EAGRE: Towards Scalable I/O Efficient SPARQL Query Evaluation on the Cloud

Xiaofei Zhang #1, Lei Chen #2, Yongxin Tong #3 , Min Wang *4

#Dept. of Computer Science & Engineering, HKUST
Clear Water Bay, Kowloon, HKSAR
{ 1zhangxf, 2leichen, 3yxtong}@cse.ust.hk

*HP Labs China
Beijing, China
4min.wang6@hp.com

***Abstract*— To benefit from the Cloud platform's unlimited resources, managing and evaluating huge volume of RDF data in a scalable manner has attracted intensive research efforts recently. Progresses have been made on evaluating SPARQL queries with either high-level declarative programming languages, like Pig [1], or a sequence of sophisticated designed MapReduce jobs, both of which tend to answer the query with multiple join operations. However, due to the simplicity of Cloud storage and the coarse organization of RDF data in existing solutions, multiple join operations easily bring significant I/O and network traffic which can severely degrade the system performance. In this work, we first propose *EAGRE*, an *E*ntity-*A*ware *G*raph comp*RE*ssion technique to form a new representation of RDF data on Cloud platforms, based on which we propose an I/O efficient strategy to evaluate SPARQL queries as quickly as possible, especially queries with specified solution sequence modifiers, e.g., `PROJECTION`, `ORDER BY`, etc. We implement a prototype system and conduct extensive experiments over both real and synthetic datasets on an in-house cluster. The experimental results show that our solution can achieve over an order of magnitude of time saving for the SPARQL query evaluation compared to the state-of-art MapReduce-based solutions.**

I. Introduction

As one of the W3C standards for describing web resources and meta data, RDF (Resource Description Framework) is designed as a flexible representation of schema-relax or even schema-free information for the Semantic Web. Along with increasing supports from prevailing search engine projects, like RichSnippets from Google and SearchMoney from Yahoo!, as well as the willingness to integrate across-domain knowledge, there emerges a huge volume of public RDF data for management and analysis. For example, the largest RDF dataset (Billion Challenge 2010) available in the Linked Data community[1] has over 3.2 billion triples, and the second largest dataset containing various bio- and gene- related data (Bio2RDF) has over 2.7 billion triples. The increasing demands of massive data-intensive RDF data analysis and the great scalability of Cloud platforms have made them the nail and the hammer. Although various efforts have been made to explore the effective analysis of large RDF data on Cloud platforms via RDF-specific querying interface, SPARQL[2], the query time efficiency remains a bottleneck to forward Cloud-based RDF services in the real world.

As a matter of fact, the huge gap between the simplicity of RDF data and the complexity of queries and inferences that people want to perform over the data, raises the essential challenge of RDF data management and analysis. Most of the state-of-art centralized systems address the problem by taking advantages of heavy or even exhaustive indexes or semantic based data partition to answer RDF queries. For example, Jena[3] introduces property tables obtained through pattern mining techniques. RDF-3X [2] takes the advantage of extensive set of statistics obtained from data preprocessing. T. Neumann and et. al. further improve RDF-3X in [3] and [4] by adopting a different join order selection strategy and introducing side-way information passing to filter out undesired data as early as possible. However, all the above techniques rely on effective data preprocessing and the global view of the entire dataset, which make their solutions hardly scalable for distributed RDF data of large scale. Although distributed solutions for RDF query processing and storage have been proposed such as [5][6][7], these proposals are built upon self-defined computing policies and specific distributed or parallel databases, which limit their expandability to conduct the cross-platform integration and become an open standard.

Although the Cloud platform promises great scalability and "unlimited" resources, sophisticated scheduling of parallel jobs for query evaluation remains a challenging problem. The most recent works, like [8][9][10][11] and [12], basically look into the same general problem, how to map a query to a number of MapReduce jobs and how to schedule the jobs to achieve different optimization goals. However, as shown in the experiments results in these works, it can easily take hundreds of seconds to evaluate a query, which makes it impractical to deliver the RDF query services in the real world. The drawbacks come from two aspects. First, existing efforts heavily rely on the simple (*key,value*) storage format

[1]http://linkeddata.org/

[2]http://www.w3.org/TR/sparql11-query/

[3]http://jena.apache.org/

provided by most Cloud platforms and totally ignore the inner correlations existed in the RDF data. For example, RDF tuples sharing the same *Subject* are semantically correlated; *Subjects* sharing the same set of *Predicates* (properties) probably belong to the same category. Second, MapReduce jobs bring inevitable cost on the disk I/O and shuffling of intermediate results, which is caused by not knowing the minimum number of data blocks that contain the valid answer beforehand.

Therefore, to improve the efficiency in answering SPARQL queries on the Cloud platform, we need to consider the following two factors: First, it is necessary to have RDF data re-modeled and better organized on the Cloud platform. Specifically speaking, right above the simple triplets' format, we need to store some structure information to have both the semantic and structure information being taken into account. Second, care should be taken on starting a number of MapReduce jobs. As disk I/O and network shuffling have been identified as the performance bottleneck, it is desirable to reduce the unnecessary I/O of data blocks as much as possible. In other words, MapReduce jobs should be fed with the minimum set of data blocks that contains the valid query answers. Ideally speaking, no MapReduce job should be initiated if there is a way we can tell that the query result is empty.

In this paper, we propose a solution which takes the above two factors into consideration to achieve the goal of minimizing the I/O cost of block scans and network shuffling, especially for queries with range and order constraints. Specifically, we first propose ***EAGRE***, an ***E***ntity ***A***ware ***G***raph comp***RE***ssion technique, to model RDF data on (*key,value*) storage in a particular manner to preserve both the semantic and structure information of RDF data. Based on the ***EAGRE*** model, we adopt the graph partition technique to distribute RDF data to computing nodes and build in-memory index to efficiently support range and order sensitive queries. Meanwhile, we identify a distributed I/O scheduling problem to minimize the disk scan and the total time cost for query evaluations. To summarize, our contribution are summarized as follows:

- We introduce a novel RDF representation model in the (*key,value*) store to preserve the inner correlation of RDF data.
- We propose a layout solution for RDF data on the Cloud platform to efficiently support SPARQL queries with range and order constraints specified.
- We propose an I/O efficient evaluation strategy for efficient SPARQL query processing in a distributed fashion. Compared to solutions that have only MapReduce jobs employed, our strategy significantly reduces the query processing time.

The rest of paper is structured as follows. For clear illustration purpose, in Section 2 we briefly introduce RDF and SPARQL query, as well as the main trend of evaluating SPARQL queries using MapReduce. We give the problem definition and a solution overview in Section 3 and formally present the ***EAGRE*** model and RDF layout method in Section 4. We elaborate our distributed I/O scheduling algorithm in Section 5. Experiments on both real and synthetic datasets are presented in Section 6. We discuss the related work in Section 7 and conclude in Section 8.

II. Preliminary

RDF, known as *Resource Description Framework*, is originally defined to describe conceptual procedures and meta data models. SPARQL is the W3C standard interface to query RDF data in a SQL-like style. In this section, we first describe the RDF data model and the current state of the SPARQL query standard. As our work targets at efficient SPARQL query evaluation over the Cloud platform, we shall briefly introduce the essential techniques that have been explored in existing literature for MapReduce-based SPARQL query processing.

