Extending SQL for Computing Shortest Paths

Dean De Leo
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
dleo@cwi.nl

Peter Boncz
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
P.Boncz@cwi.nl

ABSTRACT

Reachability and shortest paths are among two of the most common queries realized on graphs. While graph frameworks and property graph databases provide an extensive and convenient built-in support for these operations, it is still both clunky and inefficient to perform on standard SQL DBMSs. In this paper, we present an extension to the standard SQL language to compute both reachability predicates and many-to-many shortest path queries. We first describe a methodology to represent a directed graph starting from virtual table expressions. Second, we introduce a new type of operator to compute shortest paths on the given graph. Our semantic abides by the rules of operating with table expressions, ensuring that the property of the closure from the relational algebra is retained. Finally, we developed a prototype implementation of our extension on top of MonetDB, an existing open source relational DBMS. Our preliminary results still show that dynamically building our representation of the underlying graph overly dominates the query time. Currently, this cost can only be amortized when executing multiple shortest paths on the same graph.

ACM Reference format:
Dean De Leo and Peter Boncz. 2017. Extending SQL for Computing Shortest Paths. In *Proceedings of GRADES'17, Chicago, IL, USA, May 19, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3078447.3078457

1 INTRODUCTION

Reachability and shortest (or cheapest) paths are among two of common operations of interest to perform on a graph. There exists a broad range of domains where these naturally occur: analysis of social networks, routing in transportation networks, control flow optimization, internet protocol routing, and so on.

The two operations can be formally defined. Let $G(V, E)$ be a directed graph with $V = \{v_1, \ldots, v_{|V|}\}$ the set of *nodes* or *vertices* and E the set of *edges*, where $e_{ij} \in E$ represents a directed edge from v_i to v_j. We say a node $v_s \in V$ *reaches* another node $v_d \in V$ if there exists some path[1] p of finite length between the two nodes, i.e. $p = [e_{s,k_1}, e_{k_1,k_2}, \ldots, e_{k_m,d}]$. Shortest paths can be unweighted or weighted. An unweighted shortest path between two nodes v_s and v_d is a path between v_s and v_d having the minimum possible length. In a weighted shortest path, we associate to each edge e_{ij} a weight[2] $w_{ij} > 0$. Then, the weight $w(p)$ of a path $p = [e_{k_1,k_2}, e_{k_2,k_3}, \ldots e_{k_{n-1},k_n}]$ is the sum of its weights:

$$w(p) = \sum_{\forall e_{ij} \in p} w_{ij}$$

Therefore a weighted shortest path between two nodes v_s and v_d is a path between v_s and v_d with the minimum possible weight.

Reachability and shortest path queries can be cumbersome to perform in relational databases using standard SQL. First of all, it might not be clear how to express a graph or a path in SQL. A possibility would be to introduce new types and ad-hoc functions to operate on them and extract their content, an approach similar to XML documents. Here, we propose to model graphs and paths only relying on table expressions and primitive types. The fundamental benefit is that all operators and functions that already exist can also be further applied with no change.

Currently there are three customary means to perform reachability and shortest path queries in standard SQL: recursion, persistent stored modules (PSM) and, to a more limited extent, explicit chains of joins. The common idea of these methods is to mimic the Breadth-First Search (BFS) and Dijkstra algorithms. With recursion, starting from a source node v_s, each recursive step adds to the result set the neighbours of an unvisited node having the current minimum distance from v_s. The recursion stops when the destination node is found in the result set or there are no more nodes to explore. With PSM, the idea is to create temporary tables to maintain the data structures of BFS/Dijkstra and then use the procedural constructs to implement a shortest path algorithm. Finally, if the number of iterations can be limited by some number N, then a simple popular technique is, starting with a table T only containing the source node, execute N-1 self-joins to incrementally extend the result set with the neighbours of the nodes discovered at the previous step.

Regrettably, all the three aforementioned approaches have some unpleasant shortcomings. First, it can easily become quite verbose and error prone to write a correct query following these methods. Second, all the above approaches breach the *declarative* paradigm of SQL: they specify *how* to compute the result rather than *what* result is needed. Last, performance-wise, such approaches can be very slow either due to missed algorithmic opportunities (full search instead of Dijkstra), or due to interpretation overhead (PSM).

In this paper, we try to address the problem of reachability and shortest paths through a new extension to the SQL language. We hope that our proposed syntax is concise and rather intuitive to regular RDBMS users, while still being expressive enough for a wide range of applications. We introduce a new operator that adequately

[1] Note that, in this text a path is the sequence of *edges*, rather than *vertices*, that need to be traversed to reach the destination from its source.

[2] We refer to the terms *weight*, *cost* and *distance* as synonyms, freely interchanging them thereafter.

