Powerful and Efficient Bulk Shortest-Path Queries: Cypher language extension & Giraph implementation

Peter Rutgers, Claudio Martella, Spyros Voulgaris, Peter Boncz
VU University Amsterdam, The Netherlands

ABSTRACT

Shortest-path computation is central to many graph queries. However, current graph-processing platforms tend to offer limited solutions, typically supporting only single-source and all-pairs shortest path algorithms, with poor filtering options. In this paper we address the shortest-path computation problem in two complementary directions. First, we introduce a restrictable, top-N "bulk" shortest-weighted-paths operator in the Cypher graph query language, that subsumes all previously known shortest path variants. In addition to ease of use, both in terms of short notation and more robust performance thanks to guaranteed amenability to pruning, this operator supports calculated path weights, as well as filtering on the path edges and vertices. Second, we provide a scalable algorithm for the parallel implementation of this top-N operator on Giraph, a graph-processing system based on the Bulk Synchronous Parallel (BSP) model. We present an initial evaluation on a number of queries executed over the LDBC-SNB dataset.

1. INTRODUCTION

The popularity of modeling connected data with graphs, with social, transportation, and knowledge networks as few examples, is growing both in industry and academia. With wide adoption comes a number of disparate workloads that range from graph pattern matching to graph analytics. In recent years, a number of graph-processing platforms have been introduced for scalable computation of such workloads on commodity machines. Examples of these platforms are so-called graph databases like Neo4j and OrientDB, and batch processing systems like Pregel [7], Giraph [2], and GraphX [3]. Graph databases usually offer a high-level query language to express graph patterns and aggregations (e.g., Neo4j offers a language called *Cypher* while OrientDB can be queried with a SQL-like language with proprietary extensions for path-like queries), while batch processing systems expose a simplified graph-specific programming interface, usually following a so-called vertex-centric paradigm. Workloads on these platforms tend to have different expected latencies that vary from milliseconds to hours or sometimes days. More recently, hybrid systems like PGX [4] offer both a high-level language and a low-level programming API, and target both interactive and analytical workloads.

Central to the analysis of graphs is the computation of shortest paths. As such, it is crucial for query languages to provide simple but expressive and integrated functions to define shortest paths as part of complex graph patterns, and for graph-processing platforms to be able to execute such queries on graphs of massive scale. However, we found that current graph query languages, like Cypher and SPARQL, lack such features, and that scalable parallel solutions for shortest paths computations are limited to the traditional single-source and all-pairs shortest paths algorithms designed to target high-end hardware and supercomputers.

Contributions. In this paper, we propose an operator to compute (restricted) top-N shortest-weighted-paths queries between any combination of source and destination vertices. This "bulk" definition subsumes known definitions of the single-path, single-source and all-pairs shortest path problems. Our operator supports filters of edges and vertices on the paths based on labels and properties, and it requires a monotonic cost function to evaluate path costs as well as path-independent filters on edges and vertices. Moreover, we present a parallel implementation of the operator based the Bulk Synchronous Parallel (BSP) model [12]. We integrate our operator in the Cypher query language, due to the popularity of the open-source Neo4j graph database, and implement our prototype on top of Giraph, an open-source implementation of Pregel running on Hadoop clusters.

This paper is organized as follows. First, we provide an overview of the existing literature related scalable shortest paths computation. After we propose our new operator within the Cypher language, we present its implementation on top of Giraph. Then, we present an initial evaluation of the operator on a number of queries executed over the LDBC-SNB dataset. Finally, we conclude with a discussion about limitations and opportunities for future work.

2. RELATED WORK

Shortest path problems are discussed in the paper presenting Pregel [7]. Three variants are mentioned: s-t shortest path (i.e. fixed destination), single-source, and all pairs. Because the last yields a $\Theta(V^2)$ space complexity, it is not discussed. The first one, s-t, is considered a quite easy problem; experiments by Lumsdaine et al. [5] showed that only a small part of the vertices are visited before finding the shortest path. Only for the single-source variant an imple-

© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

mentation was provided for Pregel, a parallel version of the Bellman-Ford algorithm [7]. They note that this parallel (BSP) implementation performs more comparisons than sequential versions (Dijkstra and Bellman-Ford), but is also much more scalable. More advanced parallel versions do exist: for this they refer to Thorup [11] and the Delta-stepping method by Meyers and Sanders [9]. The Delta-stepping algorithm is designed for parallel computing, running in linear average time for most graphs. It uses buckets with a fixed width (Δ), containing the vertices to evaluate in each iteration. Edges with a small weight ($<\Delta$) are traversed in the same iteration, while edges with a large weight ($>\Delta$) are postponed to later iterations. By setting Δ very high, the algorithm behaves like Bellman-Ford; while by setting Δ very low, it behaves like Dijkstra. The best performance is obtained somewhere in between, where the best Δ depends on the edge weight distribution. Madduri et al. [6] applied this algorithm on billions of vertices and edges and found almost linear speedup on the Cray MTA-2.

Wang et al. [13] note that usually single-source or all pairs algorithms are used to solve the *multiple-pairs* shortest path problem. Each single-source run can be used to calculate all pairs with the same source or the same destination. To calculate the minimum number of such single-source runs, the minimum vertex cover can be calculated on a bipartite graph representing the pairs. They present a new algorithm that first applies a preprocessing step, determining an order to process the vertices based on the graph structure. Therefore, they consider their algorithm especially useful when the graph is fixed, but edge costs change between runs. In an experiment using grid graphs, the algorithm turns out to be comparable to other optimized implementations of Dijkstra and others. We call approaches that compute the multiple-pairs problem in a way that seeks to create synergy between the work **bulk** shortest paths algorithms, and note that graph query languages with path-finding functionality typically set the stage for bulk path-finding, which thus poses an interesting algorithmic research area.

Regarding distributed algorithms, two important algorithms for routing are Chandy-Misra and Merlin-Segall [8]. Both are used for a single-source shortest path computation, where each node in the network finds the length of the shortest path to the source and the first hop in the shortest path. However, Chandy-Misra has exponential message complexity in the worst case, while Merlin-Segall has a message complexity of $\Theta(N^2\Delta E)$. For very large graphs, this will probably generate too many messages to perform well.

Neo4j supports searching for all paths, all simple paths, shortest paths, paths with fewest hops, or the paths with a fixed number of hops. For cheapest path calculation, two algorithms are available: A* and Dijkstra. The fewest hops calculation uses breadth-first search from the source and the destination, alternating between exploring a level forward from the source and a level backward from the destination. These algorithms can be called from Cypher or by the REST API of Neo4J. The A* implementation of shortest path can only be called from the Java API, after implementing an estimation heuristic. To calculate shortest paths from many source vertices, Neo4j implemented the Floyd-Warshall algorithm for all pairs shortest paths, though this does not scale to larger graphs and is therefore no longer used. Returning the top-N shortest paths is possible only with the non-scalable all paths algorithm.

3. A FIRST-CLASS-CITIZEN SHORTEST-PATHS OPERATOR FOR CYPHER

Calculating weighted shortest paths in Neo4j/Cypher is possible, but it is not efficient. Variable length paths allow for queries such as calculating all cities within 200 km, but only by matching all paths towards cities and filtering them on total length. The language does not support general recursion or calculating fixed points.

Requirements. We identified three related features that would be useful in a shortest-paths operator. They are listed in order of increasing expressiveness, where each feature is a more general version of the previous one.
1. Paths of variable length, also known as reachability queries or relationship closure
2. Shortest path queries, with (i) conditions on vertices and edges within a path (ii) support both the shortest path length and the path itself as result (iii) support multiple shortest paths (top-N) (iv) efficient calculation of multiple-source, multiple-destination paths
3. Support recursive queries using an inflationary fixed point

Current situation. Cypher provides two ways to specify a shortest path query. The first one is based on the number of hops, and does not support edge weights nor a top-N feature. In addition, the only way to apply conditions is in the `WHERE` clause, potentially removing the only result from the query. Filtering is applied after finding the shortest path, not before, so it is not possible to find the shortest path over a specified subset of the graph. For example, the query below will first find the shortest path from `Start` to `Finish`. If a node on that path then has a property called "danger", the query returns an empty set.

```
MATCH p=shortestPath((a:Start)-[:ROAD_TO*]->(b:Finish))
WHERE ALL (x in nodes(p) WHERE NOT x.danger)
RETURN p, length(p)
```

To work around the restrictions of the built-in `shortestPath` function, it is common to apply a combination of the reduce function and the `ORDER BY / LIMIT` clauses. In this way, it is at least possible to express a weighted shortest path, and to apply a condition before choosing the path:

```
MATCH p=(a:Start)-[e:ROAD_TO*]->(b:Finish)
WHERE ALL (x in nodes(p) WHERE NOT x.danger)
RETURN p, reduce(time=0, r IN e | time +
    (r.distance/r.maxSpeed)) AS TotalTime
ORDER BY TotalTime ASC LIMIT 5
```

The addition of `LIMIT 5`, makes the query return the top 5 shortest paths, instead of just one.

