University of Zurich[UZH]

Query-Driven Index Partitioning for TripleRush

Thesis August 23, 2014

Christian Tschanz
of Zürich ZH, Switzerland

Student-ID: 08-918-773
christian.tschanz@uzh.ch

Advisor: **Philip Stutz**

Prof. Abraham Bernstein, PhD
Institut für Informatik
Universität Zürich
http://www.ifi.uzh.ch/ddis

Query-Driven Index Partitioning for TripleRush

Christian Tschanz
University of Zurich

TripleRush is a distributed RDF triple store that can be queried with a subset of SPARQL. The index structure of TripleRush is represented as a graph. Executing a query on TripleRush will send messages along the edges in the index graph. This leads to messages being sent between different server nodes as the index graph vertices are distributed over multiple servers. The hypothesis is, that a big part of the query execution time is due to network latency. Reducing the amount of messages traversing the network during query execution would result in improved execution times.

The goal of this thesis is to improve the query execution times of a specific set of queries by analyzing how TripleRush executes them and devising optimization strategies to reduce the inter-node network traffic. These optimizations are based on query execution logs of these queries. These logs represent a sub-graph, the query graph, of the index structure. The discussed approach uses the query graph to re-distribute the relevant parts of the index structure. This optimizes the index graph in such a way that the studied set of queries will run faster due to the optimized vertex placement. The aim is to re-partition parts of the index structure to reduce inter-node edges while maintaining an even distribution for load balancing.

Two approaches are proposed to re-partition and transform the query graph to produce an optimized distribution of TripleRush index vertices. The resulting datasets can be re-introduced into TripleRush with minimal modifications to TripleRush. It has been found that one of the approaches shows promise to significantly improve query execution times for the studied set of queries, while maintaining the distributed load balancing.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed System—*Distributed databases*; H.2.4 [**Database Management**]: Systems—*Query processing*; E.1 [**Data Structures**]: Graphs and networks

General Terms: Performance, Measurement

Additional Key Words and Phrases: graph partitioning, triple store, signal-collect

1. INTRODUCTION

TripleRush [Stutz et al. 2013] is a distributed RDF triple store that can be queried with a subset of SPARQL. Comparisons of query execution performance to state-of-the-art triple stores have been strongly in favor of TripleRush. Until now only little optimizations have focused on how the index structure of TripleRush could be adjusted for greater query execution performance in a multi-node, distributed setup.

The index structure of TripleRush is represented as a graph. Executing a query on TripleRush will send messages along the edges in the index graph. This leads to messages being sent between different server nodes as the index graph vertices are distributed over multiple servers. The hypothesis is, that a big part of the query execution time is due to network latency. Reducing the amount of messages traversing the network during query execution would result in improved execution times. This can be done by studying the execution traces of a specific set of queries. These logs span a sub-graph of the index structure. Re-distributing the vertices of this sub-graph with the aim of reducing inter-node edges, will optimize the execution performance for this specific set of queries. However, reducing network traversal increases execution locality on some nodes. Maximizing locality would destroy the load balancing over multiple nodes. The goal of this thesis is to find a balance between reducing inter-node communication and load balancing. If such a balance can be found, query execution will be significantly increased due to the reduced number of messages traversing the network while conserving the load balancing over multiple nodes.

The following topics are the main focus of this work:

Inter-node communication Distributed TripleRush [Internal Sources] disperses the dataset among multiple servers (**nodes**). Depending on the query composition, it is possible that a big part of the execution time is due to network latency. It is, therefore, favorable to reduce inter-node communication as much as possible so as to increase processing locality which leads to a gain in query execution performance.

Distribution The distributed and inherently parallel features of TripleRush should be conserved as much as possible. Some increases in locality can be achieved with no impact on the node distribution, but it will become a trade-off between locality and load balancing depending on the investigated queries and the level of optimization.

Dictionary TripleRush only operates on datasets where all triples have been encoded into integer (ID) triples by using a dictionary. The default node assignment is dynamically computed on the resulting encoded IDs. This facility can be used to define placement of datasets on specified nodes. Creating an optimized dictionary is one of the main objectives.

This thesis explores different approaches to optimize the execution times of a specific set of queries by balancing the reduction of inter-node communication and load balancing. The main contribution is an analysis of the viability of using query traces to re-distribute index vertices and a discussion of possible performance gains.

The following chapter introduces and explains the fundamental systems, mechanisms and data structures that are central to this thesis. After that follows the presentation of the two optimization approaches and the evaluation thereof. Based on the gathered data, a preliminary investigation of a cost model is discussed, leading the the limitations of this thesis and the topics which are still open for future work. I will then conclude with a discussion of the results.

Bachelor's Thesis, August 22, 2014
Christian Tschanz; Zurich, Switzerland
Student-ID: 08-918-773
christian.tschanz@uzh.ch

Advisor: Philip Stutz

Prof. Abraham Bernstein, PhD
Department of Informatics, University of Zurich
http://www.ifi.uzh.ch/ddis

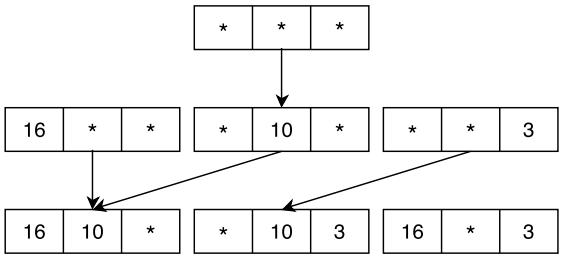

Fig. 1. Index graph, representing the encoded triple `[ 16 10 3 ]`

2. FUNDAMENTALS

This chapter introduces the systems, mechanisms, data structures and tools on which the optimization approaches are based. The next subsection will introduce TripleRush and explain the relevant subsystems in more detail. The subsections following this introduction will present the main data structure to handle the gathered query traces followed by the presentation of the used tools.

2.1 TripleRush

TripleRush is a parallel and distributed in-memory RDF triple store, designed to answer queries over large graphs. It is built on top of the graph processing framework Signal/Collect [Stutz et al. 2010]. The index structure itself is represented as a graph (**index graph**) that can be queried with a subset of SPARQL (basic graph patterns). A distinguishing design aspect is that TripleRush routes query descriptions through the data. Partially matched query copies are routed along different paths in the index structure in parallel. Every vertex in the index graph corresponds to a triple pattern (**TP**). Query execution can transparently create implicit edges in the index structure during execution. The actual triple vertices are stored inside the TP vertices, which have only one wildcard (`*`) element and are, therefore, only conceptually another level in the index graph. TripleRush operates on an encoded version of the data. This encoding will be produced on the fly while the data is loaded into the store. The following parts of TripleRush are of special interest to the proposed (see section 1) ideas and goals.

2.1.1 *Index Graph.* The index structure of TripleRush is represented as a graph in which each vertex represents a triple pattern [Fig. 1]. Partially matched copies of queries will be routed along the edges in this graph and processed in the vertices. The graph in [Fig. 1] has previously been optimized for locality by Stutz et al. [Internal Sources]. Disregarding implicit edges (see section 2.1.3), some edges in the index graph can inherently traverse node boundaries (see section 2.1.4).

