On Brewing Fresh Espresso: LinkedIn's Distributed Data Serving Platform

Lin Qiao, Kapil Surlaker, Shirshanka Das, Tom Quiggle, Bob Schulman, Bhaskar Ghosh, Antony Curtis, Oliver Seeliger, Zhen Zhang, Aditya Auradkar, Chris Beavers, Gregory Brandt, Mihir Gandhi, Kishore Gopalakrishna, Wai Ip, Swaroop Jagadish, Shi Lu, Alexander Pachev, Aditya Ramesh, Abraham Sebastian, Rupa Shanbhag, Subbu Subramaniam, Yun Sun, Sajid Topiwala, Cuong Tran, Jemiah Westerman, David Zhang
LinkedIn, Inc.
Mountain View, CA, USA
{lqiao,ksurlaker,sdas,tquiggle,bschulman,bghosh,acurtis,oseeliger,zzhang,aauradkar,
cbeavers,gbrandt,mgandhi,kgopalakrishna,wip,sjagadish,slu,apachev,aramesh,asebastian,rshanbhag,
ssubramanian,ysun,stopiwal,ctran,jwesterman,dzhang}@linkedin.com

ABSTRACT

Espresso is a document-oriented distributed data serving platform that has been built to address LinkedIn's requirements for a scalable, performant, source-of-truth primary store. It provides a hierarchical document model, transactional support for modifications to related documents, real-time secondary indexing, on-the-fly schema evolution and provides a timeline consistent change capture stream. This paper describes the motivation and design principles involved in building Espresso, the data model and capabilities exposed to clients, details of the replication and secondary indexing implementation and presents a set of experimental results that characterize the performance of the system along various dimensions.

When we set out to build Espresso, we chose to apply best practices in industry, already published works in research and our own internal experience with different consistency models. Along the way, we built a novel generic distributed cluster management framework, a partition-aware change-capture pipeline and a high-performance inverted index implementation.

Categories and Subject Descriptors: C.2.4 [**Distributed Systems**]: Distributed databases; H.2.4 [**Database Management**]: Systems–concurrency, distributed databases
General Terms: Algorithms, Design, Performance, Reliability
Keywords: Large Databases, Transactions, Secondary Indexing, Cluster Management, Change Data Capture, MySQL

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

1. INTRODUCTION

To meet the needs of online applications, Relational Database Management Systems (RDBMSs) have been developed and deployed widely, providing support for data schema, rich transactions, and enterprise scale.

In its early days, the LinkedIn data ecosystem was quite simple. A single RDBMS contained a handful of tables for user data such as profiles, connections, etc. This RDBMS was augmented with two specialized systems: one provided full text search of the corpus of user profile data, the other provided efficient traversal of the relationship graph. These latter two systems were kept up-to-date by Databus [14], a change capture stream that propagates writes to the RDBMS primary data store, in commit order, to the search and graph clusters.

Over the years, as LinkedIn evolved, so did its data needs. LinkedIn now provides a diverse offering of products and services to over 200 million members worldwide, as well as a comprehensive set of tools for our Talent Solutions and Marketing Solutions businesses. The early pattern of a primary, strongly consistent, data store that accepts reads and writes, then generates a change capture stream to fulfill nearline and offline processing requirements, has become a common design pattern. Many, if not most, of the primary data requirements of LinkedIn do not require the full functionality of a RDBMS; nor can they justify the associated costs.

Using RDBMS technology has some associated pain points. First, the existing RDBMS installation requires costly, specialized hardware and extensive caching to meet scale and latency requirements. Second, adding capacity requires a long planning cycle, and is difficult to do at scale with 100% uptime. Third, product agility introduces a new set of challenges for schemas and their management. Often the data models don't readily map to relational normalized forms and schema changes on the production database incur a lot of Database Administrators (DBAs) time as well as machine time on large datasets. All of the above add up to a costly solution both in terms of licensing and hardware costs as well as human operations costs.

In 2009, LinkedIn introduced Voldemort [8] to our data ecosystem. Voldemort is inspired by Dynamo [15] and is

a simple, eventually consistent, key:value store. Voldemort was initially used for soft-state and derived data sets and is increasingly being used for primary data that does not require a timeline consistent [12] change capture stream.

In early 2011, we took a step back and identified several key patterns that were emerging from our experience with the RDBMS and Voldemort stack that defined our requirements for a primary source of truth system.

- Scale and Elasticity: horizontal scale, and ability to add capacity incrementally, with *zero downtime*.
- Consistency: cross-entity transactions and index consistency with base data for query-after-write cases
- Integration: ability to consume a *timeline consistent* change stream directly from the source-of-truth system
- Bulk Operations: ability to load/copy all or part of a database from/to other instances, Hadoop, and other datacenters, without downtime
- Secondary Indexing: keyword search, relational predicates
- Schema Evolution: forward and backward compatible schema evolution
- Cost to Serve: RAM provisioning proportional to active data rather than total data size

To meet all of these requirements, we designed *Espresso*, a timeline-consistent document-oriented distributed database. Espresso has been deployed into production for several key use-cases.

The paper is organized as follows. We present key feature support in Section 2. We describe the data model, external API, and bulk operations in Section 3. Architecture and system components are presented in Section 4. We elaborate system implementation in Section 5 . Lessons learned from our software development and operational experience are summarized in Section 8. Section 6 describes various use-cases in production that use Espresso as the source of truth data store. We present our experimental evaluation of Espresso in Section 7, describe future work in Section 9, discuss related work in Section 10, and conclude in Section 11.

2. FEATURES

As we set out to meet the requirements, we chose these design principles as we built Espresso:

- Use proven off-the-shelf open-source components
- Learn from the design decisions of similar systems in other companies
- Make sure operability is designed in from the beginning

In this paper, we will discuss some hard problems we did have to solve to meet our requirements. For example, native MySQL replication had to be modified to support our scale and elasticity requirements, and Lucene for full-text indexing required modifications to use at our scale. In the design and building of Espresso, these key capabilities required us to engineer new solutions to meet our set of requirements. In addition to being a scalable and elastic distributed system with low cost to serve, the following is a list of key features Espresso provide.

Transaction Support. Most NoSQL stores do not provide transaction support beyond a single record/document. A large number of use-cases at Linkedin lend themselves to a natural partitioning of the data into collections that share a common partitioning key e.g. memberid, companyid, or groupid. Applications often require that related entities be updated atomically. To model these use-cases, Espresso supports a hierarchical data model and provides transaction support on related entities.

Consistency Model. Brewer's CAP theorem [10] states that a distributed system can only achieve 2 out of consistency, availability and partition-failure tolerance. Espresso is built to be a source-of-truth system for primary user data at Linkedin and needs to maintain consistency. As we describe later in this paper, Espresso follows the master-slave model, where read and write requests for each partition are served by the node acting as the master, while the slaves replicate from the masters to provide durability in case of master failure. However, using synchronous replication between replicas comes at the cost of high latency. Since Espresso supports online user requests, maintaining low latency for operations is critical. For this reason, we relax the consistency to be *timeline consistent*, where the replication between master and slave is either asynchronous or semi-synchronous, depending on the application requirements. In the event of a master failure, the cluster manager promotes one of the slave replicas to be the new master and thus the system maintains availability.

Integration with the complete data ecosystem. Providing a data plaform as a service for application developers is a big motivation for building Espresso. An equally important goal is to support developer agility by ensuring tight integration with rest of the data ecosystem at Linkedin. A large number of specialized online systems such as graph and search indexes rely on getting a low-latency stream from the primary system. In addition, our applications depend on offline data analysis in Hadoop. Providing out-of-the-box access to the change stream is an afterthought in many other NoSQL solutions. In Espresso, we have made this feature a first-class citizen. This ensures that processors in a nearline or offline environment can be added independently, without making any change to the source system. Results from nearline or offline computation often need to be served back to users and supporting these flows natively is also a key capability of Espresso.

Schema Awareness and Rich Functionality. Unlike some other NoSQL stores [2, 5] that are *schema-free*, Espresso supports schema definition for the documents. Enforcing schemas allows systems across the entire data ecosystem to reason about the data in a consistent way and also enables key features such as secondary indexing and search, partial updates to documents and projections of fields within documents.

Espresso avoids the rigidity of RDBMSs by allowing on-the-fly schema evolution. It is possible to add a new field to a document at any time and this change is backward-compatible across the entire ecosystem. When such a change is made, producers as well as downstream consumers of the data do not have to change simultaneously. In an environment where rapid change and innovation is required, the ability to make changes in production without complex process and restrictions is essential.

3. EXTERNAL INTERFACE

3.1 Data Model

Espresso's data model emerged out of our observations of typical use-cases and access patterns at LinkedIn. We wanted to provide something richer than a pure key-value data model, while not forcing ourselves into non-scalable patterns. To that end, we recognized the presence of two primary forms of relationships that exist commonly in our eco-system:

- *Nested Entities*: We often find a group of entities that logically share a common nesting hierarchy. e.g. All messages that belong to a mailbox and any statistics associated with the mailbox, or all comments that belong to a discussion and the meta-data associated with the discussion. The primary write pattern to these entity groups involve creating new entities and/or updating existing entities. Since the entities are logically related, mutations often happen in a group, and atomicity guarantees here are very helpful in simplifying the application logic. The read patterns are typically unique-key based lookups of the entities, filtering queries on a collection of like entities or consistent reads of related entities. For example, show me all the messages in the mailbox that are marked with the flag isInbox and sort them based on the createdOn field.
- *Independent Entities*: Complementary to the Nested Entities model, we also find independent entities which have many:many relationships. e.g. People and Jobs. The primary write pattern to these tend to be independent inserts/updates. Applications tend to be more forgiving of atomicity constraints around updates to top-level Entities, but do need guarantees that any updates that apply to both Entities must eventually happen. When a Person applies for a Job, the person must see the effect of that write right away; in her jobs dashboard, she must see that she has applied for the job. The Job poster who is monitoring all job applications for the job must eventually see that the person has applied for the job, but the update may be delayed.

Espresso uses a hierarchical data model to model Nested Entities efficiently. Independent Entities with relationships are modeled as disjoint Entities with the change capture stream available for materializing both sides of the relationship. Semantically, the data hierarchy is composed of Databases, Document Groups, Tables and finally Documents. We describe these terms in more detail below:

Document. A Document is the smallest unit of data represented in Espresso. Documents logically map to entities, are schema-ed and can have nested data-structures such as lists, arrays and maps as fields within the document. In SQL terms, documents are like rows in a table. They are identified by a primary key which can be composed of multiple key parts. Document schemas allow annotations on fields for supporting indexed access. Secondary indexing is covered in depth in Section 5.

Table. An Espresso Table is a collection of like-schema-ed documents. This is completely analogous to a table in a relational database. A table defines the key structure that is used for uniquely identifying documents that it stores. In addition to externally defined keys, Espresso also supports auto-generation of keys.

Document Group. Document Groups are a logical concept and represent a collection of documents that live within the same database and share a common partitioning key. Readers familiar with MegaStore [9] will recognize the similarity to the Entity Groups concept in the MegaStore data model. Document Groups are not explicitly represented, but inferred by the mere act of sharing the partitioning key. Document Groups span across tables and form the largest unit of transactionality that is currently supported by Espresso. For example, one could insert a new document in the Messages table as well as update a document in the MailboxStats table atomically, as long as both the documents are part of the same document group (keyed by the same mailboxId).

Database. Espresso Databases are the largest unit of data management. They are analogous to databases in any RDMBS in that they contain tables within them. All documents within a Database are partitioned using the same partitioning strategy. The partitioning strategy defines the data partitioning method, e.g. hash partitioning or range partitioning, and other details such as the total number of partitions. Databases are physically represented by a Database schema that encodes the required information such as the partitioning function, number of buckets etc.

3.2 API

Espresso offers a REST API for ease of integration. In this section, we will focus less on the details of the API and more on the capabilities provided to the application developer.

3.2.1 Read Operations

Espresso provides document lookup via keys or secondary indexes. There are three ways to lookup via keys: 1) specify a complete primary key and a single document gets returned, 2) specify a list of resource keys sharing the same leading key, and a list of documents get returned, and 3) specify a projection of fields of a document, and only the required fields are returned. Local secondary index queries are performed by providing the partition key that identifies the document group, the table for which the query needs to be run and the attributes or search terms in the query.

3.2.2 Write Operations

Espresso supports insertion or full update of a single document via a complete key. Advanced operations that are supported include 1) partial update to a document given a complete key, 2) auto-increment of a partial key, and 3) transactional update to document groups (across multiple tables but sharing the same partitioning key). Examples of partial updates include increment/decrement operations on "number" fields, a typical requirement for implementing counters. Deletes are supported on the full keys as well as partial keys.

3.2.3 Conditionals

Conditionals are supported on both Reads and Writes and currently support simple predicates on time-last-modified and etag (CRC of the document contents). These are typically used to implement the equivalent of compare-and-swap style operations. In practice, since the rich API allows fairly complex server-side processing, we rarely see conditional writes.

3.2.4 Multi Operations

All read and write operations have their "Multi" counterparts to support multiple operations grouped into one transaction. These are typically used for batch operations.

3.2.5 Change Stream Listener

Espresso also provides a Change Stream Listener API through Databus. This allows an external observer to observe all mutations happening on the database while preserving the commit-order of the mutations within a document group. The API allows an observer to consume the stream, while noticing transaction boundaries to maintain consistency at all times if required. Each individual change record contains the type of the change (Insert, Update, Delete), the key of the document modified, the pre and post-image (if applicable) and the SCN of the change. This allows very easy integration with nearline processors that are performing streaming computations and other kinds of processing on the change stream.

