GeaFlow: A Graph Extended and Accelerated Dataflow System

ZHENXUAN PAN, Ant Group, China
TAO WU, Ant Group, China
QINGWEN ZHAO, Ant Group, China
QIANG ZHOU, Ant Group, China
ZHIWEI PENG, Ant Group, China
JIEFENG LI, Ant Group, China
QI ZHANG, Ant Group, China
GUANYU FENG, Ant Group, China
XIAOWEI ZHU, Ant Group, China

GeaFlow is a distributed dataflow system optimized for streaming graph processing, and has been widely adopted at Ant Group, serving various scenarios ranging from risk control of financial activities to analytics on social networks and knowledge graphs. It is built on top of a base with full-fledged stateful stream processing capabilities, extended with a series of graph-aware optimizations to address the space explosion and programming complexity issues of conventional join-based approaches. We propose new state backends and streaming operators that facilitate processing on dynamic graph-structured datasets, reducing space consumed by states. We develop a hybrid domain-specific language that embeds Gremlin into SQL, supporting both table and graph abstractions over streaming data. In addition to streaming workloads, GeaFlow is also extensively used for some batch processing jobs. In the largest deployments to date, GeaFlow is able to process tens of millions of events per second and manage hundreds of terabytes of states.

CCS Concepts: • **Information systems → Stream management**; **Graph-based database models**.

Additional Key Words and Phrases: stream processing, graph processing

ACM Reference Format:
Zhenxuan Pan, Tao Wu, Qingwen Zhao, Qiang Zhou, Zhiwei Peng, Jiefeng Li, Qi Zhang, Guanyu Feng, and Xiaowei Zhu. 2023. GeaFlow: A Graph Extended and Accelerated Dataflow System. *Proc. ACM Manag. Data* 1, 2, Article 191 (June 2023), 27 pages. https://doi.org/10.1145/3589771

1 INTRODUCTION

Stream processing has been widely used in various application domains, such as monitoring, recommendation, advertising, fraud detection, and many more [21, 22, 24, 31, 32, 39, 41, 45, 47, 48, 54, 62, 64, 66, 67, 83, 88]. While existing stream processing systems have proved to be effective for

Authors' addresses: Zhenxuan Pan, zhenxuan.panzx@antgroup.com, Ant Group, Hangzhou, China; Tao Wu, wt133031@antgroup.com, Ant Group, Hangzhou, China; Qingwen Zhao, qingwen.zqw@antgroup.com, Ant Group, Shanghai, China; Qiang Zhou, zhichen.zq@antgroup.com, Ant Group, Hangzhou, China; Zhiwei Peng, zhiwei.pzw@antgroup.com, Ant Group, Hangzhou, China; Jiefeng Li, jiefeng.ljf@antgroup.com, Ant Group, Hangzhou, China; Qi Zhang, zq268021@antgroup.com, Ant Group, Hangzhou, China; Guanyu Feng, fengguanyu.fgy@antgroup.com, Ant Group, Beijing, China; Xiaowei Zhu, robert.zxw@antgroup.com, Ant Group, Beijing, China.

2836-6573/2023/6-ART191 $15.00
https://doi.org/10.1145/3589771

handling windowed-aggregation-style queries, they face significant challenges when performing more complex processing on data streams that are organized as graphs.

The additional complexities come from both data and processing dimensions. Conventional streaming applications typically perform windowed aggregations whose states can usually be "reduced" to a size much smaller than the concerned window span. Even when states need to include all events within the window for processing, the fact that windows are often small (minutes to days) and moving in a tumbling or hopping fashion simplifies state management [41].

In contrast, a streaming graph query issues traversals on a constantly changing graph in response to events. The data-dependency characteristics of graph traversals make it necessary to hold vertex/edge data that may still be referenced with respect to the latest events. Furthermore, applications like risk control of credit and loan management need to access windows of states as large as months or even years that are several orders of magnitude larger than those in the cases of typical windowed aggregations.

The processing complexities further exacerbate the issue. While joining is known as the most expensive operation in batch processing, the dynamic nature of data streams makes it even more challenging in streaming scenarios. As streams to be joined are producing new data continuously, the system needs to index and materialize each stream as states, and performs look-ups into the other stream for each newly arrived item of any stream.

As a result, even though graph traversals can be emulated in existing stream processing systems with joins, excessive space and communication costs are inevitable in such implementations, as intermediate states (i.e. joined streams) have to be materialized and shuffled. The space consumption could keep growing as traversals go deeper. On the other hand, programming is also tedious and error-prone, even with declarative high-level domain-specific languages such as SQL due to the lack of graph semantics.

To address the aforementioned issues of existing solutions, we design and implement GeaFlow [1], an industrial-strength dataflow system that brings graph-parallel abstractions and optimizations to stateful stream processing. GeaFlow has been serving a wide variety of scenarios at Ant Group, and has undergone tests under which are possibly among one of the most challenging during each year's Singles' Day [2] Shopping Festival. Besides social networks and knowledge graphs that are natural applications for graph processing, the largest domain currently in use is financial risk control, in which vertices and edges are extracted from heterogeneous sources of events including transactions and logs, and a lot of graph queries are running continuously to assist critical tasks such as fraud detection and anti-money laundering.

GeaFlow is built on top of a base with full-fledged stateful stream processing capabilities. It supports distributed fault-tolerant dataflow program execution with exactly-once processing guarantees. Besides the ability to build general-purpose data processing pipelines, in order to enable efficient and scalable processing on graph-structured datasets, we enhance GeaFlow with several graph-oriented optimizations.

We first co-design new state backends and streaming operators that are tailored to the needs for processing on dynamic graphs. `GraphState` is provided as the common interface for managing vertex/edge data, with multiple backends implemented for different use cases. A set of vertex-centric [58] and edge-centric [73] streaming operators are offered for implementing custom graph processing logic in an iterative manner. We also develop traversal operators to increase the query processing throughput in a user-friendly way, which makes use of shared state accesses by merging the execution of a batch of inputs, and especially suits the high-throughput complex graph querying

[1]GeaFlow stands for the <u>g</u>raph <u>e</u>xtended and <u>a</u>ccelerated data<u>flow</u> system.

[2]https://en.wikipedia.org/wiki/Singles%27_Day

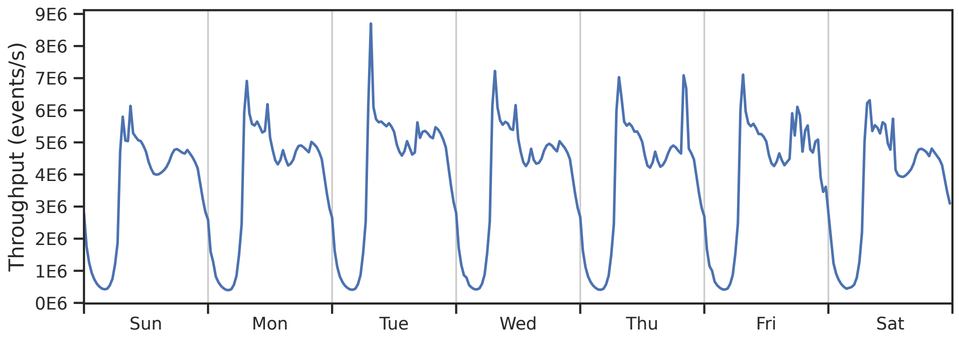


Fig. 1. Performance on one of the largest production workloads (average throughput per hour of a one-week duration)

workloads that GeaFlow targets. Other optimizations like off-heap memory management [1, 19], vertex-cut partitioning and mirroring [43, 44] are also explored and integrated into GeaFlow, with according streaming adaptations.

Based on the graph-aware state backends and streaming operators, we further develop a hybrid domain-specific language that embeds Gremlin [70], a popular graph query language, into SQL. We find that Gremlin alone is not sufficient to express some graph queries with complex matching and aggregation patterns, while the use of SQL as glue can address such limitations elegantly. The DSL not only relieves users from directly writing streaming graph processing code with GeaFlow's API, but also opens up some performance improvement opportunities, such as multi-query optimizations [49, 74, 89] that are cumbersome with manual coding.

In addition to streaming scenarios, GeaFlow is also extensively used for batch processing jobs. Besides processing on static graph snapshots constructed from historical events, there is an especially interesting usage model called windowed simulation. Under this model, historical events fallen within a selected simulation window (whose size is typically much larger than that of the state window) are replayed and fed into the system, as if the dataflow program is processing online events. This further increases the development efficiency, since queries verified to be working well on historical data can be ported to online applications with few extra efforts.

In our production environment, GeaFlow has been deployed on thousands of servers, serving hundreds of different applications. Figure 1 presents the average hourly throughput trend within a sampled week from one of our largest production workloads. This pipeline ingests multiple vertices and edges, and triggers a complex path query on a local subgraph expanded from a starting vertex for each event. The streaming graph contains about 1 billion vertices and 40 billion edges, with an overall space consumption of over 5 TB for managed states. The most space-hungry GeaFlow application manages states sized up to 300 TB while only using 200 containers, thanks to the holistic space optimizations. Our evaluation shows that GeaFlow outperforms Flink [32] significantly, on par with Differential Dataflow [61] but with notable space savings especially under deep traversals, which validates the effectiveness of proposed graph-specific extensions.

The rest of the paper is organized as follows. Section 2 provides an overview of the GeaFlow system, briefly introducing its overall architecture and highlighting main extended components. Section 3 describes the core optimizations, i.e. co-designed graph-aware state backends and streaming graph processing operators. Section 4 presents the hybrid SQL-Gremlin DSL, and related optimizations. Section 5 introduces the typical workloads powered by GeaFlow, including both stream and batch use cases. Section 6 shows evaluation results, comparing GeaFlow with Flink, the

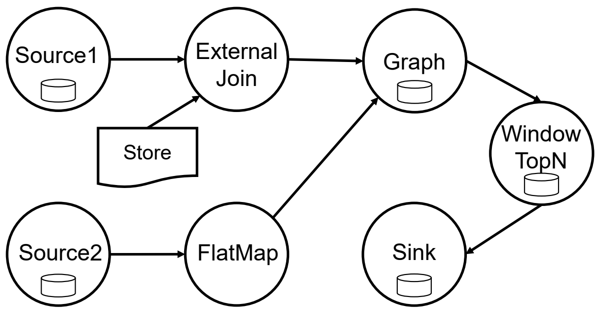


Fig. 2. A GeaFlow dataflow program example

de-facto dataflow system, along with design choice validations. Section 7 lists related work, and finally Section 8 concludes the paper with learned lessons and future exploration directions.