GRADES'17, Chicago, IL, USA

978-1-4503-5038-9/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3078447.3078457

integrates with the relational model, bridging the divergence between graphs and table expressions. Moreover, our approach opens the opportunity for the DBMS to support and choose optimized built-in algorithms to compute both reachability and shortest paths, reducing the performance gap with specialized graph frameworks.

We implemented a prototype of our extension for MonetDB. MonetDB [16] is a free and open-source column-store relational DBMS, mostly compliant with standard SQL-2003. In our experience, implementing the extension required a fair amount of changes in the existing code base for the compiler. However, MonetDB supports a *pluggable* architecture to extend the physical layer by external libraries. Using this mechanism, we were able to easily add a few shortest paths algorithms to execute the new operators. Note that our prototype is still not optimized and we can only present very preliminary results for the current state. We expect to significantly enhance the performance of our solution in the future.

The rest of the paper is organized as follows. In Section 2, we describe our extension to SQL to compute shortest path queries. Section 3 provides an overview of our implementation of the extension on top of MonetDB. In section 4 we show our preliminary evaluation results over some specific queries. We discuss related work in section 5 and conclude in section 6. Finally, the appendix shows some usage examples of the proposed extension.

2 SQL EXTENSION

We introduce an extension to the SQL language to compute reachability and shortest path queries over graphs. The basic concepts are summarized as follows. We represent a graph G by a single table expression T that indicates the edges of G. Its vertices are implicitly derived from the unique keys in T. To inquire whether two nodes are connected in the graph, we introduce a new join predicate that operates over T. Additionally, we propose a new summary function to compute shortest paths, its output is actually a pair containing the cost and the path found. Finally, we describe an operator to translate an arbitrary path into a table expression, so that all other regular constructs from SQL can be applied over.

A graph $G(V, E)$ is modeled starting from a single table expression T. Each tuple $t \in T$ represents a directed edge of the graph, so $E = T$. The table expression follows the conventional modeling of a many to many relationship to the same entity. For the sake of simplicity, we assume that a node in the graph can only be addressed by a single attribute; extending for multiple attributes is not complicated, though the notation becomes cumbersome. Thus, we define a single attribute S as the source of an edge and a second attribute D as its destination. Both attributes S and D must have the same type. It follows that, given a tuple $t \in T$, $t[S]$ and $t[D]$ refer to nodes in the graph, and the tuple t expresses a directed edge from $t[S]$ to $t[D]$. The table T can consistently comprise of further attributes to specify user properties attached to an edge. Due to the aforementioned characteristics, we conventionally name T as the *edge table*.

In a conventional many-to-many relationship, S and D should be foreign keys to some entity U, which logically characterizes the vertices of the graph. However, we do not explicitly mandate such requirement. The vertices of the graph are formally derived from the unique values of S and D, i.e. $V = T[S] \bigcup T[D]$. Nevertheless, generally users might want to attach additional properties to the vertices, besides to the edges. For this scope, an auxiliary table expression, say VP, is required and the nodes $v \in V$ should be foreign keys to VP.

To express reachability we propose to extend the SQL language with a new predicate. Given the graph $G(V, E)$ and two values x and y, we say the predicate $P(x, y, G(V, E))$ is satisfied for a pair $\langle x, y \rangle$ if the following condition holds:

$$\exists v', v'' \in V : x = v' \wedge y = v'' \\ \wedge \text{ exists a path p in } G(V, E) \text{ from } v' \text{ to } v'' \quad (1)$$

The idea is to augment SQL with a way to express P to select (filter) only the tuples satisfying the predicate. A user will state P using the following syntax inside the WHERE clause of a SQL block[3]:

```
SELECT VP.*
FROM VertexProperties VP
WHERE VP.X REACHES VP.Y OVER E EDGE (S, D)
```

The semantic is as follows. Let E be a table expression representing an edge table as described above, while S and D are the source and destination attributes, respectively. Then the vertices of the graph are given by $V = E.S \bigcup E.D$. Let VP be a table expression defining the attributes X and Y. The types for the attributes $E.S, E.D, VP.X, VP.Y$ must match, otherwise a semantic error arises. Then the result set of the whole block are all the tuples $t \in VP$ such that $P(t[X], t[Y], G(V, E))$ holds. In other words, the result is composed by all the tuples $t \in VP$ such that it exists a finite path from $t[X]$ and $t[Y]$ over $G(V, E)$. Naturally, the condition can be interpreted as a join predicate when X and Y refer to distinct table expressions, e.g. :

```
SELECT VP1.*, VP2.*
FROM VertexProp VP1, VertexProp VP2
WHERE VP1.X REACHES VP2.Y OVER E EDGE (S, D)
```

