All Aboard the Databus!

Linkedin's Scalable Consistent Change Data Capture Platform

Shirshanka Das, Chavdar Botev, Kapil Surlaker, Bhaskar Ghosh
Balaji Varadarajan, Sunil Nagaraj, David Zhang, Lei Gao
Jemiah Westerman, Phanindra Ganti, Boris Shkolnik, Sajid Topiwala
Alexander Pachev, Naveen Somasundaram and Subbu Subramaniam

LinkedIn
Mountain View, CA, USA
{sdas,cbotev,ksurlaker,bghosh,bvaradar,snagaraj,dzhang, lgao, jwesterman}@linkedin.com
{pganti,bshkolnik,stopiwala,apachev,nsomasundaram,ssubramaniam}@linkedin.com

ABSTRACT

In Internet architectures, data systems are typically categorized into source-of-truth systems that serve as primary stores for the user-generated writes, and derived data stores or indexes which serve reads and other complex queries. The data in these secondary stores is often derived from the primary data through custom transformations, sometimes involving complex processing driven by business logic. Similarly data in caching tiers is derived from reads against the primary data store, but needs to get invalidated or refreshed when the primary data gets mutated. A fundamental requirement emerging from these kinds of data architectures is the need to reliably capture, flow and process primary data changes.

We have built Databus, a source-agnostic distributed change data capture system, which is an integral part of LinkedIn's data processing pipeline. The Databus transport layer provides latencies in the low milliseconds and handles throughput of thousands of events per second per server while supporting infinite look back capabilities and rich subscription functionality. This paper covers the design, implementation and trade-offs underpinning the latest generation of Databus technology. We also present experimental results from stress-testing the system and describe our experience supporting a wide range of LinkedIn production applications built on top of Databus.

Categories and Subject Descriptors: H.3.4 [Systems and Software]: Distributed systems
General Terms: Design, Algorithms, Replication, Change Data Capture, CDC, Stream Processing

1. INTRODUCTION

As most practitioners in the domain of data management systems are discovering, One Size doesn't Fit All [13]. Today most production data management systems are following something similar to a fractured mirrors approach [12]. Primary OLTP data-stores take user facing writes and some reads, while other specialized systems serve complex queries or accelerate query results through caching. The most common data systems found in these architectures include relational databases, NoSQL data stores, caching engines, search indexes and graph query engines. This specialization has in turn made it critical to have a reliable and scalable data pipeline that can capture these changes happening for primary source-of-truth systems and route them through the rest of the complex data eco-system.

There are two families of solutions that are typically used for building such a pipeline.

- *Application-driven Dual Writes*: In this model, the application layer writes to the database and in parallel, writes to another messaging system. This looks simple to implement since the application code writing to the database is under our control. However it introduces a consistency problem because without a complex coordination protocol (e.g. Paxos [10]

SOCC'12, October 14-17, 2012, San Jose, CA USA

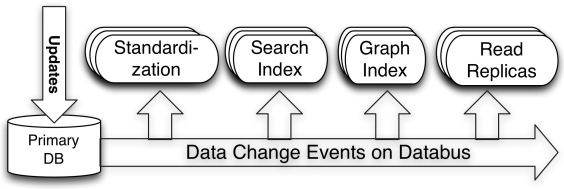

Figure 1: LinkedIn: Databus applications

or 2-Phase Commit [8]) it is hard to ensure that both the database and the messaging system are in complete lock step with each other in the face of failures. Both systems need to process exactly the same writes and need to serialize them in exactly the same order. Things get even more complex if the writes are conditional or have partial update semantics.

- *Database Log Mining*: In this model, we make the database the single source-of-truth and extract changes from its transaction or commit log. This solves our consistency issue, but is practically hard to implement because databases like Oracle and MySQL (the primary data stores in use at LinkedIn) have transaction log formats and replication solutions that are proprietary and not guaranteed to have stable on-disk or on-the-wire representations across version upgrades. Since we want to process the data changes with application code and then write to secondary data stores, we need the replication system to be user-space and source-agnostic. This independence from the data source is especially important in fast-moving technology companies, because it avoids technology lock-in and tie-in to binary formats throughout the application stack.

After evaluating the pros and cons of the two approaches, we decided to pursue the "log mining" option, prioritizing consistency and "single source of truth" over ease of implementation.

In this paper, we introduce Databus, LinkedIn's Change Data Capture pipeline, which supports Oracle and MySQL sources and a wide range of downstream applications. The Social Graph Index which serves all graph queries at LinkedIn, the People Search Index that powers all searches for members at LinkedIn and the various read replicas for our Member Profile data are all fed and kept consistent via Databus. In the rest of the paper, we focus on Databus' unique architectural features:

- Pull-model
- Externally-clocked
- Consumption from arbitrary point in the change stream
- Isolation between sources and consumers: both in terms of performance and the specifics of the implementations

While building Databus, we have encountered some practical design and implementation issues which we also cover in this paper. In particular, we discuss implementation details and performance measurements around:

- Event definition, serialization and portability
- Minimizing load on the data source
- Filter push-down and partitioned consumption
- Achievements in scalability, high-availability and low latency

The next section dives deeper into the requirements that drove us towards the design and implementation decisions outlined above.

2. REQUIREMENTS

We had the following requirements while designing Databus.

I *No additional point of failure*: Since the data source is the source of truth, we want to ensure that the pipeline does not introduce a new point of failure in the architecture.

II *Source consistency preservation*: We want to preserve the consistency semantics that the source provides. This implies needing to support the strongest consistency semantics possible. We do not support cross database transactions in the source. For transactional sources, to avoid having the subscribers see partial and/or inconsistent data we need to capture:

- *Transaction boundaries*: a single user's action can trigger atomic updates to multiple rows across tables.
- *Commit order*: the exact order in which operations happened on the primary database.
- *Consistent state*: we can miss changes but cannot provide a change-set that is not consistent at a point in the commit order. e.g. if a row got updated multiple times in quick succession, it is okay to miss an intermediate update, but not okay to miss the last update.

III *User-space processing*: By "user-space processing", we refer to the ability to perform the computation triggered by the data change outside the database server. This is in contrast to traditional database triggers that are run in the database server. Moving the computation to user space has the following benefits:

- Reduces the load on the database server
- Avoids affecting the stability of the primary data store
- Decouples the subscriber implementation from the specifics of the database server implementation
- Enables independent scaling of the subscribers

IV *No assumptions about consumer uptime*: Most of the time, consumers are caught up and processing at full speed. However, consumers can have hiccups due to variance in processing time or dependency on external systems, and downtime due to planned maintenance or failures. Sometimes, new consumers get added to increase capacity in the consumer cluster and need to get a recent snapshot of the database. In other cases, consumers might need to re-initialize their entire state by reprocessing the whole data set, e.g. if a key piece of the processing algorithm changes. Therefore, we need to support the capability to go back to an arbitrary point in time.

V *Isolation between Data-source and consumers*: Consumers often perform complex computations that may not allow a single instance to keep up with the data change rate. In those cases, a standard solution is to distribute the computation among multiple instances along some partitioning axis. Therefore, the pipeline should

- Allow multiple subscribers to process the changes as a group; i.e. support partitioning;
- Support different types of partitioning for computation tasks with different scalability requirements;
- Isolate the source database from the number of subscribers so that increasing the number of the latter should not impact the performance of the former;
- Isolate the source database from slow or failing subscribers that should not negatively impact the database performance;
- Isolate subscribers from the operational aspects of the source database: database system choice, partitioning, schema evolution, etc.

VI *Low latency of the pipeline*: Any overhead introduced by the pipeline may introduce risk of inconsistencies, negatively affect performance, or decrease the available time for the asynchronous computations. For example, any latency in updating a secondary index structures (like the previously mentioned LinkedIn social graph index) increases the risk of serving stale or inconsistent data. In the case of replication for read scaling, pipeline latency can lead to higher front-end latencies since more traffic will go to the master for the freshest results.

VII *Scalable and Highly available*: We need to scale up to thousands of consumers and support thousands of transaction logs while being highly available.

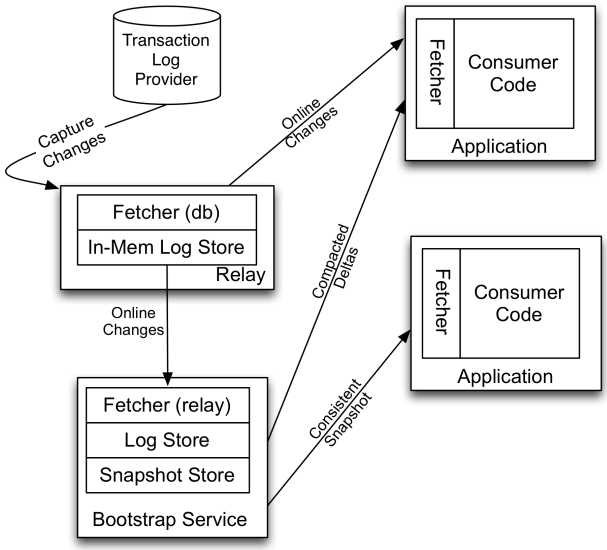

Figure 2: Databus Architecture

3. ARCHITECTURE

The Databus architecture is split into four logical components.

- a *fetcher* which extracts changes from the data source or another Databus component,
- a *log store* which caches this change stream,
- a *snapshot store* which stores a moving snapshot of the stream, and
- a *subscription client* which pulls change events seamlessly across the various components and surfaces them up to the application.

The typical deployment architecture for these components is shown in Figure 2. We collocate the fetcher and an in-memory log store in a process we call the *relay* process. We additionally collocate a persistent log store and the snapshot store in a process we call the *bootstrap service*. The subscription client is a library that is linked into the application that needs to consume changes from the stream. For a new type of data source technology (say PostgreSQL), the only component that needs to change in this architecture is the implementation of the db fetcher.

3.1 Consistency Semantics

Databus supports transactional semantics across multiple types of entities within a transactional datastore. For example, it can annotate and propagate transactions that span multiple tables within the same database. It supports guaranteed at-least once delