Transaction chains: achieving serializability with low latency in geo-distributed storage systems

Yang Zhang*, Russell Power*, Siyuan Zhou*, Yair Sovran*, Marcos K. Aguilera‡, Jinyang Li*
*New York University ‡Microsoft Research Silicon Valley

Abstract

Currently, users of geo-distributed storage systems face a hard choice between having serializable transactions with high latency, or limited or no transactions with low latency. We show that it is possible to obtain both serializable transactions and low latency, under two conditions. First, transactions are known ahead of time, permitting an a priori static analysis of conflicts. Second, transactions are structured as *transaction chains* consisting of a sequence of hops, each hop modifying data at one server. To demonstrate this idea, we built Lynx, a geo-distributed storage system that offers transaction chains, secondary indexes, materialized join views, and geo-replication. Lynx uses static analysis to determine if each hop can execute separately while preserving serializability—if so, a client needs wait only for the first hop to complete, which occurs quickly. To evaluate Lynx, we built three applications: an auction service, a Twitter-like microblogging site and a social networking site. These applications successfully use chains to achieve low latency operation and good throughput.

1 Introduction

Many Web applications rely on *geo-distributed* storage systems, such as Cassandra [2], Megastore [10] and Spanner [22]. These systems hold the promise of both high availability (by replicating data across datacenters) and low latency (by placing data close to clients). A useful feature of storage systems is serializable transactions, which group many read/write operations to ensure consistency despite failures and concurrency. Unfortunately, existing mechanisms to provide transactions [12] are expensive for a geo-distributed setting, incurring inter-datacenter delays of up to hundreds of milliseconds.

SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522729

Studies done at Google and Amazon show that Web users are sensitive to latency [46]: even a 100ms increase in latency causes measurable revenue losses. It is therefore important to reduce the latency of transactions as much as possible. A common way to achieve low-latency is to drop serializability [28] and offer relaxed consistency (e.g. causal+ [39, 40], PSI [50], Red/Blue [38], HAT [9]). Many systems with weakened consistency also have other limitations: some systems require all data to be replicated at all datacenters [38–40], while others [38,50] lack a scalable design within a datacenter.

It turns out that giving up serializability for low-latency is unnecessary. This claim is predicated on two observations. First, typical Web applications run a predefined set of transactions, so it is possible to perform a global static analysis of its transactions before execution, to find opportunities to execute them quickly without violating serializability. Second, one can decompose a general (geo-distributed) transaction into a sequence of hops, each modifying data in only one server. With the aid of static analysis, one can safely run these hops as separate transactions while preserving serializability, and return quickly to clients after the first hop (often in the local datacenter).

Using these ideas we built Lynx, a geo-distributed storage system that provides serializability with low latency. To scale, Lynx partitions tables into many shards, each possibly replicated in a subset of datacenters. Lynx provides a new primitive called *transaction chain* or simply *chain*. A chain is a sequence of hops, each accessing data on one server, such that all hops execute *exactly once* or none of them do, similar to the notion of a saga [30]. Applications submit transactions to Lynx as chains; Lynx also uses chains internally to update secondary indexes, materialized joins, and geo-distributed replicas.

Prior to application execution, Lynx performs a global static analysis of its transaction chains. The analysis determines if it is possible to execute each chain *piecewise*—that is, as a series of local transactions, one per hop—while preserving serializability of the entire chain. The analysis uses the theory of transaction chopping [48] to construct a graph based on the operations within the trans-

actions. Lynx has two ways to enhance the opportunity for piecewise execution. First, Lynx lets programmers provide annotations about the commutativity of pairs of hops that would otherwise be considered to conflict. Second, when chains are executed piecewise, Lynx ensures *origin ordering*: if chains T_1 and T_2 start at the same server, and T_1 starts before T_2, then T_1 executes before T_2 at every server where they both execute. This property eliminates many conflicts in the internal chains that Lynx uses for updating secondary indexes and join tables.

Lynx has some limitations. First, it does not reduce the total execution time of a chain; rather, Lynx can return control to the application after the chain's first hop. The first hop is often fast: it commonly executes in the local datacenter and writes some internal metadata to a nearby datacenter (for disaster tolerance), which adds only milliseconds of delay. This low first-hop latency does not benefit all applications, but we believe that it helps many Web applications where users interact—for instance, by sending friendship requests, posting messages on walls, etc. These operations are well served by a chain whose first hop modifies the user's own data, while later hops modify other users' data in the background. The second limitation is that Lynx cannot execute all chains piecewise to attain low first-hop latency: the static analysis may force some chains to execute as distributed transactions. The third limitation is that Lynx does not guarantee external consistency or order-preserving serializability [32, 54], but to compensate Lynx provides the guarantee of read-my-writes within a session [52].

Using Lynx, we built three Web applications: an auction service ported from the RUBiS benchmark [1, 7]; a Twitter-like microblogging service; and a Facebook-like social networking site. These applications were easy to build using Lynx's API, and they benefit from piecewise chains. Experiments running on three EC2 availability regions show that these applications achieve low latency with good throughput, and Lynx scales well with the number of servers.

2 Overview

Setting. Lynx is a geo-distributed storage system for large Web applications, such as social networks, Web-based email, or online auctions. Lynx scales by partitioning data into many shards spread across machines. Each shard can be geo-replicated at many datacenters, based on requirements of locality, durability, and availability. Unlike other systems [38–40], Lynx does not require that all datacenters replicate all data, so Lynx can have many datacenters with low replication cost.

Data model and usage. Application developers define a set of schematized relational tables [22] sharded based on their primary key. Lynx provides general transactions in the form of chains, and all operations are performed using chains. API details are given in Section 5.1.

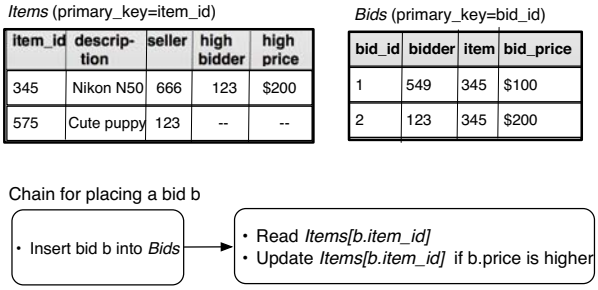

Figure 1: Example schema for a simple auction service and a chain for placing a bid.

We illustrate how applications can use Lynx with an example from RuBIS [1], a simple online auction service modeled after eBay. RuBIS stores data in many tables; two are shown in Figure 1. The *Items* table stores each item on sale with its item id, current highest bid, and user who placed that bid. The *Bids* table stores item ids that received a bid, the bid amounts, and the bidders.

The RuBIS developers denormalized the schema to duplicate the highest bid in the *Items* table, to improve the performance of a common operation: display the current highest bid price of an item. When a user places a new bid, RuBIS must insert the bid into *Bids* and update the corresponding high price in *Items* in the same transaction to ensure consistency. With Lynx, programmers write such a transaction as a chain (Figure 1, bottom).

Lynx supports *derived tables*—tables whose contents are automatically derived from *base tables*—for speeding up queries or safeguarding data. There are three types of derived tables: secondary indexes, materialized join views, and geo-replicas. For example, RuBIS has a secondary index on the item_id of *Bids*, to quickly find the bidding history of an item. Derived tables are themselves sharded according to their key (secondary index key, join key, or replicated primary key) and spread across machines. When base tables change, Lynx automatically issues sub-chains to update the derived tables. These sub-chains are called *system chains*, while *user chains* are written by application developers.

Before application deployment, Lynx performs a static analysis of all application chains to determine if Lynx can execute each chain *piecewise*—one hop at a time—while ensuring the entire chain and its sub-chains are serializable as a single transaction.

Features. In summary, Lynx has the following features:

- *Serializability*. Given an application and its chains, Lynx ensures that concurrent execution of those chains preserve serializability.
- *Low latency*. For chains that can be executed piecewise, applications can achieve low latency by having

Lynx return control after the first hop, which typically executes in the local datacenter and logs to a nearby datacenter for disaster tolerance. To the best of our knowledge, no prior geo-distributed storage system provides both serializability and low latency.

- *Derived tables*. Automatically updated secondary indexes, materialized join tables, and geo-replicas speed up common application queries.
- *Scalablity.* Lynx scales with the number of machines in a datacenter and with the number of datacenters.

Transaction chains are the fundamental mechanism underlying Lynx; we develop them fully in the next two sections. Section 3 describes the properties of chains. Section 4 explains how to ensure serializability of chains.

3 Transaction chains

A transaction chain accesses data that is distributed over many servers. A chain encodes a transaction T as a sequence of hops T=$[p_1 \dots p_k]$ with each hop p_i executing deterministically at one server, where servers can be at different datacenters and may repeat. A hop may have input parameters that depend on the output of earlier hops in the chain.