Furthermore we exploit the shown predicate to compute weighted shortest paths. Indeed, when the predicate is satisfied for a tuple t, we are then guaranteed that a path must exist between its source X and destination Y. At this stage, it is legitimate to ask what is indeed a shortest path and what it is its total cost. Our proposal is to admit a special function, `CHEAPEST SUM(e: expression)`, in the projection clause which refers to the reachability predicate using the tuple variable introduced there (`e` in this case). The whole syntax of a SQL block becomes:

```
SELECT VP1.*, VP2.*,
       CHEAPEST SUM(e: expr) AS cost
FROM   VertexProp VP1, VertexProp VP2
WHERE  VP1.X REACHES VP2.Y OVER E e EDGE (S, D)
```

The function takes as input two parameters: a tuple variable e and a numeric expression *expr*. The first parameter, e, serves as a mean to bind the function to the related edge table in the WHERE clause, and more generally, to the associated reachability predicate. While this explicit binding is not strictly needed when there is only one edge table in the WHERE clause, it is mandated when multiple reachability predicates are present and the expression has to be to unequivocally related to a specific edge table. The second parameter,

[3]The terms `REACHES`, `OVER` and `EDGE` become keywords of the language.

expr, is columnar expression to be evaluated in the context of the associated edge table *e*. It determines the weight to assign to each edge in the edge table. Its value must always be strictly greater than 0, otherwise a runtime exception is raised. Setting *expr* = 1 is equivalent to computing an unweighted shortest path, where the final cost is given by the number of hops in the path. While setting *expr* to an attribute *A* of the edge table means that *A* contains the weights for all edges. In general, *expr* can be any arbitrary columnar expression, its result is computed before executing `CHEAPEST SUM`.

To retrieve a shortest path, the function can return a second expression using the syntax `CHEAPEST SUM(e: expression) AS (cost, path)`. Note that, in general, multiple finite paths might exist having the minimum cost. In this scenario, the function always picks and returns one of the suitable alternatives among all the valid shortest paths. While the final type of the *cost* is based on the actual argument *expr*, i.e. a floating point expression implies that also *cost* will be a floating point, the type of the *path* is more intricate. From one side, the function has to return a single component per tuple to comply with the usual semantics of a projection operation. On the other hand, a path is logically a sequence of edges, which we might want to expand and manipulate as multiple components.

Our proposal is to represent a path as a *nested table*. A nested table is a special type that groups together multiple rows and columns into a single component. Each path consists of records with the columns *S* and *D* and, possibly, also additional edge property columns. While this proposal departs from the pure NF1 relational model, we note the SQL standard has already departed from there, with its support for collection (arrays and multisets) and row types [15]. Moreover, a number of newer SQL implementations, that can work on nested file formats such as ORC, Parquet or JSON, even provide support for querying multi-column and recursively nested data. These systems do not return arbitrarily nested data, and for SQL predicates for filtering, grouping/aggregation or joining to work, the data needs to be flattened, i.e. *unnested*.

The standard SQL way to unnest data is using the `UNNEST(expr)` operator. The operator can be invoked inside the `FROM` clause on array- or multiset-typed expressions, and this is usually done in the `FROM` clause using a so-called `LATERAL` join between the table expression containing the nested data and `UNNEST()` on that nested data. In a lateral join, a range variable introduced in a previous table expression in the `FROM` clause can be used as parameter to the operator `UNNEST()` appearing later in it. `UNNEST` can be used in combination with both inner and left outer joins. Specifically, this latter case is useful to preserve tuples when the nested structure is the "empty collection". Furthermore, the shortest admitted form of lateral joins actually omits the `LATERAL JOIN` keywords and just adds the `UNNEST()` separated by comma to the `FROM` clause, which implicitly denotes a lateral inner join (the default). Finally, after `UNNEST()`, SQL supports the optional `WITH ORDINALITY` clause which allows to add an extra integer column that provides a densely ascending *sequence number* with the expanded list (starting at 1).

In standard SQL, the elements in an array or a multiset can be rows. As a special rule, invoking `UNNEST` on collections of rows also flattens the fields composing the row. Effectively, a *nested table* can be mimicked in standard SQL as a collection of rows, expanded when demanded by the `UNNEST` operator.

Summarizing, `CHEAPEST SUM` is a function that can return one or two components. The first component is always the numerical *cost* of the shortest path. When a second identifier is explicitly given using the syntax `AS (ident1, ident2)`, the function returns a second component for the path, bound to the variable *ident*2. A path, represented with type nested table, is the sequence of edges that can be followed to link the source with its destination and whose cost is the minimal possible. The attributes enclosed in the nested table representing the path are the same as the attributes of the `EDGE` table expression corresponding to the `CHEAPEST SUM`. Eventually, a path can be expanded to its atomic components by the `UNNEST` operator.

