A Distributed Graph Engine for Web Scale RDF Data

Kai Zeng[†][*] Jiacheng Yang[♯][*] Haixun Wang[‡] Bin Shao[‡] Zhongyuan Wang[‡,♭]
[†]UCLA [♯]Columbia University [‡]Microsoft Research Asia [♭]Renmin University of China
kzeng@cs.ucla.edu jiachengy@cs.columbia.edu
{haixunw, binshao, zhy.wang}@microsoft.com

ABSTRACT

Much work has been devoted to supporting RDF data. But state-of-the-art systems and methods still cannot handle web scale RDF data effectively. Furthermore, many useful and general purpose graph-based operations (e.g., random walk, reachability, community discovery) on RDF data are not supported, as most existing systems store and index data in particular ways (e.g., as relational tables or as a bitmap matrix) to maximize one particular operation on RDF data: SPARQL query processing. In this paper, we introduce Trinity.RDF, a distributed, memory-based graph engine for web scale RDF data. Instead of managing the RDF data in triple stores or as bitmap matrices, we store RDF data in its native graph form. It achieves much better (sometimes orders of magnitude better) performance for SPARQL queries than the state-of-the-art approaches. Furthermore, since the data is stored in its native graph form, the system can support other operations (e.g., random walks, reachability) on RDF graphs as well. We conduct comprehensive experimental studies on real life, web scale RDF data to demonstrate the effectiveness of our approach.

1 Introduction

RDF data is becoming increasingly more available: The semantic web movement towards a web 3.0 world is proliferating a huge amount of RDF data. Commercial search engines including Google and Bing are pushing web sites to use RDFa to explicitly express the semantics of their web contents. Large public knowledge bases, such as DBpedia [9] and Probase [37] contain billions of facts in RDF format. Web content management systems, which model data in RDF, mushroom in various communities all around the world.

Challenges RDF data management systems are facing two challenges: namely, systems' scalability and generality. The challenge of scalability is particularly urgent. Tremendous efforts have been devoted to building high performance RDF systems and SPARQL engines [6, 12, 3, 36, 14, 5, 35, 27]. Still, scalability remains the biggest hurdle. Essentially, RDF data is highly connected graph data, and SPARQL queries are like subgraph matching queries. But most approaches model RDF data as a set of triples, and use RDBMS for storing, indexing, and query processing. These approaches do not scale as processing a query often involves a large number of join operations that produce large intermediate results. Furthermore, many systems, including SW-Store [5], Hexastore [35], and RDF-3x [27] are single-machine systems. As the size of RDF data keeps soaring, it is not realistic for single-machine approaches to provide good performance. Recently, several distributed RDF systems, such as SHARD [29], YARS2 [17], Virtuoso [15], and [20], have been introduced. However, they still model RDF data as a set of triples. The cost incurred by excessive join operations is further exacerbated by network communication overhead. Some distributed solutions try to overcome this limitation by brute-force replication of data [20]. However, this approach simply fails in the face of complex SPARQL queries (e.g., queries with a multi-hop chain), and has a considerable space overhead (usually exponential).

The second challenge lies in the generality of RDF systems. State-of-the-art systems are not able to support general purpose queries on RDF data. In fact, most of them are optimized for SPARQL only, but a wide range of meaningful queries and operations on RDF data cannot be expressed in SPARQL. Consider an RDF dataset that represents an entity/relationship graph. One basic query on such a graph is reachability, that is, checking whether a path exists between two given entities in the RDF data. Many other queries (e.g., community detection) on entity/relationship data rely on graph operations. For example, random walks on the graph can be used to calculate the similarity between two entities. All of the above queries and operations require some form of graph-based analytics [34, 28, 22, 33]. Unfortunately, none of these can be supported in current RDF systems, and one of the reasons is that they manage RDF data in some foreign forms (e.g., relational tables or bitmap matrices) instead of its native graph form.

Overview of Our Approach We introduce Trinity.RDF, a distributed in-memory RDF system that is capable of handling web scale RDF data (billion or even trillion triples). Unlike existing systems that use relational tables (triple stores) or bitmap matrices to manage RDF, Trinity.RDF builds on top of a memory cloud, and models RDF data in its native graph form (i.e., representing entities as graph nodes, and relationships as graph edges). We argue that such a memory-based architecture that logically and physi-

[*]This work is done at Microsoft Research Asia.

Proceedings of the VLDB Endowment, Vol. 6, No. 4
Copyright 2013 VLDB Endowment 2150-8097/13/02... $ 10.00.

cally models RDF in native graphs opens up a new paradigm for RDF management. It not only leads to new optimization opportunities for SPARQL query processing, but also supports more advanced graph analytics on RDF data.

To see this, we must first understand that most graph operations do not have locality [23, 31], and rely exclusively on random accesses. As a result, storing RDF graphs in disk-based triple stores is not a feasible solution since random accesses on hard disks are notoriously slow. Although sophisticated indices can be created to speed up query processing, they introduce excessive join operations, which become a major cost for SPARQL query processing.

Trinity.RDF models RDF data as an in-memory graph. Naturally, it supports fast random accesses on the RDF graph. But in order to process SPARQL queries efficiently, we still need to address the issues of how to reduce the number of join operations, and how to reduce the size of intermediary results. In this paper, we develop novel techniques that use efficient in-memory graph exploration instead of join operations for SPARQL processing. Specifically, we decompose a SPARQL query into a set of triple patterns, and conduct a sequence of graph explorations to generate bindings for each of the triple pattern. The exploration-based approach uses the binding information of the explored subgraphs to prune candidate matches in a greedy manner. In contrast, previous approaches isolate individual triple patterns, that is, they generate bindings for them separately, and make excessive use of costly join operations to combine those bindings, which inevitably results in large intermediate results. Our new query paradigm greatly reduces the amount of intermediate results, boosts the query performance in a distributed environment, and makes the system scale. We show in experiments that even without a smart graph partitioning scheme, Trinity.RDF achieves several orders of magnitude speed-up on web scale RDF data over state-of-the-art RDF systems.

We also note that since Trinity.RDF models data as a native graph, we enable a large range of advanced graph analytics on RDF data. For example, random walks, regular expression queries, reachability queries, distance oracles, community searches can be performed on web scale RDF data directly. Even large scale vertex-based analytical tasks on graph platforms such as Pregel [24] can be easily supported in our system. However, these topics are out of the scope of this paper, and we refer interested readers to the Trinity system [30, 4] for detailed information.

Contributions We summarize the novelty and advantages of our work as follows.

1. We introduce a novel graph-based scheme for managing RDF data. Trinity.RDF has the potential to support efficient graph-based queries, as well as advanced graph analytics, on RDF.

2. We leverage graph exploration for SPARQL processing. The new query paradigm greatly reduces the volume of intermediate results, which in turn boosts query performance and system scalability.

3. We introduce a new cost model, novel cardinality estimation techniques, and optimization algorithms for distributed query plan generation. These approaches ensure excellent performance on web scale RDF data.

Paper Layout The rest of the paper is organized as follows. Section 2 describes the difference between join operations and graph exploration. Section 3 presents the architecture of the Trinity.RDF system. Section 4 describes how we model RDF data as native graphs. Section 5 describes SPARQL query processing techniques. Section 6 shows experimental results. We conclude in Section 8.

2 Join vs. Graph Exploration

Joins are the major operator in SPARQL query processing. Trinity.RDF outperforms existing systems by orders of magnitude because it replaces expensive join operations by efficient graph exploration. In this section, we discuss the performance implications of the two different approaches.

2.1 RDF and SPARQL

Before we discuss join operations vs. graph exploration, we first introduce RDF and SPARQL query processing on RDF data. An RDF data set consists of statements in the form of (*subject*, *predicate*, *object*). Each statement, also known as as a triple, is about a fact, which can be interpreted as *subject* has a *predicate* property whose value is *object.* For example, a movie knowledge base may contain the following triples about the movie "Titanic":

```
(Titanic, has_award, Best_Picture)
(Titanic, casts, L_DiCaprio),
(J_Cameron, directs, Titanic)
(J_Cameron, wins, Oscar_Award)
...
```

An RDF dataset can be considered as representing a directed graph, with entities (i.e. *subjects* and *objects*) as nodes, and relationships (i.e. *predicates*) as directed edges. SPARQL is the standard query language for retrieving data stored in RDF format. The core syntax of SPARQL is a conjunctive set of triple patterns called a *basic graph pattern.* A *triple pattern* is similar to an RDF triple except that any component in the triple pattern can be a variable. A basic graph pattern describes a subgraph which a user wants to match against the RDF data. Thus, SPARQL query processing is essentially a subgraph matching problem. For example, we can retrieve the cast of an award-winning movie directed by a award-winning director using the following query:

Example 1.

```
SELECT ?movie, ?actor WHERE{
 ?director wins ?award .
 ?director directs ?movie .
 ?movie has_award ?movie_award .
 ?movie casts ?actor .}
```

SPARQL also contains other language constructs that support disjunctive queries and filtering.

