Data model for analysis of scholarly documents in the MapReduce paradigm

Adam Kawa, Łukasz Bolikowski, Artur Czeczko, Piotr Jan Dendek, and Dominika Tkaczyk

Interdisciplinary Centre for Mathematical and Computational Modelling, University of Warsaw
{a.kawa, l.bolikowski, a.czeczko, p.dendek, d.tkaczyk}@icm.edu.pl

Abstract. At CEON ICM UW we are in possession of a large collection of scholarly documents that we store and process using MapReduce paradigm. One of the main challenges is to design a simple, but effective data model that fits various data access patterns and allows us to perform diverse analysis efficiently. In this paper, we will describe the organization of our data and explain how this data is accessed and processed by open-source tools from Apache Hadoop Ecosystem.

Keywords: large-scale analysis, MapReduce paradigm, data modelling

1 Introduction

At CEON ICM UW[1] (Centre for Open Science in Interdisciplinary Centre for Mathematical and Computational Modelling, University of Warsaw) we are in possession of a vast collections of scholarly documents to store and analyze. Currently, there are about 10 million of full texts (PDF or plain text) and 17 million of document metadata records that together occupy several terabytes of disk space. The data grows at the rate of approximately 100 thousand of document metadata records and PDF files (i.e., about 50 GB) a month.

XML-based BWMeta format [6] is used to describe the document metadata records. It contains information like title, subtitle, abstract, keywords, references, contributors and their affiliations, publishing venue and so on.

We are using this data as input for algorithms to analyze and discover various relationships between documents, contributors, references and other entities. Some of the algorithms are relatively simple such as searching documents with given title or finding scientific teams, but some of them are quite complex tasks, based on state-of-the art machine learning and network analysis methods such as author name disambiguation, classification code assignment or finding most influential papers in given domains.

Our current approach suffers from performance/scalability issues since the data was usually located on a several separate machines, but processed by a single machine. Although we can easily add more machines to store more data,

[1] http://ceon.pl/en

it will not solve our performance problems as our computations are not fully distributed. Therefore, we made a decision to move from a single-machine processing to a multi-machine configuration.

2 Problem definition

The research problem can be stated as designing a scalable and efficient storage scheme for RDF triples.However, rather than designing one-fit-all storage scheme for RDF triples, we are interested in a solution that is better adapted to our data and data access pattern. As an additional restriction, the solution should be implementable utilizing open-source tools from Apache Hadoop Ecosystem.

We investigated various frameworks for scalable, distributed and efficient storage and processing of a large amounts of data. Apache Hadoop[2] and related projects like Apache HBase[3], Apache Hive[4] and Apache Pig[5] attracted us the most since they are commonly used, open-source solutions that provide reliable and cost-effective way to persist and process big data.

We focused on designing a simple, but still handy data model that copes with storing terabytes of detailed information about scholarly documents. Various requirements regarding data access were imposed on the model, mainly in terms of flexibility (possibility to add, update and delete records, and enhance their content by implicit information discovered by our algorithms), latency requirements (from batch offline processing to random, realtime read and write requests) and client interface (accessed by programmers and analysts with diverse language preferences and expertise).

We observed that our data can be represented in a flexible way as a collection of triples, each representing a statement of the form *subject-predicate-object*, which denotes that a resource (*subject*) has an attribute (*predicate*) with some value (*object*). In other words, such a collection of triples describes a directed, labeled graph, where nodes represent subjects and objects, and edges represent predicates which connect subject nodes to object nodes.

The concept of triples is used by the Resource Description Framework (RDF)[6] to describe information about Web resources (documents, files, images, services etc.) and model various relationships between them (see Fig. 1 for an example). The RDF data model is extremely flexible since it allows anyone to make any statements about any resource using the *subject-predicate-object.* In a similar way, we can describe any relationship that explicitly exists in our data, or may be discovered later by our machine learning algorithms.

---

[2] http://hadoop.apache.org
[3] http://hbase.apache.org
[4] http://hive.apache.org
[5] http://pig.apache.org
[6] http://www.w3.org/RDF

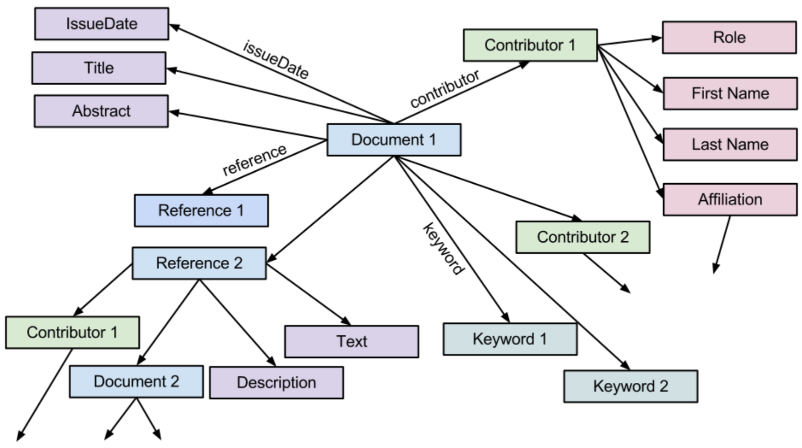


Fig. 1: An example RDF graph data for scholarly documents (some predicate names are omitted for clarity)

3 Related work

During recent years, significant efforts have been made to develop scalable, high-performance and low-cost distributed systems for storing and querying large collections of triples (called triplestores) [1, 12]. Some of them were focused on taking advantage of open-source projects such as Apache Hadoop, Apache HBase and Apache Pig to attain this goal [9, 10, 7].

Rohloff and Schantz [9] authors introduce a concept of scalable triple-store built on Hadoop and HDFS[7] and processed by MapReduce paradigm (see [2] for a description of this paradigm). The paper and resulting system (called SHARD) has aroused significant interest as it leverages Hadoop to scale sub-graph pattern matching (which is quite difficult task) using technique which is 2–3 times more efficient than the naive way of using Hadoop for this task[8]. The system persists RDF triples in "flat" text files in HDFS (each line stores all triples associated with a different subject). The disadvantage of this approach is that data cannot be modified randomly; moreover, this solution is less efficient for queries that require the inspection of only a small number of triples [9].

One-table-per-property approach that uses a concept of vertical-partitioning of RDF data in column-oriented stores was presented in [1]. Here, separate two-column tables are constructed for each property (predicate), where the first column contains the subjects that define that property and the second column contains the object related to those subjects. This approach has several advantages such as support for multi-valued attributes, reduced IO costs and a potential use

[7] http://hadoop.apache.org/hdfs/
[8] http://dbmsmusings.blogspot.com/2011/07/hadoops-tremendous-inefficiency-on.html

of linear merge joins, however it suffers from performance drawbacks on queries that are not bound by a predicate value [12].

Weiss et al. [12] address above-mentioned performance problems by enhancing the idea of vertical partitioning and creating a six indices structure for storing RDF triples. The six indices (*pso*, *pos*, *spo*, *sop*, *ops* and *osp*) cover all the six possible ordering of three elements in a triple. This format allows for quick lookups and partial scans of data at the price of an increase of storage space and complication of the update operation. Since data is stored with multiple indices, all first-step pairwise joins are fast (linear) merge-joins.

Papailiou et al. [8] adopt the idea of [12], but reduce the number of indices to three. Here, all triples are persisted in HBase using three "flat-wide" tables (*sp_o*, *po_s* and *os_p*). The authors argue that the six-index approach can have better performance only for certain queries that contain filters on variables, while for all other queries, three indices suffice for optimal performance. The paper also presents several different join strategies which take the query selectivity and the inherent features of the MapReduce and HBase into account to minimize the processing time.

Performance of six different HBase storage schemas on a subset of queries from SP^{2}Bench[9] is evaluated and discussed in [7]. The hybrid storage schema (consisting of three "flat-wide" tables each indexed by subjects, predicates and objects plus set of two tables for every unique predicate, each indexed by subjects and objects) achieved the best performance results.

The default way of querying RDF data stored in Hadoop is execution of MapReduce algorithms. Since developing such algorithms is still considered challenging, especially for non-technical analysts, some researchers investigated alternative methods for querying RDF data on a Hadoop cluster exploiting high-level languages. One of the approaches is to utilize Apache Pig framework. The paper [10] introduces PigSPARQL, a system that translates SPARQL[10] (a query language for RDF) queries to Pig Latin programs and executes them on Hadoop cluster. The input RDF data resides in HDFS. In accordance with the vertical partitioning idea, RDF triples with the same predicate are stored in HDFS files in the same folder and each predicate has its own folder.

