Pattern Analysis and Applications (2020) 23:1493–1504
https://doi.org/10.1007/s10044-019-00841-z

INDUSTRIAL AND COMMERCIAL APPLICATION

PathQuery Pregel: high-performance graph query with bulk synchronous processing

Bogdan Arsintescu[1] · **Shardul Deo**[1] · **Warren Harris**[1]

Received: 24 June 2019 / Accepted: 22 July 2019 / Published online: 1 August 2019

Abstract
High-performance graph query systems are a scalable way to mine information in Knowledge Graphs, especially when the queries benefit from a high-level expressive query language. This paper presents techniques to algorithmically compile queries expressed in a high-level language (e.g., Datalog) into a directed acyclic graph query plan and details how these queries can be run on a Pregel graph vertex-centric compute system. Our solution, called PathQuery Pregel, creates plans for any conjunctive or disjunctive queries with aggregation and negation; we describe how the query execution extracts graph results optimally while avoiding many join operations where parallel map execution is permitted. We provide details of how we scaled this system out to execute large set of queries in parallel over the Google Knowledge Graph, a graph of 70B edges, or facts; we describe our production experience with PathQuery Pregel.

Keywords Distributed graph compute · Pregel · Graph query · Bulk synchronous parallel computing · Graph database

1 Introduction

In recent years, large knowledge bases (a.k.a. Knowledge Graphs) have been built to store factual structured data, for example the Freebase Knowledge Graph [5] which seeded the Google Knowledge Graph [35] and others, social networks and economic graphs [26]. Such data sets are closer in structure to relational databases because although they generally have an ontological ordering of the data, a.k.a. schema, that describes the relationship between objects, they are, indeed, graphs. The most generic description of these graphs is as 'triples': an adjacency list of edges, each edge a (subject, predicate, object) triple [2].

This paper describes PathQuery Pregel, a semantic search system for a graph knowledge base which can execute concurrently a large number (thousands) of analytic queries (each with multi-million results); the queries are expressed using a high-level graph query language. The main contributions of this work are (1) the automatic construction of imperative query plans which use a graph-native execution algebra that avoid huge unnecessary joins, (2) the ability to scale the system to a large number of analytic queries in parallel. The main benefit of executing these graph query plans in a graph-compute system is its inherent scalability in all dimensions: graph size, number of queries and query results size.

A recent survey [4] provides a classification of search systems on the orthogonal dimensions of data type (knowledge/text) and query structure (structured/text); based on this, our application is a structured search over a knowledge base, a.k.a semantic search, a graph traversal constrained on the subject, predicate and object identities in the graph [35]. Another survey of big graph data management systems [24] identifies graph systems as graph databases and graph-compute systems, on the same categorization system that defines OLTP (online transaction processing) versus OLAP (online application processing), respectively: the former are characterized by a large number of short online transactions (INSERT, UPDATE, DELETE) and the latter are

Warren Harris: Deceased.

Bogdan Arsintescu: Currently employed by LinkedIn; work performed 100% while at Google.

✉ Bogdan Arsintescu

Shardul Deo
sdeo@google.com

Warren Harris
warrenharris@google.com

[1] Google Inc, 1600 Amphiteatre Pkwy, Mountain View, CA 94043, USA

low volume of transactions but with complex queries and algorithms that require aggregations.

Our application requires exhaustive responses to a generic set of high-level structured user queries [30]; it must produce large categorical results like OLAP systems and allow the flexibility of a graph database's query language and the algorithmic flexibility of graph systems: we must answer thousands of queries like "`for all actors, top N most recent movies`" and "`for all musicians, all albums and most popular songs`" concurrently over the entire Google Knowledge Graph (KG hereafter), a data set of 70B edges in 2016 [16, 30, 38].

We propose a method to create query plans for any graph-shaped query (a directed acyclic graph of edges, not just a tree-shaped query); these plans support both aggregation and negation and scale to large number of analytic queries (1000+) in parallel. In our solution, queries can also be expressed using a high-level declarative query language; we exemplify with Datalog, but many query languages can be compiled to these plans. Our execution engine is Pregel, and we are using XLisp [36] parallel execution algebra to describe how we avoid the memory and fan-in skew relational engines create; we provide an example of the benefits of this algebra at the end of Sect. 3.2 using the triangle query as an example. For clarity and scope reasons, we limit this paper to describe how the compiler creates an imperative query plan; we defer declarative query planning, dynamic query planning, recursive query execution and indexing optimization techniques for later publication.

The remainder of the paper is organized as follows: we will briefly discuss prior work in the next section. In Sect. 3 we discuss the fundamental principles used to map an imperative graph query onto a Pregel graph algorithm and we walk through solutions for traversal, aggregation and negation. In Sect. 3.3 we discuss the scalability and performance aspects of our Pregel implementation. Section 4 describes our results in deploying and running this system in practice, and in Sect. 5 we conclude with a summary and future work.

2 Related work

The vast majority of related work either works from terse graph query languages toward familiar, traditional relational databases query execution engines or, conversely, maps familiar but verbose SQL language toward newer graph or MapReduce systems [18]. Few methods fall, like ours, in the category of graph query engines over graph-compute platforms.

Relational databases have been extensively studied in the past decades, and many still argue vigorously against graph databases or specialized graph engines [20]; such systems do not naturally scale for web-sized graph and, like SQL systems in general, require specialized tuning for the query profiles (index creation, etc). Specialized indices for RDF graphs [25] help reduce the size of the intermediate results that queries with a large number of joins produce. All these systems are extremely costly to scale out both in terms of hardware and development cost: web-size graphs need be partitioned over many computers to accommodate the storage needed, many CPUs are needed to accommodate the compute load, and custom concurrency models are needed to solve this.

Indices and inverted indices of graph adjacency lists are another solution, like those used in traditional IR search systems [3, 12]. These systems are easy to scale out, and traditional relational database query evaluation algebra can be used to fetch the data from the database shards, then execute a query plan and provide an answer. Recent implementation is described in [1] and [34]. Such systems are more suitable to OLTP workloads rather than OLAP because the large communication pattern of the latter would stress the network and memory layer of the system with the amount of duplicated data or the join sizes. The pre-existing graph-compute platform (Pregel) and its scalability and ease of use for graph applications biased our efforts toward a Pregel application.