A. RDF data

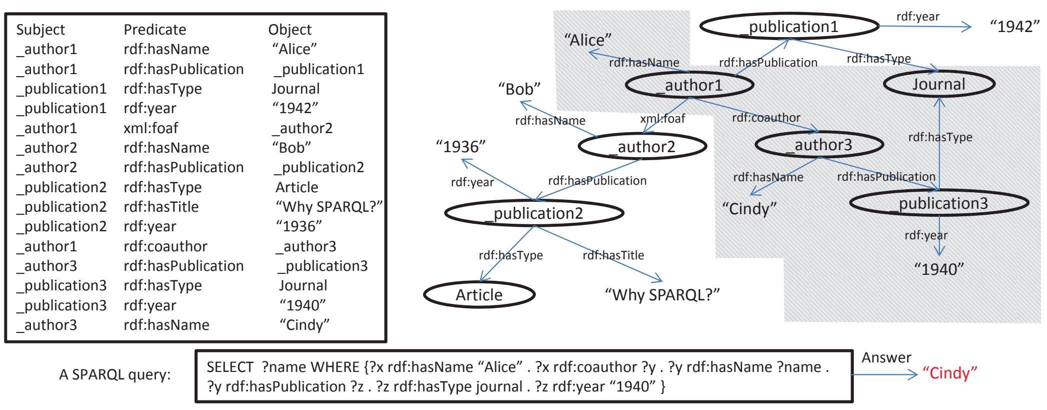
RDF model can be viewed as a description of the schema-relax relational model in the finest granularity. Each piece of RDF data is defined as a *Subject-Predicate-Object* triplet, describing the value (*Object*) of a *Subject*' particular property (*Predicate*). Fig.1 shows an example of RDF data describing authors and their publications. By making *Subjects* and *Objects* the nodes, and *Predicates* the directed edges pointing from the corresponding *Subject* to *Object*, RDF data can be viewed as a directed labeled graph.[4]

B. SPARQL Queries

SPARQL is the W3C standard interface for RDF query. It is designed to query RDF data in a SQL-like style. A SPARQL query specifies several *Basic Query Patterns*, *a.k.a.* *BQP*s, and the query returns are in fact the desired labels of subgraph(s) that exactly matches with given *BQP*s. Like the example given in Fig.1, to query the name of an author who has coauthored with "Alice" and has a journal published in 1940, a small subgraph (shown in gray) is identified to be the match of given *BQP*s. So far, SPARQL(1.1) allows four types of queries to be performed:

- `SELECT`: returns the desired variable value, as the query example shown in Fig.1.
- `CONSTRUCT`: returns a subgraph of RDF graph G that satisfies all the given *BQP*s. For example, in Fig.1, if we substitute the "`SELECT`" keyword with "`CONSTRUCT`", the returned result would be the subgraph covered in the gray shadow.
- `ASK`: instead of returning the variable value, it is a boolean function to indicate that a given variable has a value or not.
- `DESCRIBE`: returns all the associated labels and literal values. Intuitively, it represent some query like "`SELECT` ?p1 ?o ?s ?p2 `WHERE` {?x ?p1 ?o. ?s ?p2 ?x}".

[4]More rigorously, it is a directed multi-edge labeled graph, as some *Subject* may have multiple values of the same *Predicate*.[13]



Subject	Predicate	Object
_author1	rdf:hasName	“Alice”
_author1	rdf:hasPublication	_publication1
_publication1	rdf:hasType	Journal
_publication1	rdf:year	“1942”
_author1	xml:foaf	_author2
_author2	rdf:hasName	“Bob”
_author2	rdf:hasPublication	_publication2
_publication2	rdf:hasType	Article
_publication2	rdf:hasTitle	“Why SPARQL?”
_publication2	rdf:year	“1936”
_author1	rdf:coauthor	_author3
_author3	rdf:hasPublication	_publication3
_publication3	rdf:hasType	Journal
_publication3	rdf:year	“1940”
_author3	rdf:hasName	“Cindy”

A SPARQL query:

```
SELECT  ?name WHERE {?x rdf:hasName “Alice” . ?x rdf:coauthor ?y . ?y rdf:hasName ?name .
?y rdf:hasPublication ?z . ?z rdf:hasType journal . ?z rdf:year “1940” }
```

Answer “Cindy”

Fig. 1. An illustration example of RDF data and SPARQL query

Studies on the real world SPARQL queries [14][15][16] find that over 99% queries are `SELECT` queries. Therefore, in this work we only focus on this type of query. So far, SPARQL has evolved to support more advanced functions for flexible queries and the result representation, as well as some simple aggregation functions.

C. *SPARQL Evaluation Using MapReduce*

There have been extensive efforts to evaluate SPARQL queries over large volumes of RDF data using the MapReduce paradigm. For the purpose of comparison, we first summarize the essential ideas of current solutions.

As explained in Fig.1, conceptually the SPARQL query evaluation can be considered as a subgraph matching problem. However, to leverage the massive parallelism and scalability of the MapReduce framework, SPARQL queries are usually evaluated in a multi-way join fashion. To be specific, considering the variables as the join-keys, Map tasks first scan over the dataset to find all the data that satisfy given *BQP*s, then shuffle the data of the same join-key to the same Reduce task to examine all the possible valid join results. The state-of-art optimization techniques fall into three categories. First, reducing the volume of file scan. Solutions like “*Predicate* split” [9][8] and pre-computed query forwarding [17] try to evaluate queries only on the computing nodes that hold the desired data. Shared scan [18] is also widely adopted as an effective tool to reduce the file scan cost. Second, reducing the I/O cost of intermediate results with *bloom filter* [8] and effective selectivity estimation. By adopting the selectivity estimation, multiple MapReduce jobs can be organized and scheduled to achieve the minimum time cost of the query evaluation. Third, introducing filters or new hash functions to optimize the performance of MapReduce jobs conducting the join operation. For example, work [19] studies the optimal network shuffling function in case of performing multi-way join with one MapReduce job.

Compared to the existing work, our solution also targets at reducing unnecessary disk block scans. The novel idea is to postpone the Map tasks as late as possible until we find the minimum number of data blocks that contain the query results.

III. Problem Definition and Solution Overview

In this work, we target at evaluating SPARQL queries on the Cloud platform as quickly as possible. According to the claim made in [15], although most SPARQL queries in the real practice may have only a few number of *BQP*s specified, the `OPTIONAL` clause and the solution sequence modifier functions (`DISTINCT`, `ORDER BY`, `PROJECTION`, `OFFSET`, `LIMIT`, `REDUCE`) are frequently adopted. Surprisingly, there are very limited efforts towards the optimization of such kinds of SPARQL queries on the Cloud platform. Therefore, we try to derive a solution that particularly favours the query with range and order constraints specified.

As illustrated in the last section, SPARQL queries are generally processed as multi-way joins using MapReduce jobs. Since the performance bottleneck of a MapReduce job heavily lies in the cost of I/O and network traffic, we take the I/O efficiency as our primary optimization goal. In this section, we shall first define the I/O efficiency problem for SPARQL query evaluation, then describe a general picture of our solution framework.

A. *Problem Statement*

Intuitively, given a SPARQL query Q, the most I/O efficient evaluation is performed in the following manner: 1) Only the data blocks containing the desired query results are involved; 2) No network traffic is introduced. However, the second condition holds only when the computing node contains all the data. As a matter of fact, there could be more efficiency gains if the I/O cost can be amortized to multiple nodes. Like every coin has two sides, query processing over multiple nodes bring inevitable network traffic. Therefore, a desired solution is to trade-off the benefit of distributed I/O and the cost of network traffic.

Consider the following scenario, query Q is forwarded to k computing nodes for evaluation. Each computing node i needs to scan N_i blocks. If we start the Map tasks simultaneously on all the k nodes, the earliest time point to start the final Reduce task is dominated by $\texttt{MAX}(N_i)$ (assume there is no remote read). Plus, the volume of network traffic is at least $\sum_0^{k-1} Map\,(N_i)$, where $Map\,(N_i)$ is the output volume of the Map tasks on node i. Since there is no heavy computation during the Map phase, the total CPU and I/O cost for N_i data blocks can be considered as $O(N_i)$. However, the total complexity for network traffic is a function of both k and $\{Map(N_0),...,Map(N_{k-1})\}$. Apparently, $N_i \geq Map(N_i)$, therefore, we need to minimize $\texttt{MAX}(N_i)$ in the first place.

One way to reduce N_i on computing node i is to first consult other nodes before initiating the scan. The trick is that by querying the value range of particular variable on different computing nodes, it is possible to reduce the search space. We define such an operation as *Consulting*.

Definition 1 ***Consulting*** *is an all-to-all communication to inform other computation participants the variable's value range on each node.*

For example, a variable ?x has a value range (1,100) on node i while ?x has a value range (50,200) on node j, then after consulting, both node i and j know that it is sufficient to scan the data blocks having ?x fall in the range of (50,100). Since the consulting process is an all-to-all conversation among k nodes and only range information is passed, therefore, it can be considered as a constant cost factor. Then the problem becomes a *Consulting*-based scheduling of distributed I/Os.

Problem Definition: Given a SPARQL query $\mathcal{Q}$, which is forwarded to k computing nodes where each node has N_i data blocks to scan at the initial stage. Partition the k I/O operations into m disjoint stages. The I/O operations in the same stage are performed simultaneously, while the entire evaluation is performed stage by stage. Between each stage a *Consulting* is performed to reduce the I/O volumes in the next stage. Let the cost of stage s_i be denoted as $Cost(s_i)$=$\texttt{MAX}_{N_i \in s_i}(N_i)$. Find a scheduling function $\mathcal{F}$: $\{N_0, ..., N_{k-1}\} \rightarrow \underbrace{\{\{N_0...\}, ..., \{N_i...\}, ...\{N_j...\}\}}_{m}$, such that $\sum_{i=0}^{m-1} Cost(s_i)$ is minimized.

To elaborate, consider the example shown in Fig.2, query $\mathcal{Q}$ is forwarded to 5 nodes for evaluation. One naive scheduling function F_1 is to conduct the 5 I/O operations in 5 consecutive stages. At each stage, only the I/O operation of the minimal volume is performed. For example, originally N_2 is the minimum one, therefore s_0 contains N_2. After s_0 is done, a consulting process is initiated to reduce the I/O volume on other computing nodes. Like shown in the example, after s_0 N_1 is reduced from 4 to 3 and becomes the minimum value, therefore s_1 contains N_1. Another scheduling function F_2, however, is more greedier to take the advantage of parallelism.

According to the problem definition, intuitively a smart scheduling function should minimize the number of stages as well as the cost of each stage. In practice, the challenge to address the problem lies in two folds. First, it is non-trivial to estimate the reduced I/O volume on each node after a certain stage. Second, to leverage the variable's value range intersection for I/O reduction needs an order preserving layout of RDF data on the Cloud platform. The solution lies in a novel organization and layout of RDF data on the Cloud storage and an adaptive I/O scheduling algorithm, which shall be elaborated in Section 4 and Section 5, respectively. Instead of directly going into technical details, we shall first briefly present the solution framework and highlight the particular techniques we adopted.

	$N_0 = 6$	$N_1 = 4$	$N_2 = 3$	$N_3 = 20$	$N_4 = 7$
$s_0 = \{N_2\}$	$N_0 = 5$	$N_1 = 3$	–	$N_3 = 6$	$N_4 = 7$
$s_1 = \{N_1\}$	$N_0 = 4$	–	–	$N_3 = 5$	$N_4 = 3$
$s_2 = \{N_4\}$	$N_0 = 4$	–	–	$N_3 = 5$	–
$s_3 = \{N_0\}$	–	–	–	$N_3 = 5$	–
$s_4 = \{N_3\}$	–	–	–	–	–
F_1 $Cost = 3+3+3+4+5 = 18$					
$s_0 = \{N_1, N_2\}$	$N_0 = 4$	–	–	$N_3 = 5$	$N_4 = 3$
$s_0 = \{N_0, N_3, N_4\}$	–	–	–	–	–
F_2 $Cost = 4+5 = 9$					

Fig. 2. An illustration example of consulting-based scheduling of distributed I/Os

B. Solution Overview

We describe the general picture of our solution in this section. Briefly speaking, there are two steps. The first step is to organize RDF data on the Cloud (*key,value*) storage. As shown in the Fig.3, we extract "entities" and "entity classes" (both will be defined in Section 4) from the original RDF graph G and build the "compressed RDF entity graph". Using graph partition tools, we further partition the entity classes to distributed computing nodes, such that the structure locality of the original RDF graph can be well preserved. On each computing node, we treat the "entities" of the same entity class as high dimensional data. For example, the entity "publication" has properties like *title*, *year*, *type* and etc. By adopting the *Space Filling Curve* technique, we can maintain an order preserving layout of the RDF data that particularly fits for the queries with range and order constraints.

The second step, as shown in the upper part of Fig.3, is to introduce the query coordinator on each computing node to vote and decide the scheduling function of distributed I/Os. To be specific, a SPARQL query is evaluated as the following. Given a query $\mathcal{Q}$, we first identify the entity classes that contains the valid results. This can be done easily by introducing an in-memory index of the "compressed RDF entity graph" on a query engine. Afterwards, the query is forwarded to the computing nodes that hold the RDF data. Then, the *Consulting* protocol is initiated immediately among these query-involved computing nodes to decide further scheduling of I/O operations on each node. Note that the scheduling function is dynamically evolved along with the computation. Whenever some nodes finished the local I/O operation, they utilize the *Consulting* protocol to inform other computing nodes the collected statistics of scanned data. If the query has valid output, after all the query-involved computing nodes

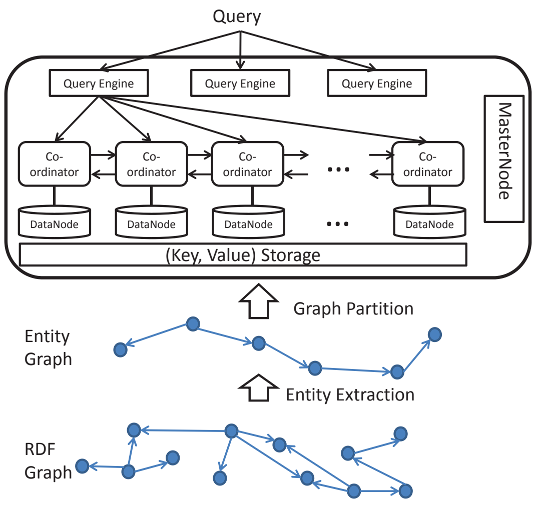


Fig. 3. An overview of the solution framework

finish the I/O operations, the finial result can be obtained with one Reduce task. We believe the novel representation and layout of RDF data on the Cloud platform, as well as the distributed I/O scheduling, are the core technique of our solution framework. Therefore, we mainly focus on the elaboration of these two points in this paper.

IV. Data Model

As elaborated in the problem statement, we intend to evaluate SPARQL queries efficiently by reducing the unnecessary I/O traffics as much as possible and postponing the MapReduce jobs until we are perfectly sure about the minimum set of data blocks that contain the desired results. However, to support such an evaluation, range-aware index must be built on each data node. In this section, we present a novel RDF representation on the (*key,value*) storage, namely the *E*ntity *A*ware *G*raph comp*RE*ssion model, shorted as *EAGRE*, and the layout strategy of RDF data to effectively support the range-aware index.

A. The EAGRE Model

One essential difference between RDF data and other data records with multiple attributes is that RDF implies semantic description of entities in the real world. Meanwhile, SPARQL is designed to facilitate the query over correlated entities. Therefore, instead of employing attribute-based partition and indexing techniques from traditional database, like row-store and column-store, we explore an entity-base model of RDF data to facilitate the SPARQL query evaluation.

There have been some interesting efforts to extract entities from RDF depositary[20][21][22]. However, the comparisons of RDF entity definition and extraction functions are beyond the scope of this work. We try to adopt a simple entity concept that serves as the atomic elements for query analysis and processing. Intuitively, given a *Subject* s, the set of all triplets having s as the *Subject* can be considered as a description of s, which contains the entire knowledge about s. Thus, we consider an entity to be a *Subject* and its complete description. Formally, we define an RDF entity as follows:

Definition 2 (RDF Entity) *A RDF entity, denoted as* $\mathrm{En} = (v_{\mathrm{R}}, Des(v_{\mathrm{R}}))$*, is a 2-level tree structured subgraph of RDF graph* G*, where* v_{R} *is the root, and* $Des(v_{\mathrm{R}})$ *is the set of all out-going edges from* v_{R} *and* v_{R}*'s one-hop neighbors in* G*, as well as the binding labels.*

Consider the example given in Fig.1, entities “_publication2”, “_publication3” and “_author1” are shown in Fig.4. Note that we adopt a prefix “E:” to denote an entity. Clearly, entities can be recursively defined. In other words, an entity may be described by many other entities, e.g. “E:_author1” is described by entities “E:_publication1”, “E:_author3” and “E:_author2”.

