ON CONSISTENT AND EFFICIENT GRAPH DATA MANAGEMENT

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Ayush Dubey

May 2018

ON CONSISTENT AND EFFICIENT GRAPH DATA MANAGEMENT

Ayush Dubey, Ph.D.

Cornell University 2018

This dissertation describes techniques to store and process large graphs in modern datacenters with high performance and strong consistency guarantees. Graph-structured data is ubiquitous: social networks, content networks, cryptocurrency transaction histories, and business analytics routinely store and manipulate large graphs. For reasons of scale, both in terms of data size as well as workload volume, it is necessary to store such large graphs in a distributed fashion. Moreover, graph workloads have unique characteristics, such as long running read queries interspersed with shorter updates, that naturally lead to a programming interface consisting of a hybrid of transactions and analytics. Providing efficient and consistent access to graph-structured data is a significant challenge.

This dissertation makes three contributions. First, it describes a novel technique to order distributed transactions by introducing the concept of an ordering service. An ordering service seeks to simplify the design of modern distributed systems by factoring out the task of ordering from the core system into a separate service. Second, it details techniques that scale up the performance of a centralized ordering service by combining it with a lightweight timestamping mechanism. Third, it describes a full implementation of WEAVER, a new distributed, transactional graph store that includes mechanisms for practical and efficient graph data management, such as dynamic resharding of graph partitions and caching of query results. Overall, these techniques lead to a scalable

and consistent graph store that is capable of supporting modern distributed applications with high performance.

BIOGRAPHICAL SKETCH

Ayush Dubey was born in New Delhi, India. He received his Bachelor of Technology degree in 2012 from the Indian Institute of Technology in New Delhi. During his undergraduate studies, he had the opportunity to conduct research on computer networking, both at IIT Delhi as well as IBM Research. This led to the pursuit of a Ph.D in beautiful Ithaca at Cornell University with Emin Gün Sirer, where he enjoyed building large-scale distributed systems. He was also lucky to get the opportunity to experience research in industry at Microsoft Research. He got his Master of Science degree in Computer Science in 2015, and his Doctor of Philosophy degree, also in Computer Science, in 2018.

To my parents

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor Emin Gün Sirer. Gün has been the most influential teacher in my life. He has taught me the value of building real systems and has instilled a strong work ethic in me. He has deeply shaped my taste in research and systems problems. In addition, Gün has provided unwavering support during the past and has believed in me when I did not believe in myself. For these and many other reasons, I am sincerely grateful towards Gün.

I would also like to thank Srikanth Kandula, with whom I worked closely during the last 2 years of my graduate career. Srikanth provided me with a fantastic opportunity to work on production systems at Microsoft. I learnt important skills on system building, careful design, and detailed experimentation from him. For the chance to perform impactful research on real-world systems, I am grateful towards Srikanth.

I am thankful to my committee members: Nate Foster and Jon Kleinberg. I gained fundamental and theoretical insight on graphs and social networks from attending Jon's course as well as over multiple conversations. Nate has provided great advice on my research and on my dissertation drafts. Most of all, both Nate and Jon have made the path to graduation straightforward.

I would like to thank the Computer Science faculty, students, and staff at Cornell University. First, the department displayed great belief in admitting me to the program and providing me with the opportunity to perform world-class research. Second, all conversations and interactions with various faculty have been intellectually stimulating and led to my personal and professional growth. Third, my student colleagues in Gün's group and the systems lab, including Deniz, Robert, Sean, Efe, Greg, Kai, Soumya, Ted, Natacha, Tom, Eliza, Han,

Kevin, and many others, have made the past 5.5 years all the more enjoyable. Finally, the admins in our department really made it easy to navigate the bureacratic process at Cornell.

Last, but not least, I am eternally grateful to my family and friends for supporting me during my graduate career. These are the folks I leaned on when the times got tough, and they have pushed me to achieve more while also maintaining a good work-life balance. Among my friends, I would especially like to thank Ajay, Aman, Archit, Aseem, Chaitanya, Nitin, Pooja, Ritika, Sachin, Shretima, Tanay, and Vaibhavi for the many hangouts and trips that made the last few years really enjoyable. My parents and my sister have always stuck by my side and supported all my endeavours while sacrificing a lot themselves, and for that I will always be grateful. Finally, my fiancée Kritika has been my rock. She has seen the worst and the best of me. She has been there when I was down and she enabled me to move and achieve well beyond what I thought possible. For this and much more, I would like to thank her.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In this dissertation, we present techniques for storing and processing large graph-structured data with high efficiency and strong guarantees.

1.1 Motivation

Graph-structured data arises naturally in a wide range of fields that span science, engineering, and business. Social networks, such as Facebook, store both the social graph as well as the content network in graph stores [22]. Social interaction analyses [85], such as those on Twitter, model interactions between users as large graphs. Cryptocurrency transaction histories also comprise a graph where vertices are wallets and edges are transactions. Knowledge graphs [5, 107] structure topics as vertices while edges capture the underlying semantics of the relationships. Many modern datasets and applications are modeled as a set of vertices with directed edges between them.

Because graph-structured data is ubiquitous, there is a growing need for systems that can store and process such large graphs. A modern graph store has at least three desirable features. First, similar to traditional databases, the graph store should provide a well-defined and strong consistency guarantee for its programming interface. The gold standard for consistency models is linearizability for single operations and strict serializability for transactional interfaces [60]. Many data stores preemptively eschew strong consistency models in the quest for better performance. Guarantees like eventual consistency [126, 34] and causal consistency [77], while useful in certain scenarios, are often hard to

understand and program against and lead to developer bugs and access violations [79].

Second, modern graph stores power applications that have high efficiency requirements. Efficiency is a two-pronged challenge that consists of throughput and latency. *Throughput* denotes the number of queries processed by the system per unit time. *Latency* is a measure of the amount of time taken per query by the system. Both of these metrics are important when measuring the efficiency of a graph store. For example, in the case of Google Search which is powered by the aforementioned knowledge graph [5], low latency is essential because it is a user-facing application. On the other hand, throughput is the preferred metric for performance of analytics and data mining workloads on large graphs [51].

Finally, typical operations on graphs comprise a hybrid of transactional updates and large analytical queries. Transactional operations on graphs are useful for point queries and updates. For example, operations such as reading a vertex and its adjacent edges, or adding an edge between two vertices, are well-suited for the transactional paradigm. Analytical queries, on the other hand, enable bulk read of large portions of the graph and associated metadata. For example, a breadth-first search style traversal that starts at a vertex may read the entire connected component, which in general may be large. Similarly, bulk synchronous algorithms such as PageRank [81] traverse the entire graph structure, consisting of all vertices and edges, multiple times. Essentially, graph stores are a specific instance of HTAP [64], systems designed to support workloads that are a hybrid of transactional and analytical processing.

The combination of a strongly consistent and highly efficient graph store that enables both transactional and analytical operations is the primary focus of this

dissertation. In the next section, we discuss challenges afforded by modern applications and graph-structure data that complicate the design and implementation of a modern graph store.

1.2 Challenges

Three key challenges with modern graph structured data and applications that work with such graphs is the volume of the data, the scale of the workload, and the specific nature of graph queries.

1.2.1 Data Volume

Modern graph structured data is extremely large. Facebook's social graph cache contains a vertex for every user as well as content item, and an edge for interactions between users and content, which totals many petabytes of data [22]. Since Facebook ingests multiple billions of items per day [2], conservative estimates put the size of the graph on the order of trillions of graph elements. Google's knowledge graph stores upwards of 70 billion items [3]. Large web graphs, which are used for data mining and page ranking purposes, comprise hundreds of billions of vertices and edges [19, 6].

Such massive graphs pose a great challenge in efficient storage and query processing. The scale of such data far exceeds the limits of commodity servers, and techniques that distribute the storage without compromising efficiency are a key challenge for applications that work with this data.

1.2.2 Workload Scale

In addition to the scale of the data, the scale of the workload is an orthogonal, yet equally important challenge. Twitter received 300 million tweets per day in 2016 [4], and 660 million tweets per day at its peak in 2014. Facebook received 350 million photo uploads per day in 2013 [1], and 6 billion likes per day in 2015 [2], each of which is a vertex or an edge in the social graph [22]. Google Search, which is powered by their knowledge graph [5], received 1 billion search queries per day in 2017 [113].

Such scale of queries represents a challenging workload for modern databases. It would be hard to serve these queries from a single centralized server, and thus distribution is an important challenge for query processing on modern graphs as well.

1.2.3 Nature of Graph Queries

Finally, the inherent nature of graph queries represents a challenging workload. Analytical 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 [15], 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 point queries are much smaller; the `NewOrder` transaction in the TPC-C benchmark [120], which comprises 45% of the frequency distribution, consists of 26 reads and writes on average [42]. Techniques such as optimistic concurrency control or distributed two-phase locking result in poor throughput when con-

current queries try to read and write large subsets of the graph.

A technique for ordering hybrid transactional and analytical workloads on large graphs with both strong consistency guarantees and high efficiency is a key challenge for the success of any modern graph store.

1.3 Contributions

In this dissertation, we describe the design and implementation of distributed graph stores that aim to achieve the triple of strong consistency, high efficiency, and support for hybrid transactional and analytical programming interface for graphs. We specifically focus on techniques that provide consistent operation ordering, increasing scalability, and automatically managing the distributed data and computation.

First, we achieve consistent and reliable operation ordering by introducing the idea of an *event ordering service*. At a high-level, an event ordering service is a new component designed to efficiently manage and order events in a distributed system. It seeks to revolutionize the design of distributed systems by factoring out the task of ordering—one of the toughest problems in asynchronous networks—into a separate service. We implement a centralized event ordering service called KRONOS. KRONOS uses abstract event identifiers in order to efficiently order events and enable a variety of application-dependent correctness guarantees. We use KRONOS to implement KRONOGRAPH, a distributed graph store that orders transactions in a strictly serializable transaction history.

Next, we improve the scalability of the centralized transaction ordering mechanism using a novel timeline management technique called *refinable timestamps*. In the refinable timestamps paradigm, each query and transaction receives a timestamp upon entry in the system that enables partial order between operations. For a small subset of the queries that are conflicting, the protocol further refines the timeline of operations using a centralized, reactive mechanism. Together, the combination of a lightweight proactive timestamping technique with a reactive precise refinement results in high-performance, scalable event ordering. We use refinable timestamps to implement strict serializability in WEAVER, a distributed transaction graph database that achieves near linear scalability on social network workloads with 2 million transactions per second using only 8 commodity server machines.

Finally, we present the overall design and implementation of WEAVER, including novel techniques that enable a hybrid transactional and analytical interface, dynamic sharding, partial query caching, bulk ingest of data, dynamic code deployment, and historic queries. WEAVER leverages refinable timestamps to order transactions, and in spite of a rich feature set, achieves throughput that is $10\times$ higher than commercial state of the art graph databases on mixed read-write workloads.

We give an overview of each of the three contributions in the following sections.

1.4 Reliable and Consistent Distributed Ordering

Ordering transactions is a key challenge to the correctness and performance of a modern graph store. However, ordering is a challenge that transcends data stores and is ubiquitous across many distributed systems. For example, consistent updates to a software-defined network require reasoning about the order in which configuration updates took place relative to routing requests. Similarly, determining order is crucial in the forensic analysis of a network compromised due to unauthorized intrusion.

In order to simplify distributed system design and enable ordering across systems, we introduce the concept of an *event ordering service*. Such a service enables users to dictate and to query order between events, both within and across systems. Modern networked applications can leverage an ordering service to determine the order of different operations, thereby avoiding the complexity of complicated application-level logic. Moreover, an ordering service enables composing the ordering decisions of different applications, as well as ordering events that span applications—the events and ordering decisions of one application may be used meaningfully by another application.

In Chapter 2, we discuss the spectrum of implementation choices for an event ordering service, and we describe the design of KRONOS, an instantiation of this concept. KRONOS enables two key functionalities. First, it allows applications to record and enforce a desired order between events. Second, it enables applications to query and deduce an already established order between events.

While multiple implementations are possible, KRONOS keeps track of depen-

dencies at a very fine granularity. KRONOS stores an event dependency graph where each event corresponds to a vertex, and a directed edge between two vertices signifies a happens-before relationship between the corresponding events. By storing the ordering relationships in a logically centralized graph, KRONOS enables composing ordering decisions across applications. We show in Chapter 2 that our implementation is able to correctly and efficiently maintain the order between events, and that it can lead to a graph store with a serializable transaction history and good performance.

1.5 Scalable Transaction Ordering

We build upon the event ordering techniques described in the previous section to improve their scalability. We introduce WEAVER, a new distributed graph database that can scalably order transactions using refinable timestamps. This technique uses a highly scalable and lightweight vector timestamp for ordering the majority of operations and relies on a fine-grained event ordering service, KRONOS, for ordering the remaining, potentially-conflicting reads and writes. By relying on loosely synchronized clocks, the refinable timestamps mechanism can reduce dependence on a centralized component without compromising on the consistency guarantees.

The two-step ordering mechanism brings to light a tradeoff in distributed ordering. One source of ordering overhead is proactive synchronization of clocks in WEAVER and other similar techniques like TrueTime [31]. The other ordering overhead is reactive in techniques such as KRONOS. We discuss the implications and balance of this tradeoff in Chapter 3.

1.6 Graph Data Management

In addition to transaction ordering, a distributed graph store needs other careful design decisions for high performance. WEAVER introduces node programs, a new technique for executing read-only graph analyses on large dynamic graphs. This, combined with a traditional transactional programming interface for updates, leads to a hybrid transactional analytical API.

WEAVER includes several features designed for high performance and ease of use. The key enabler for many of these features is the choice of multi-version concurrency control. The different versions of the graph are defined by the refinable timestamps attached to each piece of data. This allows for many optimizations. While initially WEAVER randomly partitions the graph by vertices, it includes a dynamic resharding technique that migrates portions of the graph data on-the-fly, even as the system is concurrently serving queries and transactions. WEAVER enables partial and complete results of previous queries to be cached for later reuse, without compromising consistency guarantees. Users may issue queries against a historic version of the graph by leveraging the timestamp metadata attached to each version of graph vertices and edges. Finally, a specialized implementation enables bulk loading a large graph for the initial data ingest that sidesteps slower transactional paths. We describe these implementation details in Chapter 4.

1.7 Summary and Organization

Overall, this dissertation describes multiple techniques that come together to enable an efficient and consistent graph store. The key contribution is novel distributed ordering paradigms, which we describe in Chapter 2 and Chapter 3. In addition, a number of different features and performance optimizations enable a practical and fast graph store, WEAVER, which are detailed in Chapter 4. We describe a full evaluation of the various components of the system, which includes microbenchmarks focused on ordering, caching, and graph partitioning, as well as end to end benchmarks, in Chapter 5. Chapter 6 discusses a wide variety of related work that spans distributed ordering, modern data stores, and graph management. Finally, we conclude and discuss directions for future research in Chapter 7.

CHAPTER 2

DISTRIBUTED ORDERING SERVICES

2.1 Introduction

We first tackle the problem of consistently ordering transactions and analytical queries in a graph store. Because modern graph data and applications have immense scale, both in terms of data size as well as workload volume (Section 1.2), we focus on a distributed system architecture and discuss the challenges of ordering transactions in such a scenario.

While ordering is a key challenge in the context of distributed data stores, reasoning about time and order is central to the design and implementation of nearly all distributed systems. In addition to distributed transactions, many common applications such as network policy enforcement and forensic analysis require a notion of order between events.

However, timing guarantees and reasoning about ordering is, at best, an after-thought—modern distributed systems do not support time as a first-class citizen. Many techniques have been previously suggested to capture dependencies and ordering in distributed systems. The three most commonly used approaches are Lamport timestamps [72], vector clocks [45, 83], and consensus-based approaches [71, 96]. While these schemes differ in how they capture dependencies (whether they are expressed in a happens-before relationship, a time vector, or an assigned slot in a timeline), they share the same structure. Namely, they are instantiated separately within each independent distributed system and track dependencies solely within the purview of that system, often

by monitoring communication at the boundaries of internal components. This leads to the following problems:

- False negatives: Because a given system only knows of relationships within its purview, it will miss any dependencies that are formed over external channels [29, 72].
- False positives: Because false negatives have significant consequences, distributed systems often err by conservatively assuming a causal relationship even when a true dependence might not exist. For instance, many vector clock implementations will establish a happens-before relationship between every message sent out and all messages received previously by the same process, even if those messages did not play a causal role.
- Early assignment: Time ordering systems often impose an order too early on concurrent events, thereby reducing the flexibility of the system. For instance, Lamport timestamps and vector clocks order events at the time when timestamps are assigned.
- Composition: Modern networked applications, including almost all high-performance web services, are increasingly built on top of multiple distributed subsystems, and would benefit from a notion of dependence that carries over and composes between independent subsystems.

The need for order in distributed systems manifests itself in two ways. First, applications may need to *establish* a synthetically-created order between events as dictated by the application logic. For example, achieving linearizability in a distributed data store requires that an operation B get processed at every node after a previously completed operation A. To achieve this, the application needs

to record and maintain that A happened before B. Second, applications may need to *discover* a naturally-occurring order between extant events in the system. For instance, a per-packet consistent SDN requires that each packet discover whether a network update happened before, or after, the packet arrival and accordingly choose the forwarding rule at each switch. This scenario requires discovery of order between events *a posteriori* to order assignment.

In this chapter, we propose a radically different approach to solve the challenge of managing order in distributed systems: an *event ordering service*. Such a service factors event ordering out of independent subsystems into a shared component that tracks timing dependencies between actions that traverse multiple subsystems, thereby reducing the complexity of the application. The event ordering service is entirely responsible for managing the order between distributed events, and presents a simple API that enables applications to manage the timeline of events. Moreover, the event ordering service enables composition of ordering decisions of different applications, as well as ordering events across applications.

We first describe the API of our proposed event ordering service. Specifically, the API is centered around abstract events that hold application-specific meaning. The various subsystems may contact the ordering service to create or delete events with corresponding event handles. Subsequently, they may either query the ordering service for a preexisting order between events, or issue a call to assign an order between two events.

Next, we discuss the range of designs possible for such an ordering service. Some system designs may rely on clock synchronization to achieve ordering between events [88, 74, 31]. Other designs may refrain from making any syn-

chronicity assumptions. This chapter includes a survey of the design space, including other contemporary event ordering services inspired by our work [20], in Section 2.3.

In the rest of this chapter, we describe KRONOS, a concrete instantiation of the event ordering service concept. The main tenets of KRONOS's approach are threefold. First, it keeps track of dependencies at a very fine granularity; specifically, it makes the case for maintaining the full *event dependency graph*. This yields expressive systems that can distinguish and take advantage of concurrency where available. Second, it implements late *time-binding*, that is, picking an absolute order of events that is congruent with constraints as late as possible. Late assignment of time order provides extensive freedom to applications on how to schedule a set of concurrent events whose time order is underconstrained, a situation commonly encountered in practice. Finally and most importantly, KRONOS enables applications to query the graph and determine if two events are concurrent, which in turn identifies those instances where the application can make its own decision on how to order these concurrent events.

We have built several applications and examples on top of KRONOS. Our first application illustrates how KRONOS can be used to improve the user experience in a social network. The second application is KRONOGRAPH, an online, strongly-consistent graph store that uses KRONOS to order writes and graph traversals. Finally, we describe a transactional key-value store that uses KRONOS to serialize transactions in an off-the-shelf key-value store.

Overall, this chapter makes three contributions. First, we introduce a new abstraction for event ordering and propose a new service and minimal API for distributed systems. Second, we describe the implementation of multiple appli-

cations on top of an event ordering service, focusing mainly on the way in which each application exploits the event ordering provided by KRONOS. Finally, we present the full implementation of KRONOS, an instance of a centralized event ordering service. We also discuss techniques to improve performance, both for KRONOS as well as the applications built using KRONOS.

2.2 Event-based Abstractions

This section describes the core abstractions for an event ordering service, including the programming interface, as well as the implementation of several applications based on such a service. We discuss potential implementations in Section 2.3.

2.2.1 Events

We propose a standalone shared service that tracks dependencies and provides time ordering for distributed applications. The central entity in the ordering service is an *event*, an application-determined set of state changes that take place atomically, associated with a unique identifier. Events are akin to basic blocks in programming languages; they may be as fine-grained as the execution of a single instruction or receipt of a single message, or as coarse grained as system-wide state changes spanning multiple hosts. In practice, applications create events that correspond to any number of actions they take internally in response to externally-provided inputs. For example, a transactional key-value store could map each transaction to a KRONOS event. KRONOS leaves the pre-

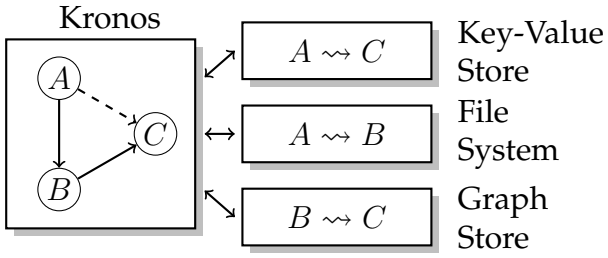
cise semantics associated with events up to application and concerns itself with establishing a partial order between events.