A SQL block involving both paths and unnesting can be exemplified as follows:

```
SELECT T.X, T.Y, T.cost, R.S, R.D
FROM (
    SELECT VP1.*, VP2.*,
           CHEAPEST SUM(e: expr) AS (cost, path)
    FROM VertexProp VP1, VertexProp VP2
    WHERE VP1.X REACHES VP2.Y OVER E e EDGE (S, D)
) T, UNNEST(T.path) AS R
```

3 IMPLEMENTATION

A tentative prototype for our proposal has been implemented for MonetDB. Its main components are the changes to the SQL compiler, an external library implementing the BFS & Dijkstra algorithms and finally a new type *nested table*.

On a first approximation, MonetDB can be separated into three recognizable layers:

(1) the compiler or SQL front-end.
(2) the intermediate language, named *MonetDB Assembly Language* (MAL), and its interpreter.
(3) the physical layer.

Logically an incoming query is initially compiled by the SQL front-end and translated into a sequence of MAL instructions. The MAL layer performs a second optimization pass and generates the final physical plan, still in the form of MAL statements. The physical layer is responsible for the implementation of the core features, the storage, the main operators and the primitive types. Moreover, the MAL environment is pluggable, supplementary physical operators and types can be provided as external libraries and linked with the rest of the system through custom *MAL scripts*.

Currently our prototype exploits this *plug-in* feature to supply the runtime with additional operators. A consistent set of changes were required to the SQL front-end to describe the new semantics. A small set of algorithms (runtime), to compute a shortest path, are supplied as an external C++ library, avoiding any alteration to the existing MAL & physical layers. Finally, the *nested table* type was added to the kernel of MonetDB.

3.1 Compiler

The compiler was definitely the area of work that requested the most attention. In the SQL front-end, a query is represented in an abstract syntax tree (AST) where the nodes resemble the common operators or their variants from the relational algebra. Sample

existing operators are projection, group by, select (filter), join, semi-join... The AST was augmented with two more operators to capture the semantics of both reachability and shortest paths. These additions were justified by the need to intermix various elements of other operators: a reachability clause is similar to a *select* or *join* in terms of filtering out the tuples from a given table expression, however it also applies on a secondary table expression, the *edge table*. Moreover, it can also yield new expressions when computing shortest paths, similarly to what occurs with a *group by* or *projection*.

The first added operator was the *graph select*, based on the regular *select*. In both relational algebra and the MonetDB query model, a select operator $\sigma_P(T)$ takes as arguments a table expression T and one or more predicates P that denote conditions that must be satisfied. The result of the operator are the tuples $t \in T$ that satisfy $P(t)$. The operator graph select $\hat{\sigma}_{\bar{P}}(T, E)$ applies the reachability predicate $\bar{P}(X, Y, S, D)$, resembling (1), over the table expression T and the *edge table* E. The attributes X and Y define the sequence of source/destination vertices to filter out, while S and D are the source and destination keys for the edges in E. The semantic of $\hat{\sigma}$ is to first model a graph $G(V, E)$ according to the edges of E and the vertices $V = S \bigcup D$, then verify for each tuple $t \in T$ whether $t[X]$ is connected to $t[Y]$ over $G(V, E)$.

The second added operator was the *graph join*, based on the regular *join*. A regular join $\bowtie_P (T_1, T_2)$ is conceptually given by the sequence of a cross product $T = T_1 \times T_2$ followed by a select $\sigma_P(T)$. Similarly, a graph join $\hat{\bowtie}$ is defined as a cross product followed by a graph select: $\hat{\bowtie}_{\bar{P}}(T_1, T_2, E) = \hat{\sigma}_{\bar{P}}(T_1 \times T_2, E)$. The semantic stage of the compiler always creates a graph select when detecting a reachability predicate. Graph joins are only unfolded in the query rewriter when it recognizes the sequence of a cross product plus a graph select.

Shortest paths are expressions generated by a graph select or a graph joins. Their treatment is similar regardless to which operator they are attached. A complication with respect to ordinary selects and joins is that shortest paths create a new set of dependencies for graph selects/joins in the relational tree, which need to be respected in the rewriting rules of the optimiser.

In the *code generation* stage, the treatment of a graph select and a graph join is almost equivalent. Given $\bar{P}(X, Y, S, D)$, the set of vertices $V = S \bigcup D$ is indeed computed. The values from X and Y are then joined with V, performing an initial filtering on the values that are not vertices. Thereafter, the weights attached to each shortest path function are materialized. Regardless of their type, all the values from X, Y, S and D are translated into integers from the domain $H = \{0, \ldots, |V - 1|\}$. The external library, described in the next section, is then invoked, to filter out the connected vertices and compute the shortest paths. Eventually, the final result set is fully materialized back from the retrieved values.