2 GEAFLOW OVERVIEW

GeaFlow targets event-driven applications that continuously process complex graph queries (e.g. deep traversals, subgraph matching, and even iterative global analytics). Taking financial risk control as a concrete example, where user accounts and money transfers are modelled as vertices and edges respectively. Sometimes we need to explore very long paths for analyses, since fraudsters have already been trying to construct increasingly complex transfer networks whose depths can be up to dozens, involving hundreds of thousands of mule [20] or zombie accounts [59], to obscure the audit trail.

It is clearly challenging for graph databases to support such complex graph queries especially under intense high-throughput scenarios. While adding more shards/replicas may scale the performance, this comes at a huge cost. On the other hand, graph processing systems are able to provide high throughputs, but suffer from excessively long latencies. To this end, we take the streaming approach which can exploit the fixed query patterns known in advance (as compared to ad-hoc alternatives) and make effective use of micro-batch execution to get a better balance between throughput and latency demands, i.e. achieving throughputs higher than databases and latencies lower than batch processing systems to meet the targeting requirements with an excellent cost efficiency.

Figure 2 gives an example of a typical GeaFlow dataflow program, consisting of source/sink operators for input/output, a graph operator for streaming graph processing, as well as non-graph transformation operators for pre-/post-processing. Input events can come from various types of data sources, mostly message queues and logging services that support replaying for fault tolerance. Vertices and edges are extracted from these events, and then merged[3] to the graph that are checkpointed periodically like conventional states. Some events can trigger graph queries[4] whose results are written to idempotent output sinks, mostly key-value stores that serve read requests in other pipelines independent to GeaFlow (e.g. generating features for later use), or sometimes message brokers connecting directly to downstream tasks (e.g. generating alerts when suspicious activities are detected).

[3]Usually "upsert" (update existing or insert new) semantics take place here, but deletions may be performed for retraction events according to users' needs.
[4]Some events may be used for only updates but trigger no queries; and some queries may further update user-defined states (e.g. to assist incremental computation).

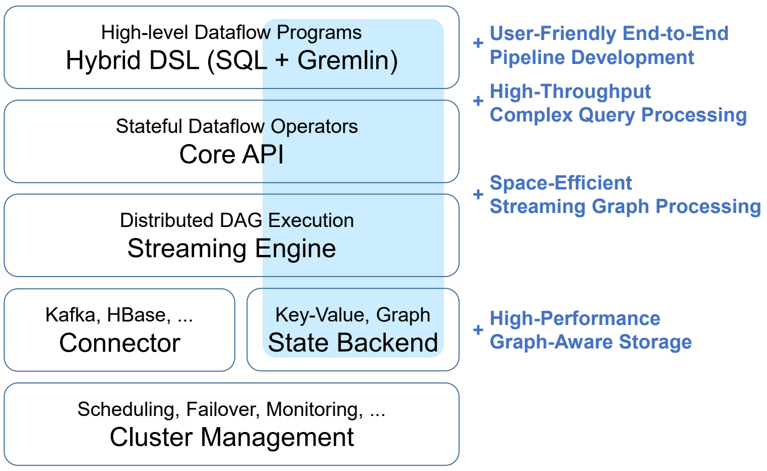

High-level Dataflow Programs
Hybrid DSL (SQL + Gremlin)
Stateful Dataflow Operators
Core API
Distributed DAG Execution
Streaming Engine
Kafka, HBase, ...
Connector
Key-Value, Graph
State Backend
Scheduling, Failover, Monitoring, ...
Cluster Management
+ User-Friendly End-to-End Pipeline Development
+ High-Throughput Complex Query Processing
+ Space-Efficient Streaming Graph Processing
+ High-Performance Graph-Aware Storage

Fig. 3. The GeaFlow software stack (shaded areas indicating graph extended and accelerated components)

Besides the graph-related extensions, GeaFlow is no more different from existing streaming solutions. We adopt a scale-out architecture like many popular stream processing systems [24, 32, 39, 83], to keep up with the ever-increasing large data volumes and make use of existing mature big data infrastructures. Users can use either the high-level DSL or the low-level API to define dataflow programs that are typically represented as directed acyclic graphs (DAGs) consisting of stateful operators as vertices and dependency relationships as edges. DAGs and affiliated operators are partitioned to support pipeline and data parallelisms [46]. Operators run continuously to keep latency low during normal execution [32, 64], with adaptive batching to keep up with high throughput demands during initialization, failover, reconfiguration, etc. [24, 86]. Consistent snapshots of operators' states are periodically created using checkpointing mechanisms [31], and transferred to reliable external storage for fault tolerance.

Figure 3 shows the overall software stack of GeaFlow, in which components with graph-aware optimizations are shaded. We provide the `GraphState` abstraction that offers functionalities beyond conventional keyed states, with a set of backends implemented with different advantages. Off-heap memory management techniques are applied to reduce GC pressure, and an optional graph object caching layer can be used to mask deserialization overheads [1, 19]. We support both vertex-centric [58] and edge-centric [43, 44, 72, 73] streaming operators, as well as both 1D (i.e. edge-cut) and 2D [44] (i.e. vertex-cut) graph partitioning policies, to enable flexible processing options under different circumstances. The co-designed states and operators together help us to address the space consumption issues in conventional streaming join-based approaches.

On top of GeaFlow's core API, we build the hybrid DSL that embeds Gremlin [70] into SQL, so that streaming queries over graph-structured datasets can be translated to corresponding graph-specific stateful operators with Gremlin, while the whole dataflow processing pipeline, including pre-processing Extract-Transform-Load (ETL) tasks (e.g. the "FlatMap" and "External Join" operators in Figure 2) and post-processing operations (e.g. the "Windowed TopN" operator in Figure 2 for aggregation) where tabular representations are more suitable, can be composed with SQL. This makes it easy to write end-to-end streaming graph processing pipelines and largely reduces development costs.

To sustain high throughputs for complex graph querying workloads that GeaFlow targets, we exploit the sharing opportunities in state accesses which usually appear to be the bottlenecks,

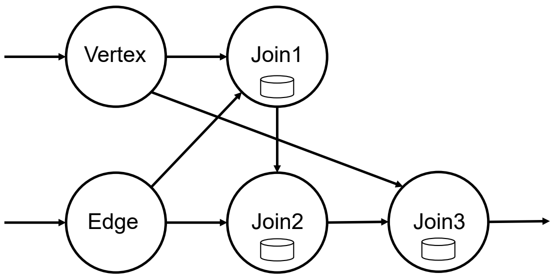


Fig. 4. A sample join-based partial DAG

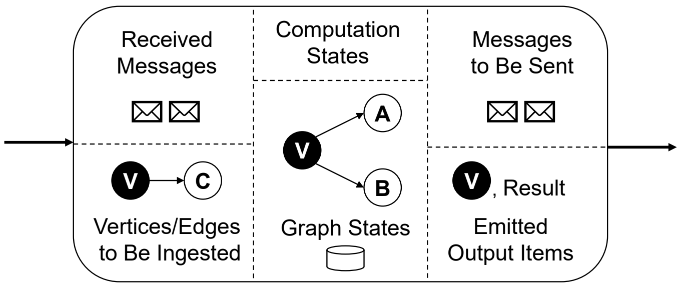


Fig. 5. An inner view of graph-specific operators

with the simple yet effective batching methodology. We devise a set of traversal operators and apply multi-query optimizations to batch the execution at inter-input and inter-query granularities respectively, which provides substantial performance improvements.

GeaFlow is written in Java, consisting of nearly 600K lines of code. About 1/4 of the code is related to our graph-specific extensions, i.e. the content described in this paper. While the base streaming framework is developed from scratch, we believe our extensions can also be applied to other general-purpose stream processing systems such as Flink.

3 GRAPH-SPECIFIC STATES AND OPERATORS

Before introducing our graph extensions towards states and operators in GeaFlow, we first illustrate the necessities of new mechanisms for streaming over graph-structured datasets.

Figure 4 shows a sample partial DAG fetching the data of 2-hop neighbors with three chained streaming join operators. The first join operator joins the vertex stream and the edge stream to get the direct 1-hop neighbors. The second join operator joins the output of the first join operator and the edge stream, emitting the 2-hop neighbors downstream. The third join operator joins the 2-hop neighbors and the vertex stream to fetch the data of those neighbors.

In the implementation of stream-stream join operators, we need to manage both input streams as keyed states, which inevitably leads to a huge space cost. Worse still, as the traversal goes deeper, the space can increase accordingly, e.g. the states managed by the third join operator in the above example would be significantly larger than the first one in size.

Even though we can manage states in out-of-core backends like RocksDB [35], or even external distributed storage systems like HBase [40], the space consumption is still hardly affordable for complex graph queries. Such implementations are also inefficient, since data accesses would then involve tons of unnecessarily expensive disk/network I/Os.

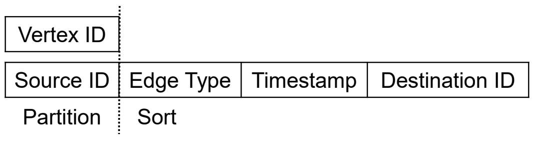


Fig. 6. The default key layouts for vertices/edges

To this end, we co-design new state backends and operators to replace the use of chained streaming joins for dynamic graph processing with graph-specific mechanisms. The key insight is to drive graph queries with iterative message passing between vertices, which decouples computation states [5] (i.e. transient data defined for computation) from managed states, avoiding the materialization of intermediate results in join-based approaches. The operators (e.g. the sub-DAG in Figure 4) are condensed to a single streaming graph processing operator (as shown in Figure 5) where each partition (of the operator) can read/write its local subgraph, use message passing to propagate updates, and perform iterative computations. Only ingested vertices and edges need to be maintained as states of the operator, partitioned and managed in backends that are implemented with graph-aware storage layouts to enable highly-efficient vertex/edge accesses, while other data such as messages, computation states, and input/output objects are only generated for each batch of computation and do not persist across batches.

3.1 Graph-Specific States

While conventional keyed state primitives can also be used for graph-structured states, e.g. Flink's `ValueState` for vertex data and `ListState` or `MapState`[6] for edge data, we find them insufficient for maximal efficiency due to missing semantics that are useful for graph processing.

The limitation of existing approaches stems from the unordered map abstraction that cannot exploit ordering to perform selective range scans. However, ordering is helpful in a variety of scenarios, and many storage backends including both embedded RocksDB and external HBase actually support ordering data by keys. As a straightforward example, it is common to iterate over only a specified type of edges in a heterogeneous graph (with multiple types of vertices and edges) for a specific query, sometimes with additional time range conditions [23, 25, 29]. In this case, partitioning by the source vertex alone is not enough; we need to further sort incident edges by types and timestamps to avoid touching useless data with partial range scans.

