RAPID: Enabling Scalable Ad-Hoc Analytics on the Semantic Web

Radhika Sridhar, Padmashree Ravindra, and Kemafor Anyanwu

North Carolina State University
{rsridha,pravind2,kogan}@ncsu.edu

Abstract. As the amount of available RDF data continues to increase steadily, there is growing interest in developing efficient methods for analyzing such data. While recent efforts have focused on developing efficient methods for traditional data processing, analytical processing which typically involves more complex queries has received much less attention. The use of cost effective parallelization techniques such as Google's Map-Reduce offer significant promise for achieving Web scale analytics. However, currently available implementations are designed for simple data processing on structured data.

In this paper, we present a language, RAPID, for scalable ad-hoc analytical processing of RDF data on Map-Reduce frameworks. It builds on Yahoo's Pig Latin by introducing primitives based on a specialized join operator, the MD-join, for expressing analytical tasks in a manner that is more amenable to parallel processing, as well as primitives for coping with semi-structured nature of RDF data. Experimental evaluation results demonstrate significant performance improvements for analytical processing of RDF data over existing Map-Reduce based techniques.

Keywords: RDF, Scalable Analytical Processing, Map-Reduce, Pig Latin.

1 Introduction

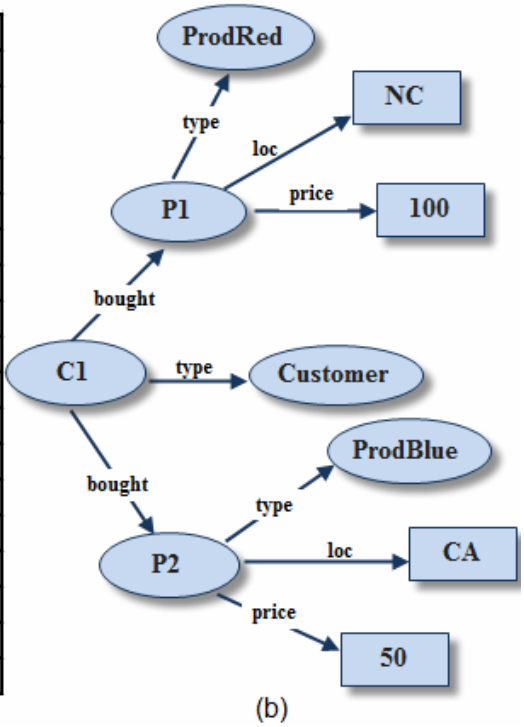
The broadening adoption of Semantic Web tenets is giving rise to a growing amount of data represented using the foundational metadata representation language, Resource Description Framework (RDF) [19] In order to provide adequate support for knowledge discovery tasks such as exists in scientific research communities, many of which have adopted Semantic Web technologies, it is important to consider how more complex data analysis can be enabled efficiently at Semantic Web scale. Analytical processing involves more complex queries than traditional data processing often requiring multiple aggregations over multiple groupings of data. These queries are often difficult to express and optimize using traditional join, grouping and aggregation operators and algorithms. The following two examples based on the simple Sales data in Figure 1 illustrate the challenges with analytical queries. Assume we would like to find "*for each customer*, *their total sales amounts for Jan, Jun and Nov for purchases made in the state NC*", i.e., to compute the relation (*cust, jansales, junsales, novsales*).

A. Bernstein et al. (Eds.): ISWC 2009, LNCS 5823, pp. 715–730, 2009.

Using traditional query operators this query will be expressed as a union query, resulting in three sub queries (each computing the aggregates for each of the months specified) and then an outer join for merging the related tuples for each customer. Each of these sub queries will need a separate scan of the same typically large table. A slightly more demanding example would be to find *"for each product and month of 2000, the number of sales that were between the previous and following months' average sales"*. Computing the answer to this query requires that for each product and month, we compute aggregates from tuples outside the group (the next and previous month's average sales). After these values are computed, we have enough information to compute the output aggregate (count). This query also requires multiple pass aggregation with a lot of repeated processing of the same set of tuples such as repeated scans on relations just to compute slightly different values e.g. previous month aggregate vs. next month aggregates. High-end database systems and OLAP servers such as Teradata, Tandem, NCR, Oracle-n CUBE, and Microsoft and SAS OLAP servers with specialized parallel architectures and sophisticated indexing schemes employ techniques to mitigate this inefficiency. However, such systems are very expensive and are targeted at enterprise scale processing making it difficult to scale them to the Web in a straightforward and cost effective way.

Sub	Prop	Obj
C1	TYPE	CUSTOMER
C1	BOUGHT	P1
P1	LOC	NC
P1	PRICE	100
P1	MONTH	JAN
C1	BOUGHT	P2
P2	LOC	CA
P2	PRICE	50
P2	MONTH	JUNE
C1	BOUGHT	P4
P4	LOC	NC
P4	PRICE	50
P4	MONTH	JUNE
C2	TYPE	CUSTOMER
C2	BOUGHT	P3
P3	LOC	MC
P3	PRICE	100
P3	MONTH	NOV

(a)



(b)

