POLITECNICO DI MILANO

Faculty of Engineering
School of Industrial and Information Engineering

Master of Science in
Computer Science and Engineering

NeX: a vertex-centric SPARQL engine

Supervisor:
PROF. ALESSANDRO MARGARA

Master Graduation Thesis by:
NEVENA KOTLAJA
Student Id n. 919147

Academic Year 2020-2021

Acknowledgments

I would like to thank my supervision, prof. Alessandro Margara, whose expertise and patience was invaluable in formulating the research questions and methodology. Your insightful feedback pushed me to sharpen my thinking and brought my work to a higher level.

In addition, I would like to thank my family for being always there for me. Mom Milijana and sister Jelena, your unselfish love is the main motivation for my personal growth. Jelena, you were always my hero and my best trustworthy listener. Mom, without your fearless love and patience nothing would be possible. My niece Elena, you are a symbol of pure and positive energy that is needed whenever an obstacle comes.

Moreover, heartfelt thanks goes to my friends, Sofija and Jefimija, for their noble support and empathy throughout my entire studies.

Finally, I could not have completed this dissertation without the support of my boyfriend, Tommaso, who taught me about loyalty and provided happy distractions to rest my mind outside of my research.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACRONYM

URL	Uniform Resource Locator
URI	Uniform Resource Identificator
API	Application Programming Interface
AWS	Amazon Web Services
CPU	Central Processing Unit
EC2	Elastic Compute Cloud
IEEE	Institute of Electrical and Electronics Engineers
JVM	Java Virtual Machine
LUBM	Lehigh University Benchmark
MPI	Message Passing Interface
NeX	Vertex-Centric Sparql Engine
RDF	Resource Description Framework
SPARQL	SPARQL Protocol and RDF Query Language
SPO	Subject-Predicate-Object
TLAV	Think Like A Vertex
W3C	World Wide Web Consortium
XML	eXtensible Markup Language

Abstract

Large graphs are increasingly common in many applications, which creates a tendency for distributed systems and faster extraction of data than the relational databases. Technologies that supports these systems are RDF, a general proposition language for the Web that unifies data from diverse sources, and SPARQL, a query language for RDF that can join data from different databases, documents, inference engines, or anything else that might express its knowledge as a directed labeled graph. In RDF, all data is declared in the form of subject-predicate-object, represented as graph's nodes and relationships. In practice, RDF graphs can be large, which is demanding in terms of memory and not applicable on centralized systems. The need to process graphs at large scale motivates our work, and our engine is based on Pregel - the programming abstraction that allows processing of large-scale graphs. A Pregel computation consists of a sequence of iterations, each called a superstep. Within each superstep the vertices compute in parallel, each executing the same user-defined function that expresses the logic of a given algorithm. Our engine, NeX, allows accessing distributed data in RDF graphs, pursuing scalability in parallel queries' executions. Furthermore, NeX is evidently a faster engine than Apache Jena - a Java framework for building semantic web and Linked Data applications.

Sommario

I grafi di grandi dimensioni stanno diventando sempre più comuni in molte applicazioni, il che crea una tendenza per sistemi distribuiti e un'estrazione di dati più veloce rispetto ai database relazionali. Le tecnologie che supportano questi sistemi sono RDF, un linguaggio di proposizione generale per il Web che unifica i dati provenienti da fonti diverse, e SPARQL, un linguaggio di query per RDF che può unire dati da diversi database, documenti, motori di inferenza, o qualsiasi altra cosa che possa esprimere la sua conoscenza come un grafo diretto etichettato. In RDF, tutti i dati sono dichiarati nella forma di soggetto-predicato-oggetto, rappresentati come nodi e relazioni del grafo. In pratica, i grafi RDF possono essere grandi, il che è impegnativo in termini di memoria e non applicabile su sistemi centralizzati. La necessità di elaborare grafi su larga scala motiva il nostro lavoro, e il nostro motore è basato su Pregel - l'astrazione di programmazione che permette l'elaborazione di grafi su larga scala. Un calcolo Pregel consiste in una sequenza di iterazioni, ognuna chiamata superstep. All'interno di ogni superstep i vertici calcolano in parallelo, ognuno eseguendo la stessa funzione definita dall'utente che esprime la logica di un dato algoritmo. Il nostro motore, NeX, permette di accedere a dati distribuiti in grafi RDF, perseguendo la scalabilità nell'esecuzione di query parallele. Inoltre, NeX è evidentemente un motore più veloce di Apache Jena - un framework Java per costruire applicazioni di web semantico e Linked Data.

1. Introduction

Knowledge graphs are at the core of many of the tools that we use in our daily lives, such as voice assistants (Alexa, Siri or Google Assistant)[1], intuitive search applications and even online store recommendations. Not only internet giants but also companies from other industries such as BBC, Capital One, Electronic Arts or AstraZeneca have already integrated the technology and are using knowledge graphs to harness the power of all of the data they have accumulated over the years. Structured as an additional virtual data layer, the knowledge graph lies on top of existing databases or data sets to link all data together.

Furthermore, large knowledge graphs require memory of more than one system. This creates a necessity of distributed graph processing approaches. One such approach is introduced with Google as a vertex-centric computing, known as a “think like a vertex” (TLAV). This approach consists of a sequence of iterations, each called a superstep. Moreover, it implements user-defined programs from the perspective of a vertex rather than a graph. A vertex can modify its state, receive messages sent to it in the previous superstep, send messages to other vertices (to be received in the next superstep), or even mutate the topology of the graph, all via message-passing communications. Such an approach improves locality, demonstrates linear scalability, and provides a natural way to express and compute many iterative graph algorithms.

The World Wide Web Consortium (W3C) has introduced Semantic Web - an extension of the World Wide Web. Semantic Web aims at converting the current web dominated by unstructured and semi-structured documents into a “web of data” that can be read directly by the computers. It is empowered by technologies such as RDF - a general language for the Web that unifies data from diverse sources, and SPARQL - a query language for RDF.

In this work, we present NeX, a distributed vertex-centric SPARQL query engine. Nex’s foundation is a distributed algorithm that runs on a computational model very close to the original Pregel idea. Our system is

[1] https://www.aboutamazon.com/devices/how-alexa-keeps-getting-smarter
https://techcrunch.com/2017/05/16/lattice-data-gives-apple-talent-and-a-next-generation-knowledge-graph/
https://www.bbc.co.uk/rd/blog/2012-09-search-engine-optimisation-kno

implemented using C++ and MPI, in order to exploit all the performance that modern hardware has to offer. We evaluate NeX both for its sequential and parallel scalability and for its query response time. Furthermore, we present a comparison with Apache Jena, an open-source Java framework for RDF data manipulation, and Verne, one more distributed vertex-centric SPARQL query engine.

The rest of this work is organized as follows. Chapter 2 offers background information and motivations. Chapter 3 introduces our system with a high-level description of its data model and algorithm, and Chapter 4 dives into the implementation details. Chapter 5 evaluates the scalability of the system and its performance compared to Apache Jena and Verne. Finally, Chapter 6 offers possible future work and optimizations.

2. Background and motivation

2.1. Semantic Web

There are many synonyms for Semantic Web, such as: Web 3.0, the Linked Data Web, the Web of Data. In any case, it represents the next major evolution in connecting information. It enables data to be linked from a source to any other source and to be understood by computers so that they can perform increasingly sophisticated tasks on our behalf.

Semantic Web is the technology that is still not widely used in comparison to the traditional Web. Still, one example of its users' daily visible usage is the Google Knowledge Graph[2], which is the information gathered from a variety of sources, presented in an infobox next to the search results in the Google Search Engine. Scale of Google Knowledge Graph by May 2020 is around 500 billion facts on 5 billion entities, which is hard to manipulate and to do extraction of data with other versions of the Web.

Moreover, the Semantic Web is a collaborative effort led by W3C, World Wide Web Consortium, with participation from a large number of researchers and industrial partners. In other words, it is a common framework that allows data to be shared and reused across application, enterprise, and community boundaries.

Strictly speaking, the Semantic Web is a web of data. Furthermore, its goal is to make Internet data machine-readable. There is lots of data we all use every day, and it is not part of the web. Since we don't have a web of data but the data is controlled by applications which keep it to themselves, this information cannot be used together. For example, the user can see his photographs on the phone, his appointments in a calendar, bank statements on bank applications, but the user cannot see what he was doing at the time of taking those photos.

Furthermore, the Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation [1].

[2] https://en.wikipedia.org/wiki/Google_Knowledge_Graph

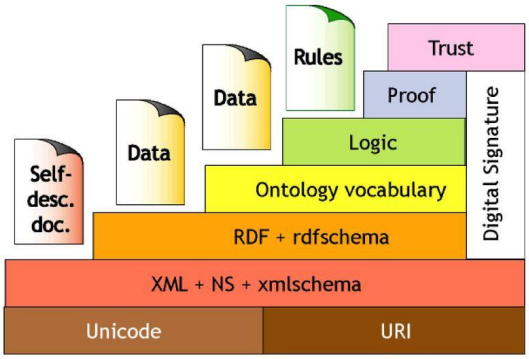


Figure 2.1.a - Semantic Web Tower

On the figure 2.1.a is a tower of the Semantic Web, which is showing different layers of the Semantic Web. On the very bottom is Unicode, common code on the Web, and URI[3], a unique sequence of characters that identifies a logical or physical resource used by web technologies. On top of the Unicode/URI base, is XML. The XML is the common language structure that connects everything together. To be more specific, through XML, the Semantic Web can use RDF[4], which are triplets that connect different objects together. Using RDF, it is able to build an ontology or vocabulary with basic rules that will tie on top with logic. Next, the proof can tell if the results are valid. Furthermore, RDF, Ontology vocabulary, Logic, and Proof, all form a digital signature which shows the validity of the document. On top of that validity is the last layer, the Trust, which gives a factor of data's truthfulness.

2.2. RDF - Resource Description Framework

The Resource Description Framework (RDF) is a family of World Wide Web Consortium (W3C), the main international standards organization for the World Wide Web, and represents a framework, standard model for data interchange on the Web. RDF is significant because it facilitates data merging even if the underlying schemas diverge, and it specifically

[3] https://www.w3.org/Addressing/URL/uri-spec.html
[4] https://www.w3.org/RDF/

supports the evolution of schemas over time without requiring all the data consumers to be changed. Moreover, RDF supports distributed and decentralized systems, which is of big significance for the Web.

In the RDF, all data is declared in the form of subject-predicate-object (Figure 2.2.a). This form of expression as a set of three entities that codifies a statement about semantic data, is known as triple. The subject denotes the resource, and the predicate denotes traits or aspects of the resource, and expresses a relationship between the subject and the object. For example, the statement “April is sunny” can be represented as the triple: “April” is the subject, a predicate denoting “is”, and an object is “sunny”. In contrast to the previously accepted approach, entity-attribute-value model, the RDF uses subject instead of object or entity.

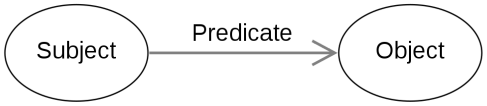


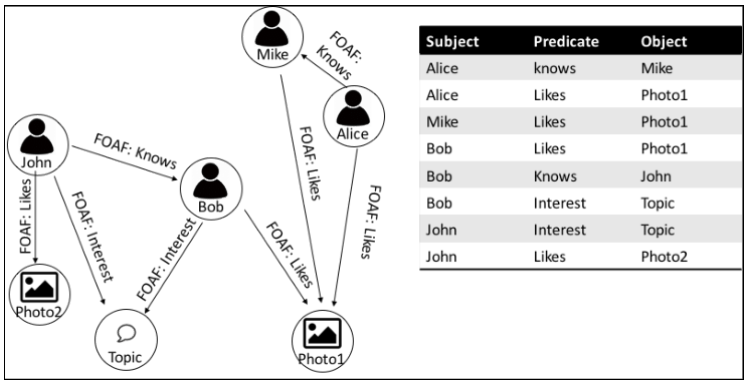
Figure 2.2.a - Subject, predicate, object representation in the RDF

Furthermore, one important attribute of the Resource Description Framework is URI, Uniform Resource Identifier. It is the fundamental component of the current Web and the foundation of semantic daya. URIs are not URLs (Uniform Resource Locator) since they are not addresses or locations, they are only identifiers. To be more specific, URIs eliminate ambiguity when data comes from different sources. In the RDF, the subject is either a uniform resource identifier or a blank node. Resources indicated by blank nodes are called anonymous resources. They are not directly identifiable from the RDF statement. The predicate is a URI which also indicates a resource, representing a relationship. The object is a URI, blank node or a Unicode string literal. Besides URIs, it can be used IRIs, Internationalized Resource Identifiers, which extends the URIs by including characters beyond ASCII ones.

RDF query language, similar to SQL, is SPARQL language. More of the SPARQL will be in the next chapter of this document.

Through the past few decades, relational databases have been used as the standard for storing data. Very important fact of this type of databases is that they have some drawbacks. For example, in order to have a good performance, it must be known upfront how to model the data and what questions could be asked from it. Moreover, relational databases

can be compared to concrete ones, where everything can be built, but changes are very hard to be implemented. In contrast to the fact that relational databases are not flexible, the RDF databases are very flexible. Databases that store semantic data are called RDF triple stores. They do not have SQL, nor schema design upfront. Likewise, RDF triple stores are fast and scalable, which means they can handle huge amounts of data, billions of triples. It is a form of graph database where subjects and objects are represented as nodes, and predicates are expressed as edges, which can be seen on the figure 2.2.b. Additionally, any number of edges can connect to a single node. This directed multi graph is made up of interconnected triples that can grow in a way the data dictates.



Subject	Predicate	Object
Alice	knows	Mike
Alice	Likes	Photo1
Mike	Likes	Photo1
Bob	Likes	Photo1
Bob	Knows	John
Bob	Interest	Topic
John	Interest	Topic
John	Likes	Photo2

Figure 2.2.b - RDF Triple Store

In order to translate an RDF data structure into a format that can be stored, there are in use a couple of serialization formats, like Turtle, N-Triples, N-Quads, RDF/XML. To be more specific, Turtle is a compact, human friendly format, commonly used. N-Triples is easy to parse, a simple format that is not as compact as Turtle.On the other hand, N-Quads is a superset of N-Triples, which can be used for serializing multiple RDF graphs. Furthermore, the first standard format for serializing RDF was RDF/XML, which can be sometimes misleadingly called RDF. RDF/XML is a syntax whose purpose is to express an RDF graph as an XML (Extensible Markup Language) document.

2.3. SPARQL

SPARQL, an acronym for Sparql Protocol And RDF Query Language, is recognized as one of the key technologies of the semantic web and has implementations for multiple programming languages. The protocol part resolves an issue of writing programs that pass SPARQL queries back and forth between different machines. For most users, SPARQL's greatest value is as a query language for RDF.

SPARQL allows the query to consist of triple patterns, conjunctions, disjunctions, and optional patterns. Additionally, SPARQL's formal definition is that the basic graph patterns are sets of triple patterns. SPARQL graph pattern matching is defined in terms of combining the results from matching basic graph patterns [2]. Next, conjunction represents the truth functional operator of logical conjunction where the and of the set of operands is true if and only if all of its operands are true. In this case it means that the SPARQL query will return results which satisfies all triples in the query constraints. Additionally, the disjunction represents the set of operands where at least one operand is true. To be more specific, inside the RDF logic, the disjunction would return all results for which at least one triple constraint is true.

In SPARQL are specified different types of query forms. First one is select form, where select query is used to extract raw values from a SPARQL endpoint and return the results in a table format. Second query form is the construct form, the one which is used to extract information from the SPARQL endpoint and transform the results into valid RDF. Third one is ask query form, which is used to provide a simple True/False result for a query on a SPARQL endpoint. The last query form is the describe query form, which is used to extract an RDF graph from the SPARQL endpoint, the content of which is left to the endpoint to decide, based on what the maintainer deems as useful information. Moreover, every type of the query has a “where” block, except “describe” query where it is optional.

Furthermore, in order to make it more readable and simpler to understand from the side of the users, SPARQL offen shortens the URIs by having an abbreviated prefix stand in for everything in the URI before the last part.

On the left part of the figure 2.3.a is shown part of SPARQL dataset triples. It can be seen, among others, that there exists an employee

“emp1”, which has given the name “Heidi”, family name “Smith”, title “CEO”, etcetera.

```
@prefix vcard: <http://www.w3.org/2006/vcard/ns#>
@prefix sn:    <http://www.snee.com/hr/>.

sn:emp1  vcard:given-name  "Heidi".
sn:emp1  vcard:family-name  "Smith".
sn:emp1  vcard:title  "CEO".
sn:emp1  sn:hireDate  "2015-01-13".
sn:emp1  sn:completedOrientation  "2015-01-30".
```

```
PREFIX vcard: <http://www.w3.org/2006/vcard/ns#>
PREFIX sn: <http://www.snee.com/hr/>

SELECT ?givenName ?familyName ?hd
WHERE
{
  ?person  vcard:given-name  ?givenName .
  ?person  vcard:family-name  ?familyName .
  ?person  sn:hireDate  ?hd .
}
```

Figure 2.3.a - Example of SPARQL
Left - part of triples; Right - query example

In addition to the figure 2.3.a, on the right part is the example of the SPARQL query. There are prefixes, which definition is early mentioned, there is a “select” clause which is asking for given name and family name. Furthermore, there is a “where” clause with three triple constraints. In this example the result is a set of all people in the dataset with this URI prefix, which is shown on figure 2.3.b.

?givenName	?familyName	?hd
Jane	Berger	2015-02-01
Francis	Jones	2015-03-10
John	Smith	2015-01-28

Figure 2.3.b - Result of SPARQL query

Moreover, there exist some extensions to SPARQL, like SPARUL. It enables the RDF store to be updated with this declarative query language, by adding “insert” and “delete” methods.

2.4. Graph processing - Pregel

In graphs, computations can be dependent on vertex's neighbours and their current information. To address this issue, Google's Pregel [3] architecture employs a message passing system creating a "large-scale graph processing" framework.

In Pregel, the computation consists of a sequence of iterations, called supersteps, with synchronization between workers occurring at superstep barriers (Figure 2.4.a). An inherent limitation of this synchronous approach is that workers can encounter the straggler problem, where fast workers must wait for the slow ones.

At each superstep, a vertex can execute a user-defined function, send or receive messages to its neighbours or any other vertex with a known identification, and change its state from active to inactive. Supersteps end with a synchronization barrier, Figure 2.4.a, ensuring that messages sent from one superstep are delivered at the beginning of the following superstep [4].

A vertex may vote to halt at any superstep and is woken up when it receives a message. Pregel terminates when all vertices are inactive and no more messages are in transit.

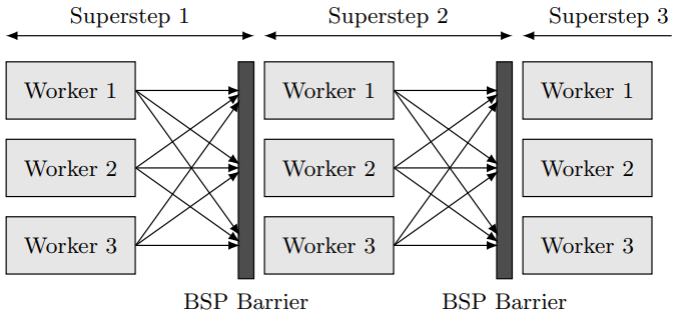


Figure 2.4.a - Basic computation model of Pregel, illustrated with three supersteps and three workers

Furthermore, let's see an example of the Pregel, a computation of the maximum value. Given a connected graph where each vertex contains a value, it propagates the latest value to every vertex (Figure 2.4.b). Any vertex that has known a larger value from its messages sends it to all its neighbors in each superstep. When no more vertices change in a

superstep, the algorithm terminates. Algorithm 2.4.c represents the pseudocode of the example.

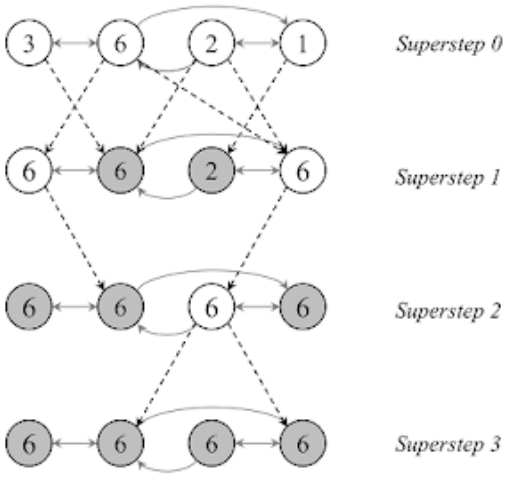


Figure 2.4.b - Maximum Value Example in Pregel.
Dotted lines are messages. Shaded vertices have voted to halt.

Algorithm 2.4.c. Maximum Value Pregel Pseudocode

```
for (all vertices) {
       received_values = recv_messages();
       if (received_values) active = true;
       if (active) {
              if (my_value < max(received_values)){
                     my_value = max(received_values);
                     send_message(my_value);
              }
       else active = false;
}
```

2.5. Message Passing Protocol

In some cases, information may not fit in the memory of a single machine, or the machine may not have enough processing resources to improve response time. When implementing distributed applications, developers rely on programming abstractions that simplify certain tasks. We rely on pure message passing and we use the implementation offered by MPI [14] for portability.

There are two approaches for synchronization between machines in concurrent programs - shared memory and message passing. Since distributed systems do not have shared memory, we use message passing.

The Message Passing Interface Standard, for which the acronym is MPI, is a message passing library standard. MPI is a communication protocol for programming parallel computers. Both point-to-point and collective communication are supported. The ultimate goal of the message passing interface is to establish a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs. MPI is not an IEEE or ISO standard. Still, it is the first standardized, vendor independent, message passing library. The advantages of developing message passing software using MPI closely match the design goals of portability, efficiency, and flexibility. MPI became the industry standard for writing message passing programs on high performance computing platforms.

Originally, MPI was designed for distributed memory architectures which can be seen at the figure 2.5.a. In this time of development, there was one central processing unit connected with one memory, the figure 2.5.b. With architecture standard evolution and changes, it is created a system with many different CPUs on one shared memory. In this case MPI was very flexible since the implementers adapted MPI’s libraries to handle both types of memory architectures.

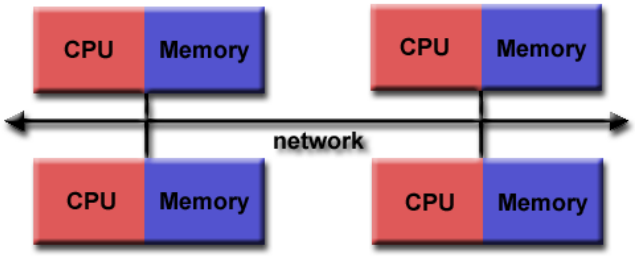


Figure 2.5.a - Memory architecture with one process by memory

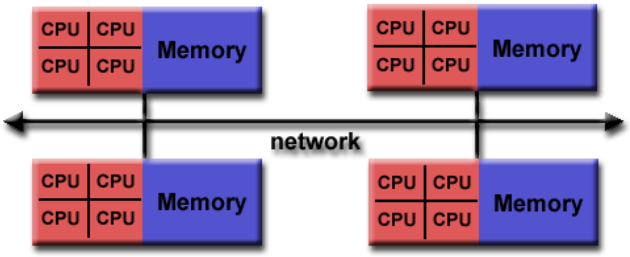


Figure 2.5.b - Memory architecture with many processes by memory

Today, MPI is available on any kind of hardware platform, even if it is a distributed memory, shared memory, or both - hybrid.

The general MPI program structure has a couple of steps, shown on the figure 2.5.c. First of all, MPI needs to be initialized, the programmer needs to include the file, add optional flags if necessary. The command “MPI_Init” initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program. For C programs, MPI_Init may be used to pass the command line arguments to all processes, although this is not required by the standard and is implementation dependent. When the program is started, for some period the system will be executed as a serial one. At the point of MPI environment initialization, the parallel code begins. For example, inside the MPI initialization are considered initialization with program arguments, specifying number of tasks, rank of each process. Rank is within a communicator where every process has its own unique, integer identifier assigned by the system when the process initializes. A rank is sometimes also called a “task ID”. Ranks are contiguous and begin at zero. Moreover, ranks can be used by the programmer to specify the source and destination of messages. After the point of MPI initialization, the further

code is executed in parallel by making message passing calls in between of MPI processes. The parallel execution stops when the process comes to a MPI termination, after which the code is serial once again. To do so, the command “MPI_Finalize” is used. This command terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program. No other MPI routines may be called after it.

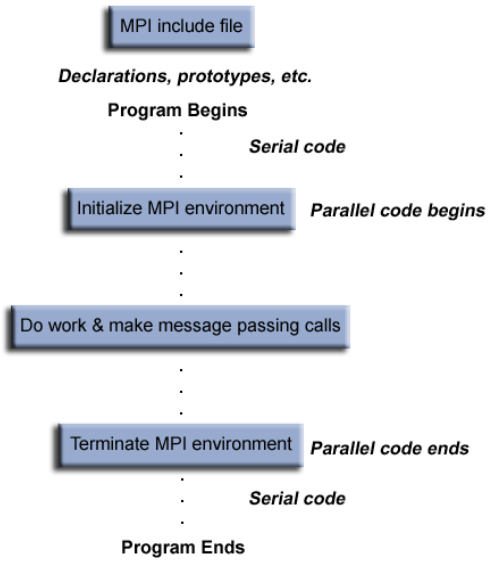


Figure 2.5.c - General MPI program structure

Message Passing Interface provides a diverse range of abilities. First of all, MPI uses objects called communicators and groups to define which collection of processes may communicate with each other. Most MPI routines require you to specify a communicator as an argument. Furthermore, a commonly used communicator is “MPI_COMM_WORLD”, a predefined communicator that includes all MPI processes.

Moreover, each communicator gives each contained process an independent identifier and arranges its contained processes in an ordered topology. From the part of point-to-point MPI communication, many important MPI functions involve communication between two specific processes. A popular example is “MPI_Send”, which allows one specified process to send a message to a second specified process. The MPI specifies mechanisms for both blocking and non-blocking point-to-point communication mechanisms, as well as the ready-send mechanism whereby a send request can be made only when the matching receive request has already been made. For the perspective of the MPI datatypes, many MPI functions require that you specify the type of data which is sent

between processes. This constraint is mainly because MPI aims to support heterogeneous environments where types might be represented differently on the different nodes. For example, some already defined MPI datatypes are: “MPI_INT”, “MPI_CHAR”, “MPI_DOUBLE”. In the case of collective communication routines, there exists “MPI_Barrier”, which is a synchronization operation. It creates a barrier synchronization in a group where each task, when reaching the MPI_Barrier call, blocks until all tasks in the group reach the same “MPI_Barrier” call. Then all tasks are free to proceed.

For the blocking message protocol routines, a number of controls are used. First of all, “MPI_Send(buf, count, datatype, dest, tag, comm,ierr)” control represents basic blocking send operation. Routine returns only after the application buffer in the sending task is free for reuse. For basic, blocking receive is used “MPI_Recv(buf, count, datatype, source, tag, comm, status, ierr)”, which receives a message and blocks until the requested data is available in the application buffer in the receiving task. Moreover, “MPI_Ssend(buf, count, datatype, dest, tag, comm, ierr)” command can be used for synchronous blocking send since it sends a message and blocks until the application buffer in the sending task is free for reuse and the destination process has started to receive the message. The “MPI_Sendrecv” sends a message and posts a receive before blocking. This command will block the process until the sending application buffer is free for reuse and until the receiving application buffer contains the received message. One more important command is “MPI_Probe” which performs a blocking test for a message. The tokens “MPI_ANY_SOURCE” and “MPI_ANY_TAG" may be used to test for a message from any source or with any tag. In this case, the actual source and tag will be returned in the status structure as status.

For the non-blocking message protocol routines, the basic send command is “MPI_Isend”. This command identifies an area in memory to serve as a send buffer. Processing continues immediately without waiting for the message to be copied out from the application buffer. A communication request handle is returned for handling the pending message status. The program should not modify the application buffer until subsequent calls to “MPI_Wait” or “MPI_Test” indicate that the non-blocking send has completed. Furthermore, for the non-blocking receive is used “MPI_Irecv”, which identifies an area in memory to serve as a receive buffer. It is similar to the MPI non blocking send command. Processing continues immediately without actually waiting for the message to be received and copied into the application buffer. A

communication request handle is returned for handling the pending message status. The program must use calls to “MPI_Wait” or “MPI_Test” to determine when the non-blocking receive operation completes and the requested message is available in the application buffer.

2.6. Motivation - overview of existing query languages

RDF’s development includes accepting the SPARQL as the standard query language for RDF, as well as the development of a variety of local and distributed engines for processing queries over RDF graphs. These engines implement a diverse range of specialized techniques for indexing, query processing, and storage. Our idea was to create our own SPARQL engine from scratch in C++, in order to gain better performance by mixing already existing techniques and adding new ones.

Let’s see and compare different types of these techniques, and why exactly they were the motivation for NeX.

2.6.1. Indexing

Indexing enables efficient lookup operations on RDF graphs (usually O(1) or O(log |G|) time to return the first result or an empty result). We now discuss some indexing techniques proposed for RDF graphs.

- ❖ Triple Indexes

The goal of triple-pattern indexes is to efficiently find all triples that match a given triple pattern. Specifically, letting s, p, o denote RDF terms as subject, predicate, and object, and s, p, o variables, there are 2^3 = 8 abstract triple patterns: (**s**, p, o), (s, **p**, o), (s, p, **o**), (**s**, **p**, o), (**s**, p, **o**), (s, **p**, **o**), (**s, p, o**) and (s, p, o). Unlike relational databases, where often only the primary key of a relation will be indexed by default and further indexes

must be specified by the database administrator, most RDF stores aim to have a complete index by default, covering all eight possible triple patterns.

For example, specialized indexes can be used to quickly evaluate such patterns, where QDags [5] uses quadtrees: a hierarchical index structure that recursively divides the matrix into four sub-matrices.

- Entity-Based Indexes

Entity-based indexes refer to indexes that support the efficient evaluation of graph patterns that “center on” a particular entity.

For example, a flexible approach for this technique is to index signatures of entities, which are bit-vectors encoding the set of property–value pairs of the entity. One such example is the vertex signature tree of gStore [6], which first encodes all outgoing (p, o) pairs for a given entity into a bit vector, and then indexes these bit vectors hierarchically allowing for fast, approximate containment checks that quickly find candidate entities given a subset of such pairs. A similar approach is adopted by the GRaSS engine [7], which optimizes for star subgraphs that include both outcoming and incoming edges on nodes.

- Path Indexes

A path join involves successive subject-object joins between triple patterns, for example {(w, p, x),(x, q, y),(y, r, z)}, where the start and end nodes (w, z) may be variables or constants. A path can be seen as a string of arbitrary length; e.g., a path {(w, p, x),(x, q, y),(y, r, z)} can be seen as a string wpxqyrz$, where $ indicates the end of the string.

Example for this approach is the Yaanii system [8] which builds an index of paths of the form wpxqyrz$.

Our approach for indexing is motivated by the mix of triple indexes and entity-based indexes. To be more specific, we have six triple patterns which are responsible for finding the right entity for indexing: (**s**, p, o), (s, **p**, o), (s, p, **o**), (**s**, **p**, o), (**s**, p, **o**), and (s, **p**, **o**). Furthermore, that entity is specifically indexed and can be retained in O(1) time since we are using hash maps.

2.6.2. Join Processing

RDF stores employ diverse query processing strategies, but all require translating logical operators, which represent the query, into “physical operators”, which implement algorithms for efficient evaluation of the operation. The most important such operators are joins.

- Pairwise Join

The simplest and most well-known such algorithms perform pairwise joins; for example, a pairwise strategy for computing {t1, . . . tn}(G) may evaluate ((t1(G) & t2(G)) & . . .) & tn(G), where “&” denotes join operation. Well-known algorithms for performing pairwise joins include nested-loop joins, where P1(G) & P2(G) is reduced to evaluating S μ∈P1(G) {μ} & μ(P2)(G); hash joins, where each solution μ ∈ P1(G) is indexed by hashing on the key (μ(v1), . . . , μ(vn)) and thereafter a key is computed likewise for each solution in P2(G) to probe the index with; and sort-merge joins, where P1(G) and P2(G) are (sorted if necessary and) read in the same order, allowing the join to be reduced to a merge sort.

For example, pairwise join algorithms are used in many RDF stores [9,10].

- Multiway Joins

Multiway join algorithms exploit the commutativity and associativity of joins to evaluate them over two or more operands at once. For example, in order to compute {t1, . . . tn}(G), a multiway join algorithm may evaluate ((t1(G) & . . . & tk(G)) & (tk+1(G) & . . . & tn(G)) where k ≥ 2, or even simply (t1(G) & . . . & tn(G)).

For example, SMJoin [11] decomposes BGPs into sub-BGPs corresponding to star joins, evaluating these separately before combining the results.

- Join Reordering

The order of join processing can have a dramatic effect on computational costs. If we evaluate the join in the order, the second plan should thus be more efficient than the first. If considering a graph at a larger scale, the differences may reach orders of magnitude. Furthermore, a good plan depends not only on the query, but also the graph. Selecting a good plan thus typically requires some assumptions or statistics over the graph.

To conclude, for joins we use a mix of pairwise and multiway joins. By going from one node to the next one, the new node does the join in a pairwise strategy by joining its results with the old ones. Still, the new node can join more than one variable at the same Pregel's superstep, which is a property of multiway joins.

Join reordering can be crucial for the performance of NeX. In order to gain maximum performance, users need to know in which order to write the triple constraints before running NeX. Usually, the best orders are the ones where triples which return the least number of results are at the beginning of query constraints. The reason why is to gain better performance by filtering, joining, and sending less information.

2.6.3. Storing

Data storage refers to how data is represented in memory. Different storage mechanisms store different elements of data contiguously in memory, leading to different trade-offs in terms of compression and efficient data access.

- Triple Table

A triple table stores an RDF graph as a single ternary relation, often in a relational database. One complication is that relational engines typically assume a column to be associated with a single type, which may not be true for RDF objects in particular.

The most obvious physical storage is to store triples contiguously, row-wise. This allows for quickly retrieving the full triples that match a given triple pattern. However, some RDF engines based on relational storage, like Virtuoso [12], rather use column-wise storage, where the values along a column are stored contiguously, often following a particular order.

- ❖ Property Table

Property tables aim to emulate the n-ary relations typical of relational databases. A property table usually contains one subject column, and n further columns to store objects for the corresponding properties of the given subject. The subject column then forms a primary key for the table. Property tables allow for storing and retrieving multiple triples with a given subject as one tuple without requiring joins.

Such queries are common in practice, like Jena2 [13].

Our storing approach is explained in section 3.1.

3. Design

3.1. The model

As previously explained, the core structure of the abstract syntax of RDF is a set of triples, each consisting of a subject, a predicate and an object. Let's call subject, predicate, and object by triple elements. In this project, subject and object are nodes of the graph, and predicate is a relation between the nodes. Each triple element has its own id as an integer. That id is responsible for the machine in which the triple element will be stored. To be more specific, number of the machine which has the element is calculated by next equation:

3.1.0. Formula for machine number

machine_id = element_id % num_of_machines

where "element_id" is the id of the element, "%" is module operation, and "num_of_machines" is the number of MPI machines. For example, on the MPI machine with the rank 1 will be stored all nodes and relations which id by module of number of machines is giving result 1. The reason why this is very important is that any kind of node or relationship activity is connected with the machines on which they are on.
In order to gain better performance, this research paper is using more memory than usual for additional structures that each MPI process has. Not only that each MPI machine is storing its own triple elements, but according to each subject and object there exists a hash map (unorder_map in C++) in which the key is the id of the triple element and the value is a pointer to the element stored on that machine. The purpose of the hashmaps is quicker detection of the triple elements. To be more specific, there are two hashmaps, one for all nodes (subjects and objects) and one for the relations since there are two different structures - Node and Relationship.

Additionally, on the figure 3.1.1. is shown the model class diagram. Node structure has its own id (URI) and types. Node's types are hashmaps with key of relationship id and value of pointer to the structure "RelationshipType". The RelationshipType structure has two vectors of pointers to Relationships - one for relationships that are directing into the node, and one for the relationships that are directing out of the node into some other node. In this way, when the node is found through the machine listener's hashmaps, node's hashmaps are used in order to find all or specific relationships. The Relationship structure has a unique id, start node and end node as an integer.

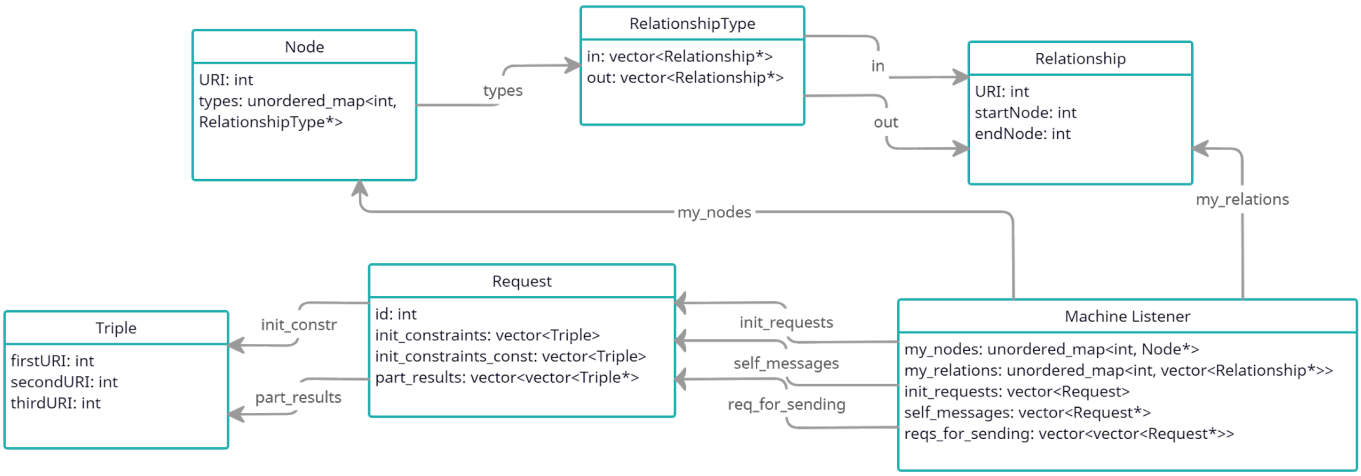


Figure 3.1.1 - The model class diagram of the project

Furthermore, at the figure 3.1.1, a triple structure is also demonstrated, which corresponds to the RDF triple with subject, predicate, and object. In addition to that, the most important structure for sending and receiving of the MPI packages is the "Request" structure. In it is stored id of the request since multiple query requests can be executed, the init constraints as a vector of initial triple constraints. Moreover, there is one more initial constraint structure which is constant and cannot be changed, the opposite of the first one which changes while a query is forwarded between machines. The results of a request are saved inside "part_results" in the Request structure, which will be explained better in the implementation part of this paper.

