A Parallel Processing Framework for RDF Design and Issues

Paolo Castagna, Andy Seaborne, Chris Dollin

HP Laboratories
HPL-2009-346

Keyword(s):
rdf, parallel processing

Abstract:

This document describes the hardware and software architecture of a parallel processing framework for RDF data. The aim of the framework is to simplify the implementation of parallel programs and to support execution over cluster of off-the-shelf machines. From a scripting language, logical execution plans are constructed and then compiled into physical execution plans which are scheduled for execution in parallel over a cluster. At runtime, processing elements simply exchange data as streams of tuples. We are not pioneers in this field; however, not many of the existing systems support parallel RDF data processing.

External Posting Date: October 21, 2009 [Fulltext] Approved for External Publication
Internal Posting Date: October 21, 2009 [Fulltext]

© Copyright 2009 Hewlett-Packard Development Company, L.P.

A Parallel Processing Framework for RDF
Design and Issues

Paolo Castagna, Andy Seaborne, Chris Dollin

Hewlett-Packard Laboratories, Long Down Avenue, Stoke Gifford, Bristol BS34 8QZ, UK

ABSTRACT

This document describes the hardware and software architecture of a parallel processing framework for RDF data. The aim of the framework is to simplify the implementation of parallel programs and to support execution over cluster of off-the-shelf machines. From a scripting language, logical execution plans are constructed and then compiled into physical execution plans which are scheduled for execution in parallel over a cluster. At runtime, processing elements simply exchange data as streams of tuples. We are not pioneers in this field; however, not many of the existing systems support parallel RDF data processing.

1. INTRODUCTION

Internet companies such as Google [17], Yahoo! [16], Microsoft, Facebook, Amazon and Last.fm have a daily need to deal with massive (i.e. GB or TB) datasets which are generated, directly or indirectly, by users interacting with their systems. Once data is collected, there is the need to mine it and analyze it. Because of the size of these datasets relational database solutions are often too expensive; therefore we have witnessed the rise of custom solutions to store and process such large amounts of data.

A similar situation exists in the context of the semantic web as the amount of data available in RDF format grows to outpace the capacity of current storage systems, also known as triple stores. Once people start to expose datasets in machine readable format, RDF provides a reasonable data model and we expect the amount of data available to grow quickly because often the data is automatically generated from existing data sources.

Examples of this trend are: DBPedia[1], UniProt[2], GeoNames[3], GeneOntology[4], Freebase[5], etc. or, more generally, initiatives such as LinkedData[6]. Moreover, the number of HTML pages using Microformats[7], GRDDL[8] or RDFa[9] is growing and the growth accelerates as content management systems integrate those technologies.

So called web 2.0 mashups are a widespread example of what a web of data can achieve in terms of functionalities. However, current mashup technologies do not help developers to easily aggregate data from many sources: each website has its own data model, serialization format and APIs. Mashups do not cascade: once the data has been aggregated, it usually cannot be reused for other mashups. There is no common data model and query language in the web 2.0 universe.

[1] http://dbpedia.org/

[2] http://www.uniprot.org/

[3] http://www.geonames.org/

[4] http://www.geneontology.org/

[5] http://rdf.freebase.com/

[6] http://linkeddata.org/

[7] http://microformats.org/

[8] http://www.w3.org/TR/grddl/

[9] http://www.w3.org/TR/rdfa-syntax/

Overall, incentives for people to publish more structured data are beginning to emerge, such as Yahoo!'s SearchMonkey[10], Google's Rich Snippet[11], the Data Portability[12] initiative and semantic sitemaps[13].

In the context of semantic web, SPARQL [18] is the standard way to query RDF data, while logic inference using rules is one way to derive new data from known facts (ref. RDF Schema [19], OWL [20] and OWL 2 [21]). However, these are not the only means to process data in RDF. Often data needs transforming from one format to another, it requires cleaning and validation or statistical information needs extracting.

Solutions that vertically scale using a single machine to process a growing amount of data quickly become infeasible as the amount of data grows. One possibility to achieve significant improvements is to scale horizontally by distributing the workload over a cluster of machines and use them concurrently to do parallel processing.

This document describes the direction we are heading and some of the challenges we have met in designing and implementing a parallel processing framework for processing RDF datasets using a cluster of off-the-shelf machines.

The aim is to provide developers with an extensible framework which can be easily used to process large amount of RDF data. The direction we are heading is to design a scripting language which can be used to construct logical execution plans. The framework provides a library of processing elements, but developers can extend it by adding user-defined processing elements. Logical execution plans are then automatically compiled into physical ones that are scheduled to execute in parallel over the cluster. At runtime, processing elements simply exchange data as streams of tuples. We are neither the first nor alone in this effort. However, few of the existing systems support RDF datasets and use cases.

The rest of this report is organized as follows. Section 2 briefly discusses related work. Section 3 introduces four motivating examples for parallel processing. Section 4 describes the high-level hardware and software architecture for the framework. Finally, Section 5 provides our conclusions so far.

2. RELATED WORK

Extensive research on parallel processing has been carried out. Only recently, however, driven by the huge increase in dataset sizes and a divergence in Moore's law, we have witnessed the development of massively distributed scalable hardware and software infrastructures.

MapReduce [6], the parallel programming model proposed and used internally by Google, quickly became the state-of-the-art for embarrassingly parallel workloads which involve a complete scan over large datasets. Open source implementations, such as Hadoop[14], are available and other projects use Hadoop/MapReduce clusters as execution engines for their processing.

Alongside MapReduce, parallel databases [29] represent the state-of-the-art for workloads which often involve retrieving just a fraction of the entire data. In this case, a complete scan over a large dataset seems unnecessary and the existence of pre-computed indexes can make a big difference in terms of performance. However, in this scenario knowledge of the schema of the data, as well as insights on access pattern, is necessary in order to pre-compute the necessary indexes. In this regard, the *schema-less* approach assumed by RDF storage systems can be advantageous as it allows building complete indexes. Moreover, the MapReduce programming model does not exclude the adoption of indexes and, indeed, could benefit from them; in particular for early stages of processing when relevant data is selected.

Dataflow programming languages and flow based programming systems have been adopted and implemented for different domains, ranging from signal/image processing to digital circuit design. One of the initial requirements for dataflow programming languages was to make parallel programming easier. According to the dataflow principles and architecture, a program is modeled as a directed acyclic graph of operators interconnected according to data dependencies. Recursive and iterative algorithms do not naturally fit into this model, but this is balanced by explicit declaration of dependencies between operators which allows simplifying automatic parallelism at compile time.

Even if the research on these broad areas is driven by different communities, the borders are becoming more and more blurred. On one hand, researchers and developers are trying to leverage or extend the MapReduce programming model to

[10] http://developer.yahoo.com/searchmonkey/

[11] http://www.google.com/support/webmasters/bin/answer.py?hl=en&answer=99170

[12] http://www.dataportability.org/

[13] http://sw.deri.org/2007/07/sitemapextension/

[14] http://hadoop.apache.org/

better support joins, complex data flows and queries. Apache Pig [12][13], Apache Hive[15], TupleFlow[16], Pervasive DataRush[17] and Cascading[18] are examples of these efforts. On the other hand, some parallel databases [25][[26]][[27] have embraced the MapReduce programming model and are trying to simplify the addition of user-defined functions to extend the processing capabilities of their systems.

While the debate [30] among researchers is still ongoing, there are efforts to unify these different approaches and provide more general solutions, for example Microsoft Dryad [11] or Clustera [22]. These are generic dataflow processing engines for developing and executing coarse-grain parallel applications. Currently, Dryad is only available internally to Microsoft. Similarly, Clustera is a research project at the University of Wisconsin-Madison and it is not available for broad usage.

From the point of view of developers - the end users of these solutions - many scripting languages and development environments have been proposed. Here, we want to focus on the ones that in our opinion are more relevant, even if they do not all currently offer a parallel runtime environment.

SPARQLScript[19] and SPARQLMotion[20] are two efforts trying to extend or use SPARQL query language and SPARQL Update[21] to process RDF data. While SPARQLScript aims at minimizing changes to the SPARQL syntax, SPARQLMotion aims at a visual representation and assumes a model-driven approach, using OWL to represent the wiring between processing modules. Other similar visual approaches are Yahoo! Pipes[22] and, in the context of RDF data, DERI Pipes[23].

Another interesting direction has been proposed by Microsoft with Language Integrated Query (LINQ) for their .NET framework. This solution tries to eliminate the impedance mismatch between imperative object oriented programming languages, such as C# or VB.NET, and declarative query languages, such as SQL. It is worth noting that a parallel implementation (PLINQ) is planned[24] for .NET framework 4.0.

None of the parallel processing solutions described so far is explicitly designed for RDF data, although some could be adapted or extended to process RDF datasets. MapReduce can be used over RDF datasets[25]. However, processing RDF often involves integration of multiple datasets, joins [31] and other operations that are not naturally supported by the MapReduce model.

3. PARALLEL PROCESSING

The number and size of RDF datasets is growing. This growth demands scalable systems capable of storing, querying, inferencing and processing these datasets.

While it is possible to adapt current solutions by improving their efficiency and increasing the physical resources of a single machine, horizontal scalability (a.k.a. scale out) can offer significant improvements. However, using a large number of off-the-shelf machines causes new challenges in terms of management costs, fault tolerance and complexity of the programming model.

The parallel processing framework described in this paper assumes a shared-nothing architecture over a cluster of off-the-shelf machines. Once an RDF dataset is scattered across multiple machines, the disks, memory and processors of those machines can be used to process the data in parallel, hopefully achieving close to linear speed-up and scale-up behaviors as the number of machines and the amount of data to process grow.

[15] http://mirror.facebook.com/facebook/hive/

[16] http://www.galagosearch.org/

[17] http://www.pervasivedatarush.com/

[18] http://www.cascading.org/

[19] http://arc.semsol.org/docs/v2/sparqlscript

[20] http://www.topquadrant.com/sparqlmotion/

[21] http://www.w3.org/Submission/2008/SUBM-SPARQL-Update-20080715/

[22] http://pipes.yahoo.com/

[23] http://pipes.deri.org/

[24] http://msdn.microsoft.com/en-gb/magazine/cc163329.aspx

[25] It is particularly easy if RDF is serialized in N-Triples format

The following sections describe the motivating examples which have been driving the design of the parallel processing framework for RDF. On the whole, the scenario is a batch processing one and the processing involves inspection of the totality (or a large proportion) of the dataset. We assume that the access patterns are not known in advance, indexes over the data might not be available and users can provide *user-defined functions* to extend the framework.

3.1 Motivating Examples

3.1.1 Co-authorship Network

The UniProt RDF dataset[26] contains about 477,000 *literature citations*[27] (the term UniProt uses for papers).

As an example, George wants to know who knows who in the community of researchers publishing papers about data contained in UniProt. He would like to understand if there is correlation between being collaborative and being a prolific writer for that particular group of authors. As initial step for his investigation, George wants to identify the 20 most prolific co-author couples, that is who wrote the highest number of papers together.

We assume if two authors wrote a paper together they know each other and the *know* relationship is symmetric. To satisfy George's needs, we will need to extract a network of co-authors and count how many papers each couple wrote together.

Within the UniProt dataset a paper is represented using the Turtle syntax and these prefixes: core=http://purl.uniprot.org/core/, pubmed=http://purl.uniprot.org/pubmed/, rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns# and dc=http://purl.org/dc/elements/1.1/:

```
pubmed:22177139
    rdf:type uniprot:Journal_Citation ;
    core:title "The chloroplast and mitochondrial genome sequences [...]" ;
    core:author "Turmel M." ;
    core:author "Otis C." ;
    core:author "Lemieux C." ;
    core:date "2002" ;
    core:name "Proc. Natl. Acad. Sci. U.S.A." ;
    core:volume "99" ;
    core:pages "11275-11280"
    dc:identifier "doi:10.1073/pnas.162203299" ;
    .
```

For the sake of processing, it is often easier to work with RDF data serialized into N-Triples[28] format which is easier to parse and it can be easily split if necessary.

The required processing to produce the answer George needs can be described as follows. Reading one RDF triple at the time, all the authors need to be grouped by document. Then, for each document, all the couples of co-authors need to be generated and properly counted. Finally, co-authors couples need to be sorted by their count and the top 20 couples reported back to George.

It is out of scope of this report to discuss modeling or provide solutions to disambiguate names.

3.1.2 PageRank Over a Citation Network

CiteSeer[29] is a well known service that crawls and indexes papers in computer science available on the Web. Amongst other information, CiteSeer extracts citation information from the references of each paper. The CiteSeer dataset is available[30] via the Open Archive Initiative (OAI) Protocol for Metadata Harvesting (PMH) and as a compressed file to download. The dataset contains about 700,000 papers and about 1,700,000 citations.

The new version of CiteSeer, called CiteSeerX[31], claims to index 1,366,867 papers and 26,435,805 citations. However the dataset is available exclusively via OAI and not as a compressed file to download. We have tried to harvest the content, but failed due to malformed XML on many of the records. We decided that, as a motivating example, the old CiteSeer dataset was good enough for us.

[26] http://dev.isb-sib.ch/projects/uniprot-rdf/

[27] http://www.uniprot.org/citations/?query=*

[28] http://www.w3.org/TR/rdf-testcases/#ntriples

[29] http://citeseer.ist.psu.edu/

[30] http://citeseer.ist.psu.edu/oai.html

[31] http://citeseerx.ist.psu.edu/

PageRank is a well known ranking function that can be computed over a network of interlinked web pages. The network formed by citations between papers is not dissimilar from the Web, so it might be interesting to see the result of applying PageRank over the papers indexed by CiteSeer to identify the 20 (or 100) most (or less) *interesting* papers in computer science according to PageRank. Other researchers have tried a similar approach, which they called CiteRank [40], over Physics papers with interesting results.

Although it is possible to implement PageRank or CiteRank with a serial algorithm and, using efficient compression techniques, vertically scale-up as the number of links grows, in the context of this motivating example we decided to focus on parallel solutions.

It is possible to represent the graph of citations between papers using an adjacency list. Each row in the adjacency list is a tuple. The first field of each tuple represents a paper, while the subsequent fields are the papers linked from the current paper. The initial dataset can be partitioned and the processing progresses one tuple at the time.

It is worth noting PageRank and similar iterative algorithms require multiple iterations, eventually with convergence check points, therefore the overall processing network may contain loops.

3.1.3 LUBM

In the context of semantic web, people often need to apply a specific type of processing called *inference* over their datasets. With rule based inference, users express processing as a set of rules. In a forward inference engine, the application of a rule is triggered whenever there is some data (typically one or more RDF triples) that matches the premise of the rule. When a rule is applied, conclusions (i.e. one or more RDF triples) are generated and, eventually, added to the initial data. Conclusions may trigger the application of other rules.

The Lehigh University Benchmark (LUBM) is a proposed benchmark to evaluate systems which store large datasets and use a moderately complex OWL ontology. The *Univ-Bench*[32] ontology is an OWL Lite ontology which describes the university domain. The dataset is synthetically generated and the benchmark describes fourteen test queries over the data and their expected results.

Given as input a ruleset and a large RDF dataset, our problem is to apply those rules in a forward mode computing all the possible deductions to be added to the initial dataset. In practice, this means that for each RDF triple in our dataset we need to check if there is any rule that should be triggered and if there is, the RDF triples stated by the conclusions should be added to the initial dataset.

Our premise is forward inference could be performed using multiple machines in parallel to achieve an order of magnitude improvement in overall performance as the number of machines is increased.

3.1.4 The Wine Ontology

The OWL recommendation uses a wine ontology[33] to show examples throughout their document. The wine ontology is one of the classical examples for OWL DL reasoning. It is a small ontology and the reasoning is limited to TBox (i.e. terminological box) reasoning with no ABox (i.e. assertional box).

We intend to investigate why inference engines based on rule processing tend not to scale well with the wine ontology and see whether parallel processing offers opportunities to improve this.

4. ARCHITECTURE AND DESIGN

The main objective of our parallel processing solution for RDF is to facilitate the writing of parallel programs utilizing off-the-shelf machines ideally to linearly scale-up the processing.

Figure 1 – High-level Hardware Architecture provides a high-level view of the hardware architecture. The only assumption we make on the hardware architecture is that all the machines in the cluster are collocated (i.e. they are in the same datacenter) and 1 or 10 gigabit Ethernet connections are available between machines. Each machine in the cluster has its own processors, memory and disks.

[32] http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl

[33] http://www.w3.org/TR/owl-guide/wine.rdf

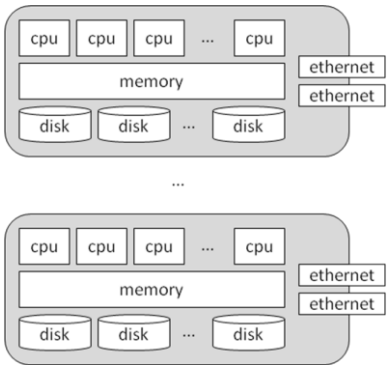

Figure 1 – High-level Hardware Architecture

Figure 2 provides a high-level view of the software architecture and the relation between the principal components. For each software component the data structures they take as input and generate as output are represented using white boxes.

