: *Subjects* can also be *Objects*, this way a graph can be formulated out of RDF statements

SPARQL is a query language and a protocol for RDF. Similarly to Cypher, SPARQL allows the the retrieval and modifying of data in graph databases [34].

An example of a SPARQL query would look something like this:

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/> #foaf is basically a text
    document
SELECT ?name ?email # selects the names and emails
FROM  <http://www.w3.org/People/Berners-Lee/card> # is a graph
WHERE {
     ?person foaf:name ?name ;
             foaf:age ?age .
     FILTER (?age > 18) # uses the filter
}
ORDER BY ?name # sorts the results in an alphabetical order
LIMIT 10 # limits the number of solutions
OFFSET 10 # offsets the presented solutions by 10 from the start
```

This query searches for persons from a graph, persons that pass through the filter are presented as results in the alphabetical order.

2.3 Graph Streams

RSP-QL is a model for defining the semantics of RDF Stream Processing (RSP) system. It is based on a model from the streaming data world called Continuous Query Language, which is a continuous extention of SQL. "CQL is a continuous extension of SQL: its semantics define a formal model with three kinds of operators (S2R, R2R and R2S) that process and transform streams and relations." [35]. In the case of S2R operators, CQL uses time-based and partitioned window operators. R2S operators generate new streams according to the stream-type that is chosen(R-stream, I-Stream, D-stream). The R2R operators take one or more relations and turn it into a new one [36].

In this subsection the integral parts of the RSP-QL model will be explained. In RSP-QL a RDF statement is a pair (d, t) where r is an RDF statement and t is an instant of time. Let's call S a sequence of timestamped RDF statements that are in a non-decreasing order of time.

$S = ((d_1, t_1), (d_2, t_2), (d_3, t_3), ...)$

A time-based sliding window operator W is a Stream-to-Relation(S2R) operator. It takes a stream S as an input and produces a Time-Varying Graph G_W. The window operator W is defined as a triple (α, β, t^0) where the parameters stand for:

- α - the width of window
- β - the sliding step length
- t^0 - the time instant when W starts operating

The Time-Varying Graph is a function, it takes a time instant as an input and produces the result as a RDF graph which is called an Instantaneous RDF Graph. The window operator W on a RDF Stream S produces a Time-Varying Graph T. For any time instant t where W is defined the window operator outputs an Instantaneous RDF Graph - it is a result of combining all of the RDF Graphs or triples that are in the current window.

An SDS(Streaming Data Set) that is an extension of the SPARQL data set. It is composed of three parts - an (optional) default graph, n amount of named graphs where $(n \geq 0)$ and m named Time-Varying Graphs where $(m \geq 0)$.

The result of the RSP-QL query is a multi-set of Solution Mappings for each evaluation. Relation-to-Stream(R2S) operators are necessary for the transformation of the instantaneous multi-set to a stream. The R2S operators that RSP-QL uses are these: **IStream** - streams out the difference between the answer of the current evaluation and the previous one, **DStream** - outputs the part of the answers from the previous result that did not appear in the current one and **RStream** - streams out the whole output every time an evaluation occurs [35].

An example of a RSP-QL query counting the number of blue colors in a time window would look something like this:

```
SELECT (COUNT(?b) as ?numBlues)
FROM NAMED WINDOW <bw> ON :colorstream [RANGE PT15S STEP PT5S]
WHERE {
    WINDOW ?bw { ?b a color:Blue .} }
```

In RSP-QL the query is continuously evaluated against an SDS by an RSP engine. The reporting policy of the RSP engine that is set determines the time instants at which the evaluations occur. There are many different strategies for the reporting policy and they can me used in combination with each other. These strategies include:

- **Content Change** - report is made if the content of the current window changes
- **Window Close** - report is made if the current window closes
- **Periodic** - reports are made at regular intervals
- **Non-empty Content** - report is made if the current window is not empty

2.3.1 Yasper

YASPER - Yet another RDF stream processing engine is a library that can be used to build RDF stream processing Engines according to the reference model of RSP-QL [35].

The main goal for the creation of Yasper was to further the research of the Semantic Web by means of practical and usable software tools.

In this section, we will go over Yasper's architecture and the way that it uses the concepts of RSP-QL. Windowing, Querying, Stream and SDS - these are the modules that make up Yasper. Figure 2.8 illustrates these modules and how they relate to each other [37].

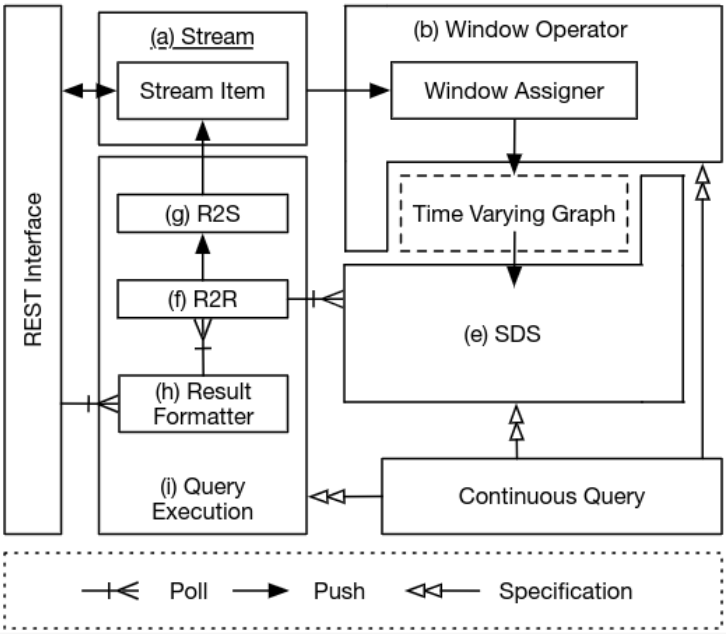

Figure 2.8: Modules that Yasper consists of

For the Stream module an URI is used to identify the Stream and the content of it. As expected with RDF data the StreamItem contains a triple (t_i, t_e, D) - the ingestion time of the stream to the system t_i, occurrence time of the event t_e, and lastly the data itself D.

The default setting of Yasper is to use the time of occurred events t_e although it can be adjusted to use the time of ingestion t_i instead.

Because the occurrence time of the event is decided when reading the Stream Items, each time there is an evaluation a RSP-QL query can be run against an instantaneous SDS. That is handeled by the R2R operator in the Query Execution module. The result of that query is then formatted into a suitable form by the Result Formatter. An SDS is a streaming data set that can be combined into a group of instantaneous graphs at the time when time-varying-graph is updated.

The module of Windowing deals with the incoming streams of data and partitions them into segments based on time-windows

The elements for the execution and instantation of the query lie in the module of Query Execution. CONSTRUCT and SELECT queries written in the syntax of RSP-QL are suitable to use for Yasper. Taking all of this into consideration the CQL model of Yasper is illustrated in Figure 2.9.

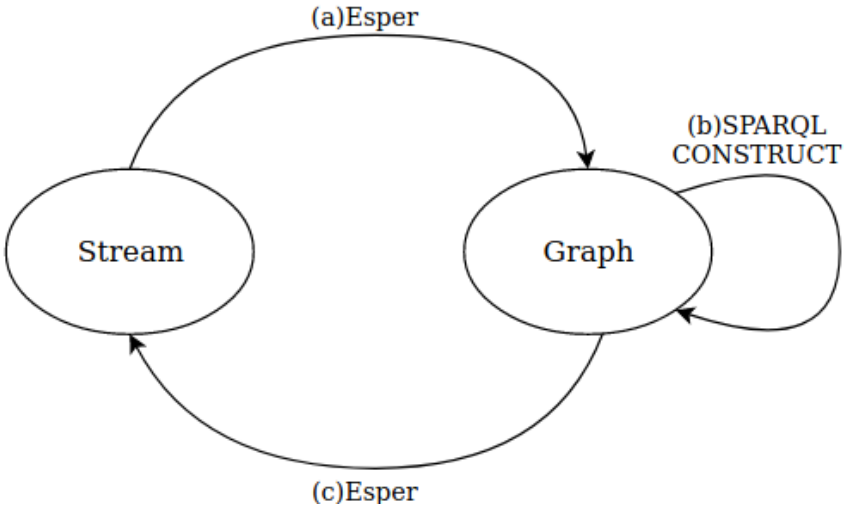

Figure 2.9: The semantic model of RSP-QL

In a simplified form the operational flow of Yasper looks like Figure 2.10.

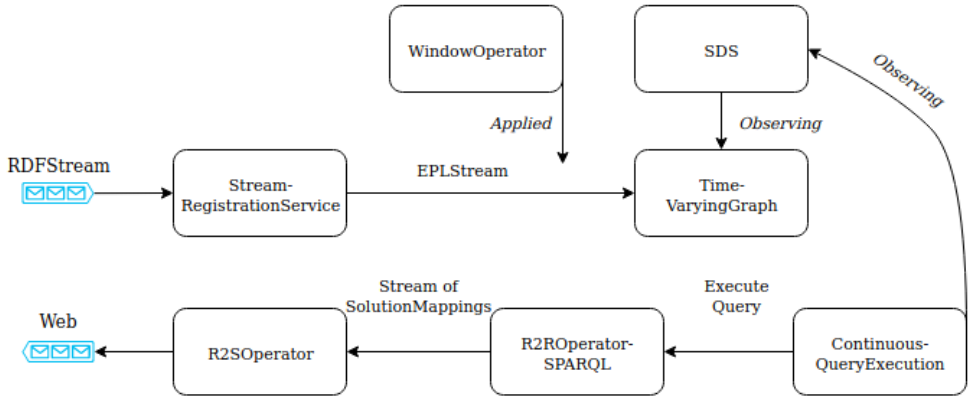

Figure 2.10: The operational flow of Yasper

2.3.2 Jasper

Jasper, also known as the CSPARQL Engine 2.0 is a Yasper runtime based on Esper and Jena.

The module of Windowing is based on Esper, *listener* objects that continuously receive the outputs of the EPL queries. The EPL statements and listeners represent a window that is based on time and is named on a RDF stream.