Fig. 1. RDF representation of Sales relation

One promising direction is to leverage the recent advancements made by search engine companies that utilize parallel data processing frameworks based on clusters of commodity grade machines that can be scaled easily and cost effectively for processing at Web scale. Examples are Google's *Map-Reduce [6]* framework and an open source version developed by Yahoo called *Hadoop [18]*. These frameworks also offer the benefit of allowing users abstract from low level parallelization issues, greatly simplifying the process of writing parallel programs for their tasks. In furtherance of the ease-of-use goal, a high level dataflow language called Pig Latin [12] in the spirit of SQL has been developed for Map-Reduce data processing tasks. Such a high level language offers clear advantages like automatic optimization and code reuse over the traditional Map-Reduce framework that relies on a black box approach in which users writing their own implementations for their tasks making it difficult to automatically optimize and reuse them. However, the current Pig Latin language is targeted at processing structured data, limiting its use for semi-structured data such as RDF. Further, it currently provides only a limited set of primitives that is insufficient for efficient expression of complex analytical queries. This limitation leads to the earlier mentioned problem of avoidable multi-pass aggregations. As an example, encoding our first example in a Pig Latin program yields a 10-step whereas the approach that we propose here results in a 3 step program. This not only improves usability of such systems but also leads to a significant performance savings.

Related Work. The earlier generation RDF storage engines were architected either as main memory stores [3][5] or layered on top of relational databases [2][4][15][16]. These enabled enterprise-scale performance for traditional data processing queries but would be challenged by Web scale processing and analytics-style workloads. More recently, native stores have been developed with a focus on achieving efficient Web scale performance using specialized storage schemes. Some notable ones include (i) vertically partitioned column stores [1] which combine the advantages of a vertically partitioned scheme for property bound queries with the compressibility advantages of column-oriented stores; (ii) multi-indexing techniques [14] that use multiple indexes on triples to produce different sort orders on the different components of a triple, thereby trading off space for the possibility of utilizing only fast merge joins for join processing. In [11], the multi-indexing approach was combined with cost-based query optimization that targets optimizing join-order using statistical synopses for entire join paths. Multi-indexing techniques have also been combined with distributed query processing techniques in [9] for processing queries in federated RDF databases. However, these techniques may pose limitations for ad-hoc processing on the Web, because they require that RDF document content be exported into a database systems for preprocessing (index construction) before data processing can occur. Further, as analytical queries involve filter, join as well as multiple aggregations over different grouping, the indexes will only confer an advantage on the filter and join operations, while the aggregations over groupings would need to occur on the intermediate results that are not indexed. Finally, as we shall see later, the nature of query operators used in the query expression plays a significant role in optimizability of such complex queries.

The second line of work that is relevant is that based on extending Google's Map-Reduce parallel processing framework with support for data processing tasks. The original Map-Reduce framework is designed to work with a single dataset which creates a problem when we would need to perform binary operations like `JOIN` operations. There are methods for overcoming this limitation by doing what are known as *Map-Side* or a *Reduce-Side Joins* respectively. However, the rigid Map-Reduce execution cycle of a Map phase followed by a Reduce, forces such join operations to only be realizable with additional Map-Reduce cycles in which some phases perform no-op. Such wasted cycles lead to inefficiencies and motivated an extension of the Map-Reduce framework called *Map-Reduce-Merge* [17] framework. In this programming model, an additional *merge* phase is added to the typical 2-phase cycle, where the `JOIN` operation is performed between the two sets of data that are partitioned and grouped together in the *map* and the *reduce* phases. However, in these approaches, users bear the responsibility of implementing their tasks in terms of Map and Reduce functions which makes them less amenable to automatic query optimization techniques and offers limited opportunity for code reuse. Pig Latin[12] is a high level dataflow language that overcomes this limitation by provide dataflow primitives with clear semantics similar to query primitives in declarative query languages. However, the set of primitives provided are still limited to basic data processing of structured relational data. Some techniques [8][10] have been developed for processing RDF data at Web scale based on the Map-Reduce platform. However, these efforts have primarily focused on graph pattern matching queries and queries involving inferencing and not analytical queries requiring complex grouping and aggregation.

Contributions and Outline. The goal of the work presented here is to offer a framework for scalable ad-hoc analytical processing of RDF data that would be easy to use, cost effective and directly applicable to RDF document sources without requiring expensive preprocessing. The approach is based on integrating RDF-sensitive and advanced analytical query operators into Map-Reduce frameworks. Specifically, we make the following contributions:

- We propose a dataflow language, RAPID, which extends Yahoo's Pig Latin language with query primitives for dealing with the graph structured nature of RDF.
- We propose a technique for efficient expression of complex analytical queries that offers better opportunities for optimization and parallelization on Map-Reduce platforms. The technique is based on integrating into Pig Latin, primitives for implementing an efficient query operator for analytical processing called MD-join [6].
- We present evaluation results on both synthetically generated benchmark datasets as well as real life dataset.

The rest of the paper is organized as follows: Section 2 gives an overview of analytical queries using a sophisticated operator called MD-join and introduces the Map-Reduce framework and the data flow language, Pig Latin. Section 3 presents the RAPID approach and section 4 shows its experimental evaluation. Section 5 concludes the paper.