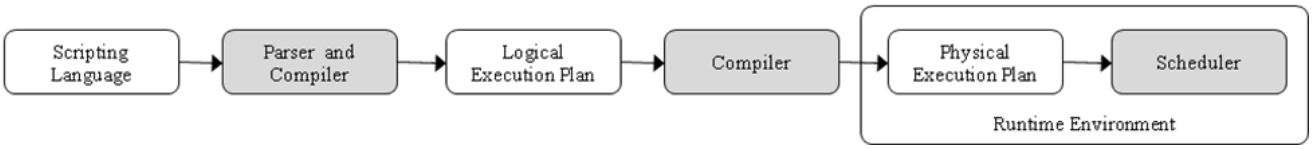

Figure 2 – High-level Software Architecture

The script provided by a developer is parsed and compiled into a network of logical processing elements: the directed edges between nodes represent data dependencies between nodes. We call this network a *logical execution plan*: it does not contain any notion of parallelism and it is agnostic with respect to the physical properties of the cluster (i.e. how many machines, how many processors per machine or how many disks per machine).

A logical execution plan is translated by a compiler into what we call a *physical execution plan*. In a physical execution plan the parallelism is explicitly stated as well as the algorithms for each of the physical processing elements in the plan. The same logical execution plan might be compiled into different physical execution plans on different clusters. In other words, a logical execution plan is portable across different clusters, while a physical execution plan is not.

A physical execution plan is given as input to a scheduler whose role is to allocate physical processing elements to machines in the cluster and orchestrate the execution. While the compiler needs to be aware of the static properties of the cluster, the scheduler might benefit from knowing dynamically the utilization of the cluster (in terms of utilization of disks, memory, CPU and bandwidth).

The following sections describe the details of the processing framework. First, the data model and the scripting language for the processing are described. Then, the following sections explain how the logical execution plans might be compiled into physical ones and, finally, how physical execution plans might be scheduled for execution.

4.1 Processing Data Model

An RDF graph is a set of triples. An RDF triple has three components:

- a subject, which is an RDF URI reference or a blank node,
- a predicate, which is an RDF URI reference,
- a object, which is an RDF URI reference, a literal or a blank node.

One issue to consider is the granularity of the unit of data to process. Figure 3 shows the range of possibilities: from RDF triples to RDF datasets. The granularity level for the unit of processing is an important design point and it has implications on how data is partitioned and transmitted between processing elements.

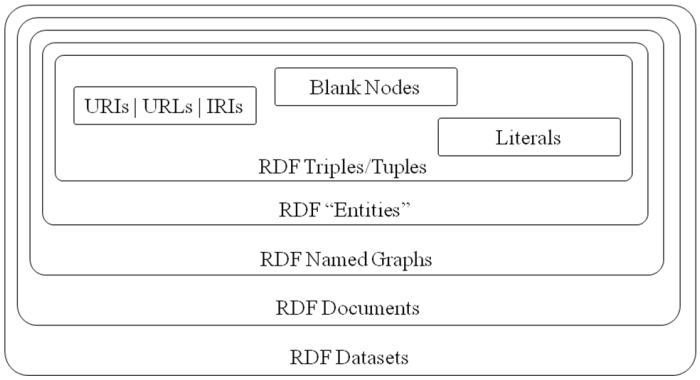

Figure 3 - Granularity levels for the unit of processing

4.1.1 Tuples and Streams

A tuple is an ordered and finite list of values of a specified type. A tuple can contain the same value more than once, so it is a list rather than a set.

Processing elements exchange data as sequences (a.k.a. *streams*) of tuples. Each tuple might have an associated schema. The schema of incoming tuples can be different from the schema of outgoing tuples. The schema usually does not change for a given stream, however we do not assume this.

Special signaling tuples are used to mark the beginning or end of a stream, or to communicate downstream metadata about the stream itself. O*ut-of-band* signals are also possible. A stream of tuples can be fully sorted, grouped or partially sorted or unsorted. Knowing and propagating this information about streams is an important factor for the physical execution plan generation process.

Tuples are serialized over the wire as a sequence of *tagged fields*. A possible optimization is to communicate the schema of tuples as associated metadata together with the tuple that signals the beginning of a stream. Another possible optimization is to apply compression techniques, as suggested in [33], such as: run length encoding, delta encoding and variable byte encoding.

Moreover, the support for URIs, blank nodes and literals (un-typed, typed and tagged with different languages) is necessary in the context of RDF.

Finally, we notice that when dealing with parallel processing, partitioning might happen and blank nodes need to be treated appropriately. Usually, blank nodes are represented with some sort of id that is unique, only in relation to a particular group of triples. For example, it might happen RDF documents are divided and their parts are processed at different machines; blank nodes ids might collide. Using UUIDs to represent and transmit blank nodes could avoid some of these problems.

4.1.2 RDF Entities

RDF triples or tuples are not the only choice available as unit of processing. For example, a developer might need to process all the instances of a certain rdf:type along with all the "relevant" RDF triples. We shall call this RDF subgraph an *RDF entity*.

Different definitions have been proposed to specify the notion of RDF entity: RDF molecules [35], Concise Bounded Description (CBD) [36] and Minimum Self-Contained Graphs (MSGs) [37]. Moreover, SPARQL has a syntactic hook (i.e. DESCRIBE) to accommodate implementations of these definitions in a query engine.

From a point of view of parallel processing, grouping RDF triples into entities is a way to partition the dataset. Partitioning is an essential operation to enable data parallelism and is particularly beneficial in the case of IO bounded type of processing.

The right level of granularity and the right RDF entity definition is determined by a particular application. Therefore, it would be desirable to have flexibility on the way RDF triples get partitioned or grouped.

Unfortunately, algorithms to compute RDF entities are recursive algorithms or involve multiple iterations and state needs to be maintained between each interaction. Iterative algorithms can be an issue for parallelism.