An Esper-based window operator is able to maintain one time-varying-graph. It can do so in two different ways - (α) it pushes everything new to the time-varying-graph every time there is an update or (β) only the differences between the old and the new window are added with the update. Yasper's default mode is (α) but it can be configured to be (β). Every time there is an update a instantaneous RDF graph is generated.

Any time Esper produces a result Jena is triggered to compute. This cycle of operation is done by different sub-classes that were all made to comply with the RSP-QL model. The standard life cycle of the operation goes as follows - the data gets into Esper, Esper triggers an answer which is then passed to Jena. The answer from Esper is then computed by Jena producing the

results of the query, after that the results are pushed out. Now the cycle of operation is described by explaining the most important Jasper classes.

First a WebStream is created that has a URI. It is then registered with the EsperStreamRegistrationService which returns a graph RDF EPLStream, it consists of the WebStream, EPStatement that can contain a *createStream* function and the URI of the stream. After that a EsperGGWindowOperator can be instantiated that provides a window with a step, range and a name. The part of RSP-QL that corresponds to R2S is the window. Then the window operator can be applied to the EPLStream the result will be a EsperTimeVaryingGraph that can be named. The EsperTimeVaryingGraph contains also the content of the stream elements and the streaming data set SDS. This means that the EsperTimeVaryingGraph will be observed by the JenaSDSBB class.

To start the part of querying the JenaContinuousQueryExecution needs to be instantiated and it will start observing the JenaSDS. So any time the EsperTimeVaryingGraph is updated the JenaSDS is notified which then notifies the JenaContinuousQueryExecutionBinding to execute the query. The R2ROperatorSPARQL is the part of the RSP-QL query in Yasper that corresponds to the SPARQL query. It is used to create a stream of SolutionMappings of a Jena data structure called Binding from the query results. The SolutionMappings correspond to the R2S part of the RSP-QL model. After that all of the data are put into the output stream. The data can be then consumed from the output stream to observe the results. In the Figure 2.11 the operational cycle of the system is illustrated. The simplified operational cycle of Jasper can be seen in Figure 2.11.

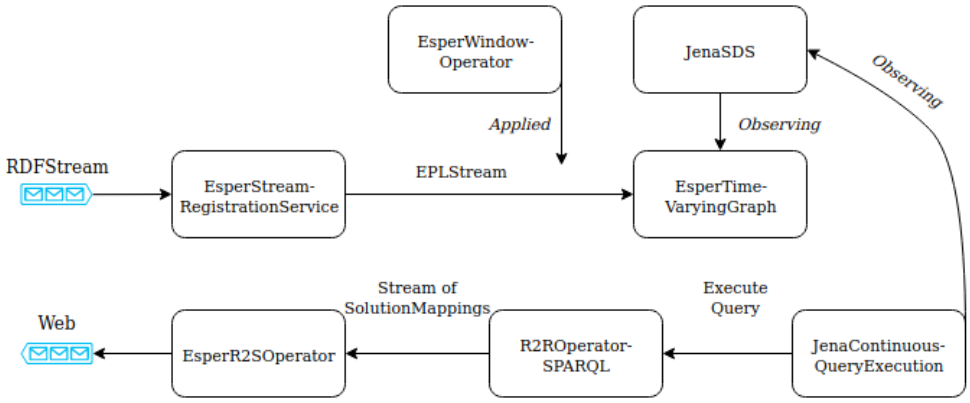

Figure 2.11: Simplified cycle of operation for Jasper

3 Design

In this chapter the process of the actual implementation of Neo4j into the existing C-SPARQL Engine 2.0 and the design of C-eraph will be discussed. The chapter is seperated into three parts, the aspect of refactoring the existing code, the implementation of Neo4j and Cypher and a walk-through of the code that runs the example of this project.

The intricate part of the design was understanding Yasper and finding out how to correctly implement Neo4j with its many differences from the previous solution. The implementation itself was not inherently grand from the aspect of the code length, rather it was the refactoring and understanding of the complex system that is Yasper that constituted to the most amount of work done.

3.1 Refactoring

