# Weaver: A High-Performance, Transactional Graph Database Based on Refinable Timestamps

Ayush Dubey
Cornell University

Greg D. Hill
Stanford University

Robert Escriva
Cornell University

Emin Gün Sirer
Cornell University

## ABSTRACT

Graph databases have become a common infrastructure component. Yet existing systems either operate on offline snapshots, provide weak consistency guarantees, or use expensive concurrency control techniques that limit performance.

In this paper, we introduce a new distributed graph database, called Weaver, which enables efficient, transactional graph analyses as well as strictly serializable ACID transactions on dynamic graphs. The key insight that allows Weaver to combine strict serializability with horizontal scalability and high performance is a novel request ordering mechanism called refinable timestamps. This technique couples coarse-grained vector timestamps with a fine-grained timeline oracle to pay the overhead of strong consistency only when needed. Experiments show that Weaver enables a Bitcoin blockchain explorer that is 8× faster than Blockchain.info, and achieves 10.9× higher throughput than the Titan graph database on social network workloads and 4× lower latency than GraphLab on offline graph traversal workloads.

## 1. INTRODUCTION

Graph-structured data arises naturally in a wide range of fields that span science, engineering, and business. Social networks, the world wide web, biological interaction networks, knowledge graphs, cryptocurrency transactions, and many classes of business analytics are naturally modeled as a set of vertices and edges that comprise a graph. Consequently, there is a growing need for systems which can store and process large-scale graph-structured data.

Correctness and consistency in the presence of changing data is a key challenge for graph databases. For example, imagine a graph database used to implement a network controller that stores the network topology shown in Fig. 1. When the network is undergoing churn, it is possible for a path discovery query to return a path through the network that did not exist at any instant in time. For instance, if the link $(n_3, n_5)$ fails, and subsequently the link $(n_5, n_7)$ goes online, a path query starting from host $n_1$ to host $n_7$ may erroneously conclude that $n_7$ is reachable from $n_1$, even though no such path ever existed.

Providing strongly consistent queries is particularly challenging for graph databases because of the unique characteristics of typical graph queries. Queries such as traversals often read a large portion of the graph, and consequently take a long time to execute. For instance, the average degree of separation in the Facebook social network is 3.5 [8], which implies that a breadth-first traversal that starts at a random vertex and traverses 4 hops will likely read all 1.59 billion users. On the other hand, typical key-value and relational queries are much smaller; the `NewOrder` transaction in the TPC-C benchmark [7], which comprises 45% of the frequency distribution, consists of 26 reads and writes on average [21]. Techniques such as optimistic concurrency control or distributed two-phase locking result in poor throughput when concurrent queries try to read large subsets of the graph.

Due to the unique nature of typical graph-structured data and queries, existing databases have offered limited support. State-of-the-art transactional graph databases such as Neo4j [6] and Titan [9] employ heavyweight coordination techniques for transactions. Weakly consistent online graph databases [11, 15] forgo strong semantics for performance, which limits their scope to applications with loose consistency needs and requires complicated client logic. Offline graph processing systems [26, 3, 41, 53] do not permit updates to the graph while processing queries. Lightweight techniques for modifying and querying a distributed graph with strong consistency guarantees have proved elusive thus far.

Weaver[1] is a new online, distributed, and transactional graph database that supports efficient graph analyses. The key insight that enables Weaver to scalably execute graph transactions in a strictly serializable order is a novel technique called *refinable timestamps*. This technique uses a highly scalable and lightweight timestamping mechanism for ordering the majority of operations and relies on a fine-grained timeline oracle for ordering the remaining, potentially-conflicting reads and writes. This unique two-step ordering technique with proactive timestamping and a reactive timeline oracle has three advantages.

First, refinable timestamps enable Weaver to distribute the graph across multiple shards and still execute transactions in a scalable fashion. There are some applications and workloads for which sharding is unnecessary [43]. However many applications support a large number of concurrent clients and

[1] http://weaver.systems, https://github.com/dubey/weaver


*Proceedings of the VLDB Endowment,* Vol. 9, No. 11

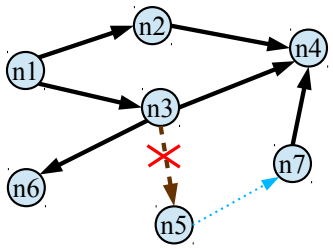



**Figure 1:** A graph undergoing an update which creates $(n_5, n_7)$ and deletes $(n_3, n_5)$ concurrently with a traversal starting at $n_1$. In absence of transactions, the query can return path $(n_1, n_3, n_5, n_7)$ which never existed.

operate on graphs of such large scale, consisting of billions of vertices and edges [61, 34, 46], that a single-machine architecture is infeasible. For such high-value applications [11, 54] it is critical to distribute the graph data in order to balance the workload and to enable highly-parallel in-memory query processing by minimizing disk accesses.

Second, refinable timestamps reduce the amount of coordination required for execution of graph analysis queries. Concurrent transactions that do not overlap in their data sets can execute independently without blocking each other. Refinable timestamps order only those transactions that overlap in their read-write sets, using a combination of vector clock ordering and the timeline oracle.

Third, refinable timestamps enable Weaver to store a multi-version graph by marking vertices and edges with the timestamps of the write operations. A multi-version graph lets long-running graph analysis queries operate on a consistent version of the graph without blocking concurrent writes. It also allows historical queries which run on past, consistent versions of the graph.

Overall, this paper makes the following contributions:

- It describes the design of an online, distributed, fault-tolerant, and strongly consistent graph database that achieves high performance and enables ACID transactions and consistent graph queries.
- It details a novel, lightweight ordering mechanism called refinable timestamps that enables the system to trade off proactive synchronization overheads with reactive discovery of ordering information.
- It shows that these techniques are practical through a full implementation and an evaluation that shows that Weaver scales well to handle graphs with over a billion edges and significantly outperforms state-of-the-art systems such as Titan [9] and GraphLab [26] and applications such as Blockchain.info [1].

# 2. APPROACH

Weaver combines the strong semantics of ACID transactions with high-performance, transactional graph analyses. In this section, we describe the data and query model of the system as well as sample applications.

## 2.1 Data Model

Weaver provides the abstraction of a property graph, i.e. a directed graph consisting of a set of vertices with directed edges between them. Vertices and edges may be labeled with named properties defined by the application. For example, an edge $(u, v)$ may have both "weight=3.0" and "color=red" properties, while another edge $(v, w)$ may have just the "color=blue" property. This enables applications to attach data to vertices and edges.

```
begin_weaver_tx()
photo    = create_node()
own_edge = create_edge(user, photo)
assign_property(own_edge, "OWNS")
for nbr in permitted_neighbors:
  access_edge = create_edge(photo, nbr)
  assign_property(access_edge, "VISIBLE")
commit_weaver_tx()
```

**Figure 2:** A Weaver transaction which posts a photo in a social network and makes it visible to a subset of the user's friends.

## 2.2 Transactions for Graph Updates

Weaver provides transactions over the directed graph abstraction. These transactions comprise reads and writes on vertices and edges, as well as their associated attributes. The operations are encapsulated in a `weaver_tx` block and may use methods such as `get_vertex` and `get_edge` to read the graph, `create/delete_vertex` and `create/delete_edge` to modify the graph structure, and `assign/delete_properties` to assign or remove attribute data on vertices and edges. Fig. 2 shows the code for an update to a social network that posts content and manages the access control for that content in the same atomic transaction.

## 2.3 Node Programs for Graph Analyses

Weaver also provides specialized, efficient support for a class of read-only graph queries called node programs. Similar to stored procedures in databases [24], node programs traverse the graph in an application-specific fashion, reading the vertices, edges, and associated attributes via the `node` argument. For example, Fig. 3 describes a node program that executes BFS using only edges annotated with a specified `edge_property`. Such queries operate atomically and in isolation on a logically consistent snapshot of the graph. Weaver queries wishing to modify the graph must collate the changes they wish to make in a node program and submit them as a transaction.