Finally, a minor set of alterations involved the lexical analyzer and the parser. The terms `CHEAPEST`, `REACHES`, `EDGE` and `UNNEST` are now treated as keywords in the language. Further extensions to the parser and to the semantic phase regarded the recognition of the reachability predicate, the special function `CHEAPEST SUM '(' identifier: expression ')'` and the aliasing format `AS '(' identifier_list ')'` to refer to multiple variables generated from a single expression.

3.2 Runtime

The algorithms to compute the reachability and shortest paths have been implemented in a external library. It is dynamically linked with the rest of the system through a *MAL script*. The parameters to invoke the library are:

(1) the columns S and D, denoting the edges of the graph;
(2) the source X and destination Y vertices to filter;
(3) in case, the additional columns $W = \{W_1, \ldots, W_N\}$ for the weights associated to the edges.

The library returns the sequence of row ids t such that $t[S]$ is connected to $t[D]$ and the requested shortest paths to compute.

Our implementation always builds a Compressed Sparse Row (CSR) representation [17] of the underlying graph, somewhat resembling an adjacency list. The columns $\{S, D\} \bigcup W$ are sorted according to S, thus a prefix sum is computed on S itself. As the keys in S and D are the vertices from the domain $H = \{0, \ldots, |V-1|\}$, we exploit this property to address both the outgoing edges and their weights. Indeed, given a vertex id $\eta \in H$, all the outgoing edges of η are stored in D from the position $S[\eta - 1]$ up to the position $S[\eta] - 1$. The same property is also valid for the weights W. The library uses this data structure to compute the shortest paths.

Multiple shortest paths can be computed on the same graph. The library provides an implementation of the Breadth First Search (BFS) algorithm for unweighted shortest paths and the Dijkstra algorithm combined with the Radix Queue [11] for weighted shortest paths. When computing the first shortest path, tuples are also filtered (in case of the select semantic) or joined (in case of the join semantic). If the query only requires to assess the reachability predicate, without demanding to evaluate any shortest paths, the library still performs a BFS over the source and destination vertices, discarding the computed shortest paths.

3.3 Nested tables

As mentioned, we chose to represent paths as nested tables. However, in our current implementation a nested table is not a *collection of rows*, in the standard SQL sense, but a custom type. While it allows to fulfill the requirements of our proposal, several limitations exist. At the present time, it is not possible to explicitly create an attribute or a variable having this type, it can only be produced when computing a shortest path over a graph. Moreover, it cannot be permanently stored into a physical table, nor it can returned as it is to the connected client, but it has to be flattened before. The implementation provides the `UNNEST` operator to flatten the content of the nested table, though the clause `WITH ORDINALITY` is not supported yet.

At the physical layer, a nested table is represented as a list of references to the actual rows of the table expression that generated it. This is a handy solution because in the MonetDB execution model all intermediate results are fully materialized by its operators. Therefore, the rows composing a nested table can always be referred in a later stage. The `UNNEST` operator merely materializes the contained rows according to these references. Note that we

expect to change this representation in the future as we remove the current limitations for nested tables.

4 EXPERIMENTAL EVALUATION

We present an early evaluation of our implementation based on two queries specified in the LDBC SNB Interactive Workload benchmark [6, 13]. The benchmark is designed to model a social network, akin to Facebook. For our use case, our graph is derived by assuming the vertices are the users of the social network while the edges are their friendship relationships. We generated the data sets with the LDBC DATAGEN [5] using the scale factors 1, 3, 10, 30, 100 and 300. Table 1 shows the number of vertices and edges per scale factor. Note that the number of edges is actually double the amount of friendship relationships generated by the LDBC data generator, as relationships are undirected whereas our model assumes the graph is directed.

All our experiments were executed on a dual socket Intel Xeon e5-2650. The machine has 8 cores per socket, 16 cores, 32 physical threads and 256 GB of RAM in total. The server runs on top of Fedora 24 while GCC 6.3.1 was used to compile MonetDB. To run the queries, we employed a custom Java client placed on a different physical machine and interacting with the server through JDBC. The client and the server reside on a shared 1Gbit LAN network. The times reported in our experiments are based on the average measured latencies from the time the query is issued until the results are available back to the client. To test our implementation, we sequentially issued the same query with varying parameters 1000 times for scale factors SF 1 - 30 and 100 times for SF 100 and 300. The query parameters were randomly generated out of the set of the generated persons and according to a uniform distribution.

We evaluated Q13 and a simpler variant of Q14 from the set of the complex queries of the LDBC SNB Interactive Workload benchmark. Q13 determines the cost of the unweighted shortest paths between two given persons. We cannot perform Q14 as it is defined in the LDBC specification since it involves computing *all* shortest paths between two persons, while with our proposal we can only report one of them. However, for the purpose of testing weighted shortest paths, we considered an alternative of the same query. Each edge has attached a weight representing a measure of the *affinity* between the two friends, precomputed in the same manner described in the benchmark specification. In our variant, the query directly returns a weighted shortest path between two given persons using the affinities as weights.