2.2.2 Event Dependency Timeline

The central task of the ordering service, then, is to enable applications to quickly order events along a timeline using the event dependency graph. The ordering service provides interfaces by which applications may:

- create new events,
- record and enforce a desired order between events, that is `assign_order(e_1, e_2)`, and
- query and deduce an established order between events, that is `query_order(e_1, e_2)`.

To permit applications using the ordering service to make decisions that rely upon the timeline, it upholds two invariants called the *coherency* and *monotonicity* invariants. The coherency invariant ensures that the events can be arranged into a possible timeline by ensuring that the dependency graph is free of cycles. The existence of an ordering relationship between events, potentially deduced transitively through a series of direct orders, implies that the ordering service has made a series of commitments that force one event to necessarily succeed the other, in which case the ordering service communicates this ordering to applications so that they can act accordingly. The coherency invariant prevents logical contradictions within the timeline represented by the event dependency graph.



A: Alice updates new photos which only her friends may access. The ACL is stored in the key-value store, and the photos themselves are stored on the file system.
B: Alice uploads a photo to the album and tags Bob in the photo. The photo is stored on the file system, and the graph store records that Bob is tagged by the photo.
C: Bob likes Alice's photographs. This action checks the ACL, and records the "like" in the graph store.

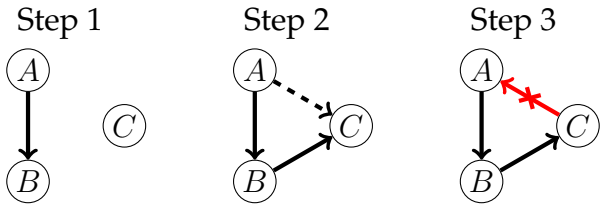
Figure 2.1: A social network built using an ordering service, a key-value store, a graph store, and a file system. Each ordering service event corresponds to an action in the application. The ordering service ensures that the transitive dependency $A \rightsquigarrow B \rightsquigarrow C$ will be enforced at the key-value store as $A \rightsquigarrow C$, even though the key-value store is unaware of event B.

The ordering service's monotonicity invariant ensures that happens-before relationships, once established, are incontrovertible. Applications may safely commit to a particular time order once established by the service, as subsequent operations can only further constrain, but never violate, established dependencies. This enables clients to be able to issue side-effects and produce user-visible output based upon the responses of the ordering service.

2.2.3 Using the Abstraction

To see how the event dependency graph may be used by applications, consider a social network that allows users to upload, tag, and like photographs of each other. This application stores users' photos in a file system, records tags and "likes" in a graph store, and maintains ACLs in a key-value store. When Alice uploads her photos to the application, it stores her photos in the file system and updates the key-value store to store the ACLs. Similarly, when Alice uploads a photo in which she has tagged Bob, the application stores the photo on the file system and records in the graph store that Bob is tagged in the photo. Finally, Bob can like the photo, which records Bob's actions in the graph store only after checking that Bob is permitted to do so by the ACLs stored in the key-value store. Since the system is consists of three separate components, in the absence of order, it is possible for the ACLs setup by Alice in the first step to be improperly retrieved in the third step, potentially exposing her photos to an unintended audience.

This example social network application can use the ordering service to ensure that this disastrous situation is reliably avoided. Each user-facing change to the social network is represented in the ordering service as an event. Thus, when Alice initially uploads her photos or tags Bob, or when Bob likes Alice's photo, the application creates an event in the ordering service to represent the user's interaction with the service. Individual components of the social network application will each process a different subset of these events, and each can impose an order on the subset they process. The ordering service can then maintain an application-wide consistent timeline that spans all events, as shown in Figure 2.1. Figure 2.2 illustrates how the application may incrementally build



A: Alice updates new photos to an album.
B: Alice uploads a photo to the album and tags Bob.
C: Bob likes Alice's photographs.

Figure 2.2: As dependencies are added between events, edges are added to the event dependency graph. The application adds dependencies between A, B, and C insteps 1 and 2. KRONOS prohibits the application from adding the dependency $C \rightsquigarrow A$ in step 3 because the application already established $A \rightsquigarrow B \rightsquigarrow C$.

the timeline within the ordering service. After Alice's actions are recorded by the ordering service, it ensures that Bob's request will be correctly ordered after Alice's actions.

2.3 Design Space

The design space for an ordering service in distributed systems is surprisingly rich.

2.3.1 Synchronous Designs

A natural approach to ordering using timestamps is the use of wall clock time. The advantage of using real timestamps is that they automatically have semantics that transcend system boundaries. If two systems timestamp their events

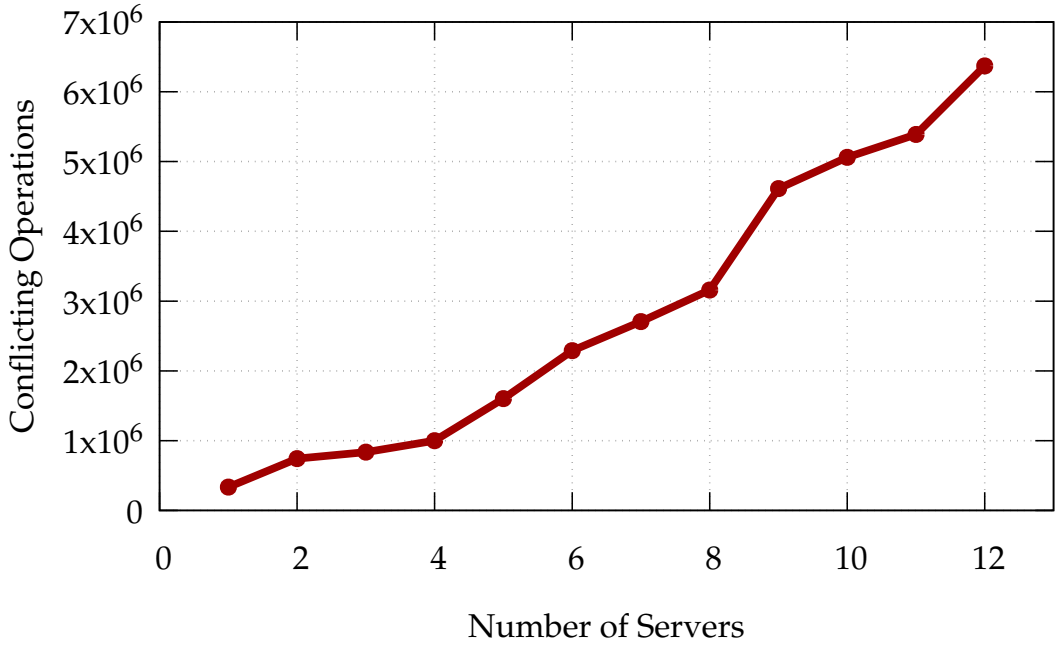


Figure 2.3: The number of conflicting operations due to ambiguous timestamps issued by an NTP-based timeline oracle. This graph depicts a lower bound; actual numbers may be higher. As more servers are added to the cluster, the uncertainty in timestamps increases, reducing the efficacy of such a design.

internally using real clocks with infinite precision, they can share timestamp data since the timestamps have common semantics for both the systems.

The ordering primitives for real timestamps have trivial, efficient implementations. `query_order` returns the real time order between the two timestamps while `assign_order` ($t_1 \prec t_2$) returns true if t_1 happens before t_2, otherwise it returns false. Consequently, if the ordering service can be implemented using real time, the distributed components of the application can execute ordering primitives locally without ever having to contact the oracle.

Of course, the key challenge in such an implementation is ensuring the clocks across the network stay synchronized. Clocks, when left to their own

devices, tend to drift; looser time bounds extend the range where two time markers are incomparable. In Figure 2.3 we show the number of conflicting timestamps issued by an NTP service [88] running for 10 minutes on a 12 server cluster. In this experiment, each server synchronizes against a local stratum 2 server and is responsible for assigning timestamps to operations and we conservatively estimate an incoming rate of 100,000 operations per server. The figure shows that as we add more servers to the NTP cluster, the uncertainty in timestamps increases, and hence more operations receive incomparable timestamps.

There have been techniques in the past that improve on the synchronization bounds [74] and subsequently use real time to achieve a total order among events [31]. One example is Google's TrueTime, which uses expensive high-precision synchronization references like GPS and atomic clocks. To handle the issue of incomparable timestamps, TrueTime-backed applications require artificial delays. More importantly, such timestamps do not permit the discovery of total order *a posteriori* between events, which is unacceptable for applications such as forensic analysis of data breaches.

2.3.2 Asynchronous Designs

At the other end of the design space, an ordering service can be asynchronous. Such an implementation requires little to no synchronization between servers. Any component of the distributed system can request an event handle from the service, and can be issued an opaque handle as a response from the timeline oracle. It can subsequently query the order between events by contacting the oracle. Any component can also impose an order between events, by contacting

the service.

2.3.2.1 Centralized Architecture

The event ordering service, as described in this chapter, implements a happens-before relationship between the opaque event handles to realize the asynchronous implementation. The simplest implementation of this concept is through the use of an increasing counter implemented as a replicated state machine. One of the main drawbacks of using a replicated counter as an ordering service is that the service unnecessarily imposes order on events that may never need to be ordered. The centralized counter is draconian in its ordering guarantees—every event is ordered with respect to every other event. This leaves no room for concurrency, even when the application can tolerate it. In Section 2.4, we discuss a centralized implementation that relaxes the total ordering constraint.

The main drawback of centralized designs is the inherent scalability bottleneck. As the number of networked components in the system increases, the centralized service inevitably blocks fast progress in the system.

2.3.2.2 Distributed Architecture

In contrast, a completely distributed design of an ordering service is also possible. Lamport, in his seminal paper on distributed event ordering [72], describes the Lamport clock technique for achieving a partial order between distributed events. An ordering service that comprises multiple servers, each of which synchronize their logical clocks using the Lamport clock algorithm, thereby pro-

vides an entirely distributed implementation of an ordering service. Saturn is yet another distributed event ordering service [20]. It relies on scalar, per-client timestamps to achieve partially ordered events and enable a geo-replicated causally consistent datastore.

Of course, the main drawback with such a design is that it guarantees only partial order between events. Lamport clocks guarantee that if event a happens-before event b, then $clk(a)$ is smaller than $clk(b)$. The converse is not guaranteed, that is if $clk(a) < clk(b)$, then a may not happen-before b. Even though the paper describes a technique for achieving total order by breaking ties deterministically, such a total ordering does not capture causality. Similar techniques such as vector clocks [45, 83] and version vectors [82] also guarantee only partial order.

2.3.2.3 Hybrid Architecture

We can combine the centralized and distributed implementations to get the best of both worlds. The Weaver graph database [38] describes such an asynchronous ordering service. The main idea behind Weaver's *refinable timestamps* is to use loosely synchronized logical clocks, such as those described in Section 2.3.2.2, to order the bulk of the operations, and resort to a centralized ordering service, such as those described in Section 2.3.2.1, for a small fraction of the total number of operations that are concurrent and conflicting. This provides the scalability of sharding as well as the precision of centralization. We describe refinable timestamps in detail in Chapter 3.

2.3.3 Discussion

A whole range of designs are possible for an ordering service, from entirely asynchronous ones to those that rely on tight clock synchronization. The remainder of this chapter describes a specific implementation of the centralized architecture called KRONOS. Chapter 3 describes the implementation of a hybrid technique that combines semi-synchronized vector clocks with asynchronous KRONOS, and we defer a deeper evaluation of alternate designs to future work.

2.4 KRONOS: A Graph-Based Ordering Service

Having discussed the event ordering abstractions and possible design space in Section 2.2 and Section 2.3, we now describe, in detail, a particular implementation of an ordering service, called KRONOS.

Internally, KRONOS builds and maintains an *event dependency graph*, a directed acyclic graph whose vertices correspond to events and whose edges correspond to *happens-before relationships*[1]. An edge therefore succinctly represents all the ordering related constraints between events spanning multiple applications.

Each of the methods described in Section 2.2 translates to an operation on the graph. When the application creates a new event, KRONOS creates a new vertex in the event dependency graph. Similarly, when the application establishes a

[1]We use the terms *dependency* and *happens-before relationship* synonymously throughout this chapter. The term *causal relationship* is related but more specific and not synonymous; a happens-before relationship can emerge without a causal relationship.

`create_event()`	Create a new event and return a unique identifier e.
`acquire_ref(`e`)`	Increment the reference count on e.
`release_ref(`e`)`	Decrement the reference count on e.
`query_order([(`e_1`, `e_2`), ...])`	Check the relationship between event pairs e_i e_j in specified list, returning $e_i \rightsquigarrow e_j$, $e_j \rightsquigarrow e_i$, or *concurrent* for each.
`assign_order([(`e_1`, `*order*`, `e_2`, `*must/prefer*`), ...])`	Create the set of relationships $e_i \rightsquigarrow e_j$ in specified list, if possible.

Table 2.1: The KRONOS API. Applications primarily use `query_order` and `assign_order` to establish dependencies.

happens-before relationship, KRONOS constructs a directed edge between the two vertices. To check for a pre-existing relationship between two events, KRONOS looks for a directed path between them. The direction of the path directly encodes the happens-before relationship. The absence of a path between two events indicates that they are concurrent.

2.4.1 KRONOS API

Applications interact with KRONOS through a simple API (Table 2.1) designed around the event and dependency abstractions. This API enables applications to manipulate, refine, and query the event timeline represented by the event dependency graph. KRONOS's API also permits atomic batching for efficiency, and conditional operations for additional application-level control.

Broadly speaking, the KRONOS API is split into event-oriented calls and traversal-oriented calls. The former allow applications to create and manage events and control the garbage collection mechanism, while the latter to help

discover precedence relationships between events of interest to the application.

2.4.1.1 Event Creation

Applications can add events to the KRONOS timeline with the `create_event` call, which creates a new vertex and returns a globally unique identifier. This identifier may be passed to subsequent calls to query the graph or establish happens-before relationships with the event.

2.4.1.2 Dependency Creation

The fundamental purpose of KRONOS is to enable applications to establish a time order for events. It does this by permitting applications to incrementally refine the timeline with new pairwise dependencies between events. KRONOS ensures that any refinement specified by the application is logically coherent, and maintains the abstraction's invariants; it does not permit the application to perform any refinement that violates them.

Dependencies may be created at any time during the lifespan of the event dependency graph. For instance, in our social network application, each time Alice and Bob interact with the service, KRONOS assigns the interaction a unique event identifier, and orders this event identifier with respect to other events that the application has previously created. These additional ordering constraints enable KRONOS to clarify the order of events in the timeline without withdrawing from any previously upheld guarantees. Consequently, events may be ordered by the application long after the interaction that precipitated the event's creation.

Applications may use the `assign_order` call to establish a dependency between a pair of events. On each call to `assign_order`, KRONOS maintains the coherency invariant by implicitly performing a graph traversal on the event pair. Any operations that request an order that contradicts the result of the traversal are aborted by KRONOS and the client is informed of the true order of operations.

To enable a wide array of application behaviors, the KRONOS API enables applications to express how to deal with requests that contradict a previously established order. KRONOS applications may specify two kinds ordering behavior: `must` and `prefer`. A `must` ordering conveys a hard constraint from the application that two events must be ordered in a specific way. Applications can use `must` constraints to store pre-existing relationships within KRONOS, such as relationships that arise from the natural execution of the system. For instance, when an application deletes an object, the delete is necessarily ordered after the preceding create. If a `must` request cannot be satisfied, KRONOS aborts the entire `assign_order` request without any side effects and returns an error to the application. In contrast, a `prefer` ordering preference indicates that the application would prefer that the events be ordered as specified in the request, but is willing to accept a reversal if previously established constraints make the request impossible. For example, applications typically prefer to respond to events in their arrival order, as long as doing so does not violate timing constraints. The application can use the `prefer` option to instruct KRONOS to maintain the arrival order where possible and reorder them when necessary. This permissive ordering is invaluable to applications that can reorder events, as it improves performance while maintaining correctness.

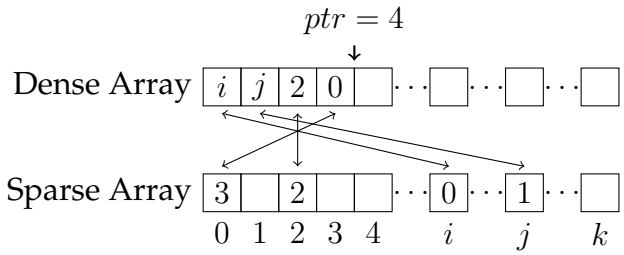


Figure 2.4: A diagram of the set data structure used to track visited vertices. A vertex i is in the set if and only if `sparse[i] < ptr && dense[sparse[i]] == i`. Adding an element to the set is done with `sparse[i] = ptr; dense[ptr++] = i;`. Clearing the set is done in constant time by setting `ptr = 0`.

For performance reasons, KRONOS does not attempt to discover the minimal set of `prefer` reversals to render a suggested `assign_order` request coherent with respect to the existing event dependency graph. Instead, KRONOS applies all `must` edges before `prefer` edges, thereby ensuring that a `prefer` edge is never established ahead of a `must` and thus will never cause an order assignment to abort when it could have been satisfied. Once all `must` edges are satisfied, the `prefer` edges are applied in the order specified by the application. An application can have some degree of control over which `prefer` edges are prioritized through the order in which they appear in the `assign_order` request. Not providing a guarantee of optimality avoids an NP-complete problem while providing a degree of control to the programmer.

KRONOS provides a powerful primitive reminiscent of test-and-set atomic instructions that enables applications to specify a mix of `must` and `prefer` operations that execute as one atomic batch. Clients may specify constraints to check with the `must` flag set. Should all of the constraints be met, the batch will be applied atomically, but if any constraint is not met, the batch will be aborted

without effect. A mixed batch of `must` and `prefer` operations resembles conditional test-and-set, where the `must` operations act as a conditional, and the entire batch will succeed or fail atomically. These atomicity guarantees enable safe yet concurrent use of the KRONOS service without requiring an external lock service [24, 61].

2.4.1.3 Graph Traversal

The `query_order` call enables applications to discover happens-before relationships captured by KRONOS. This call takes a pair of events, e_1 and e_2, and returns whether $e_1 \rightsquigarrow e_2$[2], $e_2 \rightsquigarrow e_1$, or they are concurrent. To do this, KRONOS performs a standard breadth-first search (BFS) to discover paths between e_1 and e_2.

The KRONOS implementation pays careful attention to the cost of creating new events and happens-before relationships. BFS is potentially a costly operation, whose latency can be $O(|V|)$ where $|V|$ is the number of events managed by the system. Since a naive BFS would either require $\Omega(|V|)$ operations to initialize a visited bit field in every vertex or else dynamically allocate memory, and since $|V|$ can be large, KRONOS instead uses a technique that makes use of uninitialized memory [21] to make the running time of BFS proportional to the number of vertices traversed. To avoid dynamic allocation, and linear initialization costs, KRONOS preallocates all memory required for graph traversal at the time of vertex creation by creating two arrays, `dense` and `sparse`, of size $|V|$. The sparse array corresponds to vertices, and maintains indices into the dense array, which, in turn, indexes back into the sparse array. Initially, `ptr` is

[2] $e_1 \rightsquigarrow e_2$ may be read as e_1 happens before e_2

set to 0. When BFS visits a node i for the first time, KRONOS sets `sparse[i]` to `ptr`, sets `dense[ptr]` to `i` and increments `ptr`. Checking to see if a node i has been visited can then be accomplished by checking if `sparse[i]` $<$ `ptr` and `dense[sparse[i]] == i`. This optimization enables the core traversal algorithm in KRONOS to require no memory allocation and only a single cache line worth of initialization.

2.4.2 Garbage Collection

The event dependency graph abstraction described so far will grow without bound as long as the distributed system is active. KRONOS employs garbage collection to enable clients to safely shrink the event dependency graph. A critical invariant that KRONOS maintains is that all events that could be submitted as arguments to any of the KRONOS API calls remain within the graph, since they can be used as starting points in traversal operations. KRONOS enables clients to dictate exactly which events can be used as arguments by exposing a reference counting API to clients.

KRONOS associates a reference count with each event and enables clients to acquire and release references through the `acquire_ref` and `release_ref` calls. Each time a client acquires a handle to an event, the reference count is incremented. Clients may at any time release the handle through a call to `release_ref`, which decrements the reference count. Once an event's reference count reaches zero, the event may be garbage collected. Overall, this reference counting mechanism ensures that all events that can be named by clients have non-zero reference counts and are pinned in memory.

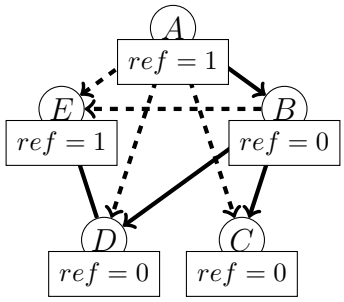


Figure 2.5: KRONOS uses reference counting to determine when it is safe to collect events. Because events are collected after their dependencies are collected, B, C, and D remain in the graph despite their 0 reference count.

To preserve transitive happens-before relationships, KRONOS does not garbage collect events until their dependencies are garbage collected. For example, Figure 2.5 shows an event dependency graph in which multiple events remain in memory despite having zero references. Event A pins events B, C, and D into memory, delaying their garbage collection until after `release_ref` is called on A.

Garbage collection is strict: each `release_ref` call performs a topological sort on the graph, removing vertices with zero references and their outgoing edges. Thus, a single `release_ref` call garbage collects a subset of all vertices with zero references. In our example above, this means that once A's reference count goes to 0, A, B, C, and D will be collected immediately. The acyclic property of the graph ensures that the operation will complete in bounded time, and that all vertices may be eventually collected.

