Distributed Real-Time Knowledge Graph Serving

(Invited Paper)

Liang He[†*], Bin Shao[‡], Yatao Li[‡], Enhong Chen[†]
[†]University of Science and Technology of China, Hefei, China
[‡]Microsoft Research, Beijing, China
hshl05@mail.ustc.edu.cn, binshao@microsoft.com, yatli@microsoft.com, cheneh@ustc.edu.cn

***Abstract*—The acquisition of knowledge becomes scalable. Due to the great connectedness, knowledge data by its very nature are complex entity graphs with rich schemata. The machine-processable knowledge keeps its pace with the phenomenal "Big Data" era. On the one hand, we have a revolutionary way of piling knowledge up; on the other hand, the technology of making the knowledge graph accessible, i.e. how to serve the knowledge to support real-life applications, evolves slowly. This paper presents our efforts of serving real-world knowledge graphs at scale for real-time query processing.**

I. Introduction

Semantic web has attracted many attentions from both academia and industry for many years. Machine-accessible knowledge is becoming pervasive. To realize the semantic web vision [1], many standards are proposed for knowledge management. Among the proposed standards, RDF (Resource Description Framework) recommendated by W3C is the dominating way of representing knowledge. RDF is simple yet powerful. An RDF dataset is logically a collection of triples – $\langle subject, predicate, object \rangle$ where $subject$ and $object$ are resources and $predicate$ is the relationship connecting them. This representation structure forms a directed labeled graph, where the nodes represent resources and the edges represent the named relationships between the resources.

On the one hand, many knowledge repositories are built either by communities, such as MPI's Yago [2] and Freebase [3], or by knowledge extraction from text corpus such as DBpedia [4], [5], CMU's NELL [6], IBM's DeepQA [7], and Microsoft's Probase [8]. A large number of knowledge graph data sets are available thanks to the *Linked Open Data* [9] project. This project was started for publishing and interlinking the knowledge data scattered over the web. Many important data sets are published under this project, for example, the DBpedia dataset.

On the other hand, a few RDF stores are proposed to manage the ever growing knowledge data sets. Most of them are either RDBMS backed or native stores. And these RDF stores usually support SPARQL query language. SPARQL is the de facto query language for RDF data sets. The SPARQL query execution process is essentially a subgraph matching process on a knowledge graph.

Serving a large knowledge graph is never an easy task. The challenges come from: 1) Large data size: For a real-life system, data size does matter. Graph query processing algorithms with $O(n^2)$ or higher complexity are common; they are infeasible for large graph data [10]. 2) Complex data schema: Compared to social graphs which tend to have a few data types, e.g. person and post, a real-world knowledge graph usually has thousands of entity types and relationships. 3) Diversity of queries: Users may need to query knowledge data in different ways other than SPARQL, including knowledge exploration, keyword search, and so on.

The paper presents our experience obtained in developing a system for serving real-world knowledge graphs. The paper is organized as follows: Section II introduces how knowledge graphs are modeled and served on top of the Trinity graph engine. The system implementation details are presented in Section III. Section IV concludes.

II. Knowledge Serving Framework atop Trinity

Let us start from a real-life knowledge graph query example. Fig. 1 is an example of *relation search* on a graph with about 25.49 billion triple facts. In this example, we want to find the relations between a given set of entities. For this example, we can find 94 relations between *Tom Cruise*, *Mimi Rogers*, *Nicole Kidman*, and *Katie Holmes* within 100 milliseconds using the system we are going to introduce. Fig. 1 visualizes some selected relations.

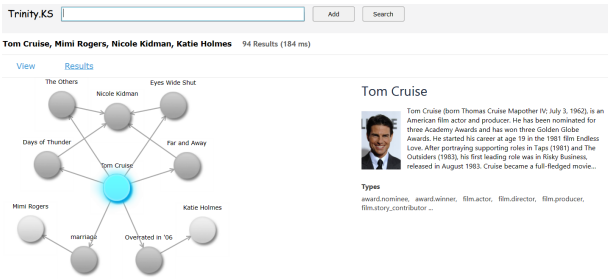

Fig. 1: Relation Search over Knowledge Graph

Our goal is to design a knowledge graph system which serves online queries in real time. Specifically, the system is designed for large knowledge graphs with billions of entities; it supports index-free graph exploration; it must be flexible enough to support a large range of knowledge graph queries. In this section, we will introduce how we build a system satisfying these requirements.