To be more specific, let's examine the figure 3.1.2. in order to understand how the connection between nodes and relationships is. Let's suppose there exists a node with id equals to 1. The node has its types of

relationships, as seen on the model at the figure 3.1.1. Moreover, each type has an in and out vector of relationships, where type1 on the figure has an incoming arrow from the node 6 and an outgoing arrow to the node 2. With this approach, it can be found any relationship that each node has, and connections between them - relationships.

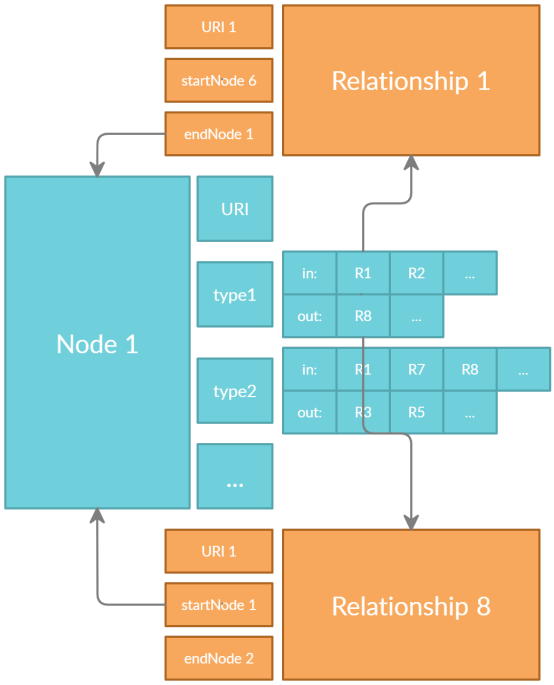


Figure 3.1.2 - Connection between nodes and relationships in the model

Moreover, the query 3.1.3. explains the nodes, relationships, and their connections.

3.1.3. Query

```
SELECT ?x ?y ?z WHERE {
             ?x rdf:type ub:Student .
             ?x ub:takesCourse ?y .
             ?y rdf:type ub:Course .
             ?y ub:name ?z .
}
```

When the first triple, “?x rdf:type ub:Student”, is executed, the corresponding machine finds node “ub:Student”. Once found, this node is used to find the type inside types that stands for “rdf:type”. Next step is looking into the “in” vector of relationships and gathering all of them. This

is the case of the unknown subject. Very similarly, when the object is unknown, the “out” vector of relationship is examined.

3.2. Initial configuration

The program does not recognize the form of the query that is shown with the query 3.1.3. Istead, it needs some initial configuration to be able to run the SPARQL query format. The program gets the file whose name is added as the fifth argument, parses it, and looks at it as one or more requests. For example, the previously mentioned query 3.1.3. needs to be converted in the way it is presented in the figure 3.2.1.

3.2.1. Transformed Query

```
REQUEST
-1 2 11
-1 3 -2
-2 2 12
-2 4 -3
END REQUEST
```

Furthermore, this transformation has a couple of differences in comparison to the Query 3.1.3:

1. It does not require adding required output variables since the program returns all variables,
2. Each triple element is converted into its own unique id. For example, “Student” has id equals to 11,
3. Variables are presented as negative integers. For example, “?x” is represented as “-1”, “?y” as “-2”, “?z” as “-3”.

One usual format for RDF nodes is the ".nt” format, which stores all RDF triples. Moreover, this program has the ability to convert from “.nt” format into its own structures. In order to do the conversion, a user needs to write number 1 as the second program argument and path to the “.nt” file as the third argument. In this way, one MPI process parses the “.nt” file and creates three new files which are in the format that the program uses

if it is run with number 0 as the second argument. To clarify, second argument of the program is used for two modes:

1. Conversion mode - the second argument is 1. Converts“.nt” file to project’s acceptable files,
2. Query mode - the second argument is 0. Feeds the structures with the information gathered from parsing project’s acceptable files, and runs the queries.

3.2.2. Running command

```
mpiexec -n 6 PatternMatching 6 1 0 ../Universities/Universities1.nt
../Queries/all.txt
```

The example of the program’s command used for its running is shown at the command 3.2.2. Arguments are:

1. First argument is number of MPI processes,
2. Second argument is 1 or 0, depending on Conversion and Query mode,
3. Third argument is the flag for parallel or sequential query execution (not important in the Conversion mode),
4. Forth argument is path to the “.nt” file (not important in the Query mode),
5. Fifth argument is path to the query file (not important in the Conversion mode),

Additionally, multiple query requests can be added inside the query file, in the format shown at the Transformed Query 3.2.1. After that, a user can choose between parallel and sequential query execution of those queries, which will be explained further in the implementation and evaluation parts of this document.

3.3. The algorithm

In the center of the algorithm is the Machine Listener class (figure 3.1.1) which is responsible for executing previously explained Pregel supersteps. Moreover, this class not only receives and sends MPI messages, but also executes queries gathered from MPI and self messages. Self messages are messages that a process sends to itself. In this case, when a process (one MPI machine) needs to send a message to itself, the process doesn't need to use the MPI send function. Instead, the process just saves that message in local storage Furthermore, each main listener represents one MPI machine.

Main Listener execution design is demonstrated on the algorithm 3.3.1, which shows basic execution of MPI machines. The algorithm is divided into four parts:

1. Sending and receiving of the MPI messages,
2. Execution of MPI messages,
3. Execution of self messages,
4. Checking of termination.

Algorithm 3.3.1. Main Listener execution

```
while (!stop) {
	// FIRST
	vector<Request*> mpi_mess = send_recv_mpi();
	// SECOND
	if (mpi_mess.size() > 0)
		execute_mpi();
	// THIRD
	if (self_messages.size() > 0)
		execute_self_messages();
	// FOURTH
	check_termination(&stop);
}
```

First of all, there is a while loop which is executed until all MPI processes do not have anything to send. In other words, while at least one MPI process has something to send to some MPI process, the while loop continues its execution. The first part, sending and receiving of the MPI messages is done always at the beginning since it sends previously

gathered messages to MPI processes and then receives messages from them. These, received MPI messages, are then used for the second and third part of the algorithm, execution of MPI messages and self messages. At the fourth stage, termination of the main listener is checked. If there is no MPI process which has messages to send, the while loop stops.

More precisely, the flow of the query (presented at the 3.1.3) between MPI machines is demonstrated on the figure 3.3.2. First of all, one node must be responsible for the first forwarding of the queries. This information can be modified at the “QUERY_NODE” variable inside the program. Let’s say the MPI machine number 1 is the one responsible for forwarding the queries for the first time. Depending on the parallel and sequential mode, query forwarding is different.

First let’s start with sequential mode when there is one query, the previously mentioned one. At the first execution of the while loop (algorithm 3.3.1) the only machine that has something to send is the query machine, machine 1. Moreover, the assumption is done that there are 5 MPI machines. On the figure 3.3.2, machine “M1” needs to forward the query to the next machine responsible for the first triple constraint, “-1 2 11”. In this case it is forwarded to the machine on which node 11 is stored. By applying the formula 3.1.0, the machine responsible for node 11 is machine 1. To conclude, machine 1 needs to forward the query to itself, part “a" on the figure. This is the occasion in which the machine does not send the query by the MPI, but saves it inside the local vector responsible for self messages (look Machine Listener at figure 3.1.1). All machines, except machine 1, have nothing to do since there are no MPI, nor self messages, and they skip this iteration of the while loop. The machine 1 skips the “execute_mpi” function, and executes only the “execute_self_messages” function in which it finds all triples corresponding to the first constraint, which will be better explained in the implementation part of the document. Next step of the while loop is the termination checking. Since there are still messages to be forwarded, “while” loop continues to the second iteration. Furthermore, the next triple constraint is “-1 3 -2” in which machine 1 sends the query to machine 3, shown on “b” part of the figure 3.3.2. Machine 3 will execute MPI messages since they arrived through MPI and are not self messages. In this part, machine 3 will filter previous results received from machine 1 with the new results, doing a hash join on the variable -1. To be more specific, first and second triple constraints are connected with the variable -1 and their join needs to be done at this point. This join creates partial results, which the machine 3 forwards to the machine 2 (12 mod 5), shown

on the “c” part. Machine 2 executes the third constraint of the query on the third iteration of the while loop, by executing “execute_mpi” function. Here, new results filter the old ones with -2 variable. Next step is the fourth execution of the while loop in which machine 2 sends the partial results to machine 4 (4 mod 5). Machine 4 executes the fourth triple constraint and filters the old results with variable -2 from the new results. At the end of execution of the fourth machine, partial results become final results.

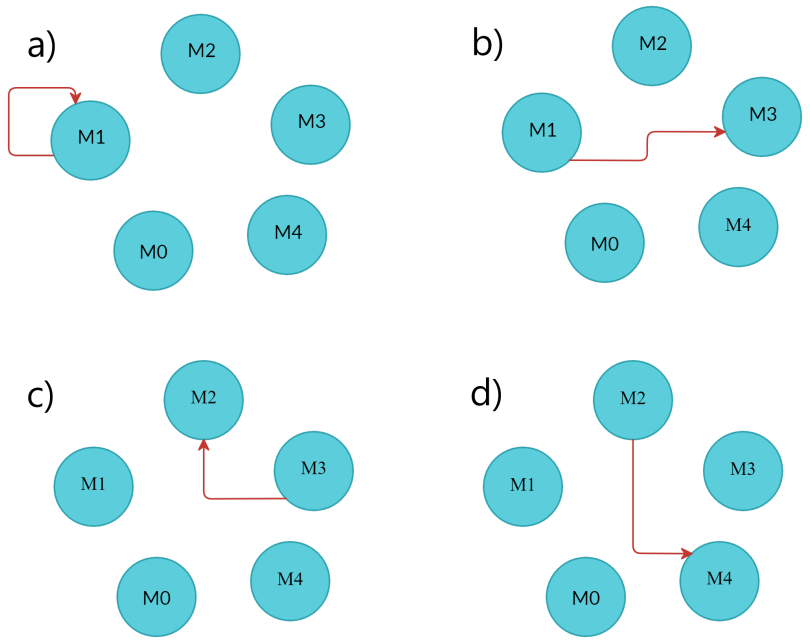


Figure 3.3.2. - The flow of query 3.1.3. between MPI machines

Furthermore, more queries can be executed in parallel, using the third program’s argument. Let’s add one more query to the previously explained one:

3.3.4. Query Example

```
SELECT ?x ?y WHERE {
       ?x rdf:type ub:Course .
       ?y rdf:type ub:Student .
       ?y ub:takesCourse ?x .
}
```

with the 3.3.5. query, which represents transformed 3.3.4 query:

3.3.5. Transformed Query

```
REQUEST
-1 2 12
-2 2 11
-2 3 -1
END REQUEST
```

In other words, in the same query file are two query requests: query 3.2.1. and query 3.3.5. Their parallel execution is demonstrated on the figure 3.3.3. Moreover, the same assumptions stay, where numbers on edges on the flow graph describe the type of query - number one is query 3.2.1. and number two is query 3.3.3. In this example it can be seen that at every iteration of the while loop, one triple constraint is executed from both queries, in parallel. In details, at the part “a” on figure 3.3.3. machine 1 communicate with machine 1 (first constraint of first query) and with machine 2 (first constraint of second query). Next, part “b” shows further forwardings and their executions - the second constraints from both queries. In this case the first machine is communicating with the third one (first query), and the second machine with the first one (second query).

The main difference between parallel and sequential mode is that in the sequential mode the second query will start executing after the first query finishes. On the other hand, parallel mode can be faster since more multiple queries can be executed in parallel.

