TripleRush: A Fast and Scalable Triple Store

Philip Stutz, Mihaela Verman, Lorenz Fischer, and Abraham Bernstein

DDIS, Department of Informatics, University of Zurich, Zurich, Switzerland
{stutz, verman, lfischer, bernstein}@ifi.uzh.ch

Abstract. TripleRush is a parallel in-memory triple store designed to address the need for efficient graph stores that quickly answer queries over large-scale graph data. To that end it leverages a novel, graph-based architecture.
Specifically, TripleRush is built on our parallel and distributed graph processing framework Signal/Collect. The index structure is represented as a graph where each index vertex corresponds to a triple pattern. Partially matched copies of a query are routed in parallel along different paths of this index structure.
We show experimentally that TripleRush takes less than a third of the time to answer queries compared to the fastest of three state-of-the-art triple stores, when measuring time as the geometric mean of all queries for two benchmarks. On individual queries, TripleRush is up to three orders of magnitude faster than other triple stores.

1 Introduction

Many applications such as social network analysis, monitoring of financial transactions or analysis of web pages and their links require large-scale graph computation. To address this need, many have researched the development of efficient triple stores [1, 14, 11]. 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. This avenue has had great success both in research [1, 14, 11] and practice.[1] However, many of these systems are built like a centralized database, raising the question of scalability and parallelism within query execution. One way to increase the efficiency of parallel pipelined joins in such centralized databases is the use of sideways information passing [10].

Other approaches focus on adapting triple stores to a distributed setting: MapReduce [3] has been used to aggregate results from multiple single-node RDF stores in order to support distributed query processing [5].

Others have mapped SPARQL query execution pipelines to chains of MapReduce jobs (e.g., [6]). Whilst this provides scalability, the authors point out that the rigid structure of MapReduce and the high latencies when starting new jobs constrain the possibilities for optimizations [6]. Distributed graph processing frameworks such as Pregel [8], GraphLab/Powergraph [7, 4], Trinity [12], and

[1] http://virtuoso.openlinksw.com, http://www.ontotext.com/owlim

our own SIGNAL/COLLECT [13] can offer more flexibility for scalable querying of graphs.

Among the existing triple stores we only know of Trinity.RDF [15] to be implemented on top of such an abstraction. Trinity.RDF is a graph engine for SPARQL queries that was built on the Trinity distributed graph processing system. To answer queries, Trinity.RDF represents the graph with adjacency lists and combines traditional query processing with graph exploration.

In this paper we introduce TripleRush, a triple store which is based on an *index graph*, where a basic graph pattern SPARQL query is answered by routing partially matched query copies through this index graph. Whilst traditional stores pipe data through query processing operators, TripleRush routes query descriptions to data. For this reason, TripleRush does not use any joins in the traditional sense but searches the index graph in parallel. We implemented TripleRush on top of SIGNAL/COLLECT, a scalable, distributed, parallel and vertex-centric graph processing framework [13].

The contributions of this paper are the following: First, we present the TripleRush architecture, with an index graph consisting of many active processing elements. Each of these elements contains a part of the processing logic as well as a part of the index. The result is a highly parallel triple store based on graph-exploration. Second, as a proof of concept, we implemented the TripleRush architecture within our graph processing framework SIGNAL/COLLECT, benefiting from transparent parallel scheduling, efficient messaging between the active elements, and the capability to modify a graph during processing. Third, we evaluated our implementation and compared it with three other state-of-the-art in-memory triple stores using two benchmarks based on the LUBM and DBPSB datasets. We showed experimentally that TripleRush outperforms the other triple stores by a factor ranging from 3.7 to 103 times in the geometric mean of all queries. Fourth, we evaluated and showed data scalability for the LUBM benchmark. Fifth, we measured memory usage for TripleRush, which is comparable to that of traditional approaches. Last, we open sourced our implementation.[2]

In the next section we discuss the infrastructural underpinnings of TripleRush. This is followed by a description of the TripleRush architecture, as well as the functionality and interactions of its building blocks. We continue with a description of the optimizations that reduce memory usage and increase performance. We evaluate our implementation, discuss some of this paper's findings as well as limitations, and finish with some closing remarks.

2 Signal/Collect

In this section we describe the scalable graph processing system SIGNAL/COLLECT and some of the features that make it a suitable foundation for TripleRush.