The KRONOS API exposes no means of removing edges to applications because doing so would violate the monotonicity invariant. Edges are removed only after their source vertex is garbage collected. This ensures that edges per-

sist until they may no longer affect any traversal.

2.4.3 Fault Tolerance

KRONOS achieves fault tolerance by replicating the event dependency graph with state machine replication. Applications may treat KRONOS as a single, logically centralized service, and, due to state machine replication, the graph will be transparently maintained on several physical servers simultaneously. Because the KRONOS API is entirely deterministic, each API call directly corresponds to a state transition in the replicated state machine.

KRONOS replicates the event dependency graph using chain replication, which guarantees linearizability [123]. The exact number of KRONOS replicas in the chain is a deployment specific decision and should reflect the maximum number of simultaneous faults the system is likely to experience. A system looking to tolerate f faults deploys $f + 1$ replicas. In response to a replica failure, KRONOS requests reconfiguration of the chain via a coordination service [61, 24]. Both the normal case and failure case performance behavior follow from the standard chain replication protocol.

The functionality provided by chain replication is not fundamental to KRONOS's design and could easily be provided by other strongly consistent replication protocols. We use chain replication because the linear nature of the chain allows transactions to be pipelined at line rate without the fan-out/fan-in exhibited by Paxos-based techniques.

2.4.4 Scaling and Caching

The replicas necessary for fault tolerance provide a natural way to scale the system. KRONOS can perform traversals on potentially stale replicas for improved parallelism. Only traversals which indicate that events are concurrent must execute on an up-to-date copy of the graph. The monotonicity invariant upheld by KRONOS guarantees that any ordered answer returned by a stale replica is indistinguishable from the answer that would be returned had the query executed on the latest version of the graph.

Similarly, the monotonicity invariant permits widespread caching of traversal results without sacrificing correctness. KRONOS and applications are free to cache the results of traversals where doing so can improve performance. For example, KRONOS can maintain an internal cache of traversal results for high-degree vertices in order to improve traversal efficiency. Applications can freely pass around traversal results related to events within the messages used to commit the events.

2.5 Applications

In this section, we examine illustrative distributed applications to describe exactly how these systems use KRONOS in practice. To simplify exposition, we present these applications in their most simple form, omitting implementation details about caching and batching in favor of straightforward explanations of how they interact with KRONOS.

2.5.1 Social Network

Social networks are often built around the notion of providing users with a timeline of activity drawn from their social circles. A user's timeline captures both public posts and personal interactions between users, displaying social activity along the timeline. While much of the activity in a social network is generated independently, there are certain classes of interaction where the user expects ordering to be preserved. For instance, communication between users should be preserved within the timeline—the timeline should never show a reply earlier in the timeline than the message to which it is replying.

KRONOS provides a straightforward way to ensure that users' timelines reflect these communication patterns without enforcing a total order on all timeline activity. The social network may assign to each timeline post a KRONOS event identifier, and then record communication patterns in KRONOS with `assign_order`. When displaying user's timelines, the application can issue a corresponding `query_order` call to detect the partial order between events. Figure 2.6 shows pseudocode for this social network application.

2.5.2 Graph Store

Graph structured data is ubiquitous and analysis of these large graphs has prompted the development of specialized storage systems that directly store and maintain these graphs [22, 33, 51, 104, 81]. We have used KRONOS to build a horizontally scalable data store for graph-structured data called KRONOGRAPH. KRONOGRAPH is built around a sharded architecture where the graph data is partitioned across servers. The KRONOGRAPH API enables applications to

```
def post_message(user, message):
    e = kronos.create_event()
    for friend in friends_of(user):
        enqueue_in_timeline_for_user(timeline=friend,
                                     source=user,
                                     message=message,
                                     event=e)

def reply_to_message(user, message, in_reply_to):
    e = kronos.create_event()
    kronos.assign_order([(in_reply_to, '->', e, 'must')])
    for friend in friends_of(user):
        enqueue_in_timeline_for_user(timeline=friend,
                                     source=user,
                                     message=message,
                                     event=e)

def render_timeline(user):
    # messages is a list of (id, message) pairs
    messages = get_messages_enqueued_for(timeline=user)
    # message_pairs is every pair of message ids selected
    # from the messages
    message_pairs = all_pairs([m.id for m in messages])
    orderings = kronos.query_order(message_pairs)
    # This will perform a topological sort of the messages
    # to ensure that sorted_messages abides by the partial
    # orders specified within orderings.  The remaining
    # messages will be unaffected by the sort, enabling
    # them to be displayed in their arrival order
    sorted_messages = topological_sort(messages, orderings)
    return sorted_messages
```

Figure 2.6: Pseudocode for maintaining social network timelines with KRONOS. Users may post messages, which appear on timelines in the order in which the system processes them. When users use the social network's reply mechanism, the network uses KRONOS to order the messages. Users' timelines are rendered with respect to the order recorded within KRONOS, ensuring that conversations flow naturally.

incrementally build and maintain graph-structured data and perform isolated queries on the graph.

KRONOGRAPH permits updates and queries to the graph that span multiple hosts; consequently, KRONOGRAPH needs to apply operations in the same order across multiple hosts. In the absence of ordering, graph queries that are concurrent with updates could be applied in different orders at different hosts simply because the underlying messages used to transmit the operations arrive in a different order on each host. For example, imagine a graph consisting of edge $A - B$, where the application removes $A - B$ and adds $B - C$ as one update. An incorrect implementation could indicate that C is reachable from A, when, in fact, there was no instance in time when that was true.

The intuition behind KRONOGRAPH is that shard servers process updates and queries in their natural arrival order, except in cases where KRONOS indicates that the natural arrival order would not form a coherent timeline. To do this, KRONOGRAPH assigns to each update or query a unique KRONOS event identifier as it enters the system. Upon receipt of a new update or query operation, a shard server determines which vertices and edges are relevant to the operation, and gathers the event identifiers for all previous operations that affected these vertices and edges. The shard server then constructs a batch `assign_order` call to KRONOS that `prefers` that each of these previously-processed events be ordered prior to the current operation.

Given the information available, the preferred order specified within an `assign_order` call is the most efficient ordering for the events. Should this order be satisfiable, the shard server may perform the operation immediately, without reordering it with respect to previously applied operations. Sometimes,

the preferred order cannot be satisfied. For instance, if a pair of events arrive on two different shard servers in a different order, the first shard server's `assign_order` call will fix the order between these events. The second shard server's `assign_order` call must necessarily indicate a reversal to match the order returned in the first call. KRONOGRAPH shard servers can tolerate a reversed order that does not match their preferred ordering by reordering operations on the graph.

For updates, shard servers maintain version information for each vertex and edge in the graph to order the updates. Vertices and edges contain a list of modifications and their associated event identifiers, sorted by the relative order of events. When KRONOS upholds the ordering specified in the `assign_order` call, the shard server simply appends the update to the list. Should KRONOS indicate a reversal, the shard server inserts the update into its sorted position within the list. The coherency invariant prevents cycles in the order, ensuring that it is always possible to insert into the list and maintain its sorted order.

For queries, shard servers decide on their execution time using the information returned from the KRONOS assign order call. If the assign order call succeeds with no reversals, the KRONOGRAPH shard server should execute the query on the graph that contains all previous updates. When KRONOS indicates a reversal within the timeline, the shard server can construct an older version of the graph that omits all updates that happen after the query. Updates that are ordered strictly later than the query can easily be masked because of the timeline information maintained alongside the graph.

KRONOS ensures that the shard servers execute queries in matching order even as the queries traverse multiple shard servers. Every `assign_order` call

orders a query with respect to some subset of updates. KRONOS ensures that all shard servers order a given query the same way with respect to a given update; subsequent iterations of a query refine its place within the timeline by ordering it with respect to additional updates. Localized queries that traverse a small portion of the graph are ordered only with respect to updates on the same portion, and will likely remain concurrent with respect to updates occurring elsewhere in the graph.

While a straightforward implementation of KRONOGRAPH would query KRONOS once per vertex or edge during a query, these costs may be avoided with judicious use of batching and caching. Upon receipt of a query operation, the KRONOGRAPH shard server optimistically selects the events for vertices and edges in the graph could be traversed by the query operation, and requests that KRONOS order the query consistently with respect to these optimistically chosen events. This permits KRONOGRAPH to reduce the total number of calls to KRONOS, and enables queries to traverse larger portions of the graph between calls.

Internally, KRONOGRAPH relies upon caching to avoid unnecessary calls and to limit the size of each batched call. Each KRONOGRAPH server independently maintains an LRU cache of the pairwise order between events. Because of the monotonicity invariant, KRONOGRAPH servers may actively pre-fill this cache with transitive relationships. For example, if KRONOGRAPH queries KRONOS and sees that $u \rightsquigarrow v$, and the cache already contains $v \rightsquigarrow w$, the KRONOGRAPH server can infer that $u \rightsquigarrow w$ without another call to KRONOS.

2.5.3 Transactional Key-Value Store

Key-value stores have recently emerged as widely-used components in distributed services, mainly due to the high performance and scalability they offer. Existing key-value stores, however, achieve high performance by limiting their API; specifically, they restrict their clients to operate on a single object at a time. We have used KRONOS to build a transactional key-value store that provides ACID transactions, where each transaction may update multiple objects atomically and with full serializability.

Transactional key-value operations are inherently difficult because transactions may span multiple hosts. Without coordination, concurrently executing transactions would be processed in a different order on different hosts, violating serializability. One approach to adding this coordination would be to assign a total order across all transactions, where the total order ensures that transactions execute in the same order across all hosts. While such an approach would safely ensure serializability, it would do so at the expense of concurrency. Transactions which operate on disjoint sets of keys are able to execute concurrently, but the system would expend resources enforcing a total order across these keys.

The key insight in our prototype key-value store is to create a new KRONOS event for each transaction, and to order transactions that read or write the same keys using KRONOS. This enforces a partial order across all transactions using the event dependency graph, and ensures that transactions are serializable, without actually serializing them. Servers incrementally build the dependency graph by establishing an order between transactions within their purview. Upon receipt of a transaction, a server examines the keys within its partition, and issues an `assign_order` call specifying that the transaction must

be ordered after the last transaction which read or wrote each key. Should the `assign_order` call fail, the transaction will abort without effect.

Globally, the event dependency graph captures and enforces all dependencies between transactions. The system does not enforce any order between transactions not already ordered by the event dependency graph, as these transactions' individual operations may be applied in any order without violating serializability. Put another way, any topological sort of the event dependency graph will yield a schedule of transactions that is equivalent to the actual execution that produced the event dependency graph. This permits maximum flexibility between transactions, without requiring that they be applied in a total order.

2.6 Chapter Summary

This chapter proposed a new abstraction for tracking and managing dependencies between events in a distributed system. This abstraction opens the door for a new class of services in distributed systems, namely, event ordering services, which enable applications to explicitly manage and refine the possible timeline of events within the system. These new services provide a lingua franca for timeline management, enabling multiple independently developed components to form one integrated system that uses a common interface for time and event ordering. This approach facilitates the implementation of high-performance distributed systems that can provide strong guarantees by identifying potential cases of concurrency wherever possible. This chapter also described the design of KRONOGRAPH, a graph store built using KRONOS, a cen-

tralized event ordering service.

CHAPTER 3

SCALING TRANSACTION ORDERING WITH REFINABLE TIMESTAMPS

3.1 Introduction

In the previous chapter, we introduced a new service-oriented paradigm for event ordering in distributed systems that externalizes all ordering tasks to a separate service. While this results in many benefits, such as simplified application design, ordering compositionality across services, and opportunities for greater concurrency, there is one key drawback to the KRONOS design. Because it is a centralized service, there are ultimate limits to the scale that such a service can support.

In this chapter, we attempt to scale up the ordering service enabled by the KRONOS approach for ordering graph transactions without breaking application consistency guarantees. 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 Figure 3.1. 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.

Due to the unique nature of typical graph-structured data and queries,

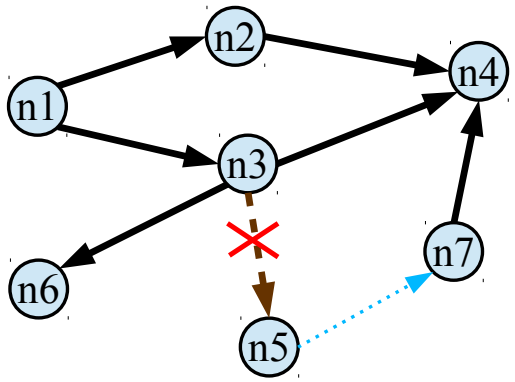


Figure 3.1: A graph undergoing an update which creates (n_5, n_7) and deletes (n_3, n_5) concurrently with a traversal starting at n_1. In absence of transactions, the query can return path (n_1, n_3, n_5, n_7) which never existed.

where queries are long-running analytics while transactions are shorter point queries (Section 1.2), existing databases have offered limited support. State-of-the-art transactional graph databases such as Neo4j [92] and Titan [33] employ heavyweight coordination techniques for transactions. Weakly consistent online graph databases [22, 28] forgo strong semantics for performance, which limits their scope to applications with loose consistency needs and requires complicated client logic. Offline graph processing systems [51, 47, 81, 105] 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.

We introduce WEAVER, 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 ordering service for ordering the re-

maining, potentially-conflicting reads and writes. This unique two-step ordering technique with proactive timestamping and a reactive ordering service 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 [87]. However many applications support a large number of concurrent clients and operate on graphs of such large scale, consisting of billions of vertices and edges [122, 68, 91], that a single-machine architecture is infeasible. For such high-value applications [22, 107] 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 centralized ordering service.

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 enables many features such as historic queries which run on past, consistent versions of the graph, dynamic resharding, and query caching.

This chapter makes three contributions. First, it details the overall architecture of WEAVER, which is key to enabling highly efficient distributed graph operations. Second, it describes the refinable timestamps mechanism for ordering distributed transactions. Third, it provides a theoretical proof that demonstrates that refinable timestamps enable a strictly serializable execution history of transactions in WEAVER. We defer discussion of WEAVER's many performance optimizations and graph data specific features to the next chapter.

3.2 WEAVER Abstractions

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 graph store.

3.2.1 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.

```
def social_network_post(user, edge_predicates):
    begin_transaction()
    photo          = create_node()
    ownership_edge = create_edge(user, photo)

    # attach metadata to the edge
    assign_property(own_edge, "OWNS")

    # iterate over edges at the user node
    # check predicates for each neighboring vertex
    # attach edge to enable visibility to the content
    for edge in user.edges:
        if edge.contains(edge_predicates):
            access_edge = create_edge(photo, edge.neighbor)
            assign_property(access_edge, "VISIBLE")

    commit_transaction()
```

Figure 3.2: A WEAVER transaction which posts a photo in a social network and makes it visible to a subset of the user's friends.

3.2.2 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. Figure 3.2 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.

```
def bfs(node, program_params):
    # return the list of next hop nodes for this program
    next_hop = []

    # visit node only if it has not been visited before
    # visited field is stored in program-local state
    if not node.program_state.visited:
        for edge in node.edges:
            # program parameters provid edge predicates
            # check if edge contains these predicates
            if edge.contains(program_params.edge_predicates):
                next_hop.append((edge.neighbor, program_params))
        node.program_state.visited = True

    return next_hop
```

Figure 3.3: A node program in WEAVER which executes a BFS query on the graph.

3.2.3 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 [49], node programs traverse the graph in an application-specific fashion, reading the vertices, edges, and associated attributes via the `node` argument. For example, Figure 3.3 describes a node program that executes BFS using only edges annotated with a specified edge property via the edge predicate check. 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 [81, 47, 51] to propagate queries to other vertices. In this approach, each vertex-level computation is passed query parameters

(program_params in Figure 3.3) 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 [18], search algorithms [106], and path discovery [107].

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.program_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 program_state at other vertices and propagate it between vertices using the program_params. This design enables applications that implement a wide array of graph algorithms. Node program state is garbage collected after the query terminates on all servers (Section 4.3).

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. In the next section, we describe WEAVER's timestamping mechanism.

3.3 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.

3.3.1 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 pro-

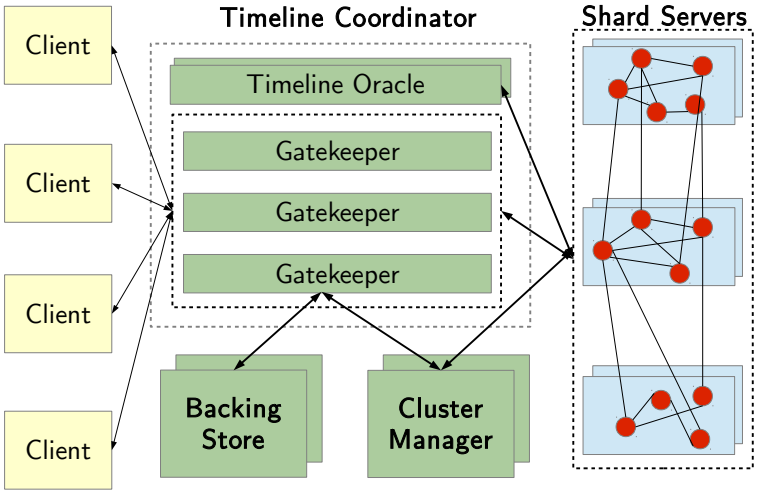


Figure 3.4: WEAVER system architecture.

cessing. 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.

3.3.2 System Architecture

WEAVER implements refinable timestamps using a timeline coordinator, a set of shard servers and a backing store. Figure 3.4 depicts the WEAVER system architecture.

3.3.2.1 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.

WEAVER shards the graph to enable the system to horizontally scale up with both data volume as well as query volume. As the graph grows larger, one may add more shards to the system to store the additional data. Additionally, the extra shard servers may also allow serving more queries.

3.3.2.2 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 [42] as the backing store.

3.3.2.3 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 *centralized ordering service*, KRONOS, for refining these timestamps when necessary (Section 3.3.3, Section 3.3.4). In addition to assigning timestamps to transactions, the gatekeepers also commit transactional updates to the backing store (Section 3.5).

3.3.2.4 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 (Section 4.6).

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 [45]. A vector clock consists of an array of counter values, one per gatekeeper, where each gatekeeper maintains a local counter

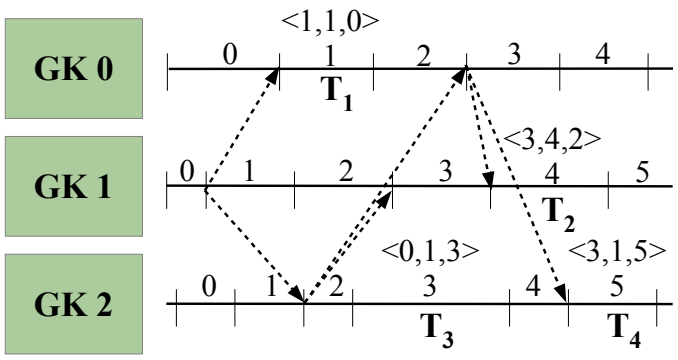


Figure 3.5: Refinable timestamps using three gatekeepers. Each gatekeeper increments its own counter for a transaction and periodically announces its counter to other gatekeepers (shown by dashed arrows). Vector timestamps are assigned locally based on announcements that a gatekeeper has collected from 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.

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 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 τ milliseconds. This proactive communication between gatekeepers establishes a happens-before partial order between refinable timestamps. Figure 3.5 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 $T_1\langle 1,1,0\rangle \prec T_2\langle 3,4,2\rangle$.[1]

[1] X $\prec$ Y denotes X happens before Y, while X $\preceq$ Y denotes either X happens before Y or X and Y occur at the same time.

Unfortunately, vector clocks are not sufficient to establish a total order. For instance, in Figure 3.5, 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.[2] Since transactions that enter the system simultaneously through multiple gatekeepers may receive concurrent vector clocks, WEAVER uses an auxiliary ordering service to put them into a serializable timeline.

3.3.4 Reactive Ordering by Centralized Ordering Service

WEAVER uses a centralized ordering service, such as KRONOS, to refine and keep track of happens-before relationships at a fine grain. The centralized ordering service 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. KRONOS ensures that transactions can be reconciled with a coherent timeline by guaranteeing that the graph remains acyclic (Chapter 2).

KRONOS's API is centered around events. In this case, events correspond to transactions in WEAVER. KRONOS enables primitives to create a new event to track a transaction, to atomically assign a happens-before relationship between sets of transactions, and to query the order between two or more transactions.

WEAVER's implementation of the ordering protocol comprises such an event-oriented API backed by an event dependency graph that keeps track of

[2] We denote this as $T_2 \approx T_4$.

transactions at a fine grain. The service is essentially a state machine that is chain replicated [124] for fault tolerance. Updates to the event dependency graph, caused by new transactions 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 6 million queries per second on a chain of 12 servers, each with 8 cores.

WEAVER uses this high-performance centralized ordering service 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 KRONOS. KRONOS 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 service for these transactions (Section 3.5.1, Section 3.5.2). 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.

3.4 Ordering Tradeoffs

The two-step ordering protocol in refinable timestamps combines loosely synchronized scalable clocks with a precise centralized ordering service. WEAVER's implementation of refinable timestamps uses vector clocks for timestamping purposes. However, in principle, any clock synchronization method which provides partial order may work, such as the use of real time in Google's Spanner database called TrueTime [31].