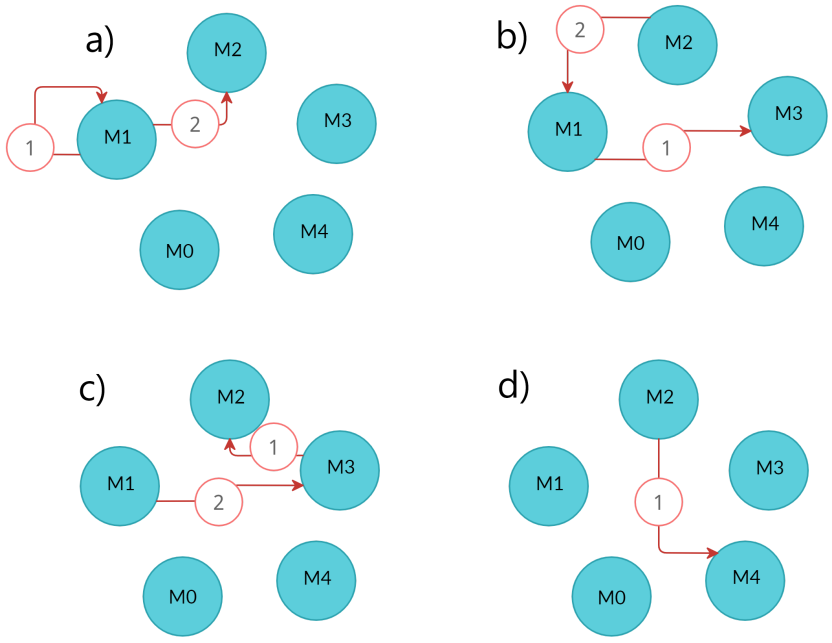


Figure 3.3.3 - The flow of query 3.1.3. between MPI machines

4. Implementation

Machine listener has fourth stages, as described at algorithm 3.3.1. This chapter describes the implementation of each stage.

4.1. First stage of Machine Listener

4.1.1. Send and Receive Function

The first phase of the machine listener's execution is the one connected to the communication between MPI machines. This part is completely connected to receiving new messages and sending new messages. Moreover, this kind of MPI send and receive implementation is free of deadlocks, starvations, and race conditioning. Let's see the algorithm 4.1.1.a.

The function send_recv_mpi receives vectors of vectors of pointers to Requests, where we store all requests from the previous superstep that should be sent to all machines. First (outside) vectors describe the machine to which messages should be sent (from 0 to NUM_OF_MACHINES). Second (inside) vectors represent all messages for the specific machine. This function has a couple of stages.

First, it is very important which machine is executing the code. Here we go through all machines and check if the current machine is ours. In other words, if machine rank is the same as the current executing machine, it should be all connected to self messages, not with MPI. If that is the case, current requests we push to self messages. If not, we transform the request and save it for the next stage - sending. Transformation of the request is crucial, and we will describe it later.

Second, if it is not a self messaging case (not our machine), we send non-blocking messages to every machine. Moreover, the messages are the ones from the previous stage, already transformed in our suitable form.

Algorithm 4.1.1.a. Send/Recv Function

```
vector<Request*> send_recv_mpi(vector<vector<Request*>> reqs){
	// Transform messages
	for(int i from 0 to NUM_OF_MACHINES){
		if(i == my_rank) self_messages.push_all(req[i]);
		else{
			transf_req = transform_request_memory(req[i]);
			req_to_send.push(transf_req);
		}
	}

	// Send message to all machine listeners without blocking
	for(int i from 0 to NUM_OF_MACHINES){
		if(i == my_rank) continue;
		else ISend(req_to_send[i]);
	}

	// Receive messages from size-1 machines with blocking
	for(int i from 0 to NUM_OF_MACHINES-1){
		MPI_Probe();
		recv_req = MPI_Recv();
		vector<Request*> curr =  translate_request(recv_req);
		result.add(curr);
	}

	// Wait for all machines to finish
	MPI_Barrier(MPI_COMM_WORLD);

	return result;
}
```

Third, each machine needs to receive NUM_OF_MACHINES-1 messages. Exactly the number of all machines except its own one since in the previous stage machines don't send messages to themselves. MPI_Probe gives us the MPI source of sender and number of bytes of message, crucial for its receiving. Next, the function *translate_request* translates the array of bytes into our Request structure, which is described in the next part. At the end, we are saving transformed messages into the result structure, the one which is the result of the *send_recv_mpi* function.

Fourth, all machines need to wait for each other with MPI_Barrier before continuing to the second stage of the machine listener.

4.1.2. Memory Transformation

4.1.2.1. Sending

We are not using any additional libraries that transform our structures to byte arrays and the opposite. We have created MPI sending/receiving memory transformations from scratch. Let's look once again how our Request structure looks like since it is important for the transform function:

4.1.2.a. Request structure

```
Request {
    int id;
    vector<Triple> init_constraints;
    vector<Triple> init_constraints_const;
    vector<vector<Triple*>> part_results;
};
```

Function *transform_request_memory* (Algorithm 4.1.2.b) transforms our structures into byte array, before MPI send part. In other words, it transforms a vector of requests into a pair of pointer and one integer, which represent a pointer to the byte array, and number of bytes of the complete transformed message.

First part is the calculation of size of the byte array - number of bytes to be sent. Second part is dynamical allocation of the byte array and writing all requests in it. The order of the writing is:

1. Number of requests
2. For each request
    - 2.1. Request id
    - 2.2. Size of init_constraints
    - 2.3. Size of init_constraints_const
    - 2.4. Size of part_results
    - 2.5. All information from init_constraints
    - 2.6. All information from init_constraints_const
    - 2.7. Part_results
        - 2.7.1. Size of part_results[i]
        - 2.7.2. All information from part_results[i]

Algorithm 4.1.2.b. Request Transform Function

```
pair<int*, int> transform_request_memory(vector<Request*> reqs){
       size_array = 1 + reqs.size* 4;
       size_array  += calculate_more_size();

       int_array = new int[size_array];
       *int_array++ = reqs.size;
       for (int num from 0 to reqs.size){
              *int_array++ = req->id;
              *int_array++ = req->init_constraints.size;
              *int_array++ = req->init_constraints_const.size;
              *int_array++ = req->part_results.size;

              for (int i from 0 to req->init_constraints.size){
                     *int_array++ = req->init_constraints[i].firstURI;
                     ...
              }
              for (int i from 0 to req->init_constraints_const.size){
                     ...
              }
              for (int i from 0 to req->part_results.size){
                     *int_array++ = req->part_results[i].size;
                     for (int j from 0 to req->part_results[i].size) {
                            *int_array++ = req->part_results[i][j];
                            ...
                     }
              }
       }
       return pair{start_array, size_array};
}
```

4.1.2.1. Receiving

The function *translate_request* (algorithm 4.1.2.b) represents the opposite function of the *transform_request_memory*. Input is a pointer to an array, and the output is the vector of requests. This function reads input, integer by integer, and fulfills the output in the same order described for the *transform_request_memory*.

Algorithm 4.1.2.b. Request Transform Function

```
vector<Request*> transform_request_memory(int* int_array){
	vector<Request*> reqs;
	num_of_reqs = *int_array++;

	for (num from 0 to num_of_reqs ) {
		req = new Request();
		req->id = *int_array++;
		init_constraints_size = *int_array++;
		init_constraints_const_size = *int_array++;
		part_results_size = *int_array++;

		for (i from 0 to init_constraints_size) {
			curr_triple = new Triple();
			curr_triple->firstURI = *int_array++;
			...
		}
		…
		reqs.push_back(req);
	}
	return reqs;
}
```

4.2. Second and Third stage of Machine Listener

Second and third stages have basically the same function (algorithm 4.2.a), just a different input. For the second stage, those are MPI messages received in the first stage. For the third stage, those are self messages. The function first gets the next destination of the request, which is calculated by the formula 3.1.0 from the next machine. The next machine depends on the case of the next triple constraint:

- Case 1: {X A B}. Subject is unknown and the next machine is the one on which is stored the object;
- Case 2: {A B X}. Object is unknown and the next machine is the one on which is stored the subject;
- Case 3: {A X B}. Predicate is unknown and the next machine is the one on which is stored the subject;

- Case 4: {X X A}. Subject and predicate are unknown and the next machine is the one on which is stored the object;
- Case 5: {A X X}. Predicate and object are unknown and the next machine is the one on which is stored the subject;
- Case 6: {X A X}. Subject and object are unknown and the next machine is the one on which is stored the predicate.

Algorithm 4.2.a. Query Translation Function

```
void translate_queries(vector<Request*> reqs) {
      for (int i from 0 to reqs.size) {
            Request* req =reqs[i];

            get_next_dest(req);
            req_case = get_case(req);

            if (req_case == 1) execute_query_c1(req);
            if (req_case == 2) execute_query_c2(req);
            if (req_case == 3) execute_query_c3(req);
            if (req_case == 4) execute_query_c4(req);
            if (req_case == 5) execute_query_c5(req);
            if (req_case == 6) execute_query_c6(req);

            if (req->init_constraints.size == 0)
                  print_results(req);
      }
}
```

4.2.1. Query execution by cases

The case of the current triple constraint is the one responsible for choosing function for execution. Let's say the current triple constraint is the one from case 1 ({X A B}). It chooses the function *execute_query_c1 (algorithm 4.2.b).* To be more specific, currently we are on the machine that is responsible for the information of the current triple constraint's object, B. First, between the machine's nodes we find a node that

corresponds to the object. If it is not found, it is an error. If it is found, then we find the responsible predicate (A) inside the object's predicates (types structure). Both of these searches have complexity O(1) since we are using hash maps. Since there can be multiple subjects with the predicate A and object B, we save them all and call the function *part_results_add* which makes the hash join of new and old results. Then we erase the current constraint from the request structure, and push the request to the responsible machine to be sent if there is the next destination (if the current triple constraint is not the last one).

Algorithm 4.2.b. C1 Query Execution Function

```
void execute_query_c1(Request* req) {
	vector<Triple*> curr_part_results;

	node = my_nodes.find(req->init_constraints[0].thirdURI);
	if(node){
		rel = node->types.find(
			req->init_constraints[0].secondURI);
		if(rel){
			for (int i from 0 to rel->in.size(); i++)
				curr_part_results.push_back(rel->in[i]);

			part_results_add(req, curr_part_res);
			erase_first_constraint(req);
			if (next_dest >= 0)
				reqs_for_sending[next_dest].
				push_back(req);
		}
	}
}
```

Other functions responsible for their cases (case two to case six) are very similar. They find the one known node or known predicate, and get the information from it about the unknown one.

4.2.2 Hash join of results

Function part_results_add is the main one responsible for joining the results. To be more specific, at this point we have two partial results, one gathered before execution of the current triple constraint - the old results, and one from the execution of the current triple constraint - the new results. If the current triple constraint is the first one, then we just add all new triples to the request's partial results structure. In the other case, we find constraints, the variables that are connecting previous triple constraints and the current ones. For example, in query 4.2.c if the current constraint is the third one, it's subject is connected to the subject of the first constraint, and it's object is connected to the subject of the second constraint. For this reason the algorithm needs the structure *init_constraints_const* (see 4.1.2.a) since the other one (*init_constraints*) is deleted dynamically, as previously described.

4.2.c. Query Example

```
REQUEST
-1 2 11
-2 2 12
-1 3 -2
END REQUEST
```

If there are no constraints, it means that in the current constraint triple is one or two new variables. In this case, final results are the ones from cartesian products of new and old partial results.

If constraints exist, the algorithm checks the validity of each old constraint inside each new triple. Let's see this algorithm with the query example 4.2.c (table 4.2.e).

Algorithm 4.2.d. Addition of Partial Results Function

```
void part_results_add(Request* req, vector<Triple*> curr_part_res) {
      if (first_pregel_superstep) {
            // Add all new triples to the req->part_results
            ...
      }
      else {
            constraints = find_conn_constr_part_results();

            if (!constraints) {
                  // Occurred new variable - Cartesian product of
                  // old and new partial results
                  ...
            }
            else {

                  for (int con from 0 to constraints.size) {
                        // create hashmap of old partial results
                        hashmap_old = create_hashmap_join(...);

                        // Do a hash join of new results and
                        // hashmap_old
                        ...
                  }
            }
      }
}
```

The first machine that receives this request is the machine which is responsible for URI 11. This machine takes the first triple constraint which has the number one case. The next destination is URI 12 from the next triple constraint. Moreover, the machine finds the node with URI 11 and then finds all subjects that are connected with URI 11 with a predicate with URI 2. In this example, those are URIs 101 and 102. At this point, the part_results_add function adds all new results (101 and 102) into the final results of the request structure, which is sent to the next destination in the first phase of the next superstep. The next Pregel's superstep, the second one, is the one on the machine responsible for URI 12. Similar to the first step, it is the case number one of the current, second, constraint. In the same way as in the first superstep, the current machine finds new URIs (201, 203, 204) that are connected to URI 12 with the predicate's URI 2. The part_results_add function sees that it is not the first superstep and

that there are no variables' connections between current triple constraint and the previous ones. Considerably, it creates the Cartasian product of old and new results, as can be seen on the table 4.2.e. The next step, third superstep, represents case 6, in which only predicate is known. The algorithm is executed in the machine responsible for URI 3, and it finds all subjects and objects that have connection with URI 3. Let's say the pairs for variables -1 and -2 are {101, 201}, {101, 204}, {102, 203}, {102, 205}, {105, 204}. In this case, part_results_add function understands that it is not the first superstep and that there are variables' connections between current constraint and the previous ones - the current subject is connected to the first constraint's subject and the current object is connected to the second constraint's subject. In this part, for each constraint the algorithm calls *create_hashmap_join* function (algorithm 4.2.f) and does comparison. The new results {101, 201}, {102, 203}, {101, 204} exist in the old ones, so they become the final results. Other old results are discarded, as the new results that are not found in the old results, like {102, 205} and {105, 204}. After the third superstep, there are no more triple constraints, and the execution of the program is finished, with three final results, as on the table 4.2.e.

