SPARQL Query Answering on a Shared-nothing Architecture

Spyros Kotoulas
kot@few.vu.nl
Department of Computer Science
VU University Amsterdam

Jacopo Urbani
j.urbani@few.vu.nl
Department of Computer Science
VU University Amsterdam

ABSTRACT

The amount of Semantic Web data is outgrowing the capacity of Semantic Web stores. Similar to traditional databases, scaling up RDF stores is faced with a design dilemma: increase the number of nodes at the cost of increased complexity or use sophisticated, and expensive, hardware that can support large amounts of memory, high disk bandwidth and low seek latency. In this paper, we propose a technique to do distributed and join-less RDF query answering based on query pattern-driven indexing. To this end, we first propose an extension of SPARQL to specify query patterns. These patterns are used to build a query-specific indexes using MapReduce, which are later queried using a NoSQL store. We provide a preliminary evaluation of our results using Hadoop and HBase, indicating that, for a predefined query pattern, our system offers very high query throughput and fast response times.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Distributed data structures; C.2.4 [**Distributed Systems**]: Distributed applications

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

RDF, SPARQL, MapReduce

1. INTRODUCTION

The amount of data in the Semantic Web is growing with a faster pace than the computational power of computers. To cope with this disparity, we either need more efficient single-machine algorithms for data management or scalable distributed methods to tap on the computational resources of more machines.

When designing an RDF store, one is often confronted with a design dilemma: use centralized or clustered storage. The former allows easier management but only scales by using more powerful hardware, which is often infeasible or prohibitively expensive. The latter allows adding additional computational resources to increase performance at the cost of higher complexity and, often, large query response times.

Typically, clustered stores split their indexes across the nodes at loading time. When resolving queries, nodes may need to exchange data. These data exchanges should be minimized since they increase load and response time. Ideally, only data that will be part of the answer should be exchanged. This is a very difficult task in the Semantic Web, because the semi-structured nature of the data makes access patterns difficult to predict.

There is another problem that we must consider. In relational databases, the information that is inserted in the system is carefully selected. In Semantic Web stores, we often encounter the situation that, while an entire dataset is included in the knowledge base, only a small subset is eventually used, for a given application.

In this work, we focus on clustered storage and we claim that we should only index the part of the data that we need, and index for specific access patterns. We propose a technique to decrease the complexity by calculating query pattern-driven indexes and this will enable us to exploit clustered architectures without the need to combine data from different nodes at query time.

There are several scenarios where the user is able to define the queries in advance. For example, application developers very often know the query patterns of their applications, or they can be very easily extracted from the application source code. Furthermore, analytical workloads typically have a fixed set of query patterns. For instance, when analysing social network data, the user may only be interested in resolving queries with the predicate "foaf:knows". Finally, such an approach could be combined with a standard store. This approach would answer queries that match some popular query patterns, offloading the standard store that would match the rest.

In section 2, we will introduce $SPARQL^P$, an extension of the $SPARQL$ [10] query language to allow for defining $SPARQL$ query patterns. These query patterns serve as a mapping between a subset of the unbound variables of the query, called the *pattern variables*, and the result set of the query.

In section 3, we will present a method for building the indexes based on the query patterns. This method constructs only the indexes required for each $SPARQL^P$ query pattern using the MapReduce[6] parallel programming paradigm.

Workshop on Semantic Data Management (SemData@VLDB) 2010, September 17, 2010, Singapore.

To this end, we have adapted some standard database techniques to MapReduce/RDF and developed some new ones. Furthermore, we will show how we query these indexes.

In section 4 we will present an implementation of our methods using the Hadoop MapReduce framework and the HBase NoSQL database. Our results indicate that our approach has good performance on large datasets, where it is able to answer specific queries in a few milliseconds.

2. SPARQL^P

In this section, we will define an extension to SPARQL for query patterns. We explain the details of this extension with an example. Consider an application that needs to retrieve the names and email addresses of all professors that work for a university in a given country. The corresponding SPARQL query[1] would be:

> SELECT ?name ?email where { ?x a professor. ?x worksfor ?y. ?y locatedin ?country. ?x email ?email. ?x name ?name. }

The application will repeatedly post the query above replacing "?country" with the country it is interested in and retrieve the corresponding values of (?name, ?email).