The TrueTime technique uses system clocks which correspond to real wall-clock time rather than logical counters. To synchronize the clocks, TrueTime uses specialized hardware such as atomic clocks and GPS clocks to enable high precision and low skew. Thus, for a successful TrueTime deployment, the datacenters need to be augmented with such specialized hardware. If such hardware is available, the TrueTime API returns a clock interval, such as $(\texttt{start\_time}, \texttt{end\_time})$, which captures the uncertainty in the clock synchronization.

In contrast, WEAVER's technique uses vector clocks. Each new transaction receives its own unique timestamp. One component of the timestamp, corresponding to the gatekeeper which received the transaction, is up to date. The other components of the vector may be slightly out of date depending on when the last synchronization occurred with the other gatekeepers.

Both refinable timestamps and TrueTime ensure a partial order. Both meth-

ods also have a similar overhead: TrueTime clocks need to be periodically updated by polling various high precision time references, and vector clocks are periodically synchronized using announce messages.

Inspite of similarities, WEAVER's implementation uses refinable timestamps due to reasons of simpler implementation, ease of deployment, and practicality. 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 KRONOS. At one extreme, one could use KRONOS for maintaining the global timeline for *all* requests, as in KRONOGRAPH, but then the throughput of the system would be bottlenecked by the throughput of the centralized ordering service. 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 centralized ordering service 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 KRONOS. This tradeoff ensures that the scalability limits of KRONOS are extended by adding gatekeeper servers. WEAVER's design provides a parameter τ—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 KRONOS. 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 τ in Section 5.3.

3.5 Implementation and Correctness of Transactional Ordering

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.

3.5.1 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 centralized ordering service.

When KRONOS 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, KRONOS will prefer arrival order. This order is then established as a commitment for all time; KRONOS 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 centralized ordering service 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 $T_1 \prec T_3$. This is because the dependency graph stored in KRONOS has edges for $T_1 \prec T_2$ as well as $T_2 \prec T_3$, thus the `query_order`(T_1, T_3) call will return T_1 happens before T_3.

Furthermore, because transactions are identified by their unique vector clocks, we modify KRONOS's implementation for WEAVER so that it is vector clock-aware. Essentially, KRONOS can infer and maintain implicit dependencies captured by the vector clocks. For example, if KRONOS first orders $\langle 0, 1\rangle \prec \langle 1, 0\rangle$ and subsequently a shard requests the order between $\langle 0, 1\rangle$ and $\langle 2, 0\rangle$, KRONOS 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.

To do this, we add a list of transactions in KRONOS per gatekeeper. Whenever KRONOS receives a new transaction, either as part of a `query_order` or an `assign_order` call, it first binary searches the new transaction in the corresponding list and locates its correct position. Subsequently, it adds happens before edges between the new transaction and its preceding and subsequent transactions in the list. For example, if the current state of the list corresponding to the first gatekeeper is $\langle 1, 0\rangle \prec \langle 2, 3\rangle \prec \langle 4, 3\rangle$, and KRONOS receives a new transaction with timestamp $\langle 3, 3\rangle$ which was timestamped by the first gatekeeper as well, KRONOS creates edges $\langle 2, 3\rangle \prec \langle 3, 3\rangle$ and $\langle 3, 3\rangle \prec 43$. This mechanism ensures that KRONOS respects both vector clock ordering as well as dependency graph ordering, at the cost of extra edges in the event dependency graph. To enable binary searching, the actual implementation is a array that is automatically dynamically resized, such as a `std::vector` in C++.

3.5.2 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.

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 [42] 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

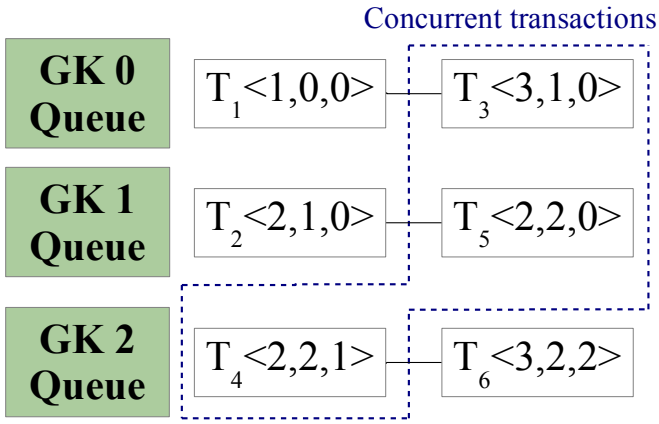


Figure 3.6: Each shard server maintains a queue of transactions per gatekeeper and executes the transaction with the lowest timestamp. When a group of transactions are concurrent (e.g. $T_3, T_4,$ and T_5), the shard server consults KRONOS to order them.

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 (Figure 3.6). 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 Figure 3.6, the shard servers will submit the set to KRONOS 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. Third, since ordering decisions made by KRONOS are irreversible and monotonic, shard servers can cache these decisions in order to reduce the number of ordering requests. Finally, KRONOS supports batched `query_order` requests, so a shard may create a batch of all transactions that are at the head of each shard queue and submit that to KRONOS. Thus, in our example of 3 concurrent transactions (T_3, T_4, T_5) in Figure 3.6, the shard server can discover the total ordering between the 3 transactions in a single request, and reuse the order in multiple iterations of the event loop.

Having decided which transaction to execute next, the shard effectively creates a new version of the graph without eliminating its current state. Effectively, the shards maintain an 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. Periodically, the outdated versions of the graph which are no longer needed for any query may be cleaned up using WEAVER's garbage collection protocol (Section 4.3).

3.5.3 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 KRONOS correctly maintains a DAG of events that ensures that no cycles can arise in the event dependency graph (Chapter 2).

Theorem 3.5.3.1. *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 KRONOS, 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 (Section 3.5.1). 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 (Section 3.5.1).

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 $\Gamma_x(T_i) < \Gamma_x(T_j)\ \forall x$. (Section 3.5.2). 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 (Section 3.5.2). 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 KRONOS. Thus we have:

$$T_j \prec T_i' \preceq T_i \tag{3.1}$$

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:

$$T_i \prec T_j' \preceq T_j \tag{3.2}$$

Equation 3.1 and Equation 3.2 yield a cycle in the dependency graph, which is not permitted by KRONOS.

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. □

Theorem 3.5.3.2. *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 $t_1 \approx t_1$, KRONOS will consistently order the two transactions across all shards. KRONOS will return a preexisting order if one exists, or order the node program after the transaction (Section 3.5.1). By always ordering node programs after transactions when no order exists already, KRONOS 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.

3.6 Chapter Summary

In this section, we described techniques that aimed at scaling up distributed transaction ordering. We introduced refinable timestamps, a new two-step ordering technique that performs lightweight partial ordering using vector timestamps, and resorts to precise total ordering based on a centralized ordering service on demand for a subset of the operations. We described the full transaction protocol and its implementation in WEAVER, a distributed strictly serializable graph database. We also detailed a proof of correctness that shows that WEAVER's ordering protocol results in a strictly serializable transaction execution history.

CHAPTER 4

PRACTICAL GRAPH DATA MANAGEMENT

The previous chapter detailed WEAVER's implementation of transaction and query ordering using refinable timestamps. In this chapter, we describe multiple features designed to improve usability and performance of storing and processing large graphs.

4.1 Bulk Data Load

While WEAVER's transaction implementation provides high throughput in many common cases, a pathological worst case scenario for the refinable timestamps protocol would be a write-heavy workload with high conflict rate. In this case, many write operations belonging to different transactions would conflict at the shards, and would require resolution from KRONOS, the centralized ordering service. This would, in turn, cause the system performance to be limited by the performance of KRONOS.

In our experience, the most common case where this arises in database management is during the initial data loading phase. The process of ingesting a large graph in WEAVER involves performing a high rate of update operations. Moreover, for vertices that have a high degree of relationships, the conflict rate on the update operations can be high. This represents a scenario with a high rate of conflict-heavy write operations.

Although it is possible to perform the initial data ingest using transactions, WEAVER's implementation includes a specialized bulk loading mode. In this mode, shards ingest graph data from files on disk at a high rate. However,

while shards are in the bulk loading mode, no node programs or transactions may execute in the system. In effect, the system enters in to a "write-only" state in order to fully optimize for the pending data ingest.

While in the bulk loading mode, each shard reads its partition of the graph from a file or multiple files from disk. The shard spawns bulk loading threads, and each thread reads chunks of the data from the file sequentially. After reading in a chunk from disk to memory, the thread processes that chunk by parsing the data format to extract vertices, edges, and associated data in the form of key-value pairs. To ensure that the threads can ingest the data with high-throughput and no conflicts, the data is hash partitioned amongst the threads. Thus, each thread is responsible for writing its own share of vertices, associated edges, and metadata, isolated from the write sets of other bulk loading threads, thereby guaranteeing no errors or conflicts.

Since the data is striped across the bulk loading threads, the data ingest process is vulnerable to stragglers due to unequal data distribution. To address the straggler problem, each shard ingests the data in chunks. Each bulk loading thread reads the next chunk from disk and then processes it to load its own share of data. Once a thread finishes its own work, it detects other straggling bulk load threads, and assumes responsibility for loading some of the straggler's share. This ensures that bulk loading is not delayed due to unequal hash partitioning of vertices to threads.

4.2 Dynamic Code Deployment

WEAVER's node programs are designed to enable users to execute arbitrary, often complicated graph algorithms with strong consistency on large graphs. Unlike the transaction API which permits multiple operations enclosed within a transactional block, the node program interface essentially permits issuing one node program query at a time. Of course, the query can be complex and may read a large portion of the graph as dictated by the user's code. However, this requires the node program code to be loaded on each shard prior to the client issuing the request.

Compiling the node program logic in to the shard server has the obvious drawback that users cannot innovate and deploy new node programs without suffering service downtime. In such a scenario, when a user wants to issue a new node program, it would first kill all database processes in the entire cluster, then recompile the shard process with the new application logic, and then redeploy the cluster. While WEAVER's fault tolerance scheme ensures that such a process would correctly recover all data from disk, the inevitable service downtime is a nuisance at best, and unacceptable for production systems with high availability requirements.

For such scenarios, WEAVER's implementation includes dynamic node program code deployment. The user submits their new node program as a shared object file. The shared object file is loaded by the shards, without service interruption, using the `dlopen` syscall. Once a shared object is loaded in to memory with `dlopen`, WEAVER can locate the necessary node program functionality from the shared object using the `dlsym` which enables locating the address of a

symbol in the shared object. A naming convention enables symbols corresponding to constructors, destructors, serialization methods, and node program logic to be located in the user submitted shard object. After all shards load the node program functionality dynamically, clients can issue new requests. This technique guarantees dynamic code deployment without service downtime.

4.3 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 former case, WEAVER retains all versions of the graph in memory and on disk. This permits users to issue node program queries with a timestamp in the past. When a shard receives such a node program query at a vertex, it compares the timestamp of the node program to all versions of the vertex and executes the query on the one (and only) version of vertex that corresponds to the node program timestamp. Such a garbage collection policy, or lack thereof, enables an eidetic graph database.

In the latter case, WEAVER employs a simple algorithm for distributed garbage collection. Each gatekeeper keeps track of outstanding node programs which have begun execution and not yet completed. The gatekeepers periodically communicate the timestamp of the earliest outstanding node program to each shard. The shards take the minimum of these timestamps to compute T_e,

the earliest possible node program that is still executing across the system. This enables the shard servers to permanently delete all state with delete timestamp earlier than T_e. WEAVER uses the same garbage collection algorithm for cleaning up events in the dependency graph of KRONOS. This is because future transactions will have a timestamp strictly greater than T_e and hence cannot conflict with the expired events.

4.4 Graph Partitioning

The dynamic nature of the graphs that WEAVER supports presents opportunities and challenges for the system. In this section, we describe how WEAVER rebalances load in response to changes to the underlying graph.

Graph partitioning is a very important factor for performance in a distributed graph store. If there are lots of edges that span across shards, then query execution would entail sending many messages between servers. Graph partitioning quality is directly proportional to network overhead, which affects the overall efficiency of the system. In general, minimizing cross-partition edges is a desirable property for sharded graphs.

State of the art graph partitioning techniques [63] require a centralized view of the graph, wherein the entire adjacency list or matrix is available on a single machine. Since such a view is not available in a distributed architecture such as WEAVER, new techniques which partition the graph in a distributed fashion are necessary.

Common graph applications often exhibit locality in their data access pat-

terns; that is, it is likely that a query which reads a particular vertex also reads the neighbors of that vertex. WEAVER leverages this by migrating nodes across shards based on changes in the graph structure. The key insight behind migration of graph vertices across shard servers is that colocating a vertex with the majority of its neighbors can result in lower communication overhead during query processing.

To facilitate dynamic migration of graph data across shards, WEAVER implements a generic graph repartitioning scheme which streams through the vertices at each shard and decides whether to migrate the vertex to another shard based on the vertex's data. The repartitioning happens independent of the timeline coordinator and permits concurrent updates to the graph. However, since the repartitioning process is completely asynchronous, care must be taken to avoid following stale edges to vertices that have recently migrated. WEAVER maintains its consistency guarantees throughout the repartitioning process by retaining a pointer at the source shard to the new location of the migrated vertex and eventually garbage collecting the pointer when the changes due to the migration is reflected in all graph store state.

The choice of migration policy is orthogonal to the rest of the system design, and WEAVER can accommodate a large class of graph partitioning algorithms which use vertex-local data. However, any practical graph partitioning heuristic needs certain guiding principles. The heuristic should take into account the structure of the graph and attempt to colocate as many neighbor vertices as possible in order to minimize the number of edges cut. An equally desirable characteristic is that the repartitioning heuristic should be lightweight. In order to handle dynamic graphs, it should be possible to run the heuristic repeatedly

without pausing graph queries and make vertex migration decisions locally at each shard.

WEAVER uses the restreaming variant [95] of the Linear Deterministic Greedy graph partitioning heuristic due to Stanton and Kliot [112]. This heuristic streams through the vertex list and, for each vertex v, attempts to relocate v to the shard which houses the majority of its neighbors, subject to server capacity constraints.

4.5 Caching in a Dynamic Graph

In addition to locality in access patterns, graph analyses can benefit from caching analysis results at vertices. Many typical traversal-oriented graph queries explore a large portion of the graph. When such traversals cross shard boundaries, they can be expensive to compute. For example, a reachability query which attempts to discover if a vertex V_n is reachable from a vertex V_1 would perform a breadth-first search starting at V_1 until it reaches V_n or explores the entire connected component. If V_n is reachable, the entire $(V_1, \ldots, V_n)$ path can be cached at each vertex V_i. Subsequent queries which attempt to reach V_j starting at vertex V_i where $i < j$ can avoid performing the costly traversal again by simply reusing the cached path. For typical social networks where V_n is a popular vertex, this optimization can save significant network traffic.

However, it is difficult to support caching in a scenario where the system guarantees strong consistency, since query results may have complicated dependencies, as graph-structured data often does. For instance, it is non-trivial for a graph store to cache the path returned by a reachability query, because

subsequent modifications to any vertex on the entire path may invalidate the cached value.

WEAVER provides a caching interface which permits applications to memoize the results of node programs along with the vertex data at the shard server. WEAVER adopts an application-driven invalidation technique which can match the semantics of the cached data. Applications built on top of WEAVER specify the watch set, i.e. the graph data that decides the validity of the cached value. When subsequent queries execute at the shard servers and find a cached value, the queries also return all changes made to the watch set along with the query results to the client. Clients may decide whether the value is still valid or needs to be purged from the cache. In our example of reachability queries on a social network, the cached value would simply be the vertex handle of the destination vertex and the watch set for a cached reachability query would be the entire cached path.

4.6 Fault Tolerance

WEAVER has a minimal approach to fault tolerance. To withstand f failures of gatekeepers or shard servers, there need to f backup servers in the datacenter. The backup servers register with the cluster manager but do not actively replicate state from gatekeepers, shards, or the backing store. This approach minimizes the common case overhead of replicating each update at gatekeeper or shard to a backup server, and it also reduces the amount of data that is persistently stored in the backing store.

In response to a gatekeeper or shard failure, the cluster manages spawns a

new process on one of the live backup servers. The new process restores corresponding graph data from the backing store and recreates the partition of the graph. 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.

Next, 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 remaining components of the architecture include the backing store. KRONOS, and the cluster manager. We treat the backing store as a fault tolerant black box—any off the shelf, state of the art, production-ready database suffices. Our implementation uses HyperDex Warp [42], which ensures fault

tolerant and durable key-value operations. KRONOS is implemented as a fault tolerant replicated state machine as described in Chapter 2. Finally, the cluster manager may also be implemented as a fault tolerant replicated state machine, similar to KRONOS, or we may employ an off-the-shelf configuration service such as Apache Zookeeper [61, 46]. WEAVER implements its own replicated cluster manager.

4.7 Demand Paging

WEAVER stores a sharded multi-version graph which is partitioned according to the scheme described in Section 4.4. The graph data is stored in in-memory at the shard servers, and a durable copy resides in the backing store (Section 3.3.2). While WEAVER attempts to store as much of the graph data in memory as possible, the overall data size may grow beyond the cumulative available memory across all shards.

If there is more data than memory available, WEAVER implements a form of demand paging. Essentially, WEAVER implements a least recently used algorithm at each shard to keep those nodes which have been used most recently in memory. WEAVER uses the clock algorithm commonly used to implement page replacement in operating systems as an approximation to the least recently used algorithm [8]. WEAVER stores a circular list data structure with pointers to graph vertices in memory. Each time a vertex is accessed that is already in memory, the corresponding entry in the circular list is marked as "used". When a "vertex fault" occurs, that is, a query attempts to access a vertex that does not reside in memory, WEAVER first chooses a vertex to evict. A pointer, called the *clock hand*,

sweeps through the circular list and resets the used field of the vertex pointers until it finds an unused vertex. WEAVER subsequently continues and evicts as many vertices from memory as required in order to page in the new vertex for the current query. Our implementation assumes that each vertex and associated data, including its adjacency list and key-value properties, can fit entirely in memory at any shard.

4.8 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.

4.8.1 Social Network

We implement a database backend for a social network, based on the Facebook TAO API [22], 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 [22]. We evaluate the performance of this social network backend against a similar one implemented on Titan [33], a popular open-source graph database, in Section 5.3.2.

4.8.2 CoinGraph

Bitcoin [91] 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 ~ 900 GB of annotated data in WEAVER.

4.8.3 RoboBrain

RoboBrain [107] 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 [128].

4.8.4 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 [79] 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.

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 [76], a well-understood API will reduce the number of such bugs. Finally, WEAVER is scalable and can support high throughput of transactions (Section 5.3), rendering weaker consistency models unnecessary.

4.9 Chapter Summary

In this chapter, we provided various implementation details that comprise the WEAVER graph store. The key enabler for many of WEAVER's features is the multi-version graph that underpins the storage layer—this permits historic queries, consistent yet dynamic data resharding, application-driven query caching, as well as dynamic code deployment. We also discussed many practical applications that are made possible by WEAVER, including a knowledge graph and a cryptocurrency blockchain explorer.

CHAPTER 5

EVALUATION

In this chapter, we perform a full evaluation of the various techniques described in this dissertation. We first evaluate the performance of KRONOS, a centralized instantiation of the transaction ordering technique used in our graph store (Chapter 2). KRONOS's evaluation includes both micro-benchmarks that measure the scalability and performance of individual API calls, as well as end to end benchmarks of applications built on top on KRONOS including KRONOGRAPH and a transactional key-value store.

Next, we evaluate the refinable timestamps protocol for ordering transactions in a graph store (Chapter 3). Since the purpose of designing refinable timestamps is to improve scalability of the centralized ordering service, we focus specifically on scalability of the ordering protocol. We also experimentally measure the effect of τ, the clock synchronization period, on the tradeoff between proactive and reactive ordering in refinable timestamps.

Finally, we perform end to end evaluations of a full implementation of WEAVER (Chapter 4). We compare WEAVER's performance to a variety of state of the art graph storage and processing systems on applications such as a Bitcoin blockchain explorer, a social network backend, and graph analytics workloads. We also measure the impact of WEAVER's performance optimizations, specifi-

cally the dynamic repartitioning and query caching features.

5.1 Centralized Ordering

We have fully implemented KRONOS to provide the functionality detailed in Section 2.4, and have built multiple applications on top of it. In the first half of our evaluation, we examine the performance of our sample applications to demonstrate that it is feasible to build real-world applications using KRONOS. In the second half of our evaluation, we use micro-benchmarks to investigate important aspects of KRONOS's design, paying careful attention to performance, scalability, and resource usage. We finish our evaluation with a brief demonstration of KRONOS's fault-tolerance.

Our experimental setup consists of fourteen well-provisioned servers. Each server is equipped with two Intel Xeon 2.5 GHz E5420 quad-core processors and 16 GB of RAM. All servers are running 64-bit Debian 7 and are connected via gigabit Ethernet.

5.1.1 Applications

In this section we attempt to answer the question, "Is it practical to build applications on KRONOS?" The performance of the resulting applications is important in evaluating whether KRONOS is a suitable choice for each application, but should not be the only deciding factor. For small- to medium-sized applications, the composition property provided by KRONOS may be worth any overhead that affects performance.