Scale factor	Vertices $\times 10^3$	Edges $\times 10^3$
1	9.892	$\approx$ 362
3	$\approx$ 24	$\approx$ 1132
10	$\approx$ 65	$\approx$ 3894
30	$\approx$ 165	$\approx$ 12115
100	$\approx$ 448	$\approx$ 39998
300	$\approx$ 1128	$\approx$ 119225

Table 1: Size of the graph at different scale factors

Figure 1a) depicts the average measured latency per query. Between the two queries considered, there is a difference in the execution time of roughly 25% starting at the smallest scale factor SF and decreasing to 10% for larger SF. This result originates from the different algorithm chosen by the optimizer to compute the query. Currently, our implementation for weighted shortest paths relies on a more tuned radix queue under the hood, while, at this stage, our BFS implementation is still largely unoptimized. Nevertheless, we expect in the future to significantly improve the BFS implementation and to outperform the rather general Dijkstra algorithm.

The execution time is almost entirely dominated by the construction of the graph representation. Nevertheless, our extension is designed to compute multiple shortest paths over the same graph. The second experiment repeats the execution of Query 13, but grouping together multiple pairs ⟨source, destination⟩ at varying batch sizes. Figure 1b) shows the average computation time *per pair*, thus the measured latency divided by the batch size. In this case, the execution time decreases almost linearly and, for larger batch sizes, it finally amortizes the cost of constructing the underlying graph representation.

5 RELATED WORK

Graph database systems and some graph processing frameworks natively support several flavors of shortest paths queries. Some notable graph database systems include Neo4j, Sparksee and Oracle PGX, and graph processing frameworks include the GraphFrames package for Apache Spark, GraphLab and the Gelly package for Apache Flink. In some other cases, such as the Gremlin language [2] and Apache Giraph [4], although they do not provide built-in functions to compute shortest paths, there still exist common and succinct recipes typically described in the system documentation.

In the context of relational DBMSes, some solutions offer to the user, aside SQL, a different model or language tailored to graph workloads. In [14], the authors explored an additional custom syntactic layer for querying graph in a vertex-centric way. Commercially, there is an ongoing progress to support Cypher [7], a graph query language, on top of SAP Hana [9] and PostgreSQL [1].

A vendor extension to SQL with constructs to express both reachability and shortest paths is already supported by Virtuoso Universal Server [8]. In Virtuoso, a `SELECT` clause followed by the `t_transitive` modifier enables a special syntax to compute the transitive closure between two columns. An additional option is available to stop the search to the minimum number of hops, whereas weighted shortest paths are not supported. The construct always yield all valid paths. Depending on the projected attributes, the result set may consist of one row per path or one row per each hop in a path. A custom function, `t_step`, can be used to discriminate the paths from each other and determine the position of a row in its associated path.

The idea of nested tables and their expansion into table expressions was also influenced by the Snowflake's `FLATTEN` construct [10] and Hive's *lateral views* [3]. Snowflake's `FLATTEN` is a table function that allows to expand an object or an array into a table expression. Similarly, Hive's *lateral views* allow to expand an attribute of a table expression by a user defined table function (UDTF), using a syntax somewhat resembling a lateral join: `table_exp LATERAL`