2.1.2 *Dictionary.* While parsing the triples, every unique triple element (Subject, Predicate or Object) will be assigned a unique integer ID. The triple `[ Elvis inspired Dylan ]` may be transformed into `[ 16 10 3 ]`. The default behavior is to assign IDs in a linear fashion; starting at 1 and assigning the next, not yet encoded, element the subsequent, unassigned ID. Wildcard elements will always be assigned ID `0`. These encoded triples are then stored and the corresponding triple pattern index graph, will be added to the index structure. These IDs are not only used as an efficient representation of the triple elements, but are also used for node assignment and intra-node balancing (**worker** threads) (see section 2.1.4). The dictionary will only be built once during initialization and does not incur any overhead to the query execution. As node placement can be directly influenced by assigning a triple element a specific ID, generating an optimized dictionary in advance is the optimal way to achieve a specific index graph layout.

2.1.3 *Particle Routing.* The query execution starts with the addition of a query vertex to the TripleRush graph. The query vertex will then emit a single particle, which will be routed to the index vertex matching its first unmatched triple pattern (representing a sub-query). A copy of it is then sent along all edges. Once a particle reaches a triple vertex, its bindings and remaining triple patterns are updated. The particle is then redirected to the vertex matching the next unmatched pattern. To optimize query execution one can't just improve the distributed layout of the index graph; the implicitly added redirection edges resulting from query execution must also be considered. After all bindings have been determined and the particle does no longer contain unmatched queries (or it could not match/bind all variables), it will be sent back to the query vertex. Particles addressed to inexistent triple pattern vertices will be marked as failed and redirected back to the query vertex. Edges to inexistent index vertices are also subject to optimization, as particles sent along such edges they might still traverse the network before the existence of a certain pattern is checked.

2.1.4 *Triple Mapper.* The TripleMapper's responsibility is to uniquely assign a node and worker thread to any given triple pattern. By default, node assignment is governed by only one of the IDs in the triple: the **significant ID**. As the the index vertices have wildcards in different positions, to decide which ID is significant for node assignment, the following hierarchy is respected: `subject > object > predicate`. Starting from the top (subject), the first non-wildcard element in the hierarchy is considered the significant ID. Node assignment is then decided through the following function:

$$a(i, n) = i \bmod n \tag{1}$$

where i is the significant ID and n the total number of nodes over which TripleRush is distributed.

Worker thread assignment can be done through different strategies, depending on the desired load balancing properties.

Distributed Worker assignment is decided by summing the non-wildcard elements in the triple pattern (or 0 if all elements are wildcards) and then applying function (1) with i as the summation of the IDs and n the available worker threads per node.

Alternative This strategy is similar to the $Distributed$ approach but only uses the significant ID (or 0) instead of the summation of non-wildcard IDs.

Hashing To generate a good distribution over worker threads, the sum of the non-wildcard IDs (or 0) is hashed by using the mixing functions of MurmurHash 3 [Appleby 2010] before applying (1), with i as the hash value and n the available worker threads per node.

2.2 Query Graph

To analyze and optimize the index graph distribution, query traces were gathered. Every sent particle traveling along explicit or implicit index graph edges, logged the edge they traversed, which query they belong to and wether its an explicit or implicit edge. Particles sent back to the query vertex, failed or successful, were not logged for these edges were not considered part of the query execution on the index graph. These logs describe the sub-graph which is actively used for query execution. The gathered data is then transformed into an undirected, weighted graph, the **query graph** [Fig.

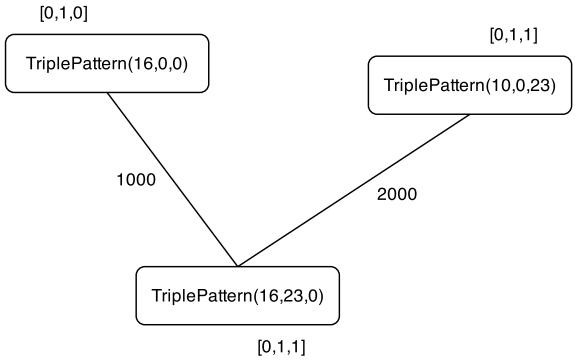

Fig. 2. Example Query Graph. Brackets beside vertices contain a vector of binary vertex weights equal to the number of analyzed queries.

2]. Both the edges and the vertices in the query graph are weighted. Edges were assigned weights proportional to the number of query particles sent along these paths. Vertex weights are represented as a vector of binary weights. The length of the vector is equal to the number of different queries for which traces were generated.

The undirected and weighted query graph is the basis upon which all optimizations were based.

2.3 Tooling

To transform the gathered traces, a tool has been developed[1] which can build the query graph raw, or reduced to the node assignment IDs, partition the resulting graph using METIS [Karypis and Kumar 1998] and generate the resulting dictionary or lookup table (see chapter 3). A second, small tool[2] has been created to gather statistics on the query traces.

3. OPTIMIZATIONS

Two different approaches to improve query execution, using the analysis of query traces, are proposed.

The general idea is to use the query graph to re-partition the vertices over the worker nodes in order to reduce edge cut (inter-node communication). The weighting of edges and aggregation of multiple queries in one graph favor the placement of strongly connected vertices on the same node. A naive partitioning of the query graph could yield an unbalanced partitioning where the vertices participating in a specific query could be entirely clustered on one node. Clustering all vertices which are responsible for a certain query on the same node would forgo the load balancing and parallelism of spreading the vertices over multiple nodes. This poses a trade-off between increased locality and load balancing. The goal is to find a good balance between these extremes to increase locality while still conserving the load balancing. This is achieved by partitioning the query graph in a multi-constrained fashion using the vertex weights. Every element of the vertex weight vector represents a resource that acts as a constraint for the partitioning algorithm. It will try to spread every resource evenly over as many partitions as possible. This will ensure that the distributed nature will not be lost while optimizing edge cut. This approach allows to optimize a group of queries rather than of just one, but will also introduce

[1] https://github.com/chrigi/TripleRush-QueryTraceTool
[2] https://github.com/chrigi/TripleRush-QueryStatsTool

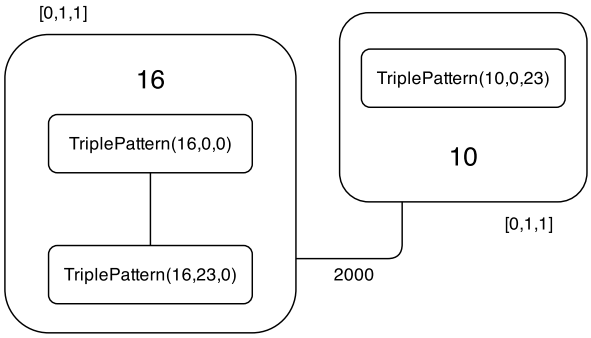

Fig. 3. Transformed significant ID graph based on Fig. 2. The larger numbers are the respective significant IDs. The Triple pattern vertices are only shown for illustrative purposes.

possible bad vertex placements when assigning a vertex in favor of one query instead of others that share this vertex. As it is not possible to produce the best partitioning for any single query nor for the group as a whole, only an approximation can be generated. It is expected that there will be uneven vertex distribution to some extent, due to the fact that only an approximation can be computed and that the multi-constrained approach will introduce a further trade-off between query distribution and edge cut. The aim is to create a good distribution of the index vertices relevant to the selected set of queries. As only the query graph is re-partitioned, no optimizations over the whole index structure is performed. This will reduce inter-node communication and increase processing locality only for the selected queries. Other queries which are similar to the ones used to generate the query graph, should also profit from the optimizations. Queries which are entirely different won't be explicitly optimized and might even get de-optimized.