Weaver's node programs employ a mechanism similar to the commonly used scatter-gather approach [41, 3, 26] to propagate queries to other vertices. In this approach, each vertex-level computation is passed query parameters (`prog_params` in Fig. 3) from the previous hop vertex, similar to the gather phase. Once a node program completes execution on a given vertex, it returns a list of vertex handles to traverse next, analogous to the scatter phase. A node program may visit a vertex any number of times; Weaver enables applications to direct all aspects of node program propagation. This approach is sufficiently expressive to capture common graph analyses such as graph exploration [1], search algorithms [5], and path discovery [54].

Many node programs are stateful. For instance, a traversal query may store a bit per vertex visited, while a shortest path query may require state to save the distance from the source vertex. This per-query state is represented in Weaver's node programs by `node.prog_state`. Each active node program has its own state object that persists within the `node` object until the node program runs to completion throughout the graph. As a node program traverses the graph, the application can create `prog_state` at other vertices and propagate it between vertices using the `prog_params`. This design enables applications that implement a wide array of graph

```
node_program(node, prog_params):
 nxt_hop = []
 if not node.prog_state.visited:
  for edge in node.neighbors:
   if edge.check(prog_params.edge_prop):
    nxt_hop.append((edge.nbr,prog_params))
  node.prog_state.visited = true
 return nxt_hop
```

**Figure 3:** A node program in Weaver which executes a BFS query on the graph.

algorithms. Node program state is garbage collected after the query terminates on all servers (§ 4.5).

Since node programs are typically long-running, it is a challenge to ensure that these queries operate on a consistent snapshot of the graph. Weaver addresses this problem by storing a multi-version graph with associated timestamps. This enables transactional graph updates to proceed without blocking on node program reads.

# 3. REFINABLE TIMESTAMPS

The key challenge in any transactional system is to ensure that distributed operations taking place on different machines follow a coherent timeline. Weaver addresses this challenge with refinable timestamps, a lightweight mechanism for achieving a rough order when sufficient and fine-grained order when necessary.

## 3.1 Overview

At a high level, refinable timestamps factor the task of achieving a strictly serializable order of transaction execution into two stages. The first stage, which assigns a timestamp to each transaction, is cheap but imprecise. Any server in the system that receives the transaction from a client can assign the timestamp, without coordinating with other servers. There is no distributed coordination, resulting in high scalability. However, timestamps assigned in this manner are imprecise and do not give a total order between transactions.

The second stage resolves conflicts that may arise during execution of transactions with imprecise timestamps. This stage is more expensive and less scalable but leads to a precise ordering of transactions. The system resorts to the second stage only for a small subset of transactions, i.e. those that are concurrent and overlap in their read-write sets.

The key benefit of using refinable timestamps, compared to traditional distributed locking techniques, is reduced coordination. The proactive stage is lightweight and scalable, and imposes very little overhead on transaction processing. The system pays the cost of establishing a total order only when conflicts arise between timestamped operations. Thus, refinable timestamps avoid coordinating transactions that do not conflict.

This benefit is even more critical for a graph database because of the characteristics of graph analysis queries: long execution time and large read set. For example, a breadth-first search traversal can explore an expansive connected component starting from a single vertex. Refinable timestamps execute such large-scale reads without blocking concurrent, conflicting transactions.

## 3.2 System Architecture

Weaver implements refinable timestamps using a timeline coordinator, a set of shard servers and a backing store. Fig. 4 depicts the Weaver system architecture.

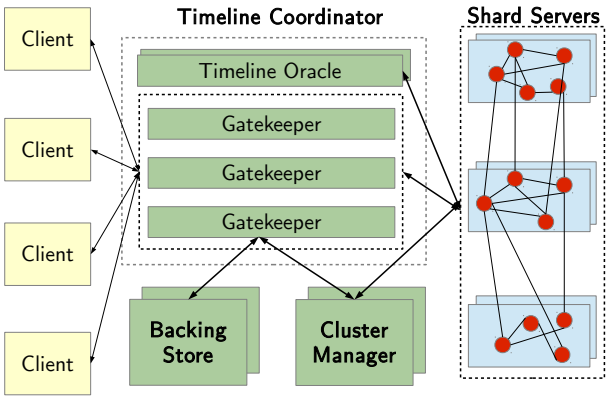



**Figure 4:** Weaver system architecture.

**Shard Servers**: Weaver distributes the graph by partitioning it into smaller pieces, each of which is stored in memory on a shard server. This sharding enables both memory storage capacity and query throughput to scale as servers are added to the system. Each graph partition consists of a set of vertices, all outgoing edges rooted at those vertices, and associate attributes. The shard servers are responsible for executing both node programs and transactions on the in-memory graph data.

**Backing Store**: The backing store is a key-value store that supports ACID transactions and serves two purposes. First, it stores the graph data in a durable and fault-tolerant manner. When a shard server fails, the graph data that belongs to the shard is recovered from the backing store. Second, the backing store directs transactions on a vertex to the shard server responsible for that vertex by storing a mapping from vertices to associated shard servers. Our implementation uses HyperDex Warp [21] as the backing store.

**Timeline Coordinator**: The critical component behind Weaver's strict serializability guarantees is the timeline coordinator. This coordinator consists of a user-configured number of *gatekeeper* servers for coarse timestamp-based ordering and a *timeline oracle* for refining these timestamps when necessary (§ 3.3, § 3.4). In addition to assigning timestamps to transactions, the gatekeepers also commit transactional updates to the backing store (§ 4).

**Cluster Manager**: Weaver also deploys a cluster manager process for failure detection and system reconfiguration. The cluster manager keeps track of all shard servers and gatekeepers that are currently part of the Weaver deployment. When a new gatekeeper or shard server boots up, it registers its presence with the cluster manager and then regularly sends heartbeat messages. If the cluster manager detects that a server has failed, it reconfigures the cluster according to Weaver's fault tolerance scheme (§ 4.3).

## 3.3 Proactive Ordering by Gatekeepers

The core function of gatekeepers is to assign to every transaction a timestamp that can scalably achieve a partial order. To accomplish this, Weaver directs each transaction through any one server in a bank of gatekeepers, each of which maintains a vector clock [23]. A vector clock consists of an array of counter values, one per gatekeeper, where each gatekeeper maintains a local counter as well as the maximum counter value it has seen from the other gatekeepers. Gatekeepers increment their local clock on receipt of a client request, attach the vector clock to every such transaction, and forward

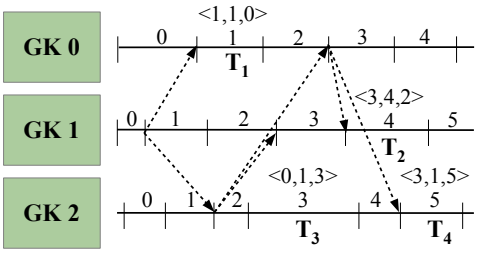



**Figure 5:** Refinable timestamps using three gatekeepers. Each gatekeeper increments its own counter for a transaction and periodically announces its counter to other gatekeepers (shown by dashed arrows). Vector timestamps are assigned locally based on announcements that a gatekeeper has collected from peers. $T_1\langle 1,1,0\rangle \prec T_2\langle 3,4,2\rangle$ and $T_3\langle 0,1,3\rangle \prec T_4\langle 3,1,5\rangle$. $T_2$ and $T_4$ are concurrent and require fine-grain ordering only if they conflict. There is no need for lockstep synchrony between gatekeepers.

it to the shards involved in the transaction.

Gatekeepers ensure that the majority of transaction timestamps are directly comparable by exchanging vector clocks with each other every $\tau$ milliseconds. This proactive communication between gatekeepers establishes a *happens-before* partial order between refinable timestamps. Fig. 5 shows how these vector clocks can order transactions with the help of these happens-before relationships. In this example, since $T_1$ and $T_2$ are separated by an announce message from gatekeeper 0, their vector timestamps are sufficient to determine that $T_1\langle 1,1,0\rangle \prec T_2\langle 3,4,2\rangle$ (X $\prec$ Y denotes X happens before Y, while X $\preceq$ Y denotes either X $\prec$ Y or X = Y).

Unfortunately, vector clocks are not sufficient to establish a total order. For instance, in Fig. 5, transactions $T_2$ (with timestamp $\langle 3,4,2\rangle$) and $T_4$ (with timestamp $\langle 3,1,5\rangle$) cannot be ordered with respect to each other and need a more refined ordering if they overlap in their read-write sets (we denote this as $T_2 \approx T_4$). Since transactions that enter the system simultaneously through multiple gatekeepers may receive concurrent vector clocks, Weaver uses an auxiliary service called a timeline oracle to put them into a serializable timeline.

## 3.4 Reactive Ordering by Timeline Oracle

A timeline oracle is an event ordering service that keeps track of the happens-before relationships between transactions [20]. The timeline oracle maintains a dependency graph between outstanding transactions, completely independent of the graph stored in Weaver. Each vertex in the dependency graph represents an ongoing transaction, identified by its vector timestamp, and every directed edge represents a happens-before relationship. The timeline oracle ensures that transactions can be reconciled with a coherent timeline by guaranteeing that the graph remains acyclic.

The timeline oracle carries out this task with a simple API centered around events, where an event corresponds to a transaction in Weaver. Specifically, it provides primitives to create a new event, to atomically assign a happens-before relationship between sets of events, and to query the order between two or more events.

Weaver's implementation of the timeline oracle comprises such an event-oriented API backed by an event dependency graph that keeps track of transactions at a fine grain [20]. The service is essentially a state machine that is chain replicated [62] for fault tolerance. Updates to the event dependency graph, caused by new events or new dependencies, occur at the head of the chain, while queries can execute on any copy of the graph. This results in a high-performance implementation that scales up to ∼6M queries per second on a 12 8-core server chain.

Weaver uses this high-performance timeline oracle to establish an order between concurrent transactions which may overlap in their read or write sets. Strictly speaking, such transactions must have at least one vertex or edge in common. Since discovering fine-grained overlaps between transaction operations can be costly, our implementation conservatively orders any pair of concurrent transactions that have a shard server in common. When two such transactions are committing simultaneously, the server(s) committing the transactions send an ordering request to the timeline oracle. The oracle either returns an order if it already exists, or establishes an order between the transactions. To maintain a directed acyclic graph corresponding to the happens-before relationships, it ensures that all subsequent operations follow this order.

Establishing a fine-grained order on demand has the significant advantage that Weaver will not order transactions that cannot affect each other, thereby avoiding the overhead of the centralized oracle for these transactions (§ 4.1, § 4.2). Such transactions will commit without coordination. Their operations may interleave, i.e. appear non-atomic to an omniscient observer, but this interleaving is benign because, by definition, no clients can observe this interleaving. The only transactions that need to be ordered are those whose interleaving may lead to an observable non-atomic or non-serializable outcome.

## 3.5 Discussion

Weaver's implementation of refinable timestamps combines vector clocks with a timeline oracle. Alternatively, the gatekeepers in Weaver could assign a loosely synchronized real timestamp to each transaction, similar to TrueTime [17]. Both techniques ensure a partial order. However TrueTime makes assumptions about network synchronicity and communication delay, which are not always practical, even within the confines of a datacenter. Synchronicity assumptions interfere with debugging, and maybe violated by network delays under heavy load and systems running in virtualized environments. Further, a TrueTime system synchronized with average error bound $\bar{\varepsilon}$ will necessarily incur a mean latency of $2\bar{\varepsilon}$. While TrueTime makes sense for the wide area environment for which it was developed, Weaver uses vector clocks for its first stage.

Irrespective of implementation, refinable timestamps represent a hybrid approach to timeline ordering that offers an interesting tradeoff between proactive costs due to periodic synchronization messages between gatekeepers, and the reactive costs incurred at the timeline oracle. At one extreme, one could use the timeline oracle for maintaining the global timeline for *all* requests, but then the throughput of the system would be bottlenecked by the throughput of the oracle. At the other extreme, one could use only gatekeepers and synchronize at such high frequency so as to provide no opportunity for concurrent timestamps to arise. But this approach would also incur too high an overhead, especially under high workloads. Weaver's key contribution is to reduce the load on a totally ordering timeline oracle by layering on a timestamping service that manages the bulk of the ordering, and leaves only a small number of overlapping transactions to be ordered by the oracle. This tradeoff ensures that the scalability limits of a centralized timeline service [20] are

extended by adding gatekeeper servers. Weaver's design provides a parameter $\tau$—the clock synchronization period—that manages this tradeoff.

The clock synchronization period can be adjusted dynamically based on the system workload. Initially, when the system is quiescent, the gatekeepers do not need to synchronize their clocks. As the rate of transactions processed by the different gatekeepers increases, the gatekeepers synchronize clocks more frequently to reduce the burden on the timeline oracle. Beyond a point, the overhead of synchronization itself reduces the throughput of the timestamping process. We empirically analyze how the system can discover the sweet spot for $\tau$ in § 6.

# 4. IMPLEMENTATION AND CORRECTNESS

Weaver uses refinable timestamps for ordering transactions. However, because node programs potentially have a very large read set and long execution time, Weaver processes node programs differently from read-write transactions.

## 4.1 Node Programs

Weaver includes a specialized, high-throughput implementation of refinable timestamps for node program execution. A gatekeeper assigns a timestamp $T_{prog}$ and forwards the node program to the appropriate shards. The shards execute the node program on a version of the in-memory graph consistent with $T_{prog}$ by comparing $T_{prog}$ to the timestamps of the vertices and edges in the multi-version graph and only reading the portions of the graph that exist at $T_{prog}$. In case timestamps are concurrent, the shard requests for an order from the timeline oracle.

When the timeline oracle receives an ordering request for a node program and a committed write from a shard, it returns the pre-established order between these transactions to the shard, if one exists. In cases where a pre-established order does not exist, because gatekeepers do not precisely order transactions, the oracle will prefer arrival order. This order is then established as a commitment for all time; the timeline oracle will record the happens-before relationship and ensure that all subsequent queries from all shard servers receive responses that respect this commitment.

Because arrival order may differ on different shard servers, care must be taken to ensure atomicity and isolation. For example, in a naïve implementation, a node program $P$ may arrive after a transaction $T$ on shard 2, but before $T$ on shard 1. To ensure consistent ordering, Weaver delays execution of a node program at a shard until after execution of all preceding and concurrent transactions.

In addition to providing consistent ordering for transactions, the timeline oracle ensures that transitive ordering is maintained. For instance, if $T_1 \prec T_2$ and $T_2 \prec T_3$ is pre-established, then an order query between $T_1$ and $T_3$ will return $T_1 \prec T_3$. Furthermore, because transactions are identified by their unique vector clocks, the timeline oracle can infer and maintain implicit dependencies captured by the vector clocks. For example, if the oracle first orders $\langle 0,1\rangle \prec \langle 1,0\rangle$ and subsequently a shard requests the order between $\langle 0,1\rangle$ and $\langle 2,0\rangle$, the oracle will return $\langle 0,1\rangle \prec \langle 2,0\rangle$ because $\langle 0,1\rangle \prec \langle 1,0\rangle \prec \langle 2,0\rangle$ due to transitivity.

## 4.2 Transactions

Transactions, which contain both reads and writes, result in updates to both the in-memory graph at the shard servers and the fault-tolerant graph stored in the backing store. Weaver first executes the transaction on the backing store, thereby leveraging its transactional guarantees to check transaction validity. For example, if a transaction attempts to delete an already deleted vertex, it aborts while executing on the backing store. After the transaction commits successfully on the backing store, it is forwarded to the shard servers which update the in-memory graph without coordination.

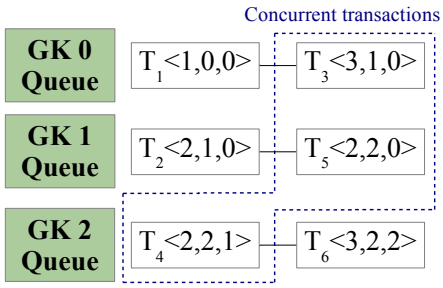



**Figure 6:** Each shard server maintains a queue of transactions per gatekeeper and executes the transaction with the lowest timestamp. When a group of transactions are concurrent (e.g. $T_3, T_4$, and $T_5$), the shard server consults the timeline oracle to order them.