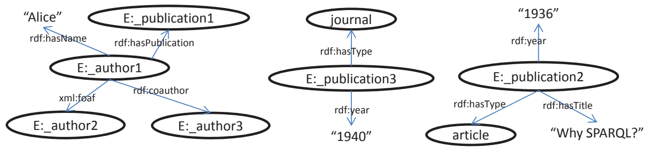


Fig. 4. Entities examples derived from Fig.1, rooted as “author1”, “publication2” and “publication3” respectively

We name an entity by the label of its root v_{R}, and model the edge labels and corresponding one-hop neighbors as (*key, value*) pairs as the description of v_{R}. For example, entity “E:_publication3” is described with {(rdf:hasType, journal), (rdf:year, “1940”)}. In general, we describe an entity in the following forms:

$$(\mathrm{E}:Subject, (key, value)\ \ list)$$

Let “E:*Subject*” serve as a key, an entity perfectly fits into well-known (*key,value*) stores that support structured *value* types, e.g. Cassandra [23]. For clear illustration purpose, we use the term “description keys” to denote the set of keys from an entity's (*key,value*) list for the rest of this paper. Since the RDF entity can be recursively defined, it implies the correlation between entities, which can be described with a RDF entity graph as follows:

Definition 3 (RDF Entity Graph) *RDF entity graph* $G_{\mathrm{En}}=\{V_{\mathrm{En}}, E_{\mathrm{En}}\}$ *is a directed graph, where* $V_{\mathrm{En}}=\{v_{\mathrm{En}}|v_{\mathrm{En}}$ *is a RDF entity*$\}$, $E_{\mathrm{En}}=\{\langle v_{\mathrm{En}}, v'_{\mathrm{En}}\rangle | v'_{\mathrm{En}} \in Des(v_{\mathrm{En}}),\ where\ v_{\mathrm{En}}, v'_{\mathrm{En}} \in V_{\mathrm{En}}\}$.

As a matter of fact, an entity graph G_{En} has almost the same topology structure comparing to the original RDF graph G, except the vertices in G that have no out-going edges are removed in G_{En}. For example, the left diagram of Fig.5 shows the RDF entity graph which is converted from the RDF graph given in Fig.1. Therefore, an entity graph can also be huge and impractical to query. However, an intuitive observation is that many entities share great similarity in terms of the keys of their description. To be specific, given two entities En and En$'$, we measure the similarity between them with the following method,

$$\tau = \frac{|Des(\mathrm{En}) \cap Des(\mathrm{En}')|}{Max\,\{|Des(\mathrm{En})|, |Des(\mathrm{En}')|\}} \qquad (1)$$

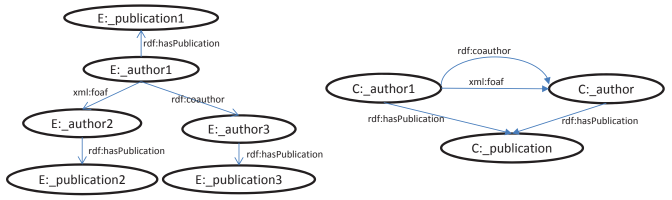


Fig. 5. The RDF entity graph and compressed RDF entity graph of the RDF graph given in Fig.1

As long as τ is larger than a given threshold, we consider two entities sharing the same set of description keys. The threshold value highly depends on how fuzzy the dataset is and only contributes to the efficiency of G_{En} compression. In our work, we setup the condition of $\tau \geq 0.9$. Thus, we can define the RDF entity class concept as follows:

Definition 4 (RDF Entity Class) *Two RDF entities* En_i *and* En_j *belong to the same entity Class* ***iff*** *they share the same set of description keys and are not on-hop neighbored vertices in* G_{En}.

By grouping the entities of the same class in G_{En}, a "compressed" RDF entity graph can be derived, which is much simpler and easier to query. We formally define the compressed RDF entity graph as follows:

Definition 5 (Compressed RDF Entity Graph) *A compressed RDF entity graph* G_{C}=$\{V_{\mathrm{C}}, E_{\mathrm{C}}\}$ *is a directed graph, where* V_{C}=$\{v_{\mathrm{C}}|v_{\mathrm{C}}$ *is a RDF entity class*$\}$, E_{C}=$\{\langle v_{\mathrm{C}}, v'_{\mathrm{C}}\rangle | \exists \mathrm{En} \in v_{\mathrm{C}}, \exists \mathrm{En}' \in v'_{\mathrm{C}}$, *and* $En' \in Des(En)$, *where* $v_{\mathrm{C}}, v'_{\mathrm{C}} \in V_{\mathrm{C}}\}$.

The right diagram in Fig.5 presents the compressed RDF entity graph of the RDF graph shown in Fig.1. Note that we adopt the prefix "C:" to denote an entity class. Clearly, a compressed RDF entity graph shields the detailed knowledge of entities, and has entity correlations be well preserved at the meanwhile.

B. RDF Data Layout

The layout of RDF data implies two decisions to make. First, which RDF data should be deployed to which computing node. Second, on each computing node, how RDF data should be organized. The data layout is a critical issue for SPARQL query evaluation, because it determines how many computing nodes are involved in a query evaluation, as well as the cost of local evaluation on each computing node. In this section, we elaborate our RDF data layout strategy to efficiently support the range and order sensitive query evaluation on local computing nodes, which can effectively reduce the total I/O cost to answer a SPARQL query.

The RDF data preprocessing involves three steps. First, we need to extract RDF entities and discover RDF entity classes, which has been discussed in Section 4.A. Second, we compute a partition of the compressed RDF entity graph to decide the deployment of RDF data to distributed storage nodes. Third, we compute the layout of RDF data on each computing node and set up the auxiliary indexing structure. Considering the huge volume of RDF data to process, the preprocessing requires a scalable computing paradigm, e.g., MapReduce. Therefore, the RDF data is first randomly partitioned to computing nodes. A re-allocation of RDF data is performed later on.

The extraction of RDF entities and the discovery of RDF entity classes can easily be implemented within the MapReduce computing framework. Pseudo codes describing the processes are described in Alg.1 and Alg.2, respectively. We employed two MapReduce jobs to do the work. First, we extract entities by grouping RDF triples by the *Subject*. Then, by making the describing key of each entity the hashing key, entities are grouped according to the similarity of their describing keys. In the implementation, we allow a little difference of entities' describing keys to form entity classes by setting up $\tau \geq 0.9$. The reason is that the real RDF data easily contain noises, missing properties and values. As the example shown in Fig.1, not all publications have the title value. We adopt a similar method introduced in [24] to enable the "set similarity join" using MapReduce.

Algorithm 1: RDF entity extraction

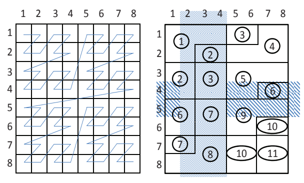
```
Data: A set of RDF $\langle s, p, o\rangle$ triples
Result: A set of RDF entities
for each Map task do
    key← $s$ and value← $\langle p, o\rangle$
    Output(key,value)
for each Reduce task do
    for each key do
        Assemble the description of given key
        value← $An\ entity$ $\mathrm{En}$
    Output(key,value)
```

Algorithm 2: Discovering RDF entity classes

```
Data: A set of RDF entities $(Label(\mathrm{En}), Des(\mathrm{En}))$
Result: A set of RDF classes
for each Map task do
    key← description key of $\mathrm{En}$ and value← $\mathrm{En}$
    Output(key,value)
for each Reduce task do
    for each key do
        Check the entity class condition given in Definition 4
    Output valid RDF entity classes
```

The second step is to re-allocate RDF entities to distributed storage nodes. After entity classes are discovered on each storage node, the entity class correlations can be computed locally on each node. Then, these partial graphs can be reduced to a central node to assemble the complete compressed RDF entity graph. Note that the reducing phase does not involve entity copying over network, only the graph structures representing entity class correlations are delivered. The data deployment is guided by the partition of the compressed RDF entity graph. There have been abundant research efforts towards graph partition, many of which are workload-driven methods. Since we do not assume any pre-knowledge of the query workload, we intend to partition the graph in some way to best preserve the local topological structures. We adopt a well-recognized graph partition tool **METIS**[25] to do the job. Eventually, we can easily re-allocate the RDF entities according to the partition results. Clearly, the first two steps of preprocessing is mainly dominated by the I/O and network copying of the entire dataset. In the worst case, a RDF triple can be read and written 3 times. Therefore, the computation complexity is $O(N)$, where N is the total number of data blocks that hold the entire RDF dataset.

In the third step, we consider the layout of RDF data on each storage node. The query evaluation scenario is that, given a query, we want to immediately decide the location (i.e., a data block) of the RDF triples that match the query's *BQP*s. Moreover, we want to know a sequence of RDF triples, probably sorted under certain *Predicate*, such that queries with result order constraint can be effectively supported. In our solution framework, we consider the RDF data as a set of RDF entities which are grouped into correlated classes. Therefore, we organize the RDF data on the basis of entity classes. Within each entity class, entities sharing the same description keys are in fact high dimensional data records. We adopt the *space filling curve*, which has been widely used for indexing high dimensional data, to compute the layout of RDF entities of the same entity class.


(a) Z-order curve based RDF entity layout

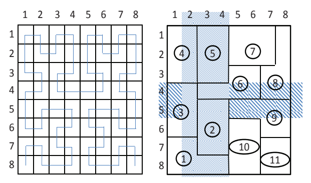

(b) Hilbert space filling curve based RDF entity layout

Fig. 6. Using space filling curves to compute the layout of RDF entities

To elaborate, consider the example shown in Fig.6. Assume the description of an entity are two dimensional data, and a disk block contains at most 6 entities. Based on the value distribution of each dimension, we can compute the layout of all 64 RDF entities using the well-known *space filling curves*. The shadowed region indicates that, given a query range [2,4]×[4,5], the data blocks containing the desired entities can be effectively located.

Our solution benefits from the *space filling curve* determined RDF entity layout in three folds. First, it provides an effective solution to save the I/O cost of unnecessary disk scan. Given the query range on multiple dimensions, we can immediately compute the inter section of data blocks satisfying each given dimension's constraints. When the dimensionality increases, the selectivity on data blocks also increases. Besides, by associating the value range information of each data block to its meta data, given a data block, we can immediately decide the value range of entities on any dimension. For example, in case of Fig.6(a), given data block 1, we know its value range is [1,4]×[1,2]; in case of Fig.6(b), we know block 1's value range is [1,2]×[7,8]∪[3,4]×[8].

The second advantage of such a layout strategy is the order preserving property of *space filling curves*. Given a space filling function, the sequence of RDF entities are determined solely by their description values, which is a one-to-one mapping. Therefore, given the pre-knowledge of the space filling function, it is possible to sort all the RDF entities with regard to any description key in only one scan, which effectively reduces the complexity of sorting to linear time.

Thirdly, such a layout schema functions well in case of appending updates. With new entities arrived, its logical position in the global "space filling curve" can be effectively computed, which implies which data block it should be deployed to. However, since the target data block is already full, we only need to associate the target data block a new block to store the new entities. When there is a read operation, a data block as well as the data blocks associated to it are scanned. A critical problem is that the increasing number of random access of disk blocks can introduce unacceptable overheads. Therefore, in the implementation we can specify a time threshold for the average data block access time. If the threshold is violated, the re-construction of the entity layout is performed.

Note that we propose the *space filling curve* based RDF entity layout because it can effectively help decide the disk I/O volume on each computing node. Although we do not hold the proof that, without any pre-knowledge on query patterns, the *space filling curve* based layout is the optimal solution in terms of filtering out unnecessary disk block scans to the most extend. As we shall elaborate in the later section, experiments show that our solution demonstrates considerable I/O saving comparing to other off-the-shelf layout strategies. An interesting observation is that, given two space filling functions that are in fact equivalent, they may demonstrate different filtering power for a specific query. In Fig.6's example, given the query range [2,4]×[4,5], *Z-order* gives data block 2,3,6 and 7 while *Hilbert* gives only data block 2,3 and 6. Apparently, with the pre-knowledge of query pattern, a better layout strategy can be derived. We consider it an interesting future problem to investigate.

V. Distributed I/O Scheduling

As elaborated in the last section, the efforts we made on the representation and the layout of RDF data on each computing node are to serve the I/O efficient distributed evaluation of SPARQL queries. Given a SPARQL query being forwarded to k computing nodes, on each node we can almost immediately decide the data blocks to retrieve and the corresponding value ranges of any variable given in the query. The real problem, as we state in Section 3, is to effectively schedule the distributed I/O to postpone MapReduce jobs such that the total time cost for query evaluation is minimized. In this section, we present an estimation-based self-tuning scheduling strategy to reduce the unnecessary I/O as much as possible.

A. *Problem Property*

Recall that our problem setting is that a query is forwarded to k distributed computing nodes, initially a node n_i needs to scan N_i data blocks to answer the query. As we introduce the *Consulting* mechanism, it is possible to do the scan on only a few nodes in the first place to obtain the fine drilled variable value distribution, such that the I/O volume on remaining nodes can be greatly saved through the *Consulting*. The problem is to find a scheduling strategy to reduce the total time cost on the I/O volume. Let S=$\{s_0, s_1, ..., s_{m-1}\}$ be a scheduling strategy, where each stage $s_i \in S$ is a set of distributed I/O operations on some computing nodes. The consulting process is performed between any two successive stages. Thus, as stated in Section 3.A, we define the problem as an optimization problem, which is to find a S_{opt} such that $\sum_{k=0}^{m-1} Cost(s_k)$ is minimized. Note that $Cost(s_i)$ is defined as the maximum number of disk I/O in stage s_i.

Before we dive into the detailed solution, we first define the following notations for clear illustration purpose:

`Scan-Contribution` $SC(n_i, S)$ denotes the number of disk blocks that can be saved by scanning node n_i, after all the stages of scanning in S have been performed.

`Scan-Heritage` $SH(n_i, S)$ denotes the number of disk blocks on node n_i that can be saved after all the stages of scanning in S have been performed.

Intuitively, to compute the optimal scheduling S_{opt}, we concern about two factors: 1) how the initial stages can affect the later stages; 2) Is there an connection between the local optimization and the global optimization? As a matter of fact, by investigating these two factors, we can easily examine that the following two lemmas are true.

Lemma 1 *Given* $S \neq S'$, *if* $\bigcup_{s_i \in S} s_i$=$\bigcup_{s'_i \in S'} s'_i$, *then* $SC(n_i, S)$=$SC(n_i, S')$ *and* $SH(n_i, S)$=$SH(n_i, S')$.

Lemma 1 indicates that, given k distributed I/Os to schedule, if we have t ($t < k$) I/Os already performed, no matter how these t distributed I/Os are scheduled, the hints for the remaining $k - t$ I/Os are the same.

Lemma 2 *Assume* $S = \{s_0, s_1, ..., s_{m-1}\}$ *is the optimal scheduling of* $\mathcal{N} = \bigcup_{s_i \in S} s$ *distributed I/Os, then* $S' = \{s_0, s_1, ..., s_{m-2}\} \subset S$ *is the optimal scheduling of* $\mathcal{N}' = \bigcup_{s_i \in S'} s_i$ *distributed I/Os.*

Proof: Prove by contradiction. Assume S^* is the optimal scheduling for $\mathcal{N}' = \bigcup_{s_i \in S'} s_i$. By Lemma 1, $SH(n_i, S^*) = SH(n_i, S')$ and $SC(n_i, S^*) = SC(n_i, S')$, where $n_i \in s_{m-1}$. Thus, the cost of s_{m-1} is the same in these two scenarios. It implies that S is also not the optimal, which contradicts to the assumption. ■

Although we can recursively define the optimal solution function based on Lemma 2, there is no dynamic programming solution to our problem. Because there is no way to obtain the initial value of $SC(n_i, \{n_j\})$ and $SH(n_i, \{n_j\})$, where $n_i \neq n_j$, unless we perform the scan on each node in the first place. However, Lemma 2 implies an important property of our problem that the optimal solution requires the best effort at each scheduling stage.

B. Scheduling Strategy

As the `Scan-Contribution` and `Scan-Heritage` value of each node can only be determined at running time, we propose an estimation based self-adaptive scheduling strategy that can achieve the optimal scheduling target.

Initially, when a query is forwarded to k distributed computing nodes, each node can determine the data blocks to access and the value ranges of every data block. Assume the query has l variables, thus, each computing node holds an array of length l which contains all the value range information. With one time of consulting, each node holds a matrix of size $l \times l$. Then, by conducting the range intersection operation, each node can filter out the data blocks that cannot possibly contain valid answers. However, there is no straightforward information about how a data node n_i's `Scan-Contribution` and `Scan-Heritage` value in case that any other computing nodes has not performed the I/O operation. Therefore, we employ an estimation-based method to find out the first stage of I/O operations.

Assume to evaluate a query of p variables $\{v_1, v_2, ..., v_p\}$, a data node n_i initially has N_i data blocks to scan, let X=$\{(a_1, b_1), (a_2, b_2), ..., (a_p, b_p)\}$ denote the value ranges of all variables in the N_i data blocks. Thus, we can estimate the selectivity on a variable v_i using $\frac{N_i}{b_i - a_i}$, denoted as $sel(v_i)$. The intuition is that, if $sel(v_i)$ is small, it implies that more fine drilled value distribution can be derived. For example, 50 data blocks contain variables v_1 and v_2's value ranges as [2,5] and [2,50], respectively. It is more likely that the 50 data blocks hit v_1's value 2,3,4 and 5. On the contrary, the 50 data blocks may only hit partial values of v_2's value range. Noted that we also introduce N_i as a dominating factor in the computation, such that the I/O operation on a data node which has a large N_i value tends to be postponed.

A distributed I/O scheduling algorithm is described in Alg.3. Lines 1-3 is the initialization phase. Given a SPARQL query Q, each node identifies the number of data blocks to scan and the value range of variables. After the first time *Consulting*(line 5), every node is fully aware of the variable value distribution on other computing nodes, therefore, they can update their I/O cost and now every node has agreed on the variables' value ranges. Note that up to this step, it is feasible to determine whether the query has a valid answer, as the examination process taken in line 8-9. Thus, by computing every variable's selectivity of each computing node(line 10), it is possible to identify the most beneficial action to take such that the I/O volumes on all computing nodes are expected to be reduced the most. Finally, we pick the set of data nodes that have the smallest selectivity value on different variables, and schedule them as the first stage of I/O operations. The iteration stops until all the k I/O operations have been scheduled.

The above scheduling algorithm does not rely on data statistics and therefore does not require the statistics collection process during the data preparing stage. However, as long as statistics are available, it can be easily plugged in to serve the scheduling. Clearly, we perform the scheduling in a best effort fashion in every stage. To elaborate, in absence of a data node's `Scan-Contribution` and `Scan-Heritage` value, we always make the decision that

Algorithm 3: Estimation-based self-adaptive scheduling of distributed I/O operations

```
Data: A SPARQL query $Q$ of $p$ variables; $k$ computing nodes $\{n_1, ..., n_k\}$
Result: A scheduling of $k$ distributed I/O operations $S$
1  for each node $n_i$ do
2      Identify $N_i$ data blocks to access
3      Generate the corresponding $p$ variable value ranges
       $X_i$={$(a_1, b_1), (a_2, b_2), ..., (a_p, b_p)$}
4  while $\exists n_i$ has not been scheduled do
5      Consulting
6      for each node $n_i$ do
7          Update $N_i$ and $X_i$
8          if $\exists v_i$'s value range is $\emptyset$ then
9              Return empty answer set and abort the evaluation process
10         Compute the variable selectivities $\{sel(v_1), sel(v_2),...,sel_(v_p)\}$
11     for each variable $v_i$ do
12         Select the data node that has the smallest selectivity on $v_i$
13     Schedule the set of selected data nodes to perform the I/O operation
```

promises the greatest reduction of I/O volumes. After each stage, we have to adaptively learn new hints from the finished I/O scans and make new decisions. According to Lemma 2, our solution is expected to achieve the optimal scheduling from a probabilistic point of view. An important concern is that if our I/O reduction oriented scheduling has negative side-effect on exploring the massive parallelism of the Cloud computing framework. As a matter of fact, our scheduling strategy only guides the variable evaluation sequence rather than the number of nodes involved for evaluation. The essential parallelism of an evaluation task only relies on the volume and the layout of data that need to be processed.

By solving the scheduling of distributed I/Os, we can effectively reduce the total number of data blocks to read for computation. Although we postpone the MapReduce job among k distributed nodes, the job is expected to accomplish more quickly since significant volume of unnecessary I/O and network traffics are eliminated.

VI. Experiments

We run all the experiments on a cluster of 28 computing nodes. Each node has 2 CPUs of 3.06GHz and 2GB memory, 200GB disk storage attached, running 2.6.35-22-server #35-Ubuntu SMP. We use Hadoop-1.0.0 to build the system. The setting of some major Hadoop parameters are given in Table I, which follows the setting suggested by [26]. Notice that we set the *fs.blocksize* to its default value, which may be too small for normal MapReduce jobs since it brings significant scheduling overhead. However, we intend to make it small to avoid unnecessary data loading for query evaluation. We use the *TestDFSIO* program to test the I/O performance of the system, and find that the system performance is stable, with average writing rate 2.98MB/sec and reading rate 18.17MB/sec. For the *Consulting* protocol, we employ MPICH2-1.4.1p1 and turn on the O3 compiler switch. We run each experiment job with 3 cold-start and report the average execution time.

In the experiments, we employ two real datasets, Billion Triple Challenge 2011 and Yago2, and three synthetic datasets generated using the SP^2 [27] benchmark data generator. Table II summarizes the brief statistics of all the datasets. In the table, we show the original number of triples of each dataset, as well as the number of entities and entity classes that can be extracted based on our definition. "M" denotes that numbers are counted in millions. Apparently, by introducing the compressed entity class graph, we can easily reduce the size of the original RDF graph by at least two orders of magnitude.

TABLE I
Hadoop parameter configuration

Parameter Name	Default	Set to
$fs.blocksize$	64MB	64MB
$io.sort.mb$	100M	512MB
$io.sort.record.percentage$	0.05	0.1
$io.sort.spill.percentage$	0.8	0.9
$io.sort.factor$	100	300
$dfs.replication$	3	3

TABLE II
Employed datasets for evaluation

Dataset	# Triples(M)	# Entity(M)	# Entity Class(M)
Yago2	295	10	0.06
BTC2011	2170	460	11.5
Syn.A	100	5	0.02
Syn.B	1000	50	0.35
Syn.C	5000	250	2

To demonstrate the effectiveness of our solution, we employ a set of benchmark queries over different datasets. In general, we focus on three aspects of query evaluations: 1) time efficiency; 2) how evaluation strategy affects the I/O cost and network traffic volume; 3) the scalability of our solution. In the experiments, we employ our earlier solution using the *Predicate*-based partition [8] and an open source project SHARD [28] as competitors.

The *Predicate*-based partition solution (shorted as *P*-Partition) works by grouping RDF triples by *Predicates*, and enumerate the sorting of two triplet patterns "*S-P-O*" and "*O-P-S*". The *P*-Partition solution from [8] also employs sophisticated scheduling of MapReduce jobs and optimization techniques like *bloom filter* to reduce the data copying volume over the network. The SHARD project treats RDF data as a graph, no special data layout function is employed. After the RDF data are uploaded into the HDFS, it builds simple index structure to represent the graph structure. Given a SPARQL query, it conducts graph matching locally on each computing node and finally merge the results.

Before giving the detailed discussion about the experimental results, we highlight the achievement of our solution. Generally, our data layout strategy and the *Consulting*-based evaluation plan can effectively reduce the disk I/O volume and the network traffic, and achieve over an order of magnitude of time saving in query evaluations.

A. Setup

We first describe the time cost to setup the *EAGRE*-based RDF depository for query evaluation. Fig.7 presents the time costs to setup the Yago2, BTC2011 and Syn.B datasets, respectively. As summarized in Fig.7(a), there are three major steps in the data preparation process. Apparently, the cost on uploading data to the HDFS and computing the compressed entity class graph is the dominating factor. On the other hand, data re-distribution is somehow less expensive (which depends on the partition rule employed in the data uploading stage), and the RDF entity layout computation on each computing node can be done very efficiently.

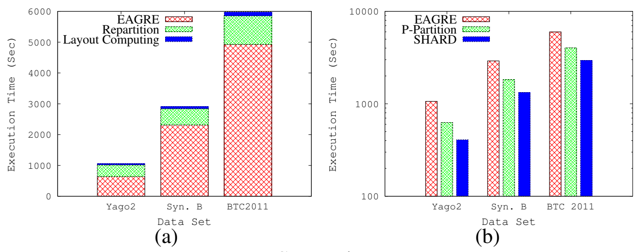


Fig. 7. Setup time cost

Fig.7(b) presents the time costs to setup the same datasets using different solutions. The *Predicate*-based partition (denoted as *P*-Partitions) only needs one MapReduce job to partition triples according to the *Predicate*, therefore, no repartition cost is introduced. SHARD performs simple local indexing on each computing node after the uploading process. Although the *EAGRE*-based solution takes longer time during the data preprocessing, however, the preprocessing is usually conducted off line and the cost can always be amortized.