3.3 Bulk Load and Export

An important part of Espresso's place in the data ecosystem is its seamless integration with the end-to-end data flow cycle. To that end, we support efficient ingestion of large amounts of data from offline environments like Hadoop into Espresso. Hadoop jobs emit to a specialized output format that writes out the primary data as well as the index updates (if applicable) and applies the partitioning function to write out files that align with the online partitions. As a final step, the output format notifies the online Espresso cluster about the new data and its location by interacting with Helix. Each storage node then pulls in the changes that are related to the partitions that it is hosting and applies them locally using efficient bulk-insert paths. The bulk ingest operation supports incremental semantics and can go on while reads and writes are happening to the online data set. Storage nodes indicate completion after processing their individual tasks allowing an external observer to monitor the progress of the bulk import very easily.

The inverse operation to bulk data ingest is data export. This is extremely important to support ETL and other kinds of large offline data analysis use-cases. We use Databus to provide the real-time stream of updates which we persist in HDFS. Periodic jobs additionally compact these incremental updates to provide snapshots for downstream consumption. This allows offline processors to either process the change activity or the entire data set depending on their needs. There are a host of interesting sub-problems here like metadata management and evolution, inexact alignment between wall-clock time and the logical clock timeline provided by Espresso's commit log, and ensuring end-to-end data integrity and consistency. For brevity, we skip these discussions in this paper.

4. SYSTEM ARCHITECTURE

4.1 System Overview

The Espresso system consists of four major components: clients and routers, storage nodes, databus relays and cluster managers. The overall system architecture is depicted in Figure 1, where the data flow is depicted in solid arrows, The data replication flow is depicted in double solid arrows, and the cluster state control flow is shown in dotted line. Espresso clients generate reads and writes to routers, and routers pass the requests to storage nodes who own the data. Storage node process the request through the primary key or the secondary index lookup. Changes to the base data is replicated from one storage node to databus relays, and consumed by another storage node which maintains a consistent replica of the primary source. Cluster manager, aka Helix, monitors and manages the state of the entire cluster, including storage nodes, and databus relays. Routers and storage nodes are also spectators of the states the cluster, so they react to cluster state changes accordingly.

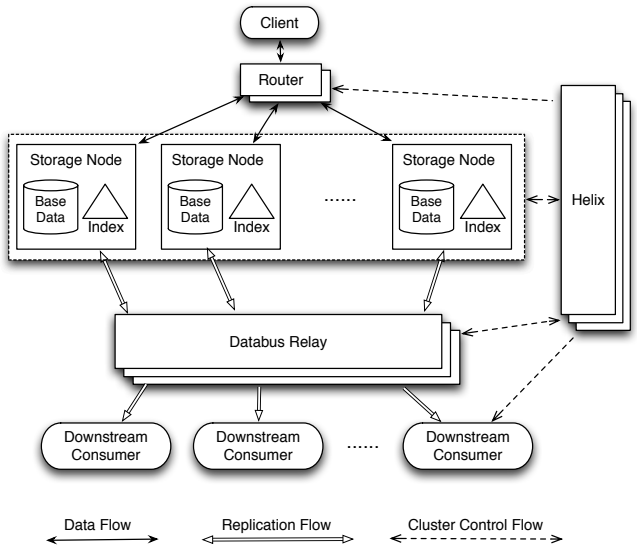


Figure 1: Espresso Architecture in a Single Colo

4.2 System Components

Client and Router. An application sends a request to an Espresso endpoint by sending an HTTP request to a router, which inspects the URI, forwards the request to the appropriate storage node(s), and assembles a response. The routing logic uses the partitioning method for the database as specified in the database schema and applies the appropriate partitioning function, e.g. a hash function, to the partition key of the URI to determine the partition. Then it uses the routing table that maps each partition to the master storage node, and sends the request to that node. For a request without a partitioning key, such as an index search query on the whole data set, the router queries all storage nodes, and sends the merged result set back to the client.

Cluster Manager. Espresso's cluster manager uses Apache Helix [16]. Given a cluster state model and system constraints as input, it computes an ideal state of resource distribution, monitors the cluster health, and redistributes resources upon node failure. Helix throttles cluster transitions during resource redistribution, and ensures the cluster is always in a valid state, whether in steady-state mode or while executing transitions.

Each Espresso database is horizontally sharded into a number of partitions as specified in the database schema, with each partition having a configurable number of replicas. For each partition, one replica is designated as master and the rest as slaves. Helix assigns partitions to storage nodes in

accordance with these constraints: 1) only one master per partition (for consistency), 2) master and slave partitions are assigned evenly across all storage nodes (to balance load), 3) no two replicas of the same partition may be located on the same node or rack (for fault-tolerance), and 4) minimize/throttle partition migration during cluster expansion (to control impact on the serving nodes).

Helix [16] is a generic cluster management system, and an Apache incubator project. Please refer to the Helix paper [16] for an in-depth description.

Storage Node. The storage node is the building block for horizontal scale. Data is partitioned and stored on storage nodes, which maintain base data and corresponding local secondary indexes. Storage nodes also provide local transaction support across entity groups with a common root key. To achieve read-after-write consistency, storage nodes apply updates transactionally to base tables and their secondary indexes.

Storage nodes maintain replicas using a change log stream provided by the replication tier. Committed changes to base data are stored locally in a transaction log that is pulled into the replication tier. Slave partitions are updated by consuming change logs from the replication tier, and applying them transactionally to both the base data and the local secondary index.

In addition to serving requests from the client and from the replication tier, a storage node also runs utility tasks periodically, including consistency checking between master and slave partitions, and backups. To minimize performance impact on read/write requests, utility tasks are throttled to control resource consumption.

Databus. For replication, Espresso uses Databus [14], LinkedIn's change data capture pipeline to ship transaction events in commit order and provide **Timeline Consistency**. *Timeline Consistency* is a form of *eventual consistency*, which guarantees that events are applied in the same order on all replicas for each partition. This feature provides the basis for implementing failover and rebalancing. Changes from master partitions are captured, transferred and applied to the replicas in the following manner. First, on a storage node, all changes are tagged with a local transaction sequence number and logged. Second, the log is shipped to Databus relay nodes. Third, a different storage node pulls changes from Databus relay and applies them to its slave partitions. All these steps follow the transaction commit order so that timeline consistency is guaranteed.

Databus achieves very low replication latency, and high throughput. It can easily scale to thousands of consumers. It provides data filtering, allowing a consumer to subscribe to changes from a specified subset of tables in a source database. A databus consumer can also selectively choose events of interest at various granularities. For example, a consumer can subscribe to a list of partitions or a set of tables. The same Databus instances used for replication provide an external change capture pipeline for downstream consumers within the data ecosystem, such as Hadoop clusters, the social graph engine, people search service, etc.

5. IMPLEMENTATION

5.1 Secondary Index

In contrast to a simple key-value data model, the Document Groups data model allows us to support certain forms of secondary indexing very efficiently. One simple use-case for this is selecting a set of documents from a document group based on matching certain predicates on the fields of the documents. In a key-value model, the application developer either has to fetch out all the rows and perform the filtering in application code, or has to maintain the primary relationship and reverse-mappings for every secondary key that could be used to access this document. The first approach doesn't scale for very large document groups, the second creates the potential for divergence between the primary and reverse-mappings due to the combination of dual-writes in the face of different failure scenarios. Additionally, if the query involves consulting several such secondary key-value pairs, the read performance of the system gets impacted. Thus even though individual key-value lookups might be very cheap, the overall cost of a secondary-key based lookup might be quite high. Secondary Indexes on Document Groups are what we call *local secondary indexes*. This is because Espresso stores all entities that belong in the same entity group on a single node. Secondary Indexes on Independent-Entity relationships are what we call *global secondary indexes*. These are typically implemented as derived tables that are guaranteed to be updated asynchronously by processing the change stream emanating from the primary entity database. In this section, we focus on Espresso's implementation of local secondary indexes.