$SPARQL^P$ aims at capturing such usage patterns by rewriting the query as a function

$$?country \rightarrow (?name, ?email)$$

In SPARQL^P we define such a function using the $DEFINE$ construct and execute it using the GET construct. Looking at our example, the mapping function is defined as:

> DEFINE professorsOfCountry(?country) AS SELECT ?name ?email where { ?x a professor. ?x worksfor ?y. ?y locatedin ?country. ?x email ?email. ?x name ?name. }

Using this definition, an RDF store prepares to handle requests on the given function. Once this operation is completed, we use the GET construct to execute this query for specific countries. For example, in order to retrieve the professors from the Netherlands we would launch the query:

> GET professorsOfCountry(Netherlands)

In the appendix A, we formally define the $SPARQL^P$ grammar by extending the EBNF notation.

3. SPARQL^P ON A SHARED-NOTHING ARCHITECTURE

To optimally support the querying model presented in the previous section, for each $SPARQL^P$ DEFINE query, we can maintain a pattern index, i.e. an index from the pattern variables to the other unbounded variables. In fact, these are the only indexes we need to answer $SPARQL^P$ GET queries.

Typically, RDF stores maintain a set of triple indexes over the entire input. These indexes consist of permutations of the triple terms (Subject-Predicate-Object, Predicate-Object-Subject etc). In our approach, we do not maintain full triple indexes, so that we avoid loading costs. Instead, whenever we process a $SPARQL^P$ DEFINE query, we usually read the entire input and construct only the indexes that we need for that query.

[1]for brevity, we will omit namespaces for the rest of this paper

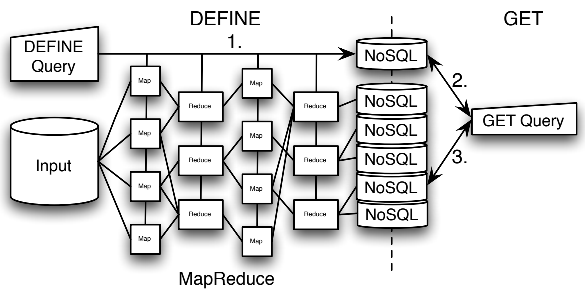

Figure 1: Overview of the approach

Calculating these indexes implies resolving the embedded SPARQL query within the $SPARQL^P$ DEFINE queries. We consider the following in the resolution of these queries:

- The embedded queries will always be more expensive to resolve and will produce more bindings than the corresponding standard SPARQL queries, since they have less bound variables.
- We are interested in the time to retrieve all results. Thus time-to-first result is not relevant, neither is streaming of results. The performance goal is to maximize result throughput.
- We are focusing in answering queries over vast amounts of data rather than the response time of individual $SPARQL^P$ DEFINE queries. Query response time is more relevant for $SPARQL^P$ GET queries, which we will discuss in section 3.2

Considering the above, we will use the MapReduce [6] paradigm to construct pattern indexes. In Appendix B, we are providing a short introduction to MapReduce, which is essential for comprehending section 3.1.

In Figure 1, we give an overview of our approach. For DEFINE queries, we extract the embedded query and we execute it using MapReduce. We index the results on the pattern variables and store the name of the query in a separate table. The indexes are loaded in a NoSQL database. For SPARQL^P GET queries, we retrieve a reference to the index using the query name, and we query it using the pattern variables. Since the variables are used as keys for the index, we do not need to perform any joins to retrieve the results.

In the next subsections, we will describe how we deal with SPARQL^P DEFINE and GET queries.

3.1 SPARQL^P DEFINE

Resolving a SPARQL^P DEFINE query implies resolving the embedded SPARQL query which will typically be an expensive query.

Querying Semantic Web data with MapReduce and without a-priori constructed indexes differs from querying in traditional stores in the following:

- Load-balancing: Clustered databases typically share data across nodes, and they aim at placing data that is used together on the same node. This decreases response time, since less data needs to be accessed over the network. In contrast, MapReduce focuses mainly at dividing computation equally across nodes, aiming at maximising parallelisation rather than response time. MapReduce achieves this by dynamically partitioning the data.

- Setting up MapReduce jobs is an expensive operation, in terms of coordination effort. Thus, it incurs high latency.

- Without a-priori indexing, selectivity estimation is difficult, since accurately calculating expression cardinality is impossible without a-priori data processing.

- Passing information between running Map or Reduce tasks in the same job is not allowed by the paradigm. Therefore, the query plan can only be modified between different jobs. This is practical because SPARQL^P DEFINE queries are expensive, and the overhead of starting new jobs is high.

- Semantic Web triples are small in size, typically containing three of four terms and occupying dozens of bytes, if dictionary encoding [12] is used, and hundreds of bytes otherwise. Furthermore, table scans are scheduled across all processors. Thus, scanning the entire input is relatively fast.

Based on the above observations, we are proposing the following for the resolution of MapReduce $SPARQL^P$ DEFINE queries:

3.1.1 Tuple-at-a-time vs operator-at-a-time

Typical RDF stores perform tuple-at-a-time evaluation. In MapReduce, it is only possible to partition data once per job. Thus, tuple-at-a-time evaluation would pose significant limitations on the supported queries. Thus, within jobs, we process one tuple at a time, while each job implements several operations. Jobs are run in sequence, so our method uses a mix of tuple-at-a-time and operation-at-a-time processing.

3.1.2 Selectivity estimation

Since, in a MapReduce architecture, full table scans are relatively cheap, we use an accurate selectivity estimation algorithm, performing a full table scan. During this scan, we calculate the cardinality of each statement pattern in the query. Furthermore, for statement patterns with few bindings, we also store the bindings.

3.1.3 Joins during the Map phase

Typically, joins are performed in the Reduce phase. This gives rise to load balancing problems, since making equally-sized partitions for the Reduce phase is not trivial. Furthermore, it incurs significant overhead, since in a MapReduce job, we can only have a single Reduce function.

Neither of the above is true for Maps. Thus, if one side of the join is small enough to fit in main memory, we perform a Grace hash-join[8]. This implies all nodes loading the small side in memory and iterating through the large side in parallel. The advantage of this approach is that we do not need to re-partition our data, eliminating load balancing problems and the need for an additional job.

If all sides of a join are large, then we resort to a hybrid of hash join and sort-merge join. First the input tuples are partitioned among nodes using a hash function. Then, each partition is sorted locally at each node and a sort-merge join is performed. Note that for larger joins, the load balancing problem is dissipated by the fact that we have more bindings per join variable.

3.1.4 Recycling

Since we are not maintaining traditional indexes, and disk storage in a MapReduce cluster is cheap, our method uses intermediate result recycling[7]. Intermediate results are not indexed, but stored in parallel as a collection of local files. In [7], it was shown that this method improves performance for systems doing operator-at-a-time processing.

3.1.5 Index construction

As the last step of $SPARQL^P$ DEFINE query evaluation, we are constructing indexes mapping template variables to the results of the query. To this end, we are tapping on the massive sorting functionality of MapReduce frameworks to construct indexes in a scalable manner.

This is accomplished by a job that groups tuples according to template variables, performs a global sort and writes the results to files. After that a suitable partitioning of the data is calculated, this task is highly parallelisable. The sorted files are written to the local disk and accessed by the $SPARQL^P$ GET method, described in the following section.

3.2 SPARQL^P GET

The indexes created during the SPARQL^P DEFINE query resolution are queried using a NoSQL store.

For each SPARQL^P DEFINE, a new table is created in the NoSQL database. We use the bindings of the pattern variables as key and the results as values. In case that we have a large number of results for a given variable, it is more efficient if we store a pointer to a file in the Distributed File System, instead of the values themselves. This implies an additional lookup for queries, but this cost is amortised over the number of retrieved results and the reduced size of the table.

Lacking any joins, SPARQL^P GET queries are very fast to evaluate: a single lookup is enough to retrieve all results for a given query. Furthermore, clustered NoSQL implementations, assure high fault tolerance, horizontal scalability and no single point of failure [4].

4. PRELIMINARY EVALUATION

We have performed a preliminary study to analyze the benefits of our approach. More thorough evaluation and comparison to existing approaches is deferred to future work.

This section is organized as follows: in subsection 4.1, we describe the testbed and the implementation details of our prototype. We continue reporting the performance of the DEFINE query in subsection 4.2. At last, we report the performance of the GET query in subsection 4.3.

4.1 Implementation and Testbed

We have implemented a prototype which implements our methods. We have used Sesame [3] for query parsing and basic query planning, the Hadoop framework for MapReduce and the HBase NoSQL database for answering GET queries.

As a preprocessing step, we compress our input data using the method described in [12] and, optionally, materialize the closure under the OWL ter horst semantics using previous work in WebPIE [11].

For the DEFINE queries, we do the following: Firstly, we extract and index the name and template variables of the query and extract the embedded SPARQL query. We use Sesame to parse the SPARQL query and launch our selectivity estimation algorithm. The results of the selectivity estimation are passed back to Sesame, which creates a static query plan. This query plan is executed in the MapReduce framework, and is dynamically adjusted to cater for recycling and bindings already calculated during the selectivity estimation step. Finally, indexes are created for the query plans. Currently, we use a very simple index construction mechanism that uses a single node, but that can be easily extended by carefully partitioning data. The index is loaded by HBase.

For GET queries, we first retrieve the query index, using the name of the query as a key. Then, we post a query consisting of the template variables on HBase on the retrieved index and retrieve the results.

The performance of our prototype was evaluated using the DAS3 cluster[2] of the Vrije Universiteit Amsterdam. We set up an Hadoop and HBase cluster using 32 data nodes, with 4 cores, a single SATA drive and 4GB of memory each.

We have used the LUBM benchmark for our evaluation. We chose query number 8, since it is one of the most challenging queries in the dataset and used the University as a parameter. We have generated the LUBM(8000) dataset, consisting of 1 billion triples. Dictionary encoding took 46 minutes and materializing the closure took 37 minutes, yielding a total of 1.52 billion triples. We created a variant of query number 8 of the standard LUBM query set as a DEFINE query and executed it against the 1.52 billion triples. The query is reported below:

```
DEFINE getStudentsOfUniversity(?P) AS
SELECT
{?X type Student .
 ?Y type Department .
 ?X memberOf ?Y .
 ?Y subOrganizationOf ?P .
}
```

For GET queries, we have extracted the URIs for all universities and evaluated by posting queries for the students of random universities. Note that, for our prototype, we do not perform dictionary decoding, of the results.

In the following two subsections we describe the performance obtained from first launching a DEFINE query and later a sequence of GET queries.

4.2 Performance of the DEFINE query

For this task, our prototype produced four MapReduce jobs in a sequence. In Table 1, we report the execution time for each job.

Job 1 performed the selectivity estimation and extracted the patterns which are small enough to fit in the main memory. The ordering of the patterns according to their increasing selectivity was: `(?X type Student)`, `(?X memberOf ?Y)`, `(?Y type Department)`, `(?Y subOrganizationOf ?P)`. The bindings for the last two patterns were small enough to fit in memory, thus they were extracted.

[2]http://www.cs.vu.nl/das3

Job	Runtime (in sec.)
job 1: selectivity estimation	61
job 2: first join job	36
job 3: second join job	89
job 4: HBase upload	305

Table 1: Execution time of the DEFINE query

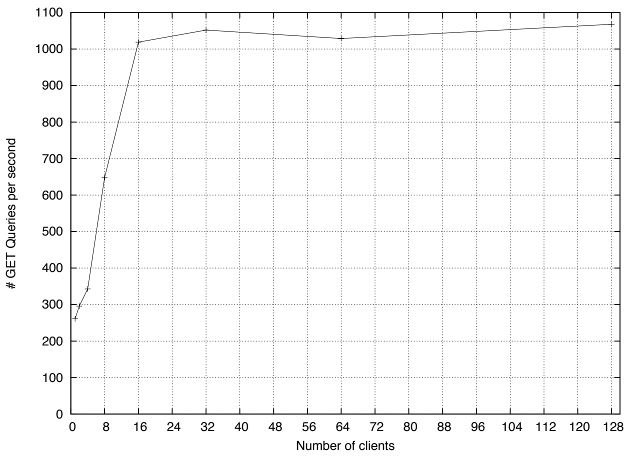

Figure 2: Total number of GET queries per second

Job 2 loaded the extracted bindings, `(?Y type Department)` and `(?Y subOrganizationOf ?P)`, and joined them with the input during the Map phase. In the Reduce phase, the results were joined with `(?X memberOf ?Y)`.

Job 3 performed a hybrid join with `(?X type Student )`, since both sets were too large to fit in memory.

Job 4 grouped the tuples according to ?P and uploaded the bindings to HBase. The poor performance is attributed to the fact that we did not create suitable partitions for the data. Also, we used the HBase APIs to create the index while we could generate it using MapReduce and then load it on HBase. The second approach would give about an order of magnitude better loading time, compared to our current implementation[3].

4.3 Performance of the GET query

After we have generated the index, we can efficiently query the knowledge base with the GET query. This query will receive in input the university and it will return all the students for that university.