It is desirable to execute a chain *piecewise*, which means that hops are executed one after the other as separate transactions. Such execution is efficient, because each hop is contained within a single server, so it can be executed as a local transaction. Chains can also improve perceived application latency, as an application can just wait for a chain's first hop to complete.

Guarantees. Chains have the following properties:

- *Per-hop isolation.* Each hop is serializable with respect to other hops in all chains. This is achieved efficiently by executing a hop as a local transaction.
- *Inner ordering.* Hop p_{i+1} never executes before hop p_i.
- *All-or-nothing atomicity.*[1] If the first hop of a chain commits, then the other hops eventually commit as well. (They may abort due to concurrency control, but in that case the system retries until they commit.) Moreover, if the first hop aborts then no hop commits. Thus, the first hop determines the outcome of the chain.
- *Origin ordering.* If two chains T=$[p_1 ...]$ and T'=$[p'_1 ...]$ start on the same server with p_1 executing before p'_1, then p_i executes before p'_j for every p_i and p'_j that execute on the same server.

When executed piecewise, chains might interleave their execution. Say, if a chain has hops p_1, p_2 and another chain has hops p'_1, p'_2, the system may execute the hops in the order p_1, p'_1, p_2, p'_2. Lynx determines whether such interleavings are serializable (Section 4) and, if not, avoids them by executing the chain as a distributed transaction. Thus, Lynx ensures the following:

[1] called simply *atomicity* in the database community

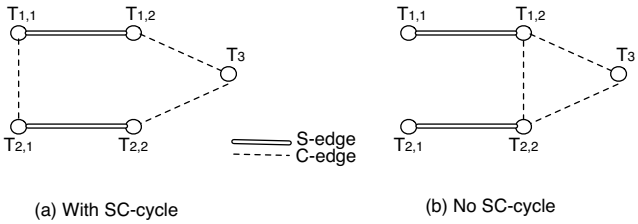

Figure 2: SC-graph analysis for transaction chopping. T_1 **is chopped into** $T_{1,1}$,$T_{1,2}$ **and** T_2 **into** $T_{2,1}, T_{2,2}$**. There is an SC-cycle in graph (a) but not (b).**

- *Serializability.* Chains are serializable as transactions.

Restrictions. A chain has two restrictions. First, application-initiated aborts can occur only at the first hop of a chain (this is needed to implement all-or-nothing atomicity). Second, chains are static: each hop executes at a server that is known when the chain starts (needed to implement origin ordering). Some transactions cannot be structured as chains. These can be executed as a distributed transaction in Lynx.

Linked chains. Applications can link together multiple chains so that they execute consecutively, like a chain of chains, where each chain individually satisfies the properties above. The set of linked chains may not be serialized as one transaction, but Lynx ensures the following atomicity property: if chains are linked and the first chain starts then the other chains eventually start. Like hops in a chain, linked chains can receive inputs from previous chains, and all linked chains must be submitted together.

4 Providing serializability

Web applications typically have an a priori known set of transactions, permitting a global static analysis of the application to determine what chains can be executed piecewise while preserving serializability. If the analysis determines that executing a chain piecewise would violate serializability, Lynx executes the chain as a distributed ACID transaction [12, 22], incurring higher latency. Alternatively, the developer can remove conflicts using annotations or linked chains, as we describe below.

In what follows, we explain how the analysis works (§4.1), how to improve the chances for piecewise execution (§4.2), how to cope with the lack of external consistency (§4.3), and what limitations chains have (§4.4).

4.1 Static analysis of chains

The analysis uses knowledge of the table schemas and the application chains, specifically the table accessed by each hop of each chain and the type of access (read or write). The analysis determines what chains can be executed piecewise while preserving serializability.

The analysis is based on the theory of transaction chopping, originally developed for breaking up large trans-

actions into smaller pieces in centralized database systems [48]. The chopping algorithm takes a set of chopped transactions and constructs a graph, which we call *SC-graph*, where vertices represent transaction pieces and edges represent relationships between pieces. There are two types of edges: S-edges connect vertices of the same unchopped transaction, C-edges connect vertices of different transactions if they access the same item and an access is a write. An *SC-cycle* is a simple cycle containing a C-edge and an S-edge (Figure 2). It is shown that serializability is assured if the SC-graph has no SC-cycles [48]. Intuitively, an SC-cycle indicates a non-serializable interleaving. For example, Figure 2(a) allows the problematic interleaving $T_{1,1}$, $T_{2,1}$, $T_{2,2}$, T_3, $T_{1,2}$.[2]

Naive construction of the SC-graph. To apply the theory of transaction chopping in our context, a chain corresponds to a chopped transaction and its hops are the pieces. Thus, in the SC-graph, S-edges connect the hops of a chain, while C-edges mark potential conflicts between hops of different chains. Static analysis cannot determine exactly what data items a hop accesses (which rows); therefore, we conservatively add a C-edge between two hops of different chains if the hops access the same table and an access is a write. Since instances of the same chain may be in conflict (if they update data), the SC-graph includes two instances of every chain that updates data;[3] for read-only chains, one instance suffices. We must also consider system sub-chains caused by user chains (recall that system chains are automatically created to update derived tables when base tables change); we want these sub-chains to be serialized with the originating chain. A simple idea is to combine a user chain and its sub-chains in the SC-graph: when a user chain hop modifies a base table, the hop is expanded into the sub-chains that update derived tables. Later, in Section 4.2, we improve on this simple idea.

As an example, consider the auction application from Section 2 (Figure 1), with the three chains: T_{bid} for placing a bid, T_{item} for adding an item to be auctioned, and T_{read} for browsing an item. T_{bid} has two hops, while the others have one hop. For simplicity, let us ignore the system chains. Figure 3 shows the resulting SC-graph. There is an SC-cycle involving two instances of T_{bid}, so this chain cannot safely execute piecewise.

4.2 Improving chances for piecewise execution

When we naively apply the theory of transaction chopping, we find little opportunity for piecewise execution, because SC-cycles are everywhere! Below, we consider the problems and propose ways to avoid these cycles.

[2]This interleaving is bad because it creates a cycle in the *serialization graph* [54], where T_1 precedes T_2 (as $T_{1,1}$ precedes $T_{2,1}$ in the interleaving), T_2 precedes T_3 (as $T_{2,2}$ precedes T_3), and T_3 precedes T_1 (as T_3 precedes $T_{1,2}$).

[3]Two instances suffice, since an SC-cycle with more than two instances implies an SC-cycle with only two instances.

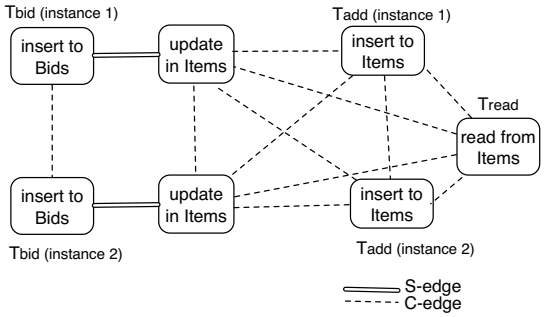

Figure 3: SC-graph for a simple auction service (Figure 1) with three chains: T_{bid}, T_{add}, T_{read}. There are two instances of T_{bid} and T_{add} to account for self-conflict. The graph has an SC-cycle involving the two instances of T_{bid}.

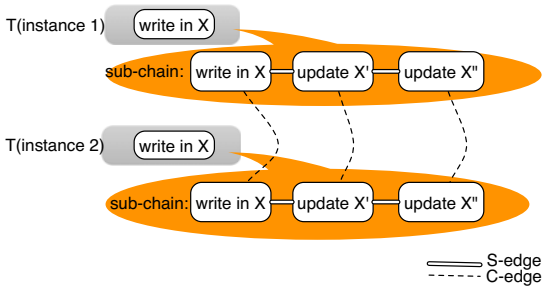

Figure 4: Lynx automatically generates sub-chains to update derived tables X' and X'' of base table X. The sub-chains cause an SC-cycle.

User chains. User chains can have spurious C-edges because the notion of conflict is coarse-grained, being based on table accesses. This problem is exacerbated by self-conflicts between instances of the same chain. In Figure 3, T_{bid} modifies two tables, creating an SC-cycle on its own instances. Closer inspection reveals that the hop "insert to Bids" inserts a row with a unique id; this hop commutes with itself, so it does not self-conflict. Developers can use *annotations* to indicate that the hop self-commutes, which removes the C-edge between its instances, breaking the cycle. Other systems also exploit commutativity [38, 47, 50], but in different ways.