The key functional requirements were:

1. Real-Time indexes: Almost all our use-cases with hierarchical data models require read your own writes consistency regardless of whether the access is by primary key or by query on the collection.
2. Ease of Schema Evolution: Given the product agility, we see a lot of changes in the schema of these documents. Therefore being able to add new fields and add indexing requirements on them with zero downtime and operational ease was a big requirement.
3. Query flexibility: Another consequence of product agility is that queries evolve quickly over time. The traditional wisdom of specifying your queries up front, spending time with the DBA tuning the query to use the appropriate indexed paths is no longer viable in this fast-paced ecosystem.
4. Text search: Apart from queries on attributes, it is fairly common to allow the user to perform free-form text search on the document group.

We require users to explicitly indicate indexing requirements on the fields by annotating them in the Document schema. Schema evolution rules allow users to add new fields and declare them indexed, or index existing fields.

Our first attempt used Apache Lucene^{TM}, which is a high-performance, full-featured text search engine library written entirely in Java. This was primarily motivated by our requirements for text search as well as our in-house expertise with Lucene. Internally, Lucene uses inverted indexes, which fulfill requirements 2, 3, and 4. However, Lucene has a few drawbacks:

- It was not initially designed for real-time indexing requirements.
- The entire index needs to be memory-resident to support low latency query response times.
- Updates to the document require deleting the old document and re-indexing the new one. This is because

Lucene is log-structured and any files it creates become immutable.

We applied several techniques to address the first two drawbacks and bring latencies and memory footprint down to acceptable levels. The first idea was to organize the index granularity be per collection key rather than per partition. This creates a lot of small indexes but allows our memory footprint to be bounded by the working set. We then store these millions of small indexes in a MySQL table to ameliorate the impact of so many small files and achieve transactionality of the index with the primary store.

In our second attempt we built an indexing solution that we call the Prefix Index. We keep the fundamental building block the same: an inverted index. However, we prefix each term in the index with the collection key for the documents. This blows up the number of document lists in the inverted index, but at query time, allows us to only look at the document lists for the collection that is being queried. The terms are stored on a B+-tree structure thus providing locality of access for queries on the same collection. This is equivalent to having a separate inverted index per collection in terms of memory footprint and working set behavior without the overhead of opening and closing indexes all the time. In addition, we support updates natively to the index because our lists are not immutable. Only the lists affected by the update are touched during the update operation. Similar to the Lucene implementation, we store the Prefix Index lists in MySQL (InnoDB) to additionally get the benefit of transactionality with the primary store. During query processing, the lists that match the query terms are consulted, bitmap indexes are constructed on the fly to speed up the intersecting of these lists and the results are then looked up from the primary store to return back to the client.

5.2 Partitions and replicas

Espresso partitions data to serve two purposes, 1) load balancing during request processing time, and 2) efficient and predictable cluster expansion. Since Espresso serves live online user traffic, maintaining the SLA during cluster expansion is mission-critical. Cluster expansion through partition splitting tends to create significant load on the existing serving nodes. Instead, we partition the data into a large number of partitions to begin with and only migrate partitions to new nodes when cluster expansion is needed. Overpartitioning also keeps the partition size small, which has a number of advantages. Ongoing maintenance operations such as backup/restore can be done in less time using parallelism, when partitions are smaller. We have made significant optimizations in our cluster manager (Helix) to manage the overhead resulting from a large number of partitions.

Each partition is mastered at one node and replicated on n nodes in a traditional master-slave model. We ensure that slave partitions do not collocate on the same node or rack. Partitions and replicas are completely invisible to external clients and downstream consumers.

5.3 Internal Clock and the Timeline

As described earlier, Espresso partitions its data set using a partitioning function that can be customized per database. Each database partition operates as an independent commit log. Each commit log acts as a timeline of data change events that occured on that partition forming an internal

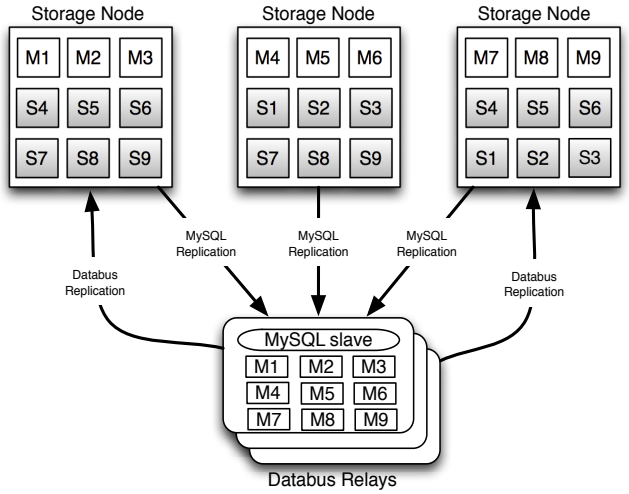


Figure 2: Partitions and Replication Flow

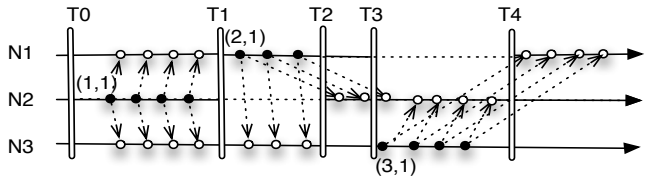


Figure 3: Timeline of Events

clock. These timelines are preserved in Databus relays, all Espresso replicas, and provided to all down-stream consumers. A timeline consists of an ordered sets of changes in a partition. Each change set is annotated with a monotonically increasing system change number (SCN). SCN has two parts, generation number and sequence number. Events committed within one transaction all share the same SCN. For each new transaction, the sequence number increments by one. For each mastership transfer, the generation number increments by one. Figure 3 depicts a timeline of a partition among all replicas *N1*, *N2*, and *N3*. At T0, *N2* is the master. It sends events starting from SCN (1,1) to other replicas. At T1, *N2* fails. *N1* becomes the new master, and starts a new generation of SCN (2,1). At T2, *N2* comes back and receives events generated by *N1*. At T3, *N1* fails, *N3* becomes the new master, and generates events from SCN (3,1). At T4, *N1* comes back, and receives all missing events. The timeline from all replicas of a given partition is the same.

5.4 Replication and Consistency

Several large web companies [3, 4, 7] have used sharded MySQL with replicas to build a scalable primary fault-tolerant data store. There are two flavors of sharded MySQL architectures, the first involves one logical partition per MySQL instance and the second involves multiple logical partitions per MySQL instance. The first approach requires re-sharding during cluster expansion which is a very painful process as described by the F1 paper [18]. Applying logical sharding on top of MySQL allows the data to be located independently on disk, which simplifies partition-level backups, re-

stores etc. However, there are still several issues due to MySQL replication that we discovered, 1) MySQL uses log shipping to maintain consistency among replicas, but its log is not partitioned although data is sharded; 2) MySQL does not support replication of partial logs. Because of these limitations, vanilla MySQL replication cannot achieve the following goals, a) **load balancing**: mixing master and slave partitions within one MySQL instance so loads are evenly spread across instances, and 2) **online cluster expansion**: moving a subset of partitions from one instance to a new instance without quiescing the cluster. We designed our replication layer to address these problems, while still using MySQL replication stack as a building block.

Espresso replication flow is shown in Figure 2. First, we enhanced MySQL's binary log by adding a new field to record the partition-specific commit SCN. Using the MySQL replication protocol, the binary log is pulled into the Databus relays. Between the MySQL master and the relays, we can use either semi-synchronous or asynchronous replication. These two options provide a tradeoff between write latency and durability. When semi-sync replication is used, the commit blocks until the change is acknowledged by at least one relay. When a slave is promoted to master, it first drains all the events from Databus, thus guaranteeing that the slave applies all changes that were committed on the master.

To ensure consistency among replicas, we developed a consistency checker. It calculates the checksum of certain number of rows of a master partition, replicates the checksum to storage nodes running slave partitions. A storage node calculates checksum against the same set of rows, and compares against the master checksum. On detection of errors, we apply recovery mechanisms such as restoring a slave partition from a backup of the master partition, to fix the inconsistent slaves.

5.5 Fault-tolerance

Espresso is designed to be resilient to hardware or software failures. Each system component in Espresso, as described in Section 4, is fault-tolerant. We elaborate failure handling for each component below.

As we mentioned before, data is stored in storage nodes in units of partitions. For each partition has replicas, one as a master, and the rest as slaves. When a storage node fails, all the master partitions on that node have to be failed over, meaning for each master partition on the failed node, a slave partition on a healthy node is selected to take over. The failover process is the following. Helix first calculate a new set of master partitions from existing slaves, so that the load is evenly spread across the remaining storage nodes. Assume a selected slave partition is at SCN (g, s). The slave partition drains any outstanding change events from databus and then transitions into a master partition. The new master starts a new generation of SCN ($g+1$, 1).

To detect storage node failures, Helix uses two approaches in combination: 1) Use Zookeeper heartbeat for hard failure. If a node fails to send heartbeat for configurable amount of time, it is treated as failure; 2) Monitor performance metrics reported by router or storage nodes. When a router or a storage node starts seeing problems, such as a large volume of very slow queries – an indication of a unhealthy node, it reports to Helix and Helix treats this as failure and initiates mastership transfer.

During the fail-over time, there is transient unavailability for the partitions mastered on the failed node. To minimize transition latency, Helix always promotes a slave partition which is closest in the timeline to the failed master to become the new master. Router can optionally enable slave reads to eliminate read unavailability due to failures. After slave to master transition finishes for a partition, Helix changes the routing table stored on ZooKeeper, so that the router can direct the requests accordingly.

Databus is also fault-tolerant. Each databus relay instance has n replicas (we use 2 or 3). For every storage node, one relay is designated to be the leader while n-1 are designated to be followers. The leader relay connects to the data source to pull the change stream while the follower relays pull the change stream from the leader. The clients, including both espresso storage nodes and external consumers, can connect to any of the relays, either leader or follower. If the leader relay fails, one of the surviving followers is elected to be the new leader. The new leader connects to the storage node and continues pulling the change stream from the last sequence number it has. The followers disconnect from the failed leader and connect to the new leader. This deployment drastically reduces the load on the data source server but when the leader fails, there is a small delay while a new leader is elected. During this window, the latest changes in the change stream from a storage node are not available to another storage node, consuming these changes.

Helix is managed by itself with several replicas using Leader-Standby state model. Each helix instance is stateless. If current leader fails, a standby instance will be elected to be a leader, and all helix clients, including storage nodes, relays and routers, will connect to the new leader.

5.6 Cluster Expansion

Espresso is elastic: new storage nodes are added as the data size or the request rate approaches the capacity limit of a cluster. Espresso supports online cluster expansion, which is a business requirement for being an online data store. When nodes are added we migrate partitions from existing nodes to new ones without pausing live traffic. When expanding an Espresso cluster, certain master and slave partitions are selected to migrate to the new nodes. Helix will calculate the smallest set of partitions to migrate to minimize data movement and cluster expansion time. This way the cluster expansion time is proportional to the percentage of nodes added. In the future, we plan to enhance Helix to take machine resource capacity into account with heterogeneous machines accumulated over years, when calculating the new partition assignment. For each partition to be migrated, we first bootstrap this partition from the most recent consistent snapshot taken from the master partition. These partitions can then become slaves. These slaves consume changes from databus relay to catch up from current masters.

5.7 Multi Datacenter

At present, LinkedIn serves the majority of its traffic from a single master data center. A warm standby is located in a geographically remote location to assume responsibility in the event of a disaster. The Espresso clusters in the disaster recovery (DR) data center are kept up to date using the change log stream from the primary clusters. These DR clusters can serve read traffic when there are no freshness

requirements but do not serve any write traffic. In the event of a switch-over to the DR site, operators must manually set the DR cluster to be a primary and enable replication to the old primary.

6. ESPRESSO IN PRODUCTION

Espresso has been deployed in production at LinkedIn since June 2012. In this section, we talk about a few use-cases that are running on Espresso today.

6.1 Company Pages

LinkedIn Company Pages allow companies to create a presence on the LinkedIn network to highlight their company, promote its products and services, engage with followers and share career opportunities. Over 2.6 million companies have created LinkedIn Company Pages. A company's profile, products and services and recommendations of those products and services are stored in the *BizProfile* database.

Company Pages is an example of a normalized relational model migrated to Espresso. Each Company Page provides profile information about the company. A Company Profile page may list one or more of the company's products and services. Finally members may provide recommendations for products and services. This hierarchy is implemented as three separate tables with products nested under companies, and recommendations nested under products.

This use case exhibits a typical read-heavy access pattern with an 1000:1 ratio of reads to writes.

6.2 MailboxDB

LinkedIn provides InMail, a mechanism for members to communicate with one another. A member's inbox contains messages sent by other members, as well as invitations to connect. The MailboxDB is currently migrating from an application-sharded RDBMS implementation to Espresso. This use case illustrates several features of Espresso including collection resources, transactional updates, partial update and local secondary indexing.

The MailboxDB contains two tables. The *Message* table contains a collection of messages for each mailbox. Each document in the *Message* table contains subject and body of the message along with message metadata such as sender, contining folder, read/unread status, etc.

The most frequent request to the InMail service is to get a summary of the inbox content that includes two counts: the number of unread messages and the number of pending invites. Particularly for large mailboxes, the cost of computing these counts on each summary request, even by consulting an index, would be prohibitive. Instead, the application actively maintains a separate summary document per mailbox. These summary documents are stored in the *Mailbox* table.

To prevent the counters in the summary document from getting out of sync with the data, the update of the *Message* table and the *Mailbox* table need to be transactional.

The MailboxDB has an atypically high write ratio. In addition to new mail delivery, each time a member reads an InMail message a write is generated to mark the message as no longer unread. As a result, we see approximately a 3:1 read to write ratio for the MailboxDB.

6.3 USCP

The Unified Social Content Platform (USCP) is a shared platform that aggregates social activity across LinkedIn. By integrating with USCP, a LinkedIn service can annotate its data with social gestures including *likes*, *comments* and *shares*. Services that utilize USCP include LinkedIn Today, the Network Update Stream on a member's home page and our mobile applications for tablets and phones.

The typical USCP access pattern is as as follows. A service selects the set of domain-specific data items to display to a user, then queries the USCP platform for any social gestures that have accrued to the selected items. The content is annotated with the social gestures, if any before it is displayed. The UI includes calls to action to allow the viewer to like, comment or share. For example, a LinkedIn Today article summary is displayed with a count of the article's Likes and Comments along with the most recent of each.

The USCP workload has a very high Read:Write ratio, and read requests are predominantly multi-GET requests.

7. EXPERIMENTAL EVALUATION

We ran a comprehensive set of experiments to evaluate various aspects of our system, including availability under failure cases, elasticity with cluster expansion, and performance under different workloads and failures.

7.1 Test Setup

Type	*CPU*	*RAM*	*Storage*
S1	2x6-core Xeon@2.67GHz	48GB	1.4TB SSD 1TB SAS
S2	2x6-core Xeon@2.67GHz	24GB	1TB SATA

Component	*Servers*	*Machine Type*
Storage Node	12	S1
Databus Relay	3	S1
Router	3	S2
Helix	3	S2
ZooKeeper	3	S1

Table 1: Machine Configuration and Cluster Setup

We set up a testing cluster in a production fabric as shown in Table 1. We used the production releases of Espresso Storage Node, Databus Relay, Helix, and Router. We used the workloads extracted from USCP and MailboxDB use cases, and enhanced the workloads to evaluate system performance from various aspects. The cluster is set up in the following way. Data on storage nodes is configured to have 3 replicas, one master and two slaves. Databus relay is configured to have 2 replicas, one leader and one stand-by. Helix is configured to have one leader node and two stand-by nodes. Finally, Zookeeper uses 3 nodes with 2-node quorum setup.

7.2 Availability

We ran a set of experiments to measure time to fail-over when a storage node fails. Since our basic fault-tolerance unit is a partition, there is per-partition management overhead. We have varied the total number of partitions, and observed how it plays a role in availability.

First, we increased the total number of partitions from 64 to 2048, and measured the fail-over latency when a single node fails. We compared two configurations of state-transitions, one with singular-commit and one with group-commit. Singular-commit invokes state transition message for each partition, and group-commit invokes state transition for a group of partitions on a storage node. With

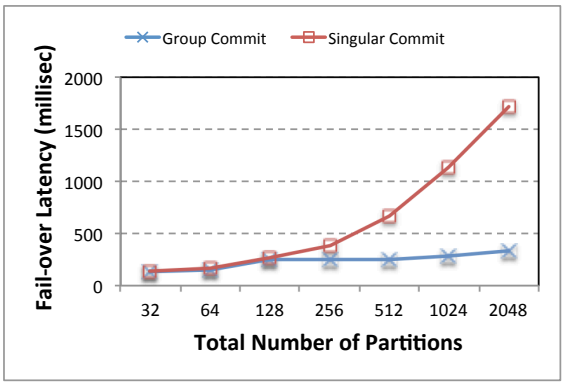


Figure 4: Fail-over Latency

group-commit the amount of disk I/O on ZooKeeper is significantly reduced. The results are shown in Figure 7.2. Fail-over latency with group commit is significantly smaller compared with singular commit when the number of partitions increases. However, the overall fail-over latency increases too. So practically, we cannot use a very large number of partitions. We found the knee of the curve to be between 512 and 1024 partitions, and is the range recommended for production.

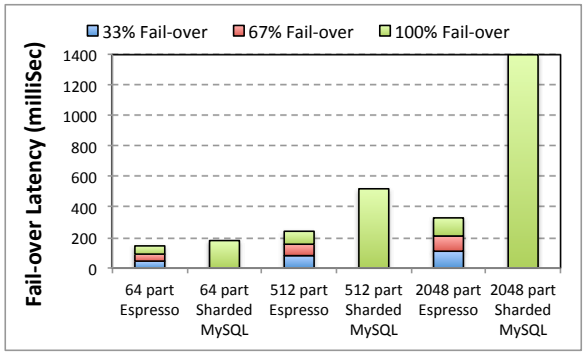


Figure 5: Espresso vs MySQL Fail-over Comparison

Second, we compared sharded MySQL with Espresso. We used the same partitioning strategy between Sharded MySQL and Espresso. However, the partition allocation is different between these two systems. We changed the Helix partition allocation and fail-over logic to make Espresso behave like Sharded MySQL. For Sharded MySQL, master partitions and slave partitions cannot be mixed on a single node (or more precisely a single MySQL instance). Partitions on a node are all master partitions or all slave partitions. When a master node fails, in Sharded MySQL, a slave node needs to completely take over all partitions, then become a master node. The fail-over latency for 64, 512, and 2048 partitions is shown in Figure 5. Espresso always outperforms Sharded MySQL, the higher the number of partitions, the bigger the gap. The reason is during fail-over, Helix evenly distributes failed partitions to the rest of the cluster, so each running node does less work and the degree of fail-over parallelism is higher. Note that we also plot the fail-over progress in Figure 5. For Espresso, it is making gradual progress to fail-over partition by partition to healthy storage nodes. For Sharded MySQL, it is all or nothing, so writes to the failed partitions is blocked until all failed partitions are taken over by a slave node.

7.3 Elasticity

As Espresso supports larger data sizes and request rates, on-line cluster expansion without down-time becomes a key requirement. It also needs to be done in a timely fashion when expansion is required. When a cluster expands, partitions will be moved from existing nodes to new nodes. The ideal amount of data movement should be proportional to the degree of cluster expansion. We varied the degree of cluster expansion from a 6-node cluster and showed the results in Figure 6. First, we executed the cluster expansion serially, one partition at a time. The cluster expansion latency matches very well with the expected % of data moved, which is derived from the expected % of cluster expanded. Second, we expanded the cluster in parallel on multiple storage nodes simultaneously. With more nodes added, the higher the degree of parallelism, the faster the expansion.