The problem here is that the above shortest-path formulation is a complex interplay between multiple language elements, and it will be hard for the query optimizer to identify this as a shortest path query that may be executed with an efficient algorithm. If the system is unable to recognize such opportunity, it would have to enumerate all paths and calculate all their lengths to decide which is shortest. This has exponential complexity and will not terminate in reasonable time for large graphs. We believe that only supporting efficient algorithms as a special case for certain queries will result in a bad user experience. Small changes to the query will lead to it no longer returning an answer. For instance, changing the keyword `ALL` into `ANY` or `SINGLE` would already be incompatible with an efficient shortest path algorithm.

Proposal. Leading to our proposal, we designed and evaluated a number of alternative extensions. We present them in

Appendix A. Compared to those alternatives, our proposal corresponds closest to the first extension for shortest paths. All of the new syntax was placed in the `MATCH` part of the query, like the existing `shortestPath` function. Because it is more powerful than a regular function, and requires more complex arguments, the syntax is different and separated from the `shortestPath` function.

```
MATCH path=src-[e*|sel(e)]->dst CHEAPEST n SUM cost(e) AS d
```

There are four arguments in this extension of `MATCH`, marked in italics and described below. All of these are optional, except for the *cost(e)* expression.

1. First, between square brackets after the `|`, there may be a boolean condition *sel(e)* on the edges e in the path, which is applied before evaluating the `WHERE` condition. This is an important difference, as it enables filtering without reducing the number of paths found. Applying a filter afterwards, in the `WHERE` clause, may result in not having any result at all. This mechanism can also be used to apply a filter on the incident vertices, accessing them via the existing Cypher functions `startNode(`e`)` and `endNode(`e`)`.

2. The second argument is the number of alternative paths for each pair n. This is different from a `LIMIT` clause, which is evaluated on the full set of results. If it is omitted, only the single best path is found for each pair. The number of paths should always be exactly this amount, unless not enough paths exist. In the case more than one path per pair is requested, we restrict the problem to returning non-cyclic paths only (simple paths).

3. The `CHEAPEST SUM`, constructs a (monotonic) cost function by computing for each traversed edge e a numeric value *cost(e)* that must be ≥ 0. The ≥ 0 requirement may be enforced at runtime. We call it `CHEAPEST` to distinguish from the `shortestPath` function, which only counts the number of hops. It is still possible to achieve the same result by specifying a constant *cost(e)* of e.g. 1, but much more complex calculations are possible as well. We denote the cost expression as a function of edge variable e, similar to the way filter conditions were defined above. The distance d of *path* is $\sum_{e \in path} cost(e)$, the sum of all values resulting from evaluating the cost expression over each edge e in *path*.

4. Finally, a method is desired to bind the distance d found to a variable. Although it is possible in Cypher to find this distance again using the reduce function on the path, adding `AS d` is clearly preferable for usability.

Note that the enclosing Cypher query provides (potentially) multiple binding combinations for (src,dst) so we can search an **arbitrary set** of source and destination combinations, not only single-single (s-t), single-multiple (single-source), or Cartesian product of some vertex set (all pairs). This sets the stage for **bulk shortest path** algorithms that try to create execution synergy between shortest path finding for each combination.

Thanks to the monotonic cost metric, and the fact that filters are independent of path-finding (can be applied before), each possible query instance is amenable to pruning techniques; such that "performance cliffs" where certain query instances are fast but others never terminate, should be avoidable – creating a good user experience.

Example. Let us assume a graph where some vertices are named "Start", and some others are named "Finish". Additionally, the graph contains some vertices marked as "Danger" that should be avoided. For each pair of Start and End vertices, we want to find their top 3 fastest routes without crossing dangerous points. The time spent on a road is the distance divided by the speed limit. This problem can be expressed in the following query.

```
MATCH path=(a:Start)-[e*|not(endNode(e).danger)]->(b:Finish)
CHEAPEST 3 SUM e.distance / e.maxSpeed AS length
RETURN a, b, path, length
```

4. GIRAPH IMPLEMENTATION

Our work is part of the **Lighthouse** system that implements a Cypher-like graph query language on top of Giraph. Hence, we implemented our shortest path operator following the Pregel model. In this so-called *vertex-centric* model, the developer expresses a computation from the perspective of a vertex receiving and sending messages to other vertices, and updating its associated value and edges. A Giraph computation is executed in a number of supersteps, where the messages sent at superstep i are delivered to the destination at superstep $i+1$. The data-model is a directed graph where edges are attached to their source vertices and both vertices and edges can have arbitrary values (e.g., used for weights, key-value properties, and labels). Vertices can vote to halt, meaning that the user-defined function (UDF) will not be executed on those vertices during the following superstep, unless they have new messages in their "inbox". Giraph follows a master-worker architecture, where vertices are partitioned across a number of worker machines. Workers execute the UDF on their vertices and exchange messages directly over the network, while the master coordinates the different phases of each superstep.

