University of Zurich[UZH]

**Zurich Open Repository and Archive**
University of Zurich
Main Library
Strickhofstrasse 39
CH-8057 Zurich
www.zora.uzh.ch

Year: 2015

# Random-walk triplerush: asynchronous graph querying and sampling

Stutz, Philip ; Paudel, Bibek ; Verman, Mihaela ; Bernstein, Abraham

Abstract: Most Semantic Web applications rely on querying graphs, typically by using SPARQL with a triple store. Increasingly, applications also analyze properties of the graph structure to compute statistical inferences. The current Semantic Web infrastructure, however, does not efficiently support such operations. Hence, developers have to painstakingly retrieve the relevant data for statistical post-processing. In this paper we propose to rethink query execution in a triple store as a highly parallelized asynchronous graph ex- ploration on an active index data structure. This approach also allows to integrate SPARQL-querying with the sam- pling of graph properties. To evaluate this architecture we implemented Random Walk TripleRush, which is built on a distributed graph processing system and operates by routing query and path descriptions through a novel active index data structure. In experiments we find that our architecture can be used to build a competitive distributed graph store. It can often return first results quickly, thanks to its asynchronous architecture. We show that our architecture supports the execution of various types of random walks with restarts that sample interesting graph properties. We also evaluate the scalability and show that the architecture supports fast answer times even on a dataset with more than a billion triples.

DOI: https://doi.org/10.1145/2736277.2741687

Posted at the Zurich Open Repository and Archive, University of Zurich
ZORA URL: https://doi.org/10.5167/uzh-111243
Conference or Workshop Item
Published Version

Originally published at:
Stutz, Philip; Paudel, Bibek; Verman, Mihaela; Bernstein, Abraham (2015). Random-walk triplerush: asynchronous graph querying and sampling. In: 24th International World Wide Web Conference (WWW 2015), Florence, Italy, 18 May 2015 - 22 May 2015, 1034-1044.
DOI: https://doi.org/10.1145/2736277.2741687

# Random Walk TripleRush: Asynchronous Graph Querying and Sampling

Philip Stutz
Department of Informatics
University of Zurich
Zurich, Switzerland
stutz@ifi.uzh.ch

Bibek Paudel
Department of Informatics
University of Zurich
Zurich, Switzerland
paudel@ifi.uzh.ch

Mihaela Verman
Department of Informatics
University of Zurich
Zurich, Switzerland
verman@ifi.uzh.ch

Abraham Bernstein
Department of Informatics
University of Zurich
Zurich, Switzerland
bernstein@ifi.uzh.ch

## ABSTRACT

Most Semantic Web applications rely on querying graphs, typically by using SPARQL with a triple store. Increasingly, applications also analyze properties of the graph structure to compute statistical inferences. The current Semantic Web infrastructure, however, does not efficiently support such operations. This forces developers to extract the relevant data for external statistical post-processing.

In this paper we propose to rethink query execution in a triple store as a highly parallelized asynchronous graph exploration on an active index data structure. This approach also allows to integrate SPARQL-querying with the sampling of graph properties.

To evaluate this architecture we implemented *Random Walk TripleRush*, which is built on a distributed graph processing system. Our evaluations show that this architecture enables both competitive graph querying, as well as the ability to execute various types of random walks with restarts that sample interesting graph properties. Thanks to the asynchronous architecture, first results are sometimes returned in a fraction of the full execution time. We also evaluate the scalability and show that the architecture supports fast query-times on a dataset with more than a billion triples.

## 1. INTRODUCTION

Use cases such as social network analysis, monitoring of financial transactions, or analysis of web pages and their links all require storage, retrieval, and analysis of large-scale graphs. To address this need, many have researched the development of efficient triple stores [1, 28, 21]. These systems borrow from the database literature to investigate efficient means for storing large graphs and retrieving subgraphs, which are usually defined via a pattern matching language such as SPARQL. Even though these systems process graphs, most of them leverage decades of research results in efficient processing of partial answer-sets by mapping the graphs into set-/array-style internal data structures. They are built like a centralized database, raising the question of scalability and parallelism within query execution.

To increase the parallelism of such graph-stores, modern solutions propose the use of parallel operators [30], sideways information-passing [20], or even pipelined operations and replication [8]. Other approaches focus on building triple stores based on specialized programming models for distributed systems: MapReduce [5] has been used to aggregate results from multiple single-node RDF stores in order to support distributed query processing [9] or to process whole SPARQL query execution pipelines (e.g., [14]). *Whilst these systems efficiently support the storage and retrieval, they mostly fall short on the support of graph-analytics. Hence, developers have to painstakingly retrieve the relevant data for statistical post-processing in a suitable tool.*

In this paper we rethink query execution within graph stores in the light of the changes of computer architectures. *We propose to exploit the large number of CPU-cores of modern servers via the parallel exploration of partial bindings.* Specifically, we explore each partial binding to a query in parallel akin to graph-exploration; this means (i) forking of the exploration whenever more than one binding is possible, (ii) returning the result when all variables of an exploration are bound, and (iii) aborting the exploration when it reaches a dead end (i.e., it cannot match a triple pattern).

*This re-conceptualization of triple-stores has the side-effect that it can efficiently support numerous graph-analytic algorithms* such as Random Walks with Restarts (RWR)—the basis of many approaches to information extraction and reasoning in noisy domains—or basic graph-algorithms such as shortest-path computations. This approach has the advantage to support the integration of statistical inference with SPARQL-based querying, which can provide better results in classification / learning [10], and simplifies the specification of restrictions on RWR via the re-use of SPARQL.

.

We implemented *Random Walk TripleRush* (RW-TR)[1] to explore this architecture. RW-TR is built on the distributed graph processing system SIGNAL/COLLECT [24].[2] Whilst traditional stores pipe data through query processing operators, RW-TR asynchronously routes query descriptions through an active data-structure. For this reason, RW-TR does not use any joins in the traditional sense, but searches the index graph in parallel.

As a consequence, the contributions of this paper are the following: first and foremost, we propose a novel active index-structure that supports the parallel and distributed exploration of answers to SPARQL-queries. Second, we show how this architecture can be extended with support for RWR – an important graph-analytic approach. Third, we present an extensive evaluation of the architecture that includes (i) vertical-, horizontal-, and data-scalability experiments, (ii) an evaluation of the time until the first result is returned, which is sometimes computed much faster than the whole results set, (iii) a benchmark against two other triple stores in the single-node scenario, where RW-TR is on average more than 10 times faster, and (iv) a comparison with two other distributed triple stores at the billon-triples scale, where RW-TR is very competitive. Fourth and last, we show the effectiveness of our RWR computations via a use case.

In the following, we succinctly discuss the relevant related work, describe the novel distributed architecture, as well as the functionality and interactions of its building blocks. We then compare the architecture with traditional graph-store approaches. Next, we evaluate the approach on multiple benchmarks and show that it can offer competitive performance, as well as good scalability. We close with a discussion of the limitations.

## 2. RELATED WORK

Studies related to RW-TR can be divided into four categories: (i) distributed in-memory triple-stores, (ii) extensions to SPARQL, (iii) graph computation frameworks, and (iv) studies into RDF index structures.