[2] Apache 2.0 licensed, `https://github.com/uzh/triplerush`

SIGNAL/COLLECT [13][3] is a parallel and distributed large-scale graph processing system written in Scala. Akin to Pregel [8], it allows to specify graph computations in terms of vertex centric methods that describe aggregation of received messages (collecting) and propagation of new messages along edges (signalling). The model is suitable for expressing data-flow algorithms, with vertices as processing stages and edges that determine message propagation. In contrast to Pregel and other systems, SIGNAL/COLLECT supports different vertex types for different processing tasks. Another key feature, also present in Pregel, is that the graph structure can be changed during the computation. The framework transparently parallelizes and distributes the processing of data-flow algorithms. SIGNAL/COLLECT also supports features such as bulk-messaging and Pregel-like message combiners to increase the message-passing efficiency.

Most graph processing systems work according to the bulk-synchronous parallel model. In such a system, all components act in lock-step and the slowest part determines the overall progress rate. In a query processing use-case, this means that one expensive partial computation would slow down all the other ones that are executed in the same step, which leads to increased latency. SIGNAL/COLLECT supports asynchronous scheduling, where different partial computations progress at their own pace, without a central bottleneck. The system is based on message-passing, which means that no expensive resource locking is required. These two features are essential for low-latency query processing.

With regard to the representation of edges, the framework is flexible. A vertex can send messages to any other vertex: Edges can either be represented explicitly or messages may contain vertex identifiers from which virtual edges are created. TripleRush uses this feature to route query messages.

3 TripleRush

The core idea of TripleRush 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, therefore, 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 built and then explain the details of how this structure enables efficient parallel graph exploration.

3.1 Building the Graph

The TripleRush architecture is based on three different types of vertices. *Index and triple vertices* form the *index graph*. In addition, the TripleRush *graph* contains a *query vertex* for every query that is currently being executed. Fig. 1 shows the index graph that is created for the triple [Elvis inspired Dylan]:

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

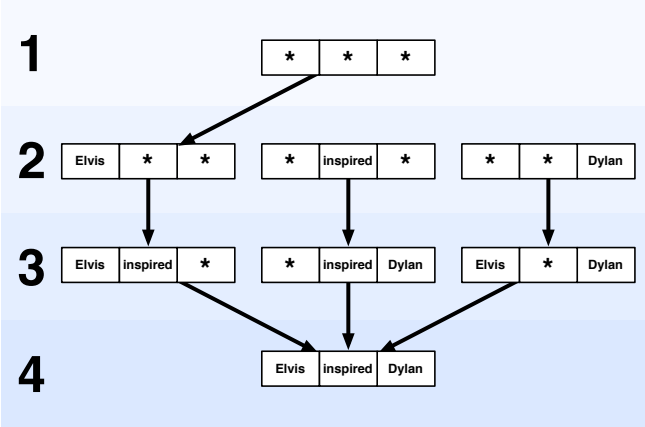

Fig. 1. TripleRush index graph that is created for the triple vertex [Elvis inspired Dylan].

Triple vertices are illustrated on level 4 of Fig. 1 and represent triples in the database. Each contains subject, predicate, and object information.

Index vertices, illustrated in levels 1 to 3 in 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 the pattern of the index vertex. Index vertices also contain subject, predicate, and object information, but one or several of them are wildcards. For example, in Fig. 1 the index vertex [* inspired *] (in the middle of the figure on level 2) routes to the index vertex [* inspired Dylan], which in turn routes to the triple vertex [Elvis inspired Dylan].

Query vertices, depicted in the example in Fig. 2, are query dependent. For each query that is being executed, a query vertex is added to the graph. The query vertex emits the first query particle that traverses the index graph. All query particles—successfully matched or not—eventually get routed back to their respective query vertex. Once all query particles have succeeded or failed the query vertex reports the results and removes itself from the graph.

The *index graph* is built by adding a *triple vertex* for each RDF triple that is added to TripleRush. These vertices are added to Signal/Collect, which turns them into parallel processing units. Upon initialization, a triple vertex will add its three parent *index vertices* (on level 3) to the graph and add an edge from these index vertices to itself. Should any parent index vertex already exist, then only the edge is added from this existing vertex.

When an *index vertex* is initialized, it adds its parent index vertex, as well as an edge from this parent index vertex to itself. Note that the parent index vertex always has one more wildcard than its child. The construction process continues recursively until the parent vertex has already been added or the index vertex has no parent. In order to ensure that there is exactly one path from an index

vertex to all triple vertices below it, an index vertex adds an edge from at most one parent index vertex, always according to the structure illustrated in Fig. 1.

Next we describe how the index graph allows parallel graph exploration in order to match SPARQL queries.

3.2 Query Processing

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 that correspond to a given triple pattern. All vertices that form the index structure are active parallel processing elements that only interact via message passing.