This leads us to a new `GraphState` abstraction to represent graph-structured states in GeaFlow. Besides basic mutation and point lookup (i.e. add/get/put/delete-vertex/edge) operations, it provides seekable iterator interfaces for locally managed vertex and edge states, so that prefix-based range scans can help to reduce unnecessary accesses. A set of corresponding backends are implemented, with some orthogonal optimizations layered on top of them. More details will be described next.

Key layout. Following the common practice, we use hash partitioning to split the whole key space into key-groups [31] which are the basic units to be used for state sharding. Concurrency control is not required thanks to co-partitioned states and operators. Vertex states are keyed and partitioned by vertex identifiers. For edge states, while a key always begins with the source vertex identifier which is also used for (1D) partitioning by default, the remaining part is used for sorting and supports custom layouts to optimize range scans for different use cases.

[5]They are usually known as vertex states in the literature of graph processing systems [72, 73, 81, 82, 92]. Here we use computation states [43] to differentiate them from vertex data managed by GeaFlow's state backends.
[6]`MapState` corresponds to an unordered map as well, which makes the keyed `MapState` in its entirety a nested two-level unordered map.

For example, while the default option (Figure 6), i.e. sorting incident edges of each vertex in the order of types, timestamps (descending order), and destination vertices, is beneficial for many situations, it is sometimes better to sort first by destination vertices, so that we can find out all edges between two known vertices efficiently without iterating over other irrelevant edges.

Vertex identifier mapping. In real applications, vertex identifiers are usually represented with business primary keys that can be long integers, strings, or even tuples. To compress the space consumption of vertex identifiers, we implement the vertex identifier mapping functionality, which maintains a map transforming original identifiers to compact integers allocated contiguously in each partition[7]. Vertex identifiers in keys of vertex/edge states are then replaced with mapped integers in storage.

The map can be an in-memory hash table, or maintained in an embedded key-value store such as RocksDB. For fault tolerance, the map is also checkpointed along with mapped data. Currently, only source vertex identifiers are mapped, as we find that mapping destination vertices as well would increase the space consumption of mapping in each partition significantly, which not only offsets the compression benefits provided by mapping, but also makes the mapping process itself expensive (i.e. the map may not be able to fit into memory, introducing additional I/O accesses).

Embedded backends. We implement two traditional embedded graph state backends in GeaFlow, i.e. a memory backend based on a hash table with each value containing a skip list, and a RocksDB-based backend for out-of-core storage. The memory backend provides higher performance but supports only limited capacity, thus is mostly used in development environments. The RocksDB backend is slower than the memory alternative due to inevitable serialization related overheads, but can scale up to very large graphs whose size is only limited by available disk space, and support conventional keyed states as well within a single storage engine.

External backends. GeaFlow supports various external graph state backends whose data are backed by external distributed stores or databases, including graph-oblivious ones like HBase, and graph databases like GeaBase [38]. While the key space is also hash-partitioned for parallel and distributed execution of the dataflow program, actual key sharding and (physical) state management is handled by external systems.

Since all operations need network communication, we take advantage of predicate pushdown capabilities offered by these storage backends, e.g. using HBase's scanner for filtering on keys or coprocessor for more complex pushdowns. When a graph database is used for state management, we can even delegate most of the graph processing logic to the backend, e.g. fetching a desired subgraph with the database's query language.

Graph-native backends. We design and implement two state backends with native graph organizations, for embedded and external usage respectively. Our solution is inspired by the key-value separation approach [55] that is optimized for managing large values. States can be stored in either local disks for low-latency accesses, or remote distributed file systems enabling nearly limitless space, with indexes always maintained locally in memory. A simplified illustration (e.g. the vertex identifier mapping table is not shown for brevity) is given in Figure 7.

The storage consists of two parts, an index and a value store. The value store is organized as a log-structured storage [71], containing log segments that are always written sequentially (i.e. append-only) and garbage collected when required to free space. The index is further composed of the main part and the delta part. The main part of the index forms a CSR-like (compressed sparse row) adjacency list, with each vertex corresponding to a contiguous region containing vertex/edge

[7]Note that vertices/edges are still partitioned by original identifiers.

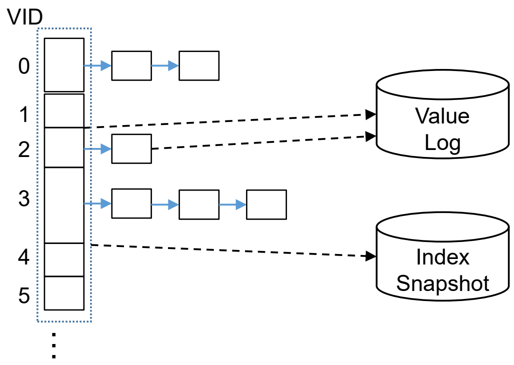


Fig. 7. A simplified illustration of graph-native backends

entries. In the delta part, each vertex is associated to a skip list, with each entry representing some recent vertex/edge updates that have not been merged to the main part.

Each entry in the index contains a key corresponding to either a vertex or an edge, and a pair of integers denoting the log segment id and the offset within that segment pointing to a location in the value store. All mutated vertex/edge states always go to the value store in the first place, and a new entry recording the location is also created and inserted to the delta part of the index.

When accessing a vertex or its incident edges, we first need to get its mapped vertex identifier so that we can locate both its main part and delta part in the index. For edge iterations, we maintain two pointers, corresponding to the main part and the delta part respectively. Since the main part and the delta part are both sorted, the implementations of seeks and scans are straightforward, i.e. similar to merging two sorted streams.

Due to the fact that iterating over skip lists inevitably introduce random accesses, and the values of each vertex's incident edges could also be scattered in non-contiguous locations, we schedule periodic compaction tasks to re-arrange the layout of the index and the value store to facilitate sequential accesses.

There are two types of compactions performed with different frequencies. The first is a fine-grained per-vertex compaction. Entries in the skip list are merged with entries in the main part, with expired or explicitly deleted entries removed. Corresponding values are read from locations indicated by these entries, and written sequentially to the value store. The merged entries (with updated locations) are then written to a new contiguous region in memory with variable-length difference encoding schemes [28]. The second type of compaction is more heavy-weight, and is thus triggered only when the values span too many segments. We iterate over each entry of each vertex, retrieve old values from locations specified by index entries, append them to new segments, update locations in entries, and finally free old segments.

When checkpointing happens, the current snapshot of the index is taken and saved to the remote storage. The log-structured nature of the value store naturally enables incremental checkpoints, and in the external version whose data is already backed by distributed file systems with redundancy, only some metadata needs to be written for checkpointing.

The improvements of our graph-native backends over other alternatives come from several aspects. The seek cost for locating a vertex is reduced from log-arithmetic of tree-based data structures to constant. While a seek within the per-vertex list is still log-arithmetic, the search range becomes much smaller, and the key-value separation further skips touching unnecessary values. The write amplification is also smaller, as most compactions are local to each vertex, and there are only two levels (the main part and the delta part) to be merged.

Graph object cache. According to our profiling results, de-/serializations in state accesses often appear to be the hotspots in GeaFlow applications except only when the embedded memory backend is used. To partly address this issue, we add a graph object caching layer on top of state backends to cache frequently accessed vertices/edges. On cache hits, we can avoid deserialization costs by accessing cached objects directly.

The cache consists of a vertex part and an edge part, and the size of each part can be configured separately. When either part runs out of space, the least recently used (LRU) object is evicted from that part of the cache. We leave the implementation of other policies and write-back caching as future work.

Multi-versioning. Another orthogonal extension regarding graph states is the multi-versioning ability, which can be used in a variety of scenarios to be introduced later. While each edge has a timestamp in its key which naturally supports multi-versioned accesses, we can extend vertex states to support this feature by suffixing a timestamp in each vertex's key. Then we can time-travel to a (historical) snapshot by providing a timestamp t and a window size w, consisting of all vertices/edges ingested within $[t - w, t)$. Note that vertex states return the latest version of each vertex (i.e. with the largest timestamp in the given window).

3.2 Graph-Specific Operators

To work with graph-specific states, we provide a set of graph-specific operators. Despite the differences in external abstractions and internal mechanisms, all graph-specific operators in GeaFlow share the same input and output interfaces. Inputs include (a batch of) vertex/edge objects collected from upstream operators, while outputs can be arbitrary types of data objects grouped by vertices, and emitted to downstream operators. Operators have access to maintained graph states, vertex/edge objects to be ingested, transient (i.e. not managed by state backends) computation states[8], and messages from other vertices. The partitioning scheme for an operator is aligned with its bound states, so that input objects can be consistently routed to the corresponding graph state shard for merging.

Processing model. While the whole GeaFlow system runs the continuous operator streaming model [32, 64] to ensure low latency, we adopt the BSP (bulk-synchronous parallel) model within our graph-specific operators for parallel and distributed execution of iterative graph processing tasks. Input vertex/edge objects are ingested in batches sized adaptively to balance between throughput and latency. All partitions of the operator perform the computation of each iteration in synchronized super-steps. This choice not only makes it easy to reason about the correctness of processing logic for complex graph queries, but also simplifies concurrency controls in state management, message passing, and vertex/edge processing. For example, We simply use synchronous checkpointing for graph states. When a worker is down, the cluster schedules a new worker for replacement. All workers then rollback states to the last checkpoint, and replay events from sources to continue streaming. Note that while the BSP model helps a lot in increasing the throughput (more details later), it inevitably introduces the overhead (tens to hundreds of milliseconds depending on the cluster scale) brought by the global synchronization barriers between iterations, which makes its average latency under very light loads (i.e. low-throughput and low-complexity scenarios) to be higher than that of Flink.

[8]Some space-efficient (i.e. constant complexity per-vertex/edge) states can be managed by backends to enable incremental computation; for brevity we consider these states as parts of the graph states.

Vertex-centric operator. We implement a vertex-centric operator in GeaFlow, providing a Pregel-like [58] interface. The main additions compared to the original Pregel abstraction include vertex/edge objects from upstream operators accessible as inputs, and output results to be collected and passed to downstream operators.

To use the vertex-centric operator, users need to define a `Compute` function, and the types of computation states attached on each vertex as well as messages to be used for graph processing. The user-defined `Compute` function specifies the processing logic in the view of a single vertex, and can perform actions including: (1) iterating over received messages sent in the previous iteration, (2) accessing vertex/edge objects emitted from upstream operators, (3) accessing and updating its own computation states, (4) updating managed graph states by mutating itself and/or incident edges, (5) sending messages to other vertices to propagate updates and schedule further computations in the next iteration, and (6) emitting outputs to downstream operators. An optional `Combine` function can be used to reduce the number of messages, by "aggregating" multiple messages to a single one.