*Distributed in-memory triple stores:* Most closely related to RW-TR are Trinity.RDF [30], which relies on parallel operators to improve SPARQL performance, and TriAD [8], which relies on pipelined operations and replication. Both systems are competitive in terms of SPARQL performance (see also Section 5.4) but are limited to pure SPARQL processing. In addition, both use a different approach to parallelization. Trinity.RDF relies on a distributed Bulk-Synchronous approach [27] whilst TriAD uses extensive pre-processing and replication of indices. RW-TR uses an asychronous querying approach, which allows efficient embedding of graph sampling.

*SPARQL extensions:* A number of projects have proposed extending SPARQL with additional functionality. Corese [3, 4] and iSPARQL [12], for example, provide support for approximate matching and SPARQL-ML [11] extends SPARQL with statistical relational learning operators. Whilst these approaches show how SPARQL could be extended, they typically do not do so efficiently. Only recently have benchmarks been proposed to integrate SPARQL processing with more traditional graph processing tasks.[3] We are unaware of any other system that combines efficient SPARQL processing with efficient graph sampling.

*Distributed graph computation frameworks:* A number of distributed graph processing frameworks have been proposed in recent years [16, 24, 6]. Whilst these systems provide a basis for building distributed analytic solutions, they do not provide a high-level (querying) language such as a SPARQL extension to answer analytic queries. Trinity.RDF [30], which is layered on top of Trinity, does offer SPARQL querying but does not provide any support for sampling queries or analytics, which would have to be implemented manually.

Sedge [29] introduces different graph partition management techniques to minimize inter-machine communication during query processing. The system's effectiveness is demonstrated by answering SPARQL queries. In contrast, the focus of our work is not to find better partitions of the graph or manage them effectively, but to propose a new way of thinking about distributed triple stores with sampling capabilities.

*RDF index structures:* RW-TR reflects the insights gathered about RDF indexing in the past years [21, 28] in that it builds a multi-level structure of increasingly specific nodes. It differs significantly from these investigations in that it proposes a query execution as a highly parallelized, asynchronous routing of partially bound results through the index. Some aspects of RW-TR have similarities with the pointer-chasing problem [18, 17, 13], where future references are prefetched to achieve locality. RW-TR's general query execution is, however, fundamentally different, as it routes (passive) partial solutions through an actively processing index structure rather than employing a (possibly parallelized) program that operates on an optimized but passive index structure. RW-TR does exploit locality when compressing lower-level lists with delta-encoding.

## 3. RW-TR ARCHITECTURE

RW-TR is built leveraging the large-scale, parallel and distributed graph processing framework SIGNAL/COLLECT[4] [24, 25]. It allows to specify graph computations in terms of vertex-centric methods. In contrast to other frameworks, SIGNAL/COLLECT allows for asynchronous execution, multiple vertex types, and the ability to change the graph structure during the execution. Conceptually, SIGNAL/COLLECT-vertices can be seen as actor-like active elements, where the framework handles messaging, parallelization and distribution.

The core idea of RW-TR is to build a triple store with three types of SIGNAL/COLLECT vertices: Each *index vertex* corresponds to a triple pattern, each *triple vertex* corresponds to an RDF triple, and *query vertices* coordinate query execution. Partially matched copies of queries are routed in parallel along different paths of this structure. The index graph is optimized for efficient routing of query descriptions to data and its vertices are addressable by an ID, which is a unique [ subject predicate object ] tuple.

We first describe how the graph is conceptually built and then explain the details of how this structure enables efficient parallel graph exploration.

[1] *RW-TR* is a significant redesign and extension of TripleRush [26], which was limited to parallelising basic-graph-pattern queries on a single machine.

[2] SIGNAL/COLLECT is similar to Pregel [19], GraphLab/PowerGraph [6], and Trinity [22].

[3] `http://ldbcouncil.org/benchmarks/snb`

[4] `http://uzh.github.io/signal-collect/`

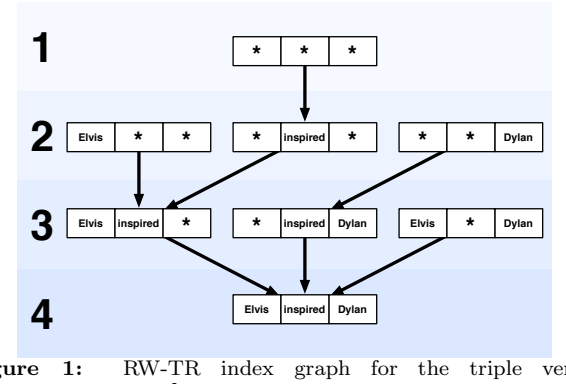



**Figure 1:** RW-TR index graph for the triple vertex [ Elvis inspired Dylan ].

## 3.1 Building the Index Graph

As mentioned before, RW-TR is a triple store with three types of SIGNAL/COLLECT vertices:

**Triple vertices** (level 4, Fig. 1) represent triples in the database. Each contains subject, predicate, and object information.

**Index vertices** (levels 1-3, Fig. 1) represent triple patterns and are responsible for routing partially matched copies of queries (referred to as *query particles*) towards triple vertices that match their respective patterns. They also contain subject, predicate, and object information, but one or several of them are wildcards.

**Query vertices** (Fig. 2) are added to the graph for each query that is being executed. A query vertex emits the first query particle that traverses the index structure. All query particles—successfully matched or not—get routed back to their respective query vertex and successful ones get reported as results. Once the query execution has finished, the query vertex removes itself from the graph.

The graph is built bottom-up, starting by creating a *triple vertex* for each RDF triple. These vertices are added to SIGNAL/COLLECT, which turns them into parallel processing units. A triple vertex will add its immediate *index vertices* (if they do not exist yet) and an edge from each of those vertices to itself. The construction process continues recursively for the index vertices until the parent vertex has already been added or the index vertex has no parent.

The index structure illustrated in Fig. 1 ensures that there is exactly one path from an index vertex to each triple vertex below it.

*Observations:* The number of predicates is usually much smaller than the number of distinct subjects or objects. Hence, storing edges from the root to [ * P * ] vertices requires the least amount of memory. The index graph we just described is different from traditional index structures, because it is designed for the efficient parallel routing of messages to triples corresponding to a given triple pattern. All vertices that form the index structure are active parallel processing elements that only interact via message passing.

## 3.2 Query Execution

We now look into how a query is executed, and then we follow with the description of the query optimizer.

Consider the subgraph shown in Fig. 2 and the query processing for the query: (unmatched = [ ?X inspired ?Y ], [ ?Y inspired ?Z ]; bindings = {}). The query execution starts by adding the query vertex to the TripleRush graph. After the query optimizer determines the execution order of the triple patterns, the query gets processed as follows:

1. The query vertex emits a single query particle, which is routed (by SIGNAL/COLLECT) to the index vertex that matches its first unmatched triple pattern. To determine when a query has finished processing, the initial query particle is endowed with a large number of tickets (Long.MaxValue). Should the tickets ever run out, new tickets could be acquired from the query vertex.[5]
2. When a query particle arrives at an index vertex, a copy of it is sent along each edge. The original particle evenly splits up its tickets among its copies.
3. Once a query particle reaches a triple vertex, the vertex attempts to match the next unmatched query pattern to its triple. If this succeeds, then a variable binding is created and the remaining triple patterns are updated with the new binding. The query particle gets sent to the index or triple vertex that matches its next unmatched triple pattern.
4. If all triple patterns are matched, then the query particle gets routed back to its query vertex.
5. If no vertex with a matching pattern is found, then a handler for undeliverable messages routes the failed query particle back to its query vertex.
6. Query execution finishes when the sum of tickets of all failed and successful query particles received by the query vertex equals the initial ticket endowment of the first particle that was sent out. The query vertex reports that all results have been delivered and removes itself from the graph.