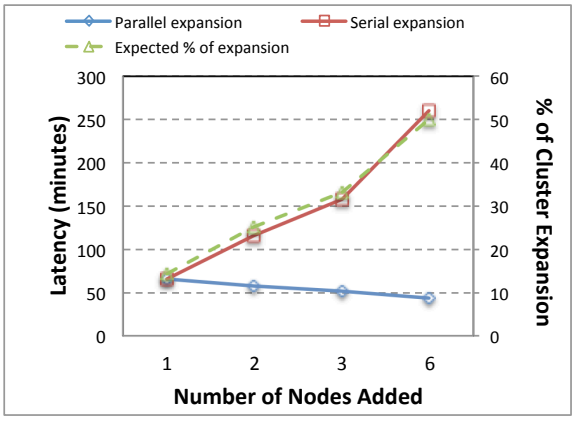


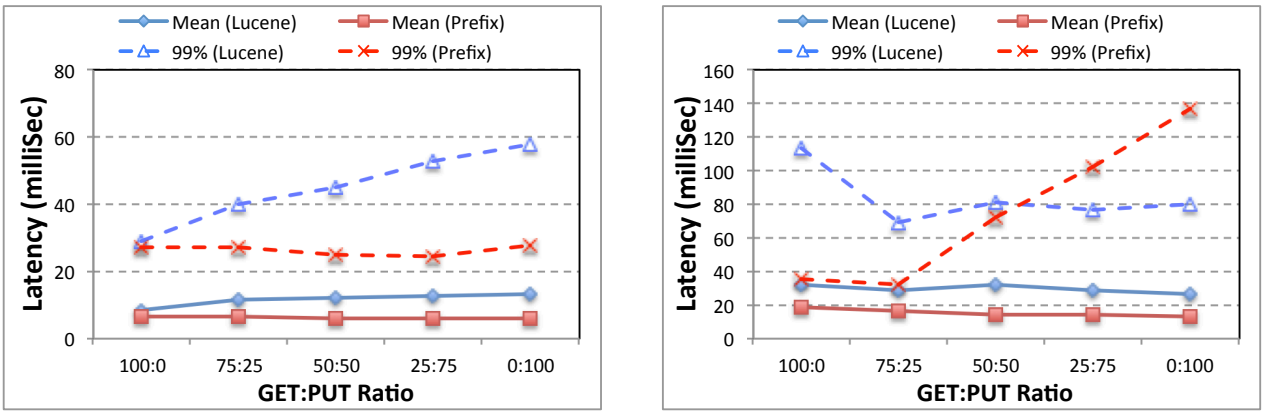
Figure 6: Cluster Expansion Performance

7.4 Performance

In this section, we present performance results with workloads representing MailBoxDB use cases. All tests are running against a single storage node.

The first workload we use is from the MailBoxDB use case. We ran tests with 2 data sets. The data sets differs in the size of the mailboxes. Set A is for small mailbox test, each mailbox having 100 messages and a total of 400,000 mailboxes. Set B is for large mailbox test, each mailbox having 50K messages and a total of 800 mailboxes. The Message schema has 39 fields, 4 of which are indexed. The workload simulates 1% users having concurrent sessions, each session lasting 10 requests. Users are selected uniformly randomly. In this set of tests, we compare two index implementations, namely Lucene and Prefix-index as described in Section 5.1. In each test, we varied the read write ratio, and collect latency stats. For small mailbox, we run 2000 requests per second, and for large mailbox, we run 250 requests per second. The results are shown below in Figure 7(a) and 7(b).

For small mailboxes, Prefix index outperforms Lucene by



(a) Small MailBoxes

(b) Large MailBoxes

Figure 7: Index Performance Comparison

2x. The reason is for each PUT, Lucene invokes much more I/O than Prefix index because the Lucene index is immutable, so new index files are generated, while Prefix index does in-place update. The Prefix index is more stable than Lucene. Its mean and 99% latency stays constant when the workload mix changes, while Lucene performance degrades with more writes. This is because for Prefix index, the amount of I/O is fixed for PUT and GET regardless of the mix. For Lucene, more PUT causes index fragmentation, and hence degrade GET performance. Lucene also has compaction overhead for garbage collection, which makes the 99% latency much higher.

For large mailboxes, Prefix index outperforms Lucene by 1.8 to 2x. Lucene mean latency is more stable when workload mix changes with large mailboxes compared with small mailboxes, because for large mailboxes, GET and PUT latency is quite similar. The reason is with large mailboxes, there is a big index segment per mailbox with a few small delta segments. So both GET and PUT latency is dominated by access to the big segment. The Prefix index mean latency is better when there is more PUTs, because GETs are more expansive than PUTs, as they needs to read in all inverted list rows. However, Lucene has better 99% latency than Prefix index. This is because Prefix index starts a transaction, fetches the inverted index rows, modifies them, then write them back. Because the mailboxes are big, so the PUT cost is higher, and hence the transaction spans longer period. With concurrent users, transactions can collide on locking the same rows. In real COMM application, the chance that the same user update his/her own mailbox is very rare. So in practice, this problem is not very crucial. However, we still plan to mitigate this problem by the following improvement: we push down the inverted index mutation to MySQL, to avoid the overhead of parsing and client/server I/O, so the transaction time is much shorter, and hence the possibility of collision is much smaller.

8. LESSONS LEARNED

The deployment and testing experience with Espresso has provided us several valuable lessons in the area of high-availability and performance tuning in various parts of the system stack.

Availability: As we have described in this paper, Espresso uses master-slave architecture where the master handles writes (and fresh reads). When there are node failures, the cluster manager transfers the mastership to surviving replicas after a configured timeout. To avoid flapping where mastership is repeatedly transferred due to transient failures, this is set to a few seconds. During this period, certain partitions become briefly unavailable. We have allowed slave reads to address read unavailability but the partitions remain unavailable for writes till a new master is chosen. This is significant outage for some applications and we are working on reducing this downtime.

Storage Devices: One of the big challenge of building a data platform at Linkedin is the large variety of use-cases and their access patterns. To enable multi-tenancy while guaranteeing application SLAs required a lot of system tuning. The use of SSDs and their massive IOPs capacity which reduce the dependency on buffer caches and the locality of access, has been very effective. In fact, we found that SSDs challenged a lot of the conventional wisdom including cost-effectiveness. For example, the mailbox usecase has a large data footprint and is storage bound. But to satisfy the request pattern of this usecase, especially for secondary indexes, we found that it was in fact cheaper to use SSDs as SAS disks would have cost much more for the same IOPs.

Java Tuning: Espresso has been developed in Java and while this has allowed for accelerated development, JVM tuning has been a challenge especially with mixed workloads. Use of SSDs in fact exacerbates this problem as SSDs allow the data to be read at a very high rate, draining free memory and causing Linux to swap out memory pages of the JVM heap. We had to invest significant time in JVM tuning to address this issue. Using InnoDB as the storage engine has also helped since the bulk of the data is held in InnoDB buffer pool which is outside the JVM heap. During performance tuning, we also found that the complex interaction between different components make it rather difficult to analyze performance bottlenecks using off the shelf tools. To address this, we ended up building a performance tooling framework that allows developers to automate runs with their code and visualize the impact of their changes on performance.

Cluster Management: Apache Helix (the generic cluster-manager used by Espresso), uses Zookeeper as the coordination service to store cluster metadata. Although Zookeeper supports a high throughput using sequential log writes, we found the latencies to be high even with dedicated disk drives. This is particularly significant during fail-over, especially when the number of affected partitions is very high. We have made a number of improvements in Helix like group commit that improve the latency of zookeeper operations, which has allowed us to scale to tens of thousands of partitions in a single cluster. We plan further improvements in this area that will allow us to scale even more.