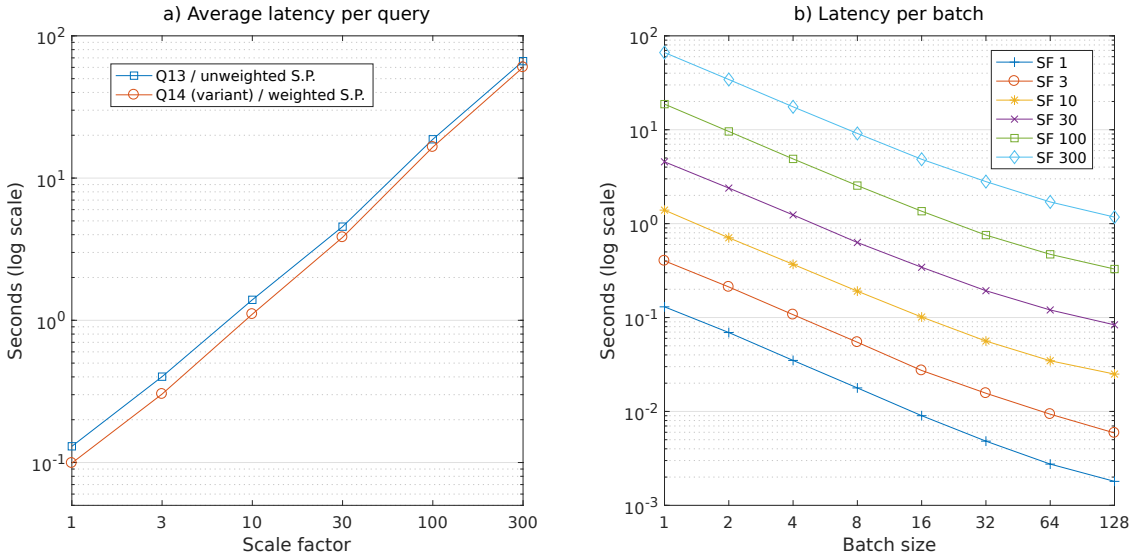

Figure 1: a) Average execution time for the queries Q13 and Q14 (variant). b) Execution of Q13 at varying batch sizes, the reported time is the average time per pair ⟨source, destination⟩: execution time / batch size.

`VIEW udtf(attribute) AS (ident_list)`. The text [12] further investigates the concept of nested tables, named *Relation Valued Attributes*, and their implications to the relational algebra.

6 CONCLUSIONS

In this paper we presented a manner to assess the reachability predicate and calculate shortest paths among user defined graphs. We first proposed an extension to the SQL language, exploiting the *where* clause to filter out tuples based on the reachability predicate and the *select* clause to compute shortest paths according to weights defined by the user. Then, we described our prototype implementation on top of MonetDB and the changes necessary to enable our extension. Finally we showed an early evaluation of our implementation based on two queries from the LDBC SNB benchmark.

Our model enables to specify arbitrary graphs starting out of (virtual) table expressions. This conceals a pitfall: the involved graphs need to be entirely materialized at run-time, likely dominating the execution time of the whole query. While this situation can be improved by optimizing the implementation and rendering it parallel, it might still remain the major bottleneck, especially when only a single shortest path needs to be computed at the end. To mitigate this scenario, we are investigating how to expand our system with the option of creating special 'graph' indices. These indices will store the full graph, ready to be used when a query matches the edge table that generated the graph. Nevertheless, they also need to be amenable to the updates on the underlying tables, challenging the currently adopted runtime CSR representation.

ACKNOWLEDGMENTS

This publication was supported by the Dutch national program COMMIT/ under the *SWALLOW* project *Graphalyzing4Security*.

REFERENCES

[1] 2017. AgensGraph. (2017). http://bitnine.net/agensgraph/
[2] 2017. The Gremlin Graph Traversal Machine and Language. (2017). http://tinkerpop.apache.org/gremlin.html
[3] 2017. Hive documentation: Lateral views. (2017). Retrieved March 2017 from https://cwiki.apache.org/confluence/display/Hive/LanguageManual+LateralView
[4] 2017. Introduction to Apache Giraph. (2017). http://giraph.apache.org/intro.html
[5] 2017. LDBC-SNB Data Generator (DATAGEN. (2017). Retrieved March 2017 from https://github.com/ldbc/ldbc_snb_datagen
[6] 2017. LDBC Social Network Benchmark (SNB) specification. (2017). Retrieved March 2017 from https://github.com/ldbc/ldbc_snb_docs
[7] 2017. The openCypher project. (2017). http://www.opencypher.org/
[8] 2017. OpenLink Virtuoso documentation. Transitivity in SQL. (2017). http://docs.openlinksw.com/virtuoso/transitivityinsql/
[9] 2017. SAP Hana Graph documentation. (2017). help.sap.com/hana/SAP_HANA_Graph_Reference_en.pdf
[10] 2017. Snowflake documentation: FLATTEN table function. (2017). Retrieved March 2017 from https://docs.snowflake.net/manuals/sql-reference/functions/flatten.html
[11] Ravindra K. Ahuja, Kurt Mehlhorn, James Orlin, and Robert E. Tarjan. 1993. Faster algorithms for the shortest path problem. *Journal of the ACM (JACM)* 37, 2 (April 1993), 212–223. DOI: https://doi.org/10.1145/77600.77615
[12] C.J. Date. 2011. *SQL and Relational Theory: How to Write Accurate SQL Code* (2nd ed.). O'Really.
[13] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 619–630. DOI: https://doi.org/10.1145/2723372.2742786
[14] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. 2015. The Case Against Specialized Graph Analytics Engines. In *CIDR*.
[15] ISO. 2011. *ISO/IEC 9075-2:2011 Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation)*. International Organization for Standardization. https://www.iso.org/standard/53682.html
[16] MonetDB team. 2017. MonetDB. (2017). Retrieved March 2017 from https://www.monetdb.org/
[17] Adam Welc, Raghavan Raman, Zhe Wu, Sungpack Hong, Hassan Chafi, and Jay Banerjee. 2013. Graph Analysis: Do We Have to Reinvent the Wheel?. In *First International Workshop on Graph Data Management Experiences and Systems (GRADES '13)*. ACM, New York, NY, USA, Article 7, 6 pages. DOI: https://doi.org/10.1145/2484425.2484432

A EXAMPLES

This appendix features some examples showing the usage of our proposal. In the rest of the section we assume a schema of two tables, *Persons(id, firstName, lastName)* and *Friends(person1, person2, creationDate, weight)*, somewhat based on the LDBC SNB specification. *Friends* models a many-to-many relationships between *Persons*, together with the extra attributes *creationDate* and *weight*. From a graph perspective, *Friends* is the *edge table*, the set union *Friends.person1* $\bigcup$ *Friends.person2* determines the vertices of the graph, while the table *Persons* carries the vertex properties. Figure 2 reports the sample data considered in the following examples.