Supporting RDF entities is an important open issue to be solved and an RDF processing framework should be able to repartition a dataset according to different and customizable definitions of RDF entities.

4.1.3 Granularity Levels

RDF triples and RDF entities are just two of the lowest level of granularity that can be used to process RDF data.

As shown in Figure 3, RDF triples or entities can be grouped into named graphs[34]. Multiple named graphs could be serialized into an RDF document or grouped together into an RDF dataset[35].

Moreover, multiple RDF datasets and documents could be grouped together into a corporate or local *knowledge base*.

Finally, a conceptual universal RDF graph includes the union of all RDF known (or available) triples.

No matter which level of granularity is chosen, RDF entities, named graphs, datasets, etc. can be seen as a way to group triples. Group of triples (or tuples) can be serialized, one after the other, to a stream. A group of triple at each level of granularity could be considered a unit of processing and transmitted as partially sorted stream of triples.

A processing framework can partition data at each granularity levels and units at each level might be processed in parallel. However, it is worth noting that this is not possible for all type of processing and sometimes a different partitioning scheme is required.

4.2 Scripting Language

Writing parallel applications involves two intertwined but different tasks:

- defining and implementing processing elements
- wiring processing elements to form logical execution plans

Our intuition, still to be proved, is that the first task is better performed using a procedural language, such as Java. The second task, however, is better done using a declarative approach.

Referring back to Section 2 on related work, we observe that with respect to the two tasks mentioned above, the MapReduce programming model assumes the wiring is fixed and can only be slightly modified by providing a partitioning function and by configuring the number of reducers (eventually to zero). Developers are required to implement only a *map* and a *reduce* function using an imperative object oriented language, such as C++ or Java. This has proven to be a great simplification both for using and implementing MapReduce frameworks. On the other hand, it is limited whenever workloads involve complex workflows of MapReduce jobs.

Another important feature we observed in some of the solutions listed in section 2 is the possibility to extend the language with *user-defined* functions (or operators). User-defined functions are an important feature to be included in our framework.

We decided to use Java as the language to implement processing elements. However we have not yet committed to a particular syntax for a scripting language. Logical execution plans are graphs, therefore, two candidate solutions are: RDF and S-expressions[36] extended to support links. A custom scripting language similar to Pig Latin or SPARQLScript might be another valid alternative.

4.3 Execution Plans

The type of processing discussed in this document can be represented as a network of node. We refer to nodes as *processing elements* and to the network as *execution plan*. Execution plans are directed graphs. A processing element is a program written in a conventional programming language such as Java.

Representing a computation as a directed graph gives useful hints to the compiler that can assume each processing element as an indivisible unit of processing. Representing loops and recursive algorithms does not naturally fit this representation and often people assume directed acyclic graphs (DAG). Avoiding loops and recursion is a useful simplification. However, it does not apply to the LUBM motivating example, for which the output could potentially trigger some of the rules. Of course, loops and recursion are available within each processing element, but those will not be automatically parallelized.

The following two sections describe details of what we call *logical* and *physical* execution plans.

[34] http://www.w3.org/2004/03/trix/

[35] http://www.w3.org/TR/rdf-sparql-query/#rdfDataset

[36] S-expressions have been used to represent SPARQL Syntax (see: http://jena.hpl.hp.com/wiki/SSE)

4.3.1 Logical Execution Plan

A *logical execution plan* represents the processing in abstract terms without any details about where data is stored or about physical properties of the cluster that will be used to actually execute the processing. We call *logical processing elements* the processing elements in a logical execution plan.

For example, Figure 1 represents the logical execution plan for the co-authorship network motivating example in Section 3.1.1.

Figure 4 – Logical execution plan for the co-authorship network

A logical execution plan does not depend on the number of machines available in the cluster. In other words, a logical execution plan is portable across different clusters. Finally, although data dependencies are represented in a logical execution plan, data parallelism is not explicit explicitly and this allows developers to focus on how to solve a problem rather than how to provide a parallel solution for the problem.

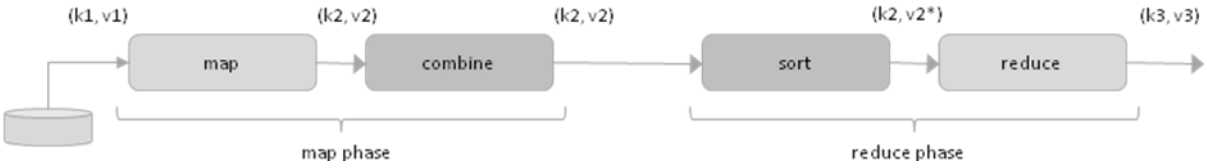

Figure 5 - Logical execution plan for MapReduce

Similarly, Figure 5 represents the logical execution plan of the MapReduce programming model.

4.3.2 Physical Execution Plan

A *physical execution plan* explicitly represents all the details of the computation. The processing for each physical processing element have been selected and parallelism has been made explicit. Physical execution plans are generated automatically by a compiler from logical execution plans.

Figure 6 represents the physical execution plan for the co-authorship network motivating example. It is worth nothing that data parallelism has been exploited in the three phases and additional physical processing elements (i.e. split and merge) have been added.

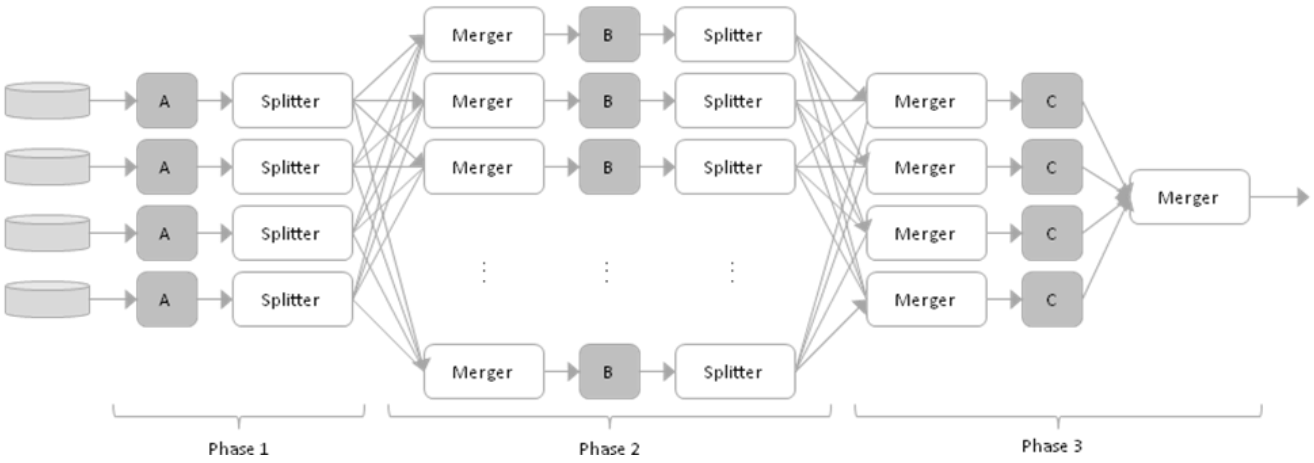

Figure 6 – Physical execution plan for the co-authorship network

4.4 Processing Elements

Processing elements are nodes in an execution plan. Logical processing elements in a logical execution plan represent the semantic of the processing or, in other words, the computation in abstract terms without specifying implementation details or algorithm used. On the other hand, physical processing elements in a physical execution plan provide the implementation of the computation.

Both logical and physical processing elements can be categorized as:

- *functional*: if they emit tuples as they are still receiving tuples, or
- *blocking*: if they need to consume all the input stream before emitting any tuple.

Functional processing elements include: filter, merge sort or merge joins. Blocking processing elements include: sort, max/min, aggregation functions such as sum or average. The amount of storage (memory or disk) that a blocking processing element needs varies and it can depend on the number of incoming tuples, as for sorting, or it can be independent from that and constant, as for aggregate functions or counting.

Processing elements can be categorized according to the number of input streams and the number of output streams. For example, filter and sort are logical processing elements which take one stream as input and produce one stream as output. Their physical implementation, however, could be composed by multiple physical processing elements including split and merge nodes which have multiple input/output streams.

When dealing with multiple input/output streams the framework needs to decide an appropriate order to process the incoming tuples and an appropriate routing function to select the output stream(s) for outgoing tuples. The FIFO (first in first out) strategy works in most cases, but there as situations where a different behavior is necessary. For example, in case of a merge sort: at least one tuple for each input stream need to be received before a tuple can be produced as output. Similarly, with multiple output streams each outgoing tuple needs to be routed to its appropriate output stream(s). Often tuples are partitioned according to one or more values of their fields. Sometimes, signaling tuples need to be sent to all the subsequent processing elements.

Another way to classify processing elements is by taking into account their position in the logical or physical execution plan:

- *entry elements* (or *sources*): if they do not have any input streams
- *processing elements*: if they have one or more input streams and one or more output streams.
- *exit elements* (or *sinks*): if they do not have any output streams.

Entry and exit physical processing elements are important extension points to allow the processing framework to consume data coming from third party storage systems: file system, distributed file systems (such as HDFS), triple/tuple stores (such as TDB) or distributed triple/tuple stores (such as CTDB). Similarly, it should be possible to store the result of processing in any of those systems. Therefore the framework should provide interfaces to allow developers to easily connect different storage systems via entry and exit elements.

Another important extension point for the framework is the possibility to provide additional processing element, a.k.a. user-defined functions. While the framework can offer a library of logical and physical processing elements, it is naïve to assume we can satisfy the diverse requirements for processing in general.

Each logical processing element is translated into one or a network of physical processing elements. The compiler which does this is described in the next section.

4.5 Compilation

Figure 2 showed two software components labeled compilers. However, their role is fundamentally different. The first compiler takes as input the tokens parsed from the scripting language and produces the logical execution plan. This can be achieved with well known techniques [41]. The second compiler takes as input the logical execution plan and transforms it into a physical execution plan. This compiler is responsible for automatically exploiting parallelism. Therefore, this section focuses only on the compiler which generates the physical execution plan.

The compiler needs to know the physical properties of the cluster in terms of number of machines, processors per machine, memory available on each machine and disks per machine. Transformation rules need to be developed to parallelize each logical processing element or particular sequences of logical processing elements. Some logical processing elements in are transformed into a network of physical processing elements and additional physical processing elements might be inserted by the compiler. Some of the logical processing elements might not get parallelized at all, instead, they might get grouped together into one group of physical processing elements each performing processing serially.

Moreover, the amount and characteristics, if available, of data to process is another important factor the compiler to consider. For some logical processing elements, such as sort or join, more than one physical execution plan is possible and the compiler needs to select the most appropriate for the data.

4.6 Scheduling

Given the physical execution plan (also known in the context of scheduling as *task graph*), scheduling usually involves allocating physical processing elements to machines in the cluster and deciding when execute those processing elements.

Figure 7 shows an example of how physical processing elements from the co-authorship physical execution plan might get allocated on a small cluster of eight machines having two processors each.