A single query took on average 3.83 milliseconds to be executed. We decided to stress the system by launching queries simultaneously from different nodes. The purpose of this stress test was to evaluate the scalability of our method as the number of queries per second increases. We started by launching the queries from a single client and we increased the amount of clients up to 128. Queries were posted from

[3]see http://hbase.apache.org/docs/current/api/org/apache/hadoop/hbase/mapreduce/package-summary.html#bulk

several nodes and we have assigned a CPU core to each client.

We report the results in Figure 2. From the table, we see how the throughput of the system increases until it can serve about 1100 queries per second (i.e. 1ms per query) and then it stabilizes regardless the number of clients. Again, the performance could be dramatically increased if we would tune the framework on our needs. For example, our index is spread in 8 different regions, which were served by only 5 nodes, effectively using only 5 of the 32 available nodes. If we further distribute the table we could reach even higher throughput.

5. RELATED WORK

This work is related to RDF stores, MapReduce processing and view materialization in SQL databases.

HadoopDB [1] is a hybrid MapReduce/database implementation, focused on the processing of SQL queries. Data is maintained by traditional databases in the data nodes. For every SQL query received by the system, a MapReduce query plan is generated. The input for the corresponding MapReduce job is retrieved from the SQL databases maintained by the data nodes. In contrast, our approach does not build indexes at the loading time and does not perform joins at runtime.

Relational views[5] is mature concept in the field of SQL databases, providing functionality to define a virtual table. Furthermore, SQLServer, DB2 and Oracle [2] also materialize these tables. Our work captures SPARQL data usage patterns at a greater detail, since we can embed any kind of query in our DEFINE queries and we specify the key on which the results of this query should be indexed. On the other hand, SQL views have the advantage that they use the same model of normal SQL tables.

RDF stores typically employ tuple-at-a-time processing and avoid materialisation of intermediate results. Furthermore, they rely on doing joins at run-time. Our preliminary results indicate that our approach vastly outperforms these stores in terms of throughput and response time of the SPARQL^P GET queries, since the latter require no joins. Nevertheless, our system is at a disadvantage when processing arbitrary SPARQL queries. Furthermore, updating data in our system is expected to be an expensive operation.

The work reported in [13] demonstrates a method to reduce a dataset to an interesting subset using a supercomputer. It is focused on extracting a relatively small portion of the input, which can be queried using a conventional RDF store (with the associated limitations). In comparison, our architecture is not limited by the size of the indexed data (since the indexes created in our NoSQL store can be very large) but it is limited by the access patterns to this data.

6. FUTURE WORK AND CONCLUSION

We intend to perform a thorough evaluation of our system using larger datasets and more queries. Our results should be compared with those of standard RDF stores. Moreover, we expect that our method will also be useful for answering standard SPARQL queries. Nevertheless, we expect to get good performance only for very expensive queries, since the overhead of the platform is very high. Furthermore, research should be carried out in update mechanisms for indexes. To this end, techniques for materialised SQL views can be used.

In this paper, we have presented a method for RDF data management based on a shared-nothing architecture. Instead of maintaining indexes over the entire dataset, only the indexes required for given query patterns are generated. We have implemented a first prototype and we tested the performance on a sample query. Our preliminary evaluation shows that the system can reach high throughput but, though this approach is promising, further work is necessary for a complete evaluation.

This work was supported by the EU-IST project LarKC (FP7-215535).

7. REFERENCES

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.

[2] R. G. Bello, K. Dias, A. Downing, J. J. Feenan, Jr., J. L. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized views in oracle. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 659–664, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[3] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 54–68, 2002.

[4] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[5] E. F. Codd. Recent investigations in relational data base systems. In *IFIP Congress*, pages 1017–1021, 1974.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 137–147, 2004.

[7] M. G. Ivanova, M. L. Kersten, N. J. Nes, and R. A. Gonçalves. An architecture for recycling intermediates in a column-store. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 309–320, New York, NY, USA, 2009. ACM.

[8] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Comput.*, 1(1):63–74, 1983.

[9] E. Maler, J. Paoli, C. M. Sperberg-McQueen, F. Yergeau, and T. Bray. Extensible markup language (XML) 1.0 (third edition). first edition of a recommendation, W3C, Feb. 2004. http://www.w3.org/TR/2004/REC-xml-20040204.

[10] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. W3C recommendation, W3C, Jan. 2008. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[11] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, and H. E. Bal. Owl reasoning with webpie: Calculating the closure of 100 billion triples. In L. Aroyo, G. Antoniou, E. Hyvönen, A. ten Teije, H. Stuckenschmidt, L. Cabral, and T. Tudorache, editors, *ESWC (1)*, volume 6088 of *Lecture Notes in Computer Science*, pages 213–227. Springer, 2010.