The query graph can either be partitioned in raw form or reduced further to its significant node assignment IDs. These strategies are discussed in the following subsections.

3.1 Significant ID Partitioning

The most efficient way to import the new vertex partitioning into TripleRush is to define a new Dictionary, which can be imported during startup. This is only possible if all triple elements (subject, object and predicate) contained in the query graph are uniquely mappable to an ID. Instead of defining a new encoding, the currently implemented Triple Mapper is used. It assigns nodes to index triples based on either the subject, object or predicate (in this order), based on which elements are non-wildcards. `TriplePattern(10,0,23)` has no wildcard in the subject position, so the significant node assignment ID (significant ID) is `10`.

The first approach is to reduce the query graph further. Vertices in the reduced graph represent only the unique, significant IDs. Edges which were between vertices with the same significant ID are discarded and edges between different significant IDs are [Fig. fig:siggraph]. The resulting graph can then be partitioned as previously described. As the vertices represent the ID responsible for node assignment, a unique `triple_element -> significant_id -> node` mapping is possible.

To generate the dictionary, the partitioning results as well as the previously valid dictionary are combined. The generated dictionary maps a triple element to an ID. The vertex ID (representing a significant ID in the reduced query graph) is used to extract the corre-

sponding triple element string form the previously valid dictionary. The new ID is then computed using the partitioning results. These results contain information about which vertex (vertex/significant ID) has been put into which partition. The new ID is generated by assigning the next (in an increasing manner), free (not yet assigned to another triple element) ID which maps to the given partition (when applying function (1)) taken form the partitioning results. The exact equation to determine the new ID is as follows:

$$n(p, n) = g(n) * n + p \quad (2)$$

where p is the partition assigned to the vertex through the re-partitioning, n is the total number of partitions and $g(n)$ is the number of vertices which were already assigned to partition p.

Using (2) ensures that a given triple element will be assigned the next free ID but might not generate a contiguous dictionary. Some IDs might be skipped and never be assigned to a triple element while IDs before and after are used. This happens when the significant ID vertices are not perfectly distributed over all partitions. Skipping is necessary to ensure that every ID maps to the correct partition when applying (1).

This approach can re-use already existing facilities of TripleRush. The necessary reduction of the query-graph however allows for sub-optimal index vertex distribution. Re-partitioning of the query graph might lead to a clustering of bigger (e.g. vertex for ID `16` in [Fig. 3], which abstracts two index vertices) significant ID vertices on one node, resulting in a disproportional amount of index vertices on that node. This is expected and can be seen as the trade-off between locality and distribution. It has to be verified that the partitioning algorithm will neither reduce vertices on one node to an unreasonably small number nor overload another.

3.2 Raw Query Graph Partitioning

Using the significant ID partitioning from 3.1 has the benefit that the resulting dictionary only has to be loaded once before the data is loaded into the store. This is an efficient way to define index vertex placement without any query execution overhead and with only small impact on distribution as the size scales well.

Another approach is to partition the raw query graph, where every vertex represents an index triple pattern. This could theoretically produce a more balanced and finer grained partitioning. This results in a `TriplePattern -> node` mapping which depends on a unique `S P O` pattern. The currently implemented TripleMappers determine the node assignment dynamically at run time, based on a single triple element ID (the significant ID). Using a specific triple element ID pattern is not supported for node assignment as it would depend on multiple IDs and their order, which can't be dynamically computed at runtime when using the available TripleMappers. One approach would be to develop a new TripleMapper function and TriplePattern encoding which would allow such an operation.

Instead of developing a new encoding scheme, the `TriplePattern -> node` mapping was implemented in the form of a lookup table. The existing, $distributed$ TripleMapper was modified to read from a lookup table based on the aforementioned mapping and to search this map before employing the default computation. This approach has several limitations:

Size The resulting lookup table can become very big. Queries which traverse a big part of the index graph would result in duplicating much of the dataset in the lookup table. As the raw query graph inherently has double the number of vertices than the reduced graph, a much bigger result has the be expected.

Distribution As the lookup table has to explicitly be accessed upon node assignment in the TripleMapper, it needs to be available on every node at all times. This can complicate the distributed deployment of TripleRush.

Overhead Node assignment lookup is no longer implicit from a simple operation on the available triple pattern elements. It becomes a more expensive lookup in a hash table combined with a possible fallback mechanism.

The following techniques were employed to try to circumvent these limitations:

Size The query graph was extended to include one additional vertex per worker node, representing the nodes. Edges from index vertices to their natural worker node with constant, small weight were added to create an affinity of the vertices to their natural node. During the partitioning step it was made sure that the node vertices were evenly distributed over the partitions. After the partitioning took place, the triple vertices which were placed in the partition of their corresponding natural node, could be excluded from the lookup table, leading to a smaller table.

Distribution Cutting down the size of the lookup table increases feasibility of the distribution.

Overhead To increase the hash table lookup speed, the triple patterns were converted into a more efficient representation. As every index triple pattern consists of a maximum of two non-wildcard integer IDs, it is possible to combine them in a single number of type long and encode which kind of pattern it is in the sign and placement as first or second in the long representation.

This representation has become the standard index vertex representation in TripleRush.

This approach introduces new mechanisms to TripleRush. The chosen lookup table is a small component which can easily be introduced into any TripleMapper. A more advanced, new encoding scheme would require deeper integration.

4. EVALUATION

This section compares the baseline performance of the current TripleRush implementation with the modified TripleRush instance by using the previously described, generated dictionary and lookup table. Different TripleMapper strategies will also be explored.

4.1 Evaluation Setup

All queries were executed on the same JVM (version `1.7.0_65-b32`) running on machines with two, twelve-core AMD Opteron™ 6174 processors and 66 GB RAM, connected through Gigabit ethernet links and running on `Debian 3.2.60-1+deb7u1 x86_64`. Every query was run for 40 seconds (executed as often as possible within 40 seconds) in order to warm up the JIT compiler and garbage collection was triggered before the actual query executions. These evaluations were repeated 15 times per query and the cold run was discarded. The execution times include SPARQL query parsing, execution and result enumeration.[3] Evaluations have been performed in a distributed setting of either 2 or 4 nodes.

4.2 Dataset and Queries

All queries were run on the LUBM (Leigh University Benchmark) [Guo et al. 2005]. Datasets were generated using UBA1.7[4]

[3]Details on the used evaluation procedure can be found here: https://drive.google.com/file/d/0B-ztn8ONihzsT0pNSGM1a1hsTnc/edit?usp=sharing

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

and transformed into the n-triples format. The evaluations were run on the datasets LUBM-40, LUBM-80 and LUBM-160. All evaluations were carried out using the DistributedTripleMapper if not noted differently. To be consistent with other evaluations, the same seven queries that were used in the Trinity.RDF evaluation [Zeng et al. 2013] and TripleRush evaluation have been run. Additional queries were devised as well as incorporated from the Semplore evaluation [Zhang et al. 2007]. All Queries can be found in appendix A.1.

4.3 Evaluations

All gathered, raw data produced by the evaluation is also available.[5] All cited values are the based on the measured minima if not noted differently.

4.3.1 *Lookup Table.* As discussed in 3.2, the use of a lookup table has many disadvantages. Its use would only be warranted if it led to significant performance improvements. A comparison of query execution times can be seen in [Table I]. The results show that L12, the most expensive query, could be executed in significantly less time but many of the other queries show considerable regressions. Overall the execution time could be significantly decreased. Still, it might not be desirable to have only one dominating query improved. A more favorable approach would have execution improvements for many queries. As long as these queries are not executed in rapid succession (batch), no immediate improvements could be found when most of them run slower than the default case. The same behavior can be found in the smaller dataset LUBM40 [Table II]. This disqualifies the Lookup Table approach as a feasible, general query execution optimization technique.

The bad performance of this approach is most likely due to the overhead of the required hash table lookup. When comparing reductions in inter-node edges, it is possible to see that the lookup table reduced the number of inter-node edges for L7, the worst performing query, even more aggressivly than the dictionary approach did, boasting an 86.84% reduction. The increased locality should yield a much better query execution time than measured. Another factor would be distribution of work over worker threads. This angle is analyzed further in 4.3.3.

4.3.2 *Dictionary.* Generating a dictionary, to improve vertex placement, was the main goal. Not only is it more practical and easier to work with and generate, but it is also exhibits superior node scalability. [Table III] shows the file size differences of the generated dictionaries and lookup tables. While the dictionary sizes remain constant when adding more nodes, the lookup table drastically increases in size. It is easy to see that scaling to many more nodes and bigger datasets, puts the lookup table approach at a clear disadvantage with regard to size efficiency.

Total query execution times could be significantly improved in larger data sets, but not to the same extent that the lookup table enabled. This disparity stems from the more even distribution of execution time improvements. Multi-constrained partitioning on the raw query graph mainly optimized the most dominant query at the cost of others [Table I]. However, applying the same partitioning to the reduced query graph, allowed for a much more balanced

[5]Measurements:https://docs.google.com/spreadsheets/d/1YI35Dj2KJPAgbw_RWCwAlXV_NXJkK1yFbRbV2Tt9jYE/edit?usp=sharing
Data: https://drive.google.com/file/d/0B-ztn8ONihzsbTFxS1phMXNmdGM/edit?usp=sharing

Table III. Filesize differences

Dataset (# Nodes)	Dictionary	Lookup
LUBM40 (2)	88M	73M
LUBM160 (2)	359M	337M
LUBM40 (4)	88M	120M
LUBM160 (4)	359M	520M

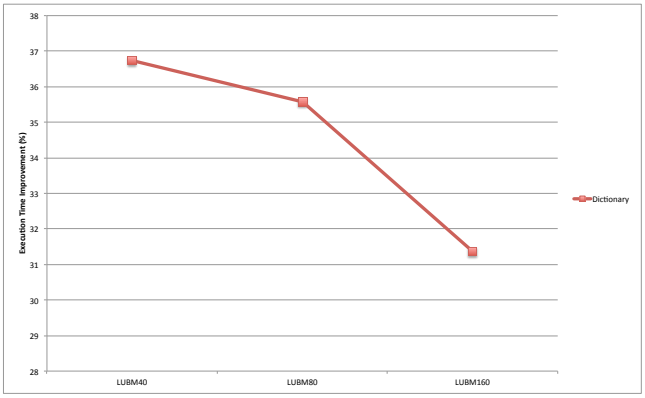

Fig. 4. Execution time improvement (based on execution time sum over all queries) vs. Dataset size, 2 Nodes

partitioning in regard to improvements in individual query executions [Table II]. Using the generated dictionary did not only improve query execution times for more queries but it also bounded the ones which ran slower. Most of the queries which show a decrease in performance, show less then a 10% increase in execution time compared to the default case. This is a considerable improvement as compared with the use of a lookup table when executing the queries individually.

Comparing execution times in a 2 node setup exhibits the most consistent improvements. [Fig. 10] and [Fig. 11] shows significant improvements over all dataset sizes. Closer inspection reveals that the biggest gains result from a drastic improvement in the dominant query L12. As previously mentioned, this is also the case when using a lookup table, which suggests to remain cautious about conclusions regarding general improvements. Fortunately the data [Table II] does not exhibit the narrow optimizations one might expect. A comparison of the gains in query execution performance over different datasets [Fig. 12] shows a clear downwards trend. By further scaling up dataset size, intra-node worker distribution and edges would become the dominating factor that diminishes the gains from optimizing inter-node communication. This can be clearly seen when taking into account the reduction of inter-node edges which result from using the optimized dictionary [Fig. 5]. Doubling the dataset size from LUBM80 to LUBM160 drastically increases the load on individual nodes, but the partitioning could not reduce the inter-edge communication in a similar fashion.

Analyzing execution times on 4 nodes shows a different situation. Deploying the generated dictionary on small datasets can even lead to worse performance. The improvements in individual queries is still better than with a lookup table [Table II], [Fig. 6] and [Fig. 7], but the dominating query could not be optimized as aggressively. Increasing the dataset size shows improvement [Fig. 8] in the summed query execution times, leading to a speedup of over 10% for LUBM160. Combining these insights with the scalability on 2 nodes [Fig. 12] and the trends in [Fig. 8] suggests that the max-

Table I. Query execution times, LUBM160, 4 nodes

Query	Default (ms)	Lookup (ms)	Dictionary (ms)	Improvement Lookup (%)	Improvement Dictionary (%)
L1	158.7	346.6	128.5	-118.40	**19.03**
L2	162.5	198.4	149.2	-22.09	**8.18**
L3	156	315.6	150.3	-102.31	**3.65**
L4	4.9	11.8	5.1	-140.82	-4.08
L5	3.2	3.1	3.1	**3.13**	**3.13**
L6	5.9	6.8	6	-15.25	-1.69
L7	89.7	312.5	62.9	-248.38	**29.88**
L8	8.4	20.1	8.6	-139.29	-2.38
L9	9.9	10.5	8.3	-6.06	**16.16**
L10	61.6	53.8	48.6	**12.66**	**21.10**
L11	29.8	17.5	40.4	**41.28**	-35.57
L12	4798.4	1622	4027.7	**66.20**	**16.06**
L13	156.9	180	167	-14.72	-6.44
L14	2.4	1.2	1.2	**50.00**	**50.00**
L15	787.2	794.7	814.3	-0.95	-3.44
L16	27.2	26.3	23.2	**3.31**	**14.71**
Sum	6462.7	3920.9	5644.4	**39.33**	12.66
Average	403.92	245.06	352.78	**39.33**	12.66
Geometric Mean	44.00	51.51	39.61	-17.07	**9.97**