A query is defined by a list of SPARQL triple patterns. Each query execution starts by adding a query vertex to the TripleRush graph. Upon initialization, a *query vertex* emits a single query particle. A query particle consists of the list of unmatched triple patterns, the ID of its query vertex, a list of variable bindings, a number of tickets, and a flag that indicates if the query execution was able to explore all matching patterns in the index graph. Next, we describe how the parts of the query particle are modified and used during query execution.

The emitted particle is routed (by SIGNAL/COLLECT) to the index vertex that matches its first unmatched triple pattern. If that pattern is, for example, [Elvis inspired ?person], where ?person is a variable, then it will be sent to the index vertex with ID [Elvis inspired *]. This index vertex then sends copies of the query particle to all its child vertices.

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. If all triple patterns are matched or a match failed,[4] then the query particle gets routed back to its query vertex. Otherwise, the query particle gets sent to the index or triple vertex that matches its next unmatched triple pattern.

If no index vertex with a matching ID is found, then a handler for undeliverable messages routes the failed query particle back to its query vertex. So no matter if a query particle found a successful binding for all variables or if it failed, it ends up being sent back to its query vertex.

In order to keep track of query execution and determine when a query has finished processing, each query particle is endowed with a number of tickets. The first query particle starts out with a very large number of tickets.[5]

When a query particle arrives at an index vertex, a copy of the particle is sent along each edge. The original particle evenly splits up its tickets among its copies. If there is a remainder, then some particles get an extra ticket. If a particle does not have at least one ticket per copy, then copies only get sent

[4] A match fails if it creates conflicting bindings: Pattern [?X inspired ?X] fails to bind to the triple [Elvis inspired Dylan], because the variable ?X cannot be bound to both Elvis and Dylan.

[5] We use `Long.MaxValue`, which has been enough for a complete exploration of all queries on all datasets that we have experimented with so far.

along edges for which at least one ticket was assigned, and those particles get flagged to inform the query vertex that not all matching paths in the graph were explored. 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.

Once query execution has finished, the query vertex reports the result that consists of the variable bindings of the successful query particles, and then removes itself from the graph.

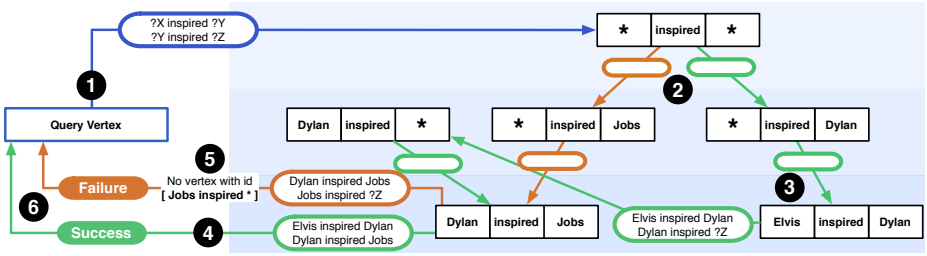

Fig. 2. Query execution on the relevant part of the index that was created for the triples [Elvis inspired Dylan] and [Dylan inspired Jobs].

As an example illustrating a full query execution, consider the relevant subgraph created for the triples [Elvis inspired Dylan] and [Dylan inspired Jobs], shown in Fig. 2 along with the query processing for the query: (unmatched = [?X inspired ?Y], [?Y inspired ?Z]; bindings = {}). Execution begins in the query vertex.

1. Once the query vertex has been added to the graph, it emits a query particle, which is illustrated in blue. Given its first triple pattern, the query particle is routed to the index vertex with ID [* inspired *].
2. Inside the index vertex, the query particle is split into two particles, one colored green and the other one orange (for illustration). The tickets of the blue particle are evenly split among the green and the orange particle. Both particles are sent down to their respective index vertex, the green one to [* inspired Dylan] and the orange one to [* inspired Jobs]. These index vertices, in turn, send the particles further down to their corresponding triple vertices.
3. The first pattern of the green particle gets matched in the triple vertex [Elvis inspired Dylan]. The triple vertex sends the updated particle (unmatched = [Dylan inspired ?Z]; bindings = { ?X=Elvis, ?Y=Dylan }) to the index vertex with ID [Dylan inspired *], which in turn routes the particle towards all triples that match the next unmatched pattern, [Dylan inspired ?Z].
4. From the index vertex, the green particle is routed down to the triple vertex [Dylan inspired Jobs], which binds ?Z to Jobs. As there are no more unmatched triple patterns, the triple vertex routes the particle containing successful bindings for all variables back to its query vertex.

5. The first pattern of the orange particle gets matched in the triple vertex [Dylan inspired Jobs]. The triple vertex sends the updated particle (unmatched = [Jobs inspired ?Z]; bindings = { ?X=Dylan, ?Y=Jobs }) to the index vertex with ID [Jobs inspired *]. The message cannot be delivered, because no index vertex with that ID is found. The handler for undeliverable messages reroutes the failed query particle to its query vertex.
6. The query vertex receives both the successfully bound green and the failed orange particle. Query execution has finished, because all tickets that were sent out with the initial blue particle have been received again. The query vertex reports the successful bindings { ?X=Elvis, ?Y=Dylan, ?Z=Jobs } and then removes itself from the graph.

The architecture and query execution scheme we described captures the essence of how TripleRush works. Next, we explain how we improved them with regard to performance and memory usage.

3.3 Optimizations

The system, as previously shown, already supports parallel graph exploration. Here we describe how we implemented and optimized different components.

Dictionary Encoding While the examples we gave so far represented triple patterns in terms of strings, the actual system implementation operates on a dictionary encoded representation, where RDF resource identifiers and literals are encoded by numeric IDs. Both wildcards and variables are also represented as numeric IDs, but variable IDs are only unique in the context of a specific query.

Index Graph Structure We remove triple vertices, and instead store the triple information inside each of the three third level index vertices that have a compatible triple pattern. We hence refer to these index vertices as *binding index vertices*, because they can bind query particles to triples, which was previously done by the triple vertices. This change saves memory and reduces the number of messages sent during query execution.

One question that arises from this change is: If the subject, predicate and object of the next unmatched pattern of a particle are already bound, where should that particle be routed to? With no single triple vertex responsible anymore, TripleRush load balances such routings over the three binding index vertices into which the triple information was folded.

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 *child-ID-refinements*. The same reasoning applies to binding index vertices, where the triples they store only vary in one position from the ID of the binding index vertex. Analogously, we refer to these differences as *triple-ID-refinements*.

Binding index vertices need to be able to check quickly if a triple exists. This requires fast search over these triple-ID-refinements. We support this by storing them in a sorted array on which we use the binary search algorithm.

Index vertices of levels 1 and 2 do not need to check for the existence of a specific child-ID, as these levels always route to all children. Routing only requires a traversal of all child-ID-refinements. To support traversal in a memory-efficient way, we sort the child-ID-refinements, perform delta encoding on them, and store the encoded deltas in a byte array, using variable length encoding.

Note that these array representations are inefficient for modifications, which is why we initially store the child/triple-ID-refinements in tree sets and switch the representation to arrays once the loading is done. This design supports fast inserts at the cost of increased memory usage during loading.

Query Optimization The number of particle splits performed depends on the order in which the triple patterns of a query are explored. One important piece of information to determine the best exploration order is the number of triples that match a given triple pattern, which we refer to as the cardinality of the pattern. Because only relative cardinality differences matter for ordering, we can assume a very large number for the root vertex. The binding index vertices already store all the triples that match their pattern and thus have access to their cardinalities. So we only need to determine the cardinality of index vertices on level 2, below the root vertex and above the binding index vertices. A level 2 index vertex requests cardinality counts from its binding index vertex children and sums up the replies. We do this once after graph loading and before executing queries, but it can be done at any time and could also be done incrementally.

Query optimization happens only once inside the query vertex before the query particle is emitted. To support it, the query vertex first sends out cardinality requests to the vertices in the index graph that are responsible for the respective triple patterns in the query. These requests get answered in parallel and, once all cardinalities have been received, we greedily select the pattern with the lowest cardinality to be matched first. If this match will bind a variable, we assume that the cardinality of all other patterns that contain this variable is reduced, because only a subset of the original triples that matched the pattern would be explored at that point. To this end, we divide the cardinality estimate for each triple pattern containing bound variables by a constant per bound variable. In our experiments we set the constant to 100, based on exploration.[6] If all variables in a pattern are bound (meaning that all its variables appear in

[6] We tried different factors and this one performed well on the LUBM benchmark. It also performed well on the DBPSB benchmark, which suggests that it generalizes at least to some degree.

patterns that will get matched before it), then we assume a cardinality of 1, designating that at most one triple could match this pattern.

Repeating the procedure we choose the next pattern with the lowest cardinality estimate, until all patterns have been ordered. Next, the initial query particle and all its copies explore the patterns in the order specified by the optimization.

Optimizations to Reduce Messaging Each TripleRush vertex is transparently assigned to a SIGNAL/COLLECT worker. Workers are comparable to a thread that is responsible for messaging and for scheduling the execution of its assigned vertices.

Sending many individual messages between different SIGNAL/COLLECT workers is inefficient, because it creates contention on the worker message queues. In order to reduce the number of messages sent, we use a bulk message bus that bundles multiple messages sent from one worker to another. In order to reduce message sizes and processing time in the query vertex, we do not send the actual failed particle back to the query vertex, but only its tickets.[7] We also use a Pregel-like combiner that sums up the tickets in the bulk message bus, to again reduce the number of messages sent.