To execute a transaction on the backing store, gatekeepers act as intermediaries. Clients buffer writes and submit them as a batch to the gatekeeper at the end of a transaction, and the gatekeeper, in turn, performs the writes on the backing store. The backing store commits the transaction if none of the data read during the transaction was modified by a concurrent transaction. HyperDex Warp, the backing store used in Weaver, employs the highly scalable acyclic transactions protocol [21] to order multi-key transactions. This protocol a form optimistic concurrency control that enables scalable execution of large volumes of transactions from gatekeepers.

Gatekeepers, in addition to executing transactions on the backing store, also assign a refinable timestamp to each transaction. Timestamps are assigned in a manner that respects the order of transaction execution on the backing store. For example, if there are two concurrent transactions $T_1$ and $T_2$ at gatekeepers $GK_1$ and $GK_2$ respectively, both of which modify the same vertex in the graph, Weaver guarantees that if $T_1$ commits before $T_2$ on the backing store, then $T_1 \prec T_2$. To this end, Weaver stores the timestamp of the last update for each vertex in the backing store. In our example, if $T_1$ commits before $T_2$ on the backing store, then the last update timestamp at the graph vertex will be $T_1$ when $GK_2$ attempts to commit $T_2$. Before committing $T_2$, $GK_2$ will check that $T_1 \prec T_2$. If it so happens that the timestamp assigned by $GK_2$ is smaller, i.e. $T_2 \prec T_1$, then $GK_2$ will abort and the client will retry the transaction. Upon retrying, $GK_2$ will assign a higher timestamp to the transaction.

While gatekeepers assign refinable timestamps to transactions and thereby establish order, shard servers obey this order. To do so, each shard server has a priority queue of incoming transactions for each gatekeeper, prioritized by their timestamps (Fig. 6). Shard servers enqueue transactions from gatekeeper $i$ on its $i$-th *gatekeeper queue*. When each gatekeeper queue is non-empty, an event loop at the shard server pulls the first transaction $T_i$ off each queue $i$ and executes the earliest transaction out of $(T_1, T_2, \ldots, T_n)$. In case a set of transactions appear concurrent, such as $(T_3, T_4, T_5)$ in Fig. 6, the shard servers will submit the set to the timeline oracle in order to discover and, if necessary, assign an order.

Weaver's implementation of refinable timestamps at shard servers has correctness and performance subtleties. First, in order to ensure that transactions are not lost or reordered in transit, Weaver maintains FIFO channels between each gatekeeper and shard pair using sequence numbers. Second, to ensure the system makes progress in periods of light workload, gatekeepers periodically send NOP transactions to shards. NOP transactions guarantee that there is always a transaction at the head of each gatekeeper queue. This provides an upper-bound on the delay in node program execution, set by default to $10\mu s$ in our current implementation. Finally, since ordering decisions made by the timeline oracle are irreversible and monotonic, shard servers can cache these decisions in order to reduce the number of ordering requests.

Shard servers also maintain the in-memory, multi-version distributed graph by marking each written object with the refinable timestamp of the transaction. For example, an operation that deletes an edge actually marks the edge as deleted and stores the refinable timestamp of the deletion in the edge object.

## 4.3 Fault Tolerance

Weaver minimizes data that is persistently stored by only storing the the graph data in the backing store. In response to a gatekeeper or shard failure, the cluster manages spawns a new process on one of the live machines. The new process restores corresponding graph data from the backing store. However, restoring the graph from the backing store is not sufficient to ensure strict serializability of transactions since timestamps and queues are not stored at the backing store.

Weaver implements additional techniques to ensure strict serializability. A transaction that has committed on the backing store before failure requires no extra handling: the backup server will read the latest copy of the data from the backing store. Transactions that have not executed on the backing store, as well as all node programs, are reexecuted by Weaver with a fresh timestamp after recovery, when resubmitted by clients. Since partially executed operations for these transactions were not persistent, it is safe to start execution from scratch. This simple strategy avoids the overhead of execution-time replication of ordering metadata such as the gatekeeper queues at shards and pays the cost of reexecution on rare server failures.

Finally, in order to maintain monotonicity of timestamps on gatekeeper failures, a backup gatekeeper restarts the vector clock for the failed gatekeeper. To order the new timestamps with respect to timestamps issued before the failure, the vector clocks in Weaver include an extra *epoch* field which the cluster manager increments on failure detection. The cluster manager imposes a barrier between epochs to guarantee that all servers move to the new epoch in unison.

The cluster manager and the timeline oracle are fault-tolerant Paxos [37] replicated state machines [55].

## 4.4 Proof of Correctness

In this section, we prove that Weaver's implementation of refinable timestamps yields a strictly serializable execution order of transactions and node programs. We structure the proof in two parts—the first part shows that the execution order of transactions is serializable, and the second part shows that the execution order respects wall-clock ordering. We assume that the timeline oracle correctly maintains a DAG of events that ensures that no cycles can arise in the event dependency graph [20].

STRICT SERIALIZABILITY 1. *Let transactions $T_1, \ldots, T_n$ have timestamps $t_1, \ldots, t_n$. Then the execution order of $T_1, \ldots, T_n$ in Weaver is equivalent to a serializable execution order.*

PROOF. We prove the claim by induction on $n$, the number of transactions in the execution.

**Basis**: The case of $n = 1$, the execution of a single transaction $T_1$ is vacuously serializable.

**Induction**: Assume all executions with $n$ transactions are serializable in Weaver. Consider an execution of $n+1$ transactions. Remove any one transaction from this execution, say $T_i, 1 \leq i \leq n+1$, resulting in a set of $n$ transactions. The execution of these transactions has an equivalent serializable order because of the induction hypothesis. We will prove that the addition of $T_i$ to the execution also yields a serializable order by considering the ordering of $T_i$ with an arbitrary transaction $T_j, 1 \leq j \leq n+1, i \neq j$ in three cases.

First, if both $T_i$ and $T_j$ are node programs, then their relative ordering does not matter as they do not modify the graph data.

Second, let $T_i$ be a node program and $T_j$ be a read-write transaction. If $T_i \prec T_j$, either due to vector clock ordering or due to the timeline oracle, then the node program $T_i$ cannot read any of $T_j$'s updates. This is because when $T_i$ executes at a vertex $v$, Weaver first iterates through the multi-version graph data (i.e. vertex properties, out-edges, and edge properties) associated with $v$, and filters out updates that happen after $t_i$ (§ 4.1). If $T_j \prec T_i$, then Weaver ensures that $T_i$ reads all updates, across all shards, due to $T_j$. This is because node program execution is delayed at a shard until the timestamp of the node program is lower than all enqueued read-write transactions (§ 4.1).

Third, we consider the case when both $T_i$ and $T_j$ are read-write transactions. Let $\Gamma_x(T_k)$ denote the real time of execution of transaction $T_k$ at shard $S_x$. If $t_i < t_j$ due to vector clock ordering, then $\Gamma_x(T_i) < \Gamma_x(T_j)\ \forall x$. (§ 4.2). Similarly if $t_j < t_i$ then $\Gamma_x(T_j) < \Gamma_x(T_i)\ \forall x$. For the case when $t_i \approx t_j$, assume if possible that $T_i$ and $T_j$ are not consistently ordered across all shards, i.e. $\Gamma_a(T_i) < \Gamma_a(T_j)$ and $\Gamma_b(T_j) < \Gamma_b(T_i)$. When $T_j$ executes at $S_b$, let $T_i'$ be the transaction that is in the gatekeeper queue corresponding to $T_i$. $T_i'$ may either be the same as $T_i$, or $T_i' \prec T_i$ due to sequence number ordering (§ 4.2). Since $\Gamma_b(T_j) < \Gamma_b(T_i')$, we must have $T_j \prec T_i'$. But since $t_i \approx t_j$, it must also be the case that $t_i' \approx t_j$, and thus the decision $T_j \prec T_i'$ was established at the timeline oracle. Thus we have:

$$T_j \prec T_i' \preceq T_i \tag{1}$$

Now when $T_i$ executes at $S_a$, let $T_j'$ be the transaction in the gatekeeper queue corresponding to $T_j$. By an argument identical to the previous reasoning, we get:

$$T_i \prec T_j' \preceq T_j \tag{2}$$

Eq. 1 and Eq. 2 yield a cycle in the dependency graph, which is not permitted by the timeline oracle.

Since the execution of $T_i$ is isolated with respect to the execution of an arbitrary transaction $T_j\ \forall j, 1 \leq j \leq n+1$, we can insert $T_i$ in the serial execution order of $T_1, \ldots, T_{i-1}$, $T_{i+1}, \ldots, T_{n+1}$ and obtain another serializable execution order comprising all $n+1$ transactions. □

STRICT SERIALIZABILITY 2. *Let transactions $T_1$ and $T_2$ have timestamps $t_1$ and $t_2$ respectively. If the invocation of transaction $T_2$ occurs after the response for transaction $T_1$ is returned to the client, then Weaver orders $T_1 \prec T_2$.*

PROOF. When both $T_1$ and $T_2$ are read-write transactions, then the natural execution order of the transactions on the transactional backing store ensures that $T_1 \prec T_2$. This is because the response of $T_1$ is returned to the client only after the transaction executes on the backing store, and the subsequent invocation of $T_2$ will see the effects of $T_1$.

Consider the case when the invocation of node program $T_2$ occurs after the response of transaction $T_1$. If either $t_1 < t_2$ or $t_2 < t_1$ by vector clock ordering, the shards will order the node program and transaction in their natural timestamp order. When $t_1 \approx t_1$, the timeline oracle will consistently order the two transactions across all shards. The oracle will return a preexisting order if one exists, or order the node program after the transaction (§ 4.1). By always ordering node programs after transactions when no order exists already, the timeline oracle ensures that node programs never miss updates due to completed transactions. □

Combining the two theorems yields that Weaver's implementation of refinable timestamps results in a strictly serializable order of execution of transactions and node programs.

## 4.5 Garbage Collection

Weaver's multi-version graph data model permits multiple garbage collection policies for deleted objects. Users may choose to not collect old state if they wish to maintain a multi-version graph with support for historic searches, or they may choose to clean up state older than the earliest operation still in progress within the system. In the latter case, gatekeepers periodically communicate the timestamp of the oldest ongoing node program to the shards. This allows the shards to delete all versions older than the oldest node program. Weaver uses a similar GC technique to clean up the dependency graph of the timeline oracle.

## 4.6 Graph Partitioning and Caching

Graph queries often exhibit locality in their data access patterns: a query that reads a vertex often also reads its neighbors. Weaver leverages this by dynamically colocating a vertex with the majority of its neighbors, using streaming graph partitioning algorithms [58, 48], to reduce communication overhead during query processing.

In addition to locality in access patterns, graph analyses can benefit from caching analysis results at vertices. For example, a path query that discovers the path $(V_1, \ldots, V_n)$ can cache the $(V_i, \ldots, V_n)$ path at each vertex $V_i$. Weaver enables applications to memoize the results of node programs at vertices and to reuse the memoized results in subsequent executions. In order to maintain consistency guarantees, Weaver enables applications to invalidate the cached results by discovering the changes in the graph structure since the result was cached. Thus, in our previous example, whenever any vertex or edge along the $(V_1, \ldots, V_n)$ path is deleted, the application can discard the cached value and reexecute the node program.

The details of the partitioning and caching mechanisms are orthogonal to the implementation of refinable timestamps and are beyond the scope of this paper. Accordingly, we disable both mechanisms in the system evaluation.

# 5. APPLICATIONS

Weaver's property graph abstraction, together with strictly serializable transactions, enable a wide variety of applications. We describe three sample applications built on Weaver.

## 5.1 Social Network

We implement a database backend for a social network, based on the Facebook TAO API [11], on Weaver. Facebook uses TAO to store both their social network as well as other graph-structured metadata such as relationship between status updates, photos, 'likes', comments, and users. Applications attributes vertices and edges with data that helps render the Facebook page and enable important application-level logic such as access control. TAO supports billions of reads and millions of writes per second and manages petabytes of data [11]. We evaluate the performance of this social network backend against a similar one implemented on Titan [9], a popular open-source graph database, in § 6.2.

## 5.2 CoinGraph

Bitcoin [46] is a decentralized cryptocurrency that maintains a publicly-accessible history of transactions stored in a datastructure called the blockchain. For each transaction, the blockchain details the source of the money as well as the output Bitcoin addresses. CoinGraph is a blockchain explorer that stores the transaction data as a directed graph in Weaver. As Bitcoin users transact, CoinGraph adds vertices and edges to Weaver in real time. CoinGraph uses Weaver's node programs to execute algorithms such as user clustering, flow analyses, and taint tracking. The application currently stores more than 80M vertices and 1.2B edges, resulting in a total of $\sim$ 900 GB of annotated data in Weaver.

## 5.3 RoboBrain

RoboBrain [54] stores a knowledge graph in Weaver that assimilates data and machine learning models from a variety of sources, such as physical robot interactions and the WWW, into a semantic network. Vertices correspond to concepts and edges represent labeled relationships between concepts. As RoboBrain incorporates potentially noisy data into the network, it merges this data into existing concepts and splits existing concepts transactionally. Weaver also enables RoboBrain applications to perform subgraph queries as a node program. This allows ML researchers to learn new concepts without worrying about data or model inconsistencies on potentially petabytes of data [2].

## 5.4 Discussion

The common theme among these applications is the need for transactions on dynamic graph structured data. For example, if the social network backend did not support strictly serializable transactions, it would be possible for reads to see an inconsistent or out-of-date view of the graph, leading to potentially serious security flaws such as access control violations. Indeed, Facebook recently published a study of consistency in the TAO database [40] which showed that in a trace of 2.7B requests over 11 days, TAO served thousands of stale reads that violate linearizability. Similarly, if CoinGraph were to be built on a non-transactional database, then it would be possible for users to see a completely incorrect view of the blockchain. This is possible because (1) the Bitcoin protocol accepts new transactions in blocks and partially executed updates can lead to an inconsistent blockchain, and (2) in the event of a blockchain fork, a database that reads a slightly stale snapshot may return incorrect transactions from the wrong branch of the blockchain fork, leading to financial losses.

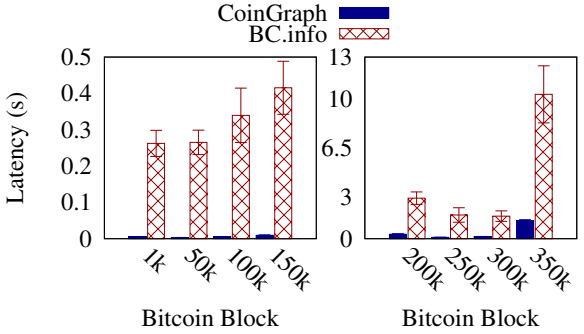



**Figure 7:** Average latency (secs) of a Bitcoin block query in blockchain application. CoinGraph, backed by Weaver, is an order of magnitude faster than Blockchain.info.

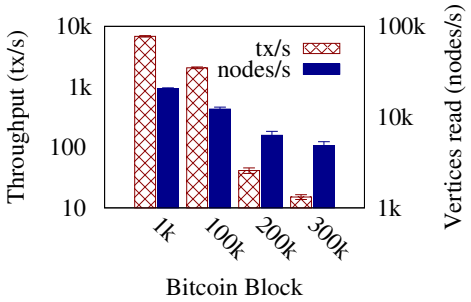



**Figure 8:** Throughput of Bitcoin block render queries in CoinGraph. Each query is a multi-hop node program. Throughput decreases as block size increases since higher blocks have more Bitcoin transactions (more nodes) per query.

While it may be possible to build specialized systems for some of these applications that relax the strict serializability guarantees, we believe that providing transactional semantics to developers greatly simplifies the design of such applications. Moreover, since semantic bugs are the leading cause of software bugs [39], a well-understood API will reduce the number of such bugs. Finally, Weaver is scalable and can support high throughput of transactions (§ 6), rendering weaker consistency models unnecessary.

# 6. EVALUATION

In this section, we evaluate the performance of refinable timestamps in a Weaver implementation comprising 40K lines of C++ code. Our evaluation shows that Weaver:

- enables CoinGraph to execute Bitcoin block queries 8× faster than Blockchain.info [1] (§ 6.1);
- outperforms Titan [9] by 10.9× on social network workload [11] (§ 6.2) and outperforms GraphLab [26] by 4× on node program workload (§ 6.3);
- scales linearly with the number of gatekeeper and shard servers for graph analysis queries (§ 6.4);
- balances the tension between proactive and reactive ordering overheads (§ 6.5).

## 6.1 CoinGraph

To evaluate the performance of CoinGraph we deploy Weaver on a cluster comprising 44 machines, each of which has two 4 core Intel Xeon 2.5 GHz L5420 processors, 16 GB of DDR2 memory, and between 500 GB and 1 TB SATA spinning disks from the same era as the CPUs. The machines are connected with gigabit ethernet via a single top of rack switch, and each machine has 64-bit Ubuntu 14.04 and the latest version of Weaver and HyperDex Warp [21]. The total data stored by CoinGraph comprises more than 1.2B edges and occupies ∼ 900 GB on disk, which exceeds the cumulative memory (704 GB). We thus implement demand paging in Weaver to read vertices and edges from HyperDex Warp in to the memory of Weaver shards to accommodate the entire blockchain data.

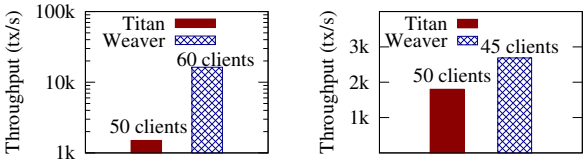
We first examine the latency of single Bitcoin block query, averaged over 20 runs. A block query is a node program in Weaver that starts at the Bitcoin block vertex, and traverses the edges to read the vertices that represent the Bitcoin transactions that comprise the block. We calibrate CoinGraph's performance by comparing with Blockchain.info [1], a state-of-the-art commercial block explorer service backed by MySQL [57]. We use their blockchain raw data API that returns data identical to CoinGraph in JSON format.

The results (Fig. 7) show that the performance of block queries is proportional to the number of Bitcoin transactions in the block for both systems, but CoinGraph is significantly faster. Blockchain.info's absolute numbers in Fig. 7 should be interpreted cautiously as they include overheads such as WAN latency (about 0.013s) and concurrent load from other web clients. The critical point to note is that CoinGraph takes about 0.6–0.8ms per transaction per block, whereas Blockchain.info takes 5–8ms per transaction per block. The marginal cost of fetching more transactions per query is an order of magnitude higher for Blockchain.info, due to expensive MySQL join queries. Weaver's lightweight node programs enable CoinGraph to fetch block 350,000, comprising 1795 Bitcoin transactions, 8× faster than Blockchain.info.

We also evaluate the throughput of block queries supported by CoinGraph. Fig. 8 reports the variation in throughput of the system as a function of the block number. In this figure, the data point corresponding to block $x$ reports the throughput, averaged over multiple runs, of executing block node programs in CoinGraph for blocks randomly chosen in the range $[x, x + 100]$. Since each node program is reads many vertices, Fig. 8 also reports the rate of vertices read by the system. The system is able to sustain node programs that perform 5,000 to 20,000 node reads per second.

| | | |
|---|---|---|
| **Reads 99.8%** | `get_edges` | 59.4% |
| | `count_edges` | 11.7% |
| | `get_node` | 28.9% |
| **Writes 0.2%** | `create_edge` | 80.0% |
| | `delete_edge` | 20.0% |

**Table 1:** Social network workload based on Facebook's TAO.




**(a)** Social Network Workload **(b)** 75% Read Workload

**Figure 9:** Throughput on a mix of read and write transactions on the LiveJournal graph. Weaver outperforms Titan by 10.9× on a read-heavy TAO workload, and by 1.5× on a 75% read workload. The numbers over each bar denote the number of concurrent clients that issued transactions. Reactively ordered transactions comprised 0.0013% of the TAO workload and 1.7% of the 75% read workload.

## 6.2 Social network benchmark

We next evaluate Weaver's performance on Facebook's

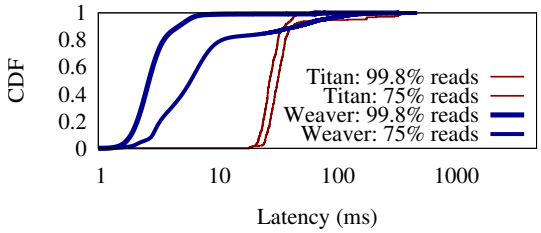



**Figure 10:** CDF of transaction latency for a social network workload on the LiveJournal graph. Weaver provides significantly lower latency than Titan for all reads and most writes.

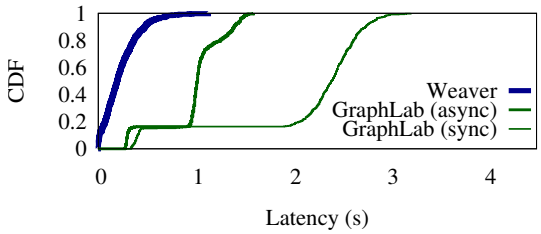



**Figure 11:** CDF of latency of traversals on the small Twitter graph. Weaver provides 4.3×-9.4× lower latency than GraphLab in spite of supporting mutating graphs with transactions.

TAO workload [11] (Table 1) using a snapshot of the LiveJournal social network [10] comprising 4.8M nodes and 68.9M edges (1.1 GB). The workload consists of a mix of reads (node programs in Weaver) and writes (transactions in Weaver) that represent the distribution of a real social network application. Since the workload consists of simple reads and writes, this experiment stresses the core transaction ordering mechanism. We compare Weaver's performance to Titan [9], a graph database similar to Weaver (distributed, OLTP) implemented on top of key-value stores. We use Titan v0.4.2 with a Cassandra backend running on identical hardware. We use a cluster of 14 machines similar to those in § 6.1.

**Throughput**: Fig. 9a shows the throughput of Weaver compared to Titan. Weaver outperforms Titan by a factor of 10.9×. Weaver also significantly outperforms Titan across benchmarks that comprise different fractions of reads and writes as shown in Fig. 9b.

Titan provides limited throughput because it uses two-phase commit with distributed locking in the commit phase to ensure serializability [51]. Since it always has to pessimistically lock all objects in the transaction, irrespective of the ratio of reads and writes, Titan gives nearly the same throughput of about 2000 transactions per second across all the workloads. Weaver, on the other hand, executes graph transactions using refinable timestamps leading to higher throughput for all workloads.

Weaver's throughput decreases as the percentage of writes increases. This is because the timeline oracle serializes concurrent transactions that modify the same vertex. Weaver's throughput is higher on read-mostly workloads because node programs can execute on a snapshot of the graph defined by the timestamp of the transaction.

**Latency**: Fig. 10 shows the cumulative distribution of the transaction latency on the same social network workloads. We find that node program execution has lower latency than write transactions in Weaver because writes include a transaction on the backing store. As the percentage of writes in the workload increases, the latency for the requests increases. In contrast, Titan's heavyweight locking results in higher latency even for reads.

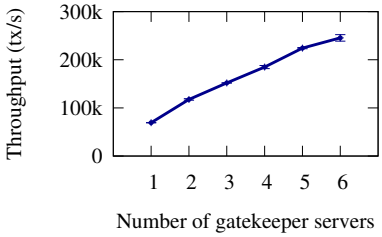



**Figure 12:** Throughput of `get_node` programs. Weaver scales linearly with the number of gatekeeper servers.

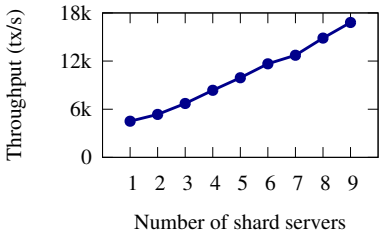



**Figure 13:** Throughput of local clustering coefficient program. Weaver scales linearly with the number of shard servers.

## 6.3 Graph analysis benchmark

Next, we evaluate Weaver's performance for workloads which involve more complicated, traversal-oriented graph queries. Such workloads are common in applications such as label propagation, connected components, and graph search [5]. For such queries, we compare Weaver's performance to GraphLab [26] v2.2, a system designed for offline graph processing. Unlike Weaver, GraphLab can optimize query execution without concern for concurrent updates. We use both the synchronous and asynchronous execution engines of GraphLab. We use the same cluster as in § 6.2.

The benchmark consists of reachability queries on a small Twitter graph [42] consisting of 1.76M edges (43 MB) between vertices chosen uniformly at random. We implement the reachability queries as breadth-first search traversals on both systems. In order to match the GraphLab execution model, we execute Weaver programs sequentially with a single client.

The results show that that, in spite of supporting strictly serializable online updates, Weaver achieves an average traversal latency that is 4× lower than asynchronous GraphLab and 9× lower than synchronous GraphLab. Fig. 11 shows that the latency variation for this workload is much higher as compared to the social network workload, because the amount of work done varies greatly across requests. Synchronous GraphLab uses barriers, whereas asynchronous GraphLab prevents neighboring vertices from executing simultaneously—both these techniques limit concurrency and adversely affect performance. Weaver allows a higher-level of concurrency due to refinable timestamps.

## 6.4 Scalability

To investigate how Weaver's implementation of refinable timestamps scales, we measure Weaver's throughput on microbenchmarks with varying number of servers. We perform the first set experiments on an Amazon EC2 cluster comprising 16 r3.2xlarge instances, each running Ubuntu 14.04 on an 8 core Intel Xeon E5-2670 (Ivy Bridge) processor, 61 GB of RAM, and 160 GB SSD storage. We perform this experiment on the Twitter 2009 snapshot [34] comprising 41.7M users and 1.47B links (24.37 GB).

Fig. 12 shows the throughput of `get_node` node programs

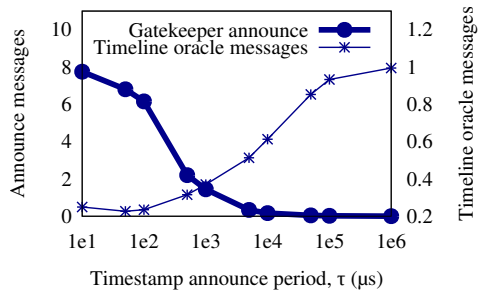



**Figure 14:** Coordination overhead, measured in terms of timestamp announce messages and timeline oracle calls, normalized by number of queries. High clock announce frequency results in large gatekeeper coordination overhead, whereas low frequency causes increased timeline oracle queries.

in Weaver with a varying number of gatekeeper servers. Since these queries are local to individual vertices, the shard servers do relatively less work and the gatekeepers comprise the bottleneck in the system. Weaver scales to about 250,000 transactions per second with just 6 gatekeepers.

However, as the complexity of the queries increases, the shard servers perform more work compared to the gatekeeper. The second scalability microbenchmark, performed on a small Twitter graph with 1.76M edges (43 MB) [42] using the same cluster as § 6.2, measures the performance of the system on local clustering coefficient node programs. These programs require more work at the shards: each vertex needs to contact all of its neighbors, resulting in a query that fans out to one hop and returns to the original vertex. Fig. 13 shows that increasing the number of shard servers, while keeping the number of gatekeepers fixed, results in linear improvement in the throughput for such queries.

The scalability microbenchmarks demonstrate that Weaver's transaction ordering mechanism scales well with additional servers, and also describe how system administrators should allocate additional servers based on the workload characteristics. In practice, an application built on Weaver can achieve additional, arbitrary scalability by turning on node program caching (§ 4.6) and also by configuring read-only replicas of shard servers if weaker consistency is acceptable, similar to TAO [11]. We do not evaluate these mechanisms as they are orthogonal to transaction ordering.

## 6.5 Coordination Overhead

Finally, we investigate the tension between proactive (gatekeeper announce messages) and reactive (timeline oracle queries) coordination in Weaver's refinable timestamps implementation. The fraction of transactions which are ordered proactively versus reactively can be adjusted in Weaver by varying the vector clock synchronization period $\tau$.

To evaluate this tradeoff, we measured the number of coordination messages due to both gatekeeper announces and timeline oracle queries, as a function of $\tau$, to order the same number of transactions. Fig. 14 shows that for small values of $\tau$, the vector clocks are sufficient for ordering a large fraction of the requests. As $\tau$ increases, the reliance on the timeline oracle increases. Both extremes are undesirable and result in high overhead—low values of $\tau$ waste gatekeeper CPU cycles in processing announce messages, while high values of $\tau$ cause increased latency to due extra timeline oracle messages. An intermediate value represents a good tradeoff leading to high-throughput timestamping with occasional concurrent transactions consulting the timeline oracle.

# 7. RELATED WORK

Past work on distributed data storage and graph processing can be roughly characterized as follows.

**Offline Graph Processing Systems**: Google's Pregel [41] computation paradigm has spurred a spate of recent systems [3, 53, 44, 26, 35, 52, 49, 64, 43, 14, 27, 68, 63, 12, 13, 4, 66, 65, 67, 60, 29, 32, 16] designed for offline processing of large graphs. Such systems do not support the rich property graph abstraction, transactional updates, and lookup operations of a typical graph database.

**Online Graph Databases**: The Scalable Hyperlink Store [45] provides the property graph abstraction over data but does not support arbitrary properties on vertices and edges. Trinity [56] is a distributed graph database that does not support ACID transactions. SQLGraph [59] embeds property graphs in a relational database and executes graph traversals as SQL queries. TAO [11] is Facebook's geographically distributed graph backend (§ 5.1). Titan [9] supports updates to the graph and a vertex-local query model.

Centralized graph databases are suitable for a number of graph processing applications on non-changing, static graphs [43]. However, centralized databases designed for online, dynamic graphs [6, 31, 36, 38] pose an inevitable scalability bottleneck in terms of both concurrent query processing and graph size. It is difficult to support the scale of modern content networks [61, 11] on a single machine.

**Temporal Graph Databases**: A number of related systems [30, 15, 25] are designed for efficient processing of graphs that change over time. Chronos [30] optimizes for spatial and temporal locality of graph data similar to Weaver, but it does not support ACID transactions.

Kineograph [15] decouples updates from queries and executes queries on a stale snapshot. It executes queries on the last available snapshot of the graph while new updates are delayed and buffered until the end of 10 second epochs. In contrast, refinable timestamps enable low-latency updates (§ 6.2, § 6.3) and ensure that node programs operate on the latest version of the graph.

**Consistency Models**: Many existing databases support only weak consistency models, such as eventual consistency [11, 40]. Weaver supports strictly serializable operations, as do few other contemporary systems [17, 21, 6, 9].

**Concurrency Control**: Pessimistic two-phase locking [28] ensures correctness and strong consistency but excessively limits concurrency. Optimistic concurrency control techniques [33] (OCC) are feasible in scenarios where the expected contention on objects is low and transaction size is small. FaRM [19] uses OCC and 2PC with version numbers over RDMA-based messaging. Graph databases that support queries that touch a large portion of the graph are not well-served by OCC techniques.

Weaver leverages refinable timestamps to implement multi-version concurrency control [50, 47], which enables long-running graph algorithms to read a consistent snapshot of the graph. Bohm [22] is a similar MVCC-based concurrency control protocol for multi-core settings which serializes timestamp assignment at a single thread. Centiman [18] introduces the watermark abstraction—the timestamp of the latest completed transaction—over traditional logical timestamps or TrueTime. Weaver uses a similar abstraction for garbage collection (§ 4.5) and node programs (§ 4.1). Deuteronomy [38] is a centralized, multi-core database that implements MVCC using a latch-free transaction table.

# 8. CONCLUSION

This paper proposed refinable timestamps, a novel, highly scalable mechanism for achieving strong consistency in a distributed database. The key idea behind refinable timestamps is to enable a coarse-grained ordering that is sufficient to resolve the majority of transactions and to fall back on a finer-grained timeline oracle for concurrent, conflicting transactions. Weaver implements refinable timestamps to support strictly serializable and fast transactions as well as graph analyses on dynamic graph data. The power of refinable timestamps enables Weaver to implement high-performance applications such as CoinGraph and RoboBrain which execute complicated analyses on online graphs.

**Acknowledgments**: We are grateful to the anonymous reviewers as well as colleagues who provided insightful feedback on earlier drafts of the paper. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1518779. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

# 9. REFERENCES

[1] Blockchain. https://blockchain.info.

[2] The Plan to Build a Massive Online Brain for All the World's Robots. http://www.wired.com/2014/08/robobrain.

[3] Apache Giraph. http://giraph.apache.org/.

[4] Apache Hama. http://hama.apache.org/.

[5] Facebook Graph Search. https://www.facebook.com/about/graphsearch.

[6] Neo4j. http://neo4j.org.

[7] TPC-C. http://www.tpc.org/tpcc.

[8] Three and a half degrees of separation. https://research.facebook.com/blog/three-and-a-half-degrees-of-separation/.

[9] Titan. https://github.com/thinkaurelius/titan/wiki.

[10] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group Formation in Large Social Networks: Membership, Growth, and Evolution. *KDD,* pages 44–54, Aug. 2006.

[11] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. *USENIX ATC,* pages 49-60, June 2013.

[12] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big(ger) Graph Analytics on A Dataflow Engine. *PVLDB,* 8(2), 2014.

[13] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li. Improving Large Graph Processing on Partitioned Graphs in the Cloud. *SoCC,* pages 25-36, Oct. 2012.

[14] R. Chen, J. Shi, Y. Chen, and H. Chen. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. *Eurosys,* Apr. 2015.

[15] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: Taking the Pulse of a Fast-Changing and Connected World. *Eurosys,* pages 85-98, Apr. 2012.

[16] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One Trillion Edges: Graph Processing at Facebook-Scale. *PVLDB,* 8(12):1804-1815, 2015.

[17] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-Distributed Database. *OSDI,* pages 251-264, Oct. 2012.

[18] B. Ding, L. Kot, A. J. Demers, and J. Gehrke. Centiman: Elastic, High Performance Optimistic Concurrency Control by Watermarking. *SoCC,* pages 262-275, Nov. 2015.

[19] A. Dragojevic, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. *SOSP,* pages 54-70, Oct. 2015.

[20] R. Escriva, A. Dubey, B. Wong, and E. G. Sirer. Kronos: The Design and Implementation of an Event Ordering Service. *Eurosys,* Apr. 2014.

[21] R. Escriva, B. Wong, and E. G. Sirer. Warp: Lightweight Multi-Key Transactions for Key-Value Stores. *CoRR,* abs/1509.07815, 2015.

[22] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *PVLDB,* 8(11):1190-1201, 2015.

[23] C. J. Fidge. Timestamps in Message-Passing Systems That Preserve the Partial Ordering. *Australian Computer Science Communications,* 10(1):56-66, 1988.

[24] H. Garcia-Molina, J. D. Ullman, and J. Widom. Database Systems. Pearson Prentice Hall, 2009.

[25] B. Gedik and R. Bordawekar. Disk-Based Management of Interaction Graphs. *IEEE Transactions on Knowledge and Data Engineering,* 26(11), 2014.

[26] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. *OSDI,* pages 17-30, Oct. 2012.

[27] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. *OSDI,* Oct. 2014.

[28] J. N. Gray. Notes on Data Base Operating Systems. Springer, 1978.

[29] M. Han and K. Daudjee. Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems. *PVLDB,* 8(9):950-961, 2015.

[30] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: A Graph Engine for Temporal Graph Analysis. *Eurosys,* Apr. 2014.

[31] B. Iordanov. HyperGraphDB: A Generalized Graph Database. *WAIM,* pages 25-36, July 2010.

[32] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. *Eurosys,* pages 169–182, Apr. 2013.

[33] H. T. Kung and J. T. Robinson. On Optimistic

Methods for Concurrency Control. *ACM ToDS,* 6(2):213–226, 1981.

[34] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a Social Network or a News Media? *WWW,* pages 591-600, Apr. 2010.

[35] A. Kyrola, G. E. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. *OSDI,* pages 31-46, Oct. 2012.

[36] A. Kyrola and C. Guestrin. GraphChi-DB: Simple Design for a Scalable Graph Database System - on Just a PC. *CoRR,* abs/1403.0701, 2014.

[37] L. Lamport. The Part-Time Parliament. *ACM ToCS,* 16(2):133-169, 1998.

[38] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. High Performance Transactions in Deuteronomy. *CIDR,* Jan. 2015.

[39] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have Things Changed Now?: An Empirical Study of Bug Characteristics in Modern Open Source Software. *ACM Workshop on Architectural and System Support for Improving Software Dependability,* pages 25-33, Oct. 2006.

[40] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential Consistency: Measuring and Understanding Consistency at Facebook. *SOSP,* Oct. 2015.

[41] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. *SIGMOD,* pages 135-146, June 2010.

[42] J. J. McAuley and J. Leskovec. Learning to Discover Social Circles in Ego Networks. *NIPS,* pages 548-556, Dec. 2012.

[43] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what COST? *HotOS Workshop,* May 2015.

[44] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. *SOSP,* pages 439-455, Nov. 2013.

[45] M. Najork. The Scalable Hyperlink Store. *Hypertext,* pages 89-98, June 2009.

[46] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008.

[47] T. Neumann, T. Mühlbauer, and A. Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. *SIGMOD,* pages 677-689, 2015.

[48] J. Nishimura and J. Ugander. Restreaming Graph Partitioning: Simple Versatile Algorithms For Advanced Balancing. *KDD,* pages 1106-1114, Aug. 2013.

[49] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing Large Graphs on Multi-Cores With Graph Awareness. *USENIX ATC,* pages 41-52, June 2012.

[50] D. P. Reed. Naming And Synchronization in a Decentralized Computer System. Massachusetts Institute of Technology, Technical Report, 1978.

[51] M. A. Rodriguez and M. Broecheler. http://www.slideshare.net/slidarko/titan-the-rise-of-big-graph-data.

[52] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. *SOSP,* pages 472–488, Nov. 2013.

[53] S. Salihoglu and J. Widom. GPS: A Graph Processing System. *SSDBM,* July 2013.

[54] A. Saxena, A. Jain, O. Sener, A. Jami, D. K. Misra, and H. S. Koppula. RoboBrain: Large-Scale Knowledge Engine for Robots. Cornell University and Stanford University, Technical Report, 2015.

[55] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys,* 22(4), 1990.

[56] B. Shao, H. Wang, and Y. Li. Trinity: A Distributed Graph Engine on a Memory Cloud. *SIGMOD,* pages 505-516, June 2013.

[57] P. Smith. Personal communication, 2015.

[58] I. Stanton and G. Kliot. Streaming Graph Partitioning for Large Distributed Graphs. *KDD,* pages 1222-1230, Aug. 2012.

[59] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. T. Xie. SQLGraph: An Efficient Relational-Based Property Graph Store. *SIGMOD,* pages 1887-1901, 2015.

[60] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "Think Like a Vertex" to "Think Like a Graph". *PVLDB,* 7(3):193-204, 2013.

[61] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The Anatomy of the Facebook Social Graph. *CoRR,* abs/1111.4503, 2011.

[62] R. van Renesse and F. B. Schneider. Chain Replication for Supporting High Throughput and Availability. *OSDI,* pages 91-104, Dec. 2004.

[63] K. Wang, G. Xu, Z. Su, and Y. D. Liu. GraphQ: Graph Query Processing with Abstraction Refinement—Scalable and Programmable Analytics over Very Large Graphs on a Single PC. *USENIX ATC,* pages 387-401, July 2015.

[64] W. Xie, G. Wang, D. Bindel, A. Demers, and J. Gehrke. Fast Iterative Graph Computation with Block Updates. *PVLDB,* 6(14):2014–2025, 2013.

[65] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A Block-Centric Framework for Distributed Computation on Real-World Graphs. *PVLDB,* 7(14):1981-1992, 2014.

[66] D. Yan, J. Cheng, Y. Lu, and W. Ng. Effective Techniques for Message Reduction and Load Balancing in Distributed Graph Computation. *WWW,* pages 1307-1317, 2015.

[67] C. Zhou, J. Gao, B. Sun, and J. X. Yu. MOCgraph: Scalable Distributed Graph Processing Using Message Online Computing. *PVLDB,* 8(4):377-388, 2014.

[68] X. Zhu, W. Han, and W. Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. *USENIX ATC,* pages 375-386, July 2015.