Table II. Number of queries which could be optimized or were de-optimized

Dataset (# Nodes)	Optimization	# optimized Queries gain>0%	# slightly de-optimzed Queries -10%≤gain≤0%	# strongly de-optimized Queries gain<-10%
LUBM40 (4)	Lookup	3	4	9
LUBM80 (4)	Lookup	0	0	0
LUBM160 (4)	Lookup	6	2	8
LUBM40 (4)	Dictionary	7	4	5
LUBM80 (4)	Dictionary	9	3	4
LUBM160 (4)	Dictionary	10	5	1
LUBM40 (2)	Dictionary	10	5	1
LUBM80 (2)	Dictionary	10	5	1
LUBM160 (2)	Dictionary	12	2	2

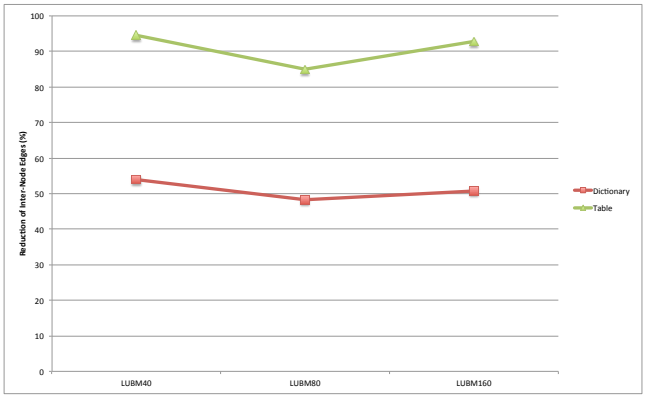

Fig. 5. Reductions of Inter-Node Edges vs. Dataset size, 2 Nodes

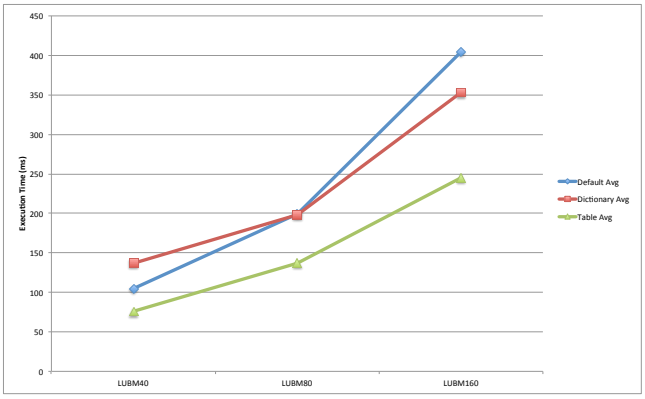

Fig. 6. Average execution time vs. Dataset size, 4 Nodes

imum possible performance improvement will most likely peak for a larger dataset and then decrease again. Reductions in inter-node edges [Fig. 9] between LUBM80 and LUBM160 is similar to the 2 nodes setting. Nevertheless, a higher performance increase for LUBM160 can be measured if compared with LUBM80 in the 4 nodes setting. Distributing the computation over 4 nodes increases the sensitivity to inter-node communication. A 4 node distribution inherently has more inter-edges compared to a 2 node setup [Fig. 13], and distributing the load over more machines won't increase the intra-work as dramatically.

The main reason for the superior performance of the dictionary over individual queries, compared to the lookup table is the reduc-

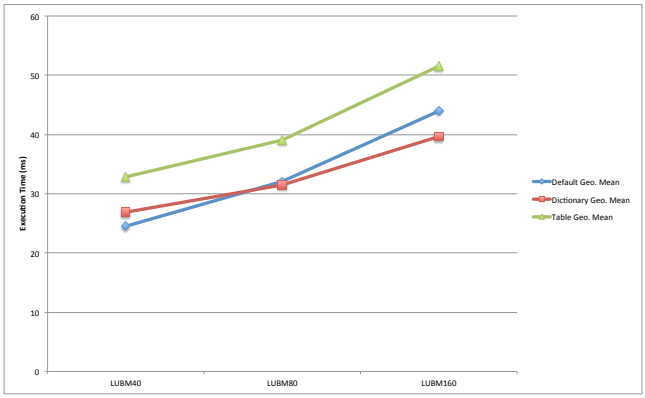

Fig. 7. Geographic mean of execution times vs. Dataset size, 4 Nodes

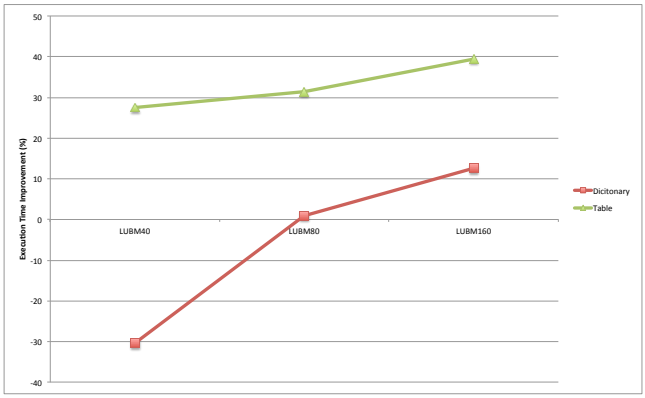

Fig. 8. Execution time improvement (based on execution time sum over all queries) vs. Dataset size, 4 Nodes

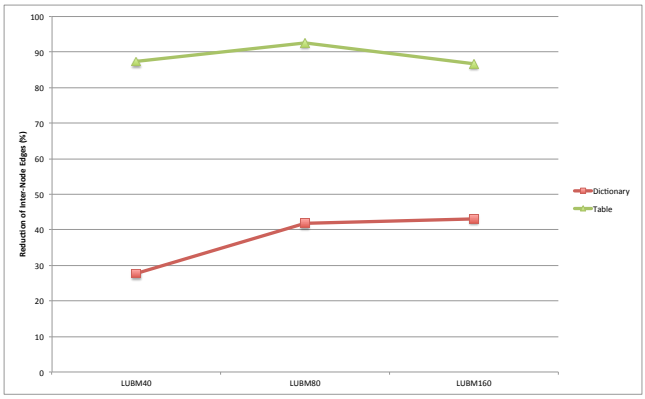

Fig. 9. Reductions of Inter-Node Edges vs. Dataset size, 4 Nodes

tion of the query graph. By aggregating multiple triple pattern vertices into a single significant ID vertex, removes many edges from the graph. This on one hand limits the abilities of the partitioning, as less vertices can be moved or edges cut, but on the other hand ensures that some edges are guaranteed to be intra-node edges. Finding a subject for a given P O combination would traverse significant ID borders and can be optimized by using the reduced query graph partitioning but further restricting the found subjects by varying

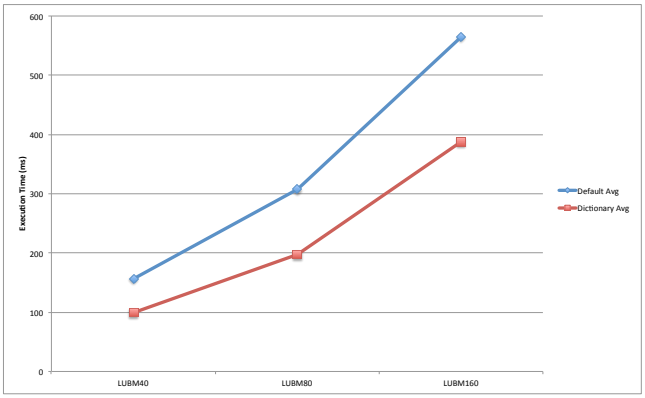

Fig. 10. Average execution time vs. Dataset size, 2 Nodes

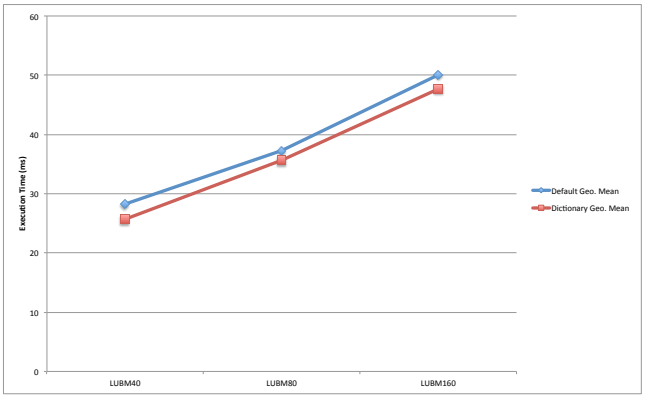

Fig. 11. Geographic mean of execution times vs. Dataset size, 2 Nodes

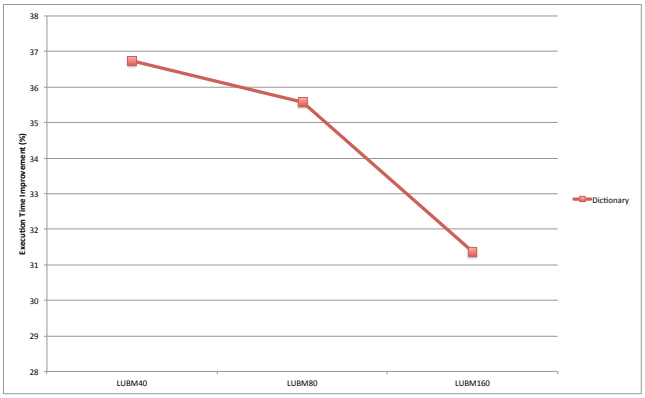

Fig. 12. Execution time improvement (based on execution time sum over all queries) vs. Dataset size, 2 Nodes

P O is guaranteed to be handled intra-node and will never traverse a node boundary. Partitioning this query on the raw query graph might introduce additional node boundary traversals. This suggests that queries where many subqueries only add restrictions to a given set of data (mostly in the form of a fixed subject) would yield bigger improvements in query execution times when using the generated dictionary than queries which contain more sub-queries which traverse significant ID boundaries.

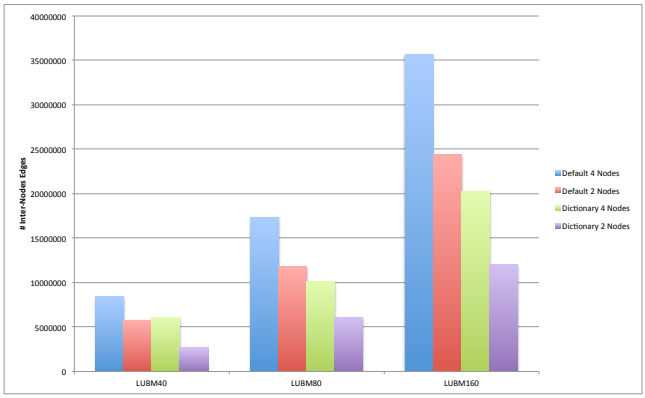

Fig. 13. Inter-Node Edges vs. Dataset size

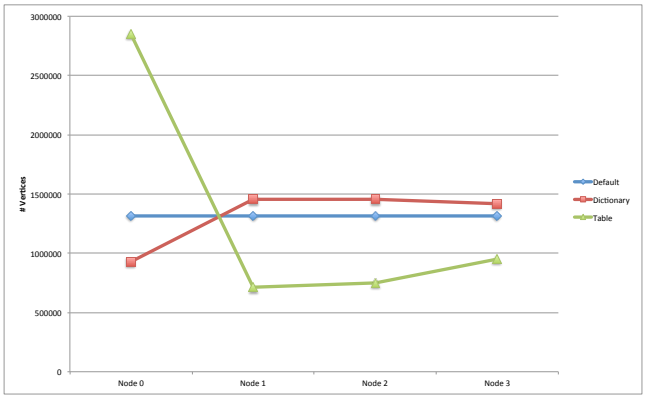

Fig. 14. Vertex Distribution for L12, LUBM160, 4 Nodes

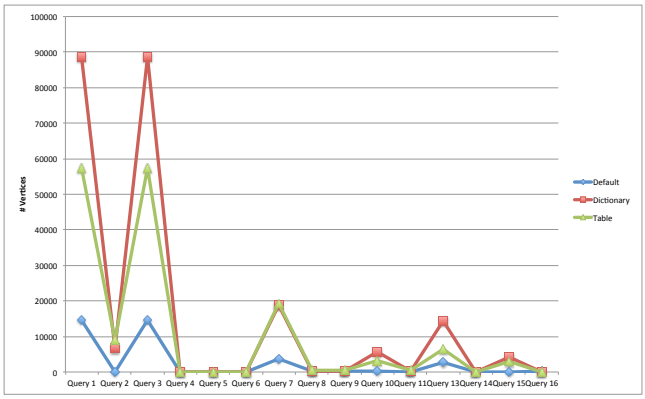

Fig. 15. Standard Deviation of vertices distributed over 4 Nodes, LUBM160

Optimizing inter-node communication results in decreased load balancing. Examining the standard deviation of the distribution of index vertices over the nodes [Fig. 15] (L12 excluded) reveals that locality was measurably increased for some queries, leading to worse load balancing. This was expected an is acceptable as long as the deviations are not too large. On average, these deviations are no more than 3.77% of the involved vertices per query for the