Because the query vertex is a bottleneck, we further reduce the number of messsages it receives and the amount of processing it does by combining multiple successful particles into one result buffer before sending them. The query vertex can concatenate these result buffers in constant time.

4 Evaluation

In the last section we described the TripleRush architecture and parallel query execution. In this section we evaluate its performance compared to other state-of-the-art triple stores.

4.1 Performance

TripleRush was built and optimized for query execution performance. In order to evaluate TripleRush, we wanted to compare it with the fastest related approaches. Trinity.RDF [15] is also based on a parallel in-memory graph store, and it is, to our knowledge, the best performing related approach. Thus, our evaluation is most interesting in a scenario where it is comparable to that of Trinity.RDF. As Trinity.RDF is not available for evaluation, we decided to make our results comparable by closely following the setup of their published parallel evaluations. The Trinity.RDF paper also includes results for other in-memory and on-disk systems that were evaluated with the same setup, which allows us to compare TripleRush with these other systems in terms of performance.

[7] The flag is also necessary and in practice we encode it in the sign.

Datasets and Queries Consistent with the parallel Trinity.RDF [15] evaluation, we benchmarked the performance of TripleRush by executing the same seven queries on the LUBM-160 dataset (∼21 million triples) and the same eight queries on the DBPSB-10 dataset (∼14 million triples). The LUBM (Lehigh University Benchmark) dataset is a synthetic one, generated with UBA1.7,[8] while the DBPSB (DBpedia SPARQL Benchmark) dataset is generated based on real-world DBpedia data [9].[9] The queries cover a range of different pattern cardinalities, result set sizes and number of joins. The queries L1-L7 and D1-D8 are listed in the Appendix. These queries only match basic graph patterns and do not use features unsupported by TripleRush, such as aggregations or filters. More information about the datasets and the queries is found in [2] and [15].

Evaluation Setup In the Trinity.RDF paper, all triple stores are evaluated in an in-memory setting, while RDF-3X and BitMat are additionally evaluated in a cold cache setting [15].

For evaluating TripleRush, we executed all queries on the same JVM running on a machine with two twelve-core AMD Opteron™ 6174 processors and 66 GB RAM, which is comparable to the setup used for the evaluation of Trinity.RDF.[10] The whole set of queries was run 100 times before the measurements in order to warm up the JIT compiler, and garbage collections were triggered before the actual query executions. All query executions were complete, no query particle ever ran out of tickets. We repeated this evaluation 10 times.[11]

The execution time covers everything from the point where a query is dispatched to TripleRush until the results are returned. Consistent with the Trinity.RDF setup[12], the execution times *do* include the time used by the query optimizer, but *do not* include the mappings from literals/resources to IDs in the query, nor the reverse mappings for the results.

Result Discussion The top entries in Tables 1 and 2 show the minimum execution times over 10 runs. According to our inquiry with the authors of the

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

[9] `http://aksw.org/Projects/DBPSB.html`, dataset downloaded from `http://dbpedia.aksw.org/benchmark.dbpedia.org/benchmark_10.nt.bz2`

[10] The evaluation in [15] was done on two Intel Xeon E5650 processors with 96 GB RAM. The review at `http://www.bit-tech.net/hardware/cpus/2010/03/31/amd-opteron-6174-vs-intel-xeon-x5650-review/11` directly compares the processors and gives our hardware a lower performance score.

[11] The operating system used was Debian 3.2.46-1 x86_64 GNU/Linux, running the Oracle JRE version 1.7.0_25-b15. More details are available in the benchmarking code on GithHub at `https://github.com/uzh/triplerush/tree/evaluation-ssws`, classes com/signalcollect/triplerush/evaluation/LubmBenchmark.scala and com/signalcollect/triplerush/evaluation/DbpsbBenchmark.scala, using dependency `https://github.com/uzh/signal-collect/tree/evaluation-ssws`. The full raw results are available at `https://docs.google.com/spreadsheet/ccc?key=0AiDJBXePHqCldEVWVU05blNLUHhTM1hhVTYySHp2MkE`

[12] We inquired about what is included in the execution time for the systems in [15].

Trinity.RDF paper [15], this is consistent with their measures. Additionally, we also report the average execution times for TripleRush. TripleRush performs fastest on six of the seven LUBM queries, and on all DBPSB queries. For the query where TripleRush is not the fastest system, it is the second fastest system.

On all queries, TripleRush is consistently faster than Trinity.RDF. In the geometric mean of both benchmarks, TripleRush is more than three times faster than Trinity.RDF, between seven and eleven times faster than RDF-3X (in memory) and between 31 and 103 times faster than BitMat (in memory). For individual queries the results are even more pronounced: On query L7 TripleRush is about ten times faster than Trinity.RDF, on L1 it is more than two orders of magnitude faster than RDF-3X (in memory) and on L4 TripleRush is more than three orders of magnitude faster than BitMat (in memory).

These results indicate that the performance of TripleRush is competitive with, or even superior to other state-of-the-art triple stores.

Fastest of 10 runs	L1	L2	L3	L4	L5	L6	L7	Geo. mean
TripleRush	**80.9**	**53.7**	78.5	**1.5**	**0.8**	**1.5**	**63.2**	**12.1**
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
Average over 10 runs								
TripleRush	89.3	60.1	84.1	1.7	1.3	2.3	69.4	14.8

Table 1. LUBM-160 benchmark, time in milliseconds for query execution on ∼21 million triples. Comparison data for Trinity.RDF, RDF-3X and BitMat from [15].

Fastest of 10 runs	D1	D2	D3	D4	D5	D6	D7	D8	Geo. mean
TripleRush	**1.8**	**73.3**	**1.1**	**1.3**	**1.2**	**6.1**	**6.4**	**8.2**	**4.1**
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
Average over 10 runs									
TripleRush	2.0	82.8	1.3	1.8	1.5	8.4	9.1	12.4	5.3

Table 2. DBPSB-10 benchmark, time in milliseconds for query execution on ∼14 million triples. Comparison data for Trinity.RDF, RDF-3X and BitMat from [15].

4.2 Data Scalability

Performance is a very important characteristic for a triple store, but it is also important that the query execution time scales reasonably when queries are executed over more triples.

We evaluate the data scalability of TripleRush by executing the LUBM queries L1-L7 with the same setup as in subsection 4.1, but ranging the dataset sizes from 10 to 320 universities and measuring the average time over 10 runs. The execution time for queries L1-L3 and L7 should increase with the size of the dataset, which is proportional to the number of universities. Queries L4-L6 are tied to a specific university and, given a good query plan, should take approximately the same time to execute, regardless of the dataset size. The left chart in Fig. 3 shows the execution times on a linear scale, while for queries L1-L3 and L7 both number of universities and the execution times are shown on a logarithmic scale. We see that queries L2 and L7 scale slightly sublinearly. Queries L1 and L3 scale almost linearly until LUBM-160, and then with a factor of more than three on the step to LUBM-320. We are uncertain about what causes this difference. As expected, the results in the left chart in Fig. 3 show that for queries L4-L6 the query execution time does not increase with the dataset size.

Overall, this evaluation suggests that TripleRush query execution times scale as expected with increased dataset sizes, but leaves an open question related to the scaling of queries L1 and L3 on LUBM-320.

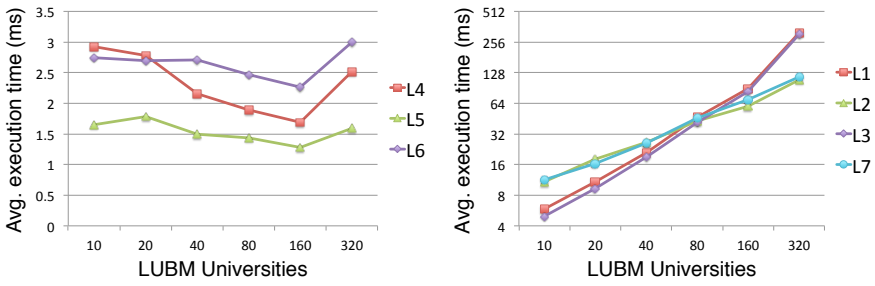

Fig. 3. Average execution times (10 runs) for queries L1-L7 on different LUBM sizes.

4.3 Memory Usage

Another important aspect of a triple store is the memory usage and how it increases with dataset size. In order to evaluate this aspect, we measured the memory usage of TripleRush after loading LUBM dataset sizes ranging from 10 to 320 universities and optimizing their index representations (smallest memory footprint of entire program from 10 runs). Figure 4 shows that the memory usage increases slighly sublinearly for this dataset. The memory footprint of TripleRush

is 5.8 GB when the 21 347 998 triples in the LUBM-160 datset are loaded. This is equivalent to ∼272 bytes per triple for this dataset size. TripleRush requires 3.8 GB for the DBPSB-10 dataset with 14 009 771 triples, which is equivalent to ∼271 bytes per triple. This is between a factor of 2 up to 3.6 larger than the index sizes measured for these datasets in Trinity.RDF [15], but far from the index size of 19 GB measured for DBPSB-10 in BitMat [15].