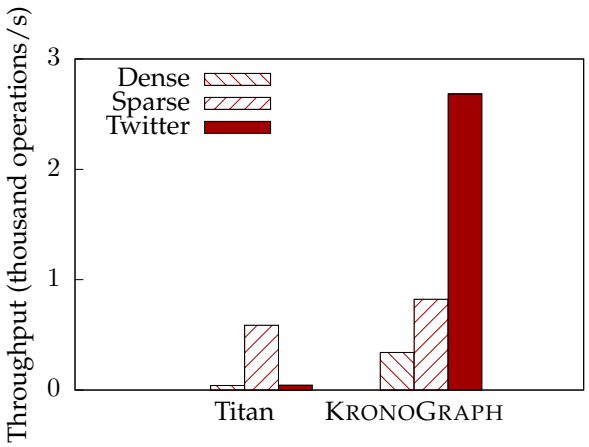


Figure 5.1: Titan and KronoGraph performing friend recommendation calculations on a mutating graph in a 95% read/5% write workload. KronoGraph outperforms Titan by more than 50× for the Twitter social network. Kronos enables KronoGraph to perform queries that are fully isolated from the ongoing write operations, while Titan uses locking to make the same guarantee.

For all of the application-specific benchmarks, we deployed a single instance of Kronos on its own server, to ensure that the cost of interacting with Kronos includes all relevant communication cost. The remaining servers in the cluster deploy the application itself. We evaluate fault tolerance overheads separately.

5.1.1.1 Graph Store

We first evaluate KronoGraph, our graph store built on top of Kronos. For an accurate comparison, we compare KronoGraph to Titan [33], another online graph store that permits users to query and incrementally alter the graph. Titan employs lock-based techniques to provide isolation guarantees comparable to KronoGraph. We omit comparisons to other notable graph systems [51, 104, 81] because they do not support online operation and are thus

incomparable to Titan and KRONOGRAPH.

Intuitively, queries and updates in KRONOGRAPH should be strictly less expensive than in Titan because Titan's lock-based techniques inhibit concurrency, while KRONOGRAPH exploits late time binding in KRONOS to allow non-blocking behavior. Titan's locks decide the order of graph operations; the first process to grab a lock is implicitly ordered earlier than later lock-holders. KRONOGRAPH explicitly manages the order of graph operations, and consequently can perform multiple operations simultaneously, and resolve their order in one call to KRONOS.

To characterize the difference in behavior between Titan and KRONOGRAPH, we implemented a friend recommendation application in a social network on top of both systems. Our application represents the social network as a graph where individuals are represented by vertices, and edges symbolize friendship. The application makes friend recommendations on the basis of maximizing mutual friendship. For a given input, the algorithm will return the user with the most number of friends in common. This mimics the behavior of many social networks, where the structure of the graph is used to make further recommendations to users [121].

We ran both of our friend recommendation algorithms on a subset of the Twitter social network [84]. This graph consists of 81,306 individuals with 1,768,149 friendship links. For both implementations, we ran 32 parallel clients with a workload generator that produced a mixed workload that performed a friend recommendation 95% of the time, and introduced new individuals or friendships to the graph the remaining 5% of the time. We can see in Figure 5.1 that the KRONOGRAPH friend recommendation algorithm outperforms the Ti-

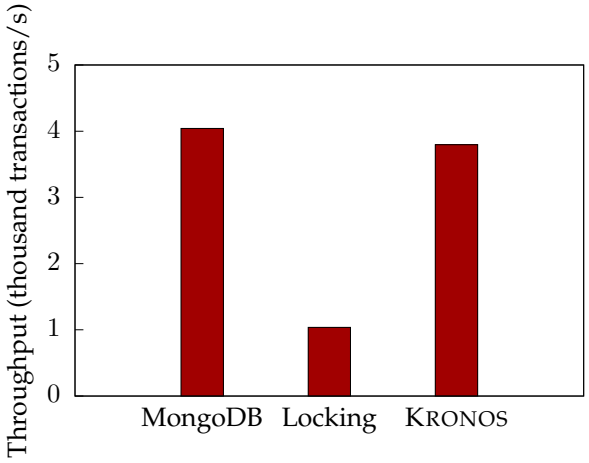


Figure 5.2: Transactional chains are fully three times faster than locking-based implementations and achieve 94% of the throughput of a "put-and-pray" approach built on MongoDB. This graph shows a sample banking application performing transfers between accounts.

tan recommendation algorithm by more than $50\times$.

The performance gap between KRONOGRAPH and Titan is largely related to the density of the graph. We generated two random graphs of varying density to use as inputs to our friend recommendation algorithm to confirm this hypothesis. The denser of the two graphs had an average degree of 100, while the sparser graph had an average degree of 10. We can see in Figure 5.1 that KRONOGRAPH outperforms Titan by a factor of $8.3\times$ and $1.4\times$ respectively.

The variation in KRONOGRAPH's performance across the three different graphs gives us deeper insight into the performance characteristics of the system beyond raw differences in throughput. Because the number of calls made to KRONOS is related to the number of operations submitted by KRONOS clients, we would expect that a bottleneck around KRONOS would limit the throughput and restrict it from varying with the density of the graph. Batching and caching in KRONOGRAPH are effective, and prevent KRONOS from becoming a bottle-

neck. In our Twitter experiment, approximately 13.4% of operations required a KRONOS traversal.

5.1.1.2 Transactions

In this section, we describe the evaluation of a transactional key-value store, that provides ACID semantics, built using KRONOS. To evaluate this application, we developed a prototypical banking application, similar to the one used in nearly every database textbook to illustrate transactions [39]. Our application processes users' debits and credits, and transfers money between bank accounts.

For comparison, we implemented the banking application on top of two other data stores for a total of three comparable bank applications. Our first bank application is built on the popular MongoDB NoSQL data store, where account transfer consists of two independent write operations to MongoDB. Because MongoDB does not offer transactional semantics—it is only eventually consistent—this application is likely to encounter undesirable behavior, such as incomplete money transfers and lost deposits. Our second bank application uses locking techniques, such as those used in Percolator [97], to synchronize access to individual accounts and provide fully-serializable semantics. Finally, we implemented transactional semantics using KRONOS as described in Section 2.5.3. We used HyperDex [41] as the underlying key-value store in the second and third implementations.

Figure 5.2 shows the throughput each implementation was able to achieve when accessed by 64 concurrent clients. We see that the KRONOS-based variant

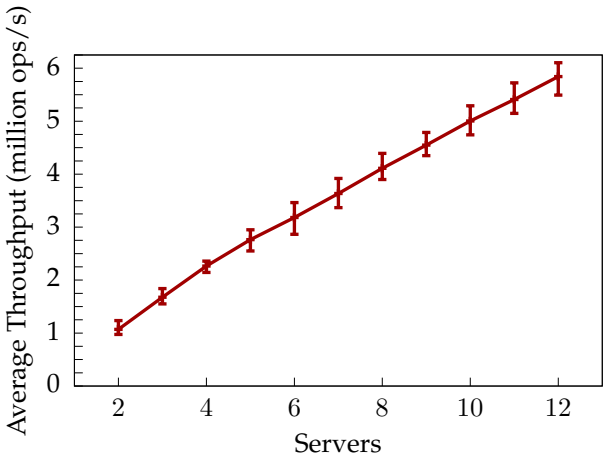


Figure 5.3: KRONOS is a scalable system. This graph shows the aggregate throughput achieved by a fixed number of clients calling `query_order` on a graph where each edge participates in, on average, 5 happens-before relationships. Aggregate throughput is measured across a 30 second window and the tight error bars show the 5^{th} and 95^{th} percentiles for throughput observed throughout the window.

outperforms the lock-based variant by a factor of 3×. The KRONOS-based transactional key-value store achieves 94% the throughput of the non-transactional, eventually-consistent MongoDB deployment. This comparison provides an advantage to MongoDB, as MongoDB provides relatively weak guarantees, while the KRONOS-based transactional key-value store provides ACID transactions.

5.1.2 Micro-Benchmarks

In order to further explore the design decisions made in KRONOS, we present several micro-benchmarks each of which explores a different aspect of KRONOS's design. KRONOS provides tools for explicit event creation and ordering. We first examine the performance and scalability of order-related API calls as they are by far the most costly aspect of KRONOS. We then investigate the time

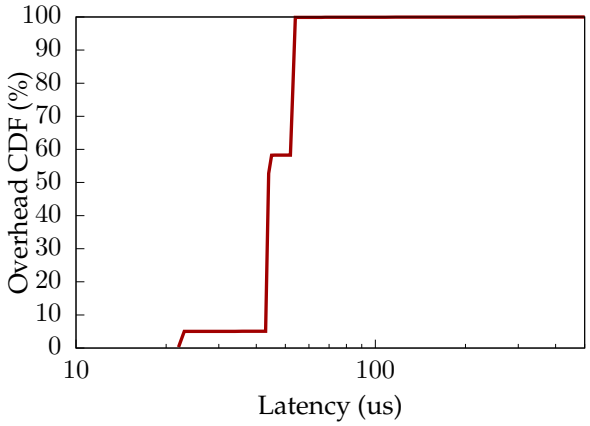


Figure 5.4: KRONOS quickly creates events. KRONOS can create a new event in less than $57\mu s$ 99% of the time.

and resource costs associated with event creation, dependency creation, and event garbage collection. Except where noted, these results do not include the overhead of state machine replication, as it is largely separable from KRONOS's implementation.

5.1.2.1 Scalability

Our first experiment measures how additional servers enable KRONOS to handle additional `query_order` requests. In this experiment, we pre-loaded KRONOS with a random graph over 10,000 vertices with 50,000 edges, and varied the number of replicas used for satisfying `query_order` requests. Each client performs random `query_order` requests on the graph, checking for preexisting relationships. We deployed 64 clients that concurrently query the replicas of the graph using the `query_order` API. Figure 5.3 shows that KRONOS scales well; each additional server enables the system to respond to proportionally more `query_server` requests.

5.1.2.2 Dependency Creation

When assigning order between two events, the dominating cost is graph traversal. Once KRONOS traverses the graph, the cost of actually recording the dependency is nearly trivial. We measured the time taken to create dependencies that require no traversal, and found that, across 1 million events, dependency creation completes in $49\mu s$ 14.7% of the time, and $50\mu s$ the remaining 85.3% of the time. These numbers also reflect the additional cost of creating events above and beyond the cost of a `query_order` operation.

5.1.2.3 Event Creation

The next experiment examines the overhead of event creation and shows that KRONOS creates events in constant time. Figure 5.4 shows a CDF of event creation latency for 100 million events. KRONOS completes a majority of `event_create` operations in $44\mu s$ and 99% of operations in less than $57\mu s$ These measurements include all allocation necessary to create the new event. For this experiment, the client uses the KRONOS Python bindings to create and acquire references to the events. The event creation latency was measured by timing 10,000 sequential calls to `event_create`, with no parallelism in the calls. To avoid confounding effects relating to network latency and to isolate the performance of the server itself, the client and server are co-located on the same machine.

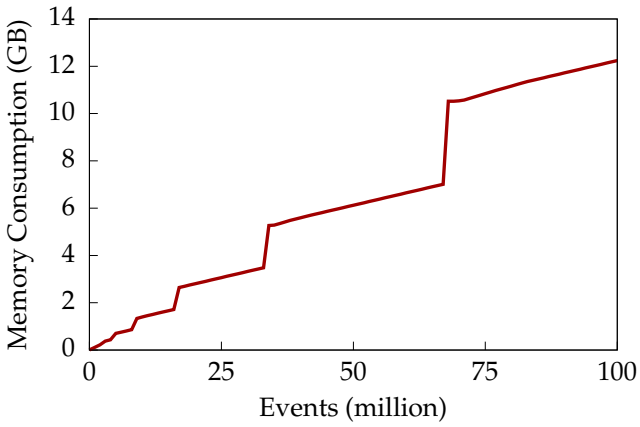


Figure 5.5: KRONOS's memory consumption scales linearly as events are added. In this graph, a single client adds a total of 100 million events sequentially, maintaining a reference to each one. The memory usage is the maximum resident set size of the process. Discontinuities in the graph are directly related to array-doubling in the implementation.

5.1.2.4 Memory Consumption

Because KRONOS allocates all memory used by a vertex at event creation time, it is important to quantify this cost. Figure 5.5 shows that 100 million events occupy 12 GB of RAM and fit within main memory of a single server. The reported memory consumption includes all memory necessary to track unique event identifiers, perform traversal using the BFS algorithm and maintain one reference per event. Applications will only allocate more memory when adding edges, where each edge occupies 8 byte of space. The implementation dynamically allocates memory to grow and shrink while remaining proportional to the number of events and dependencies in the system.

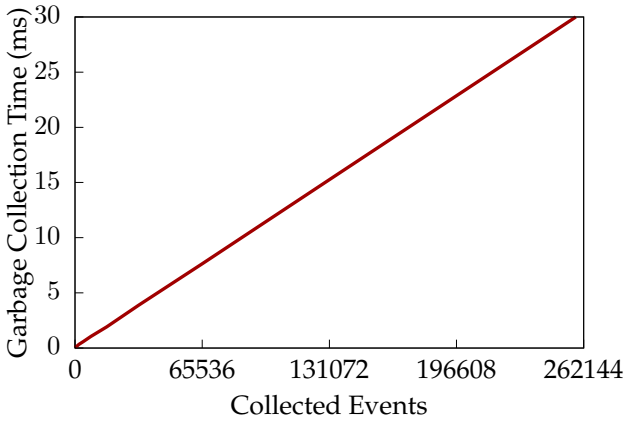


Figure 5.6: Garbage collection is efficient even for the absolute worst case event dependency graph. In this experiment, fixed length paths are created in the dependency graph such that releasing a reference to the first event in the path garbage collects the entire path.

5.1.2.5 Garbage Collection

KRONOS's strict garbage collection scheme introduces minimal overhead. Because KRONOS uses strict garbage collection, the cost of releasing the final reference to a single event is proportional to the total number of events collected. Figure 5.6 shows worst case garbage collection behavior of KRONOS. For this experiment we control the number of events to be garbage collected by a single `release_ref` call. As expected, the time taken to perform strict garbage collection grows linearly in the number of events to be collected.

5.1.2.6 Impact of Graph Structure

The cost of graph traversal is dependent upon the structure of the graph itself. Intuitively, sparse graphs are quicker to traverse as the likelihood of touching many vertices becomes lower as the graph becomes sparser. On the other hand,

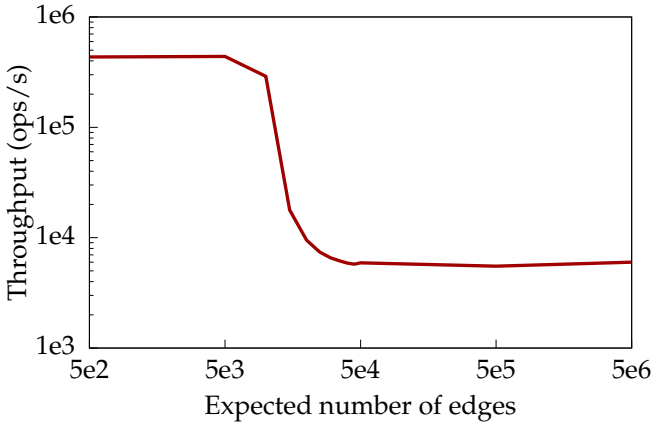


Figure 5.7: KRONOS is fast for sparse graphs. This graph shows the aggregate throughput of `query_order` operations on Erdös-Rényi graphs with 10,000 vertices and varying numbers of edges.

processing dense graphs will necessarily involve a longer traversal as more vertices belong to large connected components. To test the behavior of KRONOS on a variety of sparse and dense graphs, we generated random event dependency graphs conforming to the Erdös-Rényi Model [40]. Under this model, any two points in the graph are connected with probability p. Accordingly, these graphs have between 500 (p = 0.00001) and 5,000,000 ($p = 0.1$) edges, with larger values of p corresponding to denser graphs. Figure 5.7 demonstrates the impact of graph density on the throughput of `query_order` operations on a single instance of KRONOS. For relatively sparse graphs where each vertex belongs to, on average, less than 3 happens-before relationships, KRONOS can perform hundreds of thousands of queries per second. As the density of the graph increases, KRONOS's throughput approaches a stable point where additional edges do not alter throughput. The majority of applications will likely resemble sparse dependency graphs as most applications do not need to impose a total order across all events, but instead order small groups of events together.

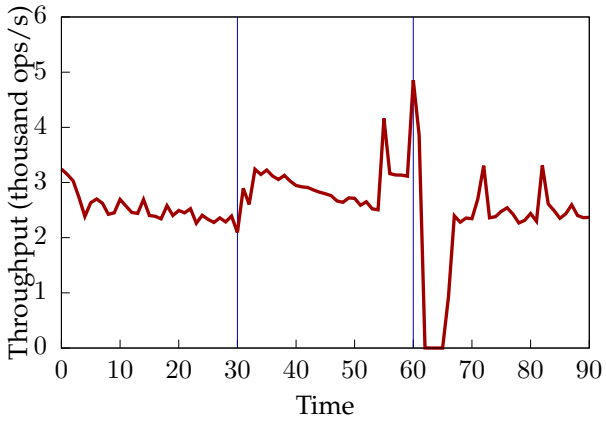


Figure 5.8: KRONOS automatically recovers from failures. This graph shows the effects of server failure in a 3-server KRONOS deployment. At the 30 second mark, the middle server in the chain is killed. Another server is brought into the cluster to take its place at the 60 second mark.

5.1.3 Fault Tolerance

KRONOS uses chain replication to provide fault tolerance. As a test of its fault tolerance capabilities, we examined the performance of a 2-fault tolerant KRONOS cluster. The underlying chain replication algorithm automatically removes failed servers from the cluster and can integrate new servers transparently. Figure 5.8 shows the throughput of the 2-fault tolerant cluster as a server fails and is re-added to the system. At the 30 second mark, the middle server in the chain is killed. The system recovers quickly and stays available for further operation. At the 60 second mark, a new server is introduced at the tail of the chain and begins the healing process to restore the service to being 2-fault tolerant. Overall, KRONOS remains available and provides high throughput when servers are removed or re-added. This graph includes all the overhead of our unoptimized state machine replication implementation.

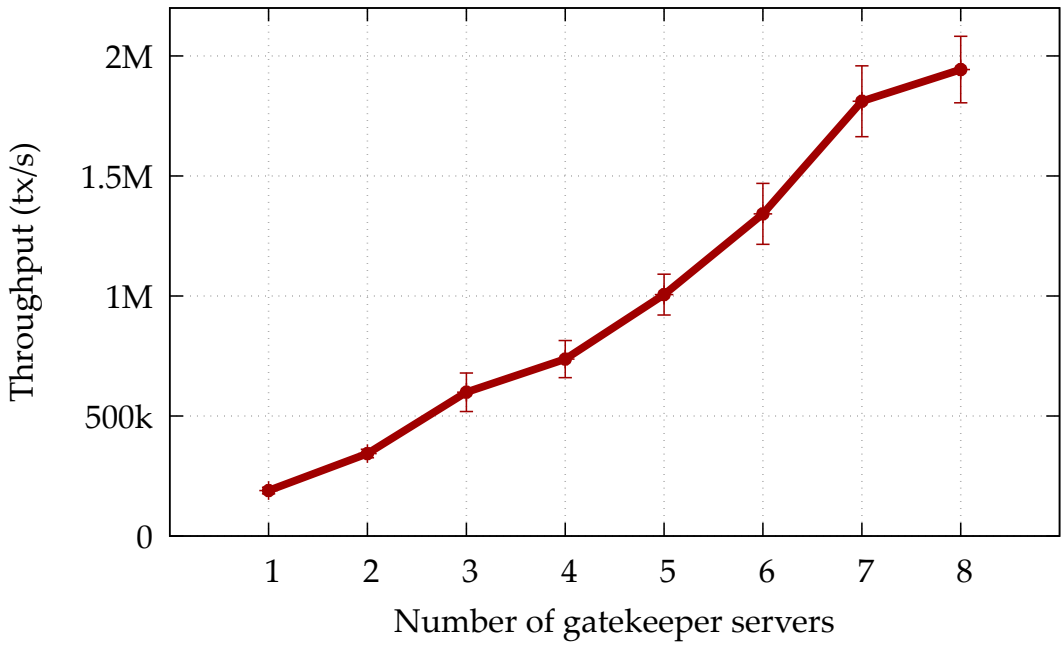


Figure 5.9: Throughput of `get_node` programs. WEAVER scales linearly with the number of gatekeeper servers.

5.2 Refinable Timestamps

In this section, we perform micro-benchmarks to measure the scalability of refinable timestamps, as well as the tradeoff between proactive lightweight timestamping and reactive precise ordering.

