Pre-print of article that appears in Journal of Parallel and Distributed Computing (JPDC), Volume 151, May 2021, Pages 94-111, https://doi.org/10.1016/j.jpdc.2021.02.004

ELSEVIER

Available online at www.sciencedirect.com
ScienceDirect

00 (2020) 1–24

JPDC

Granite: A Distributed Engine for Scalable Path Queries over Temporal Property Graphs

Shriram Ramesh[a,*], Animesh Baranawal[a], Yogesh Simmhan[a]

[a] *Department of Computational and Data Sciences,*
Indian Institute of Science, Bangalore 560012 India

Abstract

Property graphs are a common form of linked data, with path queries used to traverse and explore them for enterprise transactions and mining. *Temporal property graphs* are a recent variant where *time* is a first-class entity to be queried over, and their properties and structure vary over time. These are seen in social, telecom, transit and epidemic networks. However, current graph databases and query engines have limited support for temporal relations among graph entities, no support for time-varying entities and/or do not scale on distributed resources. We address this gap by extending a linear path query model over property graphs to include intuitive *temporal predicates* and *aggregation operators* over temporal graphs. We design a *distributed execution model* for these temporal path queries using the interval-centric computing model, and develop a novel *cost model* to select an efficient execution plan from several. We perform detailed experiments of our *Granite* distributed query engine using both static and dynamic temporal property graphs as large as 52M vertices, 218M edges and 325M properties, and a 1600-query workload, derived from the LDBC benchmark. We often offer sub-second query latencies on a commodity cluster, which is 149×–1140× faster compared to industry-leading Neo4J shared-memory graph database and the JanusGraph/Spark distributed graph query engine. *Granite* also completes 100% of the queries for all graphs, compared to only 32–92% workload completion by the baseline systems. Further, our cost model selects a query plan that is within 10% of the optimal execution time in 90% of the cases. Despite the irregular nature of graph processing, we exhibit a weak-scaling efficiency of $\geq$ 60% on 8 nodes and $\geq$ 40% on 16 nodes, for most query workloads.

Keywords: Graph processing, Temporal graphs, Distributed scheduling, Big data platforms, Query planning

1. Introduction

Graphs are a natural model to represent and analyze linked data in various domains. *Property graphs* allow vertices and edges to have associated *key–value pair* properties, besides the graph structure. This forms a rich information schema and has been used to capture knowledge graphs (concepts, relations) [1], social networks (person, forum, message) [2], epidemic networks (subject, infected status, location) [3], and financial and retail transactions (person, product, purchase) [4].

Path queries are a common class of queries over property graphs [5, 6]. Here, the user defines a sequence of predicates over vertices and edges that should match along a path in the graph. E.g., in the property graph for a community of users in Figure 1, the vertices are labeled with their *ID*s, their colors indicate their *type* – blue for *Person* and orange for a *Post*, and they have a set of properties listed as *Name:Value*. The edges are relationships,

[*] Corresponding author

Email addresses: shriramr@IISc.ac.in (Shriram Ramesh), animeshb@IISc.ac.in (Animesh Baranawal), simmhan@IISc.ac.in (Yogesh Simmhan)

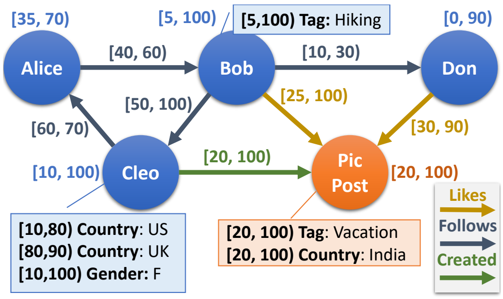


Figure 1: Sample Temporal Property Graph of a Community of Users

with *types* such as *Follows*, *Likes* and *Created*. We can define an example 3-hop path query "**[EQ1]** *Find a person (vertex type) who lives in the country 'UK' (vertex property) and follows (edge type) a person who follows another person who is tagged with the label 'Hiking' (vertex property)*". This query would match *Cleo→Alice→Bob*, if we ignore the time intervals. Path queries are used to identify concept pathways in knowledge graphs, find friends in social networks, fake news detection, and suggest products in retail websites [5, 6, 7]. They also need to be performed rapidly, within ≈ 1 *sec*, as part of interactive requests from websites or exploratory queries by analysts.

While *graph databases* are designed for transactional read and write workloads, we consider graphs that are updated infrequently but queried often. For these workloads, *graph query engines* load and retain property graphs in-memory to service requests with low latency, without the need for locking or consistency protocols [8, 9]. They may also create indexes to accelerate these searches [10, 11]. Property graphs can be large, with 10^5–10^8 vertices and edges, and 10's of properties on each vertex or edge. This can exceed the memory on a single machine, often dominated by the properties. This necessitates the use of distributed systems to scale to large graphs [12, 13].

Challenges. Time is an increasingly common graph feature in a variety of domains [14, 15, 16, 3]. However, existing property graph *data models* fail to consider it as a first-class entity. Here, we distinguish between graphs with a *time interval* or a *lifespan* associated with their entities (properties, vertices, edges), and those where the entities themselves change over time and the history is available. We call the former *static temporal graphs* and the latter *dynamic temporal graphs*. Yet another class is *streaming graphs*, where the topology and properties change in real-time, and queries are performed on this evolving structure [17, 18]; that is outside the scope of this article.

E.g., in the temporal graph in Figure 1, the lifespan, *[start, end)*, is indicated on the vertices, edges and properties. The *start* time is inclusive while the *end* time is exclusive. Other than the properties of *Cleo*, the remaining entities of the graph form a static temporal graph as they are each valid only for a single time range. But the value of the *Country* property of *Cleo* changes over time, making it a dynamic temporal graph.

This gap is reflected not just in the data model but also in the *queries supported*. We make a distinction between *time-independent (TI)* and *time-dependent (TD)* queries, both being defined on a temporal graph [19]. TI queries are those which can be answered by examining the graph at a single point in time (*a snapshot*), e.g. EQ1 executed on the temporal graph. In contrast, TD queries capture temporal relations between the entities across consecutive time intervals, e.g., "**[EQ2]** *Find people tagged with 'Hiking' who liked a post tagged as 'Vacation', before the post was liked by a person named 'Don'* ", and "**[EQ3]** *Find people who started to follow another person, after the latter stops following 'Don'* ". Treating time as just another property fails to express temporal relations such as ensuring time-ordering among the entities on the path. While EQ2 and EQ3 should match the paths *Bob→PicPost→Don* and *Alice→Bob→Don*, respectively, such queries are hard, if not impossible, to express in current graph databases. This problem is exacerbated for *path queries over dynamic temporal graphs*. E.g., the query EQ1 over the dynamic temporal graph *should not* match *Cleo→Alice→Bob* since at the time *Cleo* was living in 'UK', she was not following *Alice*.

While platforms which execute snapshot at a time [20, 19] can be adapted to support TI queries over temporal graphs, TD queries cannot be expressed meaningfully. Even those that support TD algorithms enforce strict temporal ordering [21], requiring that the time intervals along the path should be increasing or decreasing, but not both; this limits query expressivity. These motivate the need to support intuitive temporal predicates to concisely express such temporal relations, and flexible platforms to execute them. Lastly, the *scalability* of existing graph systems is also limited, with few property graph query engines that operate on distributed memory systems with low latency [8, 22], let alone on temporal property graphs.

We make the following specific contributions in this article:

- We propose a temporal property graph model, and intuitive *temporal predicates* and *aggregation operators* for path queries on them (§3).
- We design a *distributed execution model* for these queries using the interval-centric computing model (§4).
- We develop a novel *cost model* that uses graph statistics to select the best from multiple execution plans (§5).
- We conduct a detailed evaluation of the performance and scalability of *Granite* for 8 temporal graphs and up to 1600 queries, derived from the LDBC benchmark. We compare this against three configurations of Neo4J, and JanusGraph which uses Apache Spark (§6).

We discuss related work in Section 2 and our conclusions in Section 7.

A prior version of this work appeared as a conference paper [23]. This article substantially extends this. Specifically, it introduces the temporal aggregation operator to the query model (Section 3.3) and implements it within the execution model; offers details, illustrations and complexity metrics for our query model, distributed execution model and query optimizations (Sections 3, 4 and 5); and provides a rigorous empirical evaluation, including two additional large dynamic temporal graphs, aggregation query workloads, weak scaling experiments, and results on the component times of query execution, besides more detailed analysis for the cost model benefits and baseline platform comparisons (Section 6).

2. Related Work

2.1. Distributed and Temporal Graph Processing

There are several distributed graph processing platforms for running graph algorithms on commodity clusters and clouds [24]. These typically offer programming abstractions like Google Pregel's *vertex-centric computing model* [20] and its component-centric variants [25, 26] to design algorithms such as Breadth First Search, centrality scores and mining [27]. These execute using a Bulk Synchronous Parallel (BSP) model, and scale to large graphs and applications that explore the entire graph. They offer *high throughput batch processing* that take $O(mins)$–$O(hours)$. We instead focus on exploratory and transactional path queries that are to be processed in $O(secs)$. This requires careful use of distributed graph platforms and optimizations for fast responses.

There are also parallel graph platforms for HPC clusters and accelerators [28]. These optimize the memory and communication access to scale to graphs with billions of entities on thousands of cores [29]. They focus on high-throughput graph algorithms and queries over static graphs [30]. We instead target commodity hardware and cloud VMs with 10's of nodes and 100's of total cores, and are more accessible. We also address queries over temporal property graphs.

A few distributed abstractions and platforms support designing of temporal algorithms and their batch execution [31, 32, 19]. Most are limited to executing TI algorithms, snapshot at a time, and are unable to seamlessly model TD queries. Our prior work *Graphite* offers an interval-centric computing model (ICM) to represent TI and TD algorithms, but limits it to time-respecting algorithms [21]. We use it as the base framework for our proposed distributed path query engine, while relaxing the time-ordering, including indexing and proposing different query execution plans for low-latency response. There are also some platforms that support incremental computing over streaming graph updates [33, 34]. We rather focus on materialized property graphs with temporal lifespans on their vertices, edges and properties that have already been collected in the past. In future, we will also consider incremental query processing over such streaming graphs.

2.2. Property and Temporal Graph Querying

Query models over property graphs and associated query engines are popular for semantic graphs [35, 36, 30]. Languages like SPARQL offer a highly flexible declarative syntax, but are costly to execute in practice for large graphs [37, 38]. Others support a narrower set of declarative query primitives, such as finding paths, reachability and patterns over property graphs, but manage to scale to large graphs using a distributed execution model [39, 40]. However, none of these support time as a first-class entity, during query specification or execution.

There has been limited work on querying and indexing over specific temporal features of property graph. Semertzidis, et al. [41] propose a model for finding the top-k graph patterns which exist for the longest period of time over a series of graph snapshots. They offer several indexing techniques to minimize the snapshot search space, and perform a brute-force pattern mining on the restricted set. This multi-snapshot approach limits the pattern to one that fully exists at a single time-point and recurs across time, rather than spans time-points. It is also limited to a single-machine execution, which limits scaling.

TimeReach [42] supports conjunctive and disjunctive reachability queries on a series of temporal graph snapshots. It builds an index from strongly connected components (SCC) for each snapshot, condenses them across time, and use this to traverse between vertices in different SCCs within a single hop. It assumes that the graph has few SCCs that do not change much over time. They also require the path to be reachable within a single snapshot rather than allow path segments to connect across time. Likewise, *TopChain* [43] supports temporal reachability query using an index labeling scheme. It unrolls the temporal graph into a static graph, with time expanded as additional edges, finds the chain-cover over it, and stores the top-k reachable chains from each vertex as labels. It uses this to answer time-respecting reachability, earliest arrival path and fastest path queries. Paths can span time intervals. However, they do not support any predicates over the properties. Neither of these support distributed execution.

There is also literature on approximate querying over graphs. *Arrow* [44] examines reachability queries on both non-temporal and temporal graphs using random walks. These are performed from both the source and the sink vertices, and an intersection of the two vertex sets gives the result. They use approximation by bounding the walk length based on the diameter of the graph and a tunable parameter which balances accuracy and query latency. Iyer, et al. [45] consider approximate pattern mining on large non-temporal graphs. They use statistical techniques to sample the graph edges and estimate the number of occurrences of a specific pattern in the graph. However, their approach cannot enumerate the actual vertices and edges forming the pattern.

ChronoGraph [46] supports temporal traversal queries over interval property graphs, and is the closest to our work. They implement this by extending the Gremlin property graph query language with temporal properties. They propose optimizations to the Gremlin traversal operators, and parallelization and lazy traversals within a single machine, which are executed by the TinkerGraph engine. However, they do not support novel temporal operators such as the edge-temporal relationship that we introduce. They also do not use indexes or query planning to make the execution plan more efficient. Their optimizations are tightly-coupled to the execution engine, which does not support distributed execution.

Lastly, there are several open-source and proprietary graph database systems [8, 47, 48] which provide general-purpose property graph storage and querying capabilities while allowing transactional access to graph data. However, these systems do not have first class support for time-varying graphs and query models that can leverage the temporal dimension. This leads to temporal queries written in their native language which are neither intuitive in expressing temporal notion nor efficient during execution due to lack of time-aware query optimizer and execution engine.