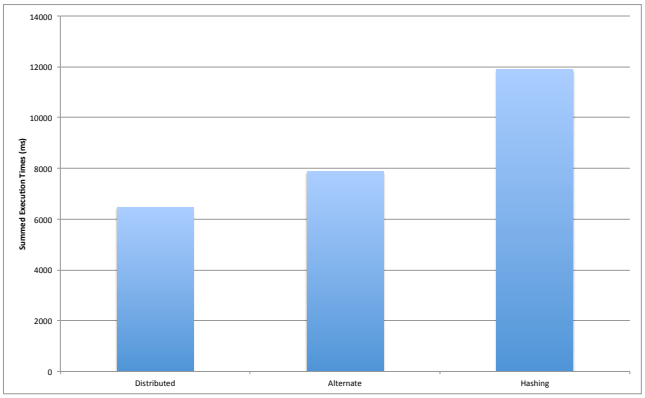

Fig. 16. Total execution times vs. Mapper strategy, LUBM160, 4 Nodes

LUBM160 dataset on 4 nodes. Half of the deviations are below 1%. Different node and dataset combinations present a similar situation.

Closer inspection of L12 [Fig. 14] shows that using the lookup table approach leads to disproportionate locality on Node 0. This is also explains the superior improvements in query execution times for L12 when using a lookup table. Such an unbalanced distribution can be considered suboptimal due to the greatly reduced load balancing.

4.3.3 *Worker Distribution.* Triple Rush offers multiple intra-node load balancing strategies (see 2.1.4). By default, the Distributed mapper is chosen. This mapper does not produce a very good distribution of work over worker threads. Generating a better intra-node distribution presents itself as a sensible approach to increase performance for intra-node processing. This is an important factor for the proposed dictionary approach as it increases locality and, therefore, creates more work for an individual node. Contrary to what one might expect, using a different worker assignment strategy introduced significant regressions in query execution times, both in total execution times [Fig. 16] and for individual queries [Fig. 17]. The Hashing strategy was first introduced and led to improved execution performance in a single node setup. However it performed very poorly in a distributed setting. The hypothesis was that increased worker locality appears to perform better than intra-node messaging. The Alternative mapping strategy would push this even further by assigning workers only based on the significant ID. Benchmarking these mappers clearly shows that the hypothesis on worker locality can be rejected, as even the alternative mapper performs rather poorly in a distributed setting.

The bad performance of these intra-node work distribution strategies is, therefore, unlikely to be found in a model property. It is entirely possible that these differences stem from implementation, library or configuration anomalies.

5. COST MODEL

Initial measurements using the Hashing strategy for worker thread distribution (see 4.3.3) suggested a strong link between intra-node communication and query execution performance. After gathering worker distribution statistics, it was attempted to fit a simple cost model to the data:

$$t = 0.1 + x * (self_{mw} * y + other_{mw} * z + node_{mw} * w)$$

with t as the expected execution time in ms, x, y, z as the fit parameters, $self_{mw}$ the number of messages of the busiest worker set to

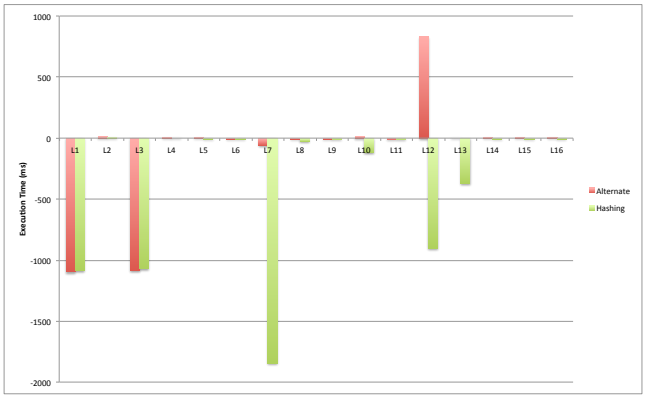

Fig. 17. Execution time differences compared to Distributed strategy, LUBM160, 4 Nodes

itself, $other_{mw}$ the number of messages of the busiest worker sent to another worker on the same node and $node_{mw}$ the number of messages of the busiest worker set to a worker on another node.

No satisfactory fit could be produced by using this model. Measurements using the alternative triple mapping strategy suggest that the link between intra-node work distribution is not decisive enough to warrant a cost model to be based on the assumed strong connection, as the wild performance deviations with different strategies does not appear to be inherent to the model.

To re-partition the query graph and evaluating the approach of using the raw or reduced graph could identify multiple factors that are relevant to the query execution performance in regard to reducing inter-node communication.

Reduction of Inter-Node edges The results could clearly establish that reducing network communication can significantly increase query performance. It could also show that the relative amount of the reduction appears to be bounded around 50%.

Worker Load Relative reduction in inter-node communication does not scale with dataset size. This increases the workload on individual worker threads, which shifts the dominant factor for network latency to processing latency.

Distribution Distributing TripleRush over more nodes, makes it more susceptible to network related performance variations.

Aggregation Suboptimal distribution of vertices over nodes can yield better improvements for individual queries while preserving much of the parallel nature of TripleRush.

Exploring a more complex cost model using these factors might allow for a more accurate prediction of query execution performance.

6. LIMITATIONS

The measurements were only conducted on small dataset sizes compared to what TripleRush is able to handle. Increasing the dataset size would allow to gather more concrete data on scalability and interactions between the identified factors. It was shown that inter-node communication is an important factor in query execution performance but other factors like intra-node workload and balancing can't be neglected.

The distributed setup is limited to 2 and 4 node setups. The design of Signal/Collect and TripleRush is highly distributed and would favor setups with even higher number of nodes. Using more nodes would also increase the sensitivity to network latency and therefore inter-node connectivity.

Partitioning using METIS limits the control over how vertices are distributed over the nodes. Using a multi-constrained partitioning can favor a very small subset of the measured queries, leading to good overall improvements but bad execution times when analyzed on a per query basis. Employing a custom partitioning algorithm to tune the parameters to the specifics of TripleRush and query execution might produce better overall results, especially for the lookup table approach.

All measurements were only done on the LUBM dataset. Evaluation the optimization approaches on different datasets could lead to further insights.

Using multi-constrained partitioning makes the crucial partitioning step highly dependent on the traced queries. Using different sets of queries could yield different results. Some combinations might not be suitable for this approach whereas others are better suited. A comparison between query sets would show the limits of this approach. Using queries with similar execution times (no dominating query) could yield in very different results.

The evaluations were done using a fixed set of queries that were also part of the optimization. The created optimizations were not tested with similar queries which were not part of the optimization step.

7. FUTURE WORK

The discussed approach of reducing inter-node communication has shown promise. Extending this to study messaging on a worker thread level might reveal a more detailed picture of which factors are involved and how to influence them to a greater degree.

TripleRush already gathers many statistics. It would increase the feasibility of generating a dictionary if it could be based on the statistics already present in TripleRush. This would allow for a self-tuning feature.