Currently, graph loading and index optimization for LUBM-160 takes as little as 106 seconds (without dictionary encoding, average over 10 runs). This is because the tree set data structure we use during graph loading supports fast insertions. The flip side is the high memory usage, which causes the graph-loading of the LUBM-320 dataset to take excessively long. Most of that time is spent on garbage collection, most likely because the entire assigned 31 GB heap is used up during loading. After loading is finished, the index representation optimizations reduce the memory usage to a bit more than 11 GB.

Overall, the index size of TripleRush is rather large, but that is in many cases a reasonable tradeoff, given the performance.

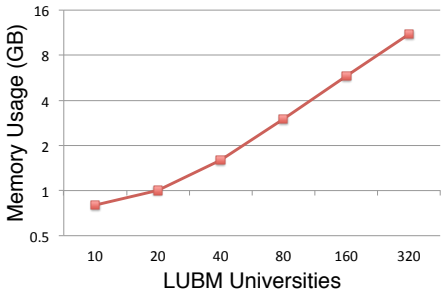

Fig. 4. Memory usage after loading LUBM datasets.

5 Limitations and Future Work

Our current investigation and design has a number of limitations that should be addressed in future work.

First, we need to evaluate TripleRush with additional benchmarks.

Second, and more importantly, we need to investigate the performance of TripleRush on larger graphs, in a distributed setting. Whilst we are optimistic that some of its optimizations will help even in the distributed setting, it is unclear what the overall performance impact of increased parallelism and increased messaging cost will be.

Third, TripleRush was not built with SPARQL feature-completeness in mind. Many SPARQL constructs such as filters and aggregates were not implemented.

Fourth, the current query optimization is very simple and could be improved.

Fifth, the root vertex is a communication bottleneck. Potential avenues for addressing this are to disallow a completely unbound query, which would retrieve the whole database, or to partition this vertex.

Sixth, the memory usage during graph loading should be reduced without overly slowing down the loading times.

Seventh, although the hardware we ran the benchmarks on had a lower performance score, it is desirable to do a comparison with Trinity.RDF on exactly the same hardware.

Even in the light of these limitations, TripleRush is a highly competitive system in terms of query execution performance. To our knowledge, it is the first triple store that decomposes both the storage and query execution into interconnected processing elements, thereby achieving a high degree of parallelism that contains the promise of allowing for transparent distributed scalability.

6 Conclusions

The need for efficient querying of large graphs lies at the heart of most Semantic Web applications. The last decade of research in this area has shown tremendous progress based on a database-inspired paradigm. Parallelizing these centralized architectures is a complex task. The advent of multi-core computers, however, calls for approaches that exploit parallelization.

In this paper we presented TripleRush, an in-memory triple store that inherently divides the query execution among a large number of active processing elements that work towards a solution in parallel. We showed that this approach is both fast and scalable.

Whilst TripleRush has its limitations, it is a step towards providing high-performance triple stores that inherently embrace parallelism.

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 as well as Cosmin Basca for their feedback on our ideas.

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. VLDB Endowment, 2007.
2. M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: a scalable lightweight join query processor for rdf data. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 41–50, New York, NY, USA, 2010. ACM.
3. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

4. J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.
5. J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *PVLDB*, 4(11):1123–1134, 2011.
6. S. Kotoulas, J. Urbani, P. A. Boncz, and P. Mika. Robust runtime optimization and skew-resistant execution of analytical sparql queries on pig. In P. Cudré-Mauroux, J. Heflin, E. Sirin, T. Tudorache, J. Euzenat, M. Hauswirth, J. X. Parreira, J. Hendler, G. Schreiber, A. Bernstein, and E. Blomqvist, editors, *International Semantic Web Conference (1)*, volume 7649 of *Lecture Notes in Computer Science*, pages 247–262. Springer, 2012.
7. 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.
8. 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 Conference*, pages 135–146, 2010.
9. M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo. Dbpedia sparql benchmark: performance assessment with real queries on real data. In *Proceedings of the 10th international conference on The semantic web - Volume Part I*, ISWC'11, pages 454–469, Berlin, Heidelberg, 2011. Springer-Verlag.
10. 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*, SIGMOD '09, pages 627–640, New York, NY, USA, 2009. ACM.
11. T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal.The International Journal on Very Large Data Bases*, 19(1):91–113, 2010.
12. B. Shao, H. Wang, and Y. Li. The trinity graph engine. Technical report, Technical Report 161291, Microsoft Research, 2012.
13. P. Stutz, A. Bernstein, and W. W. Cohen. Signal/Collect: Graph Algorithms for the (Semantic) Web. In P. P.-S. et al., editor, *International Semantic Web Conference (ISWC) 2010*, volume LNCS 6496, pages pp. 764–780. Springer, Heidelberg, 2010.
14. 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.
15. 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), 2013.