2.2 Using Join Operations

Many state-of-the-art RDF systems store RDF data as a set of triples in relational tables, and therefore, they rely excessively on join operations for processing SPARQL queries. In general, query processing consists of two phases [25]: The first phase is known as the *scan* phase. It decomposes a SPARQL query into a set of triple patterns. For the query in Example 1, the triple patterns are *?director wins ?award*, *?director directs ?movie*, *?movie has_award ?movie_award*, and *?movie casts ?actor*. For each triple pattern, we scan tables or indices to generate bindings. Assuming we are processing the query against the RDF graph in Figure 1. The

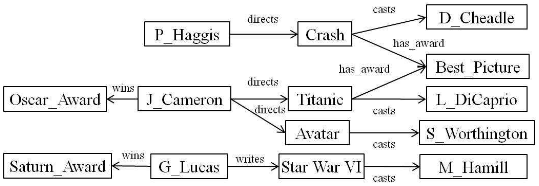


Figure 1: An example RDF graph

base tables that contain the bindings are shown in Table 1. The second phase is the *join* phase. The base tables are joined to produce the final answer to the query.

?director	**?award**
J_Cameron	*Oscar_Award*
G_Lucas	*Saturn_Award*

?director	**?movie**
P_Haggis	*Crash*
J_Cameron	*Titanic*
J_Cameron	*Avatar*

?movie	**?movie_award**
Titanic	*Best_Picture*
Crash	*Best_Picture*

?movie	**?actor**
Crash	*D_Cheadle*
Titanic	*L_Dicaprio*
Avatar	*S_Worthington*
Star War VI	*M_Hamill*

Table 1: Base tables and bound variables.

Sophisticated techniques have been used to optimize the order of joins to improve query performance. Still, the approach has inherent limitations: (1) It uses many costly join operations. (2) The scan-join process produces large redundant intermediary results. From Table 1, we can see that most intermediary join results will be produced in vain. After all, only *Titanic* directed by *J_Cameron* matches the query. Moreover, useless intermediary results may only be detected in later stages of the join process. For example, if we choose to join *?director directs ?movie* and *?movie casts ?actor* first, we will not know that the resulting rows related to *Avatar* and *Crash* are useless until joining with *?director wins ?award* and *?movie has_award ?movie_award*. Sideways Information Passing (SIP) [26] was proposed to alleviate this problem. SIP is a dynamic optimization technique for pipelined execution plans. It introduces *filters* on subject, predicate, or object identifiers, and passes these filters to joins and scans in other parts of the query that need to process similar identifiers.

2.3 Using Graph Explorations

In this paper, we adopt a new approach that greatly improves the performance of SPARQL query processing. The idea is to use *graph exploration* instead of joins.

The intuition can be illustrated by an example. Assume we perform the query in Example 1 over the RDF graph in Figure 1 starting with the pattern: *?director wins ?award*. After exploring the neighbors of *?award* connected via the *wins* edge, we find that the possible binding for *?director* is *J_Cameron* and *G_Lucas*. Then, we explore the graph further from node *J_Cameron* and *G_Lucas* via edge *directs*, and we generate bindings for *?director directs ?movie*. In the above exploration, we prune *G_Lucas* because it does not have a *directs* edge. Also, we do not produce useless bindings as those shown in Table 1, such as the binding (*P_Haggis*, *Crash*). Thus, we are able to prune unnecessary intermediate results efficiently.

The above intuition is only valid *if graph exploration can be implemented more efficiently than join.* This is not true for existing RDF systems. If the RDF graph is managed by relational tables, triple stores, or disk-based key-value stores, then we need to use join operations to implement graph exploration, which means graph exploration cannot be more efficient than join: With an index, it usually requires an $O(\log N)$ operation to access the triples relating to a *subject/object*[1]. In our work, we use native graphs to model RDF data, which enables us to perform the same operation in $O(1)$ time. With the support of the underlying architecture, we make graph exploration extremely efficient. In fact, Trinity.RDF can explore as many as 2.3 million nodes on a graph distributed over an 8-server cluster within one tenth of a second [30]. This lays the foundation for exploration-based SPARQL query processing.

We need to point out that the order of exploration is important. Starting with the highly selective pattern *?movie has_award ?movie_award*, we can prune a lot of candidate bindings of other patterns. If we explore the graph in a different order, i.e. exploring *?movie cast ?actor* followed by *?director directs ?movie*, then we will still generate useless intermediate results. Thus, query plans need to be carefully optimized to pick the optimal exploration order, which is not trivial. We will discuss our algorithm for optimal graph exploration plan generation in Section 5.

Note that graph exploration (following the links) is to certain extent similar to index-nested-loops join. However, index-nested-loops join is costly for RDBMS or disk-based data, because it needs a random access for each index lookup. Hence, in previous approaches, scan-joins, which perform sequential reads on sorted data, are preferred. Our approach further extends the random access approach in a distributed environment and minimizes the size of intermediate join results.

3 System Architecture

In this section, we give an overall description of the data model and the architecture of Trinity.RDF. We model and store RDF data as a directed graph. Each node in the graph represents a unique entity, which may appear as a *subject* and/or an *object* in an RDF statement. Each RDF statement corresponds to an edge in the graph. Edges are directed, pointing from *subjects* to *objects*. Furthermore, edges are labeled with the *predicates*. We will present the data structure for nodes and edges in more detail in Section 4.

To ensure fast random data access in graph exploration, we store RDF graphs in memory. A web scale RDF graph may contain billions of entities (nodes) and trillions of triples. It is unlikely that a web scale RDF graph can fit in the RAM of a single machine. Trinity.RDF is based on Trinity [30], which is a *distributed* in-memory key-value store. Trinity.RDF builds a graph interface on top of the key-value store. It randomly partitions an RDF graph across a cluster of commodity machines by hashing on the nodes. Thus, each machine holds a disjoint part of the graph. Given a SPARQL query, we perform search in parallel on each machine. During query processing, machines may need to exchange data as a query pattern may span multiple partitions.

Figure 2 shows the high level architecture of Trinity.RDF. A user submits his query to a proxy. The proxy generates a query plan and delivers the plan to all the Trinity machines, which hold the RDF data. Then, each machine executes the query plan under the coordination of the proxy. When the bindings for all the variables are resolved, all Trinity machines send back the bindings (answers) to the proxy where the final result is assembled and sent back to the user. As we can see, the proxy plays an important role in the architecture. Specifically, it performs the following tasks. First, it generates a query plan based on available statistics and indices. Second, it keeps track of the status of each Trinity

[1] N is the total number of RDF triples

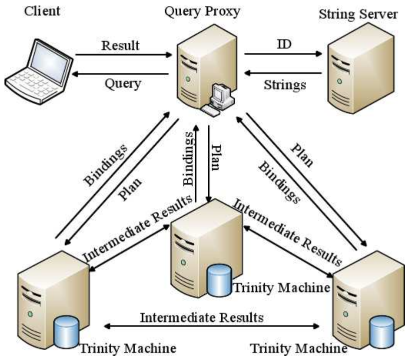


Figure 2: Distributed query processing framework

machine in query processing by, for example, synchronizing the execution of each query step. However, each Trinity machine not only communicates with the proxy. They also communicate among themselves during query execution to exchange intermediary results. All communications are handled by a message passing mechanism built in Trinity.

Besides the proxy and the Trinity machines, we also employ a string indexing server. We replace all literals in RDF triples by their ids. The string indexing server implements a literal-to-id mapping that translates literals in a SPARQL query into ids, and an id-to-literal mapping that maps ids in the output back to literals for the user. The mapping can be either implemented by a separate Trinity in-memory key-value store for efficiency, or by a persistent key-value store if memory space is a concern. Usually the cost of mapping is negligible compared to that of query processing.

4 Data Modeling

To support graph-based operations including SPARQL queries on RDF data more effectively, we store RDF data in its native graph form. In this section, we describe how we model and manipulate RDF data as distributed graphs.

4.1 Modeling Graphs

Trinity.RDF is based on Trinity, which is a key-value store in a memory cloud. We then create a graph model on top of the key-value store. Specifically, we represent each RDF entity as a graph node with a unique id, and store it as a key-value pair in the Trinity memory cloud:

$$(\textit{node-id}, \langle \textit{in-adjacency-list}, \textit{out-adjacency-list} \rangle) \qquad (1)$$

The key-value pair consists of the *node-id* as the key, and the node's adjacency list as the value. The adjacency list is divided into two lists, one for neighbors with incoming edges and the other for neighbors with outgoing edges. Each element in the adjacency lists is a (*predicate*, *node-id*) pair, which records the id of the neighbor, and the predicate on the edge.