In summary, these various platforms lack one or more of the following capabilities we offer: modeling time as a first-class graph and query concept; enabling temporal path queries that span time and match temporal relations across entities; and distributed execution on commodity clusters that scales to large graphs using a query optimizer that leverages the graph's structure, temporal features, and property values.

3. Temporal Graph and Query Models

3.1. Temporal Concepts

The temporal property graph concepts used in this paper are drawn from our earlier work [21]. Time is a linearly ordered discrete domain Ω whose range is the set of non-negative whole numbers. Each instant in this domain is called a *time-point* and an atomic increment in time is called a *time-unit*. A *time interval* is given by $\tau = [t_s, t_e)$

where $t_s, t_e \in \Omega$ which indicates an interval starting from and including t_s and extending to but excluding t_e. *Interval relations* [49] are Boolean comparators between intervals; *fully before* relation is denoted by $\ll$, *starts before* relation by $<$, *fully after* relation by $\gg$, *starts after* relation by $>$, *during* relation by $\subset$, *equals* relation by $=$, *during or equals* relation by $\subseteq$ and *overlaps* relation by $\sqcap$.

3.2. Temporal Property Graph Model

We formally define a temporal property graph as a directed graph $G = (V, E, P_V, P_E)$. V is a set of typed vertices where each vertex $\langle vid, \sigma, \tau\rangle \in V$ is a tuple with a unique vertex ID, vid, a vertex *type* (or schema) σ, and the *lifespan* of existence of the vertex given by the interval, $\tau = [t_s, t_e)$. E is a set of directed typed edges, with $\langle eid, \sigma, vid_i, vid_j, \tau\rangle \in E$. Here, eid is a unique ID of the edge, σ its type, vid_i and vid_j are its source and sink vertices respectively, and $\tau = [t_s, t_e)$ is its lifespan. We have a schema function $\mathcal{S} : \sigma \to K$, that maps a given vertex or edge type σ to the set of *property keys* (or names) it can have. P_V is a set of *vertex property values*, where each $\langle vid, \kappa, val, \tau_p\rangle \in P_V$ represents a value val for the key $\kappa \in K$ for the vertex vid, with the value valid for the interval $\tau_p \subseteq \tau$. A similar definition applies for edge property values $\langle eid, \kappa, val, \tau_p\rangle \in P_E$.

Further, the graph G must meet the *uniqueness* constraint of vertices and edges, i.e., a vertex or an edge with a given ID exist at most once and for a single continuous duration; *referential integrity* constraints, where the lifespan of an edge must be contained within the lifespan of its incident vertices; and *constant edge association*, which enforces that the vertices incident on an edge remain the same during the edge's lifespan. These are defined in [50].

A *static temporal property graph* is a restricted version of the temporal property graph such that $\tau_p = \tau$ for the vertex and edge properties, i.e., each property key has a static value that is valid for the entire vertex or edge lifespan, formally stated as:
$\forall\langle vid, \kappa, val, \tau_p\rangle \in P_V, \langle vid, \sigma, \tau\rangle \in V \implies \tau_p = \tau$ and $\forall\langle eid, \kappa, val, \tau_p\rangle \in P_E, \langle eid, \sigma, vid_i, vid_j, \tau\rangle \in E \implies \tau_p = \tau$
Temporal property graphs without this restriction are called *dynamic temporal property graphs*, and allow keys for a vertex or an edge to have different values for non-overlapping time intervals, i.e., $\tau_p \subseteq \tau$. E.g., Figure 1 is a dynamic temporal property graph as *Cleo*'s property values change over time, but omitting *Cleo* makes it a static temporal property graph.

3.3. Temporal Path Query

An *n-hop* linear chain path query matches a path with n vertex predicates and $n - 1$ edge predicates. The syntax rules for this query model and its predicates are given below, and illustrated for the example queries from earlier in Table 1.

$$\begin{array}{lcl}
\langle \textit{path} \rangle & ::= & \langle \textit{ve-fragment} \rangle\ \langle \textit{ve-int-fragment} \rangle^*\ \langle \textit{v-predicate} \rangle \\
 & | & \langle \textit{ve-fragment} \rangle\ \langle \textit{ve-int-fragment} \rangle^*\ \langle \textit{v-predicate} \rangle \oplus \langle \textit{aggregate} \rangle \\
\langle \textit{ve-fragment} \rangle & ::= & \langle \textit{v-predicate} \rangle \vdash \langle \textit{e-predicate} \rangle \\
\langle \textit{ve-int-fragment} \rangle & ::= & \langle \textit{ve-fragment} \rangle \ \mid\ \langle \textit{v-predicate} \rangle\ \langle \textit{etr-clause} \rangle \vdash \langle \textit{e-predicate} \rangle \\
\langle \textit{v-predicate} \rangle & ::= & \langle \textit{predicate} \rangle \\
\langle \textit{e-predicate} \rangle & ::= & \langle \textit{predicate} \rangle\ \langle \textit{direction} \rangle \\
\langle \textit{direction} \rangle & ::= & \rightarrow \mid \leftarrow \mid \leftrightarrow \\
\langle \textit{predicate} \rangle & ::= & \star \mid \langle \textit{bool-predicate} \rangle \mid \langle \textit{prop-clause} \rangle \mid \langle \textit{time-clause} \rangle \mid \langle \textit{time-clause} \rangle\ \texttt{AND}\ \langle \textit{bool-predicate} \rangle \\
\langle \textit{bool-predicate} \rangle & ::= & \langle \textit{prop-clause} \rangle \mid \langle \textit{prop-clause} \rangle\ \texttt{OR}\ \langle \textit{bool-predicate} \rangle \\
 & | & \langle \textit{prop-clause} \rangle\ \texttt{AND}\ \langle \textit{bool-predicate} \rangle \\
\langle \textit{prop-clause} \rangle & ::= & \texttt{ve-key}\ \langle \textit{prop-compare} \rangle\ \texttt{value} \\
\langle \textit{time-clause} \rangle & ::= & \texttt{ve-lifespan}\ \langle \textit{time-compare} \rangle\ \texttt{interval} \\
\langle \textit{etr-clause} \rangle & ::= & \texttt{el-lifespan}\ \langle \textit{time-compare} \rangle\ \texttt{er-lifespan} \\
\langle \textit{prop-compare} \rangle & ::= & `\texttt{==}' \mid `\texttt{!=}' \mid \ni \\
\langle \textit{time-compare} \rangle & ::= & < \mid \ll \mid > \mid \gg \mid \sqcap \mid \not\sqcap
\end{array}$$

Table 1: Query Syntax Examples

Example Query	**Query Syntax**
EQ1 Find a person who lives in 'UK' and follows a person who follows another person who is tagged with 'Hiking'	**Type** == *Person* AND **Country** == *UK* ⊢ **Type** == *Follows* → **Type** == *Person* ⊢ **Type** == *Follows* → **Type** == *Person* AND **Tag** ∋ *Hiking*
EQ2 Find people tagged with 'Hiking' who liked a post tagged as 'Vacation' before the post was liked by a person named 'Don'	**Type** == *Person* AND **Tag** ∋ *Hiking* ⊢ **Type** == *Likes* → **Type** == *Post* AND **Tag** ∋ *Vacation* `el-lifespan` ≺ `er-lifespan` ⊢ **Type** == *Likes* ← **Type** == *Person* AND **Name** == *Don*
EQ3 Find people who started to follow another person, after they stopped following 'Don'	**Type** == *Person* ⊢ **Type** == *Follows* → **Type** == *Person* `el-lifespan` ≫ `er-lifespan` ⊢ **Type** == *Follows* → **Type** == *Person* AND **Name** == *Don*
EQ4 Count the number of persons followed by a person 'Bob' during his existence in the network.	**Type** == *Person* AND **Name** == *Bob* ⊢ **Type** == *Follows* → **Type** == *Person* ⊕ `count` [★]

⟨*aggregate*⟩ ::= ⟨*aggregate-op*⟩[`v-key` | ★]

⟨*aggregate-op*⟩ ::= `count` | `min` | `max`

As we can see, the property and time *clauses* are the atomic elements of the *predicate* and allow one to *compare* in/equality and containment between a property value and the given value, and a more flexible set of comparisons between a vertex/edge/property lifespan and a given interval (*time-compare*). These temporal clauses allow a wide variety of comparison within the context of a single vertex or edge, and their properties. These clauses can be combined using Boolean AND and OR operators. Edge predicates can have an optional *direction*. The wildcard ★ matches all vertices or edges at a hop.

A novel and powerful temporal operator we introduce is *edge time relationship (ETR)*. Unlike the time clause, this *etr-clause* allows comparison across edge lifespans. Specifically, it is defined on an intermediate vertex in the path (*ve-int-fragment*), and allows us to compare the lifespans of its left (*el-lifespan*) and right (*er-lifespan*) edges in the path. The motivation for this operator comes from social network mining [6] and to identify flow and frauds in transactions networks [4]. E.g., the queries EQ2 and EQ3 from Section 1 can be concisely captured using this.

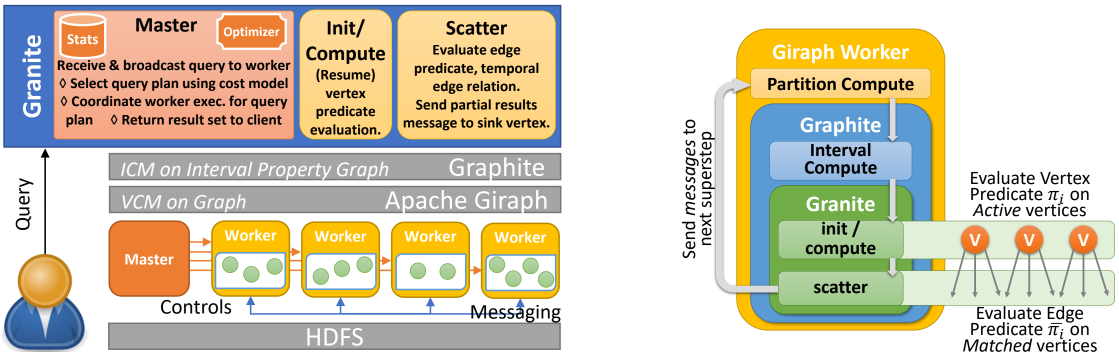
We also support a novel *temporal aggregate* operator to group the result-set from the path query. The paths are grouped on the first vertex in the resulting temporal paths, and computes a specific aggregation on a property at the last vertex of the path. The grouping is time-aware; specifically, it is based on the duration of the first vertex in the result path. E.g., if the result-set for a query contains $i = 1..m$ paths of length n each, $v_1^i - e_1^i - v_2^i - e_2^i - ... - v_n^i$, and the first vertex v_1^i in a result matches the query for the time period $\tau_i = [t_s^i, t_e^i)$, then we perform a "group by" of the result paths by the temporal vertex $\{v_1^i.id, [t_s^i, t_e^i)\}$. For all the paths j in a group, we perform an aggregation operation $\oplus$ on $v_n^j.prop$, where *prop* is a property on the last vertex that is selected by the user and may be omitted for a *count* aggregation. We return the aggregated result $\{v_1^i.vid, [t_s^i, t_e^i), \underset{j}{\oplus}(v_n^j.prop)\}$ for each unique temporal vertex group [1]. This can help answer queries such as *"[EQ4] Count the number of persons followed by a person 'Bob' during his existence in the network"*. The answer to this for Figure 1 varies across time, taking value 1 during $[10, 30) \cup [50, 100)$ and 0 during $[5, 10) \cup [30, 50)$. Our *Granite* implementation supports *count*, *min* and *max* operations for ⊕, while others can also easily be added.

4. Distributed Query Engine

4.1. Relaxed Interval Centric Computing

The high-level architecture of our distributed query engine, *Granite*, is shown in Figure 2a. Our query engine uses a distributed in-memory iterative execution model that extends and relaxes the *Interval-centric Computing Model*

[1] The valid duration for the first vertex can be disjoint, in which case each maximal contiguous interval for that vertex *vid* forms a separate temporal group.


Granite
Stats
Master
Optimizer
Receive & broadcast query to worker
◊ Select query plan using cost model
◊ Coordinate worker exec. for query plan
◊ Return result set to client
Init/ Compute
(Resume) vertex predicate evaluation.
Scatter
Evaluate edge predicate, temporal edge relation.
Send partial results message to sink vertex.
ICM on Interval Property Graph
Graphite
VCM on Graph
Apache Giraph
Query
Master
Worker
Worker
Worker
Worker
Controls
Messaging
HDFS
Giraph Worker
Partition Compute
Graphite
Interval Compute
Granite
init / compute
scatter
Send *messages* to next superstep
Evaluate Vertex Predicate π_i on *Active* vertices
V
V
V
Evaluate Edge Predicate π_i on *Matched* vertices

(a) Architecture of *Granite*

(b) Iterative query execution across ICM supersteps

Figure 2: Architecture and ICM execution model of *Granite*

(ICM) [21]. ICM adds a temporal dimension to Pregel's vertex-centric iterative computing model [20], and allows users to define their computation from the perspective of a single interval-vertex, i.e., the state and properties for a certain interval of a vertex's lifespan. In each iteration (*superstep*) of an ICM application, a user-defined `compute` function is called on each active interval-vertex, which operates on its prior state and on messages it receives from its neighbors, for that interval, and updates the current state. A *TimeWarp* function aligns the lifespans of the input messages to the lifespans of the partitioned interval states for an interval vertex. So each call to `compute` executes on the temporally intersecting messages and states for a vertex. Then, a user-defined `scatter` function is called on the out-edges of that interval-vertex, which allows them to send temporal messages containing, say, the updated vertex state to its neighboring vertices. The message lifespan is usually the intersection of the state and the edge lifespans.

Messages are delivered in bulk at a barrier after the `scatter` phase, and the `compute` phase for the next iteration starts after that. Vertices receiving a message whose interval overlaps with its lifespan are activated for the overlapping period. This repeats across supersteps until no messages are generated after a superstep. The execution of `compute` and `scatter` functions are each data-parallel within a superstep, and their invocation on different interval vertices and edges can be done by concurrent threads.

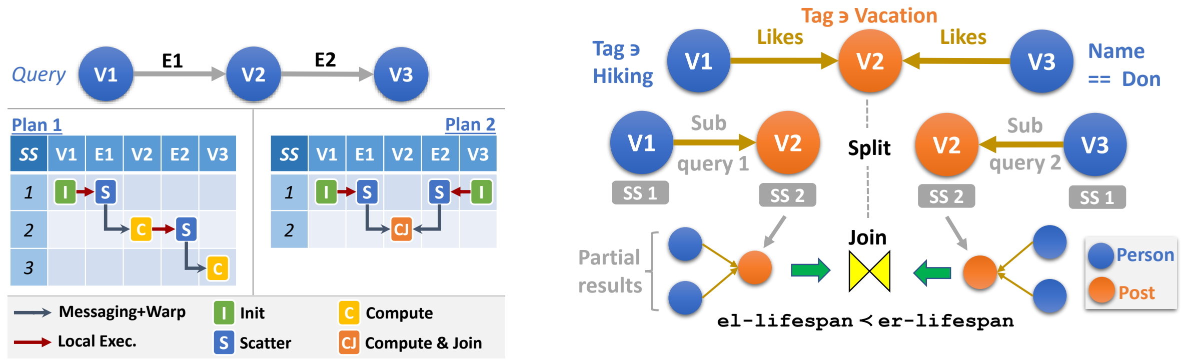
We design *Granite* using the `compute` and `scatter` primitives offered the *Graphite* implementation of ICM over Apache Giraph, as illustrated in Figure 2b. However, ICM enforces *time-respecting behavior*, i.e., the intervals between the messages and the interval-vertex state have to overlap for `compute` to be called on the messages; intervals between the states updated by the `compute` and the edge lifespans have to overlap for `scatter` to be called; and `scatter` sends messages only on edges whose lifespan overlaps with the updated states.

But the temporal path queries do not need to meet these requirements, e.g., a query may need to navigate from a vertex to an adjacent vertex that occurs *after* it. The TimeWarp operator of ICM enforces this time-respecting behavior. So we relax ICM to allow non-time respecting behavior between `compute`, `scatter` and messages to meet the execution requirements of our path queries, while leveraging its other interval-centric features.

4.2. Distributed Execution Model

In our execution model, each vertex predicate for a path query and the succeeding edge predicate, if any, are evaluated in a single ICM superstep. Specifically, the vertex predicates are evaluated in the `compute` function and the edge predicates in the `scatter` function. We use a specialized logic called `init` for evaluating the first vertex predicate in a query. This is shown in Figs. 2b and 3a.

A *Master* receives the path query from the client, and broadcasts it to all *Workers* to start the first superstep (Figure 2a). Each Worker operates over a set of graph partitions with one thread per partition, and each thread calls the `compute` and `scatter` functions on every active vertex in its partition. The `init` logic is called on all vertices in the first superstep. It resets the *vertex state* for this new query and evaluates the first vertex predicate of the query. If the vertex matches, its state is updated with a *matched* flag and `scatter` is invoked for each of its incident in or out edges, as defined in the query. `Scatter` evaluates the next edge predicate, and if it matches, sends the partial



(a) Query execution phases across different supersteps (SS) for two plans of the input query

(b) Splitting of query and Joining of results for *EQ2*, similar to Plan 2 in Figure 3a

Figure 3: Query execution plans in *Granite*

path result and the evaluated path length to the destination vertex as a message. If a match fails, this path traversal is pruned.

In the next iteration, our `compute` logic is called for vertices receiving a message. This evaluates the next vertex predicate in the path and if it matches, it puts all the partial path results from the input messages in the vertex state, and `scatter` is called on each incident edge. If the edge matches the next edge predicate, the current vertex and edge are appended to each prior partial result and sent to the destination vertex. This repeats for as many supersteps as the path length. In the last superstep, the vertices receiving matching paths in their messages send it to the Master to return to the client.

Figure 3a (Plan 1) illustrates this for a sample path query with vertex and edge predicates, $V1-E1-V2-E2-V3$. In superstep 1, `init` is called on all vertices to evaluate the vertex predicate $V1$, and for the ones that match, `scatter` is called to evaluate the edge predicate $E1$. Those edges that match send a message to their remote vertex in superstep 2, where all vertices that receive a message invoke their `compute` logic to evaluate the vertex predicate $V2$ of the second hop. This is (optionally) preceded by the TimeWarp operator on all the messages received by an interval vertex. Vertices that match $V2$ call `scatter` on their edges to match the predicate $E2$, and send messages if they too match. In the last superstep, vertices that receive messages evaluate the predicate $V3$, and if there is a match, return that result path to the user. Each vertex in the last superstep may return multiple matching paths based on the messages received, and different vertices may return result paths to the Master.

`Scatter` also evaluates the *edge temporal relationship*. Here, the `scatter` of the preceding edge passes its lifespan in the result message, and this is compared against the current edge's lifespan by the next `scatter` to decide on a match. In the case of *temporal aggregate queries*, the result set is constructed in the last superstep, similar to the non-aggregate queries. Then, the first vertex in each result path, its associated lifespan, and the count or the property value of the last vertex to be aggregated are extracted and sent to the Master. The Master temporally groups the values for each distinct temporal vertex using the *TimeWarp* operator, and applies the aggregation operator on the values in each group.

For *static temporal graphs*, we do not use any interval-centric features of ICM, such as TimeWarp, and the entire lifespan of the vertex is treated as a single interval-vertex for execution, and likewise for edges. However, we do use the property graph model and state management APIs offered by the interval-vertex.

For *dynamic temporal graphs* with time varying properties, we leverage the interval-centric features of ICM. Specifically, we enable TimeWarp of message intervals with the vertex properties' lifespans so that `compute` is called on an interval vertex with messages temporally aligned and grouped against the property intervals. `Scatter` is called only for edges whose lifespans overlap with the matching interval-vertex, and its scope is limited to the period of overlap. The `compute` or `scatter` functions only access messages and properties that are relevant to their current interval, and both can be called multiple times, for different intervals, on the same vertex and edge.

4.3. Distributed Query Execution Plans

Queries can be evaluated by splitting them into smaller *path query segments* that are independently evaluated *left-to-right*, and the results then combined. Each vertex predicate in the path query is a potential *split point*. E.g., a query $V1 - E1 - V2 - E2 - V3$ can be split at $V2$ into the segments: $V1 - E1 - V2$ and $V3 - E2 - V2$; execution proceeds *inwards*, from the outer predicates ($V1$ and $V3$) to the split vertex ($V2$) which *joins* the results. This is illustrated in Figs. 3a (Plan 2) and 3b. A trivial split at the last vertex predicate $V3$ is the default execution of the query from left-to-right, shown in Figure 3a (Plan 1), while an alternative split at the first vertex predicate $V1$ evaluates this from right-to-left as $V3 - E2 - V2 - E1 - V1$.

Each split point and plan can be beneficial based on how many vertices and edges match the predicates in the graph. Intuitively, a good plan should evaluate the most discriminating predicate first (low selectivity, few vertex/edges match) to reduce the solution space early. A *cost model*, discussed in Section 5, attempts to select the best split point.

We modify our *Granite* logic to handle the execution of two path segments concurrently. E.g., for a split point $V2$, in the first superstep, we evaluate predicates $V1 - E1$ and $V3 - E2$ in the same `compute/init` and `scatter` logic, while in the second superstep we evaluate predicate $V2$, as shown in Figure 3a (Plan 2). In the final superstep when results from both the segments are available, we do a nested loop *join* to get the cross-product of the results. This can be extended to multiple split points in the future.

4.4. System Optimizations

4.4.1. Type-based Graph Partitioning

Giraph by default does a hash-partitioning of the vertices of the graph by their vertex IDs onto workers. But we use knowledge of the entity schema types to create graph partitions hosting only a single vertex type. This helps us eliminate the evaluation of all vertices in a partition if its type does not match the vertex type specified in that hop of the query. This filtering is done before the `compute` is called, at the `partitionCompute` of Giraph.

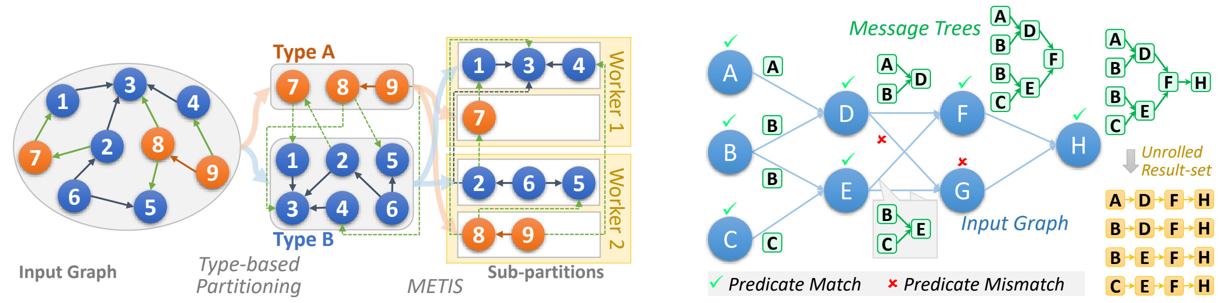
We first group vertices by *type* to form a *typed partition* each, e.g., *Type A* and *Type B*, as illustrated in Figure 4a. But these can have skewed sizes, and there may be too few types (hence partitions) to fully exploit the parallelism available on the workers and their threads. So we further perform a second-level *topological partitioning* of each typed partition into p sub-partitions using *METIS* [51]. This only considers the edges between vertices of the same type, i.e., within each typed partition, and uses the edge lifespan as their weight. This second-level partitioning can also reduce the network messaging cost between vertices of the same type. The sub-partitions from each typed partition are then distributed in a round-robin manner among all the workers. So if there are w workers, t types and p sub-partition per type, each worker is expected to have $\frac{t \times p}{w}$ sub-partitions, with $\frac{p}{w}$ of each type. Since each superstep typically evaluates a query predicate for a single vertex type, this ensures load balancing of the typed sub-partitions across all workers during a superstep execution.

In our experiments using the 100k:A-S graph, described in Section 6.1, we observe that using type-based partitioning at the first level instead of hash partitioning improves the average execution time for our query workloads by 5.8×. When we combine this with METIS partitioning in the second level, we see a further improvement of 32%. All our results we later report use this optimization.

4.4.2. Message Optimization

Path results can have a lot of overlaps. But each partial result path is separately maintained and sent in messages during query execution. This redundancy leads to large message sizes and more memory. Instead, we construct a *result tree*, where vertices/edges that match at a previous hop are higher up in the tree and subsequent vertex/edge matches are its descendants. E.g., assuming a full binary tree expansion for a path query with h hops and $n = 2^{h-1}$ matching paths, this reduces the result size from $O(h \times n)$ to $O(2n - 1)$. When execution completes, a traversal of this result tree gives the expanded result paths.

This is illustrated in Figure 4b. Here, vertices A, B and C match the vertex and edge predicates in the first hop and send their partial result to their neighbors. D receives the messages from A and B and evaluates itself for the second-hop predicate. But this execution is not unique to A or B, but rather shared across them. If D matches, rather than send a message with two sub-paths, $A - D$ and $B - D$, we instead send a sub-tree, $\{A, B\} - D$ in the message, to its neighbors. Similarly, E which receives messages fro B and C and matches for the second predicate sends a sub-tree $\{B, C\} - E$ message. F receives two sub-trees as messages, evaluates itself for the third predicate that matches, and



(a) Two-level load-balanced partitioning of Input Graph to Workers, by type and then by topology

(b) Message tree propagation during query evaluation

Figure 4: Examples of system optimizations in *Granite*

sends a larger sub-tree, $\{\{A, B\} - D\}, \{\{B, C\} - E\} - F$, to its neighbor H. G is not a match and prunes its traversal, with no messages sent. H matches the last predicate successfully, and sends the final result-tree with H as the root to the Master, which unrolls the tree to return the paths from H to every leaf as individual results to the client.

4.4.3. Memory Optimizations

In our graph data model, all property keys and values, excluding time intervals, are strings. In Java, string objects are memory-heavy. Since keys will often repeat for different vertices in the same JVM, we map every property key to a byte, and rewrite the query at the Master based on this mapping. Further, for property values that repeat, such as country, we use *interning* in Java that replaces individual string objects with shared string objects. This works as the graph is read-only. Besides reducing the base memory usage for the graph by $\approx 5\%$, it also allows predicate comparisons based on pointer equivalence.