Appendix A: Evaluation Queries

LUBM evaluation queries, originally used in the BitMat evaluation [2] and selected by them from OpenRDF LUBM Queries.

```
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns>

L1: SELECT ?X ?Y ?Z WHERE {
    ?Z ub:subOrganizationOf ?Y.
    ?Y rdf:type ub:University.
    ?Z rdf:type ub:Department.
    ?X ub:memberOf ?Z.
    ?X rdf:type ub:GraduateStudent.
    ?X ub:undergraduateDegreeFrom ?Y.
}
L2: SELECT ?X ?Y WHERE {
    ?X rdf:type ub:Course.
    ?X ub:name ?Y.
}
L3: SELECT ?X ?Y ?Z WHERE {
    ?X rdf:type ub:UndergraduateStudent.
    ?Y rdf:type ub:University.
    ?Z rdf:type ub:Department.
    ?X ub:memberOf ?Z.
    ?Z ub:subOrganizationOf ?Y.
    ?X ub:undergraduateDegreeFrom ?Y.
}
L4: SELECT ?X ?Y1 ?Y2 ?Y3 WHERE {
    ?X ub:worksFor
    <http://www.Department0.University0.edu>.
    ?X rdf:type ub:FullProfessor.
    ?X ub:name ?Y1.
    ?X ub:emailAddress ?Y2.
    ?X ub:telephone ?Y3.
}
L5: SELECT ?X WHERE {
    ?X ub:subOrganizationOf
     <http://www.Department0.University0.edu>.
    ?X rdf:type ub:ResearchGroup.
}
L6: SELECT ?X ?Y WHERE {
    ?Y ub:subOrganizationOf
       <http://www.University0.edu>.
    ?Y rdf:type ub:Department.
    ?X ub:worksFor ?Y.
    ?X rdf:type ub:FullProfessor.
}
L7: SELECT ?X ?Y ?Z WHERE {
    ?Y ub:teacherOf ?Z.
    ?Y rdf:type ub:FullProfessor.
    ?Z rdf:type ub:Course.
    ?X ub:advisor ?Y.
    ?X rdf:type ub:UndergraduateStudent.
    ?X ub:takesCourse ?Z.
}
```

DBPSB evaluation queries, received courtesy of Kai Zeng.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbpprop: <http://dbpedia.org/property/>
PREFIX dbpres: <http://dbpedia.org/resource/>
PREFIX rdfcore: <http://www.w3.org/2004/02/skos/core#>

D1: SELECT ?X WHERE {
    ?Y rdfcore:subject dbpres:Category:First-person_shooters.
    ?Y foaf:name ?X.
}
D2: SELECT ?X WHERE {
    ?Z foaf:homepage ?Y.
    ?Z rdf:type ?X.
}
D3: SELECT ?X ?Y ?Z WHERE {
    ?Z rdfcore:subject dbpres:Category:German_musicians.
    ?Z foaf:name ?X.
    ?Z rdfs:comment ?Y.
}
D4: SELECT ?W ?X ?Y ?Z WHERE {
    ?Z dbo:birthPlace dbpres:Berlin.
    ?Z dbo:birthDate ?X.
    ?Z foaf:name ?W.
    ?Z dbo:deathDate ?Y.
}
D5: SELECT ?X ?Y ?Z WHERE {
    ?Z rdfcore:subject dbpres:Category:Luxury_vehicles.
    ?Z foaf:name ?Y.
    ?Z dbo:manufacturer ?W.
    ?W foaf:name ?X.
}
D6: SELECT ?Z1 ?Z2 ?Z3 ?Z4 WHERE {
    ?X rdf:type ?Y.
    ?X dbpprop:name ?Z1.
    ?X dbpprop:pages ?Z2.
    ?X dbpprop:isbn ?Z3.
    ?X dbpprop:author ?Z4.
}
D7: SELECT ?Y WHERE {
    ?X rdf:type ?Y.
    ?X dbpprop:name ?Z1.
    ?X dbpprop:pages ?Z2.
    ?X dbpprop:isbn ?Z3.
    ?X dbpprop:author ?Z4.
}
D8: SELECT ?Y WHERE {
    ?X foaf:page ?Y.
    ?X rdf:type dbo:SoccerPlayer.
    ?X dbpprop:position ?Z1.
    ?X dbpprop:clubs ?Z2.
    ?Z2 dbo:capacity ?Z3.
    ?X dbo:birthPlace ?Z4.
    ?Z4 dbpprop:population ?Z5.
    ?X dbo:number ?Z6.
}
```