Thus, we have created a graph model on top of the key-value store. Given any node, we can find the node-id of any of its neighbors, and the underlying Trinity memory cloud will retrieve the key-value pair for that node-id. This enables us to explore the graph from any given node by accessing its adjacency lists. Figure 3 shows an example of the data structure.

4.2 Graph Partitioning

We distribute an RDF graph across multiple machines, and this is achieved by the underlying memory cloud, which partitions the key-value pairs in a cluster. However, due to the characteristics of graphs, we need to look into how the graph is partitioned in order to ensure best performance.

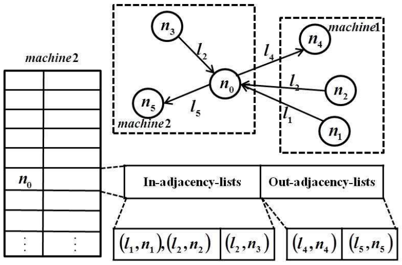


Figure 3: An example of model (1)

Two factors may have impact on network overhead when we explore a graph. The first factor is how the graph is partitioned. In our system, sharding is supported by the underlying key-value store, and the default sharding mechanism is hashing on node-id. In other words, the graph is randomly partitioned. Certainly, sophisticated graph partitioning methods can be adopted for sharding. However, graph partitioning is beyond the scope of this paper.

The second factor is how we model graphs on top of the key-value store. The model given by (1) may have potential problems for real-life large graphs. Many real-life RDF graphs are scale-free graphs whose node degrees follow the power law distribution. In DBpedia [9], for example, over 90% nodes have less than 5 neighbors, while some top nodes have more than 100,000 neighbors. The model may incur a large amount of network traffic when we explore the graph from a top node x. For simplicity, let us assume none of x's neighbors resides on the same machine as x does. To visit x's neighbors, we need to send the node-ids of its neighbors to other machines. The total amount of information we need to send across the network is exactly the entire set of node-ids in x's adjacency list. For the DBpedia data, in the worst case, whenever we encounter a top node in graph exploration, we need to send $800K$ data (each node-id is 64 bit) across the network. This is a huge cost in graph exploration.

We take the power law distribution into consideration in modeling RDF data. Specifically, we model a node x by the following key-value pair:

$$(\textit{node-id}, \langle in_1, \cdots, in_k, out_1, \cdots, out_k \rangle) \qquad (2)$$

where in_i and out_i are keys to some other key-value pairs:

$$(in_i, \textit{in-adjacency-list}_i) \qquad (out_i, \textit{out-adjacency-list}_i) \qquad (3)$$

The essence of this model is the following: The key-value pair $(in_i, \textit{in-adjacency-list}_i)$ and the nodes in $\textit{in-adjacency-list}_i$ are stored on the same machine i. In other words, we partition the adjacency lists in model (1) by machine.

The benefits of this design is obvious. No matter how many neighbors x has, we will send no more than k *nids* (in_i and out_i) over the network since each machine i, upon receiving nid$_i$, can retrieve x's neighbors that reside on machine i without incurring any network communication. However, for nodes with few neighbors, model (2) is more costly than model (1). In our work, we use a threshold t to decide which model to use. If a node has more than t neighbors, we use model (2) to map it to the key-value store; otherwise, we use model (1). Figure 4 gives an example with $t = 1$. Furthermore, in our design, all triples are stored decentralized at its subject and object. Thus, update has little cost as it only affects a few nodes. However, update is out of the scope of this paper and we omit detailed discussion here.

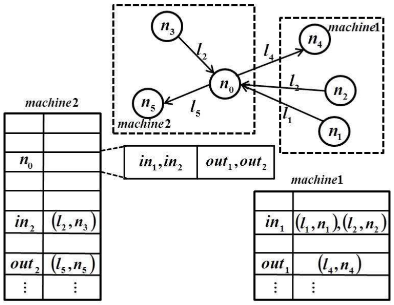


Figure 4: An example of model (2)

4.3 Indexing Predicates

Graph exploration relies on retrieving nodes connected by an edge of a given predicate. We use two additional indices for this purpose.

Local predicate indexing We create a local predicate index for each node x. We sort all (*predicate*, *node-id*) pairs in x's adjacency lists first by *predicate* then by *node-id*. This corresponds to the SPO or OPS index in traditional RDF approaches. In addition, we also create an aggregate index to enable us to quickly decide whether a node has a given *predicate* and the number of its neighbors connected by the predicate.

Global predicate indexing The global predicate index enables us to find all nodes that have incoming or outgoing neighbors labeled by a given predicate. This corresponds to the PSO or POS index in traditional approaches. Specifically, for each predicate, machine i stores a key-value pair

$$(\textit{predicate}, \langle \textit{subject-list}_i, \textit{object-list}_i \rangle)$$

where *subject-list*$_i$ (*object-list*$_i$) consists of all unique subjects (objects) with that predicate on machine i.

4.4 Basic Graph Operators

We provide the following three graph operators with which we implement graph exploration:

1. $LoadNodes(predicate, dir)$: Return nodes that have an *incoming* or *outgoing* edge labeled as *predicate*.

2. $LoadNeighborsOnMachine(node, dir, i)$: For a given *node*, return its *incoming* or *outgoing* neighbors that reside on machine i.

3. $SelectByPredicate(nid, predicate)$: From a given partial adjacency list specified by *nid*, return nodes that are labeled with the given *predicate*.

Here, *dir* is a parameter that specifies whether the predicate is an *incoming* or an *outgoing* edge. $LoadNodes()$ is straightforward to understand. When it is called, it uses the *global predicate index* on each machine to find nodes that have at least one incoming or outgoing edge that is labeled as *predicate*.

The next two operators together find specific neighbors for a given node. *LoadNeighborsOnMachine*() finds a node's incoming or outgoing neighbors on a given machine. But, instead of returning all the neighbors, it simply returns the in_i or out_i id as given in (2). Then, given the in_i or out_i id, *SelectByPredicate*() finds nodes in the adjacency list that is associated with the given *predicate*. Certainly, if the node has less than t neighbors, then its adjacency list is not distributed, and the two functions simply operate on the local adjacency list.

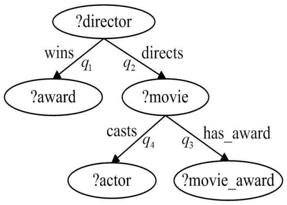
We now use some examples to illustrate the use of the above 3 operators on the RDF graph shown in Figure 4. $LoadNodes(l_2, out)$ finds n_2 on machine 1, and n_3 on machine 2. $LoadNeighborsOnMachine(n_0, in, 1)$ returns the partial adjacency list's id in_1, and $SelectByPredicate(in_1, l_2)$ returns n_2.

5 Query Processing

In this section, we present our exploration-based approach for SPARQL query processing.

5.1 Overview

We represent a SPARQL query Q by a *query graph* $\mathcal{G}$. Nodes in $\mathcal{G}$ denote subjects and objects in Q, and directed edges in $\mathcal{G}$ denote predicates. Figure 5 shows the query graph corresponding to the query in Example 1, and lists the 4 triple patterns in the query as q_1 to q_4.



- q_1: (*?director wins ?award*)
- q_2: (*?director directs ?movie*)
- q_3: (*?movie has_award ?movie_award*)
- q_4: (*?movie casts ?actor*)

Figure 5: The query graph of Example 1

With $\mathcal{G}$ defined, the problem of SPARQL query processing can be transformed to the problem of subgraph matching. However, as we pointed out in Section 2, existing RDF query processing and subgraph matching algorithms rely excessively on costly joins, which cannot scale to RDF data of billion or even trillion triples. Instead, we use efficient graph exploration in an in-memory key-value store to support fast query processing. The exploration is conducted as follows: We first decompose Q into an ordered sequence of triple patterns: $q_1, \cdots, q_n$. Then, we find matches for each q_i, and from each match, we explore the graph to find matches for q_{i+1}. Thus, to a large extent, graph exploration acts as joins. Furthermore, the exploration is carried out on all distributed machines in parallel. In the final step, we gather the matchings for all individual triple patterns to the centralized query proxy, and combine them together to produce the final results.

5.2 Single Triple Pattern Matching

We start with matching a single triple pattern. For a triple pattern q, our goal is to find all its matches $R(q)$. Let P denote the predicate in q, V denote the variables in q, and $B(V)$ denote the binding of V. If V is a free variable (not bound), we also use $B(V)$ to denote all possible values V can take. We regard a constant as a special variable with only *one* binding.

We use graph exploration to find matches for q. There are two ways of exploration: from subject to object (We first try to find matches for the subject in q, and then for each match, we find matches for the object in q. We denote this exploration as $\overrightarrow{q}$) and from object to subject (We denote this exploration as $\overleftarrow{q}$). We use *src* and *tgt* to refer to the source and target of an exploration (i.e., in $\overrightarrow{q}$ the *src* is the subject, while in $\overleftarrow{q}$ the *src* is the object).

Algorithm 1 MatchPattern(e)

```
obtain src, tgt, and predicate $p$ from $e$ ($e = \overrightarrow{q}$ or $e = \overleftarrow{q}$)

// On the src side:
if src is a free variable then
    $B(src) = \bigcup_{\forall p \in B(P)} LoadNodes(p, dir)$
set $M_i = \emptyset$ for all $i$ // initialize messages to machine $i$
for each $s$ in $B(src)$ do
    for each machine $i$ do
        $nid_i = LoadNeighborsOnMachine(s, dir, i)$
        $M_i = M_i \cup (s, nid_i)$
batch send messages $M$ to all machines

// On the tgt side:
for each ($s$, $nid$) in $M$ do
    for each $p$ in $B(P)$ do
        $N = SelectByPredicate(nid, p)$
        for each $n$ in $N \cap B(tgt)$ do
            $R = R \cup (s, p, n)$
return $R$
```

Algorithm 1 outlines the matching procedure using the basic operators introduced in Section 4.4. If *src* is a constant, we only need to explore from *one* node. If *src* is a variable, we initialize its bindings by calling *LoadNodes*, which searches the global predicate index to find the matches for *src*. Note that if the predicate itself is a free variable, then we have to load nodes for every predicate. After *src* is bound, for each node that matches *src* and for each machine i, we call *LoadNeighborsOnMachine*() to find the key nid_i. The node's neighbors on machine i are stored in the key-value pair with nid_i as the key. We then send nid_i to machine i.

Each machine, on receiving the message, starts the matching on the *tgt* side. For each eligible predicate p in $B(P)$, we filter neighbors in the adjacency list by p by calling *SelectByPredicate*(). If *tgt* is a free variable, any neighbor is eligible as a binding, so we add (s, p, n) as a match for every neighbor n. If *tgt* is a constant, however, only the constant node is eligible. As we treat a constant as a special variable with only one binding, we can uniformly handle these two cases: we match a new edge only if its target is in $B(tgt)$.

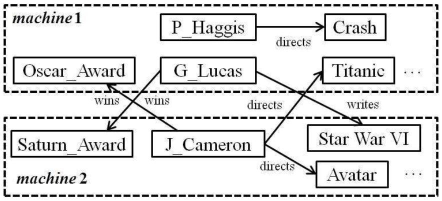


Figure 6: Distribution of the RDF graph in Figure 1

We use an example to demonstrate how *MatchPattern* works. Assume the RDF graph is distributed on two machines as shown in Figure 6. Suppose we want to find matches for $\overleftarrow{q_1}$ where q_1 is "*?director wins ?award*". In this case, *src* is *?award*. We first call *LoadNodes*(*wins*, *in*) to find $B(?award)$, which are nodes having an incoming *wins* edge. This results in *Oscar_Award* on machine 1, and *Saturn_Award* on machine 2. Next, on the target *?director* side, machine 1 gets the key of the adjacency list sent by *Saturn_Award*, and after calling *SelectByPredicate*(), it gets *G_Lucas*. Since the target *?director* is a free variable, any edge labeled with *win* will be matched. We add matching edge (*G_Lucas*, *wins*, *Saturn_Award*) to R. Similarly on machine 2, we get (*J_Cameron*, *wins*, *Oscar_Award*).

As Algorithm 1 shows, given a triple q, each machine performs *MatchPattern*() independently, and obtains and stores the results on the target side, that is, on machines where the target is matched. For the example in Figure 6, matches for $\overleftarrow{q_1}$ where q_1 is "*?director wins ?award*" are stored on machine 1, where the target *G_Lucas* is located. Table 2 shows the results on both machines for q_1. We use $R^i(q)$ to denote matches for of q on machine i. Note that the constant column *wins* is not stored.

(a) $R^1(q_1)$

?director	?award
G_Lucas	*Saturn_Award*

(b) $R^2(q_1)$

?director	?award
J_Cameron	*Oscar_Award*

Table 2: Individual matching result of q_1

5.3 Multiple Pattern Matching by Exploration

A query consists of multiple triple patterns. Traditional approaches match each pattern individually and join them afterwards. A single pattern may generate a large number of results, and this leads to large intermediary join results and costly joins. For the example of Figure 6, suppose we generate the matchings for pattern q_1, q_2 separately. The results are Table 2 for q_1 and Table 3 for q_2. We can see although *P_Haggis* has not won an award, we still generate (*Crash*, *P_Haggis*) in $R(q_2)$.

(a) $R^1(q_2)$

?movie	?director
Titanic	*J_Cameron*
Crash	*P_Haggis*

(b) $R^2(q_2)$

?movie	?director
Avatar	*J_Cameron*

Table 3: Individual matching result of q_2

Instead of matching single patterns independently, we treat the query as a sequence of patterns. The matching of the current pattern is based on the matches of previous patterns, i.e., we "explore" the RDF graph from the matches of previous patterns to find matches for the current pattern. In other words, we eagerly prune invalid matchings by exploration to avoid the cost of joining large sets of results later.

(a) $R^1(q_2)$

?movie	?director
Titanic	*J_Cameron*

(b) $R^2(q_2)$

?movie	?director
Avatar	*J_Cameron*

Table 4: Matching result of q_2 after matching q_1

We now use an example to illustrate the exploration and pruning process. Assume we explore the graph in Figure 1 in the order of $\overrightarrow{q_1}$, $\overrightarrow{q_2}$, $\overleftarrow{q_3}$, $\overrightarrow{q_4}$. Clearly, how the triple patterns are ordered may have a big impact on the intermediate results size. We discuss query plan optimization in Section 5.5.

There are two different cases in exploration and pruning, and they are examplified by matching $\overrightarrow{q_2}$ after $\overrightarrow{q_1}$, and by matching $\overleftarrow{q_3}$ after $\overrightarrow{q_2}$, repsectively. We describe them separately. *In the first case, the source of exploration is bound.* Exploring q_2 after q_1 belongs to this case, as the source *?director* is bound by q_1. So, instead of using *LoadNodes*() to find all possible directors, we start the exploration from existing bindings (*J_Cameron* and *G_Lucas*), so we won't generate movies not directed by award-winning directors. Moreover, note that in Figure 1, *G_Lucas* does not have a *directs* edge, so exploring from *G_Lucas* will not produce any matching triple. It means we can prune *G_Lucas* safely: There is no need to send the key to its adjacency-list across the network. The results are in Table 4, which contains fewer tuples than Table 3.

In the second case, the target of exploration is bound. Exploring q_3 after q_2 belongs to this case, as $?movie$ is bound to $\{Titanic, Avatar\}$ by $\overrightarrow{q_2}$. We only add results in this binding set to the matching results, namely (*Best_Picture*, *Titanic*). Independently, (*Best_Picture*, *Crash*) also satiesfies the pattern, but *Crash* is not in the binding set, so it is pruned. Furthermore, since the previous binding of *Avatar* does not match any triple in this round, it is also safely pruned from *?movie*'s binding. Finally, we incorporate the matches of q_3 into the result. As shown in Table 5, it now has three bound variables *?movie*, *?director*, and *?movie_award*, and contains one row (*Titanic*, *J_Cameron*, *Best_Picture*) on machine 1 where *Titanic* is located.

?movie	?director	?movie_award
Titanic	*J_Cameron*	*Best_Picture*

Table 5: Results after incorporating q_2 and q_3

5.4 Final Join after Exploration

We used two mechanisms to prune intermediate results: a binding is pruned if it cannot reach any bound target, or it cannot be reached from any bound source. Furthermore, once we prune the target (source), we also prune corresponding values from the source (target). This greatly reduces the size of the intermediary results, and does not incur much additional communication, as shown in the previous example.

However, invalid intermediary results may still remain after the pruning. This is because the pruning of q's intermediary results only affects the bindings of q and the immediate neighbors of q. Bindings of other patterns are not considered because otherwise we need to carry all historical bindings in exploration, which incurs big communication cost.

After the exploration finishes, we obtain all the matches in R. Since R is distributed and may contain invalid results, we gather these results to a centralized proxy and perform a final join to assemble the final answer. As we have eagerly pruned most of the invalid results in the exploration phase, our join phase is light-weight compared with traditional RDF systems that intensively rely on joins, and we simply adopt the left-deep join for this purpose.

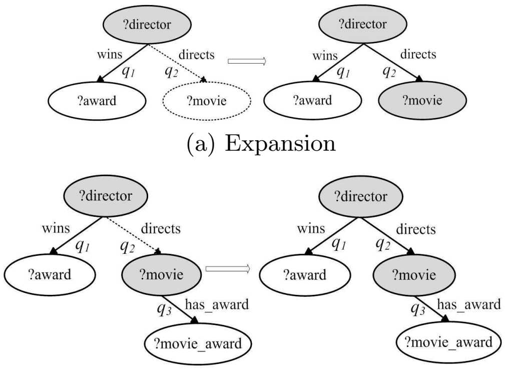
5.5 Exploration Plan Optimization

Section 5.3 described the query processing for an ordered sequence of triple patterns. The order has significant impact on the query performance. We now describe a cost-based approach for finding the optimal exploration plan.