2 Preliminaries

2.1 MD-Join and Analytical Querying

Earlier examples showed that expressing complex analytical queries using traditional query operators can lead to inefficiencies due to redundant table scans. One cause of this problem identified in [6] is the tight coupling of the GROUPBY and *aggregation clauses* in query expressions that makes it difficult for the optimizer to identify and coalesce related computations. This led to the proposal of a hybrid join and aggregation operator called the `MD-Join` [6] which decouples the *grouping* and the *aggregation clauses* in query expressions. The MD-Join operator `MD(B, R, l, Θ)` defines a relation with schema B (*base table*), R is the *fact table*, *l* is a list of aggregation functions that need to be computed over attributes and Θ is a set of conditions involving the attributes of B and R. The *fact table* contains all the data that needs to be aggregated and the *base table* contains all combinations of key values representing each group on which aggregations have to be computed. Using this operator, we can express the second example query in the following way:

MD(MD(MD(B, R, avg(sales), θ1), R, avg(sales), θ2), R, count(sales), θ3)
where B – Base relation (the list of Product-Month combinations)
R – Fact table (Sales relation)
θ – the list of conditions
θ1 – R.ProdId = ProdId and R.Month = Month+1 and R.year = "2000"
θ2 – R.ProdId = ProdId and R.Month = Month-1 and R.year = "2000"
θ3 – R.ProdId = ProdId and R.sales >= prev_month_avg and
R.sales <= next_month_avg and R.year = "2000"

Fig. 2. Expressing second example query using the MD-join

This example also shows the MD-joins can be nested where the result of each MD-Join forms the base relation for the outer MD-Join. The details of the algorithm we omit for brevity but shown by way of an example.

Table 1. (a) Fact table representing Sales data (b) Base table being updated

Cust	Prod	Month	Year
C1	P1	JAN	2008
C1	P2	JUNE	2007
C1	P4	JUNE	2009
C2	P2	JUNE	2007
C2	P3	NOV	2006

(a)

Prod	Month	Sum(Price)
P1	JAN	25
P2	JUNE	35 70
P2	DEC	NULL
P3	NOV	NULL
P4	JUNE	50

(b)

Figure 1 is the fact table consisting of the *Sales* data. Each tuple in the Base table is one of the desired groups consisting of a *Product-Month* combination. The algorithm scans the fact table R and loops over all tuples of Base table *B* to identify matches based on a condition θ. If a match is found, the appropriate aggregate columns are updated. Table 1 (a) shows an example fact table and Table 1 (b) shows the base table for *Product*, *Month* groups and a placeholder for the `Sum`(*Price*) aggregate initially set to NULL. The arrows show the tuples that have just been processed by the MD-join algorithm. The group (*P2*, *FEB*) which had an original sum of *35* based on the second tuple is now updated to *70* after the price of the next tuple (being pointed to) in that group is processed in the fact table. One attractive property of the MD-join operator is that is easily amenable to parallelization. This paper primarily focuses on how to achieve that in the context of Map-Reduce frameworks.

2.2 Data Processing in Map-Reduce Framework

In the Map-Reduce programming model, users encode their tasks in terms of two functions: the *Map* and the *Reduce*. These functions are independent and the execution of each can be parallelized on a cluster of machines. The *Map* function groups together related data items e.g. records with same key values, and the *Reduce* function focuses on performing aggregation operations on the grouped data output by the Map function. The (key, value) mappings can be represented as:

$$\textbf{Map}\ (\ \textbf{k1, v1}\ ) \rightarrow \textbf{list}\ (\ \textbf{k2, v2}\ )\ \textbf{and}\ \ \textbf{Reduce}\ (\ \textbf{k2, list}\ (\ \textbf{v2})) \rightarrow \textbf{list}\ (\ \textbf{v2}\ )$$

Recently, Pig Latin, a high level dataflow language was proposed to overcome these limitations. It achieves a sweet spot between the declarative style of the languages like SQL and the low level procedural style of the Map-Reduce programming. It provides a set of predefined functions and query expressions that can be used to describe the data processing tasks. Its data model consists of an atom that holds a single atomic value, a *tuple* that holds a series of related values, a *bag* that forms a collection of tuples and a *map* that contains collection of key value pairs. A tuple can be nested to an arbitrary depth. Table 2 provides an example of a nested tuple *t* with fields' *f1*, *f2* and *f3* with *f2* containing tuples. It also shows data expressions for retrieving components of the tuple *t*.

Table 2. Expressions in Pig Latin

$t = \left(C1, \begin{Bmatrix} NC, 25 \\ CA, 35 \end{Bmatrix}, P1 \right)$		
Expression type	**Example**	**Value for t**
Field by position	$3	P1
Field by name	f1	C1
Projection	f2.$1	{(25), (35)}
Function evaluation	SUM(f2.$1)	25 + 35 = 60

The other functions offered by Pig Latin language are `LOAD` that implements the reading an input file, `FOREACH` which is used as an iterator to loop through a collection, `FILTER` for filtering out tuples, the `JOIN` function, grouping functions like `GROUP` and `COGROUP`, and other common commands reminiscent of SQL. In addition to these primitives, the language also allows supports User Defined Functions (UDFs).

3 The RAPID Language

This section introduces a dataflow language RAPID that enables complex manipulation of RDF graphs and builds on Pig Latin and the MD-Join operator. It includes primitives for describing nodes, their properties and associated paths, as well as primitives for expressing queries in terms of MD-joins. The model of RAPID is shown in Figure 3.