A very impressive performance results of querying RDF data are presented in [5]. The paper describes optimization techniques like RDF graph partitioning scheme to exploit the spatial locality inherent in graph pattern matching. In this approach, a higher replication rate is set for the data on the border of any particular partition. The authors claim their optimizations result in a 1000 fold improvement in efficiency based on experiments done using Lehigh University Benchmark (LUBM)[11]. However, it might be argued[12] that the idea of a higher replication factor would be inappropriate for graph applications that modify the

[9] http://dbis.informatik.uni-freiburg.de/forschung/projekte/SP2B/
[10] http://www.w3.org/TR/rdf-sparql-query/
[11] http://swat.cse.lehigh.edu/projects/lubm
[12] http://muratbuffalo.blogspot.com/2011/12/scalable-sparql-querying-of-large-rdf.html

RDF graph data, because it would be very hard to maintain consistency among the replicas of the boundary vertices as they change.

On the basis of papers mentioned above, we can state that the greatest challenge for processing big collections of triples is a large number of join operations. While joins are expensive, they cannot be avoided in practice (at least for more complex queries). Some systems aim at reducing the number of joins by either storing related data co-located in the same row (in HBase) [11, 8, 7] or line (in HDFS) [9], while other systems optimize the join executions by using specialized join techniques like multi join, merge join or skewed joins [12, 10, 8]. Optimizations like property tables [13, 14] and materialized paths are also proposed [1].

4 Data Storage

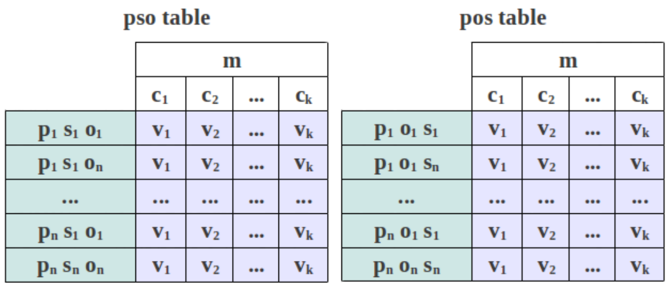
Similarly to [11, 8, 7], we have selected Apache HBase as a main tool for our storage layer. There are several reasons for that:

- **Flexible data model**. Simple, yet flexible data model provided by HBase gives us a control over data layout and format. Namely, we can dynamically add new columns (e.g. containing detailed information about triples such as certainty of relationship, data source, or inferring algorithm) or delete existing ones. There is a possibility to store multiple versions of data in a particular cell distinguished by a timestamp. Additionally, HBase does not require a fixed definition of data types during database creation.
- **Random read and write**. HBase provides random and realtime read-write access to the data allowing us to easily add, update and delete triples. HBase seems to be more suitable for semi-structured RDF data than HDFS where files cannot be modified randomly and the whole file must be read sequentially to find subset of required records.
- **Many clients available**. HBase can be accessed through interactive clients like native Java API, REST or Apache Thrift[13] as well as through batch clients like MapReduce, Pig and Hive. The integration with batch clients, especially MapReduce and Pig, seems to be crucial for us since it gives the possibility to run distributed computation asynchronously in the background, scanning and processing large amounts of data in parallel.
  As mentioned earlier, our data is analyzed by many researchers (including non-technical ones) with various language preferences. Availability of many clients will allow anyone to choose preferable way and language to access the data (e.g. Java, Python, HiveQL or Pig Latin).
- **Automatically sorted records**. HBase stores the data sorted lexicographically by a row key. When storing huge amounts of data, this feature becomes really important since data can be looked-up and scanned quickly. If joins are required (what is often the case), they can be possibly done using simple and fast (linear) merge join [1, 12].

[13] http://thrift.apache.org

5 Storage schema

Following recommendations in [4], we have organized our data in "tall-narrow" layout (many rows, few columns). The main problem with "flat-wide" (few rows, many columns) layout is worse load balancing since a single row is never split across HBase regions. Moreover if a row has an unlimited number of columns, it may outgrow the maximum region size and work against the region split facility [4]. Such a situation seems to be possible in our case as we deal with tens of billions of triples. Moreover storing really wide rows with millions of columns (e.g. a given publisher may be in relation with tens of millions of documents and authors, or a predicate *type* related to hundreds of millions of entities like documents, authors, references, keywords etc.) also reduces performance due to unoptimal load balancing (as mentioned above) and row-level locking mechanism[14].



pso table

	m			
	c_1	c_2	...	c_k
$p_1\ s_1\ o_1$	v_1	v_2	...	v_k
$p_1\ s_1\ o_n$	v_1	v_2	...	v_k
...	...	...	...	...
$p_n\ s_1\ o_1$	v_1	v_2	...	v_k
$p_n\ s_n\ o_n$	v_1	v_2	...	v_k

pos table

	m			
	c_1	c_2	...	c_k
$p_1\ o_1\ s_1$	v_1	v_2	...	v_k
$p_1\ o_1\ s_n$	v_1	v_2	...	v_k
...	...	...	...	...
$p_n\ o_1\ s_1$	v_1	v_2	...	v_k
$p_n\ o_n\ s_n$	v_1	v_2	...	v_k