5.2.1 Scalability

To investigate how WEAVER's implementation of refinable timestamps scales, we measure WEAVER's throughput on micro-benchmarks with varying number of servers. We perform the first set experiments on a cluster of 28 machines each of which has two 4 core Intel Xeon 2.5 Ghz L5420 processors, 16 GB of DDR2

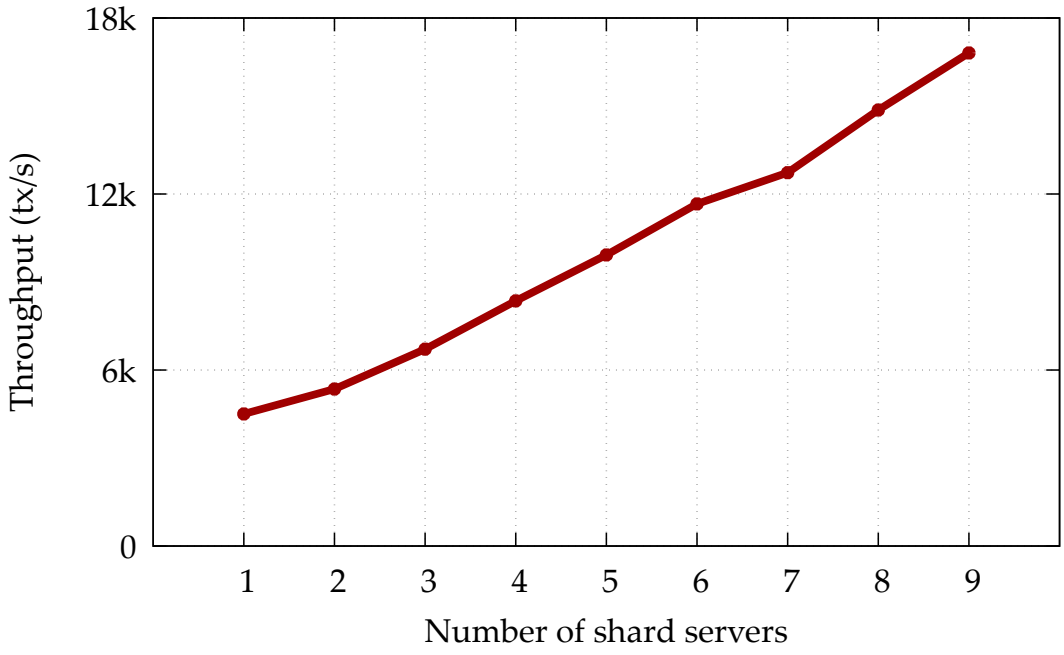


Figure 5.10: Throughput of local clustering coefficient program. WEAVER scales linearly with the number of shard servers.

memory, and between 500 GB and 1 TB SATA spinning disks from the same era as the CPUs. We perform this experiment on the Twitter 2009 snapshot [68] comprising 41.7M users and 1.47B links (24.37 GB).

Figure 5.9 shows the throughput of `get_node` node programs in WEAVER with a varying number of gatekeeper servers. For this experiment, we keep the number of shard servers fixed. 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 nearly 2 million transactions per second with just 8 gatekeepers. This is because with each additional gatekeeper, the system overall has more timestamping cycles.

However, as the complexity of the queries increases, the shard servers perform more work compared to the gatekeeper. The second scalability micro-

benchmark, performed on a small Twitter graph with 1.76M edges (43 MB) [85] using a cluster comprised of 14 machines similar to those in the previous experiment, measures the performance of the system on local clustering coefficient node programs. The local clustering coefficient of a vertex measures the degree to which the neighborhood of a vertex is tightly knit. It is defined as the number of edges between all vertices in the neighborhood of a vertex, divided by the total number of edges possible if the neighborhood was a clique. More formally, if the set of vertices that are neighbors of a vertex v is $N(v)$, then the local clustering coefficient, LCC, is defined as

$$LCC(v) = \frac{|(e_i, e_j) \ : \ e_i, e_j \in N(v)|}{|N(v)|\,(|N(v)| - 1)}$$

Local clustering coefficient 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. Figure 5.10 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 micro-benchmarks 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 (Section 4.5) and also by configuring read-only replicas of shard servers if weaker consistency is acceptable, similar to TAO [22]. We do not evaluate these mechanisms in this experiment as they are orthogonal to transaction ordering. Node program caching is separately evaluated in Section 5.3.5.

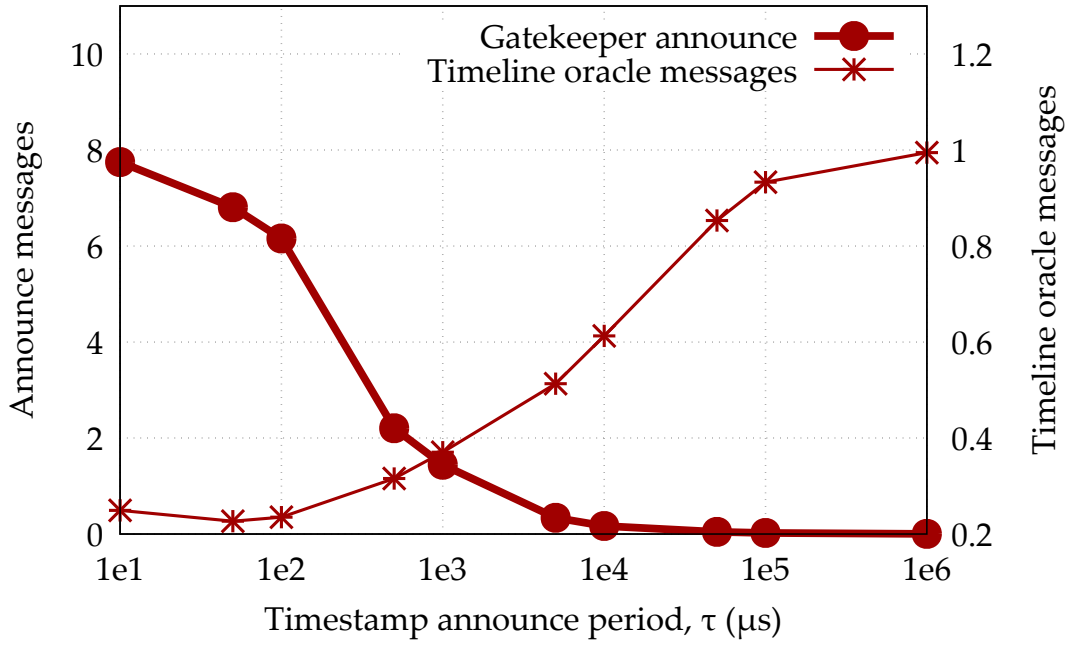


Figure 5.11: Coordination overhead, measured in terms of timestamp announce messages and KRONOS calls, normalized by number of queries. High clock announce frequency results in large gatekeeper coordination overhead, whereas low frequency causes increased KRONOS queries.

5.2.2 Coordination Overhead

Finally, we investigate the tension between proactive (gatekeeper announce messages) and reactive (centralized ordering service 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 τ.

To evaluate this tradeoff, we measured the number of coordination messages due to both gatekeeper announces and KRONOS queries, as a function of τ, to order the same number of transactions. Figure 5.11 shows that for small values of τ, the vector clocks are sufficient for ordering a large fraction of the requests. As

τ increases, the reliance on the KRONOS increases. Both extremes are undesirable and result in high overhead—low values of τ waste gatekeeper CPU cycles in processing announce messages, while high values of τ cause increased latency to due extra KRONOS messages. An intermediate value represents a good tradeoff leading to high-throughput timestamping with occasional concurrent transactions consulting KRONOS.

5.3 End to End Graph Store Evaluation

In this section, we evaluate the performance of a full WEAVER implementation comprising 40K lines of C++ code. Unless otherwise mentioned, we turn off dynamic repartitioning (Section 4.4) and node program caching (Section 4.5) features in order to measure the core system performance.

5.3.1 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 [42]. The total data stored by CoinGraph comprises more than 1.2B edges and occupies $\sim$ 900 GB on disk, which exceeds the cumulative memory (700 GB). We thus rely on WEAVER's demand paging technique to read vertices and edges from

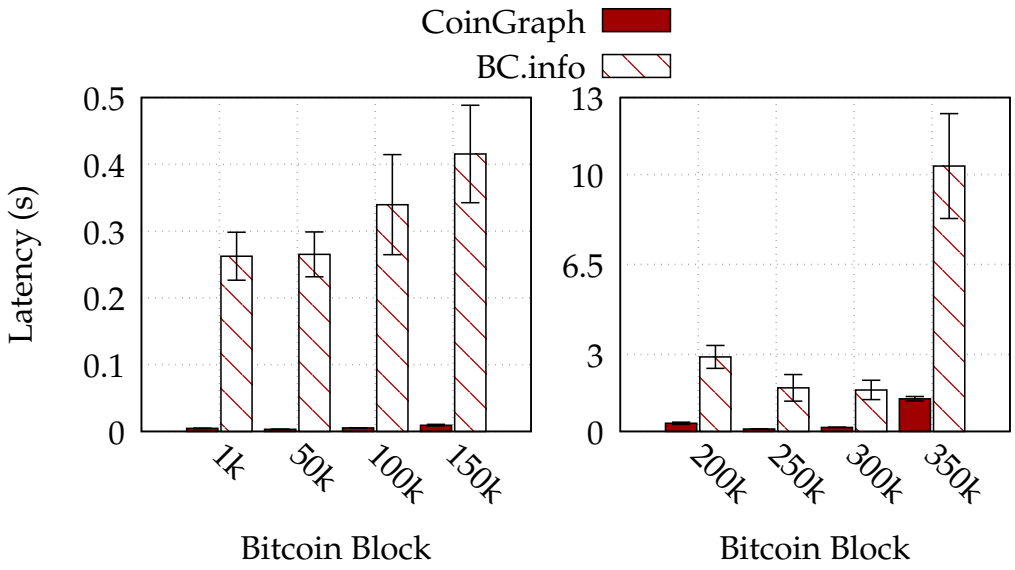


Figure 5.12: Average latency (secs) of a Bitcoin block query in blockchain application. CoinGraph, backed by WEAVER, is an order of magnitude faster than Blockchain.info.

HyperDex Warp in to the memory of WEAVER shards to accommodate the entire blockchain data (Section 4.7).

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 [18], a state-of-the-art commercial block explorer service backed by MySQL [111]. We use their blockchain raw data API that returns data identical to CoinGraph in JSON format.

The results (Figure 5.12) 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 Figure 5.12 should be interpreted cautiously as they include overheads such as WAN latency, which is 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, $8\times$ faster than Blockchain.info. The absolute numbers from this experiment are given in Table 5.1.

Block	**Number of transactions**	**CoinGraph**		**BC.info**	
		Latency	**StdErr**	**Latency**	**StdErr**
$1,000$	1	0.0045	0.0001	0.2623	0.0140
$50,000$	1	0.0031	0.0001	0.2651	0.0130
$100,000$	4	0.0053	0.0001	0.3396	0.0291
$150,000$	10	0.0093	0.0005	0.4155	0.0283
$200,000$	388	0.3161	0.0176	2.9040	0.1729
$250,000$	156	0.0985	0.0009	1.6991	0.2007
$300,000$	237	0.1594	0.0009	1.6145	0.1460
$350,000$	1795	1.2732	0.0003	10.331	0.7926

Table 5.1: Average latency, in seconds, of a Bitcoin block query in blockchain explorer applications. CoinGraph, backed by WEAVER, is an order of magnitude faster than Blockchain.info.

We also evaluate the throughput of block queries supported by CoinGraph. Figure 5.13 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, Figure 5.13 also reports the rate of vertices read by the system. The system is able to sustain node programs that perform 5,000

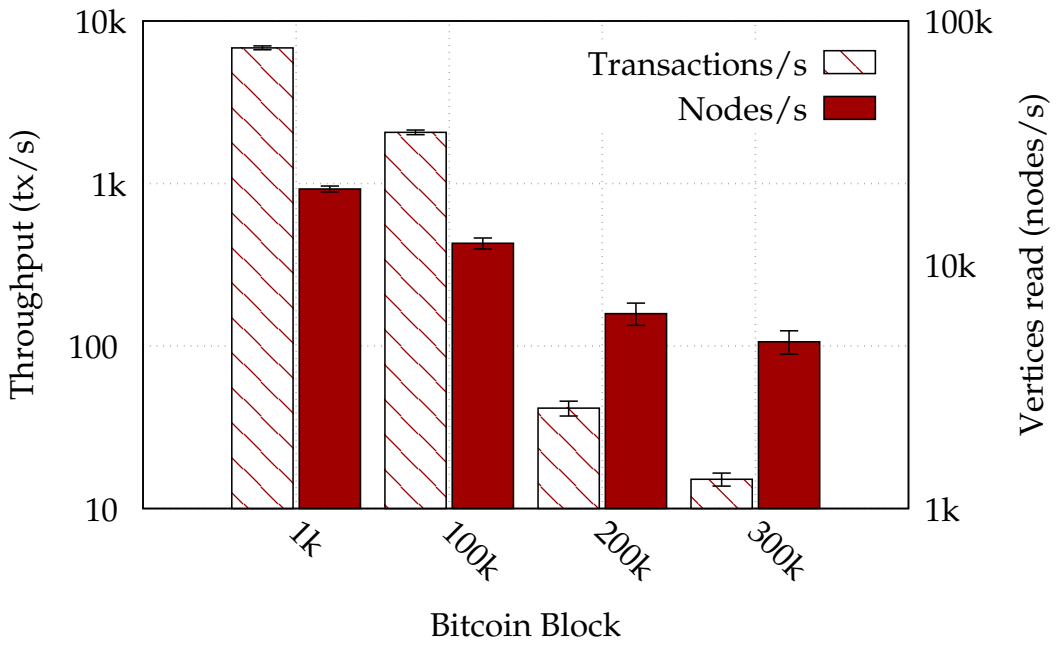


Figure 5.13: Throughput of Bitcoin block render queries in CoinGraph. Each query is a multi-hop node program. Throughput decreases as block size increases since higher blocks have more Bitcoin transactions (more nodes) per query.

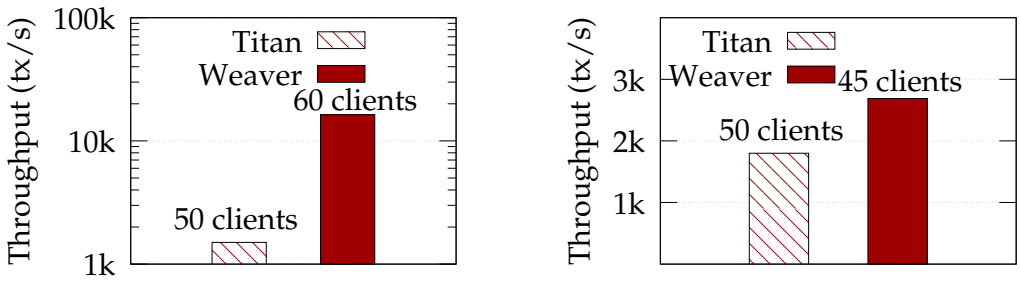
Reads 99.8%	`get_edges`	59.4%
	`count_edges`	11.7%
	`get_node`	28.9%
Writes 0.2%	`create_edge`	80.0%
	`delete_edge`	20.0%

Table 5.2: Social network workload based on Facebook's TAO.

to 20,000 node reads per second.

5.3.2 Social network benchmark

We next evaluate WEAVER's performance on Facebook's TAO workload [22] (Table 5.2) using a snapshot of the LiveJournal social network [12] comprising



(a) Social Network Workload

(b) 75% Read Workload

Figure 5.14: Throughput on a mix of read and write transactions on the LiveJournal graph. WEAVER outperforms Titan by $10\times$ on a read-heavy TAO workload, and by $1.5\times$ 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.

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 [33], 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 Section 5.3.1.

5.3.2.1 Throughput

Figure 5.14(a) shows the throughput of WEAVER compared to Titan. WEAVER outperforms Titan by a factor of $10\times$. WEAVER also significantly outperforms Titan across benchmarks that comprise different fractions of reads and writes as

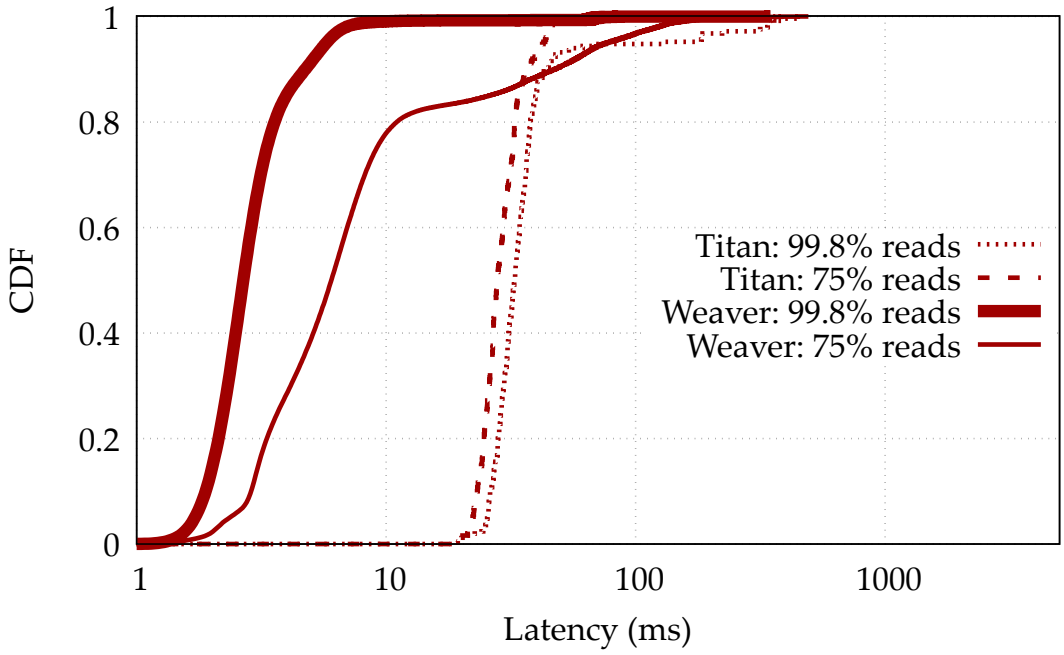


Figure 5.15: CDF of transaction latency for a social network workload on the LiveJournal graph. WEAVER provides significantly lower latency than Titan for all reads and most writes.

shown in Figure 5.14(b).

Titan provides limited throughput because it uses two-phase commit with distributed locking in the commit phase to ensure serializability [103]. 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 be-

cause node programs can execute on a snapshot of the graph defined by the timestamp of the transaction.

5.3.2.2 Latency

Figure 5.15 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.

5.3.3 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 [106]. For such queries, we compare WEAVER's performance to GraphLab [51] 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 Section 5.3.2.

The benchmark consists of reachability queries on a small Twitter graph [85] consisting of 1.76M edges (43 MB) between vertices chosen uniformly at random. We implement the reachability queries as breadth-first search traversals

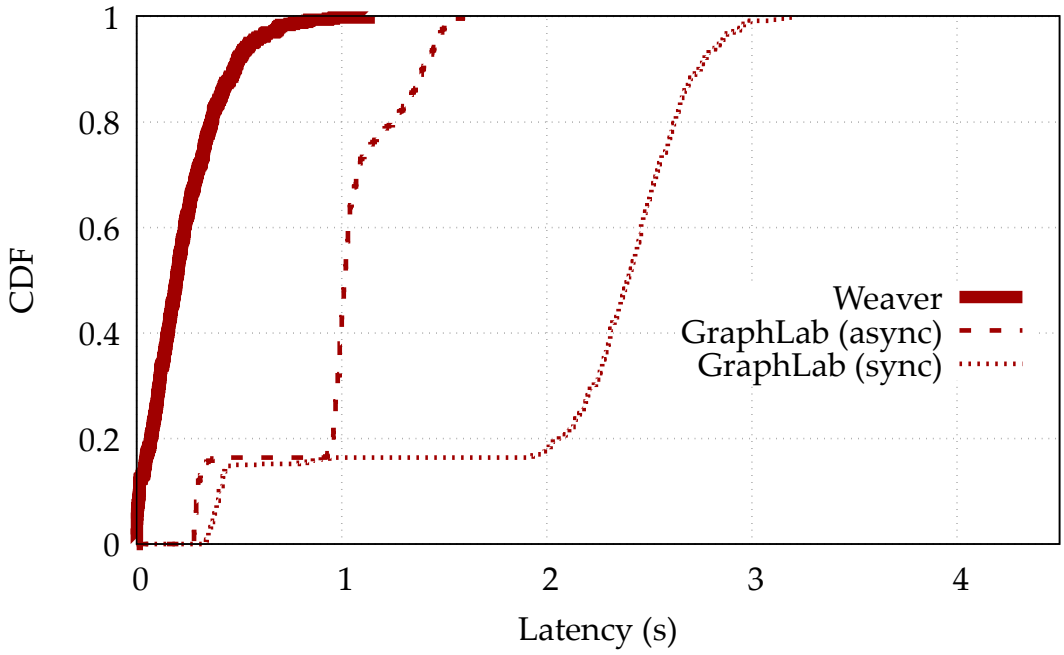


Figure 5.16: CDF of latency of traversals on the small Twitter graph. WEAVER provides 4.3×-9.4× lower latency than GraphLab in spite of supporting mutating graphs with transactions.

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. Figure 5.16 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.