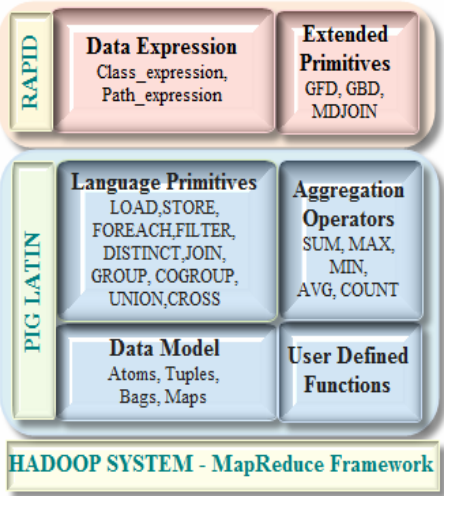


Fig. 3. Architecture of RAPID

3.1 Primitives for Basic Manipulation of RDF Graphs in RAPID

RAPID includes *Class* and *Property* expression types that are very natural for querying RDF data. It also supports *Path expressions* – defining resources in terms of their class or property types or the navigational patterns that they are reachable by.

Table 3. Expressions for manipulating RDF data

Expression	Example	Answer
Class expressions	type: Customer	{C1}
Property expressions	bought	{P1, P2 ..}
Path expressions	C1.bought.price	{25, 35}

Class expressions consist of the list of classes that are in desired groups e.g. *Customer*, *Product* denoted as *type:Class*. These expressions evaluate to the resources of type Class. Property expressions list the properties that are used in the query and evaluate to the list of objects of that property. Finally, path expressions specify navigational patterns required to access the resources participating in desired aggregations. Table 3 summarizes these expressions and shows examples based on the example data in Figure 1.

3.2 Primitives for Analytical Processing

In order to implement the MD-Join operation, we introduce three functions `GFD`, `GBD` and `MDJ` for computing the fact table, base table and MD-Join respectively.

3.2.1 Generating Fact Table (GFD)

The role of the GFD function is to reassemble related triples that represent the n-ary relationships typically stored in an OLAP fact table. However, since a query may only involve some parts of the relationship, the GFD function allows users to specify, using path expressions, the specific parts (eq. to columns) of the relationships that are required for grouping or aggregation. Desired filtering conditions (eq. rows) are specified in a similar way. The function assumes that the input RDF document is in the N3-like format where triples of the form *<Subject, Property, Object>* are separated by a special character E.g. ':'. The `GFD` function is passed as a parameter to the Pig Latin's `LOAD` command which uses it as a file handler to reads an RDF file and generate fact table tuples. An example command is shown below:

fact_dataset = LOAD 'input.rdf' USING GFD (class_expression; property_expression; aggregation_pathExpression;filter_pathExpression);

where "input.rdf" is the RDF file to be loaded, *Class_Expression* lists the classes for elements of each group, property_expressions indicate the properties whose range values need to be aggregated.

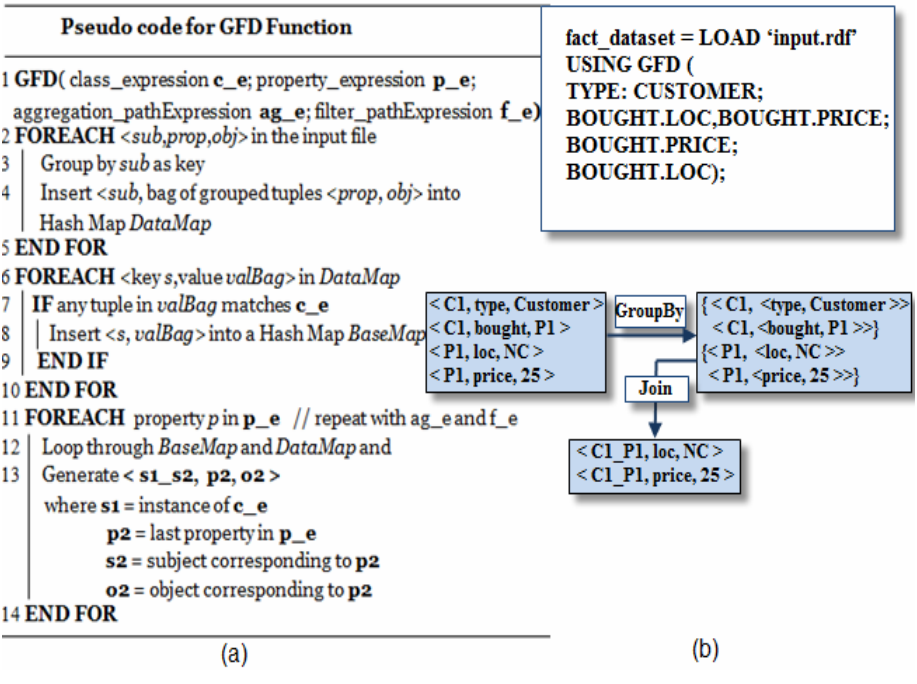


Fig. 4. GFD Function (a) Pseudo code (b) Execution

The *filter_pathExpression* and *aggregation_pathExpression* describe the navigational paths leading to the nodes that need to be filtered / aggregated. The GFD loads those triples involved in the expressions in the query and groups them using the subject values as keys. Then it performs the necessary join operations using the Pig

Latin JOIN operator (indicated by the path expressions) to reassemble the connections between values**.** Figure 4 (a) shows the pseudo code for the GFD function. Figure 4 (b) shows the steps in executing GFD.The output consists of triples of the form <*Subject*, *Property*, *Object*> where *Subject* is a canonical representation of key value generated using the path expression. In the output, the key *C1_P1* represent the nodes *C1* and *P1* leading to pairs (*loc, NC*) and (*price, 25*). The output of this function is the fact table and is stored using Pig Latin's STORE command and subsequently retrieved as an input to the MDJ function. The following section discusses this in more detail.