*Observations:* Queries are often routed along downward edges in the index structure, and placing the index vertices in a way that achieves good locality means fewer messages are sent across machines. We found that the following scheme can achieve good locality, while at the same time ensuring a high degree of parallelism: If the subject of an index vertex is defined, then it is placed on a node determined by its subject. If the subject is a wildcard, then it is placed on a node determined by the object. If only the predicate is defined, then it is placed on a node determined by the predicate. The root index vertex is hardcoded to the last node. This scheme guarantees that particles are locally routed from [ S * * ] to [ S P * ] as well as from [ * * O ] to [ * P O ].

In addition, to assign a vertex to workers on a machine identified with the above assignment scheme, we compute the sum of its (encoded, see 3.5) IDs modulo the number of workers on the assigned node. In our tests, this scheme

[5]This feature is currently not supported by our system and was not necessary for any of our evaluations.

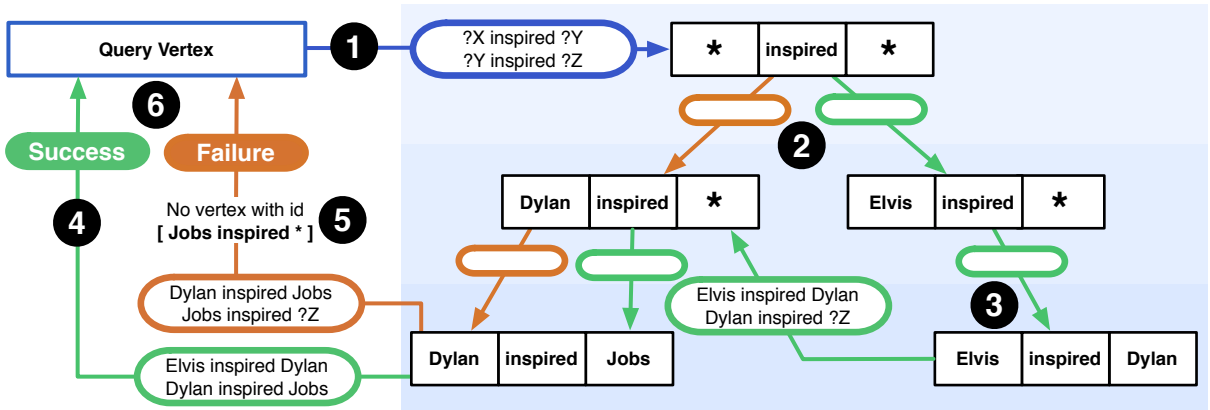



**Figure 2:** Query execution on the relevant part of the index that was created for the triples [ Elvis inspired Dylan ] and [ Dylan inspired Jobs ].

performed better than mixing the values with a collision-minimizing hash function. Signal/Collect uses the same mappings for vertex addressing and for routing messages to (potentially non-existent) index vertices.

## 3.3 Query Optimization

It has been highlighted [23] that the order in which patterns are explored affects the performance of query processing. When a triple pattern is matched at an index vertex, the bindings made in that vertex are forwarded to the index vertices responsible for subsequent triple patterns (unless there is no match). Since it is expensive to send particles, we explore the graph with a triple pattern ordering that considers both the number of particles to be sent and the branching factor for each particle when matching the next pattern. We estimate these costs by gathering several statistics on index vertices and predicates.

Statistics on the number of children are incrementally aggregated in the index vertices during graph loading. These statistics are cached and, if necessary, retrieved in parallel from the index vertices before optimization. We also compute predicate selectivity statistics after the loading is complete, by dispatching all required two-pattern queries [23] for all predicate combinations. In order to make this fast, we added special support and optimizations for queries that only compute the result count. When determining the number of bindings for the second pattern, the counts can be directly accessed in the index vertices. For the evaluated datasets, the predicate selectivity gathering is usually faster than the graph loading, but the number of dispatched queries is $O(\mid pred \mid^2)$, which could become a problem if a dataset contains many predicates. The optimizer also works, although not as well, when substituting missing selectivities with very large numbers (this was not necessary for any of the evaluated datasets).

In the remainder of this subsection we briefly introduce the cost model and discuss the optimization procedure.

### 3.3.1 Cost Model

We model a query $q$ as a sequence of triple patterns $p_i$, indexed by $i \geq 1$. For a triple pattern $p$, we denote the subject, predicate, and object by $p.sub$, $p.pred$, and $p.obj$, respectively.

The cost of executing a query can now be defined as the sum of the costs of matching individual triple patterns in a given order:

$$Cost(q) = \sum_{p_i \in q} cost(p_i) \quad (1)$$

The cost $cost(p_i)$ of matching the $i^{th}$ triple pattern depends on two factors. First, we have to consider the number of bindings or query particles created by the previous triple pattern, which we call $frontier(p_{i-1})$, in accordance with graph search algorithms. $frontier(p_{i-1})$ can be seen as a worst-case estimate of the number of particles that might reach this stage of the exploration. Second, we need to account for the exploration cost $explore(p_i)$ of the index vertex corresponding to the triple pattern $p_i$. This can be seen as a worst-case estimate of the branching factor encountered per frontier particle that matches triple pattern $p_i$. Consequently, for $i \geq 1$, we estimate the cost of matching a pattern as: $cost(p_i) = frontier(p_{i-1}) \times explore(p_i)$.

In order to define these two functions we need the statistics defined in Table 1. Given these statistics we can estimate $frontier(p_{i-1})$ as:

$$frontier(p_{i-1}) = \begin{cases} 1, \text{ if } i = 1 \\ card(p_i), \text{ if } i = 2 \\ min(explore(p_i), \min_{\forall j<i} selectivity(p_j, p_i)), \\ \quad \text{otherwise (for all available selectivities)} \end{cases}$$

and $explore(p_i)$ is expressed as:

$$explore(p_i) = min(card(p_i), branch(p_i)),$$

where $branch(p_i)$, the branching factor of the index element associated with $p_i$, is estimated as follows:

$$branch(p_i) = \begin{cases} card(p_i), \text{ if } i = 1 \\ 1, \text{ if } p_i.sub,\ p_i.pred,\ p_i.obj \text{ are bound} \\ maxObj(p_i), \text{ if } p_i.sub,\ p_i.pred \text{ are bound} \\ maxSub(p_i), \text{ if } p_i.pred,\ p_i.obj \text{ are bound} \\ \mid pred \mid, \text{ if } p_i.sub,\ p_i.obj \text{ are bound} \\ edges(p_i) \times maxObj(p_i), \text{ if } p_i.pred \text{ is bound} \\ card(p_i), \text{ otherwise} \end{cases}$$

| | |
|---|---|
| $card(p)$: | cardinality of triple pattern $p$, i.e., number of triples that can be reached following the vertex responsible for the triple pattern |
| $selectivity(p_i, p_j)$: | number of vertices that are connected by a predicate-pair ($p_i.pred$, $p_j.pred$), sharing a common subject/object (see [23]) |
| $edges(p)$: | number of outgoing edges from the [ * P * ] vertex corresponding to $p.pred$ to all its [ S P * ] vertices (Figure 1) |
| $maxObj(p)$: | the maximum number of objects of any [ S P * ] vertex corresponding to the predicate $p.pred$ |
| $maxSub(p)$: | the maximum number of subjects of any [ * P * ] vertex corresponding to the predicate $p.pred$ |
| $\|pred\|$: | number of distinct predicates |

**Table 1:** Statistics used in query optimization

### 3.3.2 Query Optimizer

The query optimizer uses uniform-cost search to find the plan with the best worst-case cost estimate. We employ a min-heap ordered by $Cost(q)$, which initially gets seeded with all possible 1-pattern plans. The optimizer then repeatedly removes the cheapest plan from the heap and computes all possible plan-extensions, which it then inserts into the min-heap.

To prevent the expansion of non-optimal (partial) plans, the optimizer maintains a map that uses the set of covered triple patterns as a key and the lowest cost ordering of the patterns as the value. Before expanding partial plans, the planner looks up their triple-pattern set in the map and only expands the partial plans that have no prior entries. Other plans are discarded as suboptimal.

If the optimizer finds a plan where $frontier(p_{i-1}) = 0$, then the optimizer reports that the query has no results and it is not executed. When the optimizer finds a plan that uses all patterns at the top of the heap, then it has found the cost-optimal plan to execute according to the model.

All operations on the planning heap and reference map take $O(log\ n)$ time, where $n$ is the number of elements in the heap/map. In the worst case, the planning heap can contain almost all incomplete plans, which is exponential in the number of patterns in a query: one can create a (partial) plan by picking or not picking each of the $\mid q \mid$ patterns, resulting in a heap size and number of insert operations of $O(2^{|q|})$. This means that the search space exploration time complexity is $O(log(n) * 2^{|q|})$.

In order to prevent this exponential increase of the planning time for queries with many patterns, we use a greedy query optimizer when the number of patterns in the query is greater than a fixed number.[6] The greedy optimizer is described in [26].

## 3.4 Extension to Random Walks

Random Walks with Restarts (RWR) are a popular graph sampling technique that can be used for various tasks from computing the similarity between two nodes in a graph to retrieving novel relations [15]. Random walk based models have been applied to many problems such as ranking webpages and segmenting images. Conceptually, random walks can be seen as starting from a given vertex and then following a random edge to a neighboring vertex. There, the walker moves again to a randomly chosen neighbor, goes back to the vertex from which the walk started, or stops its walk based on a restart rule. Example restart rules are (i) walking for a finite number of steps from the starting node, (ii) walking for any number of steps and restarting when there is no outgoing edge, or (iii) at each vertex, restart with a given probability.

In order to add support for efficient sampling queries based on RWR, we have to modify three elements of the previously described architecture: First, we extend the query particles with the extra structures required for sampling. Second, we modify the routing of sampling query particles to adhere to the rules of random walks. And third, to sample correctly we need to store additional bookkeeping information inside the second-level indices (SIndex, PIndex, OIndex). Next we describe each of these modifications in more detail.

To allow for sampling queries we *extended the query particles* with a flag that indicates if the particle is currently executing a traditional (SPARQL) part of a query or a sampling element. In addition, we extended the particles' data structure to optionally include information about the constraints of the random walk, such as the directionality (subject -> object, object -> subject, or both), any constraints on the path (e.g., if it should only follow certain properties or some specified sequence of property types), and the stopping condition. This approach allows us to combine SPARQL and sampling queries within the same execution.

A naïve routing approach would route as many particles through the index as there are tickets. Whenever a particle would arrive at a triple vertex, it would test for its random walk constraints and decide whether to stop the exploration or continue. This would lead to a high overhead, as the same path would be followed multiple times. To improve on this approach, we route as follows: each query begins with a certain number of tickets provided to it. At each index level the particle is split and sent along each index path that qualifies according to the random walks' constraints. Tickets are assigned to each particle such that the sampling of the graph is not biased by the index structure. If there are not enough tickets to assign to all particles, then we randomly choose some paths to follow and abandon the others. As a result, RW-TR computes as many random walks in parallel as there are tickets.

Once the query reaches a triple vertex, the stopping condition gets evaluated. In case it applies and the query constraints are met, then the variable bindings to the subject, predicate, and/or object stored in the current vertex are added to the particle and reported to the query vertex as a success. In case the stopping condition applies and the query constraints fail, then it is reported as a failure. Else, the exploration continues.

To assign the tickets proportionally, additional bookkeeping information is stored in the second-level indices. We need to store the total number of outgoing edges that can be traversed by following the child vertices, and the sum of both outgoing and incoming edges of all child vertices. This information is calculated during the data loading phase. To assign the tickets proportionally to the particles sent to the child vertices, we need to know the outgoing edges per child index vertex. Precomputing these would increase the index

[6] In our experiments, we fixed this number to 8.

size considerably. We, therefore, ask the child index vertices for their number of outgoing edges via a special signal and can dispatch the particles as soon as the information arrives, as we know the total number of outgoing edges.

As an example, consider the sampling query

SAMPLE ?X FROM [ Elvis inspired ?X]
CONSTRAINTS [maxhops = 3, tickets=10]

and the subgraph of Fig. 2. This is a neighborhood sampling query as it returns a sample of vertices reachable from Elvis by traversing edges labeled inspired for a maximum of three hops. Given that it uses 10 tickets, it uses 10 random walks along the inspired edges from the Elvis-vertex. We compute these random walks by starting at the vertex [ Elvis inspired *], which has only one outgoing edge, leading to [ Elvis inspired Dylan]. Given that Dylan is a correct answer to the random walk query, we would need to flip a coin to decide whether to return it as a correct answer or continue. As we are doing multiple random walks in parallel, it is efficient to do both. Hence, we assign half the tickets to the current answer and continue exploring with the other half (when we have an odd number of tickets we flip a coin to determine which path gets one more ticket). Hence, ⟨Dylan⟩, 5 is returned to the query vertex. Again, there's only one outgoing edge along which the remaining 5 tickets of the query are sent. At the vertex [ Dylan inspired Jobs], the binding ⟨Dylan, Jobs⟩, 3 is returned to the query vertex, indicating that the path to reach Jobs went via the Dylan vertex. This is the second hop of the query, and according to the constraints set to the query, the query can make one more hop. But since there are no vertices that can be traversed from here, we will also return the two remaining tickets to the query vertex. The final result of our neighborhood sampling query will be the following distribution of bindings: [ ⟨Dylan⟩, 5; ⟨Dylan, Jobs⟩, 5 ].

## 3.5 RW-TR Optimizations

Just like parallel TripleRush [26], RW-TR contains some initial optimizations: a) we do dictionary encoding, b) we remove the triple vertices and fold them into the third index level, where each index vertex stores a compact representation of all the triples that match their pattern, c) we only send the tickets of the failed particles back to the query vertex, and d) we use bulk-messaging and message-combiners.

In addition to this, RW-TR contains improvements that address previous limitations with regard to insert performance and memory usage during loading, by adopting a new data structure for the index vertices. Next, we discuss the details and motivation of these changes.

*Index Vertex Representation:* In Fig. 1, one notices that the ID of an index vertex varies only in one position—the subject, the predicate, or the object—from the IDs of its children. To reduce the size of the edge representations, we do not store the entire ID of child vertices, but only the specification of this position consisting of one dictionary encoded number per child. We refer to these numbers as *ID-refinements*. The same reasoning applies to third level index vertices, where the triples they store only vary in one position from the ID of the binding index vertex.

Routing and binding only require a traversal of all ID-refinements. To support traversal and inserts in a memory-efficient way, we store the refinements in a special-tailored Splay tree, where the key of each node is an interval and each

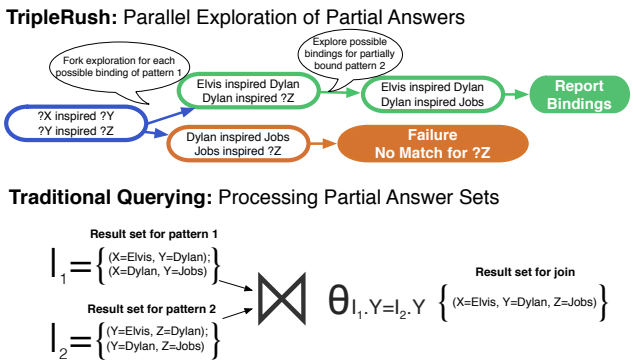



**Figure 3:** Comparison between query set processing and TripleRush parallel asynchronous partial answer exploration. Same query and data as in Fig. 2.

node stores the set of refinements contained in its interval. The data structure supports average case O(log(n)) inserts, low memory usage and fast traversal.

*Index Graph Structure:* Because we fold the triple vertices into the third index level, there is no longer an obvious place where one can verify if a fully bound pattern corresponds to a triple that exists inside the store.

To deal with this, RW-TR sends the particle that has to check for the existence of a fully-bound pattern to the corresponding [ S * O ] index vertex. These vertices do not store the ID-refinements into a Splay-tree, but into a sorted array. The existence is checked with binary search.We observe that most patterns have bound predicates, so these vertices are rarely used for anything but to check for the existence of a triple. We also observe that inserts into the array are O(n). In practice, this was not an issue, since there are usually very few predicates for a given subject/object pair.

# 4. PRELIMINARY ANALYSIS

As introduced in the last section and illustrated in Fig. 3, RW-TR processes SPARQL queries by exploring each partial binding asynchronously in parallel. Whenever an exploration encounters more than one possible partial binding, it forks the exploration and pursues both potential solutions in parallel ('green' and 'orange' explorations in the figure). When all variables of an exploration are bound, it returns a result ('green' path). Alternatively, when the remaining unbound variables of an exploration cannot be bound, it aborts that path ('orange' path). Essentially, RW-TR performs a parallel-asynchronous graph search.

Traditional DBMS use operators on indices and intermediate data structures (typically arrays or sets). Originally, these operators were executed synchronously, where each operator is executed until its full result set is available before the next operator is called (see also Figure 3). Parallelism is usually introduced by (i) executing independent operators in parallel (such as the scans that create the sets in the figure) and (ii) implementing parallelized operators resulting in a parallel but synchronous system (each operator has to find all its results before invoking the next one). Modern systems introduce additional parallelism via pipelining operators [8], which allow some operators to pass on partial results. Conceptually, these systems use parallelized approaches to process partial answer sets rather than exploring all possible partial solutions in parallel.

The central proposition of this paper is that RW-TR's parallel-asynchronous exploration approach may be a viable alternative graph store architecture for today's multi-core systems.

First, we believe that asynchronous-parallel query processing allows to exploit the many cores better than synchronous-parallel execution, as cores are less likely to wait for work during synchronization. RW-TR is built to exploit this asynchronicity. As mentioned, some current systems exploit a kind of asynchronicity via pipelining. Pipelining, however, comes at the cost of more complexity in both the operators and their coordination. Given that RW-TR does not require coordinating between its explorations, it does not incur such an overhead.

Second, we expect the exploitable performance improvement due to parallelism to be curbed by (i) the branching factor of the query, which is a function of the selectivity of the triple patterns and connectivity of the involved nodes (or join selectivity), as it limits the degree of parallelism and (ii) possible gains through locality, as forking explorations and moving them to other cores (possibly on other machines) can be costly operations.

RW-TR's index can be conceptualized as a vertical partition of the data into S-Index, P-Index, and O-Index, for subjects, predicates and objects, respectively, as well as three additional indices for each combination of two columns – SP-Index, PO-Index and SO-Index. In addition, the latter three indices are sharded by the subject key (for SP-Index and SO-Index) or object key (for PO-Index). Each shard is assigned to a processing unit in a distributed compute cluster.

# 5. EVALUATION

The goal of the evaluation was to explore the propositions that that RW-TR's parallel-asynchronous exploration approach is both competitive and scalable via the efficient exploitation of parallelism where possible, as well as to illustrate its capability to gain useful results via RWR. To that end we employ two standard benchmarks—LUBM and BSBM—and evaluate RW-TR's performance under different conditions, as well as a use case for RWR.

The experiments reported in subsection 5.3 and the distributed evaluations in subsection 5.4 were run on a cluster of 8 machines, each machine having 128 GB RAM and two E5-2680 v2 processors at 2.80GHz, with 10 cores per processor. The machines are connected with 40Gbps Infiniband. We used version 1.8.0_05-b13 of the Java Runtime. All other experiments were run on single machines of the same cluster.

We used both the LUBM[7] (Lehigh University Benchmark) and BSBM (Berlin SPARQL) [2] benchmarks. For LUBM, we used the queries used in the Trinity.RDF evaluation [30]. For BSBM, we generated the datasets and explore use case queries with the standard data generator and query test driver, but stripped the queries of advanced SPARQL features unsupported by RW-TR such as OPTIONAL or complex filters, and discarded queries 9 and 12 for relying on such features.

We executed ten runs of each LUBM query and in the diagrams report both the average and geometric mean over the fastest runs. For BSBM we executed the same ten generated queries from each category, computed the category average and reported the average and geometric mean over all categories. The measured total time for a run includes everything from query optimization until the result set is fully traversed, but the decoding of the results is not forced.

[7]http://swat.cse.lehigh.edu/projects/lubm

## 5.1 Vertical Scalability and First Results

The goal of this evaluation was to measure how well RW-TR scales with additional worker threads on a single machine of the cluster. Additionally, the first result is reported, in order to test our hypothesis that the fully asynchronous execution allows to deliver the first result much faster than the full result set. We ran this evaluation ten times on the LUBM 160 dataset with the Trinity.RDF queries and varied the number of worker threads between 1 and 20, because the hardware has 20 physical cores. We pre-planned the queries and ran them without the optimizer, in order to reduce overhead that is not directly associated with the execution engine.

In Figure 4 we see that adding more workers has a negative and at best neutral impact for queries L4, L5, and L6, which touch very little data and are answered in at most a millisecond. For queries L1, L3, and L7, which are more processing intense, the speedup for 20 workers relative to 1 worker is between 10 and 12, which is good, considering that query dispatch and result reporting is still handled by only one worker, and that all queries are answered in under 50ms at that point. Query 2 scales a bit up to 10 worker threads, but does not improve with more processing elements. This is likely due to its structure of only 2 triple-patterns, which offers RW-TR less potential for parallelization.

Figure 4(c) graphs time until the first result was reported relative to the total query execution time (Query 3 was omitted as it does not return results). For queries that profited from parallelization, the first answer was delivered in around a third of the time it took to compute the entire result. The relative benefit increased when going from 1 to 10 worker threads, but then remained approximately constant when going to 20 processing threads.

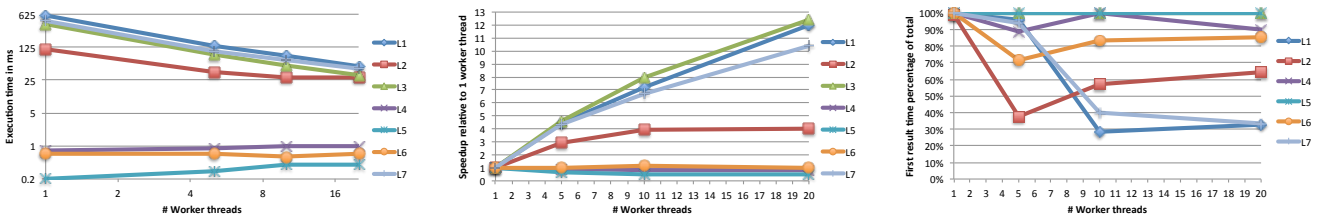
Overall, this evaluation shows that the architecture can take advantage of multicore architectures and that if there are enough workers available, then, for some queries, the asynchronous-parallel execution can deliver first results much sooner than the full results.

## 5.2 Data Scalability and Memory Usage

To measure the data scalability of RW-TR in the single-machine setup, we measured its performance for different sizes of the benchmark datasets. For comparison, we also supply the numbers for the in-memory backend of Sesame, as it is open-source and runs in the JVM, and for Virtuoso 7.1 as a comparison to on-disk approaches.

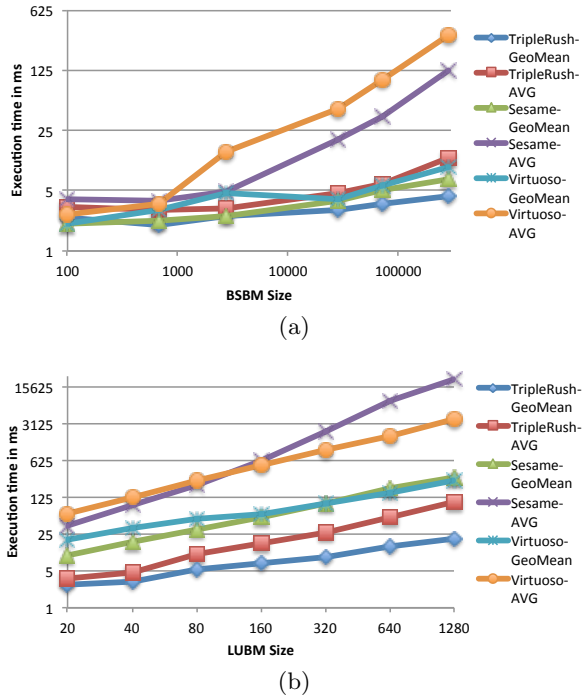
To make the comparison with the on-disk system Virtuoso fairer, we evaluated warm-cache runs and we configured it to make use of the processors and memory of the machine.

The two diagrams in Figure 5 show how the performance changes, when the LUBM and BSBM queries are executed on increasingly large datasets. On the BSBM dataset, the performance of all systems is comparable for small dataset sizes, but RW-TR scales better to large dataset sizes, for the largest BSBM dataset it is on average up to 10 times faster than Sesame and up to 25 times faster than Virtuoso. The geometric mean does not change dramatically, because most queries do not touch more data on a larger dataset.




(a) (b) (c)

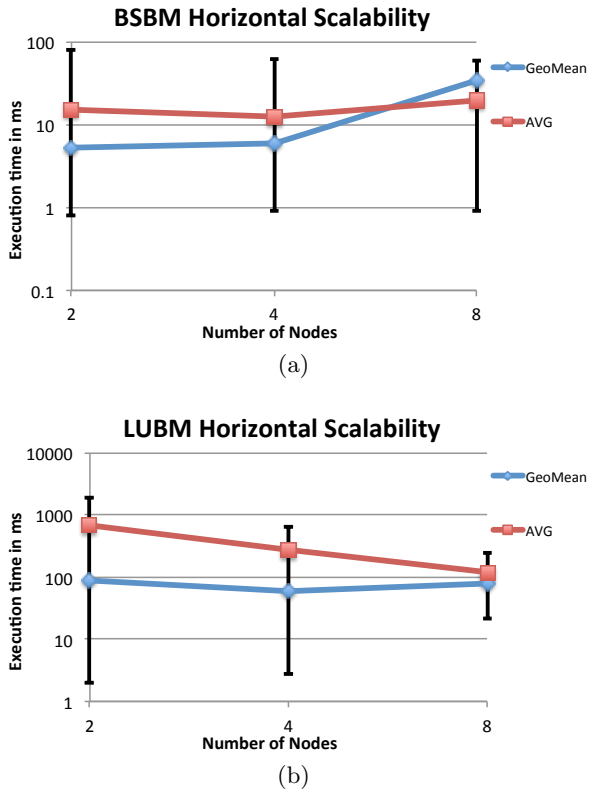
**Figure 4:** 4(a) shows the execution times of the different queries on a logarithmic scale on both axes, 4(b) shows the speedup relative to 1 worker thread for all queries, and 4(c) shows the time it took until the first result as a percentage of the total for the entire result.




(a)

(b)

**Figure 5:** 5(a) and 5(b) compare the single-node scalability of execution times with increasing BSBM and LUBM sizes. Both axes are logarithmic.




(a)

(b)

**Figure 6:** 6(a) and 6(b) compare the horizontal scalability of RW-TR with 2, 4, and 8 nodes for both BSBM 284'826 and for LUBM 1280. The aggregates are over all queries for that dataset, and for each query we used the fastest of 10 runs. Error bars indicate the runtimes for the fastest and slowest queries in the benchmark.

On the more processing intense LUBM queries, RW-TR shows better performance on any dataset size, up to more than 200 times faster for Sesame and 35 times faster for Virtuoso on average for the largest evaluated size.

We do not have any precise memory measurements, but we measured the used JVM memory, which can serve as an upper bound for the memory used by the index. We then look at the lowest such upper bound that we measured during any of the runs. For BSBM 284'826, RW-TR had a lowest upper bound of 39.7 GB, in contrast to 28.6 GB for Sesame. For LUBM 1280 RW-TR used 61.6 GB compared to the 34.7 GB used by Sesame. From this we conclude that the RW-TR index most likely uses more memory than Sesame's, but that the index size is still reasonable.

## 5.3 Horizontal Scalability

The goal of this evaluation was to measure RW-TR's scalability in the distributed setting. In particular, we wanted to explore if RW-TR's query evaluation approach would degrade when faced with messaging over the network rather than in-memory, or if the benefit of additional processors would dominate. For this, we measured the performance on large BSBM and LUBM data sets while varying the number of nodes used.