Other authors have proposed graph algorithm mappings onto MapReduce systems [17, 33]. MapReduce systems have various shortcomings for graph processing since they fundamentally implement a divide and conquer algorithm while graph queries specify traversals. Graph traversals requires iterative loops over *map* steps which produce large joins (adjacency lists) fan-out followed by recombination and a join computation in the reduce step, then iterate. These systems not scale in memory because of enormous skew created by large joins in the reduce phase. These data modeling problems can be resolved by de-normalizing the graph and converting it into a KV store, where the key is the identity of a node and the value is the N hop reachability set, which is the maximum depth of the query that can be answered.

Pregel [29] and other similar bulk synchronous parallel [37] systems [19, 28] provide a vertex-centric computation model that requires each node to compute its future state in an isolated manner and allows graph algorithms to be implemented on top of this model [23, 40, 41].

Others [31, 39] use Datalog as the high-level language and map traditional relational query evaluation engines over bulk synchronous parallel execution models, where the evaluation vertices use the graph as an index and "pull" adjacency lists into the evaluation; these solutions generate vast amounts of messages and fan-in skew when they pull large sets; solutions to mitigate this such as supervertices (subgraph as a vertex) and eager aggregation are proposed in [31]. While the query evaluation supports recursion, the use

of relational algebra adds skew and memory requirements which hinders scalability.

Recently, authors have attempted mapping graph traversal pattern matching into native graph processing like Pregel. Quegel [43] describes how a tree query is manually mapped into a nested *for* loop of evaluation and it details how manual mapping is required; there is no automation in this solution, and no solution for aggregation and negation.

The solution we propose fits into this last category: native graph query plans on a graph processing system; our method generates query plans for any query with conjunction or disjunction that handles aggregation and negation. Moreover, our query plans are pure functional and independent; hence, any number of queries can run in parallel. We use Datalog as an example query language, but any declarative query language (e.g., [8]) can be compiled into a DAG query plan we propose.

3 Graph query processing & the Pregel Connection Machine

This section describes our techniques to algorithmically map a graph query plan onto a bulk synchronous parallel compute system such as Pregel. For clarity, we limit the scope to discuss only the imperative query execution engine; hence, the order in which the query is executed is predetermined and the execution plan is static. We start by describing a simple graph path query plans, then how a tree path (a conjunction) would work; Sect. 3.2 presents the aggregation and negation solution, which make our query plans directed acyclic graphs (DAGs); In Sect. 3.2.1 we discuss the graph-shaped queries plans and their concurrency, aggregating output and disjunctions, to complete the mapping of the high-level query plans onto topologically sorted plans for synchronous execution on Pregel. We defer the discussion on query planning, dynamic execution as well as recursive queries for a future publication.

The imperative query execution engine produces query plans that (1) can execute any graph-shaped query, (2) scale to provide complete and exhaustive answers for very large graph data sets and (3) scale to execute a large set of queries in parallel on a Pregel compute system; this implements a high-performance graph query system for today's Knowledge graphs and social graphs.

A query language that describes such queries declaratively is Datalog: Throughout this paper we use Datalog [6] as the query language which itself is a restricted form of the logic programming language, Prolog [9]. Any high-level language would work as well and the query evaluator presented here is independent of the language used. Datalog is merely useful to convey the algebra used in the evaluation and well known in the community.

In Datalog, statements which end with a period are assertions (writes); statements which end with a question mark are queries (reads). Our graph data is a collection of (subject, predicate, object) triples [2], so graph edge triples are tersely declared as

```
Edge(subject, predicate, object).
```

Here is an example set of edge statements that describe a subgraph of triples for a notable person:

```
Edge("id_1", "title", "king"), Edge("id_1", "place_of_work", "id_2"),
Edge("id_2", "name", "England"), Edge("id_2", "part_of", "id_3"),
Edge("id_3", "name", "Great Britain").
```

And these edges describe, obviously, that Henry V is the King of England and in our imperfect historical account that is part of Britain; the identity of the vertex is }}*id*_1″. This data is queried as follows:

```
Edge(x, "title", "king"), Edge(x, "place_of_work", y), Edge(y, "name", "England")?
```

query for all the royals *x* with a place of work *y* named }}*England*″: variables allow us to describe trees and graphs of edges to be created or queried. Figure 1 graphically models the query using the Edge declarations as labeled circles, and the variables are the arrows connecting these.

The set of predicates defines the schema and, for a graph database, this is the set of labels on the edges. However, the choice of query language (Datalog, GRAQL, SPARQL) or schema (OWL, or Freebase) has no impact on our query planning algorithms or execution and are here merely to support our examples.

3.1 Query processing engine: graph traversals

The main difference between queries in a graph database and in a relational one is the fact that a graph traversal requires all relationships to be primary (fast) while a relational one considers relationships in a table as primary (fast lookup) and relationships between tables as secondary (slower joins). For example, `"British royal family"` will be a table that one would expect indexed in a relational database but `"blue-eyes British royal family"` is an improbable index: let's assume there was a table of

blue-eyed people and that has to be joined with the table of British royals; we would have to create indices of British (and Dutch and Belgian and other monarchies) blue-eyed to optimize the join size. In relational algebra, two sets of rows are collected from two tables (naively, "king" rows from the "occupation" table and "England" rows from the "place of birth" table which are joined.

messages, no continuation is needed and no more messages are issued, and (3) the system serializes all results and halts: we have found all results.

A compute system that describes a similar algebra is the Connection Machine [22] with its XLisp dialect [36] suitable for parallel computations. The Connection Machine is a massively parallel computer architecture which consists

Edge(x, "title", "king"), Edge(x, "eye_color", "blue"), Edge(x, "place_of_work", y), Edge(y, "name", "England"), Edge(x, "spouses", z)?

In a graph, the relationships are expected to be found in constant time for any given node: each vertex can only access its adjacency list. A query is a traversal from one vertex identity to another with the expectation that all adjacencies are "fast" or constant time. The best model for this data set is a graph, where the nodes are identities and the edges are relationships, aka a RDF subject–predicate–object triples [21]. As noted in [43], graph traversal is achieved by depth- or breadth-first "hopping" the graph from a set of nodes to another. For example, "`spouses of blue-eyes english royal family`" are, starting from each node in R above, the nodes at the object end of the edges labeled "`spouses`". Graph traversals do not require joins: we don't need all spouses of *all* kings, we need all spouses of *each* king.