Map-Reduce Workflow for GFD Operator. In this section, we discuss the details of the workflow of GFD operator in the context of the Hadoop environment. The Hadoop system consists of one *Job Tracker* which acts as a master process, reads input data, divides this dataset into chunks of equal size and assigns these chunks of data to each of the *Task Trackers*. *Task Trackers* are processors that are designed to perform the *map* or the *reduce* functions called the *Mapper* and *Reducer* respectively.

Mapper Design. The *Job Tracker* assigns each Mapper process with a chunk of the dataset, using which, the Mapper generates <*key*, *value*> pairs as output. Figure 5 (a) shows the pseudo code for the *map* function and Figure 5 (b) shows the output generated by the *map* function.

Pseudo code for Map Function

```
Input: Key: File Name
       Value: Chunk of data
1   Map (String Map, String Value)
2       For Each line in the value
            // line is of the form <Subject, Property, Object>
3           Output (<Subject, <Property, Object>>
4       End of For
5   End of Map
```

(a)

Subject	<Property, Object>
C1	TYPE, CUSTOMER
C1	BOUGHT,P1
P1	LOC, NC
....	

(b)

Fig. 5. Map function (a) Pseudo code (b) Result

A Map function contains a subroutine called the *Combiner.* All the *objects* having the same *key* are grouped together into a collection in the Combiner function. If the query contains any user specified filter conditions, then the corresponding <*key*, *Collection of values*> are filtered out based on the given condition in the Combiner function. In [6], the authors show that the tuples for which the filter condition is not true will never be considered by the `MD-Join` so these tuples can be eliminated from the dataset. This reduces the number of records that need to be processed in the *reducer* function, thus increasing the efficiency of processing.

Given the data in Figure 1, suppose the user wants to compute the total price for each customer and the product. Figure 6 (b) shows the result after the execution of the Combiner code. The subject column in the Figure 6 (b) shows how the Combiner function combines multiple properties to generate the composite key when the user query involves multi-dimensional groups. Figure 6 (a) shows the pseudo code for the reducer implementation.

Pseudo code for Combiner Function

Input: MapOutput – Which has the output of the Map function in the form <subject, <Property, Object>>

```
1   Combiner (Collection MapOutput)
2     FOREACH Subject in the Output
3       Get all the records with the same Subject
4       IF any filter condition
5         FOR EACH record
6           validate the filter condition
7         END FOR
8       END IF
9       IF Aggregation is on Multidimensional Properties
10        Key: Generate composite key based on the
                Multi-dimensional properties
11        Value: <Property, Object>
12      ELSE
13        Key: Subject
14        Value: <Property, Object>
15      END IF
16    Output (<Key, [Value]>)
17    END OF FOR
18  END OF Combiner
```

(a)

Subject	Value
C1_P1	[<Price, 100>, <Price,105>]
C1_P2	[<Price, 50>]
C1_P4	[<Price,50>]
C2_P3	[<Price,100>]
....	

(b)

Fig. 6. Combiner function (a) Pseudo code (b) Result

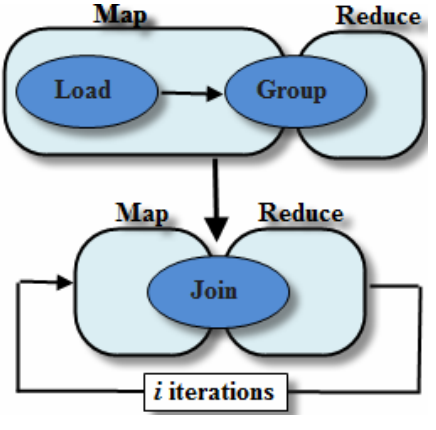


Fig. 7. Compilation of GFD on Map-Reduce framework

Reducer Design. The set of keys and the collection of values obtained from the Combiner function is the input for the reducer function. Each reducer will have a set of tuples. Same keys for the tuples are grouped together and all the corresponding properties of these keys are collected together as part of a reducer function. Algorithm for the reducer function is similar to that of the Combiner function, except that the data for the Combiner function is limited from the corresponding Mapper process, while the data for the reducer function could be from the output of various Mapper/Combiner functions. Figure 7 shows the compilation of the GFD operator on Map-Reduce framework.

3.2.2 Generating Base Table (GBD)

The role of the base table in the MD-Join algorithm is to maintain container tuples representing each group for which aggregations are desired. For every tuple in the fact dataset, the corresponding combination in the base dataset is obtained and the aggregation results are updated in the container tuples. Similar to GFD, the GBD operator is executed with the LOAD function and the class_expression and property_expression play similar roles.

base_dataset = LOAD 'input.rdf' USING GBD (class_expression; property_expression; FLAG);

The FLAG holds either the value "NULL" or "BOTH" indicating whether the aggregation is to be computed on properties got from the property_expression or a combination of the property value and the type class. The tuples generated by the GBD are of the type <Subject, Base, NULL> where "Base" is a keyword that indicates that the tuple belongs to the base table. The NULL value acts as placeholder for aggregates that will be computed by the MD-join step. Figure 8 shows the steps in executing GFD for the given example