In order to make the following work easier and be able to implement Neo4j, it was decided that an important step would be to first refactor the code to make it easier to implement Neo4j. The base code was dependent on Apache Jena and its data model. This meant that all the code that was in contact with the stream processing engine had to be made agnostic from the part of the database and querying.

In Jasper the part of R2R operator utilized the Jena TDB Dataset interface. It provides a way to communicate data without transactions, this does allow for faster communication but makes the data at risk of corruption [38]. In C-eraph there was no choice to communicate the data non-transactionally because Neo4j does not have that option. Each time the R2R operator is called for query evaluation a new transaction begins. In Jasper the result of the query evaluation was returned as a stream of RDF bindings. For the example data we used a mock social network that has data about people who have initiated and accepted friend requests. Since C-eraph uses Property Graphs we mapped the results of each query to contain the connected Node objects and their properties such as date and who was the initiator of the friend request. These results were put in a List and streamed out as a SolutionMappingImpl that contained the results and a timestamps.

In the base code the part of window operators was also reliant on Jena. The time-varying-graph was built to use Jena's JenaGraphContent class. Since Neo4j did not have an equivalent to that we had to build it on our own. ContentPGraphBean was created to server the part of the content necessary for the time-varying-graph. It takes the Neo4j in-memory database as a parameter and coalesces the data of the window to a graph.

Traditionally Neo4j does not work in-memory, it writes all of the data that it contains on disk. We saw this as a potential bottleneck for speed and decided to find a way to store the data

in-memory. After some searching and experimenting we found out that Neo4j has an ImpermanentDatabase class made for unit testing. Although this was not the perfect solution for our problem it still worked and served the purpose for building the prototype.

3.2 Implementation

In this section the necessary parts for the implementation of Neo4j will be gone over. The code was developed in Java 11 and the integral dependencies for this prototype were these:

- org.apache.commons commons-lang3 3.10
- com.github.jsonld-java jsonld-java 0.13.0
- com.espertech esper 7.1.0
- com.google.code.gson gson 2.8.6
- org.neo4 neo4j 4.0.3
- org.neo4j.community it-test-support 4.0.3
- org.slf4j slf4j-api 1.7.25

3.2.1 Structuring of the data

So since Yasper does not understand the concept of property graphs we needed to create a data structure that would somehow store property graph information. For this the PGraph interface was created that is able to hold both the nodes and edges along with the timestamp of the connections. This enables us to stream the property graph data in a simple and concise way.

As of this moment the implementation of the actual data that is used for constructing the PGraph's is done through a JSON file that holds information about a social network. Every event is in a form of one line that contains all of the necessary information. An example would look like this:

```
{"initiated":"Cory", "accepted":"Levi",
"friends":true, "date":"2019-08-08T16:13:11"}
```

This one line holds information about both the initiator and the one who accepted the friend request and the date when it happened. In a real scenario this information would come through a stream of events containing actual social network friend requests.

3.2.2 Querying

One of the first problems we faced was how to extend Cypher language to include CQL operators(R2R, R2S, S2R) to make it even more powerful. We call this extention Seraph, in order to understand what we did let us take a look at an example of Cypher and compare it to Seraph.

So a pure Cypher query would look like this:

```
MATCH (person1:Person)-[:KNOWS]->(person2:Person)
WHERE n.name = 'Alice'
RETURN person1
```

Now in order to introduce to Cypher the notion of *Windowing* and *Streams* we implemented the following logic.

```
MATCH (person1:Person)-[:KNOWS]->(person2:Person)
WHERE WINDOW [RANGE, STEP] n.name = 'Alice'
RETURN RSTREAM/ISTREAM/DSTREAM person1
```

The added implementation is highlighted in green. The line with the *MATCH* clause corresponds to the part of R2R and did not need any additions. The part of *Windowing* was implemented after the *WHERE* clause, it states the *WINDOW* and provides the *range* and *step* parameters for the window as needed for the S2R operator. In the last line we can see the part of R2S, where after the *RESULT* clause, we can specify the option for *Stream* operator.

To get a grasp of what was accomplished by this we can take a look at Figure 3.1.

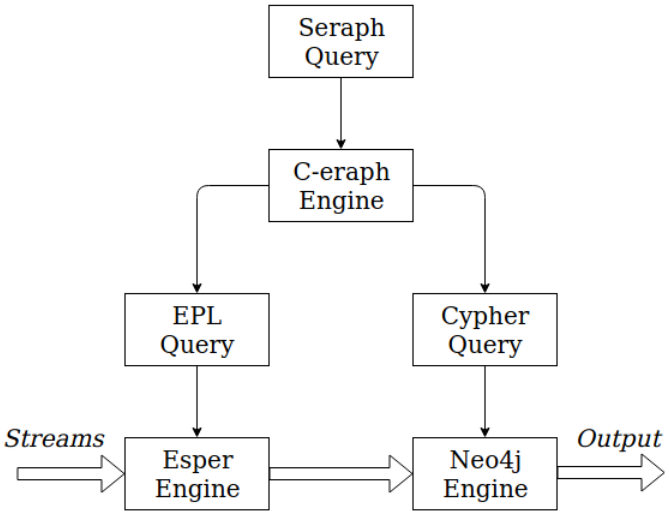

Figure 3.1: Breakdown of a Seraph Query

Because the base system was Jasper, the SPARQL querying had to be replaced by Cypher and take the part of the query execution that corresponds to the Relation-to-Relation in the CQL model. We extended the Time-Varying-Graph of the RDF graph to Time-Varying-PropertyGraph. In order to implement Cypher querying into Yasper we created the Seraph class that is able to hold the Cypher queries in a String format. The Seraph class extends the ContinuousQuery interface that corresponds to the Relation-to-Stream part of the CQL semantic model. Each time a Neo4jContinuousQueryExecution instance is created the Seraph query will be given to it as a parameter. This sets the query that will be run against the streaming data set.

3.3 Example of C-eraph

Here a walk-though of the code for the runnable example will be discussed and explained. The example was created in a way that is somewhat flexible, meaning that the configurations of certain parameters can be changed.

```
TestDatabaseManagementServiceBuilder builder = new
   TestDatabaseManagementServiceBuilder();
GraphDatabaseService db = builder.impermanent().build().database(
   DEFAULT_DATABASE_NAME);
```

Here the TestDatabaseManagementServiceBuilder is instantiated. It is a test factory for graph databases. GraphDatabaseService is the actual graph database that is built using the factory and creating an impermanent version of the database which does not write anything on disk an is in-memory. This is the database that will have transactions going through it later on in the code.

```
EngineConfiguration en = new EngineConfiguration(resource.getPath());
                     sr = new Ceraph(0, en);
```

Here the EngineConfiguration is created with the parameters that are in the */ceraph.properties* file. With these configuration parameters a Ceraph engine is instanciated. The Ceraph engine is just an extention of the existing Esper stream processing engine. The 0 just sets the start of the application time as current.

```
Seraph q = new Seraph("MATCH (n:Person)-[p]->(n1:Person)
                       WHERE WINDOW [10,10]
                       RETURN RSTREAM n, keys(n)");
```

Next a Seraph query is created that contains a String that will be used to query the data from property graph streams later on. Here the MATCH clause is used to find all *Person* nodes that have a relationship with another *Person* node, the keys() function returns a list containing the string representation for all the property names for the *Person* nodes.

```
GenericSDS<PGraph> sds = new GenericSDS<PGraph>();
WindowNode window = new WindowNode(q.getWindowMap());
```

SDS is a streaming data set that is composed of Time-Varying-Graphs, sliding windows and data streams.

The WindowNode gets the range and step parameters from the Seraph query using the getWindowMap function. For the example both of these parameters are set to 10 seconds. A visualization of the sliding window is presented in Figure 3.2. Next we create an instance of a Esper window operator, it requires several parameters that dictate the way the time is incremented, the time it was instantiated, the window, data set and the database and some other variables.

```
EsperPGWindowOperator wo = new EsperPGWindowOperator(
        Tick.TIME_DRIVEN,
        new ReportImpl(),
        true,
        Maintenance.NAIVE,
        new EsperTime(RuntimeManager.getEPRuntime(), 0),
        window,
        sds,
        db);
```

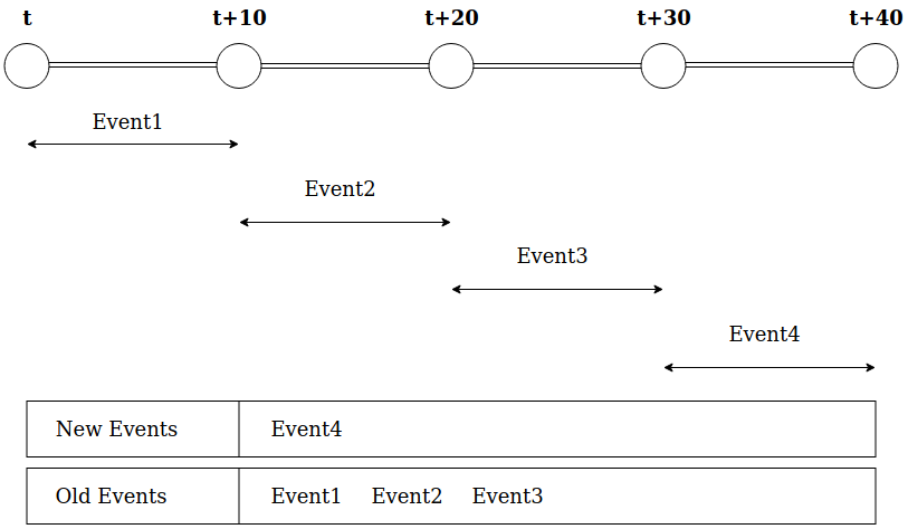

Figure 3.2: An example demonstrating how a sliding window with a 10 second range and step

Next up the PGraphStream is created and given a name *"stream1"*. After that the continuous query execution part is instantiated which is the part that actually executes the queries and streams them out as a Stream of SolutionMappings containing the results of the query. The SolutionMappings are created by the R2ROperatorCypher which begins the transaction with the database, executes the query, and then maps the all of the results of that query to a list of key-value mappings of the Node or Relationship and its values. After this each of the results are put into an *outstream* that will be used by the Relation-to-Stream operator.

```
PGraphStream writer = new PGraphStream("stream1", null);

Neo4jContinuousQueryExecution cqe = new Neo4jContinuousQueryExecution(
        new DataStreamImpl<>("stream1"),
        q,
        sds,
        new R2ROperatorCypher(q, sds, "SocialNetwork", db),
        new RelationToStreamOperator<>(q.getOuputStreamType()),
        wo);
```

Now whats left is to register the EPLStream to the C-eraph engine *sr*, apply the registered stream to the window operator, add the TimeVarying object containing a PGraph to the streaming data set *sds*, allow the streaming with the *setWritable* and add the continuous query execution *cqe* as an observer for the streaming data set *sds*.

```
EPLStream<PGraph> register = (EPLStream<PGraph>) sr.register(writer);

TimeVarying<PGraph> apply = wo.apply(register);

sds.add(apply);

writer.setWritable(register);

sds.addObserver(cqe);
```

After this the continuous streaming cycle is created and the data starts flowing through the system. It has no end point and stops the work if the program is stopped.

4 Summary

The end result of this thesis is a working software prototype written in Java that is able to stream property graph data via Esper and query it in real time using Neo4j GDMS and Cypher. We also managed to find a way to store the data in a in-memory using an extention of the Neo4j embedded Database.

The code for this prototype can be found on `https://github.com/FPBoldin/C-eraph`. As of now the evaluation of the prototype is ongoing and will be added to the webpage.

Both streaming and graph data are technologies that are continuously being more widely used, the need for real-time data and a way to efficiently store this data while making it highly available is undeniably growing. We believe that this project has taken a step towards continuous openCypher(an open-source project that makes it easy to use the Cypher language in order to incorporate graph processing capabilities within a product or application) and can be further developed into something that is truly unique and efficient in the problems it aims to solve.

5 Future work

For future work one of the things that would bring this prototype to another level would be the substitution of Esper with Kafka Streams. As discussed earlier, the prototype currently uses Esper as the processing engine, with the introduction of Kafka Streams the system would become highly scalable. The streaming data could be published to topics which are located in the Kafka cluster. This means that the streaming load is spread among different machines which could improve the throughput and lower the latency of the whole system. This would make it possible to have many streams running for different purposes at the same time communicating different kinds of data. One great thing about Kafka Streams is that it has an API which makes it easier for the users to use because they can integrate it right to their Java applications so it would fit quite easily into C-eraph.

Acknowledgements

In memory of the late Professor Dr.Sherif Sakr

Bibliography

[1] C. Humby, *Lecture, ana senior marketer's summit, kellogg school*, Nov. 2006.

[2] L. Lewis. (). Infographic: What happens in an internet minute 2020, [Online]. Available: https://www.allaccess.com/merge/archive/31294/infographic-what-happens-in-an-internet-minute. (accessed: 9.05.2020).

[3] IDC. (). The digitization of the worldfrom edge to core, [Online]. Available: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf. (accessed: 16.05.2020).

[4] F. Provost and T. Fawcett, "Data science and its relationship to big data and data-driven decision making," *Big Data*, vol. 1, Mar. 2013. DOI: 10.1089/big.2013.1508.

[5] S. Sakr, *Big Data 2.0 Processing Systems: A Survey*. 2016, p. 2.

[6] P. Russom, "Tdwi best practices report | big data analytics," pp. 1–34, 2011.

[7] C. W. Bachman, "The origin of the integrated data store (ids): The first direct-access dbms," *IEEE Ann. Hist. Comput.*, vol. 31, no. 4, pp. 42–54, Oct. 2009, ISSN: 1058-6180. DOI: 10.1109/MAHC.2009.110. [Online]. Available: https://doi.org/10.1109/MAHC.2009.110.

[8] K. D. Foote. (). A brief history of database management, [Online]. Available: https://www.dataversity.net/brief-history-database-management/. (accessed: 16.05.2020).

[9] DBEngines. (). Db-engines ranking, [Online]. Available: https://db-engines.com/en/ranking. (accessed: 16.05.2020).

[10] Neo4j. (). The native path to graph performance, [Online]. Available: https://neo4j.com/business-edge/native-path-to-graph-performance/. (accessed: 20.05.2020).

[11] C. Inc. (). Streaming data - a complete guide, [Online]. Available: https://www.confluent.io/learn/data-streaming/. (accessed: 9.05.2020).