A. Knowledge Graph Modeling

We build our system on top of Trinity [11], on which index-free graph exploration can be easily implemented. We will

*This work was done at Microsoft Research (Beijing, China).

reference our system as *Trinity.KS* in what follows. Trinity provides very efficient random data access supports. It can explore millions of edges in a graph with 800 million nodes and 100 billion edges in 100 ms [11]. This lays a good foundation for graph serving without using costly structural indexes. By supporting user-specified data modeling and graph computation paradigms, Trinity can easily morph into an efficient distributed graph serving system.

The data modeling in Trinity is facilitated by a declarative language called *Trinity Specification Language* (TSL). We use TSL to model all kinds of graphs. TSL specifies the data schema for both graph nodes and edges. The basic unit used for specifying data schema is called *cell struct*. Syntactically, *cell struct* resembles the struct construct of the programming language C/C++ or C#. A cell struct can contain an arbitrary number of data fields. A data field is either a primitive value or a collection of primitive values. Fig. 2 shows a sample TSL script that defines a knowledge graph entity type *Person*. The *Person* type contains three data fields. Among them, the *Friends* field is annotated by *EdgeType*, indicating this field is an "edge" pointing to other graph nodes. In Trinity, a graph node can be referenced by a 64-bit integer. Therefore, we can just use a list of *Int64* to represent an entity's adjacent nodes. The modifier *optional* before *DateOfBirth* indicates the data field is nullable.

```
cell struct Person {
   String Name;
   optional DateTime DateOfBirth;
   [EdgeType:SimpleEdge]
   List⟨Int64⟩ Friends;
}
```

Fig. 2: TSL script that specifies an entity type

From the graph point of view, RDF data itself is a native simple graph with subjects/objects being nodes and predicates being labeled edges. Given an entity, we can use $\langle key, value \rangle$ pairs to represent its corresponding RDF predicates/values. Despite the simplicity, the straightforward data modeling mechanism is suboptimal for both data storage and access. Linear search is needed to get a certain property of an entity. To tackle this problem, Trinity.KS models RDF data in a strongly typed way. Each data property specified by TSL can be directly accessed by invoking a pre-compiled method.

RDF schema is expressed as *meta triples*. Meta triples are special triples that use a special vocabulary to specify data types and their corresponding schemata. The mapping from RDF meta data to Trinity TSL types is shown in Fig. 3. A typical real-life knowledge data set may have thousands of entity types and properties. For these RDF data sets, it is infeasible to manually specifying the schema for each entity. To automate graph modeling process, we implemented a TSL script generator for RDF meta data extraction and TSL generation.

RDF(S) Statement	TSL Mapping
T rdf:type rdfs:Class	Trinity type *T*
p rdf:type rdfs:Property	Trinity property *p*
p rdfs:domain T	*p* belongs to type *T*
p rdfs:range ClassType	*p* is a neighbor property
p rdfs:range LiteralType	*p* is a literal property
p_1 *rdfs:subPropertyOf* p_2	p_1 has the same setting as p_2

Fig. 3: Mapping RDF Meta Triples to TSL

There is another data modeling challenge posed by RDF data: an RDF entity may belong to more than one RDF classes. For example, consider the entity named "Pal" who is a *Dog*. But at the same time "Pal" is also an *Actor*[1]. We call entities like "Pal" multi-typed entities. We cannot simply use the inheritance or subtyping mechanism to model multi-typed entities because the relationship established between the classes of an entity is not an "isA" or subtype relationship.

A straightforward approach to modeling multi-typed entities is to create a compound RDF class combining multiple classes and then map it to a single graph node type. Unfortunately, this approach is infeasible for large graphs because it causes combinatorial explosion.