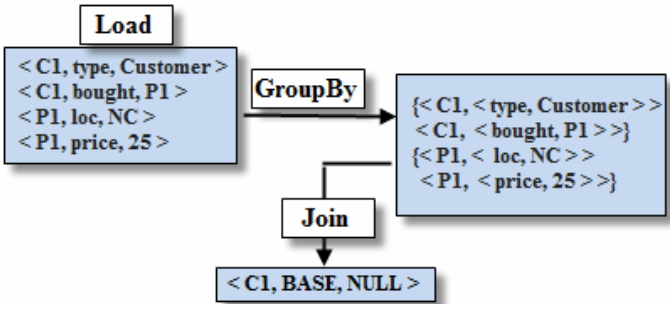


Fig. 8. Execution of GBD

Within the GBD function, we call the STORE function to append the base dataset into the same MDJ.rdf file. This file is later loaded by the MDJ operator while performing the JOIN operation, which is discussed in the next section.

base_dataset = LOAD 'input.rdf' USING GBD (TYPE: CUSTOMER; BOUGHT.LOC, BOUGHT.PRICE; NULL);

Map-Reduce workflow for GBD operator. The Map-Reduce function for the GBD operator is very similar to the GFD operator. The difference is in the Reduce function for these two operators. In the Reduce function of the GBD operator, after the <Key, <Collection>> of values is grouped together. For each key and its group; a corresponding <Key, <Base, Null>> record is created. Thus the output from the reduce function is the set of base tuples that are required for the MDJ operation. The compilation of GBD operator is similar to GFD operator as show in Figure 7.

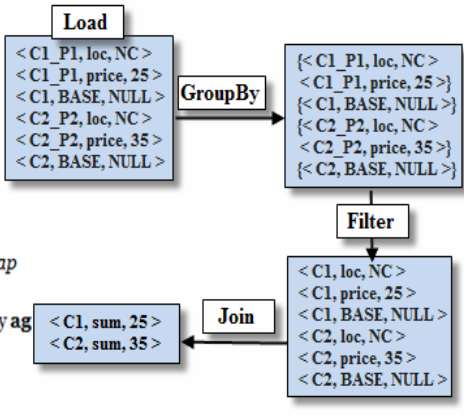
3.2.3 Multi-dimensional Join in RAPID

In [6], it was shown that if you partition the base table, execute the MD-Join on each partition and union the results, the result is equivalent to executing the MD-Join on the non-partitioned base table. This result leads to natural technique for parallelizing the MD-join computation. It is possible to derive an analogous result to the partitioning of the fact table as well. Due to this limited set of fact and base data records in each partition, the execution of the MD-Join algorithm is much faster because for each fact record we will only iterate through all the base records having the same Key as the fact record. More formally, assume that we can partition the base table B into $B_1 \cup B_2 \ldots \cup B_n$ where $B_i = \sigma_i(B)$, i.e. σ_i is the range condition that tuples in B_i satisfy. Then, $MD(\sigma_i(B), R, l, \Theta) = MD(\sigma_i(B), \sigma'_i(R), l, \Theta)$. In other words, the result of an MD-join of a member of the partition of the base table say B_i, and the entire fact table is equivalent to B_i and the corresponding partition of the fact table - $\sigma'_i(R)$. The result of the GFD and GBD operators seen in the above sections generates the input dataset for the MDJ operator in such a way that efficient partitioning of the records is possible to perform MDJ operation in parallel. Hence each Map processor will receive one set of records containing both the fact and the base dataset with the same keys set. The "MDJ.rdf" file created by the GFD and GBD operators as mentioned in 3.2.1and 3.2.2, contains fact and base tuple sets. The MDJ operator executes on these datasets and also takes as input the filter condition on which the aggregation needs to be computed and the aggregation function such as the SUM, COUNT, MAX, MIN, AVG. As is shown in the pseudo code for MDJ operator in Figure 9 (a), when there is a match between the base and fact tuple, the aggregation is computed and the base dataset is updated. The grouping of related datasets is

Pseudo code for MDJ Function

```
MDJ (class_name class; aggregation_function :
aggregation_property ag; filter_property: filter_condition f_c )
1  FOREACH t<key, prop, obj > in output of GFD, GBD functions
2    IF prop == BASE
3      Insert t into a Hash Map BaseTableMap
4    ELSE
5      Insert t into a Hash Map FactTableMap
6    END IF
7  END FOR
8  FOREACH  <key, valBag> in FactTableMap
9    Filter based on f_c
10 END FOR
11 FOREACH entry in FactTableMap and BaseTableMap
12   IF FactTableMap.Key == BaseTableMap.Key
13     Compute the Aggregation Function on property ag
14     Update the BaseTableMap with the new value
15   END IF
16 END FOR
```

(a)



(b)

Fig. 9. MDJ Function (a) Pseudo code (b) Execution

separated into the map function while the aggregation on this grouped data is computed within the reducer function. The syntax for MDJ operator is as follows:

output_dataset = LOAD 'MDJ.rdf' USING MDJ (class_name;
aggregation_func: aggregation_property; filter_property: filter_condition);

Figure 9 (b) shows the steps in executing MDJ for the given example. The output generated is stored in an output file using the Pig Latin primitive the STORE function

output_dataset = LOAD 'MDJ.rdf' using MDJ(CUST; SUM:PRICE; LOC:NC);

Map-Reduce workflow for MDJ operator. When performing the LOAD operation for MDJ we load both sets of data (the fact and base dataset) in the *map* function. The reduce function will execute the pseudo code as shown in Figure 9 (a). The compilation of MDJ on Map-Reduce is show in Figure 10.