By default, only vertices involved in the input batch are activated in the first iteration. But users can also change this behavior to scheduling all vertices to participate in the computation. Computation proceeds iteratively until there exist no active vertices, i.e. no messages are further sent out from any vertex.

Edge-centric operator. We also implement an edge-centric operator in GeaFlow, inspired by X-Stream/Chaos [72, 73] and PowerGraph/GraphX [43, 44]. The edge-centric operator decomposes the `Compute` function into three pieces: a `Scatter` function, a `Gather` function, and an `Apply` function.

`Scatter` generates an optional message to be sent along each edge (from the active source to the destination vertex). `Gather` applies the side effect of each received message to an accumulated value. `Apply` updates each vertex's computation states given the final accumulated result, and returns a boolean value indicating whether itself should be activated or not in the next iteration. The whole iterative computation terminates when there are no more active vertices, according to the results returned by `Apply`.

An additional `Initialize` function is provided with accesses to input vertex/edge objects which can be merged to managed graph states, and is only issued in the first iteration. Outputs can only be emitted in `Initialize` and `Apply`.

It is evident that the edge-centric operator has more restrictions than the vertex-centric counterpart. Operations need to be commutative and associative, akin to when `Combine` is effective. Meanwhile, message passing between arbitrary pairs of vertices other than those connected by edges are not supported yet, making some computation ill-formed. Nevertheless, decomposition of vertex-centric processing into more fine-grained edge-centric processing enables more optimization opportunities. There is no need to group messages by destination vertices anymore, and thus light-weight partitioning can be used instead of complete sorting or hash-table based maintenance of received messages [73]. Memory pressure can also be relieved as messages can be consumed by `Gather` on-the-fly.

Traversal operators. While a multi-hop graph traversal started from a single vertex can be mapped to an iterative graph processing program naturally, performing the same query on a batch of input vertices sequentially is clearly not the most efficient way especially in terms of throughput, as shown in existing literature [56, 91]. To take advantage of the BSP model and alleviate its drawbacks on synchronization overheads, we propose a batched execution approach which processes a number of query instances (the same query but initiated from different input vertices) concurrently via batching. This not only amortizes the brought overheads across a batch of inputs, but also exploits the localities arising from merged graph state accesses of different query

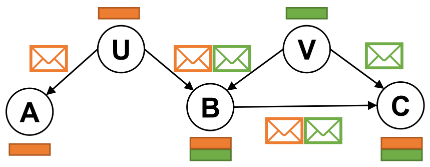


Fig. 8. Concurrent graph query processing with session-separated computation states and messages

instances in the same batch, especially in latter iterations of complex deep queries when more and more common vertices are activated.

Figure 8 gives an example of processing queries concurrently from two starting vertices "U" and "V". In the second iteration, vertex "B" is activated by both query instances, and thus accesses to its corresponding vertex and edge states can be shared if we schedule these two instances synchronously within a common batch and merge their processing on vertex "B", thereby reducing redundant I/O operations and cache misses.

Nevertheless, batching introduces additional programming complexities as we need to maintain computation states separately for each query instance (e.g. different colored elements in Figure 8). To this end, we build a set of traversal operators on top of vertex-/edge-centric operators to serve the needs for concurrent query processing. Users do not need to deal with concurrency, i.e. the user-defined functions only need to think as if only a query initiated from a single vertex is being processed.

Each query instance (corresponding to one input) acts like a separate session, and all passed messages are now attached with an additional session identifier. While operators still execute in BSP super-steps iteratively, they can differentiate multiple sessions according to the identifier. Computation states are similarly maintained for each session separately in a map, and can thus be accessed using identifiers as keys. Updates can be propagated to neighbors with messages, also tagged with the same identifier.

3.3 Joint State-Operator Optimizations

There are some other design and implementation choices that are related to both states and operators, listed as follows.

Vertex-cut partitioning and mirroring. Inspired by PowerGraph/GraphX [43, 44], we implement a hash-based 2D grid partitioning method, assigning each edge according to its source and destination[9]. More formally, for $p \times p$ partitions, we arrange them into a square containing p rows and p columns. We use two different hash-partitioning functions f and g, each of which maps a vertex identifier to an integer within $[0, p)$. Edge $(src \longrightarrow dst)$ will be assigned to partition $< f(src), f(dst) >$. As vertex vid could exist on all partitions of row/column $f(vid)$, i.e. partitions $< f(vid), * >$ and $< *, f(vid) >$, we denote partition $< f(vid), g(vid) >$ becoming the master and the rest being mirrors.

Note that vertex-cut partitioning can only be used with edge-centric operators. Since GeaFlow needs to manage graph states and there exist multiple replicas of each (source) vertex, for consistency considerations without involving complex concurrency control mechanisms, we route input vertex objects from upstream operators only to the primary partition in `Initialize`. Updates on the primary partition are then synchronized to mirrors. Computation states are handled similarly. `Apply` only executes on masters, and then updates are broadcasted and applied to mirrors.

[9]We currently only support this vertex-cut policy due to its simplicity and effectiveness, as additional complexities in streaming (compared to batching) e.g. bookkeeping of master/mirror locations can easily offset the benefits.

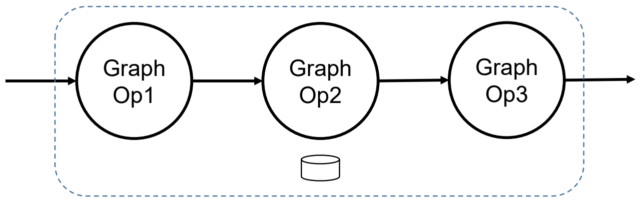


Fig. 9. Chained graph-specific streaming operators

The 2D partitioning policy can effectively address load imbalance issues caused by power-law degree distributions [43], especially in the context of concurrent multi-hop graph querying [76]. Another benefit of 2D partitioning, or more specifically, vertex replication, is that graph queries which need to fetch neighboring data (e.g. for filters or aggregations) may be completed in the current traversal step without additional messages to adjacent vertices. Nevertheless, this comes at the cost of space and write amplifications of vertex states, especially in the streaming scenario that GeaFlow targets. Thus, besides the skewness of degree distribution, the “read-heaviness” (i.e. the ratio of events triggering queries) is also an important factor for partitioning scheme choices.

Off-heap memory management. As GeaFlow is written in Java, GC (garbage collection) issues can easily become bottlenecks. We manage to mitigate the GC pressure with the use of off-heap memory management techniques that are commonly used in state-of-the-art big data systems [1, 19].

We exploit off-heap memory mainly in two aspects. The first is when disk/network I/Os are involved, e.g. reading from and writing to on-disk files, shuffling of generated messages, etc., which can reduce memory copies and occupation of the precious heap space. The second is regarding access to graph states. While our `GraphState` abstraction is schema-less which enables flexibility in data organizations, users can take advantage of fixed schemas when applicable (e.g. queries written with our hybrid DSL) by registering the type of each field for vertex/edge objects, so that operators can directly access states in binary representation on off-heap memory without deserializations, which further improves performance.

Sharing graph states. Sometimes users may want to share graph states among multiple graph processing operators. For example, a complex application can have a series of queries on the same graph, with some queries prefering edge-centric processing while others opting for vertex-centric operators.

While external state backends naturally allow sharing states, this approach is infeasible when low end-to-end latency is desired. Having each operator maintaining its own copy of the graph is also undoubtedly wasteful in this case.

To this end, we allow a chain of graph processing operators to share a common graph in GeaFlow, essentially forming a large logical operator (Figure 9). Each operator in the chain is able to collect input vertex/edge objects from its upstream operator, and emit output vertex/edge objects downstream, for connecting adjacent graph processing tasks while operating on the same shared graph.

This is attainable thanks to the adopted BSP model within graph-specific operators that simplifies concurrency control of state management since iterations (thus the whole computation of each operator) are separated by barriers. Note that partitioning of operators in the chain need to be aligned to make it work.

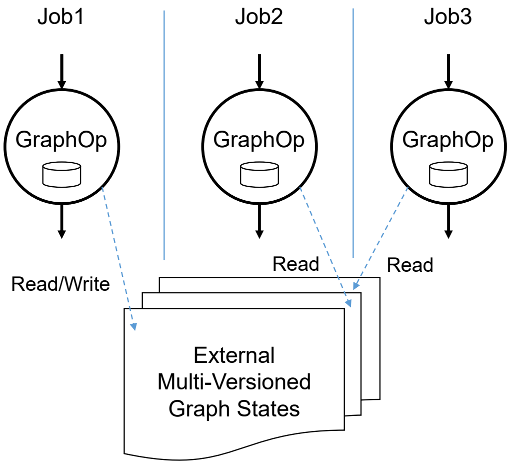


Fig. 10. Inter-job sharing of graph states

Besides inter-operator sharing within a job, we also support a restricted form of sharing among multiple GeaFlow jobs. As illustrated in Figure 10, shared graph states are accessible in a multi-reader single-writer fashion with the help of external graph-native backends with multi-versioning enabled. One of the jobs can apply updates onto the graph states, while others can read the changes by following the write logs.

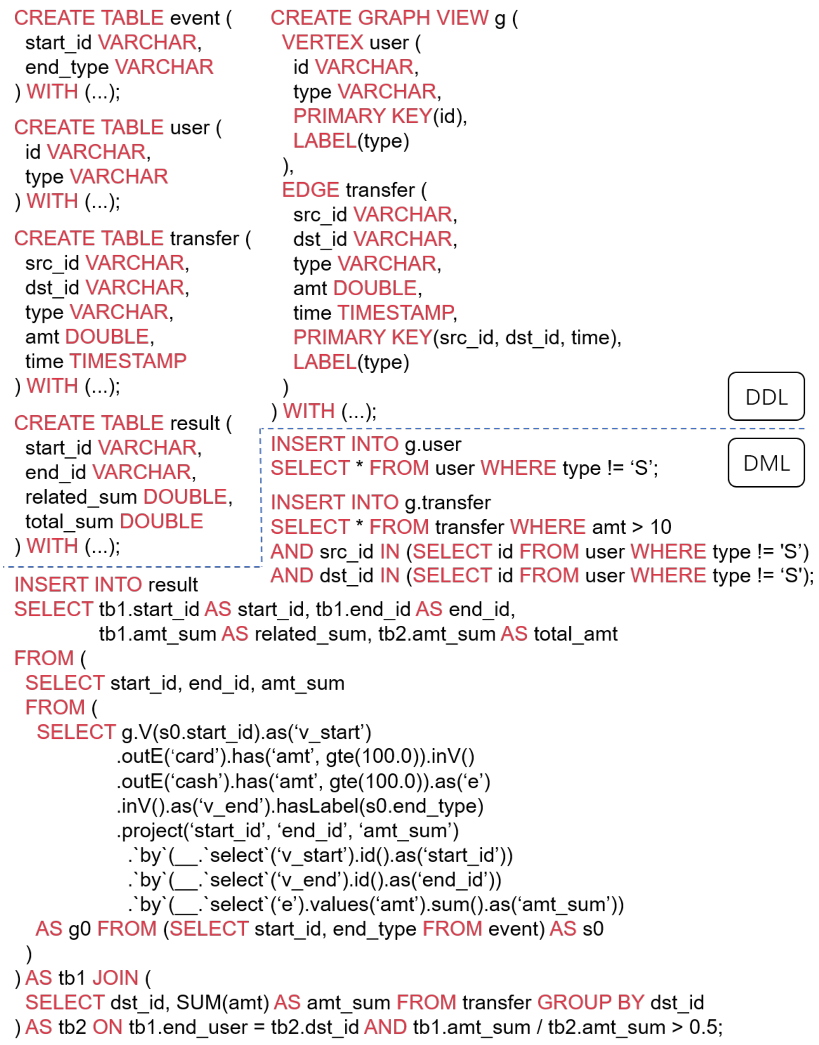
Batch processing. The decoupling of computation states from graph states and the BSP model adopted in graph-specific operators make it very easy to support batch processing in GeaFlow. We only need to specify the batch execution mode, connect the job to offline data sources, adjust batches to larger sizes, downgrade or even turn off fault tolerance measures, and finally execute the job with different scheduling policies (e.g. we can chain multiple operators with 1-to-1 connection patterns together to avoid unnecessary communication costs).

For batch processing, embedded memory and graph-native backends are preferred as they provide the best throughputs and space costs for static graphs. The multi-job sharing ability is also useful here, as a batch processing job can access a shared graph in read-only mode (e.g. Job-2 or Job-3 in Figure 10), which further reduces space consumption, especially for very large (i.e. distributed out-of-core) datasets.

4 HYBRID SQL-GREMLIN DSL

We develop a hybrid DSL that combines SQL and Gremlin, built on top of the API layer. The DSL largely reduces users' development costs as most complexities including coding, debugging, and optimization are now all handled by our query engine. Even in the cases of very complex graph queries, users can implement custom graph-specific operators and use them as UDFs (user-defined functions) in our DSL.

We choose the SQL + Gremlin combination as the DSL based on the following observations. Firstly, both SQL and Gremlin have wide adoptions in the domains of stream processing and graph processing respectively, and are thus easy to learn for new users. Secondly, while Gremlin can express navigational graph queries conveniently, the whole stream processing pipeline usually involves additional pre-/post-processing logic that is hard to express with Gremlin alone.



```
CREATE TABLE event (
  start_id VARCHAR,
  end_type VARCHAR
) WITH (...);

CREATE TABLE user (
  id VARCHAR,
  type VARCHAR
) WITH (...);

CREATE TABLE transfer (
  src_id VARCHAR,
  dst_id VARCHAR,
  type VARCHAR,
  amt DOUBLE,
  time TIMESTAMP
) WITH (...);

CREATE TABLE result (
  start_id VARCHAR,
  end_id VARCHAR,
  related_sum DOUBLE,
  total_sum DOUBLE
) WITH (...);

CREATE GRAPH VIEW g (
 VERTEX user (
  id VARCHAR,
  type VARCHAR,
  PRIMARY KEY(id),
  LABEL(type)
 ),
 EDGE transfer (
  src_id VARCHAR,
  dst_id VARCHAR,
  type VARCHAR,
  amt DOUBLE,
  time TIMESTAMP,
  PRIMARY KEY(src_id, dst_id, time),
  LABEL(type)
 )
) WITH (...);

INSERT INTO g.user
SELECT * FROM user WHERE type != ‘S’;

INSERT INTO g.transfer
SELECT * FROM transfer WHERE amt > 10
AND src_id IN (SELECT id FROM user WHERE type != 'S')
AND dst_id IN (SELECT id FROM user WHERE type != 'S');

INSERT INTO result
SELECT tb1.start_id AS start_id, tb1.end_id AS end_id,
        tb1.amt_sum AS related_sum, tb2.amt_sum AS total_amt
FROM (
 SELECT start_id, end_id, amt_sum
 FROM (
  SELECT g.V(s0.start_id).as(‘v_start’)
        .outE(‘card’).has(‘amt’, gte(100.0)).inV()
        .outE(‘cash’).has(‘amt’, gte(100.0)).as(‘e’)
        .inV().as(‘v_end’).hasLabel(s0.end_type)
        .project(‘start_id’, ‘end_id’, ‘amt_sum’)
         .`by`(__.`select`(‘v_start’).id().as(‘start_id’))
         .`by`(__.`select`(‘v_end’).id().as(‘end_id’))
         .`by`(__.`select`(‘e’).values(‘amt’).sum().as(‘amt_sum’))
  AS g0 FROM (SELECT start_id, end_type FROM event) AS s0
 )
) AS tb1 JOIN (
 SELECT dst_id, SUM(amt) AS amt_sum FROM transfer GROUP BY dst_id
) AS tb2 ON tb1.end_user = tb2.dst_id AND tb1.amt_sum / tb2.amt_sum > 0.5;
```

Fig. 11. A GeaFlow DSL example

Figure 11 illustrates the usage of our DSL through a simple example. We first define source and sink streams with `CREATE TABLE` statements. We use an extended `CREATE GRAPH VIEW`[10] statement to represent graph-structured datasets. Vertices and edges can then be ingested into the graph via the ETL pipelines defined with `INSERT ... SELECT` statements. Graph queries[11] are written in Gremlin, and can be embedded into SQL DML statements for further processing.

[10]Unlike materialized views in relational databases that are defined directly with the results of `SELECT` statements, the definition and manipulation of graph views are expressed with separate DDL and DML statements, so that vertices and edges can be ingested into the graph from multiple heterogeneous data sources with possibly different ETL pipelines.
[11]The example in Figure 11 looks for two-hop paths starting from a specified vertex and finishing at any vertex with a specified type (with edge types and properties meeting given conditions), and emits results aggregated by the end vertex of the path.

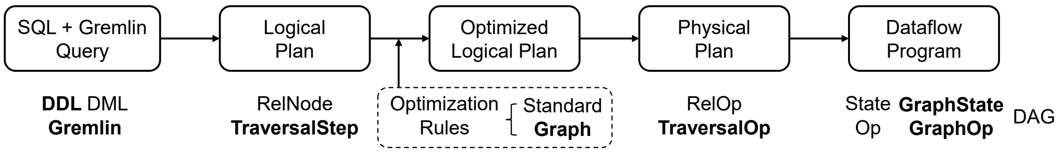


Fig. 12. Query planning in GeaFlow (bold texts indicating major enhancements)

It should be noted that the DSL still has some limitations. For example, iterative analytical tasks cannot be easily expressed in Gremlin due to inherent complexities, so users need to encapsulate the logic in a custom graph processing operator and register it as a UDF to be used in queries.

4.1 Query Planning

Our implementation of the DSL layer is mainly based on Calcite [27], extended with a set of graph-specific changes and optimizations to fulfill our requirements. Figure 12 presents the whole procedure of GeaFlow's query planning, translating a query written in the hybrid DSL to an optimized dataflow program in four phases.

Parsing. Our parser makes use of Calcite and TinkerPop [2] for parsing SQL and Gremlin statements respectively. We extend the SQL parser to support new DDL statements for definition of graph-structured datasets, as well as referencing the results of Gremlin queries as tables.

The output logical plan consists of both relational and graph operators, denoted as `RelNode` and `TraversalStep` respectively (with a special `GraphTrigger` operator bridging both parts). The former refers to an operation on table-structured datasets, while the latter corresponds to a Gremlin traversal step [17] on graph-structured datasets.

Logical optimization. The logical plan optimizer applies rules to the original plan to produce an optimized alternative. We complement the standard rules with a set of graph-specific ones mainly for optimizing `TraversalStep` portions. Thanks to the imperative nature of Gremlin, logical optimization is relatively easy, and there are currently three categories of optimization rules: (1) schema and projection reductions by removing unnecessary data fields, (2) predicate pushdowns by e.g. exploiting prefix range scan capabilities, and (3) traversal simplifications by e.g. merging combinable steps.

Physical Planning. After an optimized logical plan is determined, we transform the logical plan to a physical alternative, guided by some heursitics. Currently we have to rely on hints manually given in the query description rather than cost-based automatic selections that are still under development.

The first decision to be made is regarding partitioning schemes. We default to the 1D hash partitioning method as the space cost is lower (i.e. no vertex replicas), and switch to the 2D version when either skewness exceeds some threshold (i.e. there exist super vertices) or the query is "read-heavy" (i.e. there are much more reads than writes to graph states).

Next, we choose the key layout of graph states. The default (Figure 6) is opted for in most cases, unless when all predicates in the query can benefit from other layouts (such as the example previously introduced), or forced by hints.

For traversal operators, we choose the edge-centric variant by default, and switch to vertex-centric processing when computation is not commutative or associative.

Code generation. The final phase is code generation, which transforms a physical plan to a GeaFlow dataflow program. As the translation of relational operators has been well studied, here we mainly focus on the graph traversal parts.

Thanks to the inter-operator graph state sharing ability, we can map the whole traversal pipeline to a chain of operators rather than composing the logic in a single large operator which would not only harden bookkeeping of computation states but also make control flows complex. Nevertheless, we choose not to map operators in a 1-to-1 manner, but merge some operators to improve code and data locality as shown in previous work [65]. We break the whole pipeline into parts at dataflow edges that incur message passing (i.e. where processing moves from the current vertex to its neighbor), and merge operators within each part as a single graph traversal operator in GeaFlow.

There are some other optimizations that are performed in this phase. For example, as the schema is fixed in a DSL job, we can take advantage of the known value layouts to generate efficient code that accesses data fields of vertices/edges directly through off-heap memory.

4.2 Multi-Query Optimizations

In many cases, we want to perform a set of queries on the same streaming graph. While this can be enabled by the inter-job graph state sharing ability and is cost-effective in terms of space, more improvements in performance as well as reduction in compute resources can be achieved with our multi-query optimizations, especially when queries are "similar".

We optimize the concurrent execution of multiple queries[12] by merging them into a single large DAG according to a set of combining rules. Each rule has a `CanCombine` method indicating whether two `TraversalSteps` can be combined to one, and a `Combine` method returning a new combined `TraversalStep`. Besides steps that are exactly the same, other common combinable cases include different aggregations on the same set of property data, edge expansions from the same vertex with different edge labels, and so on.