[12] Neo4j. (). Chapter 6. kafka connect plugin, [Online]. Available: https://neo4j.com/docs/labs/neo4j-streams/current/kafka-connect/. (accessed: 20.05.2020).

[13] ——, (). Neo4j streams kafka integration, [Online]. Available: https://neo4j.com/labs/kafka/. (accessed: 12.05.2020).

[14] W. P. Bejeck, "Kafka streams in action: Real-time apps and microservices with the kafka streams api," 2018.

[15] A. Kafka. (). Introduction, [Online]. Available: https://kafka.apache.org/intro. (accessed: 17.05.2020).

[16] ——, (). Topics, [Online]. Available: https://kafka.apache.org/intro#intro_topics. (accessed: 17.05.2020).

[17] S. Langhi, "Towards extream processing with keplr," pp. 25–29, 2020.

[18] E. Inc. (). Esper, [Online]. Available: http://www.espertech.com/esper/. (accessed: 7.05.2020).

[19] ——, (). Chapter 3. processing model, [Online]. Available: http://esper.espertech.com/release-5.5.0/esper-reference/html/processingmodel.html#processingmodel_time_window. (accessed: 7.05.2020).

[20] ——, (). Esper faq, [Online]. Available: http://www.espertech.com/esper/esper-faq/. (accessed: 19.05.2020).

[21] ——, (). Esper faq, [Online]. Available: http://www.espertech.com/esper/esper-faq/#how-does-it-work-overview. (accessed: 19.05.2020).

[22] M. Hunger. (). From relational to graph: A developer's guide, [Online]. Available: https://dzone.com/refcardz/from-relational-to-graph-a-developers-guide?chapter=1. (accessed: 26.04.2020).

[23] E. E. Ian Robinson Jim Webber, *Graph Databases, 2nd Edition*.

[24] Neo4j. (). Neo4j streams kafka integration, [Online]. Available: https://neo4j.com/developer/guide-data-modeling/. (accessed: 16.05.2020).

[25] ——, (). Chapter 1. introduction, [Online]. Available: https://neo4j.com/docs/cypher-manual/current/introduction/. (accessed: 26.04.2020).

[26] A. Jena. (). Fuseki : Main server, [Online]. Available: https://jena.apache.org/documentation/fuseki2/fuseki-main. (accessed: 18.05.2020).

[27] ——, (). Apache jena fuseki, [Online]. Available: https://jena.apache.org/documentation/fuseki2/. (accessed: 18.05.2020).

[28] OpenLink. (). Rdf triple store faq, [Online]. Available: http://vos.openlinksw.com/owiki/wiki/VOS/VOSRDFFAQ. (accessed: 18.05.2020).

[29] ——, (). Rdf insert methods in virtuoso, [Online]. Available: http://vos.openlinksw.com/owiki/wiki/VOS/VirtRDFInsert. (accessed: 18.05.2020).

[30] J. Barrasa. (). Rdf triple stores vs. labeled property graphs: What's the difference? [Online]. Available: https://neo4j.com/blog/rdf-triple-store-vs-labeled-property-graph-difference/. (accessed: 18.05.2020).

[31] Oracle. (). Graph pattern matching for sql and nosql users, [Online]. Available: https://pgql-lang.org/. (accessed: 18.05.2020).

[32] C. Corner. (). Most popular graph databases, [Online]. Available: https://www.c-sharpcorner.com/article/most-popular-graph-databases/. (accessed: 18.05.2020).

[33] O. Inc. (). What is rdf? [Online]. Available: https://www.ontotext.com/knowledgehub/fundamentals/what-is-rdf/. (accessed: 11.05.2020).

[34] ——, (). What is sparql? [Online]. Available: https://www.ontotext.com/knowledgehub/fundamentals/what-is-sparql/. (accessed: 11.05.2020).

[35] D. Dell'Aglio, E. D. Valle, J.-P. Calbimonte, and O. Corcho, "Rsp-ql semantics: A unifying query model to explain heterogeneity of rdf stream processing systems," *International Journal on Semantic Web and Information Systems (IJSWIS)*, vol. 10, no. 4, pp. 17–44, 2014. [Online]. Available: https://EconPapers.repec.org/RePEc:igg:jswis0:v:10:y:2014:i:4:p:17-44.

[36] A. Gordon, *Programming Languages And Systems*, 19th.

[37] R. Tommasini and E. D. Valle, "Yasper 1.0: Towards an rsp-ql engine." [Online]. Available: http://ceur-ws.org/Vol-1963/#paper487.

[38] A. Jena. (). Tdb datasets, [Online]. Available: https://jena.apache.org/documentation/tdb/datasets.html. (accessed: 19.05.2020).

Non-exclusive licence to reproduce thesis and make thesis public

I, Fred Peter Boldin,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   "C-eraph: Towards continuous OpenCypher",

   supervised by Riccardo Tommasini.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Fred Peter Boldin
20.05.2020