In Trinity.KS, we decompose a multi-typed entity into several segments with each one corresponds to a Trinity cell type. These segments of an entity are still strongly typed and organized by a mediator. The mediator provides a set of unified data access interfaces to applications and routes applications' accesses to the right segments. The strongly typed segments are called a *cell group*, consisting of a *root cell*, several *child cell*s, and a *generic cell*. The root cell is the mediator that organizes child cells and routes data accesses to child cells' pre-compiled methods. A root cell consists of three parts:

$$\langle \textit{entity-id}, \textit{type-list}, \textit{child-list} \rangle \tag{1}$$

The *entity-id* keeps the original entity ID in RDF (usually URI or IRI). Each element of the *child-list* is a pointer pointing to one child cell. Each child cell stores a portion of an entity's properties defined by its corresponding RDF class. The "Pal" entity, for instance, has two child cells – a *Dog cell* and an *Actor cell*. To maintain user-defined or unknown properties, an optional generic cell can be defined:

$$\langle \textit{adjacency-list}, \textit{property-list} \rangle \tag{2}$$

Each element in the *adjacency-list* is a ⟨*predicate, cell-id*⟩ pair which records the cell id of the neighbor node and the predicate on the edge. The *property-list* is a collection of ⟨*predicate, value*⟩ pairs for literal properties.

For an entity, its root cell is placed at the head, followed by all its child cells. The generic cell, if it has, is placed at the end. Thus the physical layout of an entity looks like:

$$\langle \textit{root-cell}, \textit{child-cell}_1, \textit{child-cell}_2, \ldots, \textit{generic-cell} \rangle \tag{3}$$

procedure GETNEIGHBOR($entity\_id$)
  $root \leftarrow$ LOADROOTCELL($entity\_id$)
  $neis \leftarrow \emptyset$ // initialize neighbors with an empty set
  for each $cell$ in $root$'s $child\text{–}list$ **do**
    $neis \leftarrow neis \cup cell.adjacentVertex$
  end for
  return $neis$
end procedure

Fig. 4: An Entity Access API Example

To manipulate an entity as a whole, Trinity.KS provides a set of entity access APIs on top of the cell manipulation primitives provided by Trinity. For example, GETNEIGHBOR procedure

[1]As a dog actor, "Pal" portrayed Lassie in several films (e.g., *Lassie Come Home*).

is done by visiting all the adjacent neighbors of the child cells as shown in in Fig. 4. The method LOADROOTCELL loads the root cell for the specified $entity\_id$ while the $adjacentVertex$ method returns all the adjacent neighbors of a given $cell$.

III. SYSTEM IMPLEMENTATION

In this section, we introduce the modularized system architecture, the details of knowledge graph importing/loading, and the performance numbers of Trinity.KS on a real-life large knowledge graph.

A. System Architecture

Trinity.KS has three levels of service abstraction to provide the flexibility of supporting different knowledge serving tasks: *Storage Layer*, *Graph Layer*, and *Application Layer*. The storage layer serves as the data storage back-end. The purpose of this layer is to manage the knowledge graph and some runtime data in Trinity, providing data manipulation interfaces at the granularity of cells. On top of them, the graph layer implements a strongly typed graph model and provides data manipulation interfaces at the granularity of entities. These interfaces are also wrapped up as RESTful APIs. The application layer built on top of the graph layer provides knowledge powered application features (e.g. knowledgebase powered question answering) to the end users. Details of each layer are elaborated as follows.

1) Storage Layer: This layer manages the knowledge graph data in Trinity. We explain how this layer works from a data flow perspective. To manage a large knowledge graph data set, the first issue we need to address is how to place the data in the distributed storage, namely, data sharding. In Trinity.KS, data sharding is supported by Trinity, and the default mechanism is sharding by hashing cell ids. In other words, the graph is randomly partitioned. Each machine of the Trinity cluster manages a graph partition in a strongly typed manner and leverages Trinity's message passing mechanism for server side computation as illustrated in Fig.8. Cells of the same entity are guaranteed to be placed on one machine to reduce the message passing costs during the query processing. This can be easily realized by a carefully designed encoding mechanism on cell id since Trinity allows applications to override its built-in data partitioning scheme.

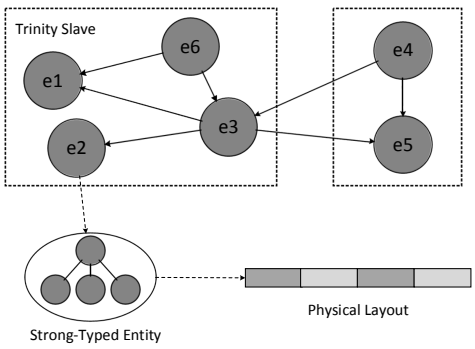

Fig. 8: Trinity.KS Storage Layer

2) Graph Layer: This layer provides an entity graph view via the strongly typed entity model. The core capability provided by this layer is a set of index-free graph exploration APIs. As elaborated in the knowledge serving framework, we are able to traverse among entities on top of Trinity cells. Advanced graph operators, such as finding shortest paths, performing random walks, can be built on these interfaces. For neighbor nodes on the local machine, we can directly perform graph exploration. We need to use the message passing mechanism provided by Trinity to perform user-defined server side computation to explore the graph across network.

To facilitate the process of building knowledge applications, utility APIs (e.g. entity auto completion APIs) are also provided by the graph layer. These helper interfaces are used as the basic building blocks of the application layer.

3) Application Layer: With a rich set of graph access interfaces provided by the graph layer and the storage layer, a large range of knowledge powered applications can be easily built. Trinity.KS implements a few built-in graph search features, such as *Relation Search*, *Graphical Knowledge Query*, and *Knowledge Graph Explorer*. Currently, the services provided by Trinity.KS is mainly consumed by knowledge powered natual language processing applications. A set of knowledge powered text comprehension applications are built on top of Trinity.KS by our peer research groups.

B. Graph Importing

Converting RDF data set to a distributed knowledge graph is a not a trivial task. The graph importing/loading procedure consists of three steps, i.e. data preprocessing, graph schema generation, and graph construction.

The input of the data preprocessing process is the raw RDF triples. Each Trinity slave in the cluster scans the shared raw RDF triples to extract its data partition. After obtaining the local data partition, each slave sorts the triples by subject and predicate.

Then, the data is transformed into an intermediate format. During the transformation, a) the data is cleaned using a set of predefined data cleaning rules; b) the RDF meta triples are extracted. The meta data on each Trinity slave will be aggregated at the end of the data transformation. Then a TSL script describing the graph schema is generated from the aggregated meta data using a tool mentioned earlier.

The graph construction process is performed after data cleaning and graph schema generation. Since the graph are already partitioned in the data preprocessing step, the distributed graph can be constructed in parallel by all Trinity slaves. Once the graph is constructed, it can be accessed and manipulated via the pre-compiled methods.

C. Query Processing

The most distinguishing characteristic of Trinity.KS is its index-free graph query processing paradigm. To illustrate the idea, we take *entity relation search* as an example. Answering a relation search query is equivalent to answering multiple shortest path queries under a max hop threshold k.

Typical shortest path query processing employs pre-built indexes. There is an obvious tradeoff between the query

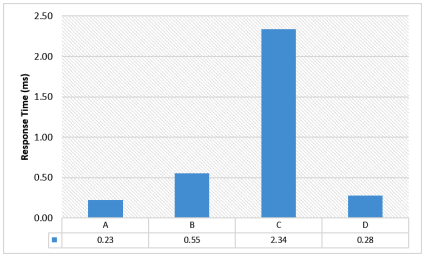

Fig. 5: API Response Time

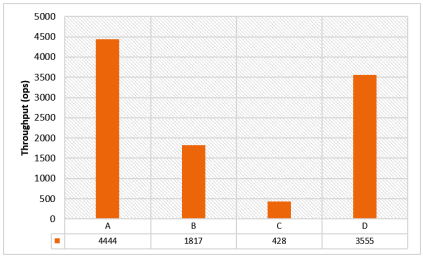

Fig. 6: API Throughput

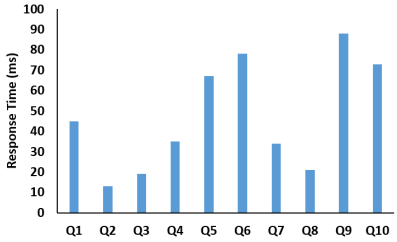

Fig. 7: Relation Search Response

response time and indexing costs. In order to serve the graph data in a real-time manner, the algorithms that heavily relies on indexes usually have a costly preprocessing procedure. Unfortunately, the state-of-the-art algorithms typically have $O(n^2)$ or higher complexities for index construction. The knowledgeq graph we are handling has 2.4 billion graph node, n^2 means 5.76×10^{18}, which is practically infeasible if we adopt these indexing based query processing methods.