8. CONCLUSIONS

TripleRush has already proven to be one of the fastest triple stores. Reducing intra-node communication to improve query execution performance has shown further performance gains of up to 35% for a specific set of queries, dataset and node configuration. These results show that index vertex placement can influence query execution with significant impact and proves the hypothesis that network latency is a considerable part of query execution.

Especially the approach of using a reduced query graph to generate an optimized dictionary has shown promise. Not only did it allow for a general improvement in query execution over many dataset sizes and node configurations but it also strikes a good balance between load balancing and increased locality, yielding good optimizations over a wide variety of queries without over optimizing one in favor of another or by diminishing vertex distribution.

It also became evident that fast queries which don't access a lot of data can't be optimized while maintaining good load balancing over different nodes. It might be favorable to situate data accessed by such queries on only one node and distribute the load over worker threads instead of nodes.

The used tooling and infrastructure is not a viable approach to implement a re-partitioning optimization approach directly into TripleRush.

Devising a an feasible implementation directly in TripleRush or optimizing default vertex placement based on heuristics is left for future work.

APPENDIX

A. APPENDIX

A.1 Queries

LUBM evaluation queries. L1-L7 were originally from [Atre et al. 2010]. L8-L13 were created by the author of this thesis. L14-L16 were taken from the Semplore evaluation[6].

```
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 {
    ?X rdf:type ub:GraduateStudent .
    ?X ub:undergraduateDegreeFrom ?Y .
    ?X ub:memberOf ?Z .
    ?Z rdf:type ub:Department .
    ?Z ub:subOrganizationOf ?Y .
    ?Y rdf:type ub:University . }

L2: SELECT ?X ?Y WHERE {
    ?X rdf:type ub:Course .
    ?X ub:name ?Y .}

L3: SELECT ?X ?Y ?Z WHERE {
    ?X ub:undergraduateDegreeFrom ?Y .
    ?X rdf:type ub:UndergraduateStudent .
    ?X ub:memberOf ?Z .
    ?Z ub:subOrganizationOf ?Y .
    ?Z rdf:type ub:Department .
    ?Y rdf:type ub:University . }

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 rdf:type ub:FullProfessor .
    ?Y ub:teacherOf ?Z .
    ?Z rdf:type ub:Course .
    ?X ub:advisor ?Y .
    ?X ub:takesCourse ?Z .
    ?X rdf:type ub:UndergraduateStudent . }

L8: SELECT ?X ?Y ?Z WHERE {
    ?X rdf:type ub:GraduateStudent .
    ?Y rdf:type ub:Department .
    ?X ub:memberOf ?Y .
    ?Y ub:subOrganizationOf <http://www.
        University0.edu> .
    ?X ub:emailAddress ?Z . }

L9: SELECT ?X ?Y ?Z WHERE {
    ?X rdf:type ub:FullProfessor .
    ?Y ub:subOrganizationOf <http://www.
        University0.edu> .
    ?Y rdf:type ub:Department .
    ?X ?Z ?Y . }

L10: SELECT ?X ?Y ?Z WHERE {
    ?Y rdf:type ub:FullProfessor .
    ?Y ub:teacherOf ?Z .
    ?Z rdf:type ub:Course .
    ?X ub:teachingAssistantOf ?Z . }

L11: SELECT ?X ?Y ?Z ?W WHERE {
    ?X rdf:type ub:FullProfessor .
    ?X ub:doctoralDegreeFrom ?Y .
    ?X ub:headOf ?Z .
    ?Z rdf:type ub:Department .
    ?Z ub:subOrganizationOf ?Y .
    ?Y rdf:type ub:University .
    ?W rdf:type ub:UndergraduateStudent .
    ?W ub:advisor ?X . }

L12: SELECT ?X ?Y ?Z WHERE {
    ?X rdf:type ub:Course .
    ?Y rdf:type ub:GraduateStudent .
    ?Y ?Z ?X . }

L13: SELECT ?X ?Y ?Z WHERE {
    ?X rdf:type ub:Course .
    ?Y rdf:type ub:GraduateStudent .
    ?Y ub:teachingAssistantOf ?X .
    ?Y ub:emailAddress ?Z . }

L14: SELECT ?X WHERE {
    ?X ub:memberOf <http://www.Department9.
        University0.edu> . }

L15: SELECT ?X ?Y ?Z WHERE {
    ?X ub:takesCourse ?Y .
    ?Z ub:teacherOf ?Y .
    ?Z rdf:type ub:FullProfessor . }

L16: SELECT ?X ?Y ?Z ?W WHERE {
    ?X ub:takesCourse ?Y .
    <http://www.Department4.University0.edu/
        FullProfessor5> ub:teacherOf ?Y .
    ?X ub:memberOf ?Z .
    ?W ub:memberOf ?Z .
    ?W ub:telephone "xxx-xxx-xxxx" . }
```

[6] http://apex.sjtu.edu.cn/apex_wiki/Semplore_QS

ACKNOWLEDGMENTS

I am grateful to my advisor Philip Stutz for his time, resources and help, which enabled me to complete my bachelor thesis. His support and readiness to help was exceptional. I would like to thank the Institute of Informatics at the University of Zürich for providing me with access to their cluster in order to conduct my experiments.

REFERENCES

Austin Appleby. 2010. MurmurHash 3. (Nov. 2010). https://code.google.com/p/smhasher/

Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. 2010. 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)*. ACM, New York, NY, USA, 41–50. DOI:http://dx.doi.org/10.1145/1772690.1772696

Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A Benchmark for OWL Knowledge Base Systems. *Web Semant.* 3, 2-3 (Oct. 2005), 158–182. DOI:http://dx.doi.org/10.1016/j.websem.2005.06.005

George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20, 1 (Dec. 1998), 359–392. DOI:http://dx.doi.org/10.1137/S1064827595287997

P Stutz, A Bernstein, and W W Cohen. 2010. Signal/Collect: graph algorithms for the (Semantic) Web. In *ISWC 2010*, P F Patel-Schneider (Ed.).

P Stutz, Mihaela Verman, Lorenz Fischer, and Abraham Bernstein. 2013. TripleRush: a fast and scalable triple store. In *9th International Workshop on Scalable Semantic Web Knowledge Base Systems*. CEUR Workshop Proceedings, http://ceur-ws.org, Aachen, Germany.

Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. 2013. A Distributed Graph Engine for Web Scale RDF Data. *Proc. VLDB Endow.* 6, 4 (Feb. 2013), 265–276. DOI:http://dx.doi.org/10.14778/2535570.2488333

Lei Zhang, QiaoLing Liu, Jie Zhang, HaoFen Wang, Yue Pan, and Yong Yu. 2007. Semplore: An IR Approach to Scalable Hybrid Query of Semantic Web Data. In *Proceedings of the 6th International The Semantic Web and 2Nd Asian Conference on Asian Semantic Web Conference (ISWC'07/ASWC'07)*. Springer-Verlag, Berlin, Heidelberg, 652–665. http://dl.acm.org/citation.cfm?id=1785162.1785210