Fig. 2: Predicate Indexed Layout - HBase Storage Schema

Our HBase storage schema layout is presented in Fig. 2. We adopted the ideas of [1, 12, 7] with some changes to make it more suitable for our use cases. The schema consists of two "tall-narrow" tables *pso* and *pos*, each indexed by *predicate-subject-object* and *predicate-object-subject* respectively. For a given triple, its subject, predicate and object values are concatenated and stored entirely in a row key and one row is added to each of two tables. Each table consists of a one column family called *m* (metadata). Columns dynamically added to this column family contain additional information about triples (this way we can describe statements about statements).

The proposed scheme requires that data is stored twice (HBase has no native support for secondary indexes [4]). Tables *pso* and *pos* can be used to efficiently retrieve triples with known predicate, predicate-subject and predicate-object values. However retrieval of triples based on subject or object values (where predicate is unknown) may be less efficient since it requires a scan of multiple parts of an appropriate table (*pso* and *pos*, respectively). To remedy this problem, we could create four additional tables with remaining indices (*spo*, *sop*, *ops*,

[14] http://www.quora.com/Is-there-a-limit-to-the-number-of-columns-in-an-HBase-row

osp). Although such a solution, may provide significant performance improvement for many interesting queries [12], currently the majority of our queries is predicate-bound (similarly to queries in commonly used benchmarks e.g. LUMB, SP^{2}Bench). Moreover our set of predicates is relatively small and most of the queries require only few predicates at the same time (so that only several partial scans are needed).

Basically, we favour simpler approach (what results in lower redundancy, smaller lower storage requirements and easier update operations) and potentially add new tables with required indices if they appear to be really necessary in the future.

The advantages of this layout can be outlined as:

- **Support of multi-valued properties**. Multi-valued properties (such as a document with multiple titles in different languages) and many-to-many relationships (such as the document and authorship relationship where a document can have multiple authors and an author can write multiple document) are easy to handle in this approach [1]. Actually, there is no difference between single-valued and multi-valued properties. If a subject has more than one object value for a particular property, then each distinct value is stored in a successive row in the table [1].
- **Support of reified statements (statements about statements)**. As mentioned, new columns qualifiers can be dynamically added to column family *m* and contain additional information about triples such as certainty, data source or inferring algorithm.
- **First-step (predicate-bound) pairwise joins as fast merge-joins**. For a given predicate, all triples are sorted by subject (*pso* table) and object (*pos* table). As a result, every pairwise join (with specified predicate) performed during the first step of query processing is a fast, linear-time merge join [12].
- **Reduced redundancy**. Although data is stored twice, the redundancy is still significantly smaller than in storage schemas presented in [11, 8] as well as in 5 out of 6 schemas from [7].
  We can potentially optimize the consumed storage space by directly adopting the idea of vertically partitioned layout [1, 7] and splitting *pso* and *pos* tables into multiple separate tables. This way, for every unique predicate, two tables are created (each respectively indexed by subjects and objects). The storage savings are gained by moving a predicate name to the name of the table and completely eliminating the storage of a predicate as a part of row keys [7].
- **Reduced complexity of update operation**. Similarly, although update operation requires modification in two HBase tables, it is still simpler than approaches presented in [11, 8, 7].
  Since HBase does not provide native support for cross-row atomicity (e.g. in the form of transactions), the consistency of *pso* and *pos* tables cannot be guaranteed. This can be partially overcome by recreating one table from another, if any inconsistency is discovered. One MapReduce, highly efficient bulk import job[15] could be easily implemented for this task.

[15] http://hbase.apache.org/book.html#arch.bulk.load

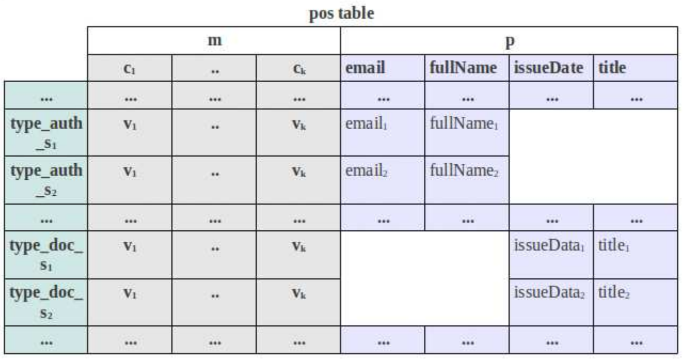
Despite advantages mentioned above, this HBase schema layout has a disadvantages that seriously affect query performance.