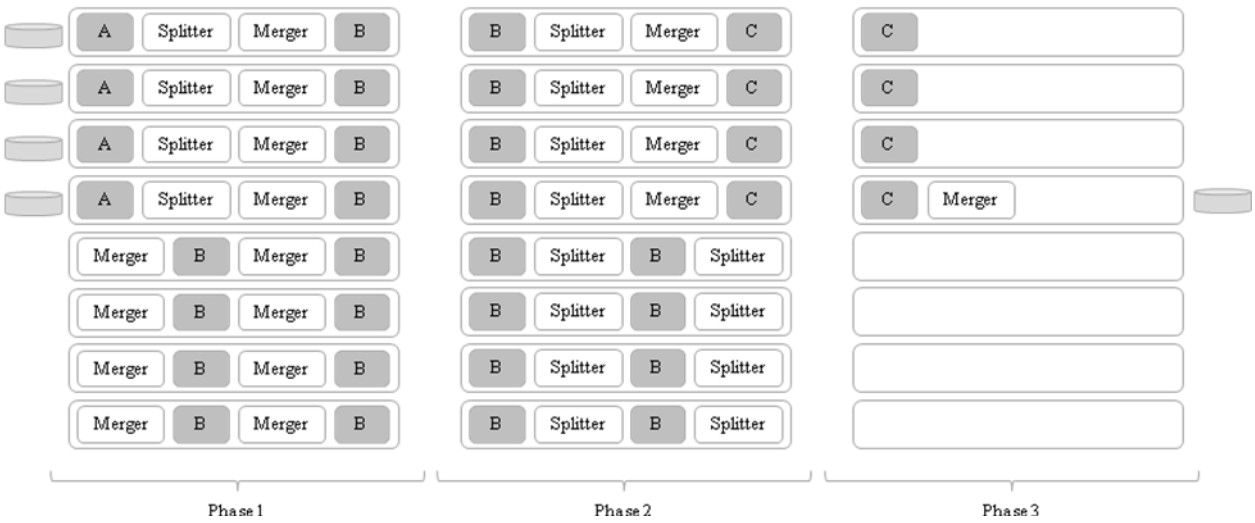

Figure 7 - Allocation of physical processing elements for the co-authorship network

Processing costs for each processing element and communication costs for each edge in the physical execution plan need to be estimated and those costs are used by the scheduler to choose between alternative solutions. Costs are represented in terms of processing, memory requirements, disk utilization and data transferred between machines. However, there are issues in estimating these costs, for example: the amount of data that needs to be processed especially as costs for user-defined

processing elements are not always known in advance. Therefore, it might be worth deferring some scheduling decisions at runtime (*dynamic scheduling*).

The objective of scheduling is to minimize execution time. A solution must balance between maximizing the degree of parallelism (using as many machines as possible) and minimizing network traffic.

Task scheduling, in general, is an NP-hard problem: its associated decision problem is NP-complete and an optimal solution cannot be found in polynomial time (unless NP=P) [32].

Our problem, however, is different from the classical model for a target parallel system which is usually considered by researchers working on scheduling problems. The biggest difference to consider is that resources are physical machines, not processors. Each machine might have more than one processor and more than one task might get allocated to a single processor using threads. For example, if two tasks are blocking processing elements, between which there is a data dependency, they can reasonably be allocated to the same machine.

Another common assumption is to consider local communication cost-free. This is reasonable approximation in our context, however, concurrent remote communications share the same medium and therefore they compete for available bandwidth. As with many of the solutions reported in Section 2, the scheduler should be aware of the network topology, which often is a *fat tree* of two levels, and allocate tasks in a way that minimize the traffic exchanged between machines belonging to different racks.

Another difference in comparison with classic scheduling problems is that, while deciding the allocation of each task is essential, in our context it is not crucial to be able to exactly specify when a task will run. For example, having an idle thread waiting for remote TCP connections running on one machine is a cost that we can ignore.

Static scheduling is done at compile time and once execution starts allocation of processing elements does not change. However, because the size of intermediate datasets cannot be known in advance, especially when user-defined processing elements are involved, it might be worth organizing the computation into stages and materialize the intermediate datasets between stages. This way, the scheduling problem can consider only the tasks in a single stage, one stage at the time, allowing the framework to eventually measure and gather statistics about intermediate datasets.

Entry processing elements (together (if possible) with filters or processing elements that have high selectivity or high projectivity ratios) should be collocated with the data, in order to maximize locality (i.e. local reads/writes vs. remote reads/writes) and reduce network traffic. Similarly, exit processing elements should be allocated where the output data should be, if this information can be derived from subsequent stages.

Finally, because functional processing elements can be grouped together or duplicated, the more likely unit of scheduling seems to be a linear pipeline between two blocking processing elements rather than a single physical processing element.

To conclude, these are just few of the issues we are facing. At this point in time, more experiments and research is needed before we can decide, for example, between static vs. dynamic scheduling. More experiments with scheduling heuristics that consider one job at the time vs. multi job scheduling are necessary in order to design and develop a scheduling solution for our framework.

5. CONCLUSIONS

Parallel programming is by no means a new research area and has benefited from extensive research. Numerous papers have been written and various systems have been implemented. However, very few of them have experienced wide spread adoption or survived the passage of time.

MapReduce is an extreme simplification, driven by a careful selection of features, and a synthesis of ideas which were scattered around in previous systems. This simplicity has freed developers from the worry of data partitioning, locking, message passing, multi-threading, synchronization, deadlocks, fault tolerance, scheduling, load balancing and other related problems. It has also enabled the implementation of MapReduce frameworks in different languages and for different architectures. Open source projects, such as Hadoop, have *de-facto* commoditized the software for MapReduce clusters.

A similar simplification process is ongoing for large data management and storage systems. We are only just beginning to see the effects of consciously trading some of the ACID properties deemed as necessary for relational databases.

These trends are driven by data growth and by the necessity to scale. Assuming semantic web technologies are successful, the exact same data growth and scalability needs will arise. Some argue the need is here already; others claim solutions are available.

The truth is there is still a lot of room for improvement. Inspired by an attitude towards simplicity and armed with a simple and schema-less approach to model our data, this is our first step to explore parallel processing for large RDF datasets and we used it to describe some of our current experiments.