B. Efficiency Test

To validate the effectiveness of our solution, we employ three sets of benchmark queries over different datasets. For each SPARQL query, we measure the evaluation make span, number of I/O Read and number of network volumes using different evaluation strategies. To compare with our *EAGRE*-based solution, we also employ the *Predicate*-based partition solution from work [8], and one Hadoop-based open source project SHARD [28] which utilizes a graph matching strategy for the SPARQL evaluation. We elaborate the results as follows:

1) BTC2011 Dataset: We employ the benchmark used in [3] for evaluation. We left Q8 out because it is no longer a valid query against the BTC2011 dataset (unmatchable *Predicates*). The results are presented in Fig.8. Apparently, no matter whether z-curve or *hilbert* curve is employed for the entity layout, which are denoted as *EAGRE*(Z) and *EAGRE*(H) respectively, our evaluation strategy achieves significant time saving. Consider two extreme examples Q4 and Q7. Q4 includes a *BQP* having the *Predicate* as a variable, which is a disaster for the *P*-Partition. Because it needs to scan large volume of data to obtain the results. Similarly, SHARD also has difficulty in answering such queries. Without the guidance of the *Predicate* value, SHARD generates as much candidates as possible to guarantee the correctness of the answer. Comparing to other queries, Q7 includes the biggest number of *BQP*s, therefore, it implies more information about the desired RDF entity structure. Thus, it benefits the evaluation in case that we organize the data according to the RDF entity correlations. On the contrary, complex *BQP*s impose hard decision on the generation and scheduling of MapReduce jobs, thus the performance of the *P*-Partition method is not satisfactory.

Fig.8(b) and 8(c) prove that the *EAGRE* solution effectively reduces the number of disk scans and the data copying volume over the network. Comparing to *P*-Partition, *EAGRE* transmit high fraction of data that it reads from the disk, which implies efficient reduction of disk scans.

2) Yago2 Dataset: We employ the benchmark given in work [2] for evaluation. The results are presented in Fig.9. As stated in [2], the 8 queries are thematically grouped into three groups. The first group of queries (Q1-Q3) concern oriented facts; the second group (Q4-Q6) asks about oriented relationships; and the third group consists of unknown *Predicate* for evaluation. As a matter of fact, the orientation of entities are well preserved in the *EAGRE* model. Thus, by inquiring the compressed entity class graph, the entity class containing the valid answer can be instantly derived. On the other hand, *P*-Partition keeps no information about the RDF graph structure. Although SHARD evaluates SPARQL queries using graph matching techniques, it performs a best-effort search on each computing node without information exchange until the final Reduce task. Therefore, as shown in Fig.9, the *EAGRE* solution achieves even more time saving comparing with that in BTC2011 dataset.

3) SP^2 Dataset: We employ dataset Syn.B and the benchmark given in work [27] for evaluation. The results are presented in Fig.10. As a matter of fact, the SP^2Benchmark queries are performed over a synthetically generated DBLP-like dataset. One thing special about the SP^2Benchmark query is that it tests how the query engine evaluates queries, which have only limited or fix number of results, over RDF datasets in different scales. For example, Q1 returns only one triple no matter how large the dataset is. As shown in Fig.10(a), comparing to *P*-Partition and SHARD, the time efficiency gains of *EAGRE* vary from query to query. However, *EAGRE* demonstrate consistent evaluation time cost for queries that have very limited returned answers, e.g. Q1,Q7,Q9 and Q11. One interesting observation is made over Fig.10(b) and 10(c). Although *EAGRE* and *P*-Partition have very different costs on disk scan, the network volume of these two solutions are not much different. The reason is that *P*-Partition employs *bloom filter* to reduce the shuffling of undesired data blocks.

C. Scalability Test

To prove the scalability of our solution, we run the SP^2 benchmark queries against three synthetic datasets in different scales, as shown in Fig.11(a). Apparently, for queries that only return very limited or fixed number of triples, our solution gives about even performances. Because we target at locating the minimum set of disk blocks that contain the valid answer in the very first place of evaluation, such that the query processing time is mainly dominated by the number of returned triplets. Moreover, the results presented in

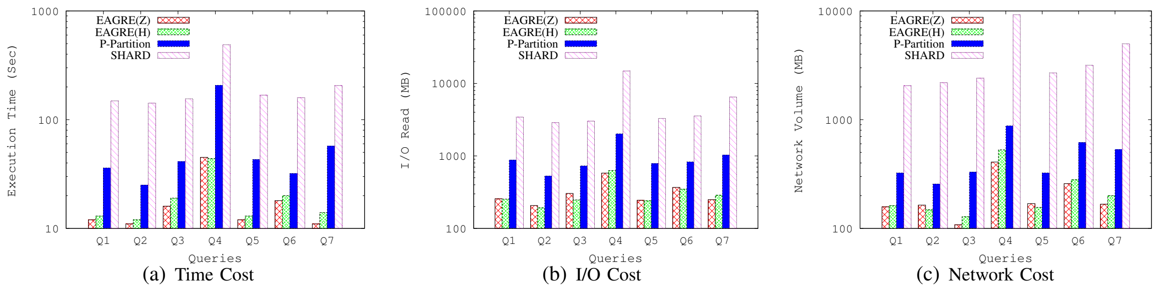


Fig. 8. Efficiency test over the BTC 2011 dataset

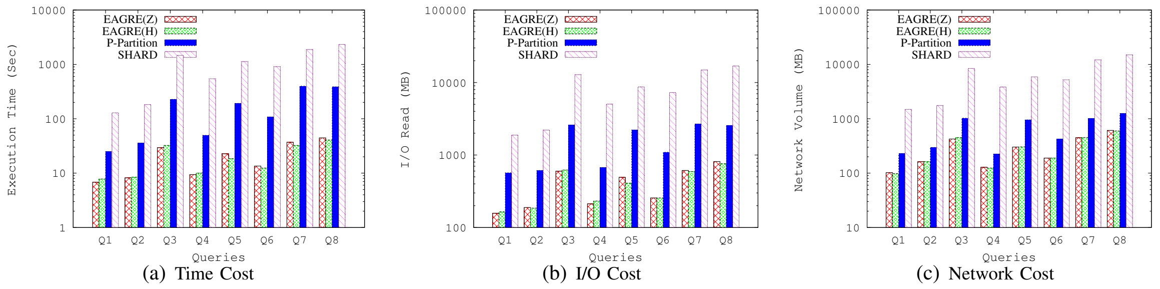


Fig. 9. Efficiency test over the Yago2 dataset

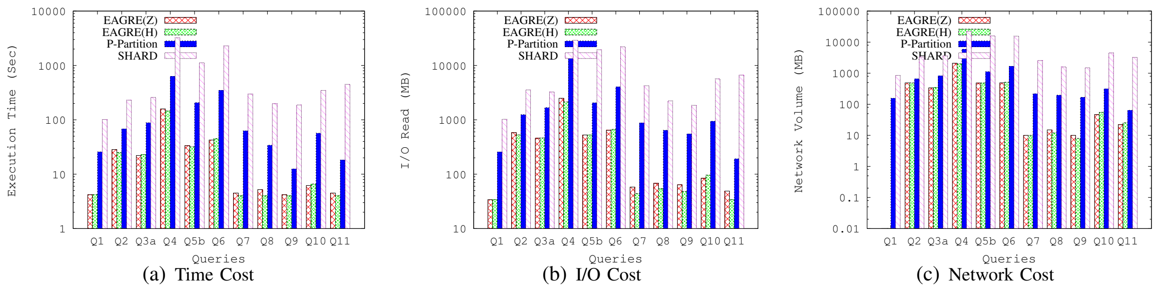


Fig. 10. Efficiency test over the synthetic SP^2 benchmark dataset B

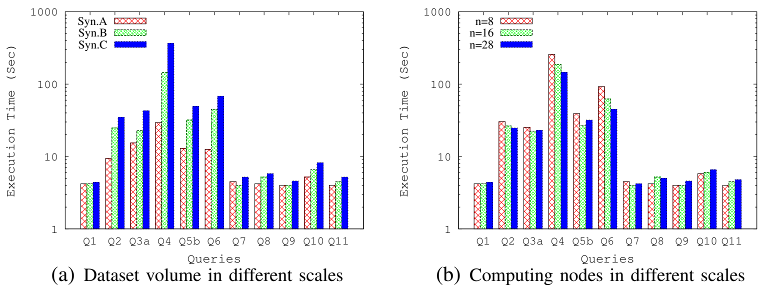


Fig. 11. Query efficiency over the synthetic SP^2 benchmark datasets in different scales

Fig.11(a) demonstrate convincing scalability of our solution. Along with the rapid growth of underlying data volume, our solution takes the advantage of the parallel processing and achieves almost linear speedup (y-axis is in log scale). Fig.11(b) presents another scalability test on how our solution adapts to different size of clusters. In the figure, we use n to denote the number of computing nodes employed to setup the system. The same observation is made on this test, that the query evaluation time of our solution is mainly dominated by the result size. Take Q4 for example, as the output volume is large, it is beneficial to include more computing nodes to amortize the I/O time cost.