Figure 6 shows the results of these evaluations. We aggregated over the fastest runs of ten executions for each query, in order to reduce confounding factors (e.g. garbage collections). We found that for the BSBM dataset/queries the average execution time stays approximately the same, while the geometric mean slightly increases. For the LUBM dataset/queries the geometric mean stays approximately the same, whilst the average execution time decreases. Our interpretation is that for queries that do not require a lot of processing the added overhead and network latency re-

duces the performance, whilst for queries that require a lot of processing the benefit of the added processing elements can overcome this drawback. This explains why adding nodes tends to slow down the execution of the fastest millisecond-range queries, whilst improving the performance for the most processing-intense queries.

## 5.4 Comparison with Trinity.RDF and TriAD

Tables 2 and 3 compare the performance of RW-TR to the numbers reported in the Trinity.RDF [30] and TriAD [8] papers. We followed the evaluation procedure described to us by the Trinity.RDF authors, which includes a partitioning of rdf:type into a different type predicate for each class referred to as the object. The RW-TR distributed evaluation on LUBM 10240 with 1.36 billion triples was run on all 8 nodes of the cluster. The comparison of the numbers in these tables has many caveats, as the cited numbers were created with different hardware and cluster sizes and the approaches require different amounts of preprocessing. We believe this comparison at least shows that the RW-TR architecture is competitive in both the single-node and in the distributed scenario.

## 5.5 A Random Walk Use Case: Path Sampling

In this section we illustrate RW-TR's capability to run sampling queries based on random walks with restarts (RWR). Our goal is to show the simplicity with which Semantic Web developers using RW-TR can attain RWR results and how they can be combined with SPARQL queries. A more general discussion of the usefulness of RWRs is beyond the scope of our use case and can be found in [15].

Consider the need for establishing the relatedness of two entities. As an example, we could have a selection of sports teams—Liverpool, Manchester United, Chicago Bulls, and The Brooklyn Dodgers—and we would like to know what championship they compete in—the UEFA Cup, the World Series, or the NBA Finals—from a dataset of relationships extracted from a large text corpus. The dataset can be extremely noisy and may have misleading and/or conflicting relationships, such as the Manchester United team playing basketball. Indeed, for example, an October 2011 The Telegraph article connects a basketball player with Manchester United.[8]

RWR have been proposed to deal with these kinds of noisy settings. The rationale is the following: Intuitively, there are more short paths between nodes that are conceptually close to each other than between nodes that are further apart. Just picking the shortest path between two vertices may be mislead by a noisy connection. A sampling of random walks will unearth which vertices are closer via many connections and is, hence, less susceptible to false relations in the graph.

For our use case we use the Never Ending Language Learning (NELL) knowledge base (version 08m.845), which has about 2 million triples. NELL contains relations extracted from natural language text and it iteratively learns new relations based on what it learned in the previous iterations. This naturally leads to some ambiguous or false relations in the knowledge base. For example, NELL contains these two relations about Manchester United (the British soccer club), the first of which is clearly noise: ⟨sportsteam:man_united team-plays-sport sport:basketball⟩ and ⟨sportsteam:man_united team-plays-in-league sportsleague:fa⟩, where sportsleague:fa is the British Football Association cup.

To illustrate the capability of RW-TR to solve the task of finding the connectedness of a team with a championship, we ran queries of the following form:[9]

```
SAMPLE ?X FROM
  [ sportsteam:man_united ?X sportsleague:uefa]
CONSTRAINTS [maxhops = 5, tickets=100]
```

whilst varying the team, the championship, the maximum number of hops (we employed 5, 10, 20), and the number of tickets (we used 100, 1'000, and 10'000). By simulating multiple independent random walks from teams, we can count all walks that reach the respective championships and estimate or calculate information such as the number of walks reaching the goal, the path lengths, or the conditional probabilities of reaching the championships from a given team.

Figure 7 graphs some of the results. In the first graph on the left we show the distributions of reaching the UEFA Cup from all four teams for 1'000 tickets whilst varying the path length. As we can see, the distribution is extremely stable. Both Manchester United and Liverpool are clearly associated with the UEFA cup, whilst the two non soccer teams do not show any relations. This illustrates the small world phenomenon, where most entities that are related are close to each other in the graph and the longer paths actually do not lead to many additional relationships. This latter observation is supported by the actual number of arriving tickets in each of the classes.

The three graphs on the right of Figure 7 show the distributions of reaching each of the championships from the four teams. Given the stability of the results, we chose a path length of 5 and varied the number of walks (i.e., tickets initially assigned). The paths to the UEFA Cup and the World Series are very stable. Indeed, the number of arriving tickets (printed in the bars) are proportional to the number of initial tickets (or walks). The NBA graph on the far left tells a more subtle story. The more tickets we assign to the exploration, the more the result reflects the noisy extractions mentioned above. When using 1'000 or 10'000 tickets, we find a small number of connections between both Liverpool and Manchester United and the NBA finals — reflecting noisy connections. Hence, a higher number of tickets is more likely to follow noisy connections. Note that the total execution time for the thirty-six path sampling queries needed for the three graphs on the right of Figure 7 was less than a second.

# 6. LIMITATIONS AND CONCLUSION

In the following we discuss limitations and threats to validity, followed by the conclusion.

There are some limitations related to the RW-TR implementation being a prototype: (i) Encoded IDs cannot exceed $2^{31}$, (ii) only a subset of SPARQL is supported, (iii) dictionary encoding/decoding is not distributed, and (iv) splay integer sets do not currently support deletions. These limi-

[8] http://www.telegraph.co.uk/sport/football/teams/liverpool/8826912/Liverpool-v-Manchester-United-basketball-star-and-Anfield-stakeholder-Lebron-James-jets-in-for-clash.html

[9] Note that this query almost uses the SPARQL SELECT syntax. We could support additional basic graph patterns in a WHERE clause employing the same semantics as in a SELECT statement.

| *Fastest of 10 runs* | L1 | L2 | L3 | L4 | L5 | L6 | L7 | Geo. mean |
|---|---|---|---|---|---|---|---|---|
| TripleRush | 22.6 | 27.8 | 0.4 | 1 | 0.4 | 0.9 | 21.2 | 2.94 |
| Trinity.RDF | 281 | 132 | 110 | 5 | 4 | 9 | 630 | 46 |
| TriAD | 427 | 117 | 210 | 2 | 0.5 | 19 | 693 | 39 |
| TriAD-SG | 97 | 140 | 31 | 1 | 0.2 | 1.8 | 711 | 14 |

**Table 2:** Single-node, LUBM 160 (∼21 million triples), time in ms. Comparison data from [30] and [8].

| *Fastest of 10 runs* | L1 | L2 | L3 | L4 | L5 | L6 | L7 | Geo. mean |
|---|---|---|---|---|---|---|---|---|
| TripleRush | 3,111.2 | 1,457.9 | 0.7 | 3.5 | 9.5 | 29.1 | 1,165.8 | 62.1 |
| Trinity.RDF | 12,648 | 6,018 | 8,735 | 5 | 4 | 9 | 31,214 | 450 |
| TriAD | 7,631 | 1,663 | 4,290 | 2.1 | 0.5 | 69 | 14,895 | 249 |
| TriAD-SG | 2,146 | 2,025 | 1,647 | 1.3 | 0.7 | 1.4 | 16,863 | 106 |