User chains may have unnecessary S-edges: a user chain may have hops that need not be serialized together, but were placed in the same chain because they require all-or-nothing atomicity. In that case, programmers can separate these hops into different chains and execute them as linked chains (Section 3), which also provide all-or-nothing atomicity but avoid S-edges.

System chains. Many self-conflicts arise among the system sub-chains created by Lynx to update derived tables. Figure 4 shows a one-hop user chain that modifies a base

table causing a system chain. Because a chain and its resulting system chains should be serialized together as one transaction, we consider the combined chain in the SC-graph. This chain unfortunately causes an SC-cycle on its two instances, because of self-conflicting hops with updates that do not always commute (Figure 4).

We eliminate these cycles using the *origin ordering* guarantee of chains. Specifically, sub-chains updating identical rows in derived tables either commute or start by updating the same base table row at the same server. In the latter case, origin ordering ensures that these sub-chains are consistently ordered and thus need not be connected in the SC-graph. Note that origin ordering cannot eliminate C-edges in *user chains*, because the static analysis cannot determine if two user chains start at the same server: that depends on what table shard they access, which may be determined only at run-time.

Complete construction of the SC-graph. With the above ideas, we modify the naive construction of the SC-graph (Section 4.1) as follows. First, we omit system chains and only consider user chains when adding C-edges. A user chain may read from derived tables but can never directly modify them. Thus, two hops from different (instances of) user chains have a C-edge between them iff (1) both hops access the same base table and an access is a write, or (2) one hop reads from a derived table T and the other hop modifies a base table from which T derives. Additionally, if two hops are annotated as commutative, we do not add a C-edge between them. Finally, chains that are linked are included as separate chains in the SC-graph; the fact they are linked does not affect the SC-graph.

4.3 A word on preserving order

The techniques we described do not ensure external consistency or order-preserving serializability [32, 54]. Order-preserving serializability requires that if a transaction commits before another one starts, the first appears before the latter in the equivalent serial order. The analogous property for chains does not hold: a client may submit chain T_2 after chain T_1 returns (after committing T_1's first hop), but T_2 may be serialized before T_1.

There are two ways to address this issue, if necessary. First, there is a barrier operation that blocks a client until its outstanding chains complete. This is analogous to memory barriers in multiprocessor systems, which allow programmers to enforce ordering when necessary. For example, the operation to change a user's privacy settings should be followed by a barrier. Doing so is akin to enforcing application-defined explicit causality rather than every possible causality [8]. Second, we can provide the simple guarantee of read-my-writes [52], which in our setting ensures that a client sees the entire effects of her previous chains (even if they return early), a useful property in practice. We explain how Lynx ensures this property in Section 6.2.

4.4 Restrictions and typical usage

Transaction chains can reduce user-perceived latency but there are some restrictions on its use. First, programmers must explicitly divide a transaction into a chain such that (1) only its first hop contains a user-initiated abort and (2) the chain is *static* in that the shards it accesses at each hop are known before the chain starts executing. This is akin to requiring transactions to have known read and write sets, so one might apply the ideas of [53] to systematically transform a general transaction into a static one. Second, to achieve low latency, programmers must design the chains so that, most of the time, the application can proceed after the chains complete their first hop (or first few hops). As discussed earlier, returning after the first hop may result in the loss of external consistency and, if misused, can generate user-perceived anomalies.

Having discussed the restrictions, we describe our experience in using transaction chains for Web applications. We focus on Web applications where users interact, which require scalability and low latency. In such applications, we recommend co-locating data owned by the same user in the same datacenter (possibly with geo-replication). To process a typical user request, one uses a transaction chain which first modifies a user's own data and then updates other users' data or global data. We give two examples.

First, in a social networking application, suppose that user X posts a message on the wall of a friend Y. To execute this request, a transaction chain first modifies X's data by inserting X's message in the message table, and then updates Y's data by inserting the message id into Y's wall in the wall table. As a second example, in Figure 1 the chain for placing a bid first inserts the user's bid into the bid table and then changes global information by updating the high price in the items table.

Since an application usually processes a request at the datacenter that stores the requesting user's data, a chain's first hop can complete quickly. In both examples, the application returns control to the user after the first hop. The lack of external consistency is partly compensated by the optional read-my-writes guarantee of chains: in the first example, with read-my-writes user X is guaranteed to see her own message when she browses Y's wall. However, unlike the external consistency guarantee, if X tells Y about her message using external channels (*e.g.*, the phone) and Y checks his wall, Y may not see X's message. This is an anomaly that applications must tolerate when taking advantage of transaction chain's low latency.

```
CREATE ENTITY_GROUP UserEnt {key int};

CREATE TABLE Bids IN_GROUP UserEnt {
   bidder ALIAS UserEnt.key,
   bid_id       int AUTOINCREMENT,
   seller       int,
   item_id      int,
   price        float
} PRIMARY_KEY(bidder, bid_id);
```

Figure 5: Syntax for defining the *Bids* base table, whose rows are co-located with those from other tables in the same (UserEnt) entity group.

```
//a materialized view joining Bids and Users
//on Bids.bidder = Users.uid
CREATE DTABLE Bids-Users IN_GROUP UserEnt
              FROM Bids, Users {
 bidder ALIAS UserEnt.key  <-- Bids.bidder,
 bid_id                    <-- Bids.bid_id,
 bidder_name               <-- Users.name,
 seller                    <-- Bids.seller,
} JOIN(Bids.bidder = Users.uid);

// secondary index for Bids-Users indexed by seller
CREATE DTABLE Bids-Users_seller IN_GROUP UserEnt
              FROM Bids-Users {
 seller ALIAS UserEnt.key <-- Bids-Users.seller,
 bidder           <-- Bids-Users.bidder,
 bid_id           <-- Bids-Users.bid_id
 bidder_name      <-- Bids-Users.bidder_name,
} INDEX_KEY(seller);
```

Figure 6: Syntax for defining derived tables. The join table *Bids-Users* unites *Bids* and *Users* tables with the join key *Bids.bidder*. The secondary index table *Bids-Users_seller* further indexes the join table on the seller column.

```
1  //chain definition
2  place_bid = new Lynx.tx_chain;
3  place_bid.add_hop('insert_bid',
4    function(ctx) {
5      var row = @Bids.insert(ctx.args.bidder,
6                             ctx.args.item_id, ...);
7      ctx.bid_id = row.bid_id;
8    }
9  );
10 place_bid.add_hop('update_price',
11   function(ctx) {
12      var seller = ctx.args.seller;
13      var id = ctx.args.item_id;
14      var curr_price = @Items.lookup(seller, id).price;
15      if (price > curr_price) {
16         @Items.update(seller, id).price = price;
17      }
18 );
19 //commutativity annotation
20 Lynx.commutes(place_bid.hops['insert_bid'], @self);
21 Lynx.commutes(place_bid.hops['update_price'], @self);
22
23 //chain execution
24 place_bid.execute({
25   args : {
26     bidder : 9999,
27     seller : 8888,
28     item_id : 123,
29     price : 1.09
30   },
31   //chain is in Session associated with user id 9999
32   session : UserSession[9999],
33   return_after_first : true
34 });
```

Figure 7: JavaScript API for writing a user chain. The example shows the chain for placing a bid in the auction service.

5 Lynx Architecture

We give an overview of Lynx's system design. We first explain its interface to applications (§5.1), then describe its system architecture (§5.2).

5.1 Programming interface

Lynx's API consists of a simple language for describing table schemas, and a client-side library for writing chains.

Creating tables. Programmers use a SQL-like syntax to define table schemas. Tables are partitioned by rows according to their primary keys. Programmers can provide hints for co-locating partitions from different tables using entity groups [10, 22].

Figure 5 shows the *Bids* table schema for the auction example of Figure 1. The CREATE TABLE…IN_GROUP syntax creates a table co-located with the given entity group. The table inherits the key of the entity group as a column, which can be renamed using ALIAS. The entity key must be part of the table's primary key. Here, each row of *Bids* is co-located with the user placing the bid.

Figure 6 shows how to define derived tables for secondary indexes and materialized join views. *Bids-Users* is a join table that unites tables *Bids* and *Users* on the join key *Bids.bidder*. *Bids-Users_seller* is a secondary index table for the join table on the *seller* column. This table allows one to find the names of bidders who placed bids on items sold by a given user. The <-- syntax serves to copy a column from the base table. Currently, Lynx supports only joins based on equality of indexed keys.

Creating and using chains. All operations are performed using chains. Figure 7 shows the chain for placing a bid using Lynx's JavaScript API. The chain has two hops, one to insert the bid (line 3) and another to update the current highest bid price of the item (line 10). Each hop has access to the chain's context (ctx) which contains input arguments of the chain. Lynx exposes relational tables as auto-generated table objects whose names start with '@'. This syntax simplifies the static analysis tool that generates the SC-graph. Since '@' is not allowed in JavaScript identifiers, it is removed before execution.