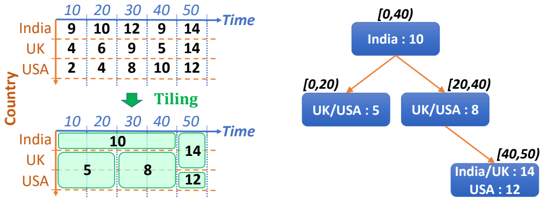
5. Query Planning and Optimization

A given path query can be executed using different distributed execution plans, each having a different execution time. The goal of the *cost model* is to quickly estimate the expected execution time of these plans and pick the optimal plan for execution. Rather than absolute accuracy of the query execution time, what matters is its ability to *distinguish poor plans* with high execution times *from good plans* with low execution times.

We propose an analytical cost model that uses statistics about the temporal property graph, combined with estimates about the time spent in different stages of the distributed execution plan, to estimate the execution time for the different plans of a given query. We first *enumerate* the possible plans, contributed by each split point in the path query. The graph statistics are then used to *predict* the number of vertices and edges that will be active at each superstep of query execution, and the number of vertices that will match the predicates in this superstep and activate the next hop of the query (superstep). Based on the number of active and matched vertices and edges, our cost model will *estimate* the runtime for each superstep of the plan. Adding these up across supersteps returns the estimated execution time for a plan. Next, we discuss the statistics that we maintain, and the models to predict the vertex and edge counts, and the execution time.

5.1. Graph Statistics

We maintain statistics about the temporal property graph to help estimate the vertices and edges matching a specific query predicate. Typically, relational databases maintain statistics on the frequency of tuples matching different value ranges, for a given column (property). A unique challenge for us is that the property values can be time variant. Hence, for each property key present in the vertex and edge types, we maintain a *2D histogram*, where the Y axis indicates the different value ranges for the property and the X axis the different time ranges. Each entry in the histogram has a count of vertices or edges that fall within that value range for that time range.



(a) 2-D Histogram of Statistics (b) Interval Tree for Statistics

Figure 5: Query planning

Table 2: Vertex and edge *count estimates* per superstep and *execution time* calculated by the model for two execution plans, for query *EQ*2 on 100k:F-S graph

Plan	**SS**	a_i	f_i	m_i	$\overline{a_i}$	$\overline{f_i}$	$\overline{m_i}$	T_i (ms)
1	1	$100k$	3.7×10^{-2}	$3.7k$	$6.2M$	$35M$	$1.3M$	531
	2	$1.3M$	7.7×10^{-4}	$1k$	–	–	–	132
2	1	$51M$	7.7×10^{-4}	$39k$	$273k$	$88M$	$67k$	4147
	2	$67k$	3.7×10^{-2}	$2.5k$	–	–	–	35

E.g., Figure 5a(top) shows such a histogram for the *Country* property. Its Y axis lists different country values appearing in the vertices of the property graph, such as *India, UK* and *US*. The X axis divides the lifespan of the graph into time intervals, say, $[0, 50)$ in steps of 10. The cell values indicate the number of vertices that have these property values for those time intervals, in the entire graph. Here, 9 vertices have the *Country* property value as *India* during time interval $[0, 10)$ and 10 vertices have it during $[10, 20)$, and similarly for other countries and time intervals.

Formally, for a given property key κ, we define a *histogram function* $\mathcal{H}_\kappa : (val, \tau) \rightarrow \langle f, \delta_{in}, \delta_{out} \rangle$, that returns an *estimate* of the frequency f of vertices or edges which have the property value *val* during a time interval τ, and the average in and out degrees δ of the matching vertices, which are maintained for a vertex property.

The *granularity* of the value and time ranges has an impact on the size of the statistics maintained and the accuracy of the estimated frequencies. We make several *optimizations* in this regard. We use Dynamic Programming (DP) to coarsen the ranges of the histogram along both axes to form a *hierarchical tiling* [52]. This ensures that the frequency variance among the individual *value–time* pairs in each tile is no more than a threshold. For example, in Figure 5a(bottom), the frequencies 9, 10, 12 and 9 for the property value *India* during the interval $[10, 40)$ are close to each other and hence tiled, i.e., aggregated and replaced by their average value 10. Similarly, *India* and *UK* have the same frequency 14 for the interval $[40, 50)$ and are tiled. This reduces the number of entries that are maintained in the histogram, i.e., the space complexity, while bounding its impact on the accuracy of the statistics.

For important properties like vertex and edge types, out-degree and in-degree, we *pre-coarsen the time steps* into, say, weeks, and for other properties into, say, months to reduce the size of the histogram – the actual coarsening factor is decided based on how often the properties change in the graph. For properties with 1000's of *enumerated values* (e.g., *Tag* in Figure 1), we sort them based on their frequency, cluster them into similar frequencies, and perform tiling on these clusters. We retain a map between property values and clusters for these, which is used to rewrite the input query to replace the property values with these cluster IDs instead.

We use an *interval tree* to maintain each histogram, with each tile inserted into this tree based on its time range. The nodes of the tree will have a set of tiles (property value ranges and their frequencies) that fall within its time interval. The invariant for all the nodes in the tree is such that the interval of a parent node will be after the left child (i.e., start time and/or end time of parent's interval is after the left child's interval), and before the right child. E.g., the interval tree in Figure 5b is constructed from the 2D histogram in Figure 5a(bottom). Every tile in the histogram becomes a node or part of a node in the interval tree. We insert a tile in the right subtree if its interval is greater than the parent node's interval, in the left sub-tree if it is lesser, and in the parent if it overlaps with it. To perform a lookup, we check if the lookup interval is greater than or less than the parent interval and prune the search space accordingly, similar to a binary search tree. Calling the $\mathcal{H}$ function performs a lookup in this interval tree, and matches within the set of property ranges.

The time complexity to construct each interval tree includes the time to *aggregate* the statistics from the graph, taking $O(n \cdot k)$, where n is the number of vertices in the graph and k the average number of property names or keys per vertex type. For each property key, the time taken is dominated by the *tiling* step that uses DP, and takes $O(p^3 t^3)$, where p is the number of (clustered) values for the property key, and t the number of (coarsened) time units they span [52]. The cost of building the interval tree is $O(m \cdot t)$, where m is the number of tiles in the coarsened histogram. The lookup time is $O(p \cdot t)$ in the worst case; for a balanced tree the expected lookup time is $O(log\ m + k)$, where k is the number of intersecting intervals in the tree. The raw size of the statistics for the graphs used in our experiments ranges from 4200–5600 kB for about 13–15 property keys.

5.2. Estimating the Active and Matching Vertex and Edge Counts

A query plan contains either one or two path query segments. The query predicates on each vertex and its edges in the segment are evaluated in a single superstep. If two path segments are present, their results are *joined* at the split point. Aggregation operators, if any, are also evaluated in the last superstep. For each segment, we estimate a count of active and matching vertices and edges in each superstep, given by the recurrence relation discussed next.

Let $P = [\pi_1, \overline{\pi_1}, ..., \pi_n]$ denote the sequence of n *vertex predicates*, π, and $n-1$ *edge predicates*, $\overline{\pi}$, for a given path query segment. Each predicate π has a set of *property clauses* $C_P(\pi) = \{\langle \kappa, val \rangle\}$ and a *temporal clause* $C_T(\pi) = \langle lifespan, \tau \rangle$, where κ is a property key, val is a value to compare its value against, and τ is the interval to compare that vertex/edge/property's lifespan against; and similarly for $\overline{\pi}$. These clauses themselves can be combined using `AND` and `OR` Boolean operators, as described in the query syntax earlier.

Let σ_i ($\overline{\sigma_i}$) denote the *type* of the vertex (edge) enforced by a clause of predicate π_i ($\overline{\pi_i}$). Let V_σ ($E_{\overline{\sigma}}$) denote the set of vertices (edges) of that type; if the vertex (edge) type is not specified in the predicate, these sets degenerate to all vertices (edges) in the graph.

As shown in Figure 2b, each superstep is decomposed into 2 stages: calling `init` or `compute` on the *active vertices* to find the vertices *matching* the vertex predicate, and calling `scatter` on the active edges (i.e., in or out edges of the matching vertices) to identify the edges matching the edge predicates. These in turn help identify the active vertices for the next superstep of execution. Initially, all vertices of the graph are active, but if a type is specified in the starting vertex predicate, we can use the type-based partitioning to limit the active vertices to the ones having that vertex type.

Let a_i and m_i denote the *number of active and matched vertices*, respectively, for vertex predicate π_i with type σ_i, and $\overline{a_i}$ and $\overline{m_i}$ denote the *number of active and matched edges*, respectively, for the edge predicate $\overline{\pi_i}$ with type $\overline{\sigma_i}$. These can be recursively defined as:

$$a_i = \begin{cases} |V_\sigma| & \text{, if } i = 1 \\ \min(\overline{m_{i-1}},\ |V_\sigma|) & \text{, otherwise} \end{cases} \tag{1}$$

$$\langle f_i, \delta^i_{in}, \delta^i_{out} \rangle = \bigotimes_{\substack{\langle \kappa, val\rangle \in C_P(\pi_i) \\ \langle lifespan, \tau \rangle \in C_T(\pi_i)}} \mathcal{H}_\kappa(val, \tau)$$

$$m_i = a_i \times \frac{f_i}{|V_\sigma|} \tag{2}$$

$$\overline{a_i} = m_i^\sigma \times (\delta^i_{in} + \delta^i_{out}) \tag{3}$$

$$\langle \overline{f_i}, -, - \rangle = \bigotimes_{\substack{\langle \kappa, val\rangle \in C_P(\overline{\pi_i}) \\ \langle lifespan, \tau \rangle \in C_T(\overline{\pi_i})}} \mathcal{H}_\kappa(val, \tau)$$

$$\overline{m_i} = \overline{a_i} \times \frac{\overline{f_i}}{|V_\sigma| \times (\overline{\delta}^\sigma_{in} + \overline{\delta}^\sigma_{out})} \tag{4}$$

In Equation 1, we set the active vertex count in the first superstep to be equal to the number of vertices of type σ. This reflects the localization of the search space in the `init` function to only vertices in the partitions matching that vertex type. For subsequent supersteps, the active vertex search space is upper-bounded by $|V_\sigma|$ but is usually expected to be the number of matching edges in the previous superstep [2], which would send a message to activate these vertices and call its `compute` function.

Next, in Equation 2, we use the graph statistics to find the fraction of vertices $\frac{f_i}{|V_\sigma|}$ that match the vertex predicate π_i (also called *selectivity*) and multiply this with the number of active vertices to estimate the matched vertices. This is the expected matched output count from `init` or `compute`. We use $\mathcal{H}$ to find the selectivity by iterating through all clauses of a predicate π_i, get their frequency, average in degree and average out degree of the vertex matches for each along with any temporal clause, and then aggregate (⊗) these frequencies. The aggregation between adjacent clauses

[2] This is in the worst case, if vertices and edges that match in the preceding hop activate mutually exclusive vertices in the next hop.

can be either AND or OR, and based on this, we apply the following aggregation logic for the frequencies and degrees.

$$f = \bigotimes(f1, f2) = \begin{cases} \min(f1, f2) & \text{, if } \otimes = \texttt{AND} \\ \max(f1, f2) & \text{, if } \otimes = \texttt{OR} \end{cases} \tag{5}$$

$$\delta = \bigotimes(\langle f_i, \delta_i \rangle, ...) = \frac{\sum_i f_i \times \delta_i}{\sum_i f_i} \tag{6}$$

Equation 5 returns the smaller of the frequencies while performing an AND, and the larger of the two with an OR; the former can be an over-estimate while the latter an under-estimate if the two properties are not statistically independent. Equation 6 finds the weighted average of the degrees of the vertices matching the predicates. Once the frequencies of the clauses are aggregated, we divide it by the number of vertices of this vertex type to get the selectivity for the vertex predicate.

Then, in Equation 3, we identify the number of edges for which `scatter` will be triggered by multiplying the matched vertices with the sum of the in and out degrees for the matching vertices δ. Lastly, in Equation 4 we estimate the number of edges matched by the edge predicate $\overline{\pi_i}$. Here, we get the edge selectivity using the frequency of edge matches returned by the graph statistics, and normalized by the number of preceding vertices of type σ, times the average of the in and out degrees of vertices of this type, δ. The edge selectivity is multiplied by the active edge count to get the matched edges that is expected from the `scatter` call. These edges will send messages to their destination vertices, and this will feed into the active vertex count in superstep $i + 1$.

E.g., Table 2 shows the cost model and statistics in action for query *EQ2* on graph *100k:F-S* that is described later in Section 6.1. It reports the counts for the active and matched vertex and edge counts (a, m) using Equations 1–4, and the frequency of the vertices and edges (f) as returned by the histogram, for each superstep of two different query plans. We see that a_1 is higher for Plan 2 than Plan 1 since the plans start at different vertex types during the `init` phase, and this will lead to different execution times for this phase (ι, discussed later). The frequency f_1 in Plan 1 is equal to f_2 in Plan, 2 and likewise for f_2 of Plan 1 and f_1 of Plan 2. This is expected since the predicate evaluated in superstep 1 of Plan 1 is same as that of superstep 2 in Plan 2. The cost model also estimates the messages sent $\overline{m_1}$ to be $1.3M$ and $67k$ for the two plans. Since we assume that the property values are independent, the selectivities remain constant. $a_2 = \overline{m_1}$ for both plans since we assume that each message from a superstep is sent to a unique vertex in the next superstep. While the compute calls for Plan 2 is higher, and the scatter calls and messages for Plan 1 is higher. The execution time model discussed next helps decide which of these plans has a lower estimated latency.