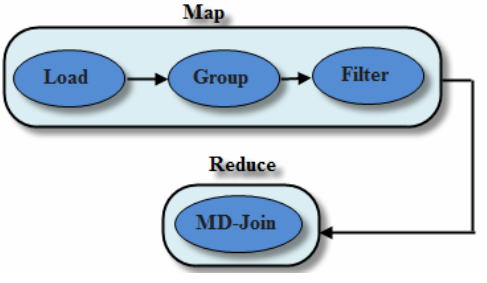


Fig. 10. Compilation of MDJ on Map-Reduce framework

4 Experiments and Results

Our experimental setup involved clusters of 5 nodes where each node is either a single or duo core Intel X 86 machines with 2.33 GHz processor speed and 4G memory and running Red Hat Linux. Our test bed includes a real life dataset, SwetoDBLP [20] and a synthetic dataset generated by the benchmark dataset generator called BSBM [21]. The queries used for the evaluation consists of 5 queries for each dataset.

4.1 Datasets and Results

SWETO dataset contains an RDF representation of the bibliographic data in DBLP such as information about Authors, Books and other types of Publications, Year of Publication, etc. Table 4 shows the set of queries for the DBLP dataset.

Table 4. DBLP Query table

Query 1: Compute the total number of books for all combinations of author-year-publication
Query 2: Compute the total number of book for all combinations of author-publication in the year "2000"
Query 3: Compute the number of books for each author for the year "1999", "2003" and "2007"
Query 4: Compute the total number of books published where the count is greater than the average count for all combinations of author-publisher-month during the year "2008"
Query 5: Compute the total number of books published where the number of books published for all combinations of author-year-publication is greater than 50

The Berlin SPARQL Benchmark (BSBM) is a synthetic benchmark dataset for evaluating the performance of SPARQL queries. The dataset contains information about the Vendor, the Offers and the Product types and the relationships between them. Table 5 shows the set of queries executed on the BSBM dataset were used for evaluation of the two approaches.

Table 6 compares the reduction in the table scans when using RAPID as the number of GROUPBY and JOIN operation reduces. This results in more scalable and efficient approach for processing analytical queries. Figure 11 and Figure 12 (b), show a comparison of the execution times for the two approaches on DBLP and BSBM dataset respectively. Figure 12 (a) shows the number of sub-queries required for each query. The figure clearly shows that the RAPID approach offers better usability. The reduced number of steps in the RAPID approach is due to the coalescing of JOINS and

Table 5. BSBM Query table

Query 6: Compute the number of offers present for all combinations of country-vendor- product-productType
Query 7: Compute the sume of price for each offer for all combinations of country-product for the month "Jan"
Query 8: Compute the number of offers made for each product in the country "US","India","China" and "Japan"
Query 9: Compute the number of offers present where the total price of the offers for all combinations of country-vendor-product-productType is less than "$1000"
Query 10: Compute the number of offers made by vendors which were above the average offer, when the data is viewed from all combinations of productType-validTo-country

Table 6. Number of full table scans for Pig and RAPID operations

Query	Pig	RAPID	Query	Pig	RAPID
Query 1	16	9	**Query 6**	32	13
Query 2	16	9	**Query 7**	16	9
Query 3	17	11	**Query 8**	17	11
Query 4	27	10	**Query 9**	52	12
Query 5	27	10	**Query 10**	27	10

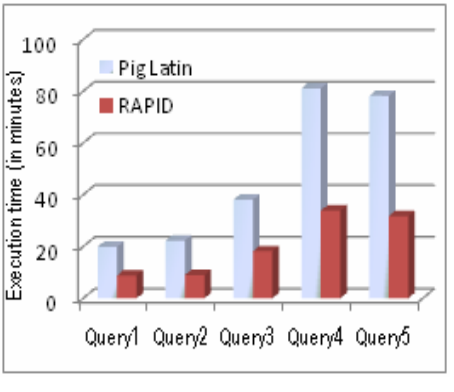


Fig. 11. Cost analysis on DBLP dataset based on execution time (In minutes)

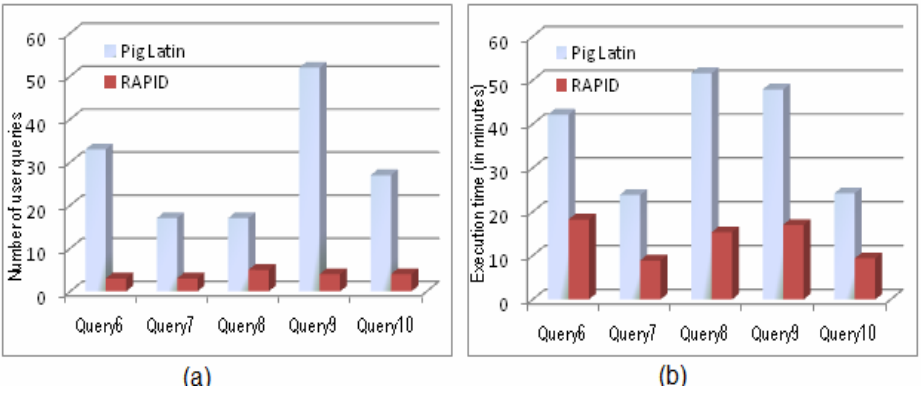


Fig. 12. Cost analysis on BSBM dataset based on (a) Number of user queries (b) Execution time (In minutes)

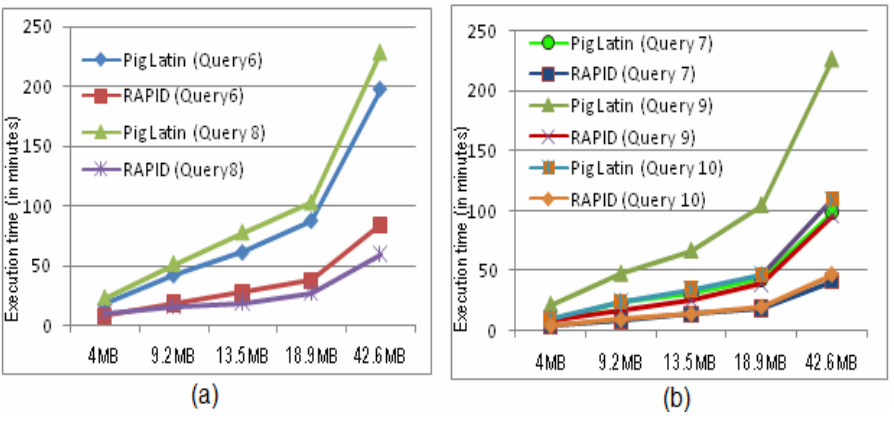


Fig. 13. Scalability evaluation for different file sizes of BSBM dataset (a) Query 6,8 (b) Query 7,9,10

GROUPBY operations by the MDJ operation. Figure 13 (a) and Figure 13 (b) show that RAPID approach scales more gracefully with increasing size of dataset.

5 Conclusion

In this paper, we present an approach for scalable analytics of RDF data. It is based on extending easy-to-use and cost-effective parallel data processing frameworks based on Map-Reduce such as Pig Latin with dataflow/query operators that support efficient expression and parallelization of complex analytical queries as well as for easier handling of graph structured data such as RDF. The results show significant query performance improvements using this approach. A worthwhile future direction to pursue is the integration of emerging scalable indexing techniques for further performance gains.

References

[1] Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable Semantic Web Data Management Using Vertical Partitioning. In: Proc. of VLDB 2007, pp. 411–422 (2007)

[2] Alexaki, S., Christophides, V., Karvounarakis, G., Plexousakis, D., Tolle, K.: The ICS-FORTH RDFSuite: Managing voluminous RDF description bases. In: SemWeb (2001)

[3] Beckett, D.: The design and implementation of the Redland RDF application framework. In: WWW (2001)

[4] Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A generic architecture for storing and querying RDF and RDF Schema. In: Horrocks, I., Hendler, J. (eds.) ISWC 2002. LNCS, vol. 2342, p. 54. Springer, Heidelberg (2002)

[5] Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: implementing the Semantic Web recommendations. In: WWW (2004)

[6] Chatziantoniou, D., Akinde, M., Johnson, T., Kim, S.: The MD-join: an operator for Complex OLAP. In: ICDE 2001, pp. 108–121 (2001)

[7] Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Proc. of OSDI 2004 (2004)

[8] Erling, O., Mikhailov, I.: Towards Web Scale RDF. In: 4th International Workshop on Scalable Semantic Web Knowledge Base Systems, SSWS 2008 (2008)

[9] Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In: Aberer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L.J.B., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P. (eds.) ASWC 2007 and ISWC 2007. LNCS, vol. 4825, pp. 211–224. Springer, Heidelberg (2007)

[10] Newman, A., Li, Y., Hunter, J.: Scalable Semantics – The Silver Lining of Cloud Computing. eScience, 2008. In: IEEE Fourth International Conference on eScience 2008 (2008)

[11] Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. PVLDB 1(1), 647–659 (2008)

[12] Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig Latin: a not-so-foreign language for data processing. In: Proc. of ACM SIGMOD 2008, pp. 1099–1110 (2008)

[13] Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., DeWitt, D.J., Madden, S., Stonebraker, M.: A Comparison of Approaches to Large-Scale Data Analysis. In: Proc. of SIGMOD 2009 (2009)

[14] Weiss, C., Karras, P., Bernstein, A.: Hexastore: Sextuple Indexing for Semantic Web Data Management. In: Proc. of VLDB (2008)

[15] Wilkinson, K.: Jena property table implementation. In: SSWS (2006)

[16] Wilkinson, K., Sayers, C., Kuno, H.A., Reynolds, D.: Efficient RDF storage and retrieval in Jena2. In: SWDB (2003)

[17] Yang, H., Dasdan, A., Hsias, R.-L., Parket, D.S.: Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In: Proc. SIGMOD 2007, pp. 1029–1040 (2007)

[18] Apache Projects Proceedings, `http://hadoop.apache.org/core/`

[19] W3C Semantic Web Activity Proceedings, `http://www.w3.org/RDF/`

[20] Swetodblp, `http://lsdis.cs.uga.edu/projects/semdis/swetodblp/`

[21] BSBM, `http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/spec/index.html#dataschema`