Programmers can read or write base tables (e.g., line 5 and 14); derived tables are updated only by the system. Programmers can specify commutative relationships (lines 20–21 specify hops that self commute). When executing a chain, programmers can optionally indicate a *session* for the chain (line 31). Lynx ensures that chains in a session see the writes of chains in the same session that have already returned (read-my-writes). We explain

how Lynx provides this guarantee in Section 6.2.

5.2 System Overview

A Lynx system consists of a number of geo-distributed datacenters, each of which contains many machines. A machine runs many logical Lynx servers in the same process. This improves concurrency as having more (logical) servers imposes fewer constraints under origin ordering. The rows of a table are partitioned into *shards* based on row keys; that is, a shard is a set of rows of a table. The rows of a shard are replicated across the same set of servers, as we now explain.

Geo-replication. Data shards can have geo-replicas across data centers. Geo-replicas are configured by a configuration service that assigns each shard to a *replica group*, which consists of a set of Lynx servers spread across datacenters. Geo-replication across data centers is implemented by Lynx using system chains as explained in Section 6. To avoid having conflicting updates at different replicas, Lynx uses home geo-replicas, similar to Walter [50]: each replica group has a designated server called the *home geo-replica* or *home server*, and the system forwards all updates on a shard to its home geo-replica. The home geo-replica can be chosen intelligently to be the server where updates are most likely to occur. For example, a Web application may have a replica group for each user, where the home geo-replica is in a datacenter close to the user.

Local replication and cluster storage system. Data shards may also be replicated *within* a datacenter to provide fast fail-over. This replication is provided by a *cluster storage system* that provides synchronous updates and transparent failover; such a service is implemented using well-known techniques (*e.g.*, [14, 31]).

Lynx also uses the cluster storage system to synchronously replicate internal metadata across *buddy datacenters*. Two datacenters are buddies if they are near enough to communicate with low latency, yet far enough so that one datacenter is safe from a disaster that affects the other. For example, this criterion may be met by datacenters that are a few hundred miles apart with roundtrip latencies of several ms, which is comparable to disk latencies. Lynx relies on buddies only to geo-replicate some internal metadata; application data can be geo-replicated using chains across *any* datacenters chosen by the developer, not just buddies.

Configuration service. Lynx relies on a separate configuration service to maintain the mapping from each shard to its replica group. Our design of this service follows other systems [18, 50, 51]. Nodes consult the service to determine the server responsible for a given shard. This information is subsequently cached. Each server obtains a lease for its responsible shards and rejects requests destined for other shards. The configuration service itself is implemented via a Paxos replicated state machine.

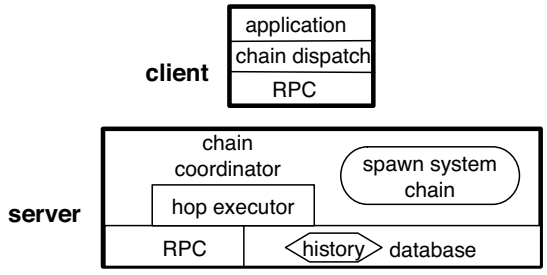

Figure 8: Lynx client library and server processes. The client dispatches chains using RPCs. The server process receives chains, queues them, and executes them against a local database. The server process also implements geo-replication, secondary indexes, and materialized join views using system chains.

Chain analysis. Prior to application execution, Lynx statically analyzes chains based on application code and table schemas (§4.1). The analysis outputs SC-cycles, if any. Programmers can use this information to add annotations or use linked chains to break the cycles (§4.2).

6 Chain execution in Lynx

We now describe how chains work at runtime. We give an overview of the implementation (§6.1), and then explain the details on how Lynx ensures the various chain properties (§6.2) and how it uses system chains (§6.3).

6.1 Overview

Chains are implemented by the Lynx client library and server process (Figure 8). The client dispatches a chain to its first hop, at a server storing the data accessed by the hop. If the first hop writes data, the client chooses the server in the shard's home datacenter; otherwise, it chooses a server in a nearby datacenter that has a replica.

The first server of a chain coordinates its execution in a coordinator thread. The coordinator first stores information about the chain in its *history table* kept in the cluster storage system. The history table keeps the chain id, the chain parameters from the client, and the origin ordering sequencers (§6.2). The coordinator may execute the chain piecewise or as a distributed transaction.

To execute the chain piecewise, the coordinator serially executes each hop of the chain, by invoking the appropriate server (the first server is local) and waiting for a completion acknowledgement. After the first server executes its hop, the coordinator returns an indication of first-hop completion to the client library. Then, if the server executed a hop that modified data, it spawns in parallel sub-chains to update derived tables, if any. These sub-chains are coordinated by the server and execute like any other chain—in particular, Lynx ensures origin ordering based on where the sub-chains start. The server waits

Property	Technique
per-hop isolation	local database transactions
all-or-nothing atomicity	chain replay and history table
inner ordering	serial execution
origin ordering	pairwise sequencers
read-my-writes	origin ordering and read sub-chains
linked chain atomicity	super-coordinator

Figure 9: Techniques used to ensure the chain properties using piecewise execution.

for the sub-chains to complete before sending an ack to the coordinator of the higher-level chain.

If a chain cannot execute piecewise, the coordinator executes it as a distributed transaction using standard two-phase locking and two-phase commit [12, 22].

6.2 Providing chain properties

We now explain how Lynx provides the properties of chains (§3) when chains execute piecewise. Figure 9 gives a summary. These techniques are efficient as they require little or no coordination across servers.

Per-hop isolation. Lynx stores each shard at one server. Because each hop of a chain accesses one shard, we can ensure per-hop isolation by simply executing it using a local serializable database transaction. Our current implementation requires shards to fit on a single machine, but it is possible to generalize this to split a shard among several machines and substitute local transactions with distributed transactions within a single datacenter.

All-or-nothing atomicity. If the first hop of chain commits, subsequent hops are executed exactly once despite failures. Lynx ensures this property by replaying chains that stop due to failures, using history tables to prevent duplicate execution, as we now explain.

Recall that a coordinator orchestrates the execution of a chain. We must address three failure types that break chain execution: (1) crashes of a Lynx server, (2) crashes of the coordinator, and (3) failures of an entire datacenter.

(1) A Lynx server crashes while executing a hop. In this case, the system recovers the server as described in the next paragraph, and the coordinator resubmits the hop for execution. To avoid duplicate execution, every Lynx server keeps a *history table*, similar to [44]. This table is kept in the same storage system as the server's tables; it records, for every hop that the server completes, its chain id, hop number, and any output produced by the hop to be passed forward in the chain. To be consistent, the history table is updated *using the same transaction that updates the server tables during the hop execution.* Before executing a hop, each server checks its history table to see whether the hop has already executed and, if so, skips execution. This checking is also done in the same transaction that updates the history table.

The server then notifies the coordinator that the hop is done, attaching the hop's output. The server deletes the hop entry from its history table when it gets an acknowledgement from the coordinator. The coordinator updates the current progress of the chain in its history table; it deletes the chain's entry after the entire chain completes.

To recover a Lynx server, the system can optionally store the server's data in a cluster storage system within the datacenter. In that case, recovery is simple: the system starts a new server and reconfigures the replica group to replace the old server with the new one. The new Lynx server operates on the same data as the old server using the cluster storage system.

If the Lynx server does not use the cluster storage system, or the cluster storage system is crashed, then recovery relies on geo-replication and reconstruction. Before using a geo-replica, the system must ensure it is up-to-date, by restarting and waiting for the completion of any replication sub-chains that might be coordinated by the failed Lynx server; how this is done is explained in (2) below. Derived tables might not be geo-replicated; these tables are reconstructed using the base tables. Then, the system reconfigures the replica groups to replace the failed server with a server holding the geo-replicas or reconstructed tables.

(2) The coordinator crashes while executing a chain. In this case, the system restarts the coordinator at another host. The new coordinator determines the outstanding chains using the history table of the previous coordinator, which is kept in the cluster storage system. To handle datacenter failures (see below), the coordinator's cluster storage system is geo-replicated at buddy datacenters (§5.2). (Note that the cluster storage of the coordinator is separate from the cluster storage of a Lynx server—only the former uses buddies; the latter is contained in a single datacenter.) For each outstanding chain, the new coordinator replays the chain from its first hop, executing one hop at a time using the origin ordering sequencers stored in the history table. Servers that already executed the chain avoid duplicate execution as explained above.

(3) An entire datacenter is destroyed or becomes unavailable beyond a time threshold. In this case, the system first recovers the Lynx servers using geo-replicas and reconstruction, as described in (1). Then, the system recovers from crashed coordinators, as described in (2).