We define an exploration plan as a graph traversal plan, and we denote it as a sequence $\langle e_1, \cdots, e_n\rangle$, where each e_i denotes a directed exploration of a predicate q_i in the query graph, that is, $e_i = \overrightarrow{q_i}$ or $e_i = \overleftarrow{q_i}$. The cost of the plan is $\Sigma_i cost(e_i)$, where $cost(e_i)$, the cost of matching $\overrightarrow{q_i}$ or $\overleftarrow{q_i}$, is roughly proportional to the size of q_i's results (Section 5.6 will describe cost estimation in more depth). Clearly, the size of q_i's results depends on the matching of some $q_j, j < i$. Thus, the total cost depends on the order of e_i's in the sequence.

Naive query plan optimization is costly. There are $n!$ different orders for a query graph with n edges, and for each q_i, there are two directions of exploration. It is also tempting to adopt the join ordering method in a relational query optimizer. However, there is a fundamental difference between our scenario and theirs. In the relational optimizer, later joins depend on previous intermediary join results, while for us, later explorations depend on previous intermediary bindings. The intermediary join results do not depend on the order of join, while the intermediary bindings do depend on the order of exploration. For example, consider two plans (1) $\{\overrightarrow{q_1}, \overrightarrow{q_2}, \overleftarrow{q_3}, \overrightarrow{q_4}\}$ and (2) $\{\overrightarrow{q_2}, \overrightarrow{q_3}, \overleftarrow{q_1}, \overrightarrow{q_4}\}$, where the first 3 elements are q_1, q_2, and q_3, but in different order. For the relational optimizer (ignore the direction of each q_i), the join results q_1, q_2, and q_3 are the same no matter how they are ordered. But in our case, plan (1) produces $\{Titanic\}$ and plan (2) produces $\{Titanic, Crash\}$ for $B(?movie)$, as shown in Table 5. The redundant *Crash* will makes $\overrightarrow{q_4}$ in plan (2) more costly than plan (1).

We now introduce our approach for exploration order optimization. For a query graph, we find exploration plans for its subgraphs (starting with single nodes), and expand/combine the plans until we derive the plan for the entire query graph. There are two ways to grow a subgraph: expansion and combination. Figure 7(a) depicts an example of expansion: we explore to a free variable or a constant and add an edge to the subgraph. The subgraph $\{q_1\}$ is expanded to a larger graph $\{q_1, q_2\}$. Another way to grow a subgraph is that we combine two disjoint subgraphs by exploring an edge starting from one subgraph to the other. Figure 7(b) shows such an example: we combine the subgraph with one edge q_1 with the subplan of q_3 by exploring $\overleftarrow{q_2}$. This way, we construct a larger subgraph from two smaller subgraphs.



(a) Expansion

(b) Combination

Figure 7: Expansion and combination examples

Now, we introduce heuristics for exploration optimization. Let $\mathcal{E}$ denote a subgraph, $R(\mathcal{E})$ denote its intermediary join results, and $B(\mathcal{E})$ denote the bindings of variables in $\mathcal{E}$. Note that in our exploration, we compute $B(\mathcal{E})$ only, but not $R(\mathcal{E})$. Furthermore, bindings for some variables in $\mathcal{E}$ may contain redundant values. We define a variable $?c$ as an *exploration point* if it satisfies $B(c) = \Pi_c R(\mathcal{E})$. Intuitively, node $?c$ is an exploration point if it does not contain any redundant value, in other words, each of its values must appear in the intermediary join results $R(\mathcal{E})$. We then adopt the following heuristics in subgraph expansion/combination.

HEURISTIC 1. *We expand a subgraph from its exploration point. We combine two subgraphs by connecting their exploration points.*

The reason we want to expand/combine at the exploration point is because the exploration points do not contain redundant values. Hence, they introduce fewer redundant values for other variables in the exploration.

After the expansion/combination, we need to determine the exploration points of the resulting graph. Heuristic 1 leads to the following property:

PROPERTY 1. *We expand a subgraph or combine two subgraphs through an edge. The two nodes on both ends of the edge are valid exploration points in the new graph.*

PROOF. For expansion from subgraph $\mathcal{E}$, we start from an exploration point c that satisfies $B(c) = \Pi_c R(\mathcal{E})$ and explore a new predicate $q = c \rightsquigarrow c'$. Based on our algorithm, we have $\Pi_c R(e) \subseteq \Pi_c R(\mathcal{E})$. Since $q \notin \mathcal{E}$ and $c' \notin \mathcal{E}$, we get $R(\mathcal{E} \cup q) = R(\mathcal{E}) \bowtie_c R(q)$. Thus:

$$\begin{aligned} \Pi_{c'} R(\mathcal{E} \cup q) &= \Pi_{c'}(R(\mathcal{E}) \bowtie_c R(q)) \\ &= \Pi_{c'} R(q) = B(c') \end{aligned}$$

which means c' is an exploration point of $\mathcal{E} \cup q$. After $B(c')$ is obtained, the algorithm uses it to prune $B(c)$ so that c's new binding satisfies $B(c) = \Pi_c R(q)$. Thus:

$$\begin{aligned} \Pi_c R(\mathcal{E} \cup q) &= \Pi_c(R(\mathcal{E}) \bowtie_c R(q)) \\ &= \Pi_c R(\mathcal{E}) \bowtie_c \Pi_c R(q) = \Pi_c R(q) = B(c) \end{aligned}$$

which means c is a valid exploration point of $\mathcal{E} \cup q$. Similarly, we can show Property 1 holds in subgraph combination. □

We use dynamic programming (DP) for exploration optimization. We use $(\mathcal{E}, c)$ to denote a state in DP. We start with subgraphs of size 1, that is, subgraphs of a single edge $q = u \rightsquigarrow v$. The states are $(\{q\}, u)$ and $(\{q\}, v)$. For their cost, we consider both explorations $\overleftarrow{q}$ and $\overrightarrow{q}$ to obtain the minimal cost of reaching the state.

After computing cost for subgraphs of size k, we perform expansion and combination to derive subgraphs of size $\geq k+1$. Specifically, assuming we are expanding $(\mathcal{E}, c)$ through edge $q = c \rightsquigarrow v$, we reach two states:

$$(\mathcal{E} \cup \{q\}, v) \quad \text{and} \quad (\mathcal{E} \cup \{q\}, c) \tag{4}$$

Let C denote the cost of the state before expansion, and C' the cost of the state after expansion. We have:

$$C' = \min\{C', C + cost(\overrightarrow{q})\} \tag{5}$$

Note that: i) We may reach the expanded state in different ways, and we record the minimal cost of reaching the state; ii) C is the cost of state of size $\leq k$, which is determined in previous iterations; iii) If q is in the other direction, i.e., $q = v \rightsquigarrow c$, then $cost(\overrightarrow{q})$ above becomes $cost(\overleftarrow{q})$.

For combining two states $(\mathcal{E}_1, c_1)$ and $(\mathcal{E}_2, c_2)$ where $\mathcal{E}_1 \cap \mathcal{E}_2 = \emptyset$ through edge $q = c_1 \rightsquigarrow c_2$, we reach two states:

$$(\mathcal{E}_1 \cup \mathcal{E}_2 \cup q, c_1) \quad \text{and} \quad (\mathcal{E}_1 \cup \mathcal{E}_2 \cup q, c_2) \tag{6}$$

Let C_1 and C_2 denote the cost of the two states before combination. We update the cost of the combined state to be:

$$C' = \min\{C', C_1 + C_2 + cost(\overrightarrow{q})\} \tag{7}$$

We now show the complexity of the DP:

THEOREM 1. *For a query graph $\mathcal{G}(V, E)$, the DP has time complexity $O(n \cdot |V| \cdot |E|)$ where n is the number of connected subgraphs in $\mathcal{G}$.*

Here is a brief sketch-proof: There are $n \cdot |V|$ states in the DP process (each subgraph $\mathcal{E}$ can have $|\mathcal{E}| \leq |V|$ nodes), and each update can take at most $O(|E|)$ time.

THEOREM 2. *Any acyclic query Q with query graph $\mathcal{G}$ is guaranteed to have an exploration plan.*

We give a brief sketch-proof. Our optimizer resembles the idea of semi-joins although we do not perform join. Bernstein et al. proved [10] that for any relation in an acyclic query, there exists a semi-join program that can fully reduce the relation by evaluating each join condition only once. By mapping each node in $\mathcal{G}$ to a relation, and an edge in $\mathcal{G}$ to a join condition, we can see that our algorithm can find an exploration plan that evaluates each pattern exactly once.