The core of the algorithm follows the structure of a Pregel Bellman-Ford single-source shortest-path distance computation. A computation starts from the source vertex, which sends a message through its outgoing edges containing the edge weight (or a hop count of 1, if no weights are used). Each time a vertex receives a message from one of its predecessors, it checks whether the new distance from the source is shorter than the current. If that is the case, the vertex updates its distance and propagates it to its neighbors through its outgoing edges (adding the respective edge weight to the distance), otherwise it ignores the update. The computation is concluded when no messages are transient.

Basic algorithm. As we want to compute distances *and* paths, as well as control the number of top-N paths, we need to use a different data-structure associated to each vertex than the sole distance from the source. For each source, each vertex needs to store a sorted list of distances with corresponding paths.

Messages between vertices have the same structure as the values stored by each vertex. They contain the new or improved paths found by the vertex sending the message. In each iteration, a vertex creates a new map to store all improvements found and then processes each received message. For each source described in the message, the (sorted) list of paths is merged with the list of the top-N shortest paths found so far. This results in a new list of shortest paths that is stored in the vertex value. Each occurrence of a received path in this new list is then an improvement to the old list. Therefore, it is added to the map of improvements. At the end of the iteration, a modified version of the map is built for each outgoing edge. For each outgoing edge e to some target vertex v, all distances in the map are in-

creased according to the cost expression for e, and all paths are updated by adding v at the end. Each modified version of the map is then sent to the target vertex v. If in a certain iteration a vertex did not find any improved path, the map of improvements is empty. In that case, nothing is sent to neighboring vertices. Instead, the vertex votes to halt the computation. If all vertices vote to halt and no messages were sent, all shortest paths have been found and are present in the vertex values.

Postponing exploration. Inspired by the Delta-stepping algorithm, an improved version of the algorithm was implemented to reduce the exponential growth of exploration in the first few supersteps. The main idea is to have each vertex store its paths to the source in a number of "buckets", depending on the length of the path. For example, bucket 2 would contain all paths with length between Δ and $2 * \Delta$. All paths in bucket i are not broadcasted to neighbors until superstep i. To find a good setting for Δ, one could use statistics about edge weights and/or properties computed before-hand. Note that this method is not equivalent to the Delta-stepping method, where the outgoing edges are divided into buckets based on their edge weights.

A multi-phase approach. A downside of the algorithm described so far is its large memory footprint, because the vertex-lists that are passed around can be long. Instead of sending lists for each path, we want to represent each path only by the vertex ID of the predecessor. When the exploration is finished and all calculated paths are fixed, the paths should be reconstructed from the references to predecessors. This way, we reduce the amount of data to be exchanged and stored in memory during exploration, at the cost of more supersteps. To reconstruct the paths, there are two main possible approaches: building paths starting at the source or in reverse direction by starting at the end. The latter approach has two advantages. First, we can forward directly to the following vertex in the path, because the predecessors are known locally. Second, all the paths with an endpoint that is not selected as destination can be ignored immediately. There is also a disadvantage: when having one source and many destinations, all paths from the source will end up at a single vertex, potentially resulting in low scalability. However, path reconstruction should in general be a small part of the total work, once distances have been discovered. Therefore, we have chosen to reconstruct paths starting from the destinations, back to the source vertices.

The basic idea of the algorithm is to route the total distances corresponding to paths from the destination to the source, using the predecessor references as a routing table, and appending vertex IDs along the way. In order to support top-N paths, it is not possible to use a regular routing table, since we need to restore the sub-optimal paths as well. Therefore a routing table is used that also contains up to N entries for each source. In this way, it is possible to reconstruct multiple different paths from a destination, even when the destination has no information to distinguish them. This routing table is the result of the shortest path exploration, and stored in the vertex value.

3-phase approach. A further optimization is to divide the first phase into two parts, resulting in a 3-phase algorithm. The goal of the first step is then to find the minimum amount of data from which the shortest paths can be reconstructed. Since the predecessors can be inferred from the shortest distances, they do not need to be tracked during the exploration. When the distances of all top-N shortest paths are fixed, each vertex can find the predecessor corresponding to each distance by comparing the distances of its neighbors.

Using message combiners. Combiners are features of the Pregel model that can be used to reduce messages by combining data to the same destination on the sender (worker) side. It is noted that they can only be used for commutative and associative operations, since the order and grouping of messages is unspecified. Messages at the same worker with the same recipient are combined into one, before they are sent to the recipient. If we want to keep track of the paths, instead of just finding distances, it will be necessary to first duplicate some information. Without using combiners, a message can be "signed" by the sender: by including its vertex ID once in a message, the recipient knows the predecessor for all distances within the message. This information about predecessors is eventually used to reconstruct the paths. With combiners, some messages are combined into one before reaching the recipient, removing the possibility for signed messages. Instead, each distance needs to be signed individually. A combined distance list could then contain distances belonging to different predecessors. In case of two top-N lists for the same source vertex and recipient, the two sorted lists are merged into one sorted list. The merge operation makes a linear-time pass over both lists and combines them into a new list, keeping at most N items.

Pruning using landmarks. If shortest-paths distance estimates are available as upper bounds, they can be also used in pruning the search for the exact shortest path. One method to find such upper bounds is to define a set of landmarks as a subset of the vertices in the graph. Instead of searching for the shortest path for each pair, only paths *from* or *to* a landmark are searched. An upper bound for an arbitrary pair (A, B) can then be found by iterating over the landmarks L and finding the shortest path $A \rightarrow L \rightarrow B$.

In a bulk shortest path query extension, with query-dependent weights, it may be unrealistic to assume that landmarks would already be available prior to query execution, as an index structure. Still, if there are many (src,dst) pairs in the bulk shortest path operator, it can be worth computing landmarks first, or better, to use earlier computed (src,dst) pairs as landmarks for paths computed thereafter.

A recent all-pairs shortest paths (APSP) algorithm using landmarks was presented by Akiba et al. [1]. We have explored the possibilities to adapt this algorithm for use in the BSP model of Giraph, while using a combination of extensions for top-N results, directed edges, weighted paths and path queries (instead of only distances). The authors note that the algorithm would perform especially well in a BSP setting, since it is based on a breadth-first search where pruning is done locally. However, the extensions introduced by our operator cause a number of complications. First, regarding the extension for weighted edges, it is suggested to replace breadth-first search by sequential applications of Dijkstra's algorithm. It is however not feasible to run N instances of Dijkstra in parallel, with N being the number of vertices. Instead, we select a number of vertices to be landmarks, and run a Floyd-Warshall algorithm for each landmark in parallel. In a second phase, we can prune all

following searches based on the results found for the landmarks. Another challenge in parallelizing the algorithm is to efficiently prune the search, based on the distances found via landmarks. In the BSP setting using weighted paths, this means each (src,dst) pair needs to exchange their distances to each landmark. In order to do this efficiently, we use aggregation to send the best distances from each source to each landmark. Then, each destination can access the aggregated result, removing the need to send a copy of the same information to each destination vertex. Additionally, pruning is more difficult when using top-N results. Since paths are stored distributively, it is not possible to determine the top-N paths from the top-N paths to the landmarks.

Hence, we use a relaxed form of pruning, using 3 rules:
1. If N paths $A \rightarrow V$ have length $\leq X$ and the best path $V \rightarrow B$ has length Y , then at least N paths $A \rightarrow B$ have length $\leq X + Y$.
2. If the best path $A \rightarrow V$ has length X and N paths $V \rightarrow B$ have length $\leq Y$, then at least N paths $A \rightarrow B$ have length $\leq X + Y$.
3. If there are N different distances $d \leq D$ where, for some V, $A \rightarrow V \rightarrow B$ has distance d; then at least N paths $A \rightarrow B$ have length $l \leq D$.

Finally, the vertices of the graph should be ordered by degree, since this has been shown to improve performance by an order of magnitude [1]. To order vertices distributively, we use an aggregator that keeps track of the top-K vertices with highest degree.

In summary, the Giraph algorithm has the following steps:
1. Initialization of sources, destinations, excluded paths; and aggregating vertex degrees.
2. Running extended Floyd-Warshall from landmarks.
3. Reversing all edges in the graph.
4. Running extended Floyd-Warshall to landmarks.
5. Reversing the edges back to original.
6. Exchanging paths from/to landmarks and calculating all upper bounds for pruning.
7. Running extended Floyd-Warshall for all sources, pruning where possible.
8. Reconstructing distributed paths from predecessors.