VII. Related Work

There are mainly two categories of solutions for RDF management and query processing. One is to use traditional RDBMS technologies, either stand-alone sever or distributed (parallel) computing framework. RDF data are represented as tables in databases. Intensive research interests focus on the RDF decomposition (SW-Store [6]) or composition (property table from Jena), the index construction and searching (Hexastore [7], RDF-3X [2]), as well as the query optimization

[29]. However, due to the limitation of RDBMS's scalability, the existing solutions cannot meet the demands for managing extremely large scale RDF data in the coming future. Moreover, these solutions are closely related with system hard states, which are hard to maintain.

The other solution category is to incorporate NoSQL database to address the scalability and flexibility issues in the first place. Many works, like [30][31][32][33][34], adopt the Cloud platform to solve the RDF data management problem. However, many of them focus on utilizing high-level definitive languages to create simplified user interface for RDF query processing, which omit all the underlying optimization opportunities and have no guarantees on efficiency. On the contrary, Husain et al. [9] focuses on effective RDF data storage and querying. It adopts a greedy strategy to pick a join that may produce the smallest size of intermediate results. Unfortunately, it has no guarantee on the overall efficiency.

An important recent work presented in [35] attempts to combine the advantages of RDBMS and the scalable Cloud platform, which adopts RDF-3X on each computing node and manages the entire cluster using Hadoop. Essentially, it follows the system architecture of HadoopDB [36]. Although it demonstrates impressive performance in query evaluation, the heavy cost on maintaining the RDBMS on all computing nodes remains a bottleneck of scalability and fault tolerance. Comparing with our solution, we made different choices on the architecture design. And our primary goal is to advance the SPARQL query evaluation on a scalable NoSQL platform.

VIII. Conclusion

In this paper, we propose a novel *E*ntity-*Aware* *G*raph comp*RE*ssion model of RDF data on the Cloud platform, and a *Consulting*-based query evaluation strategy to prune unnecessary disk scans. We propose a distributed scheduling strategy to coordinate the I/O operations on distributed computing nodes. Extensive experiments show that, compared to other MapReduce-based state-of-art solutions, our method can achieve over an order of magnitude of time saving for the SPARQL query evaluation on the Cloud.

Ackownledgement

This work is supported in part by the Hong Kong RGC GRF Project No.611411, HP IRP Project 2011, National Grand Fundamental Research 973 Program of China under Grant 2012-CB316200, Microsoft Research Asia Grant, MRA11EG05 and HKUST RPC Grant RPC10EG13.

References

[1] C. Olston and et. al., "Pig latin: a not-so-foreign language for data processing," in *SIGMOD Conference*, 2008, pp. 1099–1110.

[2] T. Neumann and et. al., "The rdf-3x engine for scalable management of rdf data," *VLDB J.*, vol. 19, no. 1, pp. 91–113, 2010.

[3] T. Neumann and G. Weikum, "Scalable join processing on very large rdf graphs," in *SIGMOD Conference*, 2009, pp. 627–640.

[4] T. Neumann and et. al., "x-rdf-3x: Fast querying, high update rates, and consistency for rdf databases," *PVLDB*, vol. 3, no. 1, pp. 256–263, 2010.

[5] D. J. Abadi and et. al., "Scalable semantic web data management using vertical partitioning," in *VLDB*, 2007, pp. 411–422.

[6] D. J. Abadi, A. Marcus, S. Madden, and K. Hollenbach, "Sw-store: a vertically partitioned dbms for semantic web data management," *VLDB J.*, vol. 18, no. 2, pp. 385–406, 2009.

[7] C. Weiss and et. al., "Hexastore: sextuple indexing for semantic web data management," *PVLDB*, vol. 1, no. 1, pp. 1008–1019, 2008.

[8] X. Zhang and et. al., "Towards efficient join processing over large rdf graph using mapreduce," in *SSDBM*, 2012, pp. 250–259.

[9] M. F. Husain and et. al., "Scalable complex query processing over large semantic web data using cloud," in *IEEE CLOUD*, 2011, pp. 187–194.

[10] H. Kim and et. al., "From sparql to mapreduce: The journey using a nested triplegroup algebra," *PVLDB*, vol. 4, no. 12, pp. 1426–1429, 2011.

[11] P. Ravindra and et. al., "Efficient processing of rdf graph pattern matching on mapreduce platforms," in *DataCloud-SC*, 2011, pp. 13–20.

[12] X. Zhang and et. al., "Efficient multi-way theta-join processing using mapreduce," *PVLDB*, vol. 5, no. 11, pp. 1184–1195, 2012.

[13] L. Zou and et. al., "gstore: Answering sparql queries via subgraph matching," *PVLDB*, vol. 4, no. 8, pp. 482–493, 2011.

[14] M. Arias and et. al., "An empirical study of real-world sparql queries," *CoRR*, vol. abs/1103.5043, 2011.

[15] F. Picalausa and et. al., "What are real sparql queries like?" in *SWIM*, 2011, pp. 7:1–7:6.

[16] S. Duan and et. al., "Apples and oranges: a comparison of rdf benchmarks and real rdf datasets," in *SIGMOD Conference*, 2011, pp. 145–156.

[17] F. Prasser and et. al., "Efficient distributed query processing for autonomous rdf databases," in *EDBT*, 2012, pp. 372–383.

[18] T. Nykiel and et. al., "Mrshare: Sharing across multiple queries in mapreduce," *PVLDB*, vol. 3, no. 1, pp. 494–505, 2010.

[19] F. N. Afrati and et. al., "Optimizing multiway joins in a map-reduce environment," *IEEE Trans. Knowl. Data Eng.*, vol. 23, no. 9, pp. 1282–1298, 2011.

[20] M. Svoboda and et. al., "Linked data indexing methods: A survey," in *OTM Workshops*, 2011, pp. 474–483.

[21] F. Goasdoué and et. al., "View selection in semantic web databases," *PVLDB*, vol. 5, no. 2, pp. 97–108, 2011.

[22] F. Du and et. al., "Partitioned indexes for entity search over rdf knowledge bases," in *DASFAA*, 2012, pp. 141–155.

[23] A. Lakshman and et. al., "Cassandra: a decentralized structured storage system," *Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[24] R. Vernica and et. al., "Efficient parallel set-similarity joins using mapreduce," in *SIGMOD Conference*, 2010, pp. 495–506.

[25] G. Karypis and et. al., "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, 1998.

[26] D. Jiang and et. al., "The performance of mapreduce: An in-depth study," *PVLDB*, vol. 3, no. 1, pp. 472–483, 2010.

[27] M. Schmidt and et. al., "Sp ^{2}bench: A sparql performance benchmark," in *ICDE*, 2009, pp. 222–233.

[28] K. Rohloff and et. al., "High-performance, massively scalable distributed systems using the mapreduce software framework: the shard triple-store," in *PSI EtA*, 2010, p. 4.

[29] T. Neumann and G. Moerkotte, "Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins," in *ICDE*, 2011, pp. 984–994.

[30] F. Bugiotti and et. al., "Rdf data management in the amazon cloud," in *Joint EDBT/ICDT Workshops*, 2012, pp. 61–72.

[31] G. Ladwig and et. al., "Cumulusrdf: Linked data management on nested key-value stores," in *SSWS Workshop*, 2011.

[32] A. Schätzle and et. al., "Pigsparql: Übersetzung von sparql nach pig latin," in *BTW*, 2011, pp. 65–84.

[33] G. Tsatsanifos and et. al., "On enhancing scalability for distributed rdf/s stores," in *EDBT*, 2011, pp. 141–152.

[34] N. Papailiou and et. al., "H2rdf: adaptive query processing on rdf data in the cloud," in *WWW (Companion Volume)*, 2012, pp. 397–400.

[35] J. Huang and et. al., "Scalable sparql querying of large rdf graphs," *PVLDB*, vol. 4, no. 11, pp. 1123–1134, 2011.

[36] A. Abouzied and et. al., "Hadoopdb in action: building real world applications," in *SIGMOD Conference*, 2010, pp. 1111–1114.