A.1 Cost of a shortest path

Query 13 of the LDBC SNB IW benchmark requires to compute the distance of the unweighted shortest paths between two persons. The input parameters are the IDs of two persons. This query can be expressed with the syntax below:

```
SELECT CHEAPEST SUM(1)
WHERE ? REACHES ? OVER friends EDGE (src, dst);
```

Where the markers ? are the host parameters to bind.

A.2 Vertex properties

Similar to the previous example, given two person IDs, the following snippet depicts how to compute the shortest distance between them and report it together with their names:

```
SELECT p1.firstName || ' ' || p1.lastName AS person1,
       p2.firstName || ' ' || p2.lastName AS person2,
       CHEAPEST SUM(1) AS distance
FROM   persons p1, persons p2
WHERE  p1.id = ?
  AND  p2.id = ?
  AND  p1.id REACHES p2.id OVER friends EDGE (src, dst);
```

For instance, using the data of figure 2, and binding the parameters to the values 933 and 8333, the result set is:

person1	person2	distance
Mahinda Perera	Chen Wang	2

A.3 Reachability

The following snippet queries all the persons reachable from id = 933 (Mahinda Perera) in the subgraph of friends with creationDate < 01/01/2011.

```
WITH friends1 AS (
    SELECT *
    FROM   friends
    WHERE  creationDate < '2011-01-01'
)
SELECT firstName || ' ' || lastName AS person
FROM   persons
WHERE  ? REACHES id OVER friends1 EDGE (person1, person2);
```

Using the sample data of figure 2, the result set with the query parameter bound to 933 is:

person
Mahinda Perera
Carmen Lepland
Chen Wang

A.4 Multiple weighted shortest paths

The following snippet augments the previous example by computing a shortest path among the reachable persons. The purpose of the query is to find a shortest path for all persons in the subgraph of friends with creationDate < 01/01/2011, starting from a given person ID. Furthermore, the shortest path is weighted, evaluated on the expression `Friends.weight * 2`.

```
WITH friends1 AS (
    SELECT *
    FROM   friends
    WHERE  creationDate < '2011-01-01'
)
SELECT
  firstName || ' ' || lastName AS person,
  CHEPEAST SUM(f: CAST(weight *2 AS int)) AS (cost, path)
FROM  persons
WHERE ? REACHES id OVER friends1 f EDGE (person1, person2);
```

Binding the query parameter to 933, it yields the derived table:

person	cost	path
Mahinda Perera	0	
Carmen Lepland	1	person1 | person2 | creationDate | weight 933 | 1129 | 2010-03-24 … | 0.5
Chen Wang	5	person1 | person2 | creationDate | weight 933 | 1129 | 2010-03-24 … | 0.5 1129 | 8333 | 2010-12-02 … | 2.0

Finally, *unnesting* the path produces the result set:

person	cost	person1	person2	creationDate	weight
Carmen Lepland	1	933	1129	2010-03-24 …	0.5
Chen Wang	5	933	1129	2010-03-24 …	0.5
Chen Wang	5	1129	8333	2010-12-02 …	2.0

Note that the first row (Mahinda Perera) is discarded as its path is empty. Nevertheless, as described in section 2, it can alternatively be retained by using a left outer lateral join.

A) Sample data for the table Persons

id	firstName	lastName
933	Mahinda	Perera
1129	Carmen	Lepland
8333	Chen	Wang
6597069771578	Peter	Taylor

B) Sample data for the table Friends

person1	person2	creationDate	weight
933	1129	2010-03-24T00:54:31	0.5
1129	933	2010-03-24T00:54:31	0.5
1129	8333	2010-12-02T12:23:33	2.0
8333	1129	2010-12-02T12:23:33	2.0
1129	6597069771578	2012-07-30T00:49:50	1.5
6597069771578	1129	2012-07-30T00:49:50	1.5

Figure 2: Sample data for the examples used in this appendix. A) depicts the data for table Persons(id, firstName, lastName, gender) B) reports the records in the table Friends(person1, person2, creationDate) where ⟨person1, person2⟩ is the primary key and both person1 and person2 are foreign keys to the table Persons.