Using a functional language such as LISP [7], this is modeled as a *mapcar* operation over the set x: the subsets of y are independent for each value of x and we are not required to join them together: map the subjects using the predicate names into the list of objects; those objects are in turn the subjects of the next wave of edges in the traversal, and we map the subjects through a predicate edge label, etc. According to Valiant's bulk synchronous model [37], these 'map' operations are easy to parallelize. This observation is a fundamental building stone for parallelism, scalability and vertex-centric execution on Pregel.

Pregel is a vertex-centric computation system aimed at parallel graph computing, where computation is planned in synchronous supersteps: in each superstep, each vertex executes a used-defined compute based on incoming messages and the vertex value and adjacency and sends messages to other vertices. These messages are grouped by vertex and delivered in the next superstep, iteratively; the computation halts when there are no new messages issued. Pregel provides no global state and each vertex only knows the messages it receives and its current state. For our example, (1) all "`royal`", "`blue eyed`" and "`British`" vertices send messages to their "spouse_id" vertices with their identity and a query continuation; (2) when receiving those

of an array of general purpose processors that perform the same operation on multiple data points simultaneously; each processor executes a pure functional λ given a set of input arguments and sends the result of that computation to a set of processors in the network. The non-Von Neumann computer architecture of the Connection Machine and XLisp defines a parallel graph computation algebra:

- each processor is a vertex and its edge adjacency list in our graph;
- the queries are the pure functional λ operations that select how the incoming messages and/or the vertex adjacency are processed;
- the adjacency list helps route the results and the program to the next nodes. For our example, in the 1st step, each compute node decides whether it is labeled "royal", "blue eyed" and "english" and has "spouse" connections. Those nodes that do use the triple ([royal id], place-of_work, [place_id]) send a remainder of the query (i.e., "match predicate 'name' to object literal 'Britain"') to each [place_id] with the accumulated data and the remainder of that query in that path; this design pattern is very similar to the Scheme call-with-current-continuation Scheme pattern [11] and Prolog Backtracking [14];
- at the end of the computation, each place_id knows its royal_id and the query and we can store the data in a distributed mode. More details about creating a set of results follow later in this section.

The Connection Machine XLisp[36] algebra helps describe how Pregel operates: in each superstep each vertex executes a pure functional operation λ over the set of vertices. XLisp defines the parallel map data type call *Xap* (a parallel map), which is the adjacency list of each vertex in a Pregel graph. For example, the small graph from the previous subsection can be represented as a *Xap* of node adjacency list V:

$$V = Xap\{''id_1'' \rightarrow \{name, title, place\_of\_work, spouses, \dots\}, \\ ''id_2'' \rightarrow \{name\}\}$$

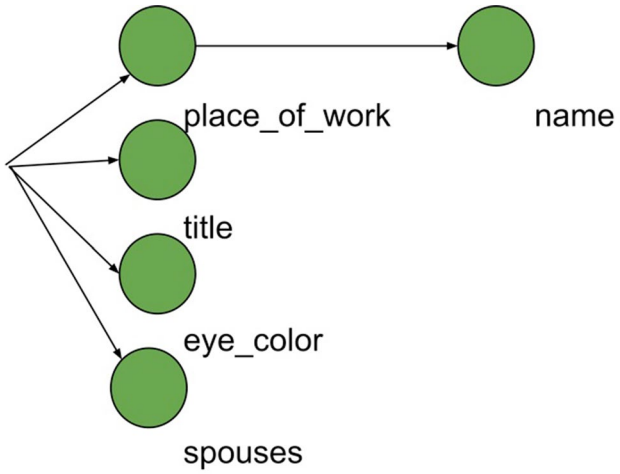

Fig. 1 Query plan diagram

Each Pregel compute superstep is, using XLisp algebra, an "apply-to-all" $\alpha F()$ which applies, like *mapcar* in Lisp, to each element separately with its own incoming messages and produces a *Xapping* of messages for the next step; just like a *Xap*, all vertices are independent of all other vertices.

$$M = (Xap\ (\alpha\ F(V\ M_i)))$$

The Pregel system groups these messages by destination and uses them in the next superstep to apply an $\alpha F()$ to each vertex and continues while there are messages to deliver. The Pregel system as a whole is described by the following XLisp while loop:

$$(while\ M$$
$$(Xap\ M_i) = (group - by\ i\ M)$$
$$M = (Xap\ (\alpha\ F(M_i))))$$

For our query plans, each $\alpha F()$ operation is an *Edge*() select operation, i.e., one hop in the graph. Similarly, disjunction is a forest of $\alpha F()$ paths. This graph-compute system can so far answer any conjunction or disjunction (set of paths in the graph) but obviously no aggregation: each path computation is independent of all other and vertices know nothing about other vertices; if one would like to know the top royals by the number of marriages, we need aggregation; we describe our aggregation solution in Sect. 3.2.

The principles we keep throughout the implementation of this system are:

1. This is a pure functional query system without side effects: we do not rely on a global state and the graph is stateless; the vertices and the edges do not have state that spans a Pregel superstep: each λ sees each vertex in the same state. This requires us to send all the information through messages (current results and remainder of the query); also, it permits us to implement graph-shaped queries (later in this paper) and recursive and declarative query evaluation (in a subsequent publication). Without a stateless adjacency graph, any query reaching the same node sees a mutated adjacency list. By contrast, the graph processing language solution proposed in [23] allows for a global state.
2. All the λ functions we apply to the vertices are pure functional as well and do not create global state used in the algorithm: Pregel implements aggregators and allows for compute global state; we do not use those in the algorithm to allow for easy recovery and to keep our graph system stateless. A global state may prevent running multiple queries in parallel.

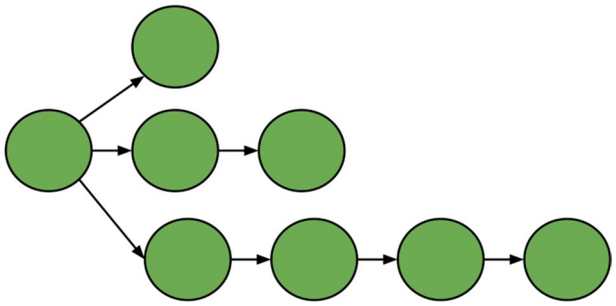

Fig. 2 Conjunctive query tree

The diagram in Fig. 2 shows a conjunctive query that follows 3 paths starting from a shared subpath; the query requires 5 superstep (the maximum depth of the tree) to complete. In tree queries each result path is independent of all other paths, a set of leaf nodes end up receiving all results for the top path, another set receives the center, and another receives the bottom path results. All paths originating from one vertex are independent results to the query; we describe how they are all aggregated in Sect. 3.2.

3.2 Aggregation

The discerning historian is obviously not interested in all British royal marriages, but in the most prolific ones: how do we modify our system to get straight to Henry V? The database answer is aggregation: collect all the answers in one place, sort and select given an intrinsic criteria: "`top 10 blue eyed British royals by marriage count`". Note, we can select the ones with more than "3" marriages independent with a $\alpha F()$ operation (that is, "3" is an independent, extrinsic parameter) while "`top 10`" is an intrinsic parameter that requires the exhaustive list.

In a Pregel graph all messages must be delivered to the same vertex, but such vertex does not exist in the naive graph. Our solution is to create a new vertex identity that is

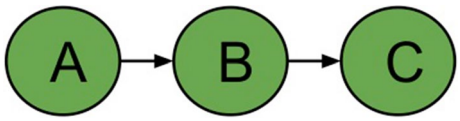

Fig. 3 Triangle query

unique and transient: this vertex receives all the messages to be aggregated, executes the aggregation compute operation and sends messages to continue the computation; the vertex is then deleted. We call this an "aggregation vertex"; its identity must be unique to ensure that we only collect the set of messages necessary and sufficient for that aggregation: for our example, we can have one aggregation node for *each* royal to select the right subset; we do not want all the spouses for *all* the royals and then deal with the sorting job. This technique allows us to implement a rich—and extensible—library of aggregation functions (e.g., min, max, count, set, filters, top, sort...) as needed by our queries.

Our reason to make the identity of the aggregation node transient is to preserve our principle that the graph should be stateless between Pregel supersteps: these compute vertices exist merely for the purpose of computing the result of the λ and have no edges or side effects. The Pregel system affords this technique: messages can be sent to any vertex identity; if the identity does not exist (that is, an "orphan message" is observed), the superstep barrier creates the missing vertex identity at the barrier and delivers the messages after the barrier. We delete all the aggregation vertices after each operation; hence, the graph is stateless after each superstep.

The Pregel system streams the messages; hence, messages have no compound memory effect on the system. Large fan-in is not a memory concern but only influences the duration of the compute because the enumeration complexity is linear with the message set size n.

A standard benchmark is the triangle query

```
Edge(x, "A", y), Edge(y, "B", z), Edge(z, "C", x)?
```

and its DAG is shown in the diagram in Fig. 3 as a series of 3 supersteps. With our algebra, the triangle query execution yields a Xap of results of cardinality T after three $\alpha\ F()$ operations:

$$M = (Xap\ (\alpha\ C(\alpha B(\alpha A()))))$$

Notably, it is not a triangle and there is no join with our algebra. In the last superstep, a test of equality is performed in *each* individual path in constant size and time and the result is part of the solution iff the root and the destination nodes are identical. Note that the messages propagate only when the mapping is successful: the number of messages testing label C is equal to the size of A followed by B rather than $A \cdot B$; in a vast majority of cases the graph schema allows us to rewrite the query such that we traverse the graph in the direction of low-cardinality predicates, hence the upper bound of this traversal is $max(A, B)$ when one of the predicates has cardinality equal to one.

Relational algebra requires joins and produces a single result with T rows:

$$M = (\beta \bigcap (\alpha\ C(),\ \beta \bigcap (\alpha A(),\ \alpha \overline{B}())))$$

where $\overline{B}$ is the inverse edge of B. XLisp algebra describes aggregation using $\beta F()$, an operation applied to all elements in a set. With the relational algebra, the join operations create potentially large fan-in at the 'join' operations. With the graph algebra the *Xap* fan-in is constant; the *Xap* data structure afforded by Pregel allows us to avoid traditional joins and produce solutions independently (that is, in a }{*foreach*″ parallel approach) instead of set of solutions (that is, a }{*forall*″ approach).

3.2.1 Concurrency of aggregation

When all paths leading to an aggregation vertex are of equal length (same number of edge traversals), then all messages arrive at the aggregation node concurrently in the same superstep; otherwise, they arrive at different supersteps and must be held indefinitely, because the system is stateless, and we do not know any global query state. A few solutions are possible, among them: (1) introduce some global, static state to lookup the wait time or (2) encode the wait time in the continuation the messages contain or synchronize the computation. The former option introduces global state and makes some of these vertices survive multiple supersteps.

We prefer the second option for 3 reasons: (a) is more native to XLisp algebra and Pregel and to the principles of a synchronous system, (b) is pure functional and does not violate our principles and (c) signal concurrency is well studied in integrated circuit design algorithms [10].

Signal concurrency in VLSI [10] ensures that logical processing units receive all stimuli within a synchronized time window and produce a response which is buffered and synchronized and then distributed at the next synchronous barrier to the inputs of the following stage; this architecture permitted the incredible scale of semiconductor growth in the last 50 years predicted by Moore's law [32]. Also, generating low level circuits from high-level description languages is a well studied problem [27] with proven solutions to signal concurrency.

Our query compilation produces synchronous query execution plans that ensure concurrency of the aggregation nodes. Shorter concurrent paths in the graph are buffered to match the length of longest path in the concurrency set, just like VLSI paths are [42]; we insert *wait_supersteps*$(m - n)$

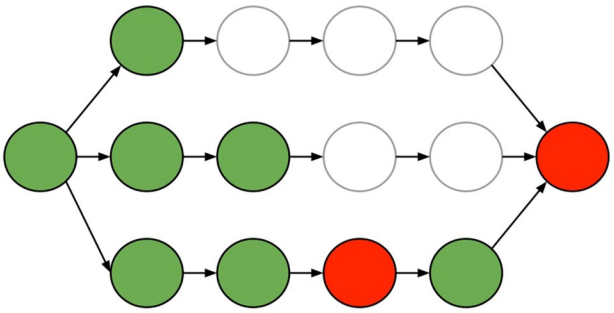

Fig. 4 Concurrency of aggregation

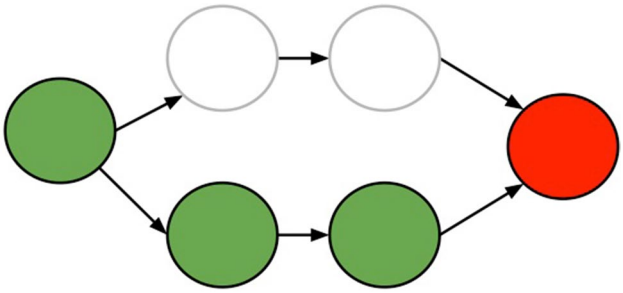

Fig. 5 Concurrency of negation

where m and n are the longest and shorter path length, respectively.

Figure 4 shows a query tree where the 3 branches to be aggregated at the rightmost red node are of different lengths: the green nodes represent the regular vertex compute aligned vertically by superstep. The aggregation node (in red above) expects all messages concurrently in the same superstep: to achieve concurrency they have to be delayed through the missing (gray boundary) nodes in the diagram above; we implement that as "buffer" operations where messages are bounced to the source vertex for a number of steps before being sent to the aggregation node. For example the messages are bounced thrice for the top path and twice for the center path; all paths have same length, 4 supersteps, after which all paths are aggregated.

Aggregation nodes may join different paths or all messages on one path. The former case is a general set intersect or union problem and allows us to answer graph-shaped queries (e.g., the query "actors with Oscar awards that are also musicians with more than 3 albums") and the latter (the red node in the bottom path) is the simple value aggregation (e.g., the query "musicians with an album count greater than 3"). The green nodes may represent a set of vertices corresponding to α parallel operations while a red one is always a single vertex in the graph corresponding to a β operation.

In the Pregel framework, this can be implemented in a number of ways, we discuss three of them and their relative merits:

1. When a *wait_superstep* continuation is encountered, change the state of the vertex and store the outgoing message for $m - n - 1$ supersteps before sending it, counting down the supersteps. This technique adds memory to the vertices and may bloat the global RAM used, which may lead to operational issues. Also, it violates our pure functional graph architecture in an inconsequential way—the adjacency is the only relevant immutable state.
2. Bounce the message to self for $m - n - 1$ time before sending it to the destination. This is entirely consistent to both Connection Machine and Pregel architecture; it is similar to the first solution in behavior, preserve the lifespan of the aggregation nodes to one superstep and does not add RAM, because messages are streamed in Pregel. In the image above it is equivalent to inserting "dummy" nodes in the graph such that all the paths are of equal length.
3. Change the Pregel SendMessage implementation and add an integer argument for "how many supersteps in the future" a message should be send, that is the default is one:

```
Vertex::Send(Message m, unsigned int future_supersteps = 1);
```

such implementation is system optimal, the messages are stored for $m - n - 1$ supersteps barrier before they are sent to the aggregation vertex.

3.2.2 Negation

We implement negation, as other database systems do, as an extension of aggregation: count the size of a set and expect a null one to negate an expression. The Pregel query mapping of a negation is an aggregation message sent with the query continuation and the computation continues iff that is the sole message received at the aggregation node: that is, we expect a null set of messages to arrive on the bottom path. Because we rely on the aggregate vertex to send continuations, we need to send a sentinel message to execute the "aggregation"; otherwise, the aggregation vertex is never created and the computation can't be executed; the intent is "if you are the lone message arriving at the aggregation vertex, compute this continuation". In the diagram in Fig. 5, the bottom, green path is the path to be negated, the top path is the one taken by the sentinel message and the red aggregation node is the sentinel that monitors that no messages are incoming through the negated path: that is, for the query "a then not(b.c)" the answer is positive iff the aggregation node receives only the sentinel message sent after A.

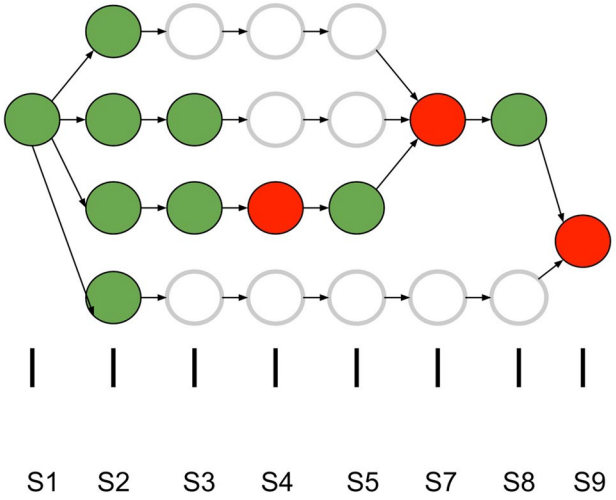

Fig. 6 Synchronous query plan

3.2.3 Query planning output

As described so far, queries are DAGs in the Pregel implementation and each message carries the results on the path to the current solution and the remaining path to be executed. After the path is exhausted, all results for each path lie at the leaf vertices, which is not ideal from the users point of view: they expect the whole tree of results associated with each root node to be aggregated together. For example, for each blue-eyed British royal, provide a set of spouses.

Output is identical to aggregation: our query planner adds an output aggregation node as a sink for each query, and all results are aggregated and converted into a tree at that sink vertex. These sink vertices are similar to any other aggregation nodes and execute a "merge" operation from all the received paths given the query logic and preserve all the grouping of the paths as specified in the query tree.

The Pregel system serializes the results each vertex chooses to store in the last superstep: the user overrides the *Vertex* : : *Write* procedure as follows: query output aggregation above allows us to (1) create one aggregation node per "root" vertex and query, (2) transform it into the desired output format to accommodate a declarative query planner or a graph transformation extension to Prolog and (3) group the results as the users of the system expect in an key-value store keyed primarily with the vertex id of the source and secondarily with the query—or any other key we choose.

Our query planner compiles each query into a DAG that has exactly one source and one sink, all paths between two nodes are equal length, and each node is a Pregel vertex compute event, either at regular vertex or an aggregation (including negation/output) vertex. A diagram of such a query is shown in Fig. 6: the green nodes are computation at vertices, the red nodes are computation at aggregation or output nodes, and the transparent ones are synchronization nodes inserted for concurrency. Our imperative execution engine traverses a Datalog statement left to right and top to bottom and creates a DAG of computation, then inserts padding to make all paths equal. In Fig. 6 the superstep sequence is labeled S1 ...S9 for the query example.

A DAG is a necessary condition for the query plans: all paths are explicit and there is no recursion in the execution graph: that is, we know the exact length of each traversal and we can synchronize the paths in the graph; also, we can perform a traversal in topological order [13]. Because we disallow cycles, there is no phylogenetic traversal or transitive closure allowed: we can't find "zoos with bears in California", where we have two recursions: "bears" is a phylogenetic regnum that zoo inhabitants belong to and California is a hierarchy of places contained in each other. We can express an explicit disjunction for all paths between people up to length 6 (or, generally N), but we cannot compute Bacon numbers for all people [15]. The recursive query plan required a different query plan strategy; we defer recursive query planning for a later publication.

Note that any graph-shaped queries can be automatically mapped to a Pregel query plan: any graph-shaped (concurrent paths) "equals" or "match" is a type of aggregation; disjunctive queries are handled as separate queries that are joined by a "union" operator: we convert a forest of queries into a tree and each query plan is a DAG with a single source and sink. Each result of a query is a graph of sets with a single source vertex and a single sink: each $\alpha\lambda()$ operation yields a *Xap* at a set of vertices, each $\beta\lambda()$ aggregation operation is a single vertex, using the *XLisp* algebra [36]. The query plans compiled into one of these DAG graphs; it can be executed by the generic Pregel compute algorithm because each node starts with all paths and send messages to subsequent nodes with (1) the nodes collected along the path and (2) the remaining operations to be executed. When the next destination is a "green" node, the message is sent to a vertex ID (i.e., object) as the vertex adjacency list (i.e., predicates and objects) and the query (i.e., predicate filter) requires. The aggregation node destinations are computed on the fly based on the selective identity of the vertices on the incoming path and the operation to be executed. That guarantees that all messages arrive at the same vertex (e.g., for paths they started at the same root, traversed the same aggregation nodes and require the same operation, they will all be sent to the same aggregation node). After that, the identity of the destination is in the messages themselves. In summary, all paths know their continuation and the query planner ensures the concurrency of each query: all results end up at a unique aggregation node.

One of the great advantages of our approach is that we have made no assumption on the λ functions that executes at each node: we can execute any pure function applied

Fig. 7 Concurrency of aggregation

to a set of incoming messages and a vertex adjacency list that produces a set of messages. That makes our compute implementation extremely simple, the incoming messages tell each vertex what to compute: all messages at a green graph vertex are independent computations and all messages at a red aggregation node are—by design—concurrent and consistent in the operation to be executed. The semantic of the computation is practically scalable to any library of functions that satisfies these constraints.

3.3 Scalability

The key to the scalability of the system is the guarantee that aggregation vertices are uniquely identified by (1) the relevant nodes in the result path and (2) a key unique to the entire query they execute. That is, each of them is *unique* to its DAG query plan and that particular result; hence, the system can independently execute any number of queries in parallel without any contention of the results. Because each query path is guaranteed to collect all nodes at a unique final identity, the Pregel graph query system (1) can execute any number of "computations" (aka queries) in parallel and (2) each query can start at any superstep.

We can execute as many queries in parallel as the memory of the system allows, because the Pregel system streams the messages at each superstep boundary without adding memory or mutating the graph. The Pregel system scales linearly with the number of compute nodes N and each compute node holds $|V|/N$ of the vertices, where $|V|$ is the cardinality of the vertex set and $|V| \gg N$. We allocate transient memory for the aggregation nodes, but that memory is freed at the boundary of the superstep. Because we have a large number of queries that execute in parallel (over 1000 categorical queries of different length in parallel in one instance) each

superstep spawns new aggregation nodes at a rate of $r \cdot |V|$ where $0.25 \leq r \leq 10$. Figure 7 shows how the number of vertices grows in each superstep for 5 different jobs and the memory footprint of the system. The queries are divided randomly between the jobs. One can note that the memory growth is less than linear: while the number of vertices for Job 3 and 4 grows 2000% in some supersteps, the memory footprint grows by 70% and 50%, respectively: these vertices have no data, no adjacency list, they are a transient computation node.

Valiant notes that skew is inevitable in bulk synchronous systems [37]: some compute processes take longer than others in a heterogeneous system because each vertex set computes different queries. However, the Pregel system assigns vertices into shards uniformly at random into a worker shard (a process in the distributed system) and each vertex has its own adjacency and computation load. As a result, the average compound computation skew of various worker shards evens out. In our system, the graph we model is heterogeneous (the Knowledge Graph contain numerous ontology domains), the queries are heterogeneous (we expect queries about marriages, football, music and food, among others) and the query set is large and we add and remove aggregation nodes with random identities based on these queries. With respect to skew, all three factors above combine into a balanced "homogeneous" compute system where the overall computation evens out: the diversity of vertex loads combines into an overall computation with negligible skew.

Most Knowledge Graphs are not densely connected, because the ontology of the knowledge domains is limited and hence the adjacency list of each node is much smaller than the number of vertices in the graph. (We do have optimization solutions for high-degree vertices which are described later in this section.) That is, the size of adjacency is a constant A for the vast majority of nodes in the graph; hence, the number of messages sent by each node or received by each regular or aggregation vertex is $O(A)$. The number of messages sent in each superstep is $O(|V| \cdot A)$ and in a random distribution of results and node identities each worker compute node receives $O(|V| \cdot A/N)$ for its $|V|/N$ vertices of the graph messages per global query.

4 Production experience & results

Our PathQuery Pregel system has been live in production for a few of years and running analytics query load over the Google Knowledge Graph, a data set of 70B edges in 2016 [16, 38]. We have built and scaled-out the system over a period of time, hence learned the optimization needed throughout the ramp-up of the system. We have run a number of separate jobs that produced independent results for different clients. In this section, we provide operational details about the two most significant jobs because their query profiles diversity informed our optimization work and priorities and explain the origin of the solutions described in Sect. 3.3.