Discussion. There are two cases we have not considered formally: i) $\mathcal{G}$ is cyclic, and ii) $\mathcal{G}$ contains a join on predicates. For the first case, our algorithm may not be able to find an exploration plan. However, we can break a cycle in $\mathcal{G}$ by duplicating some variable in the cycle. For example, one heuristic to pick the break point is that we break a cycle at node u if it has the smallest cost when we explore u's adjacent edges uv_1 and uv_2 from u; and in the case of many cycles, we repeatedly apply this process. The resulting query graph $\mathcal{G}'$ is acyclic. We can apply our algorithm to search for an approximate plan. For the second case, consider a join on predicate ($?s$ $?p$ $?u$), ($?x$ $?p$ $?y$). Here, we cannot explore from the first pattern from bound variables $?s$ or $?u$ because they are not connected with the second pattern. To handle this case, after we explore an edge with a variable predicate, we iterate through all unvisited patterns sharing the same predicate variable $?p$, i.e. ($?x$ $?p$ $?y$), and use *LoadNodes* to create an initial binding for $?x$ and $?y$. This enable us to contine the exploration.

5.6 Cost Estimation

SPARQL selectivity estimation is a challenging task. Stocker et al [32] assumes subject, predicate and object are independent and the selectivity of each triple is the product of the three. The result is far from optimal. RDF-3X [25] uses two approaches: One assumes independence between triples and relies on traditional join estimation techniques. The other mines frequent join pathes for large joins and maintains statistics for these pathes, which is very costly and unfeasible for web-scale RDF data.

We propose a novel estimation method that captures the correlation between triples but requires little extra statistics and data preprocessing. Specifically, we estimate $cost(e)$ where $e = \overrightarrow{q}$ or $\overleftarrow{q}$. In the following, we estimate $cost(\overrightarrow{q})$ only, and the estimation of $cost(\overleftarrow{q})$ can be obtained in the same way. Also, we use *src* and *tgt* to denote the source and target nodes in e. The computation cost of matching q is estimated as the size of the results, namely $|R(q)|$. Since we operate in a distributed environment, we model communication cost as well. During exploration, we send bindings and ids of adjacency lists across network, so we measure communication cost as the binding size of the source node of the exploration, i.e. $|B(src)|$. The final $cost(\overrightarrow{q})$ is a linear combination of $|R(q)|$ and $|B(src)|$.

Now, if we know $|B(src)|$, we can estimate $|R(q)|$ and $|B(tgt)|$ as

$$|R(q)| = |B(src)| \frac{C_p}{C_p(src)}, |B(tgt)| = |B(src)| \frac{C_p(tgt)}{C_p(src)}$$

where $C_q, C_q(src), C_q(tgt)$ are the number of triples and connected subject/object with predicate p, which can be obtained from a single global predicate index look-up. If the predicate of q is unknown, we consider the average case for all possible predicates. For the case where the source or target of q is constant, we use the local predicate index to get a more accurate estimation.

We then derive $|B(src)|$. For a standalone $\overrightarrow{q}$, we can derive $|B(src)|$ from the global predicate index. When $\overrightarrow{q}$ is not standalone, the binding size of *src* is affected by related patterns already explored. To capture this correlation, we maintain a two-dimensional *predicate* × *predicate* matrix[2]. Each cell (i, j) stores four statistics: the number of unique nodes with predicates p_i, p_j as its incoming/outgoing edges (4 combinations). When no confusion shall arise, we simply use $C_{p_i p_j}$ to denote the correlation.

As shown in Section 5.5, the query optimizer handles two cases: expansion and combination. In the first case, assume we expand through a new edge p_2 from variable x which is already connected with p_1. Assume the original binding size of x is N_x. We have the new binding size N'_x as

$$N'_x = N_x \frac{C_{p_1 p_2}}{C_{p_1}} \tag{8}$$

The second case is combining two edges p_1 and p_2 on x. Assume the original binding sizes of x with predicate p_1 and predicate p_2 are $N_{x,1}$ and $N_{x,2}$ respectively. We have the new binding size N'_x as

$$N'_x = N_{x,1} N_{x,2} \frac{C_{p_1 p_2}}{C_{p_1} C_{p_2}} \tag{9}$$

For more complex cases in expansion and combination during exploration, e.g. expanding a new pattern from a subgraph, or joining two subgraphs, we simply pick the most selective pair from all pairs of involved predicates.

6 Evaluation

We evaluate Trinity.RDF on both real-life and synthetic datasets, and compare it against the state-of-the-art centralized and distributed RDF systems. The results show that Trinity.RDF is a highly scalable, highly parallel RDF engine.

Systems We implement Trinity.RDF in C#, and deploy it on a cluster, wherein each machine has 96 GB DDR3 RAM, two 2.67 GHz Intel Xeon E5650 CPUs, each with 6 cores and 12 threads, and one 40Gb/s InfiniBand Network adaptor. The OS is 64-bit Windows Server 2008 R2 Enterprise with service pack 1.

We compare Trinity.RDF with centralized RDF-3X [27] and BitMat [8], as well as distributed MapReduce-RDF-3X (a Hadoop-based RDF-3X solution [20]). We deploy the three systems on machines running 64 bit Linux 2.6.32 using the same hardware configuration as used by Trinity.RDF. Just like Trinity.RDF, all of the competitor systems map literals to IDs in query processing. But BitMat relies on manual mapping. For a fair comparison, we measure the query execution time by excluding the cost of literal/ID mapping. Since all of these three systems are disk-based, we report both their warm-cache and cold-cache time.

Datasets We use two real-life and one synthetic datasets. The real-life datasets are the Billion Triple Challenge 2010 dataset (BTC-10) [1] and DBpedia's SPARQL Benchmark (DBPSB) [2]. The synthetic dataset is the Lehigh University Benchmark (LUBM) [16]. We generated 6 datasets of different sizes using the LUBM data generator v1.7. We summarize the statistics of the data and some exemplary queries (LUBM queries are also published in [8]) in Table 6 and Table 7. All of the queries used in our experiments can be found online[3].

Dataset	#Triples	#S/O
BTC-10	3,171,793,030	279,082,615
DBPSB	15,373,833	5,514,599
LUBM-40	5,309,056	1,309,072
LUBM-160	21,347,999	5,259,588
LUBM-640	85,420,588	21,037,012
LUBM-2560	341,888,947	84,202,729
LUBM-10240	1,367,122,031	336,711,191
LUBM-100000	9,956,527,583	2,452,700,932

Table 6: Statistics of datasets used in experiments

BTC-10	S1	S2	S3	S4	S5	S6	S7	
# of joins	7	5	9	12	6	9	7	
DBPSB	D1	D2	D3	D4	D5	D6	D7	D8
# of joins	1	1	2	3	3	4	4	5
LUBM	L1	L2	L3	L4	L5	L6	L7	
# of joins	6	1	6	4	1	3	6	

Table 7: Statistics of queries used in experiments

Join vs. Exploration We compare graph exploration (Trinity.RDF) with scan-join (RDF-3X and BitMat) on DBPSB and LUBM-160 datasets. The experiment results show that Trinity.RDF outperforms RDF-3X and BitMat; and more importantly, its superiority does not just come from its in-memory architecture, but from the fact that graph exploration itself is more efficient than join.

For a fair comparison, we set up Trinity.RDF on a single machine, so we have the same computation infrastructure for all three systems. Specifically, to compare the in-memory performance, we set up a 20 GB *tmpfs* (an in-memory file system supported by Linux kernel from version 2.4), and deploy the database images of RDF-3X and BitMat in the in-memory file system.

The first observation is that managing RDF data in graph form is space-efficient. The database images of LUBM-160 and DBPSB in Trinity.RDF are of 1.6G and 1.9G respectively, which are smaller or comparable to RDF-3X (2GB and 1.4GB respectively), and are much more efficient than BitMat (3.6GB and 19GB respectively even without literal/ID mapping).

The results on LUBM-160 and DBPSB are shown in Table 8 and 9. For RDF-3X and BitMat, both in-memory and on-disk (cold-cache) performances are reported. Trinity.RDF outperforms the on-disk performances of RDF-3X and BitMat by a large margin for all queries: For most queries, Trinity.RDF has 1 to 2 orders of magnitude performance gain; for some queries, it has 3 orders of magnitude speed-up. The results from the in-memory performance comparison are more interesting. Here, since all systems are memory-based, the comparison is solely about graph exploration versus scan-join. We can see that the improvement is easily 2-5 fold, and for L4, Trinity.RDF has 3 orders of magnitude speed-up. This also shows that, although SIP and semi-join are proposed to overcome the shortcomings of the scan-join approach, they are not always effective, as shown by L1, L2, L4, D1, D7, etc. Moreover, we vary the complexity of DBPSB queries from 1 join to 5 joins, where Trinity.RDF achieves very stable performance gain. It proves that our query algorithm can effectively find the optimal exploration order even for complex queries with many patterns.