URI	constr	constr2	case	new_res	final_res
11	-1 2 11 -2 2 12 -1 3 -2	-1 2 11 -2 2 12 -1 3 -2	C1	<101 2 11> <102 2 11>	<101 2 11> <102 2 11>
12	-2 2 12 -1 3 -2	-1 2 11 -2 2 12 -1 3 -2	C1	<201 2 12> <203 2 12> <204 2 12>	<101 2 11> <201 2 12> <102 2 11> <201 2 12> <101 2 11> <203 2 12> <102 2 11> <203 2 12> <101 2 11> <204 2 12> <102 2 11> <204 2 12>
3	-1 3 -2	-1 2 11 -2 2 12 -1 3 -2	C6	<101 3 201> <101 3 204> <102 3 203> <102 3 205> <105 3 204>	<101 2 11> <201 2 12> <101 3 201> <102 2 11> <203 2 12> <102 3 203> <101 2 11> <204 2 12> <101 3 204>

Table 4.2.e - Demonstration of the algorithm 4.2.d with the query 4.2.c

The function create_hashmap_join creates a hashmap of old results which is then used for hash joining of the old and new partial results. The input is the current request and the position of the triple which is the connection between current and old triple constraints. The function takes each triple from old partial results, takes the one value that is in the corresponding position (second argument of the function), and adds it's position into the hashmap. The key of the hashmap is the URI of the value on the *pos_triple* position of each old result. The value is the array (vector) of all indexes from old results where the key is. For example, the hashmap of the old results at the point of third superstep on the table 4.2.e, is:

- <101, {0,2,4}>, <102, {1,3,5}>.

This structure increase performance from O(N*M), where N is the size of new results and M is the size of old results, to O(N) since the selection of old results becomes O(1).

Algorithm 4.2.f. Creating of hashmap for results' joining

```
hashmap<int, vector<int>> create_hashmap_join(req, pos_triple) {
	hashmap<int, vector<int>> result;
	for(triple_iter from 0 to req->part_results.size) {
		curr_val = get_triple(req->part_results, pos_triple);

		map_iter = result.find(curr_val);
		if (map_iter)
			result[curr_val].push_back(triple_iter);
		else
			result[curr_val] = triple_iter;
	}
	return result;
}
```

4.3. Fourth stage of Machine Listener

Algorithm 4.3.a. is the main responsible for the fourth stage of machine listener. Each MPI machine executes the check_termination function. In the structure req_for_sending are saved messages that need to be sent in the next superstep. If there are no messages, then the stop flag remains true. The stop flag is used inside the MPI_ALLreduce function, the function from MPI that combines values from all processes and distributes the result back to all processes. The operation used in the MPI reduce function is a predefined reduction operation that will calculate the bitwise and of the values reduced. This means that the result is “true” only if all machines have the stop flag equal to “true”. To generalize, when there are no more messages to be sent from all MPI machines, the program stops - exits from the “while” loop (algorithm 3.3.1).

Algorithm 4.3.a. Program termination checking

```
void check_termination(bool stop) {
	stop = true;
	for (int i = 0; i < NUM_OF_MACHINES; i++)
		if (reqs_for_sending[i].size > 0) stop = false;

	bool stop_all;
	MPI_Allreduce(&stop, &stop_all, MPI_BAND ,MPI_COMM_WORLD);
	if (!stop_all)
		*stop = false;
}
```

5. Evaluation

For the evaluation of Nex's performance we needed a customizable generator of data, a set of test queries, and the results of those queries in order to check Nex's validity. Our approach was the Lehigh University Benchmark (LUBM).

LUBM is developed to facilitate the evaluation of Semantic Web repositories in a standard and systematic way. The benchmark is intended to evaluate the performance of those repositories with respect to extensional queries over a large data set that commits to a single realistic ontology. It consists of a university domain ontology, customizable and repeatable synthetic data, a set of test queries, and several performance metrics. Table 5.a represents details of different LUBM datasets. To be more specific, each dataset has its own number of universities used for the generation of data, and number of triples.

Furthermore, the LUBM has 14 test queries which cannot be used for evaluation of NeX since they perform reasoning. In addition to that, we used a set of 7 different test queries which are specifically crafted for those systems that do not perform reasoning. They can be found at Appendix A.

DATASET	UNIVERSITIES	TRIPLES
LUBM-1	1	103.076
LUBM-5	5	645.659
LUBM-10	10	1.316.342
LUBM-20	20	2.781.362
LUBM-50	50	6.888.742

Figure 5.a - Different LUBM datasets

For the purpose of the evaluation we used an Amazon EC2 M5zn instance which is one of Intel's fastest CPUs on AWS Amazon's cloud. The machine is powered by custom high frequency 2nd Generation Intel Xeon Scalable Processors (Cascade Lake). The instances run Ubuntu

Server 20.04.2 LTS with Linux 5.4.0 kernel and are configured to execute only a single thread per core.

Through the evaluation of NeX we found positive and negative conclusions:

1. Positive
    a. Parallel execution is faster than sum of all sequential executions,
    b. Parallel execution is linear with more processes,
    c. NeX is faster than Apache Jena.
2. Negative
    a. Sum of sequential execution is increasing with more processes.

5.1. Evaluation of parallel and sequential execution

Table 5.b shows the negative aspect of sequential execution with the different processes on LUBM-50. The rows are queries from Appendix A and the sum of all sequential queries by each machine as the last row, and the columns are processes - 1, 6, 12, 18, and 24. As the table shows, sequential execution of queries is not linear and is not scalable by addition of processes. Figure 5.c and 5.d represents the same aspect, just in different visual representations and with LUBM-20 and LUBM-50.

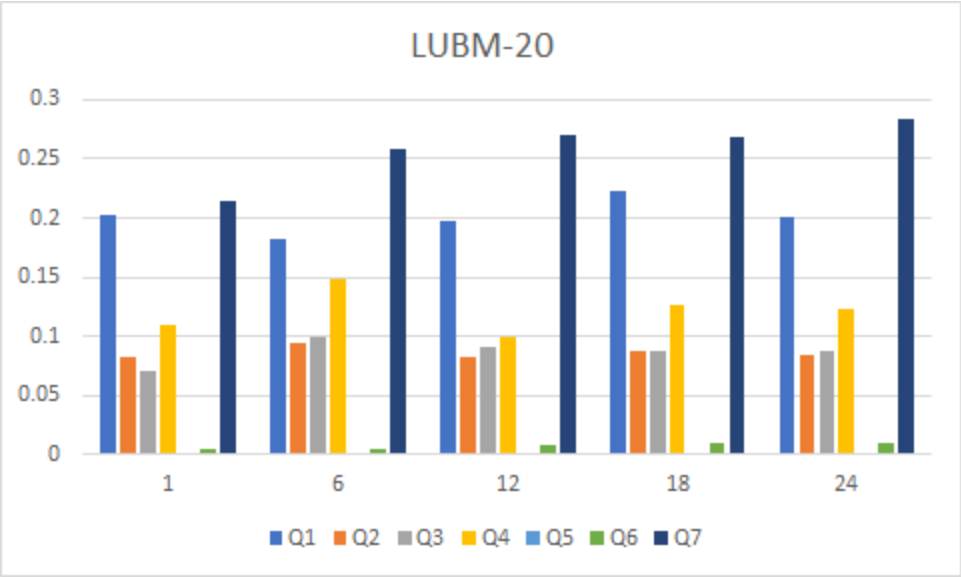


Figure 5.c - NeX sequential performance on LUBM-20 [in seconds]

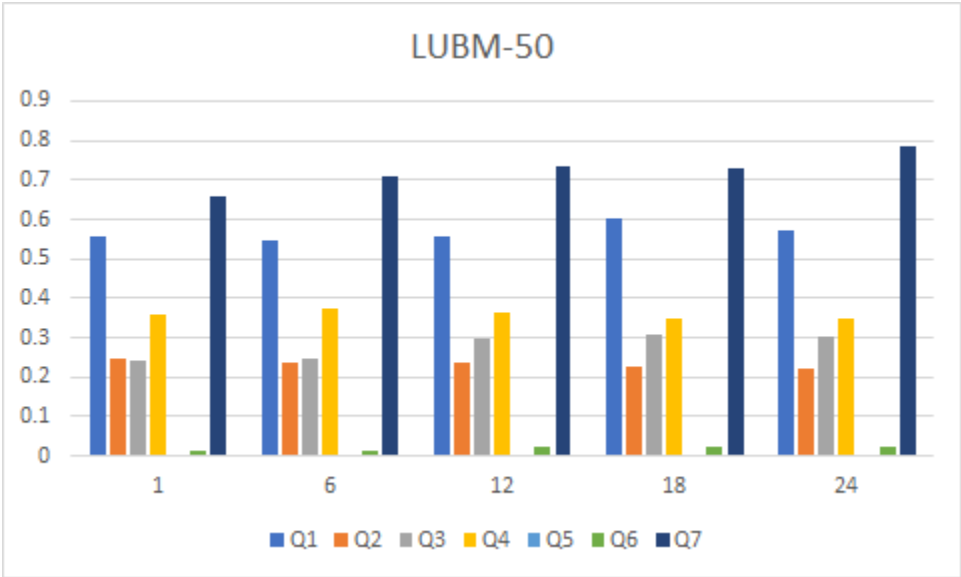


Figure 5.d - NeX sequential performance on LUBM-50 [in seconds]

	M1	M6	M12	M18	M24
Q1	0.559039	0.547532	0.555044	0.602572	0.569693
Q2	0.247672	0.237654	0.236108	0.227986	0.22317
Q3	0.24191	0.246839	0.298942	0.306494	0.300688
Q4	0.360661	0.375523	0.361883	0.346883	0.346159
Q5	0.001882	0.001577	0.001766	0.001778	0.001761
Q6	0.013252	0.013376	0.022239	0.025144	0.025247
Q7	0.659806	0.70761	0.734752	0.727066	0.786805
SUM	2.084222	2.130111	2.210734	2.237923	2.253523

Table 5.b - NeX sequential performance of all seven queries on LUBM-50 [in seconds]

The first positive conclusion is that the parallel execution of queries is faster than the sum of their sequential executions. The figures 5.e, 5.f, 5.g, and 5.h represent sequential and parallel performance of NeX on LUBM-5, LUBM-10, LUBM-20, LUBM-50. Moreover, the figures show scalability of parallel execution in comparison to the sequential one. This linearity of the parallel executions is an important difference for the systems that are independent and can extract data in parallel.

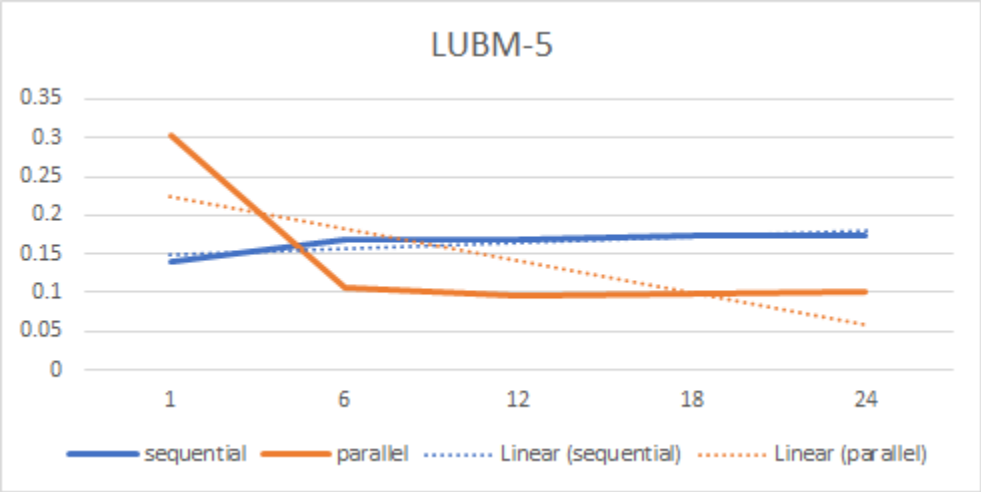


Figure 5.e - NeX sequential and parallel performance on LUBM-5 [in seconds]