The optimizer starts by trying to combine the operators of two queries from the beginning towards the end. If two operators are combinable, we put the new combined one as the common predecessor, and proceed to the next two. This process stops when the execution diverges at two imcompatible operators. Then another query is taken out as a candidate to be merged with the large DAG. Note that the merged DAG may now contain multiple operators in the same level (i.e. operators with the same BFS distance from the start operator), so we need to match the candidate's operator against each operator in the same level until divergence. The whole procedure is repeated for all remaining queries.

The performance improvement of our multi-query optimizations mainly comes from shared state accesses that are usually the bottlenecks. I/Os are merged, and de-serialization costs are amortized for multiple combined operations, which significantly improves the aggregate throughput and latency.

5 TYPICAL WORKLOADS

GeaFlow has been widely adopted in a variety of application scenarios at Ant Group surrounding the Alipay[13] ecosystem. Besides financial activities and social networks that can naturally be represented as graphs, we also use GeaFlow for many other applications. For instance, we can model tables, columns, jobs in our data warehouses and their relationships as vertices and edges. Deep queries (running up to tens of iterations) can be performed on such graphs to find potential

[12]Note that this differs from the case of traversal operators where different inputs share the same query.
[13]https://www.alipay.com/

security issues and critical bottlenecks. We classify all applications into four categories based on workload characteristics, listed and introduced as follows.

Streaming graph querying. The transfer network tracing example shown in Section 2 is a representative use case in this category. The graphs are continuously updated by ingesting vertex/edge objects extracted from the latest events. Queries trying to detect complex matching patterns are triggered upon event arrivals, with results written to external key-value stores for later use by other serving pipelines, or to message brokers for downstream analyses.

Users usually only need to write graph queries with the hybrid DSL, and deploy the compiled job on a cluster for continuous execution. Sometimes, UDFs are required to assist some special operations, such as the use of machine learning frameworks for GNN (Graph Neural Network) inference.

Windowed graph query simulation. Applications in this category can be seen as an offline version of the above one. Users propose a set of new queries and want to test their effects on historical data. Such needs can be fulfilled by selecting a window of logged events (e.g. historical transfers) as input, replaying them to trigger proposed queries, and writing out the results for further analyses to validate the effectiveness of each query. Those queries that turn out to work will then be deployed online.

As the input is bounded and known in advance, query simulation is essentially a batch processing workload, even though the performed computation pattern looks more similar to a streaming application. We thus exploit the multi-versioning feature of our graph-native backends, and decouple the execution of simulated queries (i.e. reads) from graph updates (i.e. writes) to increase the throughput.

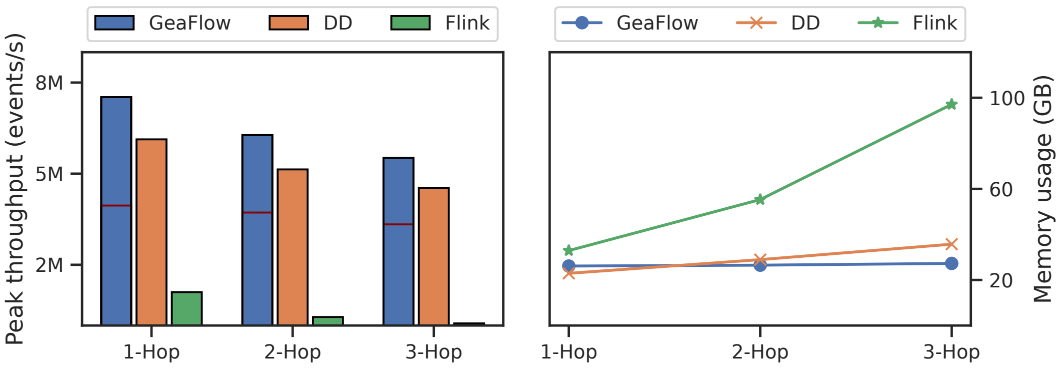
We first split the simulation window W into coarse-grained pieces e.g. days. Next, we construct the initial graph, containing vertices/edges extracted from the first w days (i.e. the state window) of events. We then process each of the following days of events by: (1) ingesting updates of this day in bulk, (2) executing queries included in this day, with each event using its event timestamp to access the corresponding version of the graph, and (3) garbage collect expired vertices/edges to bound space consumption.

Streaming graph analytics. There are also cases in which users need to perform iterative global analytics on a dynamic graph, e.g. running community detection algorithms continuously on the transfer graph to find potential suspects that are establishing connections to known fraudulent accounts. The main difference from querying workloads despite the analytical scope is that we can maintain some computation states in state backends, facilitating incremental computation.

While re-computation from scratch can also achieve the same results, and still benefit from our continuously maintained graph states, which reduces ETL costs for connecting upstream and downstream tasks, the computation can usually converge faster when a previous version of the results are available, e.g. the latest community labels. We thus can put these states inside our managed graph states, rather than drop them (as transient states) after each batch of computation.

Static graph analytics. We sometimes also use GeaFlow for offline graph processing. A static graph is constructed from input table-structured datasets, containing data in original forms. Then an iterative computation runs on the graph until convergence, with results dumped to output tables for further processing. We can also make use of the inter-job sharing feature of GeaFlow, and run a batch processing job on a historical/recent version of the graph.

Even though GeaFlow's graph processing performance on static datasets is apparently lower than state-of-the-art alternatives [36, 92] due to inevitable overheads such as the state backends that are designed for dynamic datasets, its general-purpose processing abilities and better affinity



(a) Sustainable throughput (b) Memory usage

Fig. 13. Streaming performance on LiveJournal

with the big data ecosystem makes it an attractive option for building end-to-end pipelines. As a result, the end-to-end slowdown is only about 2X while development and deployment are easier.

6 EVALUATION

We first compare GeaFlow with Flink and Differential Dataflow [61], two of the most popular open-source dataflow system, to illustrate the effectiveness of our graph-specific extensions. Then a set of experiments are conducted to validate some of the key design choices, i.e. the use of traversal operators and multi-query optimizations to increase throughputs by exploiting shared graph state accesses. Graph-native backends, 1D partitioning, and edge-centric operators are used by default unless otherwise specified.

GeaFlow vs. Flink vs. Differential Dataflow. The first set of experiments run on a cluster of 8 workers, each with 8 hardware threads and 16 GB memory. K-hop neighborhood queries are chosen as the workload for simplicity. The input stream is modeled on top of the *livejournal* [26] dataset (4.84M vertices and 69.0M edges), with each event inserting an edge and upserting two incident vertices. Each mutated vertex triggers a k-hop traversal. In each run, we first load 90% of the edges (randomly shuffled), and use the remaining for performance measurement.

The GeaFlow implementation composes the DAG with graph-specific traversal operators. While Flink also offers Gelly [15], its own graph processing library, it is built on top of the `DataSet` (rather than `DataStream`) abstraction and thus currently cannot be used for graph processing over streaming data. As a result, we still have to rely on join operators to build streaming graph traversal pipelines, manually optimized with our best efforts. Since Differential Dataflow currently supports only a volatile memory-based state backend, for fairness, we let both GeaFlow and Flink to use an optimized embedded memory backend built on top of hash maps.

Figure 13 presents the results, with sustainable throughputs and memory consumptions of each system. GeaFlow outperforms Flink significantly, with 6.83X on 1-hop queries, and up to 78.8X on 3-hop queries. While the performance of GeaFlow is only slightly higher (around 1.21X) than Differential Dataflow, its space usage stays nearly fixed while Differential Dataflow's memory consumption inevitably increases as traversals go deeper. Even though the memory usage of GeaFlow under 1-hop queries is higher (due to different memory management mechanisms of JVM vs. native programs), the situations are reversed for 2-hop and 3-hop queries.

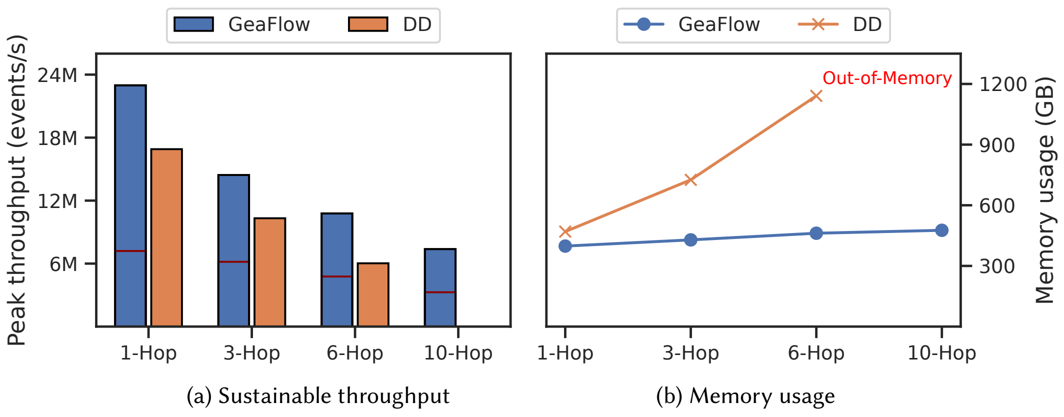


Fig. 14. Streaming performance on Twitter-2010

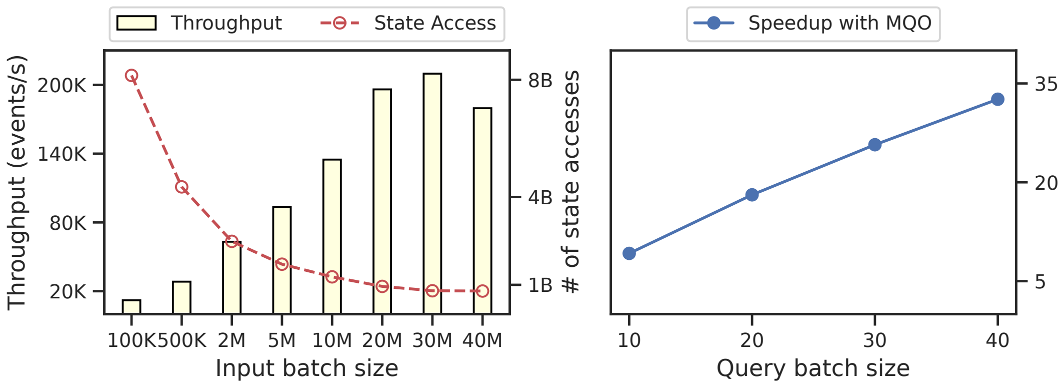
The space savings of GeaFlow over Differential Dataflow can be observed more evidently in Figure 14, where we run another set of experiments using a larger configuration, i.e. the *twitter-2010* [52] dataset (41.7M vertices and 1.47B edges) on a cluster of 10 workers, each with 24 hardware threads and 128 GB memory. While the performance improvements also slightly increase (up to 1.78X), the gaps on memory usage are widened more (up to 2.47X). Note that Differential Dataflow runs out of memory on queries of more than 6 hops. The reductions mainly come from the difference that GeaFlow decouples computation states from managed graph states, while Differential Dataflow needs to materialize intermediate joined results.

