# TripleRush

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

## 1 Introduction

TripleRush[1] 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 a parallel and distributed graph processing framework. 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.

Among the existing triple stores, we only know of Trinity.RDF [3] to be implemented on top of a distributed graph processing abstraction. Trinity.RDF represents the graph with adjacency lists and combines traditional query processing with graph exploration.

In TripleRush, an RDF triple is represented as a vertex and SPARQL queries are answered with a purely exploration-based approach. In other words, TripleRush does not use any joins in the traditional sense but searches the index graph in parallel. Whilst traditional stores pipe data through query processing operators, TripleRush routes query descriptions to data. We implemented TripleRush on top of our graph processing framework Signal/Collect [2].

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.

## 2 Foundation: Signal/Collect

Signal/Collect [2] is a parallel and distributed large-scale graph processing system written in Scala. Akin to Pregel [1], it allows to specify graph computations in terms of vertex-centric methods.

The key features of Signal/Collect are: a) it is suitable for expressing data-flow algorithms and transparently parallelizes them, using vertices as processing elements and edges for message propagation, b) it supports different types of vertices, c) the graph structure can be changed during computation, d) it supports bulk-messaging and combiners for message-passing efficiency, e) it supports asynchronous scheduling, and f) it is flexible with regard to edge representation, messages can be routed directly.

## 3 TripleRush

TripleRush is a triple store with three types of Signal/Collect vertices:

[1] Source code at https://github.com/uzh/triplerush

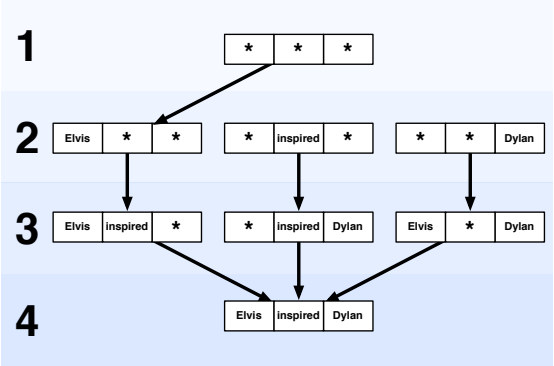


**Fig. 1.** TripleRush index structure that is created for the triple vertex [ Elvis inspired Dylan ].

**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. The query vertex emits the first query particle that traverses the index structure. Once all query particles—successfully matched or not—get routed back to their respective query vertex, it reports the results and 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 these 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.

To ensure the uniqueness of a 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 that is illustrated in Fig. 1.

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.

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.

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

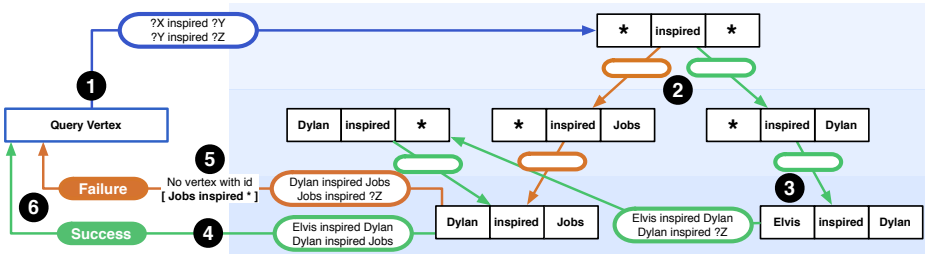


**Fig. 2.** Query execution on the relevant part of the index that was created for the triples [ Elvis inspired Dylan ] and [ Dylan inspired Jobs ].

pattern. To determine when a query has finished processing, the initial query particle is endowed with a large number of tickets.

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. Then, the query vertex reports the result that consists of the variable bindings of the successful query particles, and removes itself from the graph.

We further perform some optimizations: a) we do dictionary encoding, b) we remove the triple vertices and turn the third index level into binding index vertices, which hold a compact representation of all the triples that match their pattern, c) the index vertices on levels 1 and 2, in addition to a compact edge representation, use delta-encoding and variable length integers to further reduce memory usage, d) we use a query optimizer that reorders patterns based on cardinalities, e) we only send the tickets of the failed particles back to the query vertex, and f) we use bulk-messaging and message-combiners.

## 4 Evaluation

In order to evaluate TripleRush, we wanted to compare it with the fastest related approaches. Trinity.RDF [3] is also based on a parallel in-memory graph store, and it is, to our knowledge, the best performing related approach. As Trinity.RDF is not available for evaluations, we made 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.

Consistent with the parallel Trinity.RDF [3] evaluation, we benchmarked the performance of TripleRush by executing the same queries on the LUBM-160 and DBPSB-10 datasets. More information about the setup is found in [3].

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, the execution times *do* include the time used by the query optimizer, but *do not* include the dictionary encoding and the triple materialization.

The results in Figure 3 show the fastest execution time of 10 runs for all stores. The results for the other stores are from the Trinity.RDF paper [3] and were measured on comparable hardware. TripleRush is fastest on all but one query. These results indicate that the performance of TripleRush is competitive with, or even superior to other state-of-the-art triple stores.

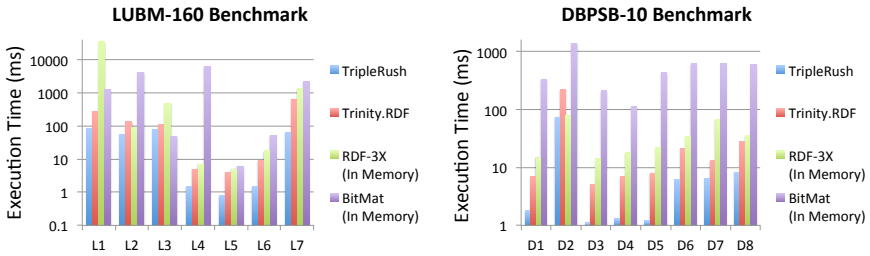


**Fig. 3.** Execution times for queries on the LUBM-160 and DBPSB-10 benchmarks (note the logarithmic scale). Comparison data for other stores from [3].

## 5 Conclusions

The need for efficient querying of large graphs lies at the heart of most Semantic Web applications. During the last decade, research in this area has made 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.

With TripleRush we presented an in-memory triple store that divides the work among a large number of active processing elements that work towards a solution in parallel, and our evaluation shows that the performance of this approach is promising.

## References

1. 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.
2. 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.
3. 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.