We also show that in-memory RDF-3X or BitMat runs slightly better than Trinity.RDF on L2, L3 and D2. This is because L2, D2 have very simple structures and few intermediate results, and Trinity has the overhead due to its C# implementation.

[2] In many RDF datasets, there is a special predicate *rdf:type* which characterizes the types of entities. Since the number of entities associated with a certain type varies greatly, we treat each type as a different *predicate*.

[3] http://research.microsoft.com/trinity/Trinity.RDF.aspx

	L1	L2	L3	L4	L5	L6	L7	Geo. mean
Trinity.RDF	**281**	132	110	**5**	**4**	**9**	**630**	**46**
RDF-3X (In Memory)	34179	**88**	485	7	5	18	1310	143
BitMat (In Memory)	1224	4176	**49**	6381	6	51	2168	376
RDF-3X (Cold Cache)	35739	653	1196	735	367	340	2089	1271
BitMat (Cold Cache)	1584	4526	286	6924	57	194	2334	866

Table 8: Query run-time in milliseconds on the LUBM-160 dataset (21 million triples)

	D1	D2	D3	D4	D5	D6	D7	D8	Geo. mean
Trinity.RDF	**7**	220	**5**	**7**	**8**	**21**	**13**	**28**	**15**
RDF-3X (In Memory)	15	**79**	14	18	22	34	68	35	29
BitMat (In Memory)	335	1375	209	113	431	619	617	593	425
RDF-3X (Cold Cache)	522	493	394	498	366	524	458	658	482
BitMat (Cold Cache)	392	1605	326	279	770	890	813	872	639

Table 9: Query run-time in milliseconds on the DBPSB dataset (15 million triples)

Performance on Large Datasets We experiment on three datasets, LUBM-10240, LUBM-100000 and BTC-10, to study the performance of Trinity.RDF on billion scale datasets, and compare it against both centralized and distributed RDF systems. The results are shown in Table 11, 12 and 13. As distributed systems, Trinity.RDF and MapReduce-RDF-3X are deployed on a 5-server cluster for LUBM-10240, a 8-server cluster for LUBM-100000 and a 5-server cluster for BTC-10. And we implement the directed 2-hop guarantee partition for MapReduce-RDF-3X.

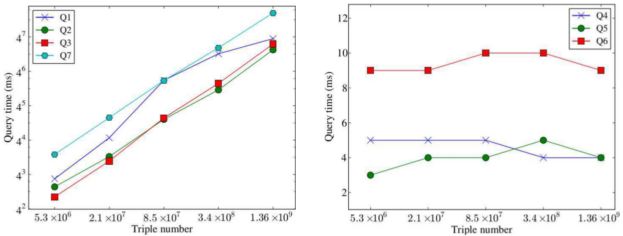
BitMat fails to run on BTC-10 as it generates terabytes of data for just a single SPO index. Similar issues happen on LUBM-100000. For some datasets and queries, BitMat and RDF-3X fail to return answers in a reasonable time (denoted as "aborted" in our experiment results).

On LUBM-10240 and LUBM-100000, Trinity.RDF gets similar performance gain over RDF-3X and BitMat as on LUBM-160. Even compared with MapReduce-RDF-3X, Trinity.RDF gives surprisingly competitive performance, and for some queries, e.g. L4-6, Trinity.RDF is even faster. These results become more remarkable if we note that all the LUBM queries are with simple structures, and MapReduce-RDF-3X specially partitions the data so that these queries can be answered fully in parallel with zero network communication. In comparison, Trinity.RDF randomly partitions the data, and has a network overhead. However, data partitioning is orthogonal to our algorithm and can be easily applied to reduce the network overhead. This is also evidenced by the results of L4-6. L4-6 only explore a small set of triples (as shown in Table 14) and incur little network overhead. Thus, Trinity.RDF outperforms even MapReduce-RDF-3X. Moreover, MapReduce-RDF-3X's partition algorithm incurs great space overhead. As shown in Table 10, MapReduce-RDF-3X indexes twice as many as triples than RDF-3X and Trinity.RDF do.

	LUBM-10240	LUBM-100000	BTC-10
#triple	2,459,450,365	20,318,973,699	6,322,986,673
Overhead	1.80X	2.04X	1.99X

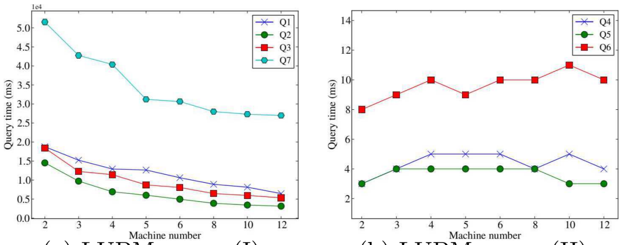
Table 10: The space overhead of MapReduce-RDF-3X compared with the original datasets

The BTC-10 benchmark has more complex queries, some with up to 13 patterns. In specific, S3, S4, S6 and S7 are not parallelizable without communication in MapReduce-RDF-3X, and additional MapReduce jobs are invoked to answer the queries. In Table 13, we list separately the time of RDF-3X jobs and MapReduce jobs for MapReduce-RDF-3X. Interestingly, Trinity.RDF shows up to 2 orders of magnitude speed-up even over the RDF-3X jobs of MapReduce-RDF-3X. This is probably because MapReduce-RDF-3X divides a query into multiple subqueries and each subquery produces a much larger result set. This result again proves the performance impact of exploiting the correlations between patterns in a query, which is the key idea behind graph exploration.



(a) LUBM group (I) (b) LUBM group (II)

Figure 8: Data scalability



(a) LUBM group (I) (b) LUBM group (II)

Figure 9: Machine scalability

	L1	L2	L3	L4	L5	L6	L7
LUBM-160	397	173040	0	10	10	125	7125
LUBM-10240	2502	11016920	0	10	10	125	450721

Table 14: The result sizes of LUBM queries

Scalability To evaluate the scalability of our systems, we carry out two experiments by (1) scaling the data while fixing the number of servers, and (2) scaling the number of servers while fixing the data. We group LUBM queries into two categories according to the sizes of their results, as shown in Table 14: (I) Q1, Q2, Q3, Q7. The results of these queries increase as the size of the dataset increases. Note that although Q3 produces an empty result set, it is more similar to queries in group (I) as its intermediate result set increases when the input dataset increases. (II) Q4, Q5, Q6. These queries are very selective, and produce results of constant size as the size of dataset increases.

Varying size of data: We test Trinity.RDF running on a 3-server cluster on 5 datasets LUBM-40 to LUBM-10240 of increasing sizes. The results are shown in Figure 8 (a) and (b). Trinity.RDF utilizes selective patterns to do efficient

	L1	L2	L3	L4	L5	L6	L7	Geo. mean
Trinity.RDF	**12648**	6018	8735	**5**	**4**	**9**	31214	**450**
RDF-3X (Warm Cache)	36m47s	14194	27245	8	8	65	69560	2197
BitMat (Warm Cache)	33097	209146	**2538**	aborted	407	1057	aborted	5966
RDF-3X (Cold Cache)	39m2s	18158	34241	1177	1017	993	98846	15003
BitMat (Cold Cache)	39716	225640	9114	aborted	494	2151	aborted	9721
MapReduce-RDF-3X (Warm Cache)	17188	**3164**	16932	14	10	720	**8868**	973
MapReduce-RDF-3X (Cold Cache)	32511	7371	19328	675	770	1834	19968	5087

Table 11: Query run-times in milliseconds for the LUBM-10240 dataset (1.36 billion triples)

	L1	L2	L3	L4	L5	L6	L7	Geo. mean
Trinity.RDF	176	21	119	**0.005**	**0.006**	**0.010**	126	**1.494**
RDF-3X (Warm Cache)	aborted	96	363	0.011	0.006	0.021	548	1.726
RDF-3X (Cold Cache)	aborted	186	1005	874	578	981	700	633.842
MapReduce-RDF-3X (Warm Cache)	**102**	**19**	**113**	0.022	0.016	0.226	**51.98**	2.645
MapReduce-RDF-3X (Cold Cache)	171	32	151	1.113	0.749	1.428	89	13.633

Table 12: Query run-times in seconds for the LUBM-100000 dataset (9.96 billion triples)

pruning. Therefore, Trinity.RDF achieves constant size of intermediate results and stable performance for group (II) regardless of the increasing data size. For group (I), Trinity.RDF scales linearly as the size of the dataset increases, which shows that the network overhead is alleviated by the efficient pruning of intermediate results in graph exploration.

Varying number of machines: We deploy Trinity.RDF in clusters with varying number of machines, and test its performance on dataset LUBM-10240. The results are shown in Figure 9 (a) and (b). For group (I), the query time of Trinity.RDF decrease reciprocally w.r.t. the number of machines. which testifies that Trinity.RDF can efficiently utilize the parallelism of a distributed system. Moreover, although more partitions increase the amount of intermediate data delivered across network, our storage model effectively bounds this overhead. For group (II), due to selective query patterns, the intermediate results are relatively small. Using more machines does not improve the performance, but again the performance is very stable and is not impacted by the extra network overhead.

7 Related Work

Tremendous efforts have been devoted to building high performance RDF management systems [12, 36, 14, 5, 35, 27, 26, 8, 7, 17]. State-of-the-art approaches can be classified into two categories:

Relational Solutions Most existing RDF systems use a relational model to manage RDF data, i.e. they store RDF triples in relational tables, and use RDBMS indexing to tune query processing, which aim solely at answering SPARQL queries. SW-Store [5] exploits the fact that RDF data has a small number of predicates. Therefore, it vertically partitions RDF data (by predicates) into a set of property tables, maps them onto a column-oriented database, and builds subject-object index on each property table; Hexastore [35] and RDF-3x [27] manage all triples in a giant triple table, and build indices of all six combinations (SPO, SOP, etc.).

The relational model decides that SPARQL queries are processed as large join queries, and most prior systems rely on SQL join optimization techniques for query processing. RDF-3x [27], which is considered the fastest existing system, proposed sophisticated bushy-join planning and fast merge join for query answering. However, this approach requires scanning large fraction of indexes even for very selective queries. Such redundancy overhead quickly becomes a bottleneck for billion triple datasets and/or complex queries. Several join optimization techniques are proposed. SIP (*sideways information passing*) is a dynamic optimization technique for pipelined execution plans [26]. It introduces filters on subject, predicate, or object identifiers, and passes these filters to other joins and scans in different parts of the operator tree that need to process similar identifiers. This introduces opportunities to avoid some unnecessary index scans. BitMat [8] uses a matrix of bitmaps to compress the indexes, and use lightweight semi-join operations on compressed data to reduce the intermediate result before actually joining. However, these optimizations do not solve the fundamental problem of the join approach. In comparison, our exploration-based approach is radically different from the join approach.

Graph-based Solutions Another direction of research investigated the possibility of storing RDF data as graphs [18, 7, 11]. Many argued that graph primitives besides pattern matching (SPARQL queries) should be incorporated into RDF languages, and several graph models for advanced applications on RDF data have been proposed [18, 7]. There are several non-distributed implementations, including one that builds an in-memory graph model for RDF data using Jena, and another that stores RDF as a graph in an object-oriented database [11]. However, both of them are single-machine solutions with limited scalability. A related research area is subgraph matching [13, 40, 19, 39] but most solutions rely on complex indexing techniques that are often very costly, and do not have the scalability to process web scale RDF graphs.

Recently, several distributed RDF systems [17, 15, 29, 20, 21] have been proposed. YARS2 [17], Virtuoso [15] and SHARD [29] hash partition triples across multiple machines and parallelize the query processing. Their solutions are limited to simple index loop queries and do not support advanced SPARQL queries, because of the need to ship data around. Huang et al. [20] deploy single-node RDF systems on multiple machines, and use the MapReduce framework to synchronize query execution. It partitions and aggressively replicates the data in order to reduce network communication. However, for complex SPARQL queries, it has high time and space overhead, because it needs additional MapReduce jobs and data replication. Furthermore, Husain et at [21] developed a batch system solely relying on MapReduce for SPARQL queries. It does not provide real-time query support. Yang et al. [38] recently proposed a graph partition management strategy for fast graph query processing, and demonstrate their system on answering SPARQL queries. However, their work focuses on partition optimization but not on developing scalable graph query engines. Further, the partitioning strategy is orthogonal to our so-

	S1	S2	S3	S4	S5	S6	S7	Geo. mean
Trinity.RDF	**12**	10	**31**	**21**	**23**	**33**	**27**	**21**
RDF-3X (Warm Cache)	108	8407	27428	62846	32	260	238	1175
RDF-3X (Cold Cache)	5265	23881	41819	91140	1041	3065	1497	8101
MapReduce-RDF-3X (Warm Cache w/o MapReduce)	132	**8**	4833	6059	24	1931	2732	453
MapReduce-RDF-3X (Cold Cache w/o MapReduce)	2617	661	13755	18712	801	4347	7950	3841
MapReduce-RDF-3X (MapReduce)	N/A	N/A	39928	39782	N/A	33699	33703	36649

Table 13: Query run-times in milliseconds for BTC-10 dataset (3.17 billion triples)

lution and Trinity.RDF can apply their algorithm on data partitioning to achieve better performance.

8 Conclusion

We propose a scalable solution for managing RDF data as graphs in a distributed in-memory key-value store. Our query processing and optimization techniques support SPARQL queries without relying on join operations, and we report performance numbers of querying against RDF datasets of billions of triples. Besides scalability, our approach also has the potential to support queries and analytical tasks that are far more advanced than SPARQL queries, as RDF data is stored as graphs. In addition, our solution only utilizes basic (distributed) key-value store functions and thus can be ported to any in-memory key-value store.

9 References

[1] Billion Triple Challenge. http://challenge.semanticweb.org/.

[2] DBpedia SPARQL Benchmark (DBPSB). http://aksw.org/Projects/DBPSB.

[3] Jena. http://jena.sourceforge.net.

[4] Trinity. http://research.microsoft.com/en-us/projects/trinity/.

[5] D. J. Abadi, A. Marcus, S. Madden, and K. Hollenbach. SW-Store: a vertically partitioned DBMS for Semantic Web data management. *VLDB J.*, 18(2):385–406, 2009.

[6] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In *SemWeb*, 2001.

[7] R. Angles and C. Gutiérrez. Querying rdf data from a graph database perspective. In *ESWC*, pages 346–360, 2005.

[8] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "Bit" loaded: a scalable lightweight join query processor for RDF data. In *WWW*, pages 41–50, 2010.

[9] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. DBpedia: A Nucleus for a Web of Open Data. In *ISWC/ASWC*, pages 722–735, 2007.

[10] P. A. Bernstein and D.-M. W. Chiu. Using Semi-Joins to Solve Relational Queries. *J. ACM*, 28(1):25–40, 1981.

[11] V. Bönström, A. Hinze, and H. Schweppe. Storing rdf as a graph. In *LA-WEB*, pages 27–36, 2003.

[12] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *ISWC*, 2002.

[13] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. Fast graph pattern matching. In *ICDE*, pages 913–922, 2008.

[14] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An Efficient SQL-based RDF Querying Scheme. In *VLDB*, 2005.

[15] O. Erling and I. Mikhailov. Virtuoso: RDF Support in a Native RDBMS. In *Semantic Web Information Management*, pages 501–519. 2009.

[16] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2-3):158–182, 2005.

[17] A. Harth, J. Umbrich, A. Hogan, and S. Decker. Yars2: A federated repository for querying graph structured data from the web. In *ISWC/ASWC*, pages 211–224, 2007.

[18] J. Hayes and C. Gutierrez. Bipartite graphs as intermediate model for rdf. In *ISWC*, 2004.

[19] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, 2008.

[20] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11), 2011.

[21] M. F. Husain, J. P. McGlothlin, M. M. Masud, L. R. Khan, and B. M. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *IEEE Trans. Knowl. Data Eng.*, 23(9):1312–1327, 2011.

[22] J. Lu, Y. Yu, K. Tu, C. Lin, and L. Zhang. An Approach to RDF(S) Query, Manipulation and Inference on Databases. In *WAIM*, pages 172–183, 2005.

[23] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry. Challenges in Parallel Graph Processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

[24] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[25] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1), 2008.

[26] T. Neumann and G. Weikum. Scalable Join Processing on Very Large RDF Graphs. In *SIGMOD*, 2009.

[27] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.

[28] M. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[29] K. Rohloff and R. E. Schantz. High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store. In *PSI EtA*, 2010.

[30] B. Shao, H. Wang, and Y. Li. The Trinity graph engine. Technical Report 161291, Microsoft Research, 2012.

[31] B. Shao, H. Wang, and Y. Xiao. Managing and mining large graphs: Systems and implementations. In *SIGMOD*, 2012.

[32] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW*, 2008.

[33] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *Proceedings of the VLDB Endowment*, 5(9):788–799, 2012.

[34] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual Labeling: Answering Graph Reachability Queries in Constant Time. In *ICDE*, page 75, 2006.

[35] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

[36] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *SWDB*, pages 131–150, 2003.

[37] W. Wu, H. Li, H. Wang, and K. Zhu. Probase: A probabilistic taxonomy for text understanding. In *SIGMOD*, 2012.

[38] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *SIGMOD Conference*, pages 517–528, 2012.

[39] F. Zhu, Q. Qu, D. Lo, X. Yan, J. Han, and P. S. Yu. Mining top-k large structural patterns in a massive network. *PVLDB*, 4(11):807–818, 2011.

[40] L. Zou, L. Chen, and M. T. Özsu. Distancejoin: Pattern match query in a large graph database. *PVLDB*, 2(1):886–897, 2009.