The clauses for time can also have comparators like $\succ$, $\prec$, etc. and property clauses can have ! =. These are supported by the histogram and cost model. E.g., we get the frequency for a $\prec$ operator by summing the frequencies for all values smaller than the given value, and for ! = by subtracting from the total frequency the frequency of values that equal the given value. All time-variant statistics are maintained in the histogram, while invariants such as the count of vertex and edges of each type are maintained as part of global statistics for the graph.

5.3. Execution Time Estimate

Given the estimates of the active/matched vertices/edges in each superstep, we incorporate them into execution time models for the different stages within a superstep to predict the overall execution time. We use micro-benchmarks to fit a *linear regression model* for the execution times, $\mathcal{I}, \mathcal{M}, \mathcal{S}, \mathcal{CC}$, and $\mathcal{IC}$, used below. These are unique to a cluster deployment of *Granite*, and can be reused across graphs and queries.

As shown in Figure 2b, the `init` function is called on the active vertices a_1 in the first superstep, and generates m_1 outputs that affect the states of the interval vertex. Its execution time estimate is given by the function $\iota = \mathcal{I}(a_0, m_0)$. For subsequent supersteps i, the `compute` function is called similarly on the active vertices, a_i, to generate the matched vertices m_i. This has a slightly different execution logic since it has to process an estimated $\overline{m_{i-1}}$ input messages from the previous superstep and does not have to initialize data structures, unlike `init`. Its execution time estimate is, $c_i = \mathcal{M}(a_i, m_i, \overline{m_{i-1}})$. In a superstep i, `scatter` is called on the active edges and generates matched edges, with an estimated time of $s_i = \mathcal{S}(\overline{a_i}, \overline{m_i})$. Besides these, there are per-superstep platform overheads: for iterating over vertices matching a given type, $cc_i = \mathcal{CC}(|V_\sigma|)$ in the *partitionCompute* phase, and a base overhead of $ic_i = \mathcal{IC}(a_i)$ per active vertex for *Graphite*.

Table 3: Cost model coefficients for linear regression fit for each execution phase, as used in our experiments

Init ($\mathcal{I}$)			**Compute ($\mathcal{C}$)**				**Scatter ($\mathcal{S}$)**			**Interval Compute ($\mathcal{IC}$)**		**Partition Compute ($\mathcal{CC}$)**	
a_0	m_0	cons.	a_i	m_i	$\overline{m_{i-1}}$	cons.	$\overline{a_i}$	$\overline{m_i}$	cons.	a_i	cons.	V_σ	cons.
9.4e-5	-3.1e-5	3.83	7.2e-5	3.3e-5	1.8e-5	1.63	7.9e-5	0	-3.81	-5.1e-6	8.6e-2	-8.0e-6	28.7

Given these, the total estimated execution time of the cost model for a query path segment with n hops is:

$$T = (\iota + s_1 + cc_1 + ic_1) + \sum_{i=2}^{n} c_i + s_i + cc_i + ic_i$$

In practice, these functions are determined by fitting simple linear regression models over query micro-benchmarks performed on the cluster on which the platform will be deployed. This is done once, and the functions are common for different graphs and query workloads on that cluster. E.g., Table 3 shows the coefficients for the linear equations that we fit for these functions, for the experiment setup in Section 6.2. Also, Table 2 shows the estimated execution time T_i in each superstep i for the two execution plans, using these coefficients. Plan 1 takes lesser time than Plan 2 due to the latter taking 7.8× longer in superstep 1. This is caused by a high `init` execution time, ι, since it has to evaluate $51M$ vertices (a_1) compared to only $100k$ in Plan 1. Since the total time is dominated by the `init` time, our cost model will choose Plan 1 for executing of this query.

We exclude the time to perform join and aggregation (for aggregate queries) from the cost model equation. This is based on our observation that this time is negligible (e.g., 20–30 *ms* in our experiments) compared to the overall execution time of a query (1000 *ms*) in most cases. In contrast, the execution time for `init` and the three `compute` functions together take about 900 *ms*. Further, the join and aggregate costs are proportional to the result set size. Even with a large result set size, there would inevitably be a large number of intermediate compute calls, and so the relative time taken by join and aggregate will remain low. Avoiding their inclusion helps keep the model concise, with only the most significant costs included. The time taken to find the optimal split point for a query using the approach described in this section is 2–9ms.

6. Results

6.1. Workload

We use the *social network benchmark* from the Linked Data Benchmark Council (LDBC) [53] for our evaluation of *Granite*. It is a community-standard workload with realistic transactional path queries over a social network property graph. There are two parts to this benchmark, a social network graph generator and a suite of benchmark queries.

Property Graph Datasets. The graph generator S3G2 [54] models a social network as a large correlated directed property graph with diverse distributions. Vertices and edges have a schema type and a set of properties for each type. Vertex types include *person, message, comment, university, country,* etc., while edge types are *follows*, *likes*, *isLocatedIn*, etc. The graph is generated for a given number of persons in the network, and a given degree distribution of the *person–follows–person* edge: *Altmann (A), Discrete Weibull (DW), Facebook (F)* or *Zipf (Z)*.

We make two changes to the LDBC property graph generator. One, we *denormalize* the schema to embed some vertex types such as *country, company, university* and *tag* directly as properties inside *person, forum, post* and *comment* vertices. This simplifies the data model. Two, while LDBC vertices are assigned a creation timestamp that can fall within a 3-year period, we include an end time of ∞ to form a time interval. We also add *lifespans* to the edges incident on vertices based on their referential integrity constraints, and replace time-related properties like *join date* and *post date* with the built-in lifespan property instead. The vertex and edge lifespans are also inherited by their properties. Figure 6 shows this modified graph schema.

However, this is still only a static temporal property graph. To address this, we introduce temporal variability into the properties, *worksAt*, *country* and *hasInterest* of the *person* vertex. For *worksAt*, we generate a new property every

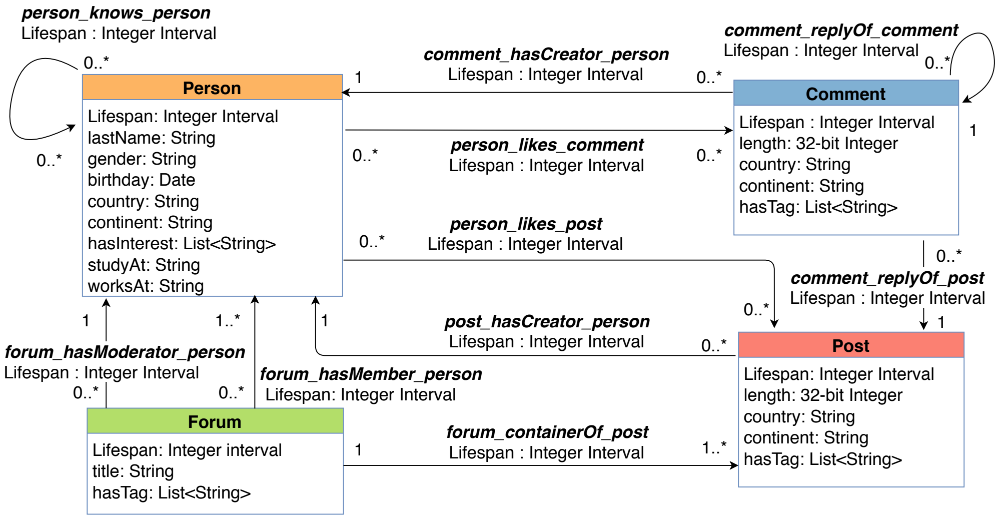


Figure 6: Modified LDBC Temporal Property Graph schema used in the evaluation

Table 4: Characteristics of graphs used in the experiments

			Frequent Vertex Types				*Frequent Edge Types*		**Unrolled**
Graph	|V|	|E|	**Persons**	**Posts**	**Comments**	**Forums**	**hasMember***	**hasCreator**†	**Properties**$^\#$
				Static Temporal Graphs					
10k:DW-S	5.5M	20.8M	8.9k	1.1M	4.3M	82k	3.3M	4.3M	35M
100k:Z-S	12.1M	23.9M	89.9k	7.4M	2.3M	815k	1.5M	2.3M	60M
100k:A-S	25.4M	78.2M	89.9k	8.7M	15.7M	816k	12.7M	15.8M	157M
100k:F-S	52.1M	217.6M	100k	12.6M	38.3M	996k	52.2M	38.4 M	325M
				Dynamic Temporal Graphs					
10k:DW-D	6.6M	29.3M	10k	1.4M	5.1M	100k	7.2M	5.1M	30M
100k:Z-D	15.2M	37.1M	100k	9.3M	4.8M	995k	3.2M	4.8M	57M
100k:A-D	32.0M	112.2M	100k	10.8M	20.1M	995k	25.6M	20.1M	132M
100k:F-D	52.0M	216.5M	100k	12.6M	38.2M	995k	51.8M	38.3M	222M

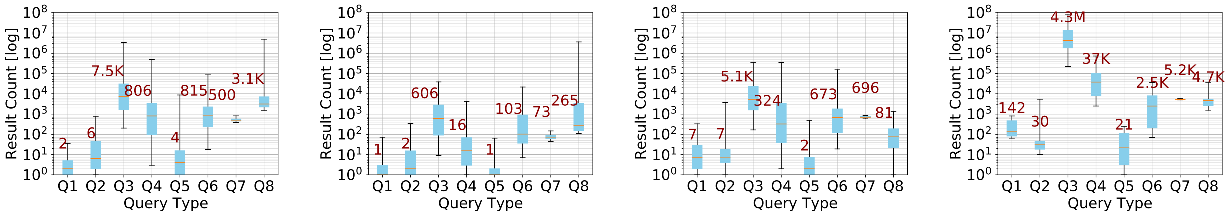
* *forum_hasMember_person* † *comment_hasCreator_person* $^\#$ *Unrolls multi-valued properties into individual ones*

year using the LDBC distribution; the *country* is correlated with *worksAt*, and hence updated as well. We update the *hasInterest* property based on the list of tags for a forum that a person joins, at different time points.

Table 4 shows the vertex and edge counts, the number of vertices of each type and the total number of property values, for graphs we generate with 10^4 (10k) or 10^5 (100k) persons, with different distributions (DW, Z, A, F), and with static (S, top) and dynamic (D, bottom) properties. As we see, the *Comments* type dominates the number of vertices, with up to 400 comments per person over a 3 year period, followed by about 100 *Posts* per person. The most frequent edge types are *forum_hasMember_person* and *comment_hasCreator_person*, while each person *Follows* 10.2 other friends on average. Properties such as *hasInterest* for person and *hasTag* for comment take up the most space since they are multi-valued, with an average of 23 interests per person and 1.22 tags per comment.

Query Workload. We select a subset of query templates provided in the LDBC query workload [53] that conform to a linear path query, and adapt them for our temporal graphs. Table 5 describes the query templates. These are either from the *Business Intelligence (BI)* or the *Interactive Workload (IW)*. We also include two additional query templates *Q*5 and *Q*6 to fully exercise our query model. Also, query template *Q*8 depends on *worksAt* which is a dynamic property and so it is only evaluated for dynamic temporal graphs.

Each template has some parameterized property or time value. We generate 100 query instances for each template by randomly selecting a value for the parameters, evaluating the query on the temporal graph, and ensuring that there



(a) 10k:DW (b) 100k:Z (c) 100k:A (d) 100k:F

Figure 7: Box and whiskers distribution plot of the result set count for the 100 instances of each non-aggregate query type. Q1–Q7 are reported on the static graphs while Q8 is on the dynamic graphs. The median result set count is labeled.

Table 5: Description of query workload used in the experiments

Query	LDBC ID	Hops	Property Predicates	Time Predicates	Has ER Predicate?	**Description of path to find** *(Parameterized property values are underlined)*
Q1	BI/Q9	3	4	1	Yes	Two *messages* with different *tags* belong to the same *forum*, with a time ordering between the messages
Q2	BI/Q10	2	6	1	No	A *person* with a given *tag* creates a *message* with the same *tag* after a given date.
Q3	BI/Q16	3	6	1	Yes	A *person* from a given *country* has commented or liked a *post* before a *person* from another given *country*.
Q4	BI/Q17	4	3	2	Yes	Mutual friendships between three *persons*, but with a time-respecting order in which they befriend each other.
Q5	–	5	7	3	Yes	A *person* posts a *message* with a given *tag* to a *forum* and, after a time offset, they post another *message* to the same *forum* with a different *tag*.
Q6	–	5	7	1	Yes	A *person* with a specific *gender* replies to a *post* after another *person* replies to it.
Q7	BI/Q23	4	5	3	Yes	A *person* posts a *message* from outside their home *country*, then befriends another *person*, and that *person* then posts another *message* from outside their home *country*.
Q8	IW/Q11	3	3	1	Yes	Two *persons* working in different *companies* have a common *friend* at a time-point.

is at least 1 valid result set in most cases. Query instances are generated for both the static and dynamic graphs. In addition to these *non-aggregate queries*, we also create another workload that includes a `count` temporal aggregate operator to these query templates, i.e., it will group the results of the original query by the first vertex and its time intervals, and return the count for each vertex-interval. This helps evaluate the performance of *aggregate queries*. For brevity, we limit these aggregate queries to the two largest graphs, *100k:A* and *100k:F*. Figures 7a, 7b, 7c and 7d show the distribution of the result set count for the (non-aggregate) queries on the different graphs in our workload.

These illustrate the expressivity of our query model, and ability to intuitively extend it to the time domain. The query length varies between 2 and 5 hops, allowing us to evaluate the cost model and *Granite* performance for different lengths. All the vertex types appear as predicates in our workload. The queries filter on both single-valued properties like *country* and *lastName*, and multi-valued properties like *hasInterest* and *hasTag*. All edge types except *forum_hasModerator_person* are used in the workload. 7 out of the 8 query types have ETR predicate and all the queries have at least 1 time predicate. They are diverse with respect to result sizes too, as shown in Figure 7a, 7b, 7c and 7d, and the result counts span several orders of magnitude, from 10^0–10^4.

In our experiments, each query is given an execution budget of 600 *secs*, after which it is terminated and marked as *failed*. The average execution times are only reported on the successful queries. We verify the correctness of all queries on *Granite* and baseline platforms. For the performance evaluations, the queries only return the count of the result sets for timeliness.

6.2. Experiment Setup

Our commodity cluster has 18 *compute nodes*, each with one Intel Xeon E5-2620 v4 CPU with 8 cores (16 HT) @ 2.10GHz, 64 GB RAM and 1 Gbps Ethernet, running CentOS v7. For some shared-memory experiments on other baseline graph platforms, we also use a "big memory" *head node* with 2 similar CPUs and 512 GB RAM. *Granite* is implemented over our in-house Graphite v1.0 ICM platform [21], Apache Giraph v1.3.0, Hadoop v3.1.1 and Java v8. By default, our distributed experiments use 8 compute nodes in this cluster, run one *Granite* Worker JVM per machine with 8 threads per Worker, and have 50 GB RAM available to the JVM. The graphs are initially loaded into *Granite* from JSON files stored in HDFS, with their pre-computed cost model statistics, and the query workloads run on this distributed in-memory copy of the graph.

6.3. Baseline Graph Platforms

We use the widely-used *Neo4J Community Edition v3.2.3* [47] as a baseline graph database to compare against. This is a single-machine, single-threaded platform. We use three variants of this. One specifies the workload queries using the community standard *Gremlin* query language (*N4J-Gr*, in our plots), and the other uses Neo4J's native *Cypher* language (*N4J-Cy*). Both these variants run on a single compute node with 50 GB heap size. A third variant uses Cypher as well, but is allocated $8 \times 50 = 400\,GB$ of heap space on the head node (*N4J-Cy-M*). As graph platforms are often memory bound, this configuration matches the total distributed memory available to our *Granite* setup by default. We build indexes on all properties in Neo4J.

There are few open source *distributed* graph engines available. *JanusGraph* [8], a fork from Titan, is popular, and uses *Apache Spark v2.4.0* as a distributed backend engine to run Gremlin queries (*Spark*, in our plots). It uses *Apache Cassandra v2.2.10* to store and access the input graph. Spark runs on 8 compute nodes with 1 Worker each and 50 GB heap memory per Worker. Cassandra is deployed on 8 additional compute nodes. This is based on the recommended configuration for JanusGraph on Cassandra [3]. Spark initially loads the graph from Cassandra into its distributed memory present on its 8 compute nodes. This load time is not considered as part of the query execution time. So effectively, only the 8 Spark nodes are used during query execution. For all baselines, we follow the standard performance tuning guidelines provided in their documentation [4] [5].

Since these platforms do not natively support temporal queries over *dynamic* temporal graphs, we transform the graphs into a static temporal graph using techniques described by Wu, et al. and used earlier by Graphite [43, 21]. This static property graph converts the time-intervals on vertices and edges of the original interval graph into an expanded set of vertices and edges that are valid for just a single discrete time point. This lets us adapt the query to operate on the static graph, albeit a bloated one. Also, temporal aggregation is not feasible internally on these platforms. So we perform the final aggregation at the client side for queries with an aggregate operator. JanusGraph/Spark is unable to load these two large graphs in-memory, and hence was not evaluated for the aggregate queries. The results from all platforms for all queries are verified to be identical.

6.4. Effectiveness of Cost Model

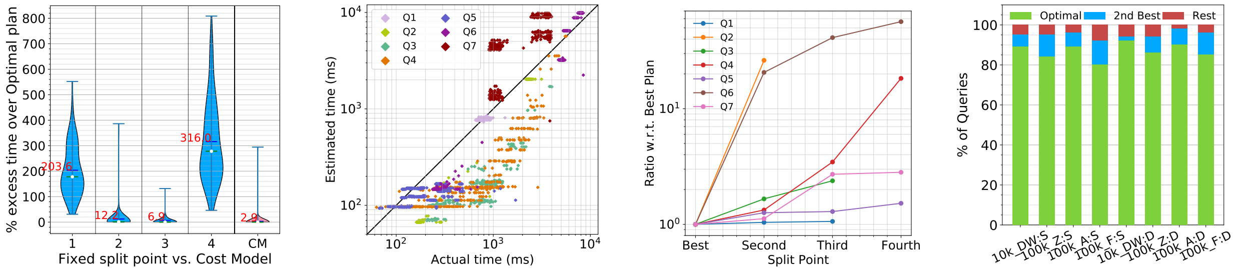
We first evaluate the effectiveness of *Granite*'s cost model in identifying the optimal split point for the distributed query execution. For each query type (template), we execute its 100 query instances using *all their possible query plans*, i.e., every possible split point is considered for each query. From the execution time of all plans for a query, we pick the smallest as its *optimal plan*. We compare this against the plan selected by our *cost model*, and report the *% of excess execution time* that our model-selected plan takes above the optimal plan. This is the effective time penalty when we select a sub-optimal plan.

Figure 8a shows a violin plot of the the distribution of this % excess time over optimal, for the different *fixed* split points 1–4 executed for the 100 queries of type $Q4$ (non-aggregate) on graph *100k:A-S*, compared to the plan selected by our cost model (CM) – lower this value, closer to optimal the performance. We see that the execution time varies widely across the plans, with some taking 8× longer than optimal. Also, some split points like 2 and 3 are in general better than the others, but among them, neither is consistently better. In contrast, our cost model plan has a low mean

[3] https://docs.janusgraph.org/storage-backend/cassandra/

[4] https://neo4j.com/docs/operations-manual/3.2/performance/

[5] https://docs.janusgraph.org/advanced-topics/hadoop/



(a) Distribution of queries that exceed the optimal plan's time by a % (Y axis), for each fixed plan and for the cost model, for *100k:A-S* graph on $Q4$ type queries

(b) Actual vs. Cost model estimated execution time for all query instances of *100k:A-S* graph, with correlation coefficient of $\rho = 0.87$

(c) Ratio of estimated average execution cost of the other plans relative to the optimal plan, for all query types of *100k:A-S* graph

(d) Cost Model Accuracy. % of times the optimal plan, 2^{nd} best plan and other plans were selected by our model for all graphs

Figure 8: Effectiveness of cost model in picking the best plan for non-aggregate queries

excess time of 2.9%, relative to 12.2% and 6.9% excess time taken by these other split points. Also, it is not possible to *a priori* find a single fixed split point which is generally better than the rest, without running the queries using all split points. These motivate the need for an automated analytical cost model for query plan selection.

We analyze the accuracy of the cost model for *100k:A*, the second largest graph, in more detail, and its impact on the execution cost. First, Figure 8b shows a scatter plot between the actual and the model-estimated execution time for the *100k:A-S* static graph; the plot has ≈ 2500 points. Overall, we see a high correlation coefficient of $\rho = 0.87$. There is an over-estimation for $Q7$ (maroon) due to an inaccurate estimation of the number of matching edges in the second hop, and under-estimates for $Q1$ to $Q5$. But $Q6$ (purple) shows a high correlation of $\rho = 0.94$.

Given these execution time inaccuracies of the model, we examine its effect on: (1) picking the optimal execution plan, and (2) on the latency penalty when it does not pick the optimal plan. Figures 9 show the fraction of times the cost model selects the optimal plan, the second best plan, and the rest of the plans, for the static and dynamic variants of *100k:A*, and for non-aggregate and aggregate queries. We also have corresponding data in Tables 6 which report for different query types (columns), and for different percentiles of their queries (rows), what is the % excess execution time over the optimal spent by the plan chosen by the cost model.

For the *non-aggregate queries*, the best or the second best plan were selected over 97% of the time across all queries, as seen in Figures 9a and 9b. For queries $Q2$, $Q4$, $Q6$ and $Q7$, the optimal plan was chosen 99% of the time. In $Q2$, this is due to a short query length of 2 that reduces the cumulative errors in the model, as well as a high difference in cost between the best and the second best plans. This is seen in Figure 8c, which gives the ratio of the 2^{nd}, 3^{rd} and 4^{th} best plan relative to the optimal. For $Q2$, the best plan evaluates the *person* vertices first, which are 500× fewer than the *message* vertices evaluated first by the other plan. As a result, the optimal execution time is 10× smaller than the other and the model easily selects the former plan. Similarly, $Q6$ also exhibits a high difference in cost between the optimal plan and the remaining three. But the top two plans for queries $Q4$ and $Q7$ have a similar cost. For $Q4$, starting at either ends causes a high fan-out and hence the plans that start at the two intermediate hops have a lower, but similar, cost. In such cases, as Figures 9a and 9b show, we may occasionally select the second best plan.

However, the consequence of choosing the second best plan on the actual execution latency is low when the top-2 plans have a similar model cost. In fact, for *100k:A-S*, we see from Table 6a that the execution time of the model-selected plan is within 2% of the optimal execution time for the 75^{th} percentile query, within a query type, and within 13% for the 90^{th} percentile query. Its only at the 95^{th} percentile query that we see higher penalties of 8–56% for 3 of the 7 query types. Even for the dynamic graph *100k:A-D*, 6 of the 8 query types have negligible time penalties at the 90^{th} percentile query in Table 6b, while two, $Q3$ and $Q8$, have higher penalties of 42–59%. The sub-optimal behavior happens when the execution model predicts a similar cost for the top-2 plans but selects the actual second-best, *and* the observed runtime for the second-best is much worse than the best. E.g., for the *100k:A-S* graph, the difference in actual execution cost between the optimum and second best plans for query $Q3$ is 18%. This causes the model to

Table 6: % excess time spent over the Optimal plan by the Cost Model selected plan, for different query percentiles of each query type

(a) 100k:A-S

%le	Q1	Q2	Q3	Q4	Q5	Q6	Q7
75	1.8	0	2.2	0	0	0	0
90	6.8	0	12.6	0	0	0	0
95	8.5	0	24.6	0	56	0	0
99	17.6	0	47.1	0	123	0	195

(b) 100k:A-D

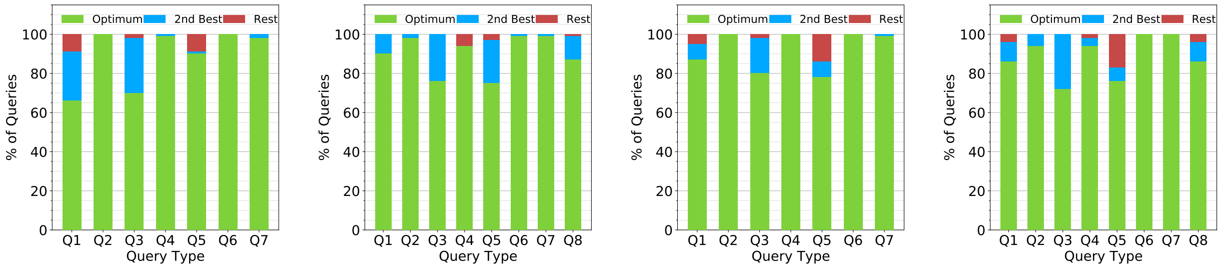
%le	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
75	0	0	0	0	0	0	0	0
90	0	0	42	0	7.1	0	0	59
95	2.4	0	124	66	8.8	0	0	112
99	3.6	0	198	191	12	0	0	277

(c) 100k:A-S (Temporal Aggregate)

%le	Q1	Q2	Q3	Q4	Q5	Q6	Q7
75	0	0	0	0	0	0	0
90	5.7	0	20	0	19	0	0
95	6.3	0	24	0	24	0	0
99	8.3	0	30	0	52	0	0

(d) 100k:A-D (Temporal Aggregate)

%le	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
75	0	0	0	0	0	0	0	0
90	4	0	6	0	28	0	0	57
95	12	28	21	132	39	0	0	175
99	18	145	84	166	57	0	0	643



(a) 100k:A-S (b) 100k:A-D (c) 100k:A-S, Temporal Agg. (d) 100k:A-D, Temporal Agg.

Figure 9: Cost Model Accuracy. % of times the optimal plan, the second best plan and the other plans were selected by our model

select the second best plan ≈ 28% of the time, and causes ≈ 5% of the queries to take 25% or longer to execute than the optimal plan.

We see similar trends for the *temporal aggregate queries* as well, in Figures 9c and 9d, and Tables 6c and 6d. The models predict the same costs for these aggregate queries since it ignores the aggregate operation and join costs due to their negligible overheads. Despite that, these queries perform on par or better than the equivalent queries without the aggregation step. In fact, this is broadly applicable to all the graphs, as observed in Figure 8d. It reports that across all queries and graphs evaluated, our cost model picks the best (optimal) or the second best plan over 95% of the time.

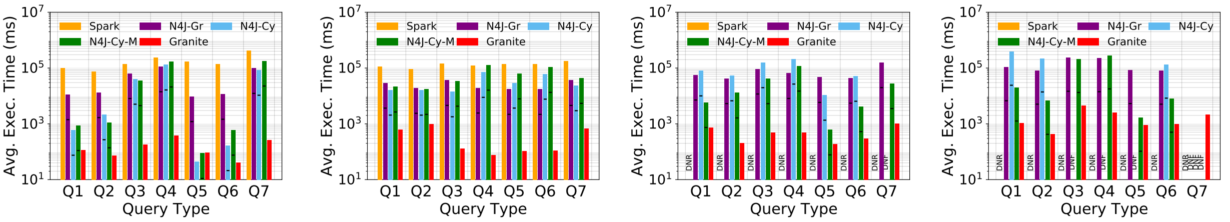
In summary, the cost model is accurate when the query is of shorter length, and accurate enough to distinguish between the similar good plans and the rest when certain predicate have high cardinalities. So we pre-dominantly pick a plan that is optimal, or has an execution time that is close to the optimal plan. Thus, while our cost model is not perfect, it is accurate enough to discriminate between the better and the worse plans, and consequently reduce the actual query execution time.

6.5. Comparison with Baselines

Figures 10 show the average execution time on *Granite* and the baseline platforms (Y axis, log scale) for the different *non-aggregate* query types (X axis) for the *static temporal graphs*, and Figures 11 for the *dynamic temporal graphs*. Only queries that complete in the 600 *sec* time budget are plotted. As Table 7 shows, Janus/Spark did not run (DNR) for several larger graphs due to resource limits when loading the graph in-memory from Cassandra. 32–79% of queries did not finish (DNF) within the time budget on Neo4J for *100k:F-S*, the largest graph. *Granite* completes all queries on all graphs, often within 1 *sec*. For the largest graph *100k:F-S*, *Granite* uses 16 nodes to ensure that the graph fits in distributed memory.

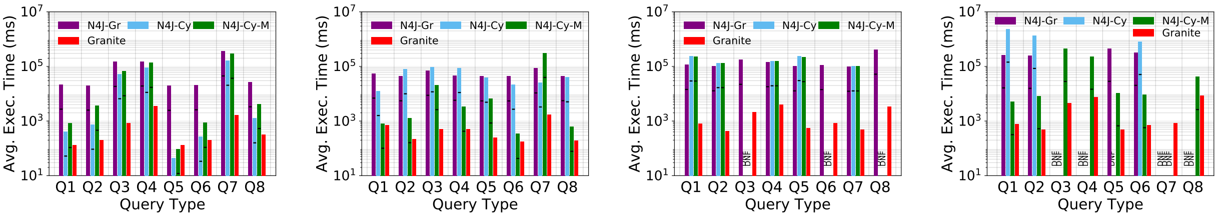
The bar plots show that *Granite* is much faster than the baselines, across *all* graphs and *all* query types, except for *Q*5 on the smallest graph, *10k:DW-S*. On average, we are 149× faster than N4J-Cy-M, 192× faster than N4J-Cy, 154× faster than N4J-Gr and 1140× faster than Spark. Other than the largest graph, *Granite* completes on an average within 500 *ms* for all static graphs and most query types, and on an average within 1000 *ms* for *100k:F-S* and all the dynamic graphs.

Focusing on specific query types for the largest static temporal graph, *100k:F-S*, *Q*2 takes the least time for *Granite* due to its short path length of 2. The left-to-right execution by the baseline platforms is the optimal query plan, but we are still able to out-perform them due to the parallelism provided by partitioning. *Granite* takes ≈ 5 *secs* for *Q*3 due to the huge number of results, ≈ 5.9*M* on average. But this query does not even complete for N4J-Cy and Spark.



(a) 10k:DW-S (b) 100k:Z-S (c) 100k:A-S (d) 100k:F-S

Figure 10: Comparison of average execution time of *Granite* with baseline systems for non-aggregate query types, on *Static Temporal Graphs*



(a) 10k:DW-D (b) 100k:Z-D (c) 100k:A-D (d) 100k:F-D

Figure 11: Comparison of average execution time of *Granite* with baseline systems for non-aggregate query types, on *Dynamic Temporal Graphs*

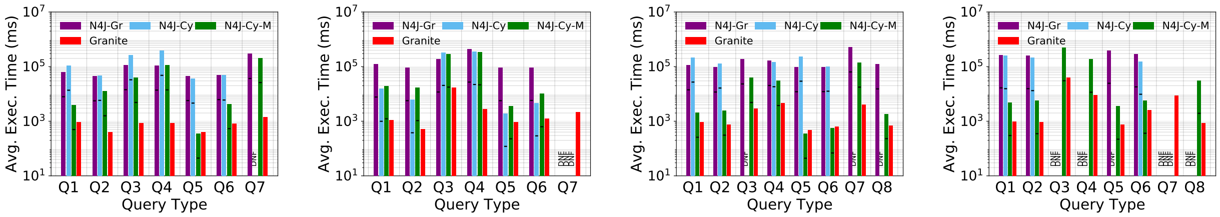
Granite's tree-based result structure is more compact, reducing memory and communication costs. *Q*4 for this graph is also 89–112× better in *Granite* than the baselines, with large result sizes of $\approx 72k$ on average. Here, there is a rapid fan-out of matching vertices followed by a fan-in as they fail to match downstream predicates, leading to high costs. *Q*7 queries are able to complete only in *Granite* and not on the baseline platforms. This query has an optimal split point of 1 or 2 which is not adopted by the baselines. In fact, baselines use the worst possible left-to-right plan, which we see is 4× slower than the optimal for *Granite*.

Granite is also consistently better for the *dynamic graphs*. Similar to the static graphs, the only time that our average query time is slower than a baseline is for *Q*5 on *10k:DW-D*. Here, the default left-to-right execution is near-optimal, and the query has a low traversal fan-out and < 10 results. So the baselines are in an ideal configuration while *Granite* has overheads for distributed execution.

Neo4J using Cypher, on the single compute node (N4J-Cy) and the big memory node (N4J-Cy-M), are the next best to *Granite*. The large memory variant gives similar performance as the regular memory one for the smaller graphs, but for larger graphs like *100k:A* and *100k:F*, it out-performs. For the latter graph, N4J-Cy could not finish several query types. Though Neo4J uses indexes to help filter the vertices for the first hop, query processing for later hops involves a breadth first traversal and pruning of paths based on the predicates. There are also complex joins between consecutive edges along the path to apply the temporal edge relation. These affect their execution times. Gremlin and Cypher variants of Neo4J are comparable in performance, with no strong performance skew either way. Interestingly, the Gremlin variant of Neo4J is able to run most query workloads for all graph, albeit with slower performance.

The *Janus/Spark* distributed baseline takes the most time for all these queries. This is despite omitting its initial graph RDD creation time (≈ 80 *secs*). *Granite* persists the graph in-memory across queries. Despite using distributed machines, Spark is unable to load large graphs in memory and often fails to complete execution within the time budget. A similar challenge was seen even for alternative engines like, Hadoop, used by JanusGraph and Spark was the best of the lot.

In the bar plots, we also show a black bar for the single-machine baselines, which is marked at the $1/8^{th}$ execution time-point – this shows the theoretical time that would be taken by these platforms if they had perfect parallel scaling on 8 machines, though they do not support parallel execution. As we see, *Granite* is often able to complete its execution within that mark, showing that our distributed engine shows scaling performance comparable or better than highly optimized single-machine platforms, even if they had ideal scaling.



(a) 100k:A-S (b) 100k:F-S (c) 100k:A-D (d) 100k:F-D

Figure 12: Comparison of average execution time of *Granite* with baseline systems for *Temporal Aggregate* query types

Table 7: % of queries that complete within 600 *seconds* for different platforms on the temporal graphs

Graph	Spark	N4J-Gr	N4J-Cy	N4J-Cy-M	*Granite*	Spark	N4J-Gr	N4J-Cy	N4J-Cy-M	*Granite*
	Static Graphs, Non-aggregate queries					*Dynamic Graphs, Non-aggregate queries*				
10k:DW	100	99	99	80	100	DNR	96	98	96	100
100k:Z	93	90	100	100	100	DNR	100	90	98	100
100k:A	DNR	100	90	98	100	DNR	97	46	46	100
100k:F	DNR	66	21	68	100	DNR	30	20	75	100
	Static Graphs, Temporal aggregate queries					*Dynamic Graphs, Temporal aggregate queries*				
100k:A	DNR	98	65	99	100	DNR	95	46	99	100
100k:F	DNR	60	68	65	100	DNR	36	19	78	100

Lastly, we compare the performance of *temporal aggregate queries* for the two largest static and dynamic graphs, *100k:A* and *100k:F*. Their execution times on the different platforms are shown in Figure 12. For the static graphs, we observe from Figures 12a and 12b that *Granite* is much faster than all the baselines for most query types. On average, we are 165× faster than N4J-Cy-M, 175× faster than N4J-Cy and 95× faster than N4J-Gr. This is 10× faster even when compared with the perfect scaling extrapolation for the baselines.

These temporal aggregate queries are slower compared to their non-aggregate equivalents. Specifically, for *100k:A-S*, *Granite* takes 64% ($\approx$ 315 *ms*) more on average while the baseline platforms on average are 56% (N4J-Cy), 42% (N4J-Gr) and 78% (N4J-Cy-M) slower, which translates to $\approx$ 24–53 *secs* longer, per query. The baselines' time increase considerably due to the additional overhead of sending the entire result set back to client to perform the temporal aggregation, as opposed to just sending the total number of results for the non-aggregate queries. Since *Granite* does this natively in a distributed manner, we mitigate this cost.

Granite completes all these queries when executed using the plan selected by the cost model (Table 7). The baseline platforms are only able to complete, on average, 79% (N4J-Gr), 67% (N4J-Cy) and 82% (N4J-Cy-M) of the queries on the static graphs, and this is worse for the dynamic graphs, ranging from 33%–89%.

Also, as Figures 12a and 12b show, we take under 1 *sec* to run all queries on *100k:A-S* except Q7, and within 2.1 *secs* for all queries on the largest graph, *100k:F-S*, except Q3 – query Q3 takes longer due to the large result count of $\approx$ 4.3*M* (Figure 7d). For dynamic graphs, we take under 3 *secs* for all queries on *100k:A-D* except Q4 and Q7, and within 9.2 *secs* for all queries on the largest graph, *100k:F-D*, except Q3 (Figures 12c and 12d). None of the baseline platforms could finish query type *Q*7 for *100k:F-S* or *100k:F-D*. This query starts and ends with the *Post* vertex type, which has a high cardinality. Also, these queries on the baseline platforms need to accumulate all the results for client-side aggregation. Both of these lead to memory-pressure for the larger graphs.

6.6. Components of Execution Time

Next, we briefly examine where the time is spent in distributed execution. As an exemplar, Figure 13 shows a stacked bar plot of the time taken by *Q*7 in different supersteps, and within different workers in a superstep, for the 100k:A-S graph. The stacks represent the time taken by the *init/compute*, *scatter*, and *join* phases of *Granite*, the *interval compute* parent phase of Graphite (ICM), the *partition compute* grand-parent phase of Giraph (VCM), and *other* residual time such as barrier synchronization and JVM garbage collection (GC), in each superstep. These times

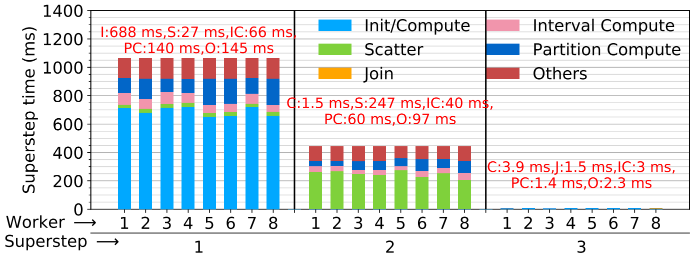


Figure 13: Stacked bar plot of component execution times in each superstep, averaged over all queries of query type *Q*7 on 100k:A-S graph. Header labels indicate average component time across Workers in a superstep.

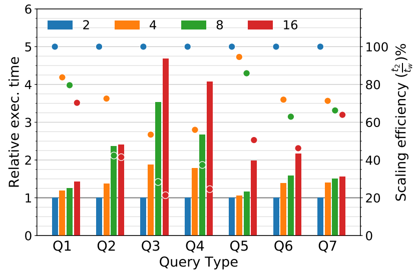


Figure 14: Relative execution time (left axis, bar) and Scaling efficiency%=$\frac{t_2}{t_w}$% (right axis, circle) for Worker counts $w = \{4, 8, 16\}$, relative to $w = 2$ for *Weak Scaling* runs with $(w \times 6.25k)$:F-S graphs

are averaged across all 100 instances of the query type. For deterministic execution, we select a fixed split point for the execution plan that is optimal for a majority of the queries, which, for *Q*7 is at the third vertex in the path.

For Q7, the first superstep time is dominated by the *init* logic as the predicate operates on the *Post* vertex type, which has 8.7*M* vertices. Its *scatter* time is minimal as only 71*k* out edges match out of 250*k* and are used to send messages. The overheads of *interval compute* are small, but *partition compute* takes longer at 140 *ms*. In the latter, the Giraph logic which we extend selects the active partitions based on the vertex type of the query predicate (*Post*, in the case of *Q*7), iterates through its active vertices, invokes interval compute on each with the incoming messages, and clears the message queue. The *other* time is non-trivial at 145 *ms*. This is caused by GC triggering due to memory pressure, and taking 110 *ms*, with the rest going to the superstep barrier.