Inner ordering. This property is provided by executing hops in the order in which they appear in the chain.

Origin ordering. A naive way to provide this property would be for coordinators to execute one chain entirely before starting the next chain. This scheme has low concurrency and poor performance.

Instead, we use pairwise sequencers: each server i keeps n counters $ctr_{i\to 1}...ctr_{i\to n}$, where n is the num-

ber of servers in the system. Server i also keep tracks of the latest sequence number that it has processed from each other server, $done_{1\to i}...done_{n\to i}$. Suppose a chain with k hops is to execute on servers $s_1, s_2, ..., s_k$. The first server, s_1, increments the respective counters $ctr_{s_1\to s_1}, ctr_{s_1\to s_2}, ..., ctr_{s_1\to s_k}$ for each hop of the chain and attaches them to the chain as sequence numbers $seq_{s_1\to s_1}, seq_{s_1\to s_2}, ..., seq_{s_1\to s_k}$. Each of the servers s_i waits until its counter $done_{s_1\to s_i}$ reaches $seq_{s_1\to s_i}-1$ before executing its corresponding hop in the chain.

This mechanism ensures origin ordering: suppose chains C_1 and C_2 start at the same server i and both execute later hops at server j. If C_1 executes before C_2 at server i, the sequence number $seq_{i\to j}$ of chain C_2 is greater than that of C_1, causing C_2 to execute after C_1 at server j. If a chain visits some server i multiple times, the hops at i will be assigned consecutive sequence numbers and thus will not be interleaved with other chains, thereby preserving the origin ordering property.

The message overhead for enforcing origin ordering is low: the number of sequence numbers attached to a chain is proportional to its length. Origin order may sometimes introduce latency overheads, but this is the behavior we desire for consistency. Specifically, if two chains start at the same server and follow different paths before overlapping again at another server, the first chain may delay the second chain.

Read-my-writes in sessions. This property ensures that a chain in a session sees the writes of chains in the same session that have already returned. To do so, the application associates a session with a server, and Lynx forces all session chains to start at that server by adding a no-op first hop if necessary. A possible optimization in practice is to pick a server where most session chains start anyways, to avoid adding the no-op hop. If a session chain reads from a base table, then origin ordering ensures the read-my-writes property. If a session chain reads from a derived table, Lynx executes the read hop differently from a regular chain: Lynx submits the read hop at the base table, which then starts a sub-chain to read the derived table. By doing so, the read of the derived table is ordered consistently with the operations on the base table, which in turn are correctly ordered by origin ordering. If a derived table has two base tables (a join table), Lynx submits the read at each base table in some arbitrary order and keeps the result of the later read.

Atomicity of linked chains. To execute a series of linked chains, the coordinator of the first chain serves as a super-coordinator. The super-coordinator stores the linked chains in its history table, for recovery, and then launches the chains one at a time at their first hop. When the chain completes, the super-coordinator marks completion in the history table. If the super-coordinator fails, recovery

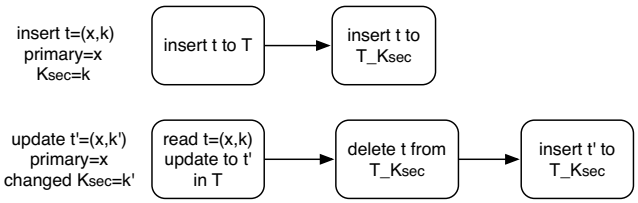

Figure 10: The chains for inserting a new row and updating an existing row's secondary index. Base table T has a secondary index table T_K$_{sec}$.

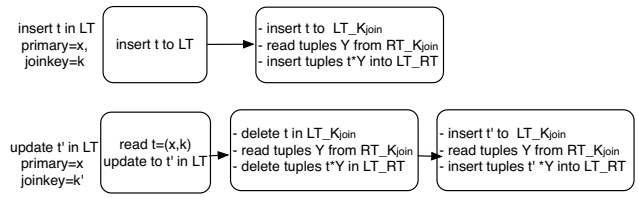

Figure 11: The chains for inserting a new row and updating an existing row's join key value. Base tables LT and RT have secondary index tables, LT_K_{join}, RT_K_{join} (corresponding to the join key K_{join}) and a join table LT-RT.

is similar to that of a coordinator.

6.3 System chains

Recall that system chains are generated internally by Lynx to update derived tables. There are three types of system chains, one for each type of derived table.

Chains for geo-replication. When a hop of the chain wishes to modify a geo-replicated base or derived table, the hop is forwarded to the corresponding shard's home datacenter for execution. The responsible server at the home datacenter generates a sub-chain to propagate the modification to replicas at other datacenters. Because of the origin ordering property of these sub-chains, all replicas are updated in the same order.

Chains for secondary index tables. When a row is inserted, deleted, or updated in a base table, the server where the modification occurred spawns a sub-chain to modify the index tables. (If an index table is geo-replicated, the corresponding server at the home datacenter generates additional sub-chains for geo-replication.) The sub-chain has one or two hops for *each* index table: if the indexed value does not change, one hop suffices to update the index table; if the indexed value changes, the old and new rows of the index table may belong to different shards, in which case two hops are needed, one to delete the old row, the other to insert the new row. Figure 10's top chain shows the case where only one hop is needed.

Chains for join views. To update materialized join views, we apply ideas from incremental join view update algorithms [13], using chains to correctly update the

views. Figure 11 shows the sub-chains for updating the derived table *LT-RT*, which joins two base tables *LT* and *RT* on join key K_{join}. We assume that the join key K_{join} is not the primary key of *LT* or *RT* (the case when the join key is a primary key is simpler). Therefore, in order to create the join view, programmers are required to add index tables (LT_K_{join}, RT_K_{join}) indexing the join key. For updating a join view, there are two cases depending on whether the base table modification changes the existing value of the join key column. The top chain of Figure 11 illustrates the case when no existing value of the join key column is changed with an insert operation to the base table *LT*. In this case, the sub-chain updates both *LT*'s secondary index table for the join key (LT_K_{join}) and the join table *LT-RT* using a local read-write transaction. The use of a local transaction is possible because the affected rows of the index and join tables LT_K_{join}, RT_K_{join} and *LT-RT* are co-located in the same shard. The bottom chain of Figure 11 is generated when the existing value of the join key column is changed. In this case, two additional hops are required to maintain *LT-RT*, one to delete the existing value, another to add the new value.

The join table may also have other index tables derived from it. In this case, Lynx spawns parallel sub-chains that start from the updated join table shard and update those index tables.

The correctness of the join process is assured by two features of the chains. First, with origin ordering, modifications on the same row of *LT* interleave correctly. Second, with per-hop isolation, the local read-write transaction updating LT_K_{join}, RT_K_{join}, and *LT-RT* ensures that *LT-RT* is always the join of the secondary indexes LT_K_{join} and RT_K_{join}. This reduces the correctness of updating the join table to the correctness of updating secondary indexes, which is evident.

7 Implementation of Lynx

The Lynx server and client library consist of ≈5000 lines of C++ code, plus 3500 lines for a custom RPC library. Programmers specify user chains using Lynx's JavaScript API; a Lynx utility reads the application table schemas and generates JavaScript objects that programmers use to read and update each table. When executing a user chain, the coordinator transfers the JavaScript code of each hop to the appropriate server, which then caches and executes the code using the V8 JavaScript engine.

The implementation stores tables in a custom storage system rather than a local database system. The custom system keeps tables in memory with transactional logging to stable storage.

Our current prototype misses four pieces from the design. First, it lacks the configuration service, instead relying on a static configuration file to indicate what server has what shards. Second, a Lynx server and coordinator have their stable storage on a local disk, not a cluster storage system. Third, our prototype does not yet implement the recovery protocol (Section 6.2) for handling server or datacenter failures. Fourth, there is no implementation for executing a chain as a distributed transaction.

8 Applications

We implemented three applications using Lynx: a social network website (L-Social), a microblogging service (L-Twitter), and an auction service (L-RUBiS). The applications use secondary indexes and join views extensively, and all of their chains can execute piecewise. This required modifying some chains slightly (while retaining the same behavior). In particular, a user chain which reads a base table and its derived table creates an SC-cycle. We addressed this by duplicating the needed columns of the base table in the derived table, so a user chain needs to only read the derived table.

Social networking. The L-Social application implements the basic operations of a website like Facebook (*e.g.*, befriending users, posting to walls). L-Social has 5 base tables: *Graph*, *Status*, *Users*, *Wall*, *Activities*. There is one join table *GraphActivities* with a secondary index to allow a user to read her friends' activities quickly, with one lookup to the secondary index.