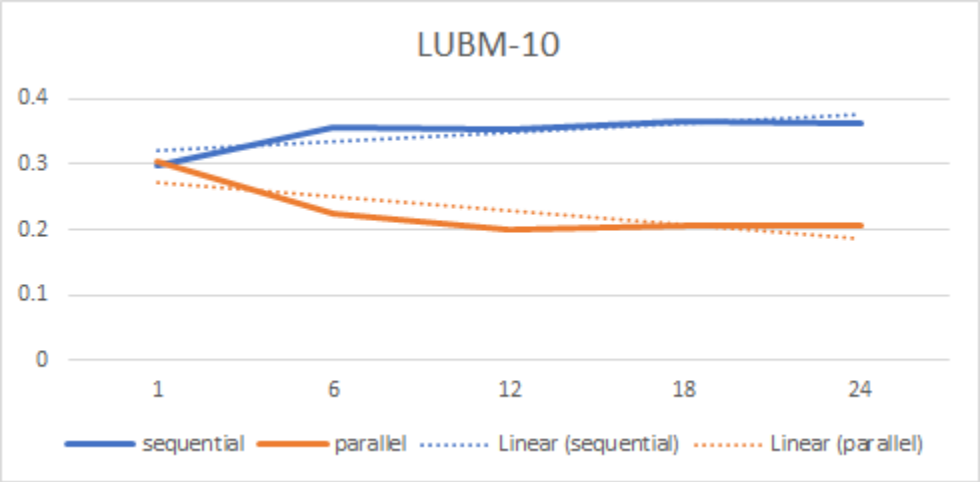


Figure 5.f - NeX sequential and parallel performance on LUBM-10 [in seconds]

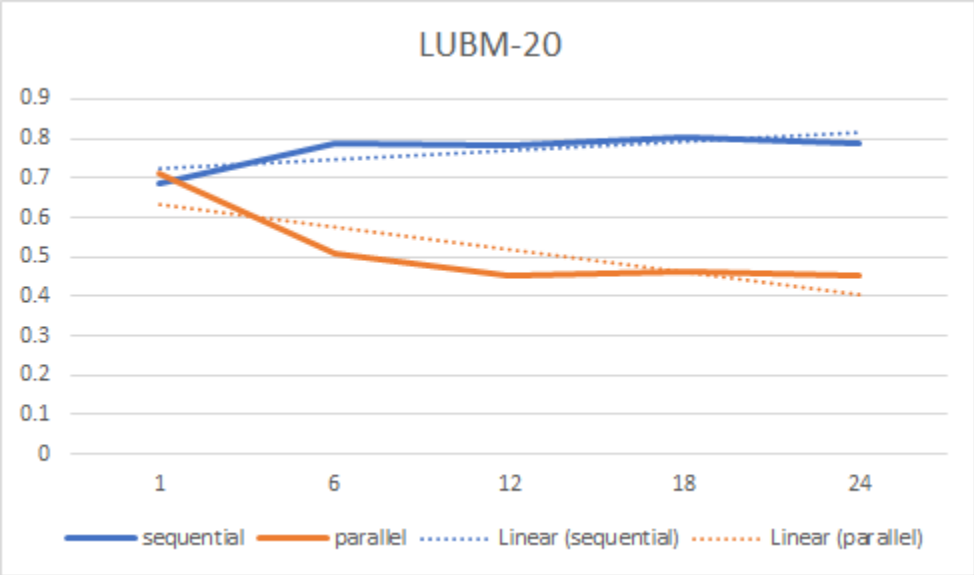


Figure 5.g - NeX sequential and parallel performance on LUBM-20 [in seconds]

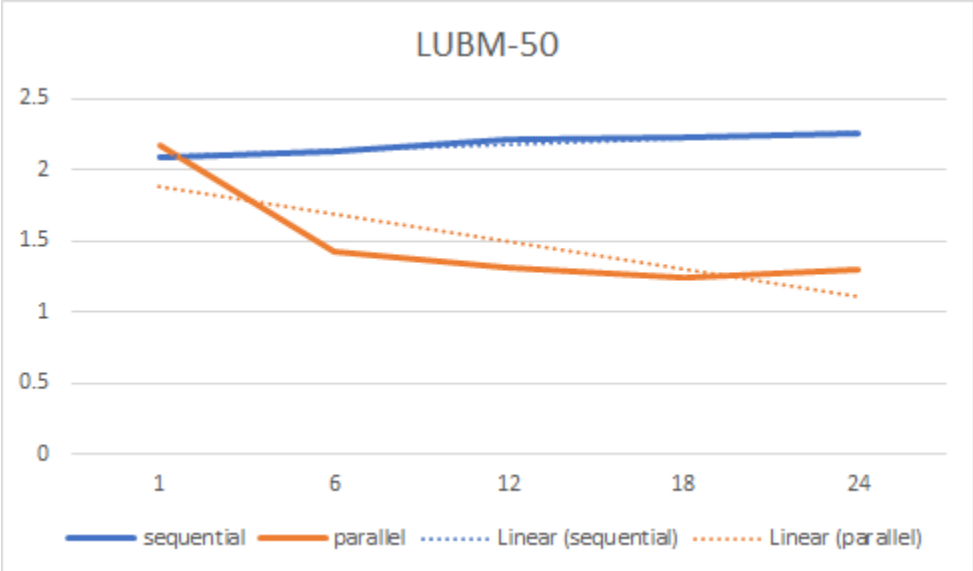


Figure 5.h - NeX sequential and parallel performance on LUBM-50 [in seconds]

5.2. Comparison with Apache Jena

For the purpose of comparing NeX to the already implemented systems, we chose Apache Jena.

Apache Jena is a free and open source framework for building semantic web and Linked Data applications, written in Java. The framework is composed of different APIs interacting together to process RDF data via SPARQL queries. A model can be sourced with data from files, Databases, URLs or combination of these.

For the testing we used the N-Triples format of LUBM. It is a line-based, plain text serialisation format for Resource Description Framework graphs, and a subset of the Turtle[5] (Terse RDF Triple Language) format. Code snippet 5.2.a represents an example of N-Triples format of one part of LUBM-1.

We have executed all seven queries on both systems, Apache Jena that uses one core, and NeX. We have fixed the number of processes on one for sequential use of queries' execution on NeX.

Code snippet 5.2.a: N-Triple format of partial LUBM-1

```
<http://www.Department0.University0.edu>
<http://www.lehigh.edu/~zhp2/2004/univ-bench.owl#subOrganizationOf>
<http://www.University0.edu> .
<http://www.University0.edu>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#University> .
<http://www.Department0.University0.edu/FullProfessor0>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.lehigh.edu/~zhp2/2004/univ-bench.owl#FullProfessor> .
<http://www.Department0.University0.edu/FullProfessor0>
<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#name>
"FullProfessor0" .
```

[5] https://www.w3.org/TeamSubmission/turtle/

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Apache Jena	0.2	0.059	0.31	0.025	0.022	0.032	0.181
Nex	0.033961	0.014343	0.014091	0.02032	0.000139	0.000826	0.045378
Speedup	83.0195	75.68983051	95.45451613	18.72	99.36818182	97.41875	74.92928177

Table 5.2.b - Comparison of NeX and Apache Jena on LUBM-5 [seconds]

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Apache Jena	0.314	0.099	0.583	0.025	0.022	0.033	0.281
Nex	0.082635	0.036349	0.032416	0.049143	0.000388	0.001979	0.096957
Speedup	73.68312102	63.28383838	94.43979417	-96.572	98.23636364	94.0030303	65.49572954

Table 5.2.c - Comparison of NeX and Apache Jena on LUBM-10 [seconds]

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Apache Jena	0.449	0.217	1.057	0.025	0.022	0.033	0.464
Nex	0.202234	0.082483	0.070634	0.109541	0.00082	0.00435	0.214868
Speedup	54.95902004	61.98940092	93.31750237	-338.164	96.27272727	86.81818182	53.69224138

Table 5.2.d - Comparison of NeX and Apache Jena on LUBM-20 [seconds]

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Apache Jena	0.973	0.308	2.649	0.024	0.022	0.032	1.037
Nex	0.559039	0.247672	0.24191	0.360661	0.001882	0.013252	0.659806
Speedup	42.544809 87	19.587012 99	90.867874 67	-1402.75 4167	91.445454 55	58.5875	36.373577 63

Table 5.2.e - Comparison of NeX and Apache Jena on LUBM-50 [seconds]

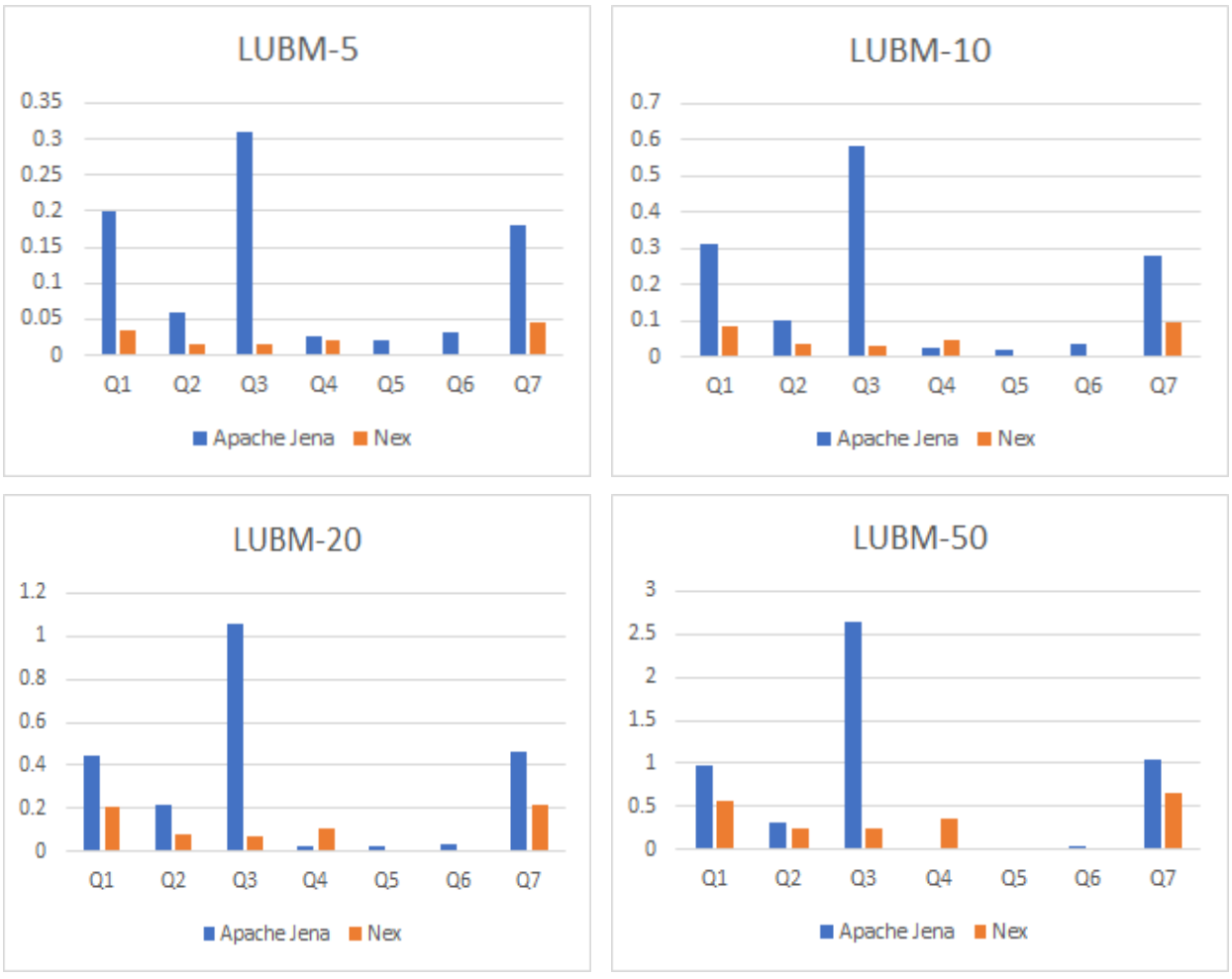


Figure 5.2.f - Comparison of NeX and Apache Jena on different LUBMs' [seconds]

Tables 5.2.b, 5.2.c, 5.2.d, and 5.2.e show response times of all seven queries on Apache Jena and Nex, in seconds. As it can be seen, queries 4, 5, and 6 have interesting response time on Apache Jena - a constant one that does not increase with the size of the dataset. Apache Jena probably performs some kind of optimization with indexing and reordering - starts execution immediately from the constant term of the query where the response time does not change with data increase.

Moreover, all queries, except query 4, are much faster on NeX than on Apache Jena. Figure 5.2.f represents this aspect visually - different visual representation of tables 5.2.b, 5.2.c, 5.2.d, and 5.2.e.

5.2.f. Formula for comparison of NeX and Apache Jena - speedup

$$speedup_q = (jena_q - nex_q) / jena_q * 100$$

Furthermore, to better represent comparison of NeX and Apache Jena, we have used the formula 5.2.f. Index “q” represents one of the seven queries. Nex’s percentage is “jena*(1-speedup)”, where nex’s time in seconds can be calculated by formula 5.2.g. As the figure 5.2.i shows, queries execution time of NeX increases with bigger datasets, but is still faster than Apache Jena’s execution. NeX is faster even with the LUBM-50 (6.888.742 triples).