In superstep 2, the *compute* time is negligible at 1.5 *ms* as only 3.4*k Person* vertices are active across both branches of the query plan, but *scatter* takes 247 *ms* since 2.83*M* edges are processed along one branch of the plan – the *Person* vertex has a high out-edge degree – out of which 31*k* satisfy the predicate. About 100 *ms* is taken by *partition and interval computes*, for selecting and iterating over the relevant active vertices, and for performing TimeWarp and state initialization, while there is a GC overhead of 64 *ms* in *other*. In the last superstep, there is a small time taken for *compute* and to *join* the results.

Interestingly, the time taken by each phase is similar across the different workers in a superstep for this query. This indicates that the partitioning manages to balance the load for this query type. However, for other queries like *Q*4 (not shown for brevity), we observe that in some supersteps, scatter takes 79% longer for the slowest worker compared to the fastest due to a skew in the number of edges activated per worker. Also, queries like *Q*4 take less time for the first superstep but a larger time in superstep 2 due to a high fanout, going from 36*k* edges processed in the first step to 1.48*M* edges in the second step. In others like *Q*3, the first superstep is dominated by scatter since the initial vertex type *Person* has only 89*k* vertices with 770 of them matching, but these cause 950*k* edges to be processed of which 122*k* match and trigger messaging.

In summary, the different supersteps have high variability in execution times and there is also variability in the time taken by each phase. Despite that, the cost model is able to discriminate and select near-optimal plans. The load is mostly balanced across workers in a superstep, though this depends on the query type. Much of the time is spent directly in processing the query using *compute* and *scatter*, with some additional overheads for the other phases.

6.7. *Weak Scaling*

We evaluate the weak scaling capabilities of *Granite* using the static Facebook-distribution graphs. We use 4 different system resource sizes – 2, 4, 8 and 16 Workers, with 1 compute node per Worker, and the graph sizes increase proportional to the Worker count – 12.5k:F-S, 25k:F-S, 50k:F-S and 100k:F-S. This attempts to keep the workload per Worker constant across the scaling configurations, with the per-Worker vertex and edge counts remaining within ±18% and ±23% of their mean, respectively. 100k:F-S is partitioned into 512 partitions (128 per vertex type), and other graphs into 256 partitions (64 per vertex type). This ensures that we have enough partitions for the compute threads to process them in parallel across all Workers. We generate and use a 100 query workload for each query type, for each graph.

The left Y axis of Figure 14 (bars) shows for each query type, the average relative execution time when using $w = 4, 8, 16$ Workers, compared to $w = 2$ Workers. The right Y axis (circles) shows the *scaling efficiency* $= \frac{t_2}{t_w}\%$, i.e., time taken on 2 Workers vs. time taken on w Workers. With perfect weak scaling, the relative time should be constant and efficiency 100%. The asymmetric nature of graph data structure makes it rare to get ideal weak scaling. However, we do see that query types $Q1$, $Q5$, $Q6$ and $Q7$ offer $\geq 60\%$ scaling efficiency on up to 8 Workers, and all queries but $Q3$ and $Q4$ have $\geq 40\%$ efficiency on up to 16 Workers. $Q3$ and $Q4$ are unable to fully exploit the additional resources due to stragglers among their threads, which are often 10× slower due to uneven load. These two queries also have the largest result cardinality, which causes more messages to be sent over the network as the number of machines increase. As a result, they have poor scaling efficiency.

7. Conclusions

In this article, we have motivated the need for querying over large temporal property graphs and the lack of such platforms. We have proposed an intuitive temporal path query model to express a wide variety of requirements over such graphs, and designed the *Granite* distributed engine to implement these at scale over the Graphite ICM platform. Our novel analytical cost model uses concise information about the graph to allow accurate selection of a distributed query execution plan from several choices. These are validated through rigorous experiments on 8 temporal graphs with a 1600-query workload, derived from the LDBC benchmark. *Granite* out-performs the baseline graph platforms and gives < 1 *sec* latency for most queries.

As future work, we plan to explore out of core execution models to scale beyond distributed memory, indexing techniques to accelerate performance, more generalized temporal tree and reachability query models, and compare performance with other research prototypes and metrics from literature. Designing incremental query execution strategies over streaming property-graph updates is also a related and under-explored challenge. The *Granite* platform is also finding relevance in analyzing epidemiological networks that form temporal property graphs constructed from, say, digital contact tracing for the COVID-19 pandemic. This may motivate the need for further query operators.

Acknowledgements

The first author of this work was supported by the Maersk CDS M.Tech. Fellowship, and the last author was supported by the Swarna Jayanti Fellowship from DST, India. We thank Ravishankar Joshi from BITS-Pilani, Goa for his assistance with the experiments.

[1] T. Mitchell, et al., Never-ending learning, Communications of the ACM 61 (2018).

[2] M. Cha, H. Haddadi, F. Benevenuto, K. P. Gummadi, Measuring user influence in twitter: The million follower fallacy, in: AAAI International conference on weblogs and social media (ICWSM), 2010.

[3] S. Liu, S. Poccia, K. S. Candan, G. Chowell, M. L. Sapino, epidms: data management and analytics for decision-making from epidemic spread simulation ensembles, The Journal of infectious diseases 214 (2016) S427–32.

[4] B. Haslhofer, R. Karl, E. Filtz, O bitcoin where art thou? insight into large-scale transaction graphs., in: International Workshop on Semantic Change & Evolving Semantics (SuCCESS), 2016.

[5] K. Shu, A. Sliva, S. Wang, J. Tang, H. Liu, Fake news detection on social media: A data mining perspective, ACM SIGKDD Explorations Newsletter 19 (2017).

[6] W. Fan, Graph pattern matching revised for social network analysis, in: International Conference on Database Theory (ICDT), 2012.

[7] Z. Huang, W. Chung, H. Chen, A graph model for e-commerce recommender systems, Journal of the American Society for information science and technology 55 (2004).

[8] Sharp, Austin et al., JanusGraph, 2020. https://janusgraph.org/.

[9] V. G. Castellana, A. Morari, J. Weaver, A. Tumeo, D. Haglin, O. Villa, J. Feo, In-memory graph databases for web-scale data, Computer 48 (2015).

[10] Y. Yan, C. Wang, A. Zhou, W. Qian, L. Ma, Y. Pan, Efficient indices using graph partitioning in rdf triple stores, in: 2009 IEEE 25th International Conference on Data Engineering, IEEE, 2009, pp. 1263–6.

[11] T. Milo, D. Suciu, Index structures for path expressions, in: International Conference on Database Theory, Springer, 1999, pp. 277–95.

[12] M. Junghanns, M. Kießling, N. Teichmann, K. Gómez, A. Petermann, E. Rahm, Declarative and distributed graph analytics with gradoop, Proceedings of the VLDB Endowment 11 (2018).

[13] B. Shao, H. Wang, Y. Li, Trinity: A distributed graph engine on a memory cloud, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, 2013, pp. 505–16.

[14] D. Greene, D. Doyle, P. Cunningham, Tracking the evolution of communities in dynamic social networks, in: 2010 International conference on advances in social networks analysis and mining, IEEE, 2010.

[15] B. George, S. Shekhar, Time-aggregated graphs for modeling spatio-temporal networks, in: Journal on Data Semantics XI, Springer, 2008.

[16] L. Zhao, G.-J. Wang, M. Wang, W. Bao, W. Li, H. E. Stanley, Stock market as temporal network, Physica A: Statistical Mechanics and its Applications 506 (2018).

[17] O. Walavalkar, A. Joshi, T. Finin, Y. Yesha, et al., Streaming knowledge bases, in: Fourth International Workshop on Scalable Semantic Web knowledge Base Systems, 2008.

[18] D. Ediger, J. Riedy, D. A. Bader, H. Meyerhenke, Tracking structure of streaming social networks, in: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, 2011.

[19] M. Then, T. Kersten, S. Günnemann, A. Kemper, T. Neumann, Automatic algorithm transformation for efficient multi-snapshot analytics on temporal graphs, Proceedings of the VLDB Endowment 10 (2017).

[20] G. Malewicz, et al., Pregel: a system for large-scale graph processing, in: SIGMOD, 2010.

[21] S. Gandhi, Y. Simmhan, An interval-centric model for distributed computing over temporal graphs, in: IEEE International Conference on Data Engineering (ICDE), 2020.

[22] K. Zeng, J. Yang, H. Wang, B. Shao, Z. Wang, A distributed graph engine for web scale rdf data, The VLDB Journal 6 (2013).

[23] S. Ramesh, A. Baranawal, Y. Simmhan, A distributed path query engine for temporal property graphs, in: IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid), 2020. doi:10.1109/CCGrid49817.2020.00-43.

[24] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, T. L. Willke, How well do graph-processing platforms perform? an empirical performance evaluation and analysis, in: IEEE IPDPS, 2014.

[25] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, V. Prasanna, GoFFish: A sub-graph centric framework for large-scale graph analytics, in: EuroPar, 2014.

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

[27] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, J. Cheng, G-miner: An efficient task-oriented graph mining system, in: ACM EuroSys, 2018.

[28] D. Gregor, A. Lumsdaine, The parallel bgl: A generic library for distributed graph computations, Parallel Object-Oriented Scientific Computing (POOSC) (2005).

[29] H.-V. Dang, R. Dathathri, G. Gill, A. Brooks, N. Dryden, A. Lenharth, L. Hoang, K. Pingali, M. Snir, A lightweight communication runtime for distributed graph analytics, in: IEEE IPDPS, 2018.

[30] J. Shi, Y. Yao, R. Chen, H. Chen, F. Li, Fast and concurrent {RDF} queries with rdma-based distributed graph exploration, in: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), 2016, pp. 317–32.

[31] Y. Simmhan, et al., Distributed programming over time-series graphs, in: IEEE IPDPS, 2015.

[32] W. Han, et al., Chronos: a graph engine for temporal graph analysis, in: ACM EuroSys, 2014.

[33] T. A. Zakian, L. A. Capelli, Z. Hu, Incrementalization of vertex-centric programs, in: IEEE IPDPS, 2019.

[34] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, E. Chen, Kineograph: taking the pulse of a fast-changing and connected world, in: ACM EuroSys, 2012.

[35] D. Chavarría-Miranda, V. G. Castellana, A. Morari, D. Haglin, J. Feo, Graql: A query language for high-performance attributed graph databases, in: IEEE IPDPS Workshops, 2016.

[36] J. Zhou, G. V. Bochmann, Z. Shi, Distributed query processing in an ad-hoc semantic web data sharing system, in: IEEE IPDPS Workshops, 2013.

[37] SPARQL Query Language for RDF, 2008. URL: http://www.w3.org/TR/rdf-sparql-query/.

[38] J. Huang, D. J. Abadi, K. Ren, Scalable sparql querying of large rdf graphs, VLDB Endowment 4 (2011).

[39] N. Jamadagni, Y. Simmhan, GoDB: From batch processing to distributed querying over property graphs, in: IEEE/ACM CCGrid, 2016.

[40] M. Sarwat, S. Elnikety, Y. He, M. F. Mokbel, Horton+: A distributed system for processing declarative reachability queries over partitioned graphs, PVLDB 6 (2013).

[41] K. Semertzidis, E. Pitoura, Top-*k* durable graph pattern queries on temporal graphs, IEEE TKDE 31 (2018).

[42] K. Semertzidis, E. Pitoura, K. Lillis, Timereach: Historical reachability queries on evolving graphs, in: EDBT, 2015.

[43] H. Wu, Y. Huang, J. Cheng, J. Li, Y. Ke, Reachability and time-based path queries in temporal graphs, in: ICDE, 2016.

[44] N. Sengupta, A. Bagchi, M. Ramanath, S. Bedathur, Arrow: Approximating reachability using random walks over web-scale graphs, in: 2019 IEEE 35th International Conference on Data Engineering (ICDE), 2019.

[45] A. P. Iyer, Z. Liu, X. Jin, S. Venkataraman, V. Braverman, I. Stoica, {ASAP}: Fast, approximate graph pattern mining at scale, in: 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18), 2018.

[46] J. Byun, S. Woo, D. Kim, Chronograph: Enabling temporal graph traversals for efficient information diffusion analysis over time, IEEE TKDE 32 (2019).

[47] Neo4j graph platform, 2020. URL: https://neo4j.com/.

[48] Orientdb graph database, 2020. URL: https://orientdb.com/graph-database/.

[49] J. Allen, Maintaining knowledge about temporal intervals, Communications of the ACM 26 (1983).

[50] V. Z. Moffitt, J. Stoyanovich, Temporal graph algebra, in: ACM International Symposium on Database Prog. Languages (DBPL), 2017.

[51] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM Journal on Scientific Computing 20 (1998).

[52] S. Muthukrishnan, V. Poosala, T. Suel, On rectangular partitionings in two dimensions: Algorithms, complexity and applications, in: International Conference on Database Theory (ICDT), 1999.

[53] The LDBC Social Network Benchmark (version 0.3.2), Technical Report, Linked Data Benchmark Council, 2019.

[54] M.-D. Pham, P. Boncz, O. Erling, S3g2: A scalable structure-correlated social graph generator, in: Selected Topics in Performance Evaluation and Benchmarking, 2013.