- **Increased number of joins**. As triples are stored in "tall-narrow" tables, a larger number of join operations is required to process data. In comparison, "flat-wide" tables approaches entirely remove the need for joins to answer queries about the same subject (or object); however, they still require joins to answer RDF path queries.
  This performance drawback can be alleviated using specialized join techniques such as multi join (when multiple sets are joined by the join key), merge join (when sets are sorted by the join key), replicated join (when one set is very large, while other sets are small enough to fit into memory) and skewed join (when a large number of records for some values of the join key is expected).

6 Optimizations

This simple, but extensible data model allows us to improve the queries performance when more use cases are defined and a deeper knowledge about our data access is gained. Following optimizations can be proposed:

- **Adding new indices**. As mentioned, introducing new indices will improve performance for queries which are not bound by the predicate [12].
- **Property tables** Property tables was proposed by researchers developing the Jena Semantic Web toolkit, Jena2 [14, 13]. By definition, a property table contains clusters of properties that tend to be defined together. One example are type, title, issue date defined as properties for scholarly documents. Such properties can be stored together in the same row for a quick access.
  In fact, this idea is already exploited by us for the storage of reified statements (statements about statements). We take a step forward with this approach and dynamically add new columns qualifiers to a new column family *p* (property) that contains useful properties related to a particular subject in a triple. Rows with the predicate *type* can be simply selected to hold these additional column qualifiers (Fig. 3). Note that some properties may be appropriate to only one or two types, while being totally inappropriate for others. This increases the sparsity of the table, however HBase deals with this issue very efficiently as NULLs are not stored on disk.
  While property tables can significantly improve performance by reducing the number of self-joins, they introduce several problems:
  - **Complexity**. Property clustering (based on explicit or implicit knowledge about data and data access) must be carefully done to create rows that are not too wide, while still being wide enough to answer most queries directly [1].
  - **Redundancy**. While a particular *predicate*, *subject*, *object* is always stored in row keys, it might be also additionally stored in column qualifier or cell (i.e. time-versioned value identified by column and row names).



pos table

	m			p			
	c_1	..	c_k	email	fullName	issueDate	title
...	...	...	...	...	...	...	...
type_auth_s_1	v_1	..	v_k	$email_1$	$fullName_1$		
type_auth_s_2	v_1	..	v_k	$email_2$	$fullName_2$		
...	...	...	...	...	...	...	...
type_doc_s_1	v_1	..	v_k			$issueData_1$	$title_1$
type_doc_s_2	v_1	..	v_k			$issueData_2$	$title_2$
...	...	...	...	...	...	...	...

Fig. 3: Graphical presentation of table *pos* exploiting the idea of a property table

  - **Increased storage**. In HBase, each cell is stored in a "fully qualified" way (with its row key, column family, column qualifier, timestamp etc.) on disk. Adding new cells causes that a row key (consisting of *predicate*, *subject*, *object*) is repeated and stored multiple times on disk, thus increasing the storage space. This is another reason, why a special care should be taken not to make a row too wide.

- **Materialized path expressions**. The concept of materialized path expressions was presented in [1] and discussed in [12]. Path expressions are expressions that match specific paths through a RDF graph[16] (see Fig. 4). Querying path expressions is a common operation on RDF data, but can be quite expensive due to the fact that it requires subject-object joins. Basically, a path of length n requires $n - 1$ subject-object joins. Since our data schema contains *pso* and *pos* indices, the first of joins in a path is a linear merge-join, while the rest $n - 2$ are sort-merge joins, i.e. each one requires one sorting operation [12].

  The idea of precalculation and materialization of the most commonly used path expressions in advance may significantly improve performance of queries. For example, we analyze various statistics about contributor cooperation e.g. finding hubs (i.e. people that collaborate with many other people), finding international scientific teams (i.e. people of different nationalities who collaborate with each other very often), calculating the Erdos number[17] for each person and so on. Generally speaking, the above-mentioned algorithms take pairs of collaborating people as input data. Precalculation and materialization of triples in form of *predicate-subject-object*, where *predicate* is *contributorPair*, *subject* is a document and *object* is a pair of people contributing to this document, would improve performance of these algorithms (see Fig. 5).

[16] http://www.openrdf.org/doc/sesame/users/ch06.html#d0e1170
[17] http://en.wikipedia.org/wiki/Erd%C5%91s_number

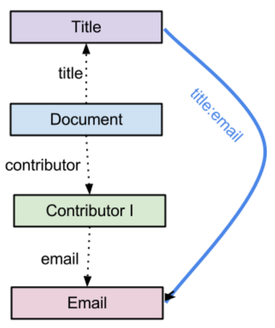


Fig. 4: Graphical presentation of an exemplary expression path query to find title and emails of contributors for each given document

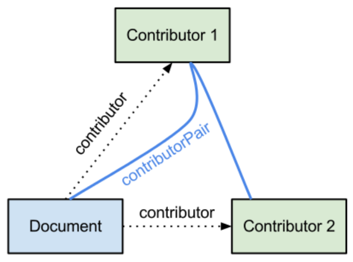


Fig. 5: Graphical presentation of an expression path query to find all pairs of people who contributed to each given document

Basically, we can think about materialized paths as an output of our algorithms (some of them very simple) that we want to store together with the input data in the same HBase tables. For these triples, a special prefix (e.g. *mp*) can be added to row keys to distinguish them from input triples.

All of described optimizations contribute to the improvement of the performance at the price of increased storage requirements and increased complication of update operations. In general, when new data is added or changed, properties tables and materialized paths must be recalculated. We do realize that optimizations above do not solve the problem of large number of joins in a general fashion. They are also not fully automated; however, in many cases the necessary calculations can be computed by regularly scheduled MapReduce jobs.

7 Processing

This section describes how our data residing in HBase tables is accessed by open-source tools from Apache Hadoop Ecosystem. We take advantage of multiple existing clients to meet our various demands like latency requirements (batch offline processing and random, realtime access) and programming preferences (object-oriented, scripting, and declarative languages).

7.1 Pig

Pig is an Apache open source project that provides an engine for executing data flows in parallel on Hadoop. In includes a high-level language (called Pig Latin) for expressing data analysis programs. Pig Latin supports many operators for the traditional data operations (such as join, union, sort, filter). It also allows developing UDFs (user-defined functions) for reading, processing, and writing data [3]. Pig Latin programs are automatically translated into a series of MapReduce jobs.

Loading data from and storing to HBase. Pig is integrated with HBase, so that Pig Latin programs can read data from and write data to HBase tables using HBaseStorage[18]. All these reads and writes are bulk operations.

When loading from HBase, a table name, start and stop row keys, column families, and column qualifiers (even specified only by a prefix) can be selected. This gives a possibility to efficiently read only the data that is needed for to perform the computation.

Listing 1.1: Loading data from HBase in Pig

```
dcraw = LOAD 'hbase://bwtall_pso'
  USING org.apache.pig.backend.hadoop.hbase.HBaseStorage(
  'm:bc',
```

[18] http://pig.apache.org/docs/r0.10.0/api/org/apache/pig/backend/hadoop/hbase/HBaseStorage.html

```
  '-loadKey true -gte contributor_ -lte contributor_\uffff
   -caching 10000')
  AS (key: chararray, blank: chararray);

dcsplit = FOREACH dcraw
  GENERATE FLATTEN(STRSPLIT(key, '_', 3))
  AS (p: chararray, cdoc: chararray, cauth: chararray);

dc = FOREACH dcsplit GENERATE cdoc, cauth;
```

Loading multiple parts of a HBase table using separate LOAD operations is translated to one MapReduce job (consisting of a map phase). In the next step, separate relations (such a *dc*) can be later joined or unioned and become an input to other Pig operators.

Joining data. Processing our data with Pig is very convenient since it supports multiple specialized join implementation (such as multi join, merge join, merge-sparse join, replicated join, skewed join)[19]. Having data sorted both by subject and object (for a given predicate), we are able to perform first step subject-subject joins and subject-object joins using simple and fast merge join operation.

Currently, Pig does not allow programmer to use any UDF in the foreach statement between the load of the sorted input and the merge join statement[20]. Unfortunately, since our subject, object, and predicate are concatenated in a row key, we need to use one UDF (i.e. STRSPLIT[21]) to split a row key into three separate parts (i.e. into a tuple with three fields containing subject, object and predicate which needs to be flattened by FLATTEN operation[22]). As a workaround, we can extend our data model to store additional three column qualifiers for storing subject, predicate and object values, so that STRSPLIT operation can be avoided. Alternatively, we can load sorted input data from HBase, split and flatten each row key into three separate parts and then store them in HDFS. After loading this (stored) data from HDFS, a merge join operation can be executed. The first approach requires more storage space and increases data redundancy, however is much faster in comparison to the second one. So far, have we decided to use the first approach until this issue is resolved by us or Apache Pig community (PIG-2673[23]).