To avoid building the prohibitive indexes, Trinity.KS uses fast index-free graph traversal to answer graph queries. For knowledge graph query processing, we have two key ingredients: a) Leveraging the rich schemata of the knowledge graph to prune graph traversal paths. As discussed earlier, knowledge graph usually has very complex data schema compared to other graphs, such as social networks or web graphs. On the one hand, the complex data schema poses great challenges for graph modeling; on the other hand, the rich schema information can help us prune the graph traversal space during query processing. For example, suppose we are to find shortest paths between two persons, from the graph schema we know that the *Person*'s predicate *profession* will never route to another *Person*, thus edges with this predicate can be pruned during traversing. b) A highly optimized asynchronous fan-out search (AFOS). AFOS is similar to what a distributed breadth-first search (BFS) does, but there is a clear difference. AFOS performs graph exploration hop by hop like BFS, however the search process is fully asynchronous, meaning a sub-search task never needs to wait for the termination of other sub-search tasks invoked at the same hop. Compared to BFS, AFOS can greatly reduce the synchronous overhead.

By leveraging the fast random data access capability provided by the underlying Trinity infrastructure, the index-free graph traversal based query processing paradigm works very well for billion node knowledge graphs. The performance numbers on a large real-life knowledge graph is presented in the next subsection.

D. Performance on Real-world Knowledge Graph

We are serving a large real-world knowledge graph with about 25.49 billion of triple facts using Trinity.KS steadily for more than a year.

Some performance numbers for primitive graph exploration APIs and *Relation Search* queries are showed in Fig. 5-7. All primitive graph exploration API calls can return within 2.5 milliseconds with the network cost included. At the same time, the system also has high throughput as shown by Figure 6. Fig. 7 shows the response time of 10 randomly sampled entity relation search queries. For most of the relation search queries, our system can respond within 100 milliseconds.

IV. Conclusion

The paper presents a real-time knowledge graph serving system Trinity.KS on top of a distributed in-memory graph engine called Trinity. It can efficiently support basic knowledge graph exploration APIs as well as advanced knowledge graph search queries for real-world billion node knowledge graphs. A set of knowledge-consuming applications, especially for machine text comprehension, are built within our research lab based on the knowledge graph serving capability provided by Trinity.KS.

Acknowledgments

This work is partially supported by the National Science Foundation for Distinguished Young Scholars of China (Grant No. 61325010).

References

[1] T. Berners-Lee, J. Hendler, O. Lassila *et al.*, "The semantic web," *Scientific american*, vol. 284, no. 5, pp. 28–37, 2001.

[2] F. M. Suchanek, G. Kasneci, and G. Weikum, "Yago: a core of semantic knowledge," in *Proceedings of the 16th international conference on World Wide Web*. ACM, 2007, pp. 697–706.

[3] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "Freebase: a collaboratively created graph database for structuring human knowledge," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 1247–1250.

[4] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann, "Dbpedia-a crystallization point for the web of data," *Web Semantics: science, services and agents on the world wide web*, vol. 7, no. 3, pp. 154–165, 2009.

[5] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, *Dbpedia: A nucleus for a web of open data*. Springer, 2007.

[6] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka Jr, and T. M. Mitchell, "Toward an architecture for never-ending language learning." in *AAAI*, vol. 5, 2010, p. 3.

[7] D. Ferrucci, E. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A. A. Kalyanpur, A. Lally, J. W. Murdock, E. Nyberg, J. Prager *et al.*, "Building watson: An overview of the deepqa project," *AI magazine*, vol. 31, no. 3, pp. 59–79, 2010.

[8] W. Wu, H. Li, H. Wang, and K. Q. Zhu, "Probase: A probabilistic taxonomy for text understanding," in *SIGMOD '12*, pp. 481–492.

[9] "http://www.linkeddata.org/."

[10] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *Proceedings of the VLDB Endowment*, vol. 5, no. 9, pp. 788–799, 2012.

[11] B. Shao, H. Wang, and Y. Li, "Trinity: a distributed graph engine on a memory cloud," in *SIGMOD '13*. New York, NY, USA: ACM, pp. 505–516.