The largest production job contained the largest categorical (high volume analytical) query set: over 1000 complex queries running in parallel. This job was running in under one hour with a thrice daily frequency. We have run various query profiles for various experiments with 200–2000 machines. The queries in this set had a depth (number of supersteps) of 2-30, that is, the most complex query was traversing 30 predicates in the graph. The query set was very heterogeneous in this case and that led us to the observations on ontology domain locality, as described in Sect. 3.3: cars and musician queries are largely independent, but when a domain is dominating the graph then that has an overall larger impact than others. For example, if the graph contains N times more data in one domain than another, then we would expect the message load for queries in that domain to be N times higher. Even more, the number of features is proportional to the domain coverage, more queries with higher complexity would exist for a domain with more data: the knowledge graphs are "open world" systems (incomplete data) when they get closer to a "closed world" (complete data) the user experience improves and drives up the demand. In practice, this tends to make the message load superlinear. This informed the strategy of grouping the queries into staggered trains, common query subpath optimization, etc.

All jobs above were run on shared resources in the Google data centers, an environment where other ephemeral jobs may compete for resources: due to the nature of the bulk synchronous query execution in Pregel the jobs tend to run longer and need stability, because a single machine failure requires re-starting the superstep for that machine, but all machines would wait for the superstep to complete. For this reason, long supersteps were most vulnerable; as they lasted longer, they were more susceptible to repeated failures and were taking the longest to rerun. We have developed many instrumentation techniques to be able to quantify and profile the message load in each superstep and used that information to optimize the health of our jobs over time.

Skew (excessive compute at one of the Pregel worker machines) was generally solved with query rewriting; we were able to resolve all large fan-out and fan-in with manual rewrites. The optimization turned out to be schema-driven rather than graph-shape driven, hence the rewrite was generally improving a whole class of queries, and the techniques were easily generalizable over schema domains. We have monitored all jobs over many months (tens or hundreds of consecutive runs) and noticed little variation in the run-time over the median, and the few large variations were generally resource limitations or machine failures. We have also

monitored the message load per worker machine to ensure that message load was largely constant and, given a randomly distributed query load, the overall compute time had negligible skew.

We are confident that this query execution engine scales naturally over any web-sized Knowledge Graph with imperative queries that avoid high-degree vertices at fan-in and fan-out; horizontal scalability and staggering are techniques to increase parallelism and scale throughput for larger query sets. Dedicated resources could be used if needed to ensure stability, but a shared-resource datacenter as the one we used provided excellent stability even at normal job priority (that is, our jobs were generally not preempted when running with average priorities).

5 Conclusions

We have presented novel techniques to map a declarative high-level query language into an imperative execution engine in Pregel; we provided algorithmic solutions to map any queries with aggregation, negation over conjunctive and disjunctive queries into a concurrent compute system called PathQuery Pregel and described how the results are computed. Our solution provides a generic mapping of generic queries to a Pregel compute without manual development and supports a broad type of queries expressed in a of high-level language, such as Datalog; the parallel map algebra used avoids skew and allows multiple analytic queries (1000+) in parallel.

Our results with production data show the scope of the solution and the impact it provided to our operations; we share our experience with Pregel and the innovative techniques we derived from seminal work on the Connection Machine and the VLSI synchronization methodologies which helped us achieve concurrency. We describe how the solution scales to any web-sized Knowledge Graph.

Future work requires assembling all the manual techniques described for optimizing the imperative execution into a query planning algorithm that supports Datalog queries as declarative queries. Also, extensions to our mapping algorithm to allow recursive queries and indexing that goes beyond naive vertex indexing have been deferred for a later publication for clarity.

Acknowledgements None of this would have happened without the passion, knowledge and vision of our late friend and colleague Warren Harris. We regret his passing immensely and we dedicate this summary of our work to his memory. We are proud of what we have achieved together and of what we learned from Warren.

References

1. Aberger CR, Lamb A, Tu S, Nötzli A, Olukotun K, Ré C (2017) Emptyheaded: a relational engine for graph processing. ACM Trans Database Syst 42(4):20:1–20:44. https://doi.org/10.1145/3129246
2. Angles R, Gutiérrez C (2008) Survey of graph database models. ACM Comput Surv 40(1):1:1–1:39. https://doi.org/10.1145/1322432.1322433
3. Baeza-Yates RA, Ribeiro-Neto BA (1999) Modern information retrieval. ACM Press, Addison-Wesley, Boston
4. Bast H, Buchhold B, Haussmann E (2016) Semantic search on text and knowledge bases. Found Trends Inf Retr 10(2–3):119–271. https://doi.org/10.1561/1500000032
5. Bollacker KD, Evans C, Paritosh P, Sturge T, Taylor J (2008) Freebase: a collaboratively created graph database for structuring human knowledge. In: Wang JT (ed) Proceedings of the ACM SIGMOD international conference on management of data, SIGMOD 2008, Vancouver, BC, Canada, June 10–12, 2008. ACM, pp 1247–1250. https://doi.org/10.1145/1376616.1376746
6. Ceri S, Gottlob G, Tanca L (1989) What you always wanted to know about datalog (and never dared to ask). IEEE Trans Knowl Data Eng 1(1):146–166. https://doi.org/10.1109/69.43410
7. Chassel RJ. The mapcar function. https://www.gnu.org/software/emacs/manual/html_node/eintr/mapcar.html
8. Chavarría-Miranda DG, Castellana VG, Morari A, Haglin D, Feo J (2016) Graql: a query language for high-performance attributed graph databases. In: 2016 IEEE international parallel and distributed processing symposium workshops, IPDPS workshops 2016, Chicago, IL, USA, May 23–27, 2016. IEEE Computer Society, pp 1453–1462. https://doi.org/10.1109/IPDPSW.2016.216
9. Colmerauer A, Roussel P (1993) The birth of prolog. In: Lee JAN, Sammet JE (eds) History of programming languages conference (HOPL-II), Preprints, Cambridge, Massachusetts, USA, April 20–23, 1993. ACM, pp 37–52. https://doi.org/10.1145/154766.155362
10. Contributors W (2015) Synchronous circuit—Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Synchronous_circuit&oldid=696626873. Online; Accessed 15 Feb 2018
11. Contributors W (2017) Call-with-current-continuation—Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Call-with-current-continuation&oldid=811008297. Online; Accessed 16 Feb 2018
12. Contributors W (2017) Inverted index—Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Inverted_index
13. Contributors W (2017) Topological sorting—Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Topological_sorting&oldid=805893309. Online; Accessed 15 Feb 2018
14. Contributors W (2018) Backtracking—Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Backtracking&oldid=824587461. Online; Accessed 16 Feb 2018
15. Contributors W (2018) Kevin Bacon—Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Kevin_Bacon&oldid=823782926. Online; Accessed 16 Feb 2018
16. Contributors W (2018) Knowledge Graph—Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Knowledge_Graph&oldid=822449387. Online; Accessed 15 Feb 2018
17. Cuzzocrea A, Cosulschi M, Virgilio RD (2016) An effective and efficient mapreduce algorithm for computing bfs-based traversals of large-scale RDF graphs. Algorithms 9(1):7. https://doi.org/10.3390/a9010007
18. Dean J, Ghemawat S (2004) Mapreduce: simplified data processing on large clusters. In: Brewer EA, Chen P (eds) 6th Symposium on operating system design and implementation (OSDI 2004), San Francisco, California, USA, December 6–8, 2004. USENIX

Association, pp 137–150. http://www.usenix.org/events/osdi04/tech/dean.html
19. Facebook: Apache giraph. http://giraph.apache.org/
20. Fan J, Raj AGS, Patel JM (2015) The case against specialized graph analytics engines. In: CIDR 2015, seventh biennial conference on innovative data systems research, Asilomar, CA, USA, January 4–7, 2015, online proceedings. www.cidrdb.org. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper20.pdf
21. Group RW Rdf. https://www.w3.org/RDF/
22. Hillis WD (1989) The connection machine. MIT Press, Cambridge
23. Hong S, Salihoglu S, Widom J, Olukotun K (2014) Simplifying scalable graph processing with a domain-specific language. In: Kaeli DR, Moseley T (eds) 12th Annual IEEE/ACM international symposium on code generation and optimization, CGO 2014, Orlando, FL, USA, February 15–19, 2014. ACM, p 208. https://doi.org/10.1145/2544137.2544162
24. Junghanns M, Petermann A, Neumann M, Rahm E (2017) Management and analysis of big graph data: current systems and open challenges. In: Zomaya AY, Sakr S (eds) Handbook of Big Data technologies. Springer, New York, pp 457–505. https://doi.org/10.1007/978-3-319-49340-4_14
25. Kim K, Moon B, Kim H (2014) Rg-index: An RDF graph index for efficient SPARQL query processing. Expert Syst Appl 41(10):4596–4607. https://doi.org/10.1016/j.eswa.2014.01.027
26. LinkedIn (2018) Linkedin economic graph. https://economicgraph.linkedin.com/
27. Lipsett R, Schaefer CF, Ussery C (1989) VHDL: hardware description and design. Kluwer, Dordrecht
28. Low Y, Gonzalez J, Kyrola A, Bickson D, Guestrin C, Hellerstein JM (2012) Distributed graphlab: a framework for machine learning in the cloud. PVLDB 5(8):716–727
29. Malewicz G, Austern MH, Bik AJC, Dehnert JC, Horn I, Leiser N, Czajkowski G (2010) Pregel: a system for large-scale graph processing. In: Elmagarmid AK, Agrawal D (eds) Proceedings of the ACM SIGMOD international conference on management of data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6–10, 2010. ACM, pp 135–146. https://doi.org/10.1145/1807167.1807184
30. Marcus G (2012) The web gets smarter. https://www.newyorker.com/culture/culture-desk/the-web-gets-smarter. Online; Accessed 15 Feb 2018
31. Moustafa WE, Papavasileiou V, Yocum K, Deutsch A (2016) Datalography: scaling datalog graph analytics on graph processing systems. In: Joshi J, Karypis G, Liu L, Hu X, Ak R, Xia Y, Xu W, Sato A, Rachuri S, Ungar LH, Yu PS, Govindaraju R, Suzumura T (eds) 2016 IEEE international conference on Big Data, BigData 2016, Washington DC, USA, December 5–8, 2016. IEEE, pp 56–65. https://doi.org/10.1109/BigData.2016.7840589
32. Reference O Moore's law. http://www.oxfordreference.com/view/10.1093/oi/authority.20110803100208256
33. Rohloff K, Schantz RE (2010) High-performance, massively scalable distributed systems using the mapreduce software framework: the SHARD triple-store. In: Tilevich E, Eugster P (eds) SPLASH workshop on programming support innovations for emerging distributed applications (PSI EtA-ΨTheta 2010), October 17, 2010, Reno/Tahoe, Nevada, USA. ACM, p 4. https://doi.org/10.1145/1940747.1940751
34. Seo J, Park J, Shin J, Lam MS (2013) Distributed socialite: a datalog-based language for large-scale graph analysis. PVLDB 6(14):1906–1917
35. Singhal A. Introducing the knowledge graph: things, not strings. https://googleblog.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html
36. Steele LS Jr, Hillis WD (1986) Connection machine LISP: fine-grained parallel symbolic processing. In: LISP and functional programming, pp 279–297
37. Valiant LG (1990) A bridging model for parallel computation. Commun ACM 33(8):103–111. https://doi.org/10.1145/79173.79181
38. Vincent J (2018) Apple boasts about sales; google boasts about how good its AI is . https://www.theverge.com/2016/10/4/13122406/google-phone-event-stats. Online; Accessed 15 Feb 2018
39. Wang J, Balazinska M, Halperin D (2015) Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. PVLDB 8(12):1542–1553
40. Yan D, Cheng J, Lu Y, Ng W (2015) Effective techniques for message reduction and load balancing in distributed graph computation. In: Gangemi A, Leonardi S, Panconesi A (eds) Proceedings of the 24th international conference on World Wide Web, WWW 2015, Florence, Italy, May 18–22, 2015. ACM, pp 1307–1317. https://doi.org/10.1145/2736277.2741096
41. Yan D, Cheng J, Xing K, Lu Y, Ng W, Bu Y (2014) Pregel algorithms for graph connectivity problems with performance guarantees. PVLDB 7(14):1821–1832
42. Yan Z, Liu M (1996) The RTL binding and mapping approach of VHDL high-level synthesis system HLS/BIT. J. Comput Sci Technol 11(6):562–569. https://doi.org/10.1007/BF02951619
43. Zhang Q, Yan D, Cheng J (2016) Quegel: a general-purpose system for querying big graphs. In: Özcan F, Koutrika G, Madden S (eds) Proceedings of the 2016 international conference on management of data, SIGMOD conference 2016, San Francisco, CA, USA, June 26–July 01, 2016. ACM, pp 2189–2192. https://doi.org/10.1145/2882903.2899398