[12] J. Urbani, J. Maassen, and H. Bal. Massive semantic web data compression with mapreduce. In *Proceedings of the MapReduce workshop at HPDC*, 2010.

[13] G. T. Williams, J. Weaver, M. Atre, and J. A. Hendler. Scalable reduction of large datasets to interesting subsets. In *8th International Semantic Web Conference (Billion Triples Challenge)*, 2009.

APPENDIX

A. SPARQL^p GRAMMAR

In this appendix we formalize the SPARQL^p grammar using the EBNF notation defined in the W3C recommendation for XML [9]. We do not define the grammar from scratch but we extend the SPARQL grammar [10] with the following constructs:

```
DefineQuery ::= 'DEFINE' ( IRIref+ )
                'AS' Query

GetQuery ::= 'GET' FunctionCall
```

B. MAPREDUCE

MapReduce is a parallel programming paradigm for parallel and distributed processing of batch jobs. Each job consists of two phases: a map and a reduce. The mapping phase partitions the input data by associating each element with a key. The reduce phase processes each partition independently. All data is processed based on key/value pairs: the map function processes a key/value pair and produces a set of new key/value pairs; the reduce merges all intermediate values with the same key into final results.

B.1 MapReduce example: term count

We illustrate the use of MapReduce through an example application that counts the occurrences of each term in a collection of triples. As shown in Algorithm 1, the *map* function partitions these triples based on each term. Thus, it emits intermediate key/value pairs, using the triple terms (*s*,*p*,*o*) as keys and blank, irrelevant, value. The framework will group all the intermediate pairs with the same key, and invoke the *reduce* function with the corresponding list of values. The *reduce* function will sum the number of values into an aggregate term count (one value was emitted for each term occurrence) and return the result as output.

This job could be executed as shown in Figure 3. The input data is split in several blocks. Each computation node operates on one or more blocks, and performs the map function on that block. All intermediate values with the same key are sent to one node, where the reduce is applied.

Algorithm 1 Counting term occurrences in RDF NTriples files

```
map(key, value):
  // key: line number
  // value: triple
  // emit a blank value, since
  // only amount of terms matters
  emit(value.subject, blank);
  emit(value.predicate, blank);
  emit(value.object, blank);

reduce(key, iterator values):
  // key: triple term (URI or literal)
  // values: list of irrelevant values
  //   for each term
  int count=0;
  for (value in values)
    count++; // count number of values,
             // equalling occurrences
  emit(key, count);
```

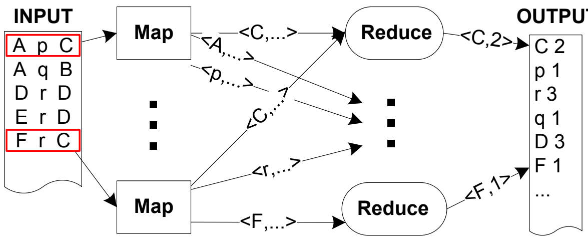

Figure 3: MapReduce processing

B.2 Characteristics of MapReduce

This simple example illustrates some important elements of the MapReduce programming model:

- since the *map* operates on single pieces of data without dependencies, partitions can be created arbitrarily and can be scheduled in parallel across many nodes. In this example, the input triples can be split across nodes arbitrarily, since the computations on these triples (emitting the key/value pairs), are independent of each other.
- the *reduce* operates on an iterator of values because the set of values is typically far too large to fit in memory. This means that the reducer can only partially use correlations between these items while processing: it receives them as a stream instead of a set. In this example, operating on the stream is trivial, since the reducer simply increments the counter for each item.
- the *reduce* operates on all pieces of data that share some key. By assigning proper keys to data items during the *map*, the data is partitioned for the *reduce* phase. A skewed partitioning (i.e. skewed key distribution) will lead to imbalances in the load of the compute nodes. If term x is relatively popular the node performing the *reduce* for term x will be slower than others. To use MapReduce efficiently, we must find balanced partitions of the data.
- since the data resides on local nodes, and the physical location of data is known, the platform performs *locality-aware scheduling*: if possible, *map* and *reduce* are scheduled on the machine holding the relevant data, moving computation instead of data. Remote data can be accessed through a Distributed File System(DFS) maintained by the framework.