5.2.g. Formula for calculation of NeX from Apache Jena and speedup

$$nex_q = jena_q * (1 - speedup_q) / 100$$

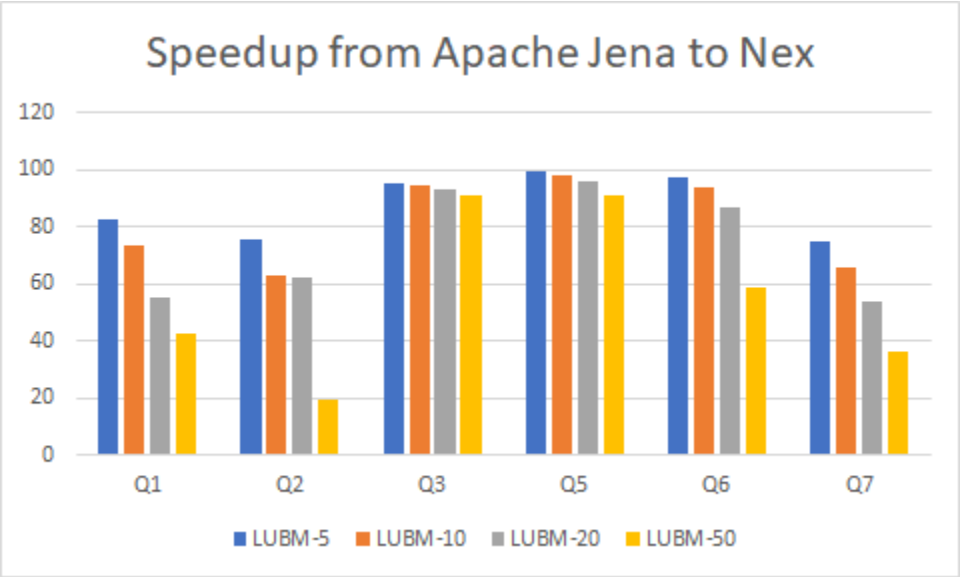


Figure 5.2.i - Speedup of NeX in comparison with Apache Jena on different LUBMs'

5.3. Comparison with Verne

Let's compare NeX with another vertex-centric SPARQL engine - Verne [15].

As previously mentioned, NeX is focused on index forwarding of subject, predicate, and object, while Verne is focused on subject forwarding and broadcasting when the subject is unknown. For this reason, NeX uses more memory, but always forwards requests to the machine that has useful information, without broadcasting to all machines in order to find some information.

Furthermore, NeX locally filters previous data with new ones on each superstep, while Verne does joining data from multiple MPI machines only when it needs a join.

The main differences of evaluations of NeX and Verne are:

1. Verne sequential execution is scalable with addition of more processes,
2. NeX has an opportunity to execute queries in parallel and has scalable parallel execution with addition of more processes, while Verne has only sequential execution,
3. NeX queries do not have limits - any order of triples is executable. Still, initial optimization of query is considered as an important improvement of performance.

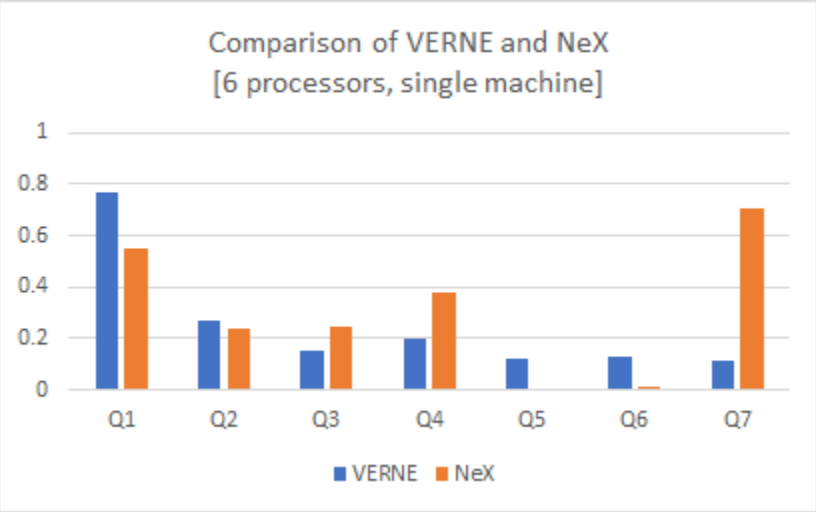


Figure 5.3.a - Comparison of VERNE and NeX for seven queries. Obtained by the same machine with 6 processors. [in seconds]

The figure 5.3.a. represents a comparison between VERNE and NeX on LUBM-50. Both systems were executed on the same machine. It can be seen that for some queries VERNE is faster, and for the others - NeX.

Even if it is not clearly evident which one of these two systems is faster, let's see the next figure, the figure 5.3.b. We have compared Sum of all seven queries sequentially executed on VERNE and NeX's parallel execution. VERNE needs 1.7579 seconds for the execution, while NeX needs 1.4234.

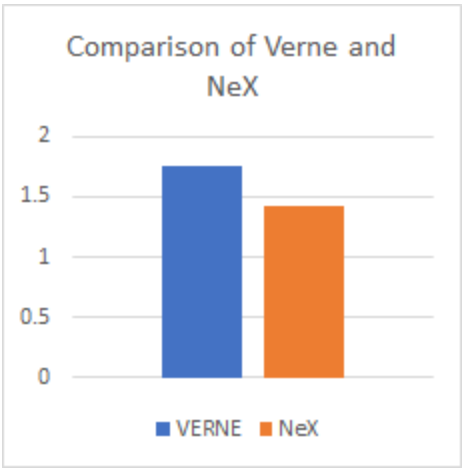


Figure 5.3.b - Comparison of sum of 7 queries on VERNE and their parallel execution on NeX. Obtained by the same machine with 6 processors. [in seconds]

6. Conclusions and Future work

In this work we introduced NeX, a distributed vertex-centric SPARQL query engine. With its foundations on C++ and MPI, NeX is able to answer complex queries efficiently. The main result from the evaluation of Nex is its parallel query execution, which is scalable with more processes (unlikely of sequential execution). Moreover, we have compared NeX with Apache Jena - one of the most common open-source RDF framework, and VERNE - one more distributed vertex-centric SPARQL query engine. Sequential execution time of NeX increases with bigger datasets, but is still faster than Apache Jena's execution. It is not clear if NeX or VERNE is faster, but parallel execution of queries on NeX is faster than the sum of their sequential execution on VERNE.

Through the development of NeX, we saw it has space for improvements and got some ideas to be implemented in the future.

Initial query optimization creates considerable difference at the NeX performance and may be done in the future. Firstly, triples that potentially have less results should be at the beginning of triple constraints since less information will be transferred by MPI and less time will be spent for partial results' filtering. Secondly, maximizing the number of different MPI machines at the same Pregel superstep in parallel execution increases the performance. In this case, one MPI machine will focus on processing less queries since the execution of queries will be balanced though machines.

Additionally, we have already spoken about different cases of triple constraints, from case 1 to case 6. One of future work can be even adding case 0, the case when a known subject, predicate, or object from any triple constraint, is stored on the current machine. In this case, more than one triple constraint can be executed in one Pregel superstep. For example, let's look at the query 3.2.1. If this example is executed on NeX with 4 MPI machines, and if it is the third Pregel superstep, not only the third triple constraint can be executed, but also the forth since URI 4 of the fourth constraint is on the same machine as the URI 12 from the third one - machine 0 (12 module of 4 and 4 module of 4).

One more possible future work of NeX would be mixing it with Verne (look section 5.3). In this case, we would need to choose when to use Verne's and when Nex's functionalities. One of possible solutions would

be declaring initial threshold and using subject, predicate, and object indexing when the number of result data is less than threshold. On the contrary, when the result data is bigger than the threshold, Verne's forwarding should be used.

BIBLIOGRAPHY

1. Tim Berners-Lee, James Hendler, Ora Lassila. The Semantic Web. In: Scientific American. 2001, page 1

2. Eric Prud'hommeaux, Andy Seaborne, SPARQL Query Language for RDF - World Wide Web Consortium., 2008

3. Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing, 2010

4. Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Ozsu, Xingfang Wang, Tianqi Jin. An Experimental Comparison of Pregel-like Graph Processing Systems. 2014, page 2

5. G. Navarro, J. L. Reutter, and J. Rojas-Ledesma. Optimal Joins Using Compact Data Structures. In International Conference on Database Theory (ICDT), pages 21:1–21:21. Schloss Dagstuhl, 2020

6. L. Zou, J. Mo, L. Chen, M. Tamer Özsu, and D. Zhao. gStore: Answering SPARQL queries via subgraph matching. PVLDB, 4(8):482–493, 2011

7. X. Lyu, X. Wang, Y. Li, Z. Feng, and J. Wang. GraSS: An Efficient Method for RDF Subgraph Matching. In Web Information Systems Engineering Conference (WISE), pages 108–122. Springer, 2015

8. P. Cappellari, R. D. Virgilio, and M. Roantree. Pathoriented keyword search over graph-modeled Web data. World Wide Web Conference (WWW), 15(5-6):631– 661, 2012

9. A. Harth and S. Decker. Optimized index structures for querying RDF from the Web. In Latin American Web Congress (LA-WEB), pages 71–80. IEEE, 2005

10. O. Erling and I. Mikhailov. Virtuoso: RDF Support in a Native RDBMS, pages 501–519. Springer, 2010

11. M. Galkin, K. M. Endris, M. Acosta, D. Collarana, M. Vidal, and S. Auer. SMJoin: A Multi-way Join Operator for SPARQL Queries. In International Conference on Semantic Systems (SEMANTICS), pages 104–111. ACM, 2017

12. O. Erling and I. Mikhailov. Virtuoso: RDF Support in a Native RDBMS, pages 501–519. Springer, 2010

13. K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in Jena2. In International Conference on Semantic Web and Databases (SWDB), pages 120–139. CEUR, 2003

14. Message Passing Interface Forum. A Message-Passing Interface Standard Version 3.1, June 4, 2015

15. Fabio Codiglioni. Verne: a vertex-centric SPARQL engine, 2021

Appendix A

All queries for Apache Jena have the same prefixes:

```
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub:   <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
```

- LUBM Query 1 for Apache Jena

```
SELECT ?x ?y ?z
WHERE { ?z ub:subOrganizationOf ?y .
       ?y rdf:type ub:University .
       ?z rdf:type ub:Department .
       ?x ub:memberOf ?z .
       ?x rdf:type ub:GraduateStudent .
       ?x ub:undergraduateDegreeFrom ?y . }
```

- LUBM Query 1 for Nex

```
REQUEST
-1 2 2009
-1 17 -2
-1 411 -3
-2 2 3
-3 9 -2
-3 2 7
END REQUEST
```

- LUBM Query 2 for Apache Jena

```
SELECT ?x
WHERE { ?x rdf:type ub:Course .
       ?x ub:name ?y . }
```

- LUBM Query 2 for Nex

```
REQUEST
-1 2 3063
-1 4 -2
```

```
END REQUEST
```

- LUBM Query 3 for Apache Jena

```
SELECT ?x ?y ?z
WHERE { ?x rdf:type ub:UndergraduateStudent.
       ?y rdf:type ub:University .
       ?z rdf:type ub:Department .
       ?x ub:memberOf ?z .
       ?z ub:subOrganizationOf ?y .
       ?x ub:undergraduateDegreeFrom ?y . }
```

- LUBM Query 3 for Nex

```
REQUEST
-1 17 -2
-1 2 409
-1 411 -3
-3 9 -2
-2 2 3
-3 2 7
END REQUEST
```

- LUBM Query 4 for Apache Jena

```
SELECT ?x
WHERE { ?x ub:worksFor
<http://www.-Department0.University0.edu> .
       ?x rdf:type ub:FullProfessor .
       ?x ub:name ?y1 .
       ?x ub:emailAddress ?y2 .
       ?x ub:telephone ?y3 . }
```

- LUBM Query 4 for Nex

```
REQUEST
-1 23 6
-1 4 -2
```

```
-1 24 -3
-1 26 -4
-1 2 11
END REQUEST
```

- LUBM Query 5 for Apache Jena

```
SELECT ?x
WHERE {
?x                                          ub:subOrganizationOf
<http://www.Department0.University0.edu> .
               ?x rdf:type ub:ResearchGroup. }
```

- LUBM Query 5 for Nex

```
REQUEST
-1 9 6
-1 2 2571
END REQUEST
```

- LUBM Query 6 for Apache Jena

```
SELECT ?x ?y
WHERE { ?y ub:subOrganizationOf <http://www.University0.edu> .
       ?y rdf:type ub:Department .
       ?x ub:worksFor ?y .
       ?x rdf:type ub:FullProfessor . }
```

- LUBM Query 6 for Nex

```
REQUEST
-1 2 11
-1 23 -2
-2 9 1
-2 2 7
END REQUEST
```

- LUBM Query 7 for Apache Jena

```
SELECT ?x ?y ?z
WHERE { ?y ub:teacherOf ?z .
       ?y rdf:type ub:FullProfessor .
       ?z rdf:type ub:Course .
       ?x ub:advisor ?y .
       ?x rdf:type ub:UndergraduateStudent .
       ?x ub:takesCourse ?z . }
```

- LUBM Query 7 for Nex

```
REQUEST
-1 426 -2
-1 2 409
-2 2 11
-1 413 -3
-3 2 3063
-2 13 -3
END REQUEST
```