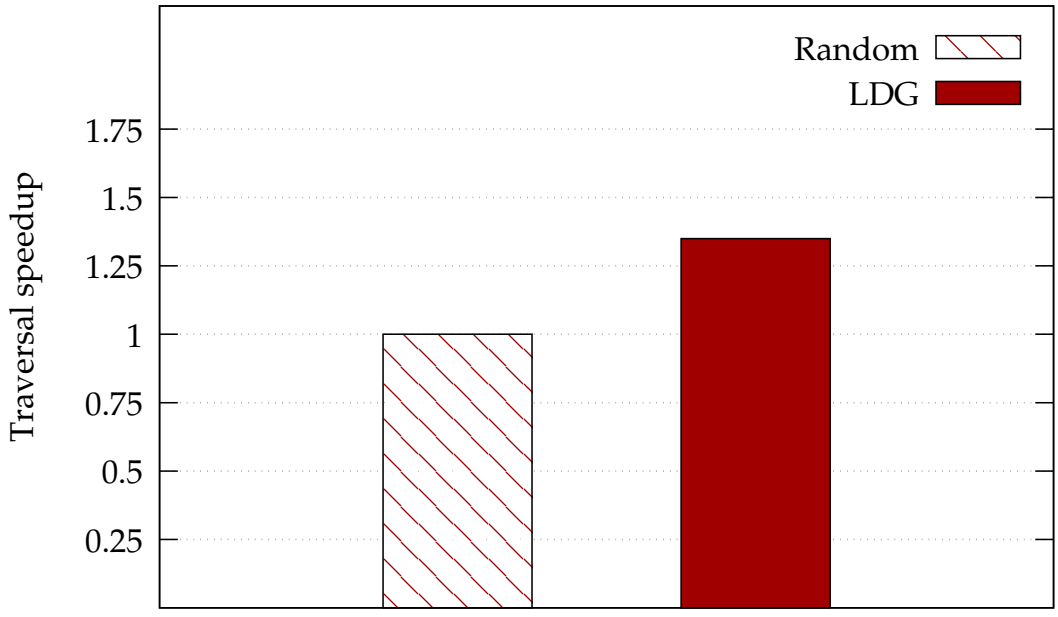


Figure 5.17: Latency for performing traversals on Twitter graph. Dynamic repartitioning results in over 35% speedup.

5.3.4 Dynamic Repartitioning

WEAVER's design permits a wide-variety of streaming graph repartitioning algorithms. Figure 5.17 shows the results of a simple evaluation of a WEAVER implementation that uses the Linear Deterministic Greedy graph partitioning heuristic introduced by Stanton et. al. [112] (Section 4.4). Compared to the execution of random BFS traversals on a hash-partitioned Twitter graph [85], an implementation that first repartitions the graph using the LDG heuristic achieves a 35% speedup in terms of average traversal latency.

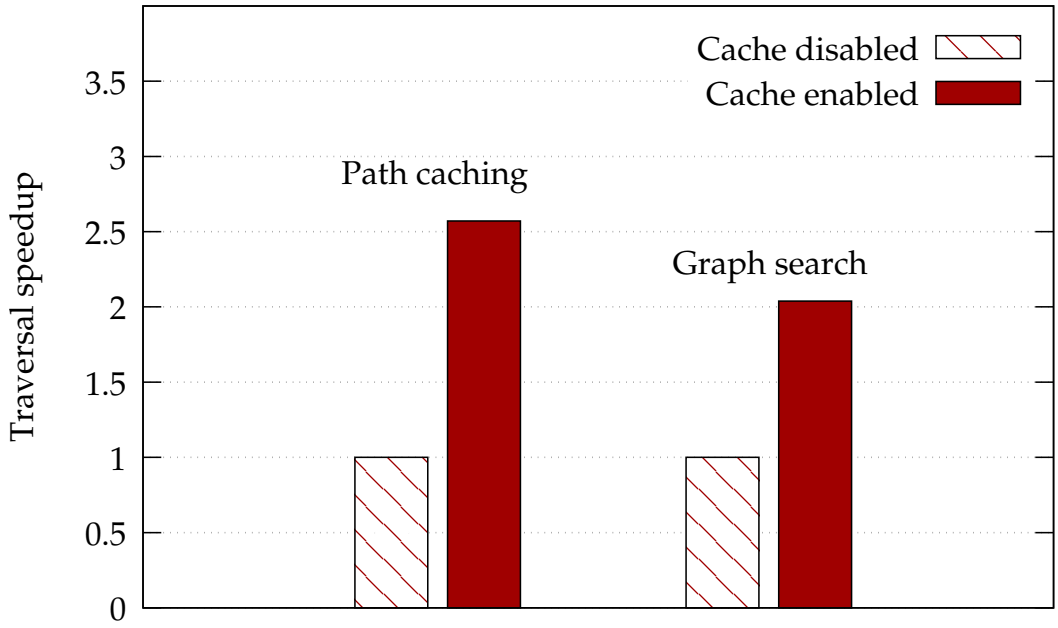


Figure 5.18: Latency for performing traversals. WEAVER's caching framework causes over $2\times$-$2.5\times$ speedup.

5.3.5 Caching

WEAVER enables graph queries to cache the results of the computation at vertices. When a query performs a lot of computation, this can lead to large gains.

To evaluate WEAVER's caching framework, we used two different workloads. The first workload simulates a social network scenario involving requests that search the k-hop neighborhood of a user and then cache the data at the users vertex. The prototypical example is Facebook graph search [106], where user's routinely ask questions such as: return the list of friends of friends who went to the same high school, live in the same city, and share similar interests as the user. Another example is rendering an image (a vertex in the graph), which requires the system to explore the image's neighborhood to figure out access control and the list of people who liked/commented on the image. These types of queries

can use extensive caching for speed up—we can cache the list of users in the first case and the access control list in the second case. We generalize such a workload by executing, on the Twitter graph, set of two-hop traversals at vertices chosen at random from a powerlaw distribution proportional to the out-degree of the vertex and caching user-specified data gathered from this traversal at the source.

The second workload simulates traversals which read a larger portion of the graph. For example, discovering paths in a large SDN or content delivery network involves traversing the source vertex's connected component until we reach the destination. The routes discovered from such queries can be cached at the source, as well as the intermediate vertices, for subsequent queries. We performed these evaluations a graph of the Gnutella peer-to-peer network.

Figure 5.18 shows that WEAVER's caching scheme provides a speedup of over $2\times$ in terms of the average latency of requests for these two representative workloads.

5.4 Chapter Summary

In this chapter, we performed various micro-benchmarks and end-to-end evaluations of the different systems built throughout the dissertation. We have the following key findings:

- Microbenchmarks on KRONOS, a full implementation of the ordering service concept, show that it can scale up to hundreds of thousands of operations per second on a single server.

- Factoring out ordering from the core distributed system design into a separate service can lead to a variety of applications, such as graph stores (KRONOGRAPH) and key-value stores, which perform favorably compared to state of the art systems in the space. Such applications can also uphold strong guarantees and reuse of ordering decisions across components.
- WEAVER enables CoinGraph to execute Bitcoin block queries $8\times$ faster than Blockchain.info [18] (Section 5.3.1).
- WEAVER outperforms Titan [33] by $10\times$ on social network workload [22] (Section 5.3.2) and outperforms GraphLab [51] by $4\times$ on node program workload (Section 5.3.3).
- WEAVER scales linearly with the number of gatekeeper and shard servers for graph analysis queries (Section 5.2.1).
- WEAVER balances the tension between proactive and reactive ordering overheads (Section 5.2.2).
- significantly improves baseline performance with the help of dynamic repartitioning (Section 4.4) and node program caching (Section 4.5).

CHAPTER 6

RELATED WORK

We characterize past work into three categories: event ordering, data storage and processing, and graph management.

6.1 Event Ordering

To our knowledge, KRONOS is the first system to introduce a service-oriented approach to ordering in distributed systems. While prior work often addresses event-ordering at the storage or communication levels, KRONOS provides a more general abstraction that enables these applications and more.

Broadly speaking, prior work on ordering may be divided into the following categories.

6.1.1 Causality Capturing Techniques

Determining the ordering of events is a classic distributed systems problem with many well-known solutions. The problem was originally articulated as the motivation for Lamport timestamps [72], which capture happens-before relationships and provide a total ordering across events. However, Lamport timestamps can create spurious relationships that do not affect the correctness of the application.

Vector clocks [45, 83] permit finer-grained partial orders than Lamport timestamps by establishing a partial order across events. Vector clocks use a vector

of logical clocks to capture happens-before relationships. They enable more parallelism in the partial order than Lamport timestamps, but consume more space to achieve this. In the worst case, vector clocks require as many entries as parallel processes in the system [25] and exhibit significant overhead in deployments where there is a high-rate of server or process churn. The trade-off inherent to vector clocks is that the incidence of false relationships is inversely proportional to the granularity at which the vector clock is maintained.

There has been much work on improving vector clocks. Clock Trees [11] provide support for nested fork-join parallelism, Plausible Clocks [119] offer constant size timestamps while retaining accuracy close to vector clocks. Hierarchical Vector Clocks [66] provide more compact timestamps that adapt to the structure of the underlying network. While these techniques improve the trade-off between granularity and performance, they still restrict the kinds of dependencies an application may specify, and are not fully general.

KRONOS takes an entirely different approach as compared to timestamp-based systems in how it captures causality. It maintains an explicit event dependency graph to track causality relationships and offers fine grain control to the application. By externalizing event/dependency handling and management and providing a unified API, KRONOS simplifies event-ordering management for applications and enables dependencies to span application boundaries.

6.1.2 Ordering Primitives

Some recent systems have offered low-level timing and ordering primitives that simplify high-level distributed system design, very much in the spirit of

a global event ordering service proposed in this dissertation. TrueTime [31], as discussed in detail in Section 3.4, is a clock primitive introduced by Google that provides highly synchronized clocks with tight error bounds using specialized hardware such as atomic/GPS clocks. Mostly-Ordered Multicast [99] is a primitive that provides best-effort ordered multicast messaging by leveraging particular characteristics in modern datacenter networks such as fixed topology and software-defined networking. Other systems use a centralized sequencer [98, 14, 43], which is one possible implementation of a centralized ordering service discussed in Chapter 2, to order appends to a shared distributed log.

6.1.3 Consensus Protocols

Consensus protocols enable applications to construct a total order across all events. Examples of consensus protocols for non-Byzantine scenarios include Paxos [71], a crash-fault tolerant consensus protocol; Viewstamped replication [96] which operates in a primary-backup fashion; Tango [14], which replicates in-memory data structures using a shared, totally-ordered log; and multi-phase commit protocols [73, 110], a class of protocols that ensure all participants in a distributed transaction agree on whether to commit or abort by special-casing consensus [54].

KRONOS permits applications to maintain a partial order across events in the system, increasing the flexibility with which events may be ordered. Of course, applications may always institute a total order across all events using KRONOS.

6.1.4 Application-Level Dependencies

Many systems rely on application-specific mechanisms to resolve and order events. Dynamo [35] is an eventually consistent key-value store that improves availability by using vector clocks to resolve concurrent writes. COPS [78] provides low-latency geo-replication using causal consistency and uses an application-specific conflict resolution mechanism to merge conflicting writes. Saturn [20] provides geo-replicated causal event ordering. Others have advocated for explicit causality by suggesting that applications explicitly select the subset of happens-before relationships that the data store should preserve to uphold application-level invariants [13].

These approaches are complementary to KRONOS because KRONOS provides a general method for event ordering in the form of a service. Applications may use KRONOS within the application-defined handlers of causally-consistent data stores. Further, applications may explicitly, and directly, declare happens-before relationships in KRONOS. Unlike other forms of application-level dependency management, KRONOS permits the development of reusable components that naturally compose to achieve application-specific guarantees.

6.2 Distributed Databases

6.2.1 Distributed Transactions

There has been much recent work on implementing protocols for transactions in distributed data stores. Distributed Data Structures [56, 55] provided the key in-

sight: for a general class of commonly-used data structures, such as hash tables, logs, and graphs, designing a specialized high-performance consistent system is well worth the engineering effort. HyperDex Warp [42] provides the linear transactions protocol which performs multiple passes over the keys participating in a transaction to enable multi-key transactions. This was soon followed by other work that applied the idea of transactional chains in geo-replicated scenarios [132]. Spanner [31] leverages tightly synchronized global clocks using specialized hardware to provide WAN transactions. Other work explicitly trades off consistency level of distributed transactions to ensure meeting service-level performance agreements [116].

6.2.2 Concurrency Control

Pessimistic two-phase locking [53] ensures correctness and strong consistency but excessively limits concurrency. Optimistic concurrency control techniques [67] are feasible in scenarios where the expected contention on objects is low and transaction size is small. FaRM [37] uses optimistic concurrency control and multi-phase commit protocol 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 [101, 93], which enables long-running graph algorithms to read a consistent snapshot of the graph. Bohm [43] is a similar MVCC-based concurrency control protocol for multi-core settings which serializes timestamp assignment at a single thread. Centiman [36] 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 (Section 4.3) and node programs (Section 3.5.1). Deuteronomy [75] is a centralized, multi-core database that implements MVCC using a latch-free transaction table.

6.2.3 Consistency Models

Many systems internally manage event ordering and track inter-process communication to provide causal consistency. Representative storage system examples include Bayou [117], a replica management system that exchanges logs between servers, allows for connection disruptions without preventing progress, and manages conflict resolution of causally conflicting operations through a set of user specified merge procedures. Depot [80] and SPORC [44] are cloud storage systems which employ variants of Fork-Join-Causal or Fork* consistency to enable practical cloud applications which can operate on untrusted cloud servers. Causal multicast [16, 17] protocols respect causal order when delivering messages to applications. Causality is also useful for supporting speculative execution [94], and bug and fault detection [10]. Externalizing event ordering to KRONOS enables causal consistency guarantees that span multiple applications.

Many existing databases support only weak consistency models, such as eventual consistency [22, 79]. WEAVER supports strictly serializable operations, as do few other contemporary systems [31, 42, 92, 33].

6.2.4 Fault Tolerance

KRONOS uses chain replication [123] to replicate its data. KRONOS's reads from stale replicas resemble the apportioned queries in CRAQ [115]. Unlike CRAQ, the implementation used in KRONOS does not require querying the chain tail to validate reads; the monotonicity invariant ensures that if a query returns a result, the result is as valid as if it were generated by the tail itself.

Other components of WEAVER uses Paxos [71] to implement a fault tolerant replicated state machine [108].

6.3 Graph Stores

6.3.1 Data Processing Systems

There exist many systems for large-scale processing of data, both graph-structured and otherwise [81, 47, 105, 89, 51, 69, 104, 100, 129, 87, 27, 52, 134, 127, 23, 26, 48, 131, 130, 133, 118, 58, 65, 30].

While many of these systems have a graph-oriented API, others can support graph operations as layer on top of their native API. In particular, dataflow-based systems as described by [89, 52, 23] can support database operations by treating each operation as a functional computation mapped over input data and sequence of transactions [86]. This approach is, of course, very different from the way traditional data stores such as WEAVER manage the system state. One may expect that database operations implemented in an iterative dataflow

model may have higher latency due to batching; the exact latency depends on workload characteristics and the batch size. Moreover, a dataflow-based system can recover from crash failures by re-executing the operations on the input data, but this technique may be prohibitively expensive for large computations.

6.3.2 Online Graph Databases

The Scalable Hyperlink Store [90] provides the property graph abstraction over data but does not support arbitrary properties on vertices and edges. Trinity [109] is a distributed graph database that does not support ACID transactions. SQLGraph [114] embeds property graphs in a relational database and executes graph traversals as SQL queries. TAO [22] is Facebook's geographically distributed graph backend (Section 4.8.1). Titan [33] supports updates to the graph and a vertex-local query model. Roditty et al. [102] provide algorithms and worst-case runtime analyses for a constrained set of operations, i.e. reachability queries, on dynamic graphs.

Centralized graph databases are suitable for a number of graph processing applications on non-changing, static graphs [87]. However, centralized databases designed for online, dynamic graphs [92, 62, 70, 75] 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 [122, 22] on a single machine.

6.3.3 Temporal Graph Databases

A number of related systems [59, 28, 50] are designed for efficient processing of graphs that change over time. [59] creates an in-memory graph optimized for spatial and temporal locality from a disk-based log of timestamped update operations. WEAVER localizes different versions of a vertex and edge over time by storing a multi-version graph, while WEAVER's graph partitioning scheme ensures spatial locality of neighboring vertices. In addition Chronos cannot process transactional updates because it does not reconcile timestamps assigned by different servers.

Kineograph [28] 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 (Section 5.3.2, Section 5.3.3) and ensure that node programs operate on the latest version of the graph.

6.3.4 Graph Partitioning

At its core, the problem of partitioning a large graph in order to optimize dual objectives—better load balancing and fewer edges cut— is an NP-complete problem [9]. WEAVER employs a distributed streaming-based partitioning technique which leverages algorithms developed by others to reduce edge cuts and communication costs [112, 95]. METIS is a state of the art partitioner that reduces the large graph into higher-level overlay graphs where each vertex corresponds to multiple vertices in the original graph, and solves the partitioning on

the smaller graph using heavyweight techniques [63]. Better graph partitioning algorithms will improve WEAVER's overall performance, and are an area of future work.

6.3.5 Query Caching

Gedik et al. [50] describe techniques to cache recent graph data in memory in an architecture with multiple levels of storage, similar to WEAVER. However, WEAVER's caching infrastructure can also store partial or entire results of previous node program runs at multiple vertices in the graph. This opens up interesting tradeoffs between storage capacity, query latency, and consistency. Nectar [57] describes techniques that automatically process workloads and cache frequently accessed datasets in large clusters for data-parallel batch processing jobs. Many production relational databases have developed tools for automatically recommending materialized views that reduce query latency for a specified workload [7, 135, 32, 125]. Automatic caching is an interesting future direction for WEAVER.

CHAPTER 7

CONCLUSIONS AND FUTURE DIRECTIONS

Graph-structured data is ubiquitous in modern applications, and poses interesting challenges. First, the scale of the data is immense: for example, social networks regularly ingest billions of data items and terabytes of data per day. Second, the workload is challenging: users of such applications cumulatively issue millions of operations per day. Finally, the inherent nature of graph workloads is a mixture of short read-write transactions combined with long read-only analytical queries. These unique challenges necessitate new techniques for storing and processing large graphs.

7.1 Contributions

This dissertation first tackles the challenge of ordering graph transactions and queries in a distributed architecture. To do so, it introduces the idea of an event ordering service. In this paradigm, the task of ordering is presented as a service in the datacenter, and the various distributed system components may issue calls to the service to query and assign order between events. This dissertation presents an API for such a service based on abstract application-defined events, and also discusses various implementation techniques that cover synchronous and asynchronous designs.

The first instantiation of the ordering service concept is KRONOS. KRONOS tracks happens-before relationships at a fine grain, by storing the entire event dependency graph as a replicated state machine. This enables the application to leverage concurrency of events whenever possible, while also enabling com-

positionality of ordering decisions across systems. KRONOS demonstrates that an ordering service is capable of implementing a variety of modern distributed applications: a transactional graph store, a transaction key-value store, as well as a social network backend. Moreover, in spite of being centralized KRONOS has good performance; applications built on top of KRONOS have comparable or better efficiency than state of the art systems. Thus, for many small applications, even a centralized event ordering service can provide great performance. Of course, KRONOS's real benefit is simplifying the application design by eliminating the complexity of ordering from the core system design.

Next, we scale up the performance of KRONOGRAPH, the graph store implemented on top of KRONOS, by introducing the idea of refinable timestamps. This novel ordering protocol adds a layer of lightweight loosely synchronized timestamps on top of the precise, centralized KRONOS ordering service. For many workloads, the timestamping layer assumes the majority of the ordering responsibilities, while KRONOS resolves the small subset of still conflicting operations. This technique uncovers a tradeoff between proactive ordering due to the clock synchronization frequency and reactive overhead of issuing KRONOS calls. This dissertation explores the tradeoff empirically and discovers a sweet spot which results in low overall overhead.

Finally, we describe the full implementation of WEAVER, a distributed graph store that uses the refinable timestamps protocol to order transactions. In addition to a novel ordering protocol, WEAVER includes many performance optimizations. We demonstrate that the quality of graph partitioning is crucial to the performance of analytical graph queries due to load balancing and inter-machine communication overhead. Because many modern graphs are dynamic

and constantly evolving, this dissertation describes a technique to dynamically reshard the graph partitions on-the-fly while the system is in production. Our experience shows that streaming graph partitioning techniques that relocate a vertex with the majority of its neighbors represent a good tradeoff between graph partition quality and ease of distributed implementation. Moreover, we also show that graph queries can benefit greatly from caching partial query results at the graph vertices, and we describe effective techniques that balance data freshness and query latency.

7.2 Future Directions

This dissertation has uncovered many interesting directions for future research. The key idea is that disparate, distributed components can cooperate on timeline management with the help of a distributed service via a standardized API. Modern production distributed systems do not make use of such distributed services as Kronos and Refinable Timestamps. While this dissertation provided many new ideas to enable a practical realization, more work needs to be done to ensure such services become standard components of the distributed software stack.

First, it would be interesting to see the extents of expressivity and flexibility of factoring out ordering as a separate service. While we described applications such as a social network backend and a graph store based on an ordering service, there may be many more applications that can be simplified and made more efficient when the ordering task is factored out. Conversely, it would be interesting to explore the boundaries of what such an ordering service can ac-

complish.

Next, the scalable refinable timestamps protocol presented in this dissertation was used as the central ordering technique in WEAVER, but there is nothing about the core protocol that limits its usage to only graph workloads. A future research goal is to implement other distributed data structures using refinable timestamps.