To befriend users A and B, the application must create two friendship edges and two new-friend activity announcements, one for each user. A naive design uses a chain with four user hops, two for inserting into *Graph*, two for inserting into *Activities*. This chain creates an SC-cycle with C-edges from each *Graph* insertion hop to the one-hop read-activity chain that reads the secondary index of *GraphActivities*. To avoid this cycle, we break the befriend chain into three linked chains: one chain inserts the friendship edges, two chains each insert once into *Activities*. The first chain still has an SC-cycle with the unfriend chain and itself. We break this cycle by making the insertion/deletion of friendship edges a commutative operation: we use a counter column in the *Graph* table, and we increment/decrement the counter to insert/delete edges. This is similar to the counting sets in Walter [50].

When user A posts a status message, L-Social uses a chain to insert the message into *Status* and add the announcement "A has changed her status" to *Activities*. Both hops commute with themselves. A similar chain is used to post messages on walls. The join table *GraphActivities* allows a user to read the activities of his friends in one hop.

A final static analysis indicates an SC-cycle: the 1-hop read chain to show a user's friends has an SC-cycle with the 2-hop befriend (or unfriend) chain: the read hop has two C-edges, to each hop of the befriend (or unfriend) chain. We use application knowledge to determine that this SC-cycle is spurious: since a user never befriends

himself, the read hop conflicts with at most one of the hops of the befriend chain, so only one of the two C-edges is a real conflict.

Microblogging. L-Twitter is a simple Twitter clone with tables and schemas modeled after [36]. There are three tables: *Users*, *Tweets*, *Graph*. *Graph* differs from L-Social's *Graph* because it captures an asymmetric follower relation.

A common Twitter operation is to show a user's *timeline*—the collection of tweets posted by users that the user follows. Twitter's original implementation on a one-node MySQL server performs a join query between *Graph* and *Tweets* [36]. Twitter's current distributed implementation no longer uses joins, but rather manually maintains the timeline of each user in memcached. L-Twitter follows the original implementation by using a distributed join table *GraphTweets* (replicating only the tweet id not its text) based on the join key *Tweets.creator* = *Graph.followee* with a secondary index on *Graph_follower*. By querying this index, L-Twitter can display a user's timeline by contacting only one server. We chose this much simpler implementation to demonstrate the materialized joins of Lynx.

There are two limitations in the current design of L-Twitter. First, when user X starts to follow Y, the underlying join chain inserts all of Y's existing tweets into X's timeline (the secondary index of *GraphTweets*). It would be better to insert only Y's recent tweets. This can be done adding a selection operation to the join view, to filter out old tweets with a smaller timestamp than the follow edge timestamp. Supporting such selection operations in Lynx is future work. Second, when a user with many followers tweets, there are large overheads to update their followers' timelines. Thus, L-Twitter's current push-based approach should be combined with pull-based queries for users marked as popular [49].

Auction service. L-RUBiS is a port of the auction website in the RUBiS benchmark [1, 7]. The original RUBiS implementation is based on PHP using a local MySQL database system. We ported the RUBiS schema to Lynx and re-wrote its PHP functions in JavaScript. L-RUBiS has 10 sharded tables with 13 secondary indexes in total, where a table has at most 3 secondary indexes. We use a join table to unite the *User* table, which maps uids to usernames, and the *Comments* table, which records users' comments. This table allows L-RUBiS to quickly find usernames of users who commented on a seller.

There are two noteworthy user chains, one to process bidding requests (discussed in §2), the other to handle new user registration while ensuring unique usernames. In our first design, a register-user chain checks if a chosen username already exists in a secondary index of *User* based on usernames; if not, the second hop inserts the user into the *User* table. This chain has an SC-cycle between two of its instances. We subsequently changed L-RUBiS to use an additional table, *Usernames*, which contains all the usernames that have ever been created. The register-user chain first checks that the chosen username is absent in *Usernames* (and if so inserts it there) and in the second hop adds the user to *Users*. If the chosen username is already taken, the second hop does nothing. The chain still has an SC-cycle with itself, but this cycle is spurious: if two register-user chains conflict on the first hop (due to both having the same username), then one of the chains sees that the username is already taken in its first hop and does nothing in its second hop, so there are no conflicts in the second hop.

9 Evaluation

We measure the performance of Lynx and its applications across geo-distributed datacenters. The highlights are the following:

- Application operations have good throughput and low-latency, despite geo-replication. The first hop of all chains execute quickly, and so user-perceived latency is only a few milliseconds.
- Lynx scales well. As we increase the number of servers in each datacenter from 1 to 8, aggregate chain throughput grows by a factor of more than 6.

9.1 Experimental setup

We perform experiments on Amazon EC2 using three availability regions, East Coast, West Coast and Europe, with the following roundtrip latencies between them:

	West Coast	Europe
East Coast	82ms	102ms
West Coast		153ms

Unless otherwise stated, in all experiments each region has 4 Lynx servers and 4 client machines, where a machine is an extra-large instance with 15GB of RAM and 4 virtual cores. The geo-replication factor is two datacenters. We perform three runs for each experiment and report the average. (Standard deviations were low.)

9.2 Microbenchmark

We evaluate three types of chains. In the simple-n experiments, a client operation is a chain with n hops, each inserting a row into a different base table. In the secondary index experiment, a client operation inserts a row into a table with a secondary index, resulting in a system chain of 2 hops. In the join experiment, a client operation inserts a row into the *LT* base table which has both a secondary index table and a join table (with another base table). In all chains, the first hop executes in the local datacenter and the subsequent hops execute in different remote datacenters. All chains run only C++ code at servers.

We perform two sets of experiments, one without geo-replication, one with geo-replication factor of two. Even

Chain type	NO GEO-REPLICATION Throughput (K chains/s)	First-hop lat. (50%; 99%)	Completion lat. (50%; 99%)	GEO-REPLICATION AT 2 DATACENTERS Throughput (K chains/s)	First-hop lat. (50%; 99%)	Completion lat. (50%; 99%)
simple-1	3,570	3.1ms; 3.3ms	3.1ms; 3.3ms	1,770	3.1ms; 3.6ms	84ms; 90ms
simple-2	1,630	3.1ms; 3.4ms	86ms; 88ms	872	3ms; 3.8ms	266ms; 283ms
simple-3	1,190	3.2ms; 3.3ms	253ms; 257ms	512	3.1ms; 3.8ms	607ms; 656ms
secondary index	1,220	3.1ms; 3.4ms	84ms; 88ms	590	3ms; 3.3ms	258ms; 291ms
join	808	3.1ms; 3.4ms	89ms; 99ms	453	2.8ms; 3.3ms	268ms; 299ms

Table 1: Microbenchmark throughput and latency results.

in experiments without geo-replication, data is spread over the three EC2 regions.

Chain throughput. Table 1 shows Lynx's throughput in thousands of chains/s. We first examine the experiments without geo-replication (left of table). The simple-1 experiment provides a baseline aggregate throughput of 3,570K chains/s using 12 servers in 3 datacenters. We expect the throughput of simple chains with m hops to be $\approx 1/m$ the throughput of a 1-hop chain. The experiments confirm this. Throughput drops by over half going from simple-1 to simple-2 because in simple-1 only clients forward chains whereas in simple-2 servers also do that.

The system chain for updating the secondary index table has two hops and its aggregate throughput is 1,220K chains/s. This is lower than simple-2 because of the overhead of checking if a table modification needs a system sub-chain and if so, coordinating the system sub-chain. The throughput of the join experiment is 808K chains/s, much lower than in the secondary index experiment, even though both chains have two hops. This is because the second hop of the join chain requires more computation: it reads rows from the *RT* table and inserts them into the join table, all in a local transaction (Figure 11). In the experiments, we pre-populated the *RT* base table so that there are 6 rows to be read and inserted into the join table every time a chain modifies a single row in *LT*.

Chain latency. Table 1 shows the median and 99-percentile latency for completing both the first hop and the entire chain. The experiments were done under low load and we measured the latency of chains starting in the West Coast. Since the first hop of a chain executes in the local datacenter, first-hop latency is below 4 ms (99-percentile) across workloads. This latency number is optimistic for two reasons. First, it does not reflect disk latency: although our server implementation synchronously writes its log to disk, the disk latency is absorbed by on-disk caching which cannot be disabled in EC2. Second, it does not reflect the delay in replicating the chain coordinator's log to a nearby buddy datacenter: our prototype currently logs to the local disk as opposed to a cluster file system.

Compared to the first hop latency, the total completion latency is much longer, as each subsequent hop executes in a different datacenter. For example, the median completion latency for a simple chain of length 2 (no replication) is 86ms, and it grows to 253ms when the length is 3.

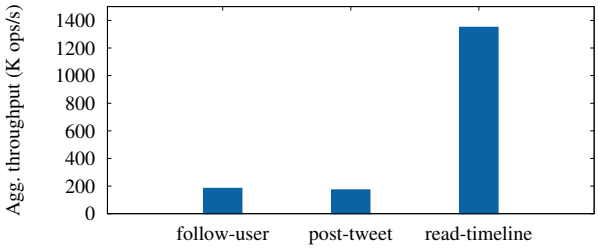

Figure 12: L-Twitter operation throughput

Geo-replication performance. The right part of Table 1 shows experiments where all base and derived tables are geo-replicated at two datacenters. Geo-replication reduces throughput by half compared to the left results, because it produces twice the work; and it increases completion latency due to the extra communication.

9.3 Application performance

Lynx's applications are implemented mostly in JavaScript, except for simple read-only one-hop chains, which have an efficient C++ interface.

L-Twitter. We evaluate three common operations: read-timeline for showing a user's timeline, follow-user for starting to follow a user, and post-tweet for posting a tweet. We populate the database with 100,000 users, each with 6 tweets and 6 followers on average. There are 3 datacenters, and we use different geo-replication levels for different tables. We geo-replicate the base tables (*Tweets*, *Graph*) at 2 datacenters, but do not geo-replicate secondary indexes or joins (*e.g.*, *GraphTweets*), which can be reconstructed if there is a disaster.

Figure 12 shows the operation throughput of L-Twitter. For operations that write data, the throughput depends on how many hops the underlying chain has. The chain for post-tweet inserts a row into *Tweets*, updates its replica across datacenters, inserts 6 rows into the join table *GraphTweets* (each user has 6 followers on average) and

Operation	First-hop lat. (50%; 99%)	Completion lat. (50%; 99%)
follow-user	3.2ms; 3.5ms	174ms; 176ms
post-tweet	3.1ms, 3.4ms	252ms; 263ms
read-timeline	3.1ms, 3.3ms	-

Table 2: Latency of operations in L-Twitter. All chains in L-Twitter return after the first hop, so first-hop latency corresponds to the user-perceived latency. Completion latency measures when the entire chain completes.

updates the secondary index of *GraphTweets*, for a total 9 hops (6 of which run in parallel). This results in an aggregate post-tweet throughput of 173K tweets/s. The follow operation also inserts 6 rows into *GraphTweets* (each user has 6 existing tweets), thus having the same number of hops as post-tweet and achieving similar throughput (184K ops/s). For follow, all 6 updates to the secondary index of *GraphTweets* have the same secondary key and thus they could have been batched in one RPC. Lynx does not currently have this optimization. The throughput for reading a user's timeline is high, at more than 1.35M ops/s. This is because the underlying chain only needs to read (many rows) in one server.

Table 2 shows chain latency for the L-Twitter operations. All chains return after the first hop, so L-Twitter achieves low user-perceived latency. The completion latency of *post-tweet* measures how long its chain takes to update the geo-replica of *Tweets*, and update the join table *GraphTweets* and its secondary index. The 99-percentile latency is 263ms, meaning that a tweet quickly appears in all followers' timelines.

L-RUBiS. The most interesting chain in L-RUBiS is the place-bid operation, with a user chain of 2 hops (Figure 7) plus 4 hops of system sub-chains for geo-replication and secondary indexes. The aggregate place-bid throughput is 168K ops/s—3 times lower than the geo-replicated simple-3 chain, which also has 6 hops (Figure 1). This difference is because place-bid runs JavaScript user code at the servers using the V8 engine, which imposes significant overhead, whereas the simple-3 chain does not.

L-Social. We evaluated a common multi-hop user chain in L-Social, post-status. Its first user hop inserts a new status to the *Status* table and the second user hop adds a message "User X has changed her status" to *Activities*. The system chains generated by the second user hop are similar to that of post-tweet in L-Twitter. The overall throughput for post-status is 64K ops/s.

9.4 Scaling

Lynx partitions data across many shards stored at many servers, to scale with both the number of servers/datacenter and the number of datacenters.

Figure 13 shows the aggregate chain throughput when

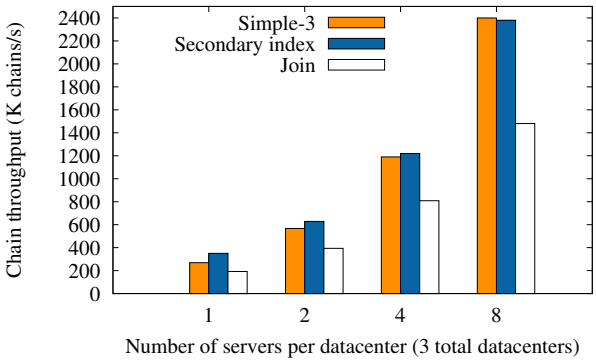

Figure 13: Aggregate chain throughput as the number of server increases in each datacenter. The experiments run on three datacenters with no geo-replication.

we increase the number of servers in each datacenter from 1 to 8. The experiments always run on three datacenters, with our largest experiments having $8 \times 3 = 24$ Lynx servers and 24 clients. We use the simple-3, secondary index, and join workloads (without geo-replication) as described in Section 9.2. We see that Lynx scales well with the number of servers. This is expected as different Lynx chains run independently. With 8 servers/datacenter, the aggregate secondary index throughput is 2.38M chains/s—6.8 times the throughput of 0.35K chains/s for 1 server/datacenter. This is close to linear scaling.

9.5 Comparison with Cassandra/Eiger

We compare the application performance of Lynx to Eiger [40], a geo-replicated key-value storage system with write-only transactions and causal+ consistency, built over Cassandra [2]. We implemented the L-Twitter operations using Eiger's column-family key-value data model. Each user X has a row with four column families: *followers* has a list of sparse columns for users that follow X; *followees* has the users that X follows; *tweets* has the list of posts written by X; and *timeline* has posts from users that X follows. To post a tweet, user X reads the list of followers and uses a write-only transaction to insert the tweet and update the followers' timelines.

The Eiger experiments use the same setup with 3 availability regions. We observe an aggregate throughput of 12K tweets/s. By comparison, L-Twitter running on Lynx achieves 173K tweets/s. Thus, Lynx has better throughput *with serializability* while Eiger offers only causal+ consistency. Admittedly, the performance difference can be an artifact of the two systems' implementation choices; an apples-to-apples comparison is impossible.

Lynx uses much less storage space than Eiger. In L-Twitter, Lynx geo-replicates base tables only once and derived tables zero times, which suffices for disaster tolerance. By contrast, Eiger forces all data to be replicated at all datacenters, causing a large space overhead.

10 Related work

Geo-distributed storage. Prior geo-distributed systems face the unpleasant tradeoff between strong semantics and low latency. Spanner provides strong semantics with order-preserving serializable transactions [22], but these are expensive: like its predecessor Megastore [10], Spanner's update transactions take many cross-datacenter roundtrips to execute and commit. Replicated Commit [41] and MDCC [37] are faster but still incur cross-datacenter latency to execute and commit transactions.

At the other end of the tradeoff, Cassandra [2] and Dynamo [24] are key-value storage systems offering eventual consistency, while PNUTS [21] offers the slightly stronger per-record timeline consistency. Other systems provide stronger but still relaxed semantics to achieve low-latency. COPS/Eiger [39, 40] offer causal+ consistency where write conflicts are resolved deterministically. These systems do not support general transactions and moreover COPS/Eiger require replication of all data across all datacenters. Walter provides parallel snapshot isolation [50] and Gemini provides Red/Blue consistency [38]. Apart from weakened semantics, the latter two systems do not have a scalable design within a datacenter.

Single datacenter storage systems. Since the network latency within a single datacenter is low (sub-millisecond), it is generally agreed that the storage system should provide strong consistency.

The late 80s saw pioneering work in distributed database systems, such as Gamma [25], Bubba [15], R* [42], Teradata, and Tandem [26], which aim to provide the same transactional updates and query interfaces present in centralized database systems. These systems pioneered distributed transactions.

Modern single-datacenter storage systems offer variants of the key-value interface (BigTable [18], H-Base [3], MongoDB [4]). Recently, there has also been strong interest in transactions, e.g. in Sinfonia [6], Percolator [43], and H-store/VoltDB [35]. These systems provide distributed transactions using two-phase commit, which is efficient within a datacenter. HyperDex [27] uses value-dependent chains to update replicas consistently within a datacenter. Value-dependent chains provide a property similar to chain's origin ordering.

View maintenance in database systems. There is much work on maintaining materialized views. Incremental maintenance schemes typically update base tables and views in the same ACID transaction [13]. Deferred maintenance schemes batch changes to tables, and update views periodically or when there is a query [20, 34, 55], for efficiency. Deferred maintenance is often used in data warehouses where only one update batch executes at any time [45]. In the same spirit, LazyBase [19] optimizes data analytics by batching writes and updating materialized secondary indexes in epochs.