Extending Pig functionality. The list of operators and functions provided by Pig can be easily extended by user by developing his own UDFs [3]. The UDFs can be written in Java or Python.

The integration with Python is even deeper since Pig Latin can be embedded in Python scripts. Thus, Python's control flow constructs like "if" and "for"

[19] http://pig.apache.org/docs/r0.10.0/perf.html#specialized-joins
[20] http://pig.apache.org/docs/r0.10.0/perf.html#merge-joins
[21] http://pig.apache.org/docs/r0.10.0/api/org/apache/pig/builtin/STRSPLIT.html
[22] http://pig.apache.org/docs/r0.7.0/piglatin_ref2.html#Flatten+Operator
[23] https://issues.apache.org/jira/browse/PIG-2673

(which are not natively supported by Pig) may be used to either repeat processing a certain number of times, or to branch based on the results of an operator. Since many machine learning algorithms require repeating a calculation until a certain error value is within an acceptable bound, this feature seems to be very useful [3].

7.2 MapReduce

Using HBase as a storage layer gives also us a possibility to process data by implementing Java MapReduce jobs that read data from and write data to HBase tables. We can implement such jobs if we want to obtain an increased performance or exploit legacy Java code. According to PigMix2 benchmark[24], Pig Latins programs are slower by a factor of 1.37–1.76 when compared with native MapReduce programs with regard to operations like "order by", "outer join", and "group by" (where the key accounts for a large portion of the record).

The greatest current limitation of running Java MapReduce jobs over HBase tables is lack of support for using multiple tables and scanners as input to the mapper in MapReduce jobs (HBASE-3996[25]). However, most of our algorithms require input records from multiple parts of two HBase tables. We alleviate this problem by preparing input data for MapReduce jobs in Pig (using LOAD, UNION and JOIN operations) and then running MapReduce jobs directly from Pig scripts with the mapreduce command [3]. This way we can avoid the burden of implementing complex (like join) or even not supported yet (like multiple tables and scanners as input) operations in Java MapReduce, but still have a possibility to incorporate processing using the MapReduce paradigm in our big data analysis.

7.3 Interactive clients

While offline, complex analysis are performed by batch processing clients, there are some use cases where we need interactive access to relatively small subset of our data. One of such examples is a web-based client that may search for entities with attributes matching a certain query and navigate amongst interconnected entities by sending client API calls on demand (such as get and scan).

HBase is a good fit for such use cases, as it is designed to perform random, realtime read and write requests to big data. Interactive clients include, e.g. native Java API, REST or Apache Thrift.

7.4 Other clients: Hive

Hive is a data warehouse system for Hadoop that supports easy data summarization, ad-hoc queries, and analysis of large datasets[26]. It provides a SQL-like

[24] https://cwiki.apache.org/confluence/display/PIG/PigMix
[25] https://issues.apache.org/jira/browse/HBASE-3996
[26] http://hive.apache.org

language called HiveQL to query the data, thus makes Hadoop accessible to analysts who already know SQL. Hive is integrated with HBase, so that HiveQL statements can access HBase tables for both read (SELECT) and write (INSERT)[27]. Similarly to Pig and MapReduce, all these reads and writes are bulk operations.

The main advantage of using Hive over HBase is possibility to use JOIN and UNION statements. This way more complex analysis on data from HBase can be performed similarly as in SQL. As many of our researches already know SQL, this approach can be widely used for a rapid development of ad-hoc queries.

8 Configuration

Our Hadoop cluster consists of a four "fat" worker nodes and a virtual machine on separate physical machine in the role of NameNode, JobTracker and HBase master. Each worker node has four AMD Opteron 6174 processors (48 cores in total), 192 GB of RAM, four 600 GB disks which work in RAID 5 array with an access to 7TB LUN of NetApp disk storage over FC. The master has 8-core CPU, 32 GB of RAM and 64 GB storage. Each worker node runs 50 mappers and 30 reducers, each consuming 1GB of RAM (may be increased for memory-consuming applications). We are using Cloudera's Distribution including Apache Hadoop (CDH)[28], mainly CDH3u3 which includes Hadoop 0.20.2, and HBase 0.90.4 by default. We decided to update to Hive (from 0.7.1 to 0.9.0) and Pig (from 0.8.1. to 0.10.0) to benefit from newly implemented features.

9 Conclusions

We have described how our data is stored and processed using open-source tools from Apache Hadoop Ecosystem. The main goal was to provide a simple, but handy data model that can be conveniently accessed using various client interfaces. Our initial work proves that Apache Hadoop and its related projects are good framework for persisting and processing of millions of scholarly documents that together occupy several terabytes of data. We see clear benefits of using Apache Hadoop for this task when compared with our previous, not fully distributed approach because now the data is stored in a reliable way (replicated in HDFS) and processed in parallel using MapReduce paradigm.

10 Plans for future

Deep examination and specification of our future algorithms may contribute to selection of the most suitable indices for storing and querying triples or precalculation of the most common materialized paths.

[27] https://cwiki.apache.org/confluence/display/Hive/HBaseIntegration
[28] http://www.cloudera.com/hadoop

We intend to investigate other HBase storage schema layouts, mainly composed of "flat-wide" tables, where the number of columns can be limited to a certain number (or its approximate value) in some way. Such an approach would allow us to reduce number of rows (and result in faster seeks) and joins (as data is colocated in the same rows) and shorten row key size (to save storage space, as the row key is stored with each KeyValue pair)[29], while the rows will still have predictable size and do not work against the Hadoop region split facility.

We will also focus on implementation of required features (or more efficient workarounds), that currently are not fully supported by modules responsible for an integration of HBase with MapReduce, Pig and Hive. Some effort has been made by Hadoop community to solve these issues, however the work still has not been finalized. We could contribute to implementation of following, still incomplete patches:

- Support multiple tables and scanners as input to the mapper in MapReduce jobs (HBASE-3996[30]).
- Allow merge join operation to follow an UDF statement (PIG-2673[31]).

11 Acknowledgements

This work is supported by the National Centre for Research and Development (NCBiR) under Grant No. SP/I/1/77065/10 by the Strategic scientific research and experimental development program: "Interdisciplinary System for Interactive Scientic and Scientic-Technical Information."

References

1. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable semantic web data management using vertical partitioning. VLDB pp. 411–422 (2007), http://dl.acm.org/citation.cfm?id=1325900
2. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters pp. 1–13 (2008), http://dl.acm.org/citation.cfm?id=1327492
3. Gates, A.: Programming Pig. OReilly Media (2011)
4. George, L.: HBase: The Definitive Guide. OReilly Media (2011)
5. Huang, J., Abadi, D.J., Ren, K.: Scalable SPARQL Querying of Large RDF Graphs. VLDB Endowment 4 (2011)
6. Jurkiewicz, J., Nowiski, A.: Detailed Presentation versus Ease of Search Towards the Universal Format of Bibliographic Metadata (2011)
7. Khadilkar, V., Kantarcioglu, M., Thuraisingham, B., Castagna, P.: Jena-HBase: A Distributed, Scalable and Efficient RDF Triple Store. Tech. rep. (2012), http://www.utdallas.edu/ vvk072000/Research/Jena-HBase-Ext/tech-report.pdf

[29] http://www.cloudera.com/resource/hbasecon-2012-lessons-learned-from-opentsdb
[30] https://issues.apache.org/jira/browse/HBASE-3996
[31] https://issues.apache.org/jira/browse/PIG-2673

8. Papailiou, N., Konstantinou, I., Tsoumakos, D., Koziris, N.: H2RDF: Adaptive Query Processing on RDF Data in the Cloud. In: Proceedings of the 21th International Conference on World Wide Web (WWW demo track). pp. 397–400 (2012)
9. Rohloff, K., Schantz, R.: Clause-Iteration with MapReduce to Scalably Query Data Graphs in the SHARD Graph-Store. In: Proceedings of the fourth international workshop on Data-intensive distributed computing. pp. 35–44. ACM (2011), http://www.dist-systems.bbn.com/people/krohloff/papers/2011/Rohloff_Schantz_DIDC_2011.pdf
10. Schätzle, A., Przyjaciel-Zablocki, M., Lausen, G.: PigSPARQL: Mapping SPARQL to Pig Latin. In: 3th International Workshop on Semantic Web Information Manage- ment (SWIM 2011), in conjunction with the 2011 ACM International Conference on Management of Data (SIGMOD 2011). Athens (Greece). (2011), http://www.informatik.uni-freiburg.de/ schaetzl/papers/PigSPARQL_SWIM2011.pdf
11. Sun, J., Jin, Q.: Scalable RDF Store based on HBase. In: 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE). pp. 633 –636 (2010)
12. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. Proceedings of the VLDB Endowment pp. 1008–1019 (2008), http://dl.acm.org/citation.cfm?id=1453965
13. Wilkinson, K.: Jena Property Table Implementation. Tech. rep. (2006)
14. Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D.: Efficient RDF Storage and Retrieval in Jena2. Tech. rep. (2003)