The clock synchronization frequency in refinable timestamps is a major tuning knob which, in the current implementation, is adjusted empirically. However, one may design a technique that automatically detects the optimal clock synchronization frequency based on past query arrival times and latency profiles. One potential way to accomplish this is to use a machine learning algorithm that predicts the expected impact of a particular frequency setting on the overall system efficiency.

Finally, WEAVER's query caching technique, although highly effective, requires manual adjustments and invalidation based on changes in the graph structure by the application, and there are plenty of opportunities to automate the caching layer. One of the key challenges for such a cache would be to maintain the overall system consistency guarantee. One the one hand, caching as much as possible will lead to low query latency; on the other hand, many of the cached items may soon become invalidated because of modifications to the graph.

7.3 Final Remarks

This dissertation is about techniques for storing large graphs consistently and efficiently. The main focus of the work is on performance, ease of use, and correctness. The key contributions are techniques for distributed ordering and efficient graph data management. We hope the ideas presented in this thesis are useful to others, and inspire future work on graph data management, and in general large scale distributed systems.

BIBLIOGRAPHY

[1] Facebook Users Are Uploading 350 Million New Photos Each Day. http://www.businessinsider.com/facebook-350-million-photos-each-day-2013-9/.

[2] How Much Data Is Generated Every Minute On Social Media? http://wersm.com/how-much-data-is-generated-every-minute-on-social-media/.

[3] Knowledge Graph - Wikipedia. https://en.wikipedia.org/wiki/Knowledge_Graph.

[4] Leaked Twitter API data shows the number of tweets is in serious decline. http://www.businessinsider.com/tweets-on-twitter-is-in-serious-decline-2016-2/.

[5] The Knowledge Graph. https://www.google.com/intl/en_us/insidesearch/features/search/knowledge.html.

[6] Web Data Commons - Hyperlink Graphs. http://webdatacommons.org/hyperlinkgraph/.

[7] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. Database Tuning Advisor for Microsoft SQL Server 2005. In Proceedings of the *International Conference on Very Large Data Bases,* Toronto, Canada, August 2004.

[8] Thomas Anderson and Michael Dahlin. Operating Systems: Principles and Practice. Recursive Books, 2012.

[9] Konstantin Andreev and Harald Räcke. Balanced Graph Partitioning. In Proceedings of the *Symposium on Parallelism in Algorithms and Architectures,* Barcelona, Spain, June 2004.

[10] Mona Attariyan and Jason Flinn. Using Causality to Diagnose Configuration Bugs. In Proceedings of the *USENIX Annual Technical Conference,* Boston, Massachusetts, June 2008.

[11] Koenraad Audenaert. Clock Trees: Logical Clocks for Programs with Nested Parallelism. In *IEEE Transactions on Software Engineering.,* 23(10), 1997.

[12] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In Proceedings of the *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, Philadelphia, Pennsylvania, August 2006.

[13] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In Proceedings of the *Symposium on Cloud Computing*, San Jose, California, October 2012.

[14] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed Data Structures over a Shared Log. In Proceedings of the *Symposium on Operating Systems Principles*, 2013.

[15] Smriti Bhagat, Moira Burke, Carlos Diuk, Ismail Onur Filiz, and Sergey Edunov. Three and a half degrees of separation. `https://research.fb.com/three-and-a-half-degrees-of-separation/`.

[16] Kenneth P. Birman, André Schiper, and Patrick Stephenson. Fast Causal Multicast. Cornell University, Technical Report TR90-1105, 1990.

[17] Kenneth P. Birman, André Schiper, and Pat Stephenson. Lightweigt Causal and Atomic Group Multicast. In *ACM Transactions on Computer Systems*, 9(3), 1991.

[18] Blockchain. Bitcoin Block Explorer. `https://blockchain.info/`.

[19] Paolo Boldi and Sebastiano Vigna. The WebGraph Framework I: Compression Techniques. In Proceedings of the *International World Wide Web Conference*, New York, New York, May 2004.

[20] Manuel Bravo, Luís Rodrigues, and Peter Van Roy. Saturn: A Distributed Metadata Service for Causal Consistency. In Proceedings of the *European Conference on Computer Systems*, Belgrade, Serbia, April 2017.

[21] Preston Briggs and Linda Torczon. An Efficient Representation for Sparse Sets. In *ACM Letters on Programming Languages and Systems*, 2(1-4), 1993.

[22] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and

Venkat Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In Proceedings of the *USENIX Annual Technical Conference*, San Jose, California, June 2013.

[23] Yingyi Bu, Vinayak Borkar, Jianfeng Jia, Michael J. Carey, and Tyson Condie. Pregelix: Big(ger) Graph Analytics on A Dataflow Engine. In *Proceedings of the VLDB Endowment*, 8(2), 2014.

[24] Michael Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In Proceedings of the *Symposium on Operating System Design and Implementation*, Seattle, Washington, November 2006.

[25] Bernadette Charron-Bost. Concerning the Size of Logical Clocks in Distributed Systems. In *Information Processing Letters*, 39(1), 1991.

[26] Rishan Chen, Mao Yang, Xuetian Weng, Byron Choi, Bingsheng He, and Xiaoming Li. Improving Large Graph Processing on Partitioned Graphs in the Cloud. In Proceedings of the *Symposium on Cloud Computing*, San Jose, California, October 2012.

[27] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In Proceedings of the *European Conference on Computer Systems*, 2015.

[28] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Taking the Pulse of a Fast-Changing and Connected World. In Proceedings of the *European Conference on Computer Systems*, Bern, Switzerland, April 2012.

[29] David R. Cheriton and Dale Skeen. Understanding the Limitations of Causally and Totally Ordered Communication. In Proceedings of the *Symposium on Operating Systems Principles*, Asheville, North Carolina, October 1993.

[30] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One Trillion Edges: Graph Processing at Facebook-Scale. In *Proceedings of the VLDB Endowment*, 8(12), 2015.

[31] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David

Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In Proceedings of the *Symposium on Operating System Design and Implementation*, 2012.

[32] Benoit Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zait, and Mohamed Ziauddin. Automatic SQL Tuning in Oracle 10g. In *Proceedings of the VLDB Endowment*, 2004.

[33] Datastax. Titan: Distributed Graph Database. `http://titan.thinkaurelius.com/`.

[34] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In Proceedings of the *Symposium on Operating Systems Principles*, Stevenson, Washington, October 2007.

[35] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In Proceedings of the *Symposium on Operating Systems Principles*, Stevenson, Washington, October 2007.

[36] Bailu Ding, Lucja Kot, Alan J. Demers, and Johannes Gehrke. Centiman: Elastic, High Performance Optimistic Concurrency Control by Watermarking. In Proceedings of the *Symposium on Cloud Computing*, Hawaii, Hawaii, August 2015.

[37] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In Proceedings of the *Symposium on Operating Systems Principles*, Monterey, California, October 2015.

[38] Ayush Dubey, Greg D. Hill, Robert Escriva, and Emin Gün Sirer. Weaver: A High-Performance, Transactional Graph Database Based on Refinable Timestamps. In *Proceedings of the VLDB Endowment*, 9(11), 2016.

[39] Ramez A. Elmasri and Shamkant Navathe. Fundamentals of Database Systems. Addison-Wesley, US, 2010.

[40] Paul Erdös and Alfréd Rényi. On the Evolution of Random Graphs. In *Mathematical Institute of the Hungarian Academy of Sciences*, 5(17–61), 1960.

[41] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A Distributed, Searchable Key-Value Store. In Proceedings of the *SIGCOMM Conference*, Helsinki, Finland, August 2012.

[42] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Warp: Multi-Key Transactions for Key-Value Stores. Cornell University, Technical Report, 2013.

[43] Jose M. Faleiro and Daniel J. Abadi. Rethinking serializable multiversion concurrency control. In *Proceedings of the VLDB Endowment*, 8(11), 2015.

[44] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: Group Collaboration using Untrusted Cloud Resources. In Proceedings of the *Symposium on Operating System Design and Implementation*, Vancouver, Canada, October 2010.

[45] Colin J. Fidge. Timestamps in Message-Passing Systems That Preserve the Partial Ordering. In *Australian Computer Science Communications*, 10(1), 1988.

[46] The Apache Software Foundation. Apache Zookeeper. `https://zookeeper.apache.org/`.

[47] The Apache Software Foundation. Apache Giraph. `http://giraph.apache.org/`.

[48] The Apache Software Foundation. Apache Hama. `http://hama.apache.org/`.

[49] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. Database Systems. Pearson Prentice Hall, 2009.

[50] Bŭgra Gedik and Rajesh Bordawekar. Disk-Based Management of Interaction Graphs. In *IEEE Transactions on Knowledge and Data Engineering*, 26(11), 2014.

[51] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Nat-

ural Graphs. In Proceedings of the *Symposium on Operating System Design and Implementation*, 2012.

[52] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In Proceedings of the *Symposium on Operating System Design and Implementation*, 2014.

[53] James N. Gray. Notes on Data Base Operating Systems. Springer, 1978.

[54] Jim Gray and Leslie Lamport. Consensus on Transaction Commit. Microsoft Research, Technical Report MSR-TR-2003-96, 2004.

[55] Steven D. Gribble. A Design Framework and a Scalable Storage Platform to Simplify Internet Service Construction. PhD thesis, University of California at Berkeley, 2000.

[56] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, Distributed Data Structures for Internet Service Construction. In Proceedings of the *Symposium on Operating System Design and Implementation*, San Diego, California, October 2000.

[57] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic Management of Data and Computation in Datacenters. In Proceedings of the *Symposium on Operating System Design and Implementation*, Vancouver, Canada, October 2010.

[58] Minyang Han and Khuzaima Daudjee. Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems. In *Proceedings of the VLDB Endowment*, 8(9), 2015.

[59] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: A Graph Engine for Temporal Graph Analysis. In Proceedings of the *European Conference on Computer Systems*, Amsterdam, The Netherlands, April 2014.

[60] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. In *ACM Transactions on Programming Languages and Systems*, 12(3), 1990.

[61] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In Proceedings of the *USENIX Annual Technical Conference*, Boston, Massachusetts, June 2010.

[62] Borislav Iordanov. HyperGraphDB: A Generalized Graph Database. In Proceedings of the *International Conference on Web-Age Information Management*, Jiuzhaigou Valley, China, July 2010.

[63] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. In *SIAM Journal on Scientific Computing*, 20(1), 1998.

[64] Alfons Kemper and Thomas Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In Proceedings of the *IEEE International Conference on Data Engineering*, Hannover, Germany, April 2011.

[65] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In Proceedings of the *European Conference on Computer Systems*, Prague, Czech Republic, April 2013.

[66] Denis Andreyevich Khotimsky. Hierarchical Vector Clock: Scalable Plausible Clock for Detecting Causality in Large Distributed Systems. In Proceedings of the *International Conference on ATM*, Colmar, France, 1999.

[67] H. T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. In *ACM Transactions on Database Systems*, 6(2), 1981.

[68] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a Social Network or a News Media? In Proceedings of the *International World Wide Web Conference*, Raleigh, North Carolina, April 2010.

[69] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In Proceedings of the *Symposium on Operating System Design and Implementation*, 2012.

[70] Aapo Kyrola and Carlos Guestrin. GraphChi-DB: Simple Design for a Scalable Graph Database System - on Just a PC. In *Computing Research Repository*, abs/1403.0701, 2014.

[71] Leslie Lamport. The Part-Time Parliament. In *ACM Transactions on Computer Systems*, 16(2), 1998.

[72] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. In *Communications of the ACM*, 21(7), 1978.

[73] Butler Lampson and Howard E. Sturgis. Crash Recovery in a Distributed Storage System. Xerox Parc, Palo Alto, California, Technical Report, 1976.

[74] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally Synchronized Time via Datacenter Networks. In Proceedings of the *SIGCOMM Conference*, 2016.

[75] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. High Performance Transactions in Deuteronomy. In Proceedings of the *Conference on Innovative Data Systems Research*, Asilomar, California, January 2015.

[76] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have Things Changed Now?: An Empirical Study of Bug Characteristics in Modern Open Source Software. In Proceedings of the *ACM Workshop on Architectural and System Support for Improving Software Dependability*, San Jose, California, October 2006.

[77] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In Proceedings of the *Symposium on Operating Systems Principles*, Cascais, Portugal, October 2011.

[78] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In Proceedings of the *Symposium on Operating Systems Principles*, Cascais, Portugal, October 2011.

[79] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential Consistency: Measuring and Understanding Consistency at Facebook. In Proceedings of the *Symposium on Operating Systems Principles*, Monterey, California, October 2015.

[80] Prince Mahajan, Srinath T. V. Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Michael Dahlin, and Michael Walfish. Depot: Cloud Storage with

Minimal Trust. In Proceedings of the *Symposium on Operating System Design and Implementation*, Vancouver, Canada, October 2010.

[81] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In Proceedings of the *SIGMOD International Conference on Management of Data*, Indianapolis, Indiana, June 2010.

[82] Dahlia Malkhi and Doug Terry. Concise Version Vectors in WinFS. In Proceedings of the *International Conference on Distributed Computing*, Cracow, Poland, September 2005.

[83] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In Proceedings of the *Workshop on Parallel and Distributed Algorithms*, Chateau de Bonas, France, October 1989.

[84] Julian J. McAuley and Jure Leskovec. Learning to Discover Social Circles in Ego Networks. In Proceedings of the *Advances in Neural Information Processing Systems*, Lake Tahoe, California, December 2012.

[85] Julian J. McAuley and Jure Leskovec. Learning to Discover Social Circles in Ego Networks. In Proceedings of the *Advances in Neural Information Processing Systems*, Lake Tahoe, California, December 2012.

[86] Frank McSherry. Dataflow as Database. `https://github.com/frankmcsherry/blog/blob/master/posts/2016-07-17.md/`.

[87] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! But at what COST? In Proceedings of the *Workshop on Hot Topics in Operating Systems*, 2015.

[88] David L. Mills. Internet Time Synchronization: The Network Time Protocol. In *IEEE Transactions on Computing*, 39(10), 1991.

[89] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. In Proceedings of the *Symposium on Operating Systems Principles*, 2013.

[90] Marc Najork. The Scalable Hyperlink Store. In Proceedings of the *ACM Conference on Hypertext and Hypermedia*, Torino, Italy, June 2009.

[91] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008.

[92] Neo4j. Neo4j: The World's Leading Graph Database. `http://neo4j.com/`.

[93] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In Proceedings of the *SIGMOD International Conference on Management of Data*, Melbourne, Victoria, Australia, May 2015.

[94] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the Sync. In Proceedings of the *Symposium on Operating System Design and Implementation*, Seattle, Washington, November 2006.

[95] Joel Nishimura and Johan Ugander. Restreaming Graph Partitioning: Simple Versatile Algorithms For Advanced Balancing. In Proceedings of the *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, Chicago, Illinois, August 2013.

[96] Brian M. Oki and Barbara Liskov. Viewstamped Replication: A General Primary Copy. In Proceedings of the *ACM Symposium on Principles of Distributed Computing*, Toronto, Canada, August 1988.

[97] Daniel Peng and Frank Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In Proceedings of the *Symposium on Operating System Design and Implementation*, Vancouver, Canada, October 2010.

[98] Daniel Peng and Frank Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In Proceedings of the *Symposium on Operating System Design and Implementation*, Vancouver, Canada, October 2010.

[99] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In Proceedings of the *Symposium on Networked System Design and Implementation*, Oakland, California, May 2015.

[100] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haridasan. Managing Large Graphs on Multi-Cores With Graph Awareness. In Proceedings of the *USENIX Annual Technical Conference*, Boston, Massachusetts, June 2012.

[101] David Patrick Reed. Naming And Synchronization in a Decentralized

Computer System. Massachusetts Institute of Technology, Technical Report, 1978.

[102] Liam Roditty and Uri Zwick. A Fully Dynamic Reachability Algorithm for Directed Graphs with an Almost Linear Update Time. In Proceedings of the *ACM Symposium on Theory of Computing*, Chicago, Illinois, June 2004.

[103] Marko A. Rodriguez and Matthias Broecheler. `http://www.slideshare.net/slidarko/titan-the-rise-of-big-graph-data/`.

[104] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In Proceedings of the *Symposium on Operating Systems Principles*, 2013.

[105] Semih Salihoglu and Jennifer Widom. GPS: A Graph Processing System. In Proceedings of the *International Conference on Scientific and Statistical Database Management*, Baltimore, Maryland, July 2013.

[106] Sriram Sankar, Soren Lassen, and Mike Curtiss. Under the Hood: Building out the infrastructure for Graph Search. `https://www.facebook.com/notes/facebook-engineering/under-the-hood-building-out-the-infrastructure-for-graph-search/10151347573598920/`.

[107] Ashutosh Saxena, Ashesh Jain, Ozan Sener, Aditya Jami, Dipendra K Misra, and Hema S. Koppula. RoboBrain: Large-Scale Knowledge Engine for Robots. Cornell University and Stanford University, Technical Report, 2015.

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

[109] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In Proceedings of the *SIGMOD International Conference on Management of Data*, New York, New York, June 2013.

[110] Dale Skeen and Michael Stonebraker. A Formal Model of Crash Recovery in a Distributed System. In *IEEE Transactions on Software Engineering.*, 9(3), 1983.

[111] Peter Smith. Personal communication, Blockchain, 2015.

[112] Isabelle Stanton and Gabriel Kliot. Streaming Graph Partitioning for Large Distributed Graphs. In Proceedings of the *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, Beijing, China, August 2012.

[113] Greg Sterling. Data: Google monthly search volume dwarfs rivals because of mobile advantage. `http://searchengineland.com/data-google-monthly-search-volume-dwarfs-rivals-mobile-advantage-269120/`.

[114] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guo Tong Xie. SQLGraph: An Efficient Relational-Based Property Graph Store. In Proceedings of the *SIGMOD International Conference on Management of Data*, Melbourne, Victoria, Australia, May 2015.

[115] Jeff Terrace and Michael J. Freedman. Object Storage on CRAQ: High-throughput chain replication for read-mostly workloads. In Proceedings of the *USENIX Annual Technical Conference*, San Diego, California, June 2009.

[116] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based Service Level Agreements for Cloud Storage. In Proceedings of the *Symposium on Operating Systems Principles*, 2013.

[117] Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In Proceedings of the *Symposium on Operating Systems Principles*, Copper Mountain, Colorado, December 1995.

[118] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From "Think Like a Vertex" to "Think Like a Graph". In *Proceedings of the VLDB Endowment*, 7(3), 2013.

[119] Francisco J. Torres-Rojas and Mustaque Ahamad. Plausible Clocks: Constant Size Logical Clocks for Distributed Systems. In *Distributed Computing*, 12(4), 1999.

[120] TPC. TPC-C Homepage. `http://www.tpc.org/tpcc/`.

[121] Johan Ugander and Lars Backstrom. Balanced Label Propagation for Partitioning Massive Graphs. In Proceedings of the *International ACM Conference on Web Search and Data Mining*, Rome, Italy, February 2013.

[122] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The Anatomy of the Facebook Social Graph. In *Computing Research Repository*, abs/1111.4503, 2011.

[123] Robbert van Renesse and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. In Proceedings of the *Symposium on Operating System Design and Implementation*, San Francisco, California, December 2004.

[124] Robbert van Renesse and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. In Proceedings of the *Symposium on Operating System Design and Implementation*, San Francisco, California, December 2004.

[125] Ramakrishna Varadarajan, Vivek Bharathan, Ariel Cary, Jaimin Dave, and Sreenath Bodagala. DBDesigner: A Customizable Physical Design Tool for Vertica Analytic Database. In Proceedings of the *IEEE International Conference on Data Engineering*, 2014.

[126] Werner Vogels. Eventually Consistent. In *Communications of the ACM*, 52(1), 2009.

[127] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. GraphQ: Graph Query Processing with Abstraction Refinement—Scalable and Programmable Analytics over Very Large Graphs on a Single PC. In Proceedings of the *USENIX Annual Technical Conference*, Santa Clara, California, July 2015.

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

[129] Wenlei Xie, Guozhang Wang, David Bindel, Alan Demers, and Johannes Gehrke. Fast Iterative Graph Computation with Block Updates. In *Proceedings of the VLDB Endowment*, 6(14), 2013.

[130] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A Block-Centric Framework for Distributed Computation on Real-World Graphs. In *Proceedings of the VLDB Endowment*, 7(14), 2014.

[131] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Effective Techniques for Message Reduction and Load Balancing in Distributed Graph Computation. In Proceedings of the *International World Wide Web Conference*, 2015.

[132] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In Proceedings of the *Symposium on Operating Systems Principles,* 2013.

[133] Chang Zhou, Jun Gao, Binbin Sun, and Jeffrey Xu Yu. MOCgraph: Scalable Distributed Graph Processing Using Message Online Computing. In *Proceedings of the VLDB Endowment,* 8(4), 2014.

[134] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In Proceedings of the *USENIX Annual Technical Conference,* Santa Clara, California, July 2015.

[135] Daniel C. Zilio, Calisto Zuzarte, Sam Lightstone, Wenbin Ma, Guy M. Lohman, Roberta J. Cochrane, Hamid Pirahesh, Latha Colby, Eric Alton, Dongming Liang, Jarek Gryz, and Gary Valentin. Recommending Materialized Views and Indexes with the IBM DB2 Design Advisor. In Proceedings of the *International Conference on Autonomic Computing,* New York, New York, May 2004.