Only a few systems offer online distributed view maintenance and even fewer do so in a geo-distributed setting. BigTable now supports secondary indexes [17]. PNUTS added support for secondary indexes and join views that are asynchronously updated [5]. Lynx also updates derived tables asynchronously, in piecewise chains. Unlike PNUTS, Lynx uses static analysis to provide serializability despite asynchronous updates.

Workflow Management [54]. Transaction chains resemble application workflows in systems like travel planning or insurance claim processing. An application workflow naturally consists of many activities, each executing as a transaction. Like Lynx, workflow systems guarantee that all activities are eventually executed completely and exactly once. However, these systems are designed to manage sophisticated workflows often involving people actions, while Lynx uses chains to efficiently execute logical transactions while guaranteeing that entire chains are serializable.

Transaction Decomposition. The database community has explored various aspects in decomposing a transaction in smaller pieces using SAGAS [30], step-decomposed transactions [11], transaction chopping [48], multi-database transactional management [16], and Spheres of Control (SoC) [23, 33]. Garcia-Molina observes that if various pieces of a decomposed transaction commute, a safe execution schedule always exists [29]. Lynx also exploits commutativity, inspired by this and other work including Walter [50], Gemini [38], and conflict-free replicated data types [47]. In addition to commutativity, Lynx also provides the origin ordering property to reduce conflicts among system chains.

11 Conclusion

Lynx provides serializability with low-latency in geo-distributed storage systems. The key insight is to express transactions as chains with multiple hops, and then perform a global static analysis of the chains, to find conflicts and determine when chains can execute piecewise without violating serializability. Chains are also useful for implementing several features: secondary indexes, materialized join views, and geo-replication. We demonstrated the use of Lynx in an auction service, a microblogging service, and a social networking site.

Acknowledgments. This research was supported in part by NSF grant CNS-1218117. We thank Nguyen Tran and Songbin Liu, who contributed to Lynx's design and an earlier implementation. Many people helped us improve the work through discussions and reviews, including Frank Dabek, Robert Grimm, Wilson Hsieh and Dennis Shasha.

References

[1] http://rubis.ow2.org/index.html as of Oct 2010.

[2] Apache cassandra database. http://cassandra.apache.org/.

[3] Hbase: Hadoop database. http://hbase.apache.org.

[4] MongoDB. http://www.mongodb.com.

[5] P. Agrawal, A. Silberstein, B. Cooper, U. Srivastava, and R. Ramakrishnan. Asynchronous view maintenance for VLSD databases. In *International Conference on Management of Data*, June 2009.

[6] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Transactions on Computer Systems*, 27(3):5:1–5:48, Nov. 2009.

[7] C. Amza, A. Chanda, A. Cox, S. Elnikety, R. Gil, K. Rajamani, W. Zwaenepoel, E. Cecchet, and J. Marguerite. Specification and implementation of dynamic Web site benchmarks. In *IEEE International Workshop on Workload Characterization*, Nov. 2002.

[8] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. The potential dangers of causal consistency and an explicit solution. In *ACM Symposium on Cloud Computing*, Oct. 2012.

[9] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. HAT, not CAP: Highly available transactions. In *Workshop on Hot Topics in Operating Systems*, May 2013.

[10] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Conference on Innovative Database Systems Research*, Jan. 2011.

[11] A. J. Bernstein, D. S. Gerstl, and P. M. Lewis. Concurrency control for step-decomposed transactions. *Information Systems*, 24(9):673–698, Dec. 1999.

[12] P. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Survey*, 13(2):185–221, June 1981.

[13] J. Blakeley, P. Larson, and F. Tompa. Efficiently updating materialized views. In *International Conference on Management of Data*, May 1986.

[14] W. Bolosky, D. Bradshaw, R. Haagens, N. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Symposium on Networked Systems Design and Implementation*, Mar. 2011.

[15] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, a highly available parallel database system. *Transactions on Knowledge and Data Engineering*, 2(1):4–24, Mar. 1990.

[16] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *The VLDB Journal*, 1(2):181–239, Oct. 1992.

[17] M. Cafarella, E. Chang, A. Fikes, A. Halevy, W. Hsieh, A. Lerner, J. Madhavan, and S. Muthukrishnan. Data management projects at Google. In *International Conference on Management of Data*, June 2008.

[18] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Symposium on Operating Systems Design and Implementation*, Nov. 2006.

[19] J. Cipar, G. Ganger, K. Keeton, C. B. Morrey, III, C. A. N. Soules, and A. Veitch. LazyBase: Trading freshness for performance in a scalable database. In *European Conference on Computer Systems*, Apr. 2012.

[20] L. Colby, T. Griffin, L. Libkin, I. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *International Conference on Management of Data*, June 1996.

[21] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. 1(2):1277–1288, Aug. 2008.

[22] J. Corbett et al. Spanner: Google's globally-distributed database. In *Symposium on Operating Systems Design and Implementation*, Oct. 2012.

[23] S. Davies. Data processing spheres of control. *IBM Systems Journal*, 17(2):179–198, June 1978.

[24] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles*, Oct. 2007.

[25] D. Dewitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. i Hsiao, and R. Rasmussen. The Gamma database machine project. *Transactions on Knowledge and Data Engineering*, 2(1):44–62, Mar. 1990.

[26] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[27] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: A distributed, searchable key-value store for cloud computing. In *ACM SIGCOMM Conference*, Aug. 2012.

[28] M. Franklin. Concurrency control and recovery. *The Computer Science and Engineering Handbook*,

pages 1058–1077, 1997.

[29] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[30] H. Garcia-Molina and K. Salem. SAGAS. In *International Conference on Management of Data*, May 1987.

[31] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *ACM Symposium on Operating Systems Principles*, Oct. 2003.

[32] D. K. Gifford. Information storage in a decentralized computer system. Technical Report CSL-81-8, Xerox Parc, Mar. 1982. Extended version of the Ph.D. thesis of D. K. Gifford.

[33] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, USA, 1993.

[34] H. He, J. Xie, J. Yang, and H. Yu. Asymmetric batch incremental view maintenance. In *International Conference on Data Engineering*, Apr. 2005.

[35] E. P. C. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *International Conference on Management of Data*, June 2010.

[36] N. Kallen. Big data in real time at Twitter. QCon: The annual international software development conference, Nov. 2010. http://www.slideshare.net/nkallen/q-con-3770885.

[37] T. Kraska, G. Pang, M. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data center consistency. In *European Conference on Computer Systems*, Apr. 2013.

[38] C. Li, D. Porto, A. Clement, R. Rodrigues, N. Preguia, and J. Gehrke. Making geo-replicated systems fast if possible, consistent when necessary. In *Symposium on Operating Systems Design and Implementation*, Oct. 2012.

[39] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen. Don't settle for eventual: Stronger consistency for wide-area storage with COPS. In *ACM Symposium on Operating Systems Principles*, Oct. 2011.

[40] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Symposium on Networked Systems Design and Implementation*, Apr. 2013.

[41] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. E. Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proceedings of the VLDB Endowment*, 6(9):661–672, July 2013.

[42] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems*, 11(4):378–396, Dec. 1986.

[43] D. Peng and F. Dabek. Incremental processing of large data sets. In *Symposium on Operating Systems Design and Implementation*, Oct. 2010.

[44] D. Pritchett. BASE: An acid alternative. *ACM Queue*, 6(3):48–55, May 2008.

[45] D. Quass and J. Widom. On-line warehouse view maintenance. In *International Conference on Management of Data*, May 1997.

[46] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes, and HTTP chunking in web search. In *Velocity Web Performance and Operations Conference*, June 2009.

[47] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, Oct. 2011.

[48] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems*, 20(3):325–363, Sept. 1995.

[49] A. Silberstein, J. Terrace, B. Cooper, and R. Ramakrishnan. Feeding frenzy: Selectively materializing users' event feeds. In *International Conference on Management of Data*, June 2010.

[50] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *ACM Symposium on Operating Systems Principles*, Oct. 2011.

[51] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, F. Kaashoek, and R. Morris. Simplifying wide-area application development with WheelFS. In *Symposium on Networked Systems Design and Implementation*, Apr. 2009.

[52] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch. Session guarantees for weakly consistent replicated data. In *International Conference on Parallel and Distributed Information Systems*, Sept. 1994.

[53] A. Thomson and D. J. Abadi. The case for determinism in database systems. *Proceedings of the VLDB Endowment*, 3(1):70–80, Sept. 2010.

[54] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[55] J. Zhou, P. Larson, and H. Elmongui. Lazy maintenance of materialized views. In *International Conference on Very Large Data Bases*, Sept. 2007.