6. REFERENCES

[1] Johnston, W. M., Paul Hanna, J. R. and Millar, R. J. *Advances in Dataflow Programming Languages*. ACM Computing Surveys, Vol. 36, No. 1, March 2004, pp. 1–34.

[2] Yang, H., Dasdan, A., Hsiao, R. and Parker, D. S. *Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters.* In Proceedings of the SIGMOD conference on …

[3] Mika, P. and Tummarello, G. *Web Semantics in the Clouds*. IEEE Intelligent Systems

[4] Chaiken, R., Jenkins, B. et. al. *SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets*. In Proceedings of the VLDB, August 2008

[5] Grossman, R. and Gu, Y. *Data Mining Using High Performance Data Clouds: Experimental Studies Using Sector and Sphere*. In Proceedings of the KKD, August 2008

[6] Dean, J. and Ghemawat, S. *MapReduce: Simplified Data Processing on Large Clusters*. In Proceedings of OSDI, 2004

[7] Ghemawat, S., Gobioff, H. and Leung, S. *The Google File System*. In Proceedings of SOSP, October 2003

[8] Chang, F., Dean, J. et. al. *Bigtable: A Distributed Storage System for Structured Data*. In Proceedings of OSDI, 2006

[9] Pike, R., Dorward, S., et. al. *Interpreting the Data: Parallel Analysis with Sawzall*. Scientific Programming, 2005

[10] TupleFlow

[11] Isard, M., Budiu, M., et. al. *Dryad: Distributed Data-Parallel Programs for Sequential Building Blocks*. In Proceedings of EuroSys, March 2007

[12] Olston, C., Reed, B., et. al. *Pig Latin: A Not-So-Foreign Language for Data Processing*. In Proceedings of SIGMOD, June 2008

[13] Olston, C., Reed, B., et. al. *Automatic Optimization of Parallel Dataflow Programs*. USENIX, 2008

[14] Dennis, J. B. *First Version of a Data Flow Procedure Language*. MIT, 1973

[15] Patterson, D. A. *The Data Center Is The Computer*. Communications of the ACM, January 2008

[16] Baeza-Yates, R. and Ramakrishnan, R. *Data Challenges at Yahoo!*. In Proceedings of EDBT, March 2008

[17] Cafarella, M., Chang, E., et. al. *Data Management Projects at Google*. In Proceedings of SIGMOD, March 2008

[18] Prud'hommeaux, E. and Seaborne, A. *SPARQL Query Language for RDF*. W3C Recommendation, January 2008

[19] Brickley, D. and Guha, R. V. *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C Recommendation, February 2004

[20] Dean, M., Schreiber, G., et. al. *OWL Web Ontology Language – Reference*. W3C Recommendation, February 2004

[21] Parsia, B. and Patel-Schneider, P. F. *OWL 2 Web Ontology Language: Primer*. W3C Working Draft, April 2008

[22] DeWitt, D. J., Robinson, E., et. al. *Clustera: An Integrated Computation And Data Management System*. In Proceedings of VLDB, August 2008

[23] Chaiken, R., Jenkins, B., et. al. *SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets*. In Proceedings of VLDB, August 2008

[24] Taniar, D., Leung, C. H. C., et. al. *High-Performance Parallel Database Processing and Grid Databases*. Wiley 2008

[25] Hellerstein, J. *Programming a Parallel Future*. Greenplum Whitepaper, 2008

[26] *A Unified Engine for RDBMS and MapReduce*. Greenplum Whitepaper, 2008

[27] *Aster nCluster In-Database MapReduce*. Aster Whitepaper, 2008

[28] Mika, P. and Reed, B. *Pearls before Swine: A large-scale triple store using MapReduce*. Preprint paper, 2008

[29] DeWitt, D. J. and Gray, J. *Parallel Database Systems: The Future of High Performance Database Processing*. Communications of the ACM, Vol. 36, No. 6, June 1992

[30] DeWitt, D. J. and Stonebraker, M. *MapReduce: A Major Step Backwards*. The Database Column (weblog), 2008

[31] Yang, H., Dasdan, A., et. al. *Map-reduce-merge: simplified relational data processing on large clusters*. In Proceedings of SIGMOD, 2007

[32] Sinnen, O., *Task Scheduling for Parallel Systems*. Wiley, 2007

[33] Strohman, T. *Efficient Processing of Complex Features for Information Retrieval*. Ph.D. Dissertation, University of Massachusetts Amherst, 2007

[34] Klyne, G., Carroll, J. J. and McBride, B., *Resource Description Framework (RDF) – Concepts and Abstract Syntax*. W3C Recommendation, 10 February 2004

[35] Ding, L., Finin, T., et. al. *Tracking RDF Graph Provenance using RDF Molecules*. In Proc. of the 4th International Semantic Web Conference (Poster), 2005

[36] Stickler , P., *CBD - Concise Bounded Description*. W3C Member Submission, 3 June 2005

[37] Tummarello, G., Morbidoni, C., et. al. *RDFSync: Efficient Remote Synchronization of RDF Models*. In Proc. of the 6th International Semantic Web Conference, 2007

[38] Newman, A., Li, Y.F. and Hunter, J., *A Scale-Out RDF Molecule Store for Improved Co-Identification, Querying and Inferencing*. 4th International Workshop of Scalable Semantic Web Knowledge Base Systems, 2008

[39] Taniar, D., Leung, C. H. C., et. al. *High-Performance Parallel Database Processing and Grid Databases*. Wiley, 2008

[40] Walker, D., Xie, H., et. al. *Ranking Scientific Publications Using a Model of Network Traffic*. Journal of Statistical Mechanics: Theory and Experiment, 2007

[41] Aho, V. A., Lam, M.S., et. al. *Compilers: Principles, Techniques & Tools*. Addison-Wesley, 2007

[42] Guo, Y., Pan, Z. and Heflin, J. *LUBM: A Benchmark for OWL Knowledge Base Systems*. Journal of Web Semantics, 2005