5. EVALUATION

We performed an initial small-scale evaluation of our Giraph implementation of the shortest-paths algorithm on graphs generated with the LDBC SNB generator [10] with scale-factor 1 (SF1) with 10,993 vertices and 451,522 edges, scale-factor 10 (SF10) with 72949 vertices and 4,641,430 edges, and with an Erdős-Rényi random graph (Rnd1K) with 1,000 vertices and 50,000 edges (hence each vertex had 50 edges). For the LDBC SNB graphs, the elapsed time since the friendship edge creation time was used as edge weight, while for the random graph we chose edge weights uniformly at random in the $[0, 1]$ range.

The evaluation was performed running Apache Giraph version 1.2 on a cluster of 8 machines with 8 cores and 64GB of RAM each. The data was stored in HDFS and loaded into main memory by Giraph at the beginning of each computation, and the resulting paths and distances were written back to HDFS at the end of the computation.

We executed the basic algorithm on the SF10 graph by varying the number of workers exponentially from 1 to 32 by choosing a source vertex at random and computing shortest-paths to all other vertices as destinations. Table 1 presents

No. workers	1	2	4	8	16	32
Total computation s.	> 1k	492	222	126	89	72
Paths computation s.		10	7	5	6	5

Table 1: Runtime in seconds of the shortest-paths computations on SF10 graph starting from a random source vertex to all other vertices as destinations.

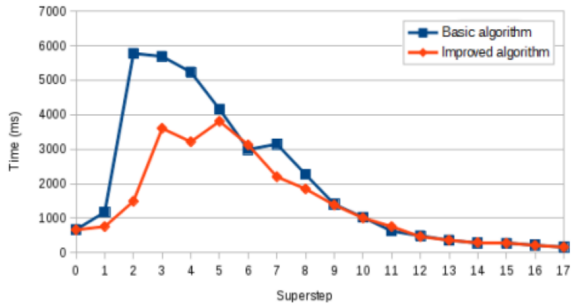

Figure 1: Impact of postponing path exploration based on the lengths of the paths discovered so far.

the wall-clock runtime of each computation on the SF10 graph. Note that the computation scales with the number of workers. The reported numbers for *Total computation* include Giraph start-up, input, and output phases.

We further evaluated the impact of the Delta-stepping expired exploration postponing on the Rnd1K graph. One vertex is selected as source, and the top-5 shortest paths are calculated from the source to each vertex. In the improved version, parameter Delta is set to 0.5. Computation is divided over three workers. Figure 1 shows the per-superstep runtime of one of such computations. Clearly, there is a significant difference in runtime and communication. Especially in early supersteps, a lot of unnecessary communication is prevented. Total runtime decreases from 35 seconds to 25 seconds with 49% less data being sent overall.

Next, we evaluated the impact of the multi-phase approach. First we compared the 2-phase algorithm to the regular shortest path computation. Afterwards, the Delta-stepping improvement was applied to the 2-phase algorithm, to evaluate if it still leads to a significant reduction in communication. In this experiment we used dataset Rnd1K. Table 3 shows that the path reconstruction phase requires less communication than the rest of the shortest path algorithm. Building the paths afterwards reduces communication by more than 50%. In the second experiment, the combination of path reconstruction and Delta-stepping was tested, and 1000 destinations were selected. Clearly, the large reduction in communication holds even when using the algorithm using Delta-stepping. This was to be expected, since Delta-stepping reduces the amount of paths explored, while storing paths distributively reduces the amount of communication for each path. In this graph, the optimal parameter setting is around 0.15. When we applied the 3-phase we noticed a further 8% decrease in communication, without relevant improvement in total runtime. In other words, the time saved exchanging data was spent during the additional superstep.

Regarding the application of message combiners, we found that combiners in most cases improve the runtime of the algorithm (in our test, up to 28%), even when the amount of data prior to combination needs to be doubled in order to use them. However, when the ratio between vertices' degree

	GBytes	Messages	Supersteps	Total
Basic	79.9	402628	18	27.1 s.
2-stage (5 dst)	80	402749	18+10	25.1 s.
2-stage (1k dst)	82.4	408203	18+17	33.0 s.

Table 2: Impact on communication (data and messages), number of supersteps and runtime of the 2-phase algorithm.

	GBytes	Messages	Supersteps	Total
Basic ($\Delta = .5$)	88.3	337032	18	25.0 s.
2-phase ($\Delta = .5$)	47.6	342607	18+17	33.6 s.
2-phase ($\Delta = .25$)	35.9	339171	19+17	30.3 s.
2-phase ($\Delta = .15$)	31.2	395713	25+17	29.9 s.
2-phase ($\Delta = .1$)	32.6	502921	36+17	30.8 s.

Table 3: Impact on communication (data and messages), number of supersteps and runtime of the different values of Δ.

and the number of workers becomes low, there is little to combine within a worker. This causes the non-combined configuration to be more scalable than the combined version.

Finally, the impact of landmarks was evaluated compared to the two-phase algorithm alone. We used the SF1 graph, selecting a random subset of 25 sources, and calculating the top-5 shortest paths from the sources to each vertex in the graph. When using landmarks, the 2 vertices of highest degree were selected to be landmarks. The path reconstruction phase is excluded from this experiment, as the modifications do not affect this part.

In Figure 2 we can see that the actual computation is pruned significantly, reducing the search time by 40% to 50%. However, in the current setting, the total computation is not faster at all. In order to use landmarks as an effective optimization, the number of sources should probably be higher than the 25 vertices selected. In addition, the runtime of constructing the landmark tables seems quite high, considering that this phase is using the same algorithm, only with a much lower number of sources. Runtime could probably be improved by calculating the two phases computing paths to and from landmarks in parallel.

6. DISCUSSION AND CONCLUSIONS

This work was done in the context of the Lighthouse system, which combines Giraph's power with the expressiveness of Cypher, to allow easy-to-write pattern matching and aggregation queries be performed efficiently at large scale.

In this paper, we specifically defined a new graph query language operator for shortest paths that allows for computation of paths, with (i) flexible weights computed over edge and vertex properties, (ii) flexible filter conditions over these, that (iii) can return not only the best, but also top-N best paths. These three features are relevant for users but unavailable in the existing graph query languages SPARQL and Cypher. The syntax for this operator is concise, and importantly it guarantees that efficient (pruning) algorithms can be used. This feature distinguishes it from recursive SQL queries, which are flexible, but systems implementing these must generate all possible paths and filter from these afterwards – an approach with exponential complexity that does not scale on large and complex real-world graphs.

We also explored a number of parallel algorithms for bulk

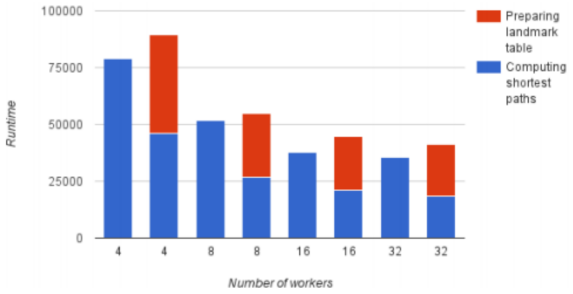

Figure 2: Runtime of shortest-paths computation with and without landmarks computations.

shortest path exploration and presented early evaluation results. These algorithms and experiments, while encouraging and interesting, are work-in-progress. Generally speaking, our operator provides opportunity for **bulk** shortest path computation, where algorithms can find synergy in the task of finding paths between many (src,dst) combinations – this we think is a relevant and promising area of future research.

While some of the proposed changes drastically reduce the number and size of messages, they do not necessarily reduce runtime. The significance of these optimization should not be undermined, as memory footprint for an in-memory analytics system constitutes a key limitation.

Finally, the computation of landmarks incurs an overhead that is not always beneficiary to the overall execution time, notably when the graph has dynamically changing properties and topology, requiring the recomputation of landmarks for each query. An automated prediction of *when* the computation of landmarks can reduce the total runtime is another future step to complement our research.

7. REFERENCES

[1] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *ACM SIGMOD*, 2013.

[2] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: graph processing at facebook-scale. *VLDB*, 2015.

[3] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[4] S. Hong, S. Depner, T. Manhardt, J. Van Der Lugt, M. Verstraaten, and H. Chafi. Pgx. d: a fast distributed graph processing engine. In *Supercomputing Conference (SC)*, 2015.

[5] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 2007.

[6] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *Proceedings of the Meeting on Algorithm Engineering & Experiments*, 2007.

[7] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *ACM SIGMOD*, 2010.

[8] P. M. Merlin and A. Segall. A failsafe distributed routing protocol. *Communications, IEEE Transactions on*, 1979.

[9] U. Meyer and P. Sanders. δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 2003.

[10] A. Prat-Pérez, D. Dominguez-Sal, and J. L. Larriba-Pey. Social based layouts for the increase of locality in graph operations. In *Database Systems for Advanced Applications*, 2011.

[11] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *JACM*, 1999.

[12] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 1990.

[13] I.-L. Wang, E. L. Johnson, and J. S. Sokol. A multiple pairs shortest path algorithm. *Transportation science*, 2005.

APPENDIX

A. OTHER POSSIBLE EXTENSIONS

For the three features described in Section 3, a number of alternative language extensions were explored and evaluated that lead to the proposal. For the first feature, variable length paths, the syntax is already provided in Cypher: `(a)-[r * 4..6]-(b)`. It allows for minimum and maximum length, both of which are optional. We now focus on the language extensions needed for the other two: shortest paths and recursion.

A.1 Shortest-paths

For shortest paths, two alternatives are evaluated: one extending the existing `shortestPath` function with additional arguments for the weight function, conditions, and number of top N paths, and one alternative adding new aggregation functions. The first has the advantage that it does not require a new function, it just adds optional arguments that extend the function in a relatively natural way (new syntax in italics).

```
MATCH p=shortestPath((a:Start)-[r:ROAD_TO*]->(b:Finish),
x in r | x.distance/x.maxSpeed,
ALL y IN nodes(p) WHERE NOT y.danger)
RETURN p
```

If no cost function is given, the cost of each edge is considered to be equal, just like in the original function. For the cost function, we can use the same syntax as used by the reduce method of Cypher, so again little new syntax is required. The same holds for the third argument, which is a condition that would be moved from the `WHERE` clause to within the function. This provides a clear distinction between filtering before and after finding shortest paths.

There are also a number of downsides to this approach. First of all, it would be preferable if an existing Cypher feature could be used. Second, the user is forced to decide whether to use a shortest path algorithm, instead of leaving this choice to the query optimizer. Finally, a natural way to add the number of pairs (top-N) is lacking, as is a way to easily refer to the distance of a path.

An alternative syntax extension is to implement shortest paths as an aggregation function. This is used in Cypher to find aggregated results over some result, such as minimum, sum, count, and more complex calculations such as arbitrary percentiles. In this variant, the "minimum" aggregation function would be combined with the reduce function into a new shortest function that operates on paths:

```
RETURN startNode, endNode,
shortest(path, x IN x.distance / x.maxSpeed, 5),
cost(path, x IN x.distance / x.maxSpeed)
```

While this syntax would be consistent with other aggregation functions, this variant would not completely solve the problem for the query optimizer: it is still possible to write complex conditions in the `WHERE` clause, making it impossible to run a shortest path algorithm. Then the query optimizer could decide to use the brute-force option of the reduce/minimum functions as a fallback method, but this decision would be complex to implement. It would have to evaluate if the path is subject to any conditions that can not be satisfied by excluding vertices or edges from the graph over which the algorithm is run.

A.2 Recursion

Inspired by the approach of SQL:99, a corresponding syntax is suggested for Cypher. This has a number of benefits: it is much more expressive than the other options, enabling variable length paths (closure) and shortest path queries. The change in syntax compared to the existing Cypher is relatively minor and one powerful mechanism could avoid the need for many different extensions to implement specific algorithms. However, it is both quite hard to optimize and hard to express shortest path queries with the desired functionality. Also, Cypher already has other syntax for most simple variable length paths, so it may be more consistent to extend the existing variable-length syntax.

The following syntax illustrates how such a recursive query could be used to traverse a hierarchy of employees and managers in an organization. They are connected to their departments through `WORKS_AT` and `IS_HEAD_OF` edges, respectively.

```
MATCH (a)-[:IS_HEAD_OF]->(b)
WHERE NOT((a)-[:WORKS_AT]->())
RETURN a AS x, 1 AS n, NULL AS manager
UNION RECURSIVE
MATCH (x)-[:IS_HEAD_OF]->(:Department)<-[:WORKS_AT]-(y)
RETURN y AS x, n + 1 AS n, x AS manager
```

From a user perspective, it may be easier to write the recursive case in one clause, using the existing `OPTIONAL MATCH` or a new `RECURSIVE MATCH`. For example, compare the queries above and below. The disadvantage in this case is that the semantics are not very clear from the syntax, such as the distinction between variables for the base case and the recursive case. Additionally, it does not solve the other problems, such as the complications for the query optimizer. Therefore, the previous option would be preferred.