**Table 3:** Distributed, LUBM 10240 (∼1.36 billion triples), time in ms. Comparison data from [30] and [8].

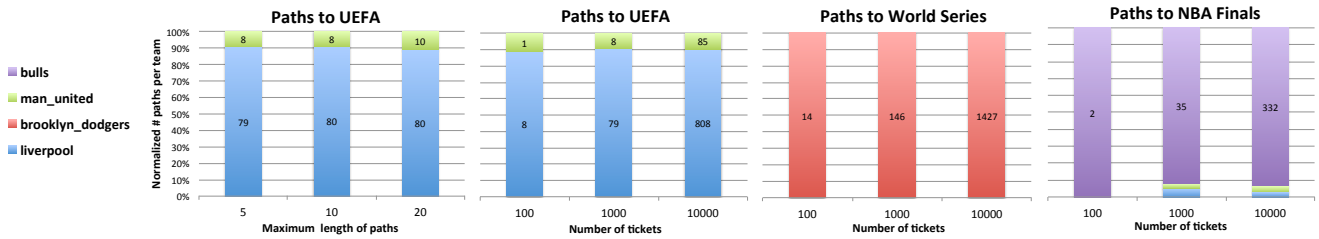



**Figure 7:** Distributions resulting from randomly walking from all four teams to a given championship. The leftmost figure graphs walks to UEFA whilst varying the maximum path-length when employing 1'000 tickets. The other graphs vary the number of tickets whilst fixing the path-length to 5.

tations are not inherent to the approach and resolving them is primarily a matter of engineering.

There are, however, limitations that are inherent to the approach: First, some operations, such as ordering the results, by definition require a synchronization. Our current approach can only handle them as post-processing steps, which is straightforward but inefficient. Second, an efficient execution of filters requires for an optimizer to be able to place them at any point during the query execution plan. Our current approach is limited to treating them as a post-processing step. More efficient handling would need to enable access to literals from inside the store.

Also, our evaluations have some limitations: Our benchmarking of RW-TR is limited to synthetic datasets, which means that the results might not generalize to real-world datasets. Furthermore, as mentioned in Section 5.4, we could not compare RW-TR's performance with Trinity.RDF and TriAD running on the same hardware, as those two software packages are not available. Nonetheless, we believe that our evaluation shows that RW-TR is competitive in those system's core strength—the evaluation of SPARQL queries—whilst also supporting sampling queries.

Our query optimizer can be improved to better deal with queries that have many patterns: typical SPARQL queries contain star-shaped patterns that can be optimized independently of others [7].

Finally, our RWR use-case is only one example and lacks a full efficiency evaluation. Also, the current version of RW-TR only supports sampling using RWR and no other analytics such as PageRank. The rationale for this limitation was that the main goal of this paper was to illustrate the versatility of our approach in supporting both SPARQL querying and RWR-style sampling. A full efficiency evaluation of RW-TR's RWR capability or an extension to other graph analytics, which would be supported by the underlying SIGNAL/COLLECT framework, is beyond the scope of this paper.

# 7. CONCLUSIONS

In this paper we proposed to exploit the large number of CPU-cores of modern servers via the parallel exploration of partial bindings implemented on a distributed graph processing system. In particular, we suggested to fork the exploration whenever more than one binding is possible, returning the result when all variables of an exploration are bound, and expiring the exploration when it reaches a dead end. This re-conceptualization of triple-stores has the side-effect that it can efficiently support random walks with restarts by tasking each parallel exploration to simultaneously explore as many random walks as it has tickets.

As such, RW-TR presents a new approach to building graph stores with integrated graph analytic operators such as sampling queries. Our evaluation shows that this architecture can serve as the basis for a graph store that is competitive with other systems. We hope that RW-TR can serve as a basis for further exploration that will help Semantic Web developers to efficiently and seamlessly analyze their graphs.

*Acknowledgments.*

We would like to thank the Hasler Foundation for the generous support of the SIGNAL/COLLECT project under grant number 11072 and Alex Averbuch, Cosmin Basca, Lorenz Fischer, Shen Gao, Tobias Grubenmann, and Katerina Papaioannou for their feedback.

# 8. REFERENCES

[1] D. Abadi, A. Marcus, S. Madden, and K. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proceedings of the 33rd international conference on Very large data bases*, pages 411–422, 2007.

[2] C. Bizer and A. Schultz. The berlin sparql benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2):1–24, 2009.

[3] O. Corby, R. Dieng-kuntz, and C. Faron-zucker. Querying the semantic web with the corese search engine. pages 705–709. IOS Press, 2004.

[4] O. Corby, R. Dieng-Kuntz, C. Faron-Zucker, and F. Gandon. Ontology-based Approximate Query Processing for Searching the Semantic Web with Corese. Research Report RR-5621, 2006.

[5] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[6] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, 2012.

[7] A. Gubichev and T. Neumann. Exploiting the query structure for efficient join ordering in sparql queries.

[8] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. Triad: a distributed shared-nothing rdf engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 289–300, 2014.

[9] J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *Proceedings of the VLDB Endowment*, 4(11):1123–1134, 2011.

[10] C. Kiefer, A. Bernstein, and A. Locher. Adding data mining support to sparql via statistical relational learning methods. In *Proceedings of the 5th European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC'08, pages 478–492, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] C. Kiefer, A. Bernstein, and A. Locher. Adding Data Mining Support to SPARQL via Statistical Relational Learning Methods. In *Proceedings of the 5th European Semantic Web Conference (ESWC)*, Lecture Notes in Computer Science. Springer, 2008.

[12] C. Kiefer, A. Bernstein, and M. Stocker. The Fundamentals of iSPARQL: A Virtual Triple Approach for Similarity-Based Semantic Web Tasks. In *Proceedings of the 6th International Semantic Web Conference*, 2007.

[13] N. Kohout, S. Choi, D. Kim, and D. Yeung. Multi-chain prefetching: Effective exploitation of inter-chain memory parallelism for pointer-chasing codes. In *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, pages 268–279. IEEE, 2001.

[14] S. Kotoulas, J. Urbani, P. A. Boncz, and P. Mika. Robust runtime optimization and skew-resistant execution of analytical sparql queries on pig. In *International Semantic Web Conference (1)*, volume LNCS 7649, pages 247–262, 2012.

[15] N. Lao and W. W. Cohen. Relational retrieval using a combination of path-constrained random walks. *Mach. Learn.*, 81(1):53–67, Oct. 2010.

[16] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.

[17] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pages 40–51. IEEE, 2001.

[18] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *ACM SIGOPS Operating Systems Review*, volume 30, pages 222–233. ACM, 1996.

[19] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[20] T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 627–640, 2009.

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

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

[23] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th international conference on World Wide Web*, pages 595–604. ACM, 2008.

[24] P. Stutz, A. Bernstein, and W. W. Cohen. Signal/Collect: Graph Algorithms for the (Semantic) Web. In *International Semantic Web Conference*, volume LNCS 6496, pages pp. 764–780. Springer, Heidelberg, 2010.

[25] P. Stutz, D. Strebel, and A. Bernstein. Signal/collect: Processing web-scale graphs in seconds. *The Semantic Web Journal – Interoperability, Usability, Applicability*, Forthcoming.

[26] P. Stutz, M. Verman, L. Fischer, and A. Bernstein. Triplerush: A fast and scalable triple store. In *9th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, volume 50, 2013.

[27] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[28] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.

[29] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 517–528. ACM, 2012.

[30] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. *Proceedings of the VLDB Endowment*, 6(4):265–276, 2013.