In production workloads, we rarely use the memory backend for state management since in most cases, the graph states cannot fit into the aggregate memory of workers. We thus also measure GeaFlow's performance for the above two sets of experiments using embedded graph-native backends, and plot the sustainble throughput in Figure 13a and Figure 14a with a red line lying in an according location within each GeaFlow's throughput bar. The slowdowns are about 1.65X to 1.90X for the smaller dataset, and 2.26X to 3.19X for the larger dataset. As the query complexities go higher, the performance gap between memory backends and graph-native backends comes closer.

Exploiting shared state accesses. Besides the space savings achieved with our co-designed graph-aware states and operators, exploiting the sharing opportunities in state accesses is another important contributing factor in our performance improvement.

We first run a set of experiments using the 3-hop query workload on the *twitter-2010* dataset, with various batch sizes for traversal operator execution. The cluster consists of 50 much smaller containers (each with 4 hardware threads, 4 GB memory, and a 25GB SSD) so that state accesses are stressed with I/O operations due to limited capacity available for graph object caching. The processing throughput and the total number of issued accesses to graph states in different runs are measured and shown in Figure 15a.

While using a batch size of 100K only achieves a throughput of 12.1K, increasing the batch size to 30M can reach 210K, providing a speedup of 17.3X. The improvement mainly comes from the reduction in accesses to graph states since there are naturally more sharing opportunities resulted from merged vertex program execution in larger batches, as can be seen in the monotonically decreasing trend in the total number of state accesses. Nevertheless, using a very large batch not only indicates higher latencies, but may also be harmful to the throughput (e.g. the 40M-batch



(a) Batched query processing (b) Multi-query optimization

Fig. 15. Effects of exploiting shared accesses to graph states

performance is lower than the 20M- and 30M-alternative), since the higher memory consumption of transient intermediate results can lead to more graph object cache misses and JVM GC overheads.

Multi-query optimization is another orthogonal way of exploiting shared state accesses, i.e. at the inter-query rather than the inter-input level of traversal operators. We use an internal graph query simulation workload to illustrate the benefits. The testbed has 38 24-thread 128-GB-memory containers, and the input comes from a selected window ingesting 935M vertices and 13.2B edges altogether, in which 176K events trigger 40 2-hop queries with similar patterns. The improvement brought by multi-query optimization under different numbers of concurrent simulated queries is given in Figure 15b. As compared to the average time required to simulate a single query, merging the execution of all 40 queries in the same batch only increases the total execution time by 22.7%, leading to a 32.6X speedup, providing substantial time savings.

7 RELATED WORK

We classify work related to GeaFlow into the following main categories: (general-purpose) stream processing systems, dynamic graph engines, and graph databases.

Stream processing systems. A number of general-purpose stream processing systems have been proposed and developed for some time, with some of them supporting batch mode execution as well [21, 24, 31, 32, 39, 41, 54, 62, 64, 66, 67, 83, 88]. GeaFlow includes the same set of functionalities for fault-tolerant scale-out stream processing, but is extended with graph-aware optimizations to facilitate processing on streaming graph-structured datasets. We believe these extensions are general enough and can be applied to other streaming systems as well. There are also some featured techniques that can further enhance GeaFlow, such as the use of coroutines to increase resource utilization [39].

Dynamic graph engines. There are a few systems designed specifically for processing on dynamic/evolving graphs [30, 34, 37, 42, 50, 51, 57, 60, 63, 69, 75, 77, 78, 80, 85, 87]. Most of them focus on iterative global analytical tasks, with some of them studying the use of incremental computation for acceleration. Nevertheless, as shown in our paper, a large category of our applications focus on querying, and graph processing capabilities alone is usually insufficient for building end-to-end streaming pipelines. This is one of the reasons why we choose to build GeaFlow on top of a general-purpose streaming system, and develop a hybrid DSL supporting both table and graph abstractions over streaming data.

Graph databases. Graph databases can be good candidates for graph querying workloads [3, 5–9, 11, 13, 14, 18, 53]. There also exist a large amount of work focusing specifically on graph query processing (e.g. [33, 68, 79, 84, 90] to name a few). At Ant Group, we also use GeaBase [38], our graph database solution, to handle a lot of such applications. We differentiate the usage scenarios between GeaBase and GeaFlow according to the following criteria: (1) GeaBase for synchronous and ad-hoc graph queries (with lower complexities) whose results need to be returned within tight latency bounds [93], and (2) GeaFlow (plus a key-value store) for asynchronous graph queries (with higher complexities and query patterns known in advance), and even more complex online analytical tasks that are hard to express with current graph query languages.

8 CONCLUSION

The motivation of GeaFlow originated from six years ago when we find existing streaming architectures ill-suited to graph processing workloads due to space explosion issues resulted from stateful join operators. The initial attempt was to wrap up graph processing tasks in a single logical operator whose actual processing are offloaded to external graph processing systems like GeaBase [38] for queries and GraphX [44] for analytical computations. However, the latencies were intolerable due to cross-system overheads, and even though the space cost in the streaming system was reduced, the total resource consumption was still high and the whole pipeline became excessively complex.

We then decided to build GeaFlow, the graph extended and accelerated dataflow system, with the base streaming framework inherited from an existing internal stream processing system. The major design choice was to implement the functionalities of graph processing directly in a graph-specific operator that is stateful, but manages only the dynamic graph-structured datasets in a partitioned manner like conventional keyed stateful operators. The preliminary prototype version included the memory and RocksDB-based state backends, and a Pregel-like vertex-centric programming interface provided to developers.

During the first few proof-of-concept tests, we quickly found that although such design reduced space costs significantly as expected thanks to decoupled computation states from managed graph states, the BSP execution model within the graph-specific operator would lead to low throughputs if queries (starting from different vertices) were to be processed one by one, or suffer from high programming complexities if users need to take care of batched query execution themselves. The idea of traversal operators was then proposed, to ease the development by masking the complexities.

After two years of incubation, GeaFlow successfully supported the anti-fraud task as one of the asynchronous data processing pipelines [93] during the 2018 Singles' Day Shopping Festival, attaining peak throughputs of more than 10 millions of events per second. The effects of those streaming queries and analytical computations were also significant, proving the benefits provided by streaming graph processing in financial scenarios.

Shortly after that, we decided to add the graph-specific DSL support to GeaFlow, so as to further ease the programming burdens of manually implementing graph queries with hand-written Java code and to widen the adoption of our system. While SQL is a natural first choice and has been adopted as the DSL in many streaming products [16, 24, 32], it is lacking in graph-related features. After some time of research, we chose Gremlin due to its good affinity with the Java ecosystem and the imperative nature that would simplify a lot of development efforts, and decided to embed it into SQL so that both expressiveness and flexibility would be better. We have been improving the query optimizer continually since then, but still there are many complex cases in which the performance of DSL-derived executions are lower than hand-optimized ones. Nevertheless, we found a large category of performance improvement opportunities that are very hard to exploit with manual coding, namely the multi-query optimizations that merge the DAGs of multiple "similar" queries into a combined larger one, which could significantly increase the aggregate throughput, and even the latency given the same restricted worker resources.

The improvements afterwards were mainly towards functional features, such as adding more state backends, the multi-versioning option for inter-job sharing capabilities, the supports for query simulation and batch processing, and so on. While the original purpose of developing a specific functionality might not be closely related with performance, we later found that many of them actually could lead to improvements in some certain scenarios, such as the simulation mode making use of the multi-versioning feature to increase throughputs. As a result, compared with four years ago, nowadays we only need less than 1/8 of the cluster configurations to meet the same performance requirements.

We plan to further evolve GeaFlow in several aspects. The first is the integration of techniques proposed in related domains of interest, such as operators with systematic incremental computation semantics. The second is regarding DSL. We are exploring the use of graph query languages with more declarative features such as OpenCypher [4], GQL [10], and SQL/PGQ [12] to replace the role of Gremlin or even have a re-design. A complete cost-based optimizer that can enhance our current DSL optimization processes, and tune a broader set of parameters automatically, is also on our roadmap. The third is exploration of a native execution engine written with system programming languages like Rust/C++, to improve the performance even further, as our long-term mission.

REFERENCES

[1] 2015. Apache Flink: Off-heap Memory in Apache Flink and the curious JIT compiler. https://flink.apache.org/news/2015/09/16/off-heap-memory.html. [Online; accessed 20-November-2022].
[2] 2022. Apache Tinkerpop. https://tinkerpop.apache.org/. [Online; accessed 20-November-2022].
[3] 2022. Azure Cosmos DB - NoSQL and Relational Database | Microsoft Azure. https://www.arangodb.com/. [Online; accessed 20-November-2022].
[4] 2022. Cypher Query Language Reference, Version 9. https://s3.amazonaws.com/artifacts.opencypher.org/openCypher9.pdf [Online; accessed 20-November-2022].
[5] 2022. Db2 Graph - IBM Documentation. https://www.ibm.com/docs/SSQNUZ_latest/svc-db2w/db2w-graph-ovu.html. [Online; accessed 20-November-2022].
[6] 2022. Enterprise Distributed Graph Database | DataStax. https://azure.microsoft.com/en-us/services/cosmos-db/. [Online; accessed 20-November-2022].
[7] 2022. Fully Managed Graph Database - Amazon Neptune - Amazon Web Services. https://aws.amazon.com/neptune/. [Online; accessed 20-November-2022].
[8] 2022. GDB. https://www.aliyun.com/product/gdb. [Online; accessed 20-November-2022].
[9] 2022. Graph Analytics Platform | Graph Database | TigerGraph. https://www.tigergraph.com/. [Online; accessed 20-November-2022].
[10] 2022. Graph Query Language GQL. https://www.gqlstandards.org/. [Online; accessed 20-November-2022].
[11] 2022. Home | OrientDB Community Edition. https://orientdb.org/. [Online; accessed 20-November-2022].
[12] 2022. ISO - ISO/IEC DIS 9075-16 - Information technology — Database languages SQL — Part 16: Property Graph Queries (SQL/PGQ). https://www.iso.org/standard/79473.html. [Online; accessed 20-November-2022].
[13] 2022. Memgraph - Open Source Graph Database. https://memgraph.com/. [Online; accessed 20-November-2022].
[14] 2022. Neo4j Graph Data Platform | Graph Database Management System. https://neo4j.com/. [Online; accessed 20-November-2022].
[15] 2022. Overview | Apache Flink. https://nightlies.apache.org/flink/flink-docs-master/docs/libs/gelly/overview/. [Online; accessed 20-November-2022].
[16] 2022. The Streaming Database | Materialize. https://materialize.com/. [Online; accessed 20-November-2022].
[17] 2022. TinkerPop Documentation. https://tinkerpop.apache.org/docs/current/reference/. [Online; accessed 20-November-2022].
[18] 2022. TuGraph. https://tech.antfin.com/products/TuGraph. [Online; accessed 20-November-2022].
[19] 2022. Tungsten - Databricks. https://databricks.com/glossary/tungsten. [Online; accessed 20-November-2022].
[20] 2023. Money mule - Wikipedia. https://en.wikipedia.org/wiki/Money_mule [Online; accessed 25-Feb-2023].

[21] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. Millwheel: Fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1033–1044.

[22] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1792–1803.

[23] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba Pey, Norbert Martínez, et al. 2020. The LDBC social network benchmark. *arXiv preprint arXiv:2001.02299* (2020).

[24] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data*. 601–613.

[25] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1185–1196.

[26] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. 44–54.

[27] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. 2018. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*. 221–230.

[28] Daniel K Blandford, Guy E Blelloch, and Ian A Kash. [n. d.]. An experimental analysis of a compact graph representation. ([n. d.]).

[29] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. {TAO}:{Facebook's} Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 49–60.

[30] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. 2012. Facilitating real-time graph mining. In *Proceedings of the fourth international workshop on Cloud data management*. 1–8.

[31] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1718–1729.

[32] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[33] Hongzhi Chen, Changji Li, Juncheng Fang, Chenghuan Huang, James Cheng, Jian Zhang, Yifan Hou, and Xiao Yan. 2019. Grasper: A high performance distributed system for OLAP on property graphs. In *Proceedings of the ACM Symposium on Cloud Computing*. 87–100.

[34] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems*. 85–98.

[35] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB.. In *CIDR*, Vol. 3. 3.

[36] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, et al. 2021. GraphScope: a unified engine for big graph processing. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2879–2892.

[37] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. RisGraph: A Real-Time Streaming System for Evolving Graphs to Support Sub-millisecond Per-update Analysis at Millions Ops/s. In *Proceedings of the 2021 International Conference on Management of Data*. 513–527.

[38] Zhisong Fu, Zhengwei Wu, Houyi Li, Yize Li, Min Wu, Xiaojie Chen, Xiaomeng Ye, Benquan Yu, and Xi Hu. 2019. Geabase: A high-performance distributed graph database for industry-scale applications. *International Journal of High Performance Computing and Networking* 15, 1-2 (2019), 12–21.

[39] Can Gencer, Marko Topolnik, Viliam Ďurina, Emin Demirci, Ensar B Kahveci, Ali Gürbüz, Ondřej Lukáš, József Bartók, Grzegorz Gierlach, František Hartman, et al. 2021. Hazelcast jet: low-latency stream processing at the 99.99 th percentile. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3110–3121.

[40] Lars George. 2011. *HBase: the definitive guide: random access to your planet-size data.* " O'Reilly Media, Inc.".

[41] Ana Sofia Gomes, João Oliveirinha, Pedro Cardoso, and Pedro Bizarro. 2021. Railgun: managing large streaming windows under MAD requirements. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3069–3082.

[42] Shufeng Gong, Chao Tian, Qiang Yin, Wenyuan Yu, Yanfeng Zhang, Liang Geng, Song Yu, Ge Yu, and Jingren Zhou. 2021. Automating incremental graph processing with flexible memoization. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1613–1625.

[43] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. {PowerGraph}: Distributed {Graph-Parallel} Computation on Natural Graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*. 17–30.

[44] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. {GraphX}: Graph Processing in a Distributed Dataflow Framework. In *11th USENIX symposium on operating systems design and implementation (OSDI 14)*. 599–613.

[45] Arpit Gupta, Rüdiger Birkner, Marco Canini, Nick Feamster, Chris Mac-Stoker, and Walter Willinger. 2016. Network monitoring as a streaming analytics problem. In *Proceedings of the 15th ACM workshop on hot topics in networks*. 106–112.

[46] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)* 46, 4 (2014), 1–34.

[47] Yanxiang Huang, Bin Cui, Wenyu Zhang, Jie Jiang, and Ying Xu. 2015. Tencentrec: Real-time stream recommendation in practice. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 227–238.

[48] Anand Iyer, Li Erran Li, and Ion Stoica. 2015. {CellIQ}:{Real-Time} Cellular Network Analytics at Scale. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 309–322.

[49] Albert Jonathan, Abhishek Chandra, and Jon Weissman. 2018. Multi-query optimization in wide-area streaming analytics. In *Proceedings of the ACM symposium on cloud computing*. 412–425.

[50] Wuyang Ju, Jianxin Li, Weiren Yu, and Richong Zhang. 2016. iGraph: an incremental data processing system for dynamic graph. *Frontiers of Computer Science* 10, 3 (2016), 462–476.

[51] Pradeep Kumar and H Howie Huang. 2020. Graphone: A data store for real-time analytics on evolving graphs. *ACM Transactions on Storage (TOS)* 15, 4 (2020), 1–40.

[52] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th international conference on World wide web*. 591–600.

[53] Changji Li, Hongzhi Chen, Shuai Zhang, Yingqian Hu, Chao Chen, Zhenjie Zhang, Meng Li, Xiangchen Li, Dongqing Han, Xiaohui Chen, et al. 2022. ByteGraph: a high-performance distributed graph database in ByteDance. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3306–3318.

[54] Wei Lin, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. 2016. {StreamScope}: Continuous Reliable Distributed Processing of Big Data Streams. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 439–453.

[55] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 1–28.

[56] Shengliang Lu, Shixuan Sun, Johns Paul, Yuchen Li, and Bingsheng He. 2021. Cache-Efficient Fork-Processing Patterns on Large Graphs. In *Proceedings of the 2021 International Conference on Management of Data*. 1208–1221.

[57] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. 2015. Llama: Efficient graph analytics using large multiversioned arrays. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 363–374.

[58] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[59] Renxin Mao, Zhao Li, and Jinhua Fu. 2015. Fraud Transaction Recognition: A Money Flow Network Approach. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management* (Melbourne, Australia) *(CIKM '15)*. Association for Computing Machinery, New York, NY, USA, 1871–1874. https://doi.org/10.1145/2806416.2806647

[60] Mugilan Mariappan and Keval Vora. 2019. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[61] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. [n. d.]. Shared Arrangements: practical inter-query sharing for streaming dataflows. *Proceedings of the VLDB Endowment* 13, 10 ([n. d.]).

[62] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow.. In *CIDR*.

[63] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. 2015. Immortalgraph: A system for storage and analysis of temporal graphs. *ACM Transactions on Storage (TOS)* 11, 3 (2015), 1–34.

[64] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455.

[65] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

[66] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. 2010. S4: Distributed stream computing platform. In *2010 IEEE International Conference on Data Mining Workshops*. IEEE, 170–177.

[67] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. 2017. Samza: stateful scalable stream processing at LinkedIn. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1634–1645.

[68] Zhengping Qian, Chenqiang Min, Longbin Lai, Yong Fang, Gaofeng Li, Youyang Yao, Bingqing Lyu, Xiaoli Zhou, Zhimin Chen, and Jingren Zhou. 2021. {GAIA}: A System for Interactive Analysis on Distributed Graphs Using a {High-Level} Language. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 321–335.

[69] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1876–1888.

[70] Marko A Rodriguez. 2015. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*. 1–10.

[71] Mendel Rosenblum and John K Ousterhout. 1991. The design and implementation of a log-structured file system. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*. 1–15.

[72] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 410–424.

[73] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 472–488.

[74] Prasan Roy, Srinivasan Seshadri, S Sudarshan, and Siddhesh Bhobe. 2000. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 249–260.

[75] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. 2016. Graphin: An online high performance incremental graph processing framework. In *European Conference on Parallel Processing*. Springer, 319–333.

[76] Marco Serafini, Gianmarco De Francisci Morales, and Georgos Siganos. 2017. Qfrag: Distributed graph search via subgraph isomorphism. In *proceedings of the 2017 symposium on cloud computing*. 214–228.

[77] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 505–516.

[78] Feng Sheng, Qiang Cao, Haoran Cai, Jie Yao, and Changsheng Xie. 2018. Grapu: Accelerate streaming graph analysis through preprocessing buffered updates. In *Proceedings of the ACM Symposium on Cloud Computing*. 301–312.

[79] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. 2016. Fast and Concurrent {RDF} Queries with {RDMA-Based} Distributed Graph Exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 317–332.

[80] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. 2016. Tornado: A system for real-time iterative analysis over evolving data. In *Proceedings of the 2016 International Conference on Management of Data*. 417–430.

[81] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[82] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. *Proc. VLDB Endow.* 8, 11 (jul 2015), 1214–1225. https://doi.org/10.14778/2809974.2809983

[83] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 147–156.

[84] Vasileios Trigonakis, Jean-Pierre Lozi, Tomáš Faltín, Nicholas P Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Călin Iorgulescu, Petr Koupy, Jinsoo Lee, et al. 2021. {aDFS}: An Almost {Depth-First-Search} Distributed {Graph-Querying} System. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 209–224.

[85] Pourya Vaziri and Keval Vora. 2021. Controlling memory footprint of stateful streaming graph processing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 269–283.

[86] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 374–389.

[87] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the twenty-second international conference on architectural support for*

programming languages and operating systems. 237–251.

[88] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. 423–438.

[89] Feng Zhang, Chenyang Zhang, Lin Yang, Shuhao Zhang, Bingsheng He, Wei Lu, and Xiaoyong Du. 2021. Fine-grained multi-query stream processing on integrated architectures. *IEEE Transactions on Parallel and Distributed Systems* 32, 9 (2021), 2303–2320.

[90] Yunhao Zhang, Rong Chen, and Haibo Chen. 2017. Sub-millisecond stateful stream querying over fast-evolving linked data. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 614–630.

[91] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Ligang He, Bingsheng He, and Haikun Liu. 2018. {CGraph}: A Correlations-aware Approach for Efficient Concurrent Iterative Graph Processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 441–452.

[92] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A {Computation-Centric} Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 301–316.

[93] Xiaowei Zhu, Zhisong Fu, Zhenxuan Pan, Jin Jiang, Chuntao Hong, Yongchao Liu, Yang Fang, Wenguang Chen, and Changhua He. 2021. Taking the Pulse of Financial Activities with Online Graph Processing. *ACM SIGOPS Operating Systems Review* 55, 1 (2021), 84–87.

Received November 2022; revised February 2023; accepted March 2023