9. FUTURE WORK

Cross-entity transactions. Espresso's hierarchical data model allows related entities in a collection to be updated consistently. This satisfied the requirements of a large number of usecases, which model entity-child relationships. In the few cases that model entity-entity edge relationship and require both the edges to be modified atomically, we currently do not provide this support and they risk seeing inconsistent data at times. We plan to add cross entity transaction support in the future to address this requirement.

Platform extensibility. Espresso was designed to be a generic platform for building distributed data systems. So far, we have focused on building a OLTP system as described in the paper. MySQL/InnoDB have proven to be very reliable and reusing MySQLs capabilities of binary log and replication have allowed for a very aggressive development schedule. There are many other usecases we plan to address with the platform. We have some ongoing work to use Espresso to build an OLAP system using a columnar storage engine. We also plan to use a log-structured engine to address usecases that have very high write rates.

Multi-master deployment. We are currently working on extending Espresso to be active in multiple data centers so that data is both readable and writable in multiple data centers. The latency between LinkedIn data centers is typically significantly greater than the latency for a local write. Thus coordination across data centers for reads or writes it not acceptable. In the multi-DC deployment, we plan to relax cross-colo consistency and service reads and writes locally in the requesting data center to avoid degrading the user experience.

Most individual user-generated data at LinkedIn is only writable by a single, logged in, member. To provide read-your-writes consistency, we need to avoid accepting a write from a member in one data center then servicing a subsequent read from the member from a different data center before the write has propagated. In the event the same row was updated in two data centers within the propagation latency, the applier of remotely originated writes applies a Last Writer Wins conflict resolution strategy to guarantee that the row resolves to the same value in all regions.

To minimize the likelihood of write conflicts we are developing inter-data center sticky routing by member id. When a member first connects to LinkedIn, a DNS server will select a data center based on geoproximity and current network conditions. Upon login, the member will be issued a cookie indicating the data center that accepted the login. All subsequent access by the same member, within a TTL, will be serviced from the same region, redirecting if necessary.

10. RELATED WORK

Espresso provides a document oriented hierarchical data model that's similar to that provided by Megastore [9] and Spanner [13]. Unlike MegaStore and Spanner, it does not provide support for global distributed transactions but the transaction support within an Entity group offered by Espresso is richer than most other distributed data systems such as MongoDB [5], HBase [1] and PNUTS [12]. Among the well-known NoSQL systems, MongoDB is the only one that offers rich secondary indexing capability at par with Espresso, although it lags Espresso in terms of RAM:disk utilization. With the exception of MongoDB, other NoSQL systems do not offer rich secondary indexing capability that Espresso offers.

Like BigTable [11] and HBase [1], Espresso chooses CA over AP in contrast to most Dynamo style systems such as Riak and Voldemort. However, HBase and BigTable follow a shared-storage paradigm by using a distributed replicated file system for storing data blocks. Espresso uses local shared nothing storage and log shipping between masters and slaves with automatic failover, similar to MongoDB. This guarantees that queries are always served out of local storage and delivers better latency on write operations.

The multi-DC operation of Espresso differs significantly from other systems. Eventually consistent systems like Voldemort [6] and Cassandra [17] implement quorums that span geographic regions. MegaStore and Spanner implement synchronous replica maintenance across data centers using Paxos. PNUTS implement record level mastership and allows writes only on the geographic master. Espresso relaxes consistency across data centers and allows concurrent writes to the same data in multiple data centers relying on the application layers to minimize write conflicts. It then employs conflict detection and resolution schemes within the system to ensure that data in different data centers eventually converges.

11. CONCLUSION

We have described Espresso, a distributed document-oriented database that emerged out of our need to build a primary source-of-truth data store for LinkedIn. Espresso is timeline consistent, provides rich operations on documents including transactions over related documents, secondary indexed access and supports seamless integration with nearline and offline environments. Espresso has been in production since June 2012 serving several key use-cases.

We are in the process of implementing several additional Espresso features such as support for global secondary indexes, accepting writes in multiple data centers with multiple master replicas and supporting more complex data structures such as lists and sets natively. As we scale our application footprint, we're also starting to work on multi-tenancy challenges to simplify our capacity planning and hardware footprint story.

Acknowledgement

Many other members of the Linkedin Data Infrastructure team helped significantly in the development and deployment of Espresso. We would like to thank the following people for their invaluable contributions: Chavdar Botev, Phanindra Ganti and Boris Shkolnik from the Databus team worked very closely with us. Eun-Gyu Kim and Krishna Puttaswamy Naga have joined the Espresso team recently.

We have also benefited from the contributions of Adam Silberstein, Milind Bhandarkar, Swee Lim, and Jean-Luc Vaillant. Our SRE team, Kevin Krawez, Zachary White, Todd Hendricks, Kavita Trivedi, David DeMaagd, Jon Heise and Brian Kozumplik, have been key partners supporting Espresso in production at LinkedIn. Finally special thanks to our early adopters in the company, the Comm team, the USCP team, and the BizProfile team. None of this would have been possible without the executive sponsorship and support provided by Kevin Scott, David Henke and Jeff Weiner.

12. REFERENCES

[1] Apache HBase. http://hbase.apache.org/.

[2] Couchbase. http://www.couchbase.com/.

[3] Facebook inc. http://www.facebook.com.

[4] Google inc. http://www.google.com.

[5] MongoDB. http://www.mongodb.org/.

[6] Project Voldemort. http://project-voldemort.com/.

[7] Twitter. http://www.twitter.com.

[8] A. Auradkar, C. Botev, S. Das, D. De Maagd, A. Feinberg, P. Ganti, L. Gao, B. Ghosh, K. Gopalakrishna, B. Harris, et al. Data infrastructure at linkedin. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1370–1381. IEEE, 2012.

[9] J. Baker, C. Bond, J. C. 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 *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[10] E. Brewer. Towards robust distributed systems. In *Proceedings of the 19th Anniversary of the ACM Symposium on Principles of Distributed Computing (PODC 00)*, pages 7–10, 2000.

[11] F. Chang et al. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[12] 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. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[13] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[14] S. Das, C. Botev, K. Surlaker, et al. All aboard the Databus! LinkedIn's scalable consistent change data capture platform. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC 2012)*. ACM, 2012.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[16] K. Gopalakrishna, S. Lu, Z. Zhang, K. Surlaker, et al. Untangling cluster management with Helix. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC 2012)*. ACM, 2012.

[17] A. Lakshman, P. Malik, and K. Ranganathan. Cassandra: A structured storage system on a P2P network. In *SIGMOD*, 2008.

[18] J. Shute, M. Oancea, S. Ellner, B. Handy, E. Rollins, B. Samwel, R. Vingralek, C. Whipkey, X. Chen, B. Jegerlehner, K. LittleïñĄeld, and P. Tong. F1 - the fault-tolerant distributed rdbms supporting google's ad business. In *SIGMOD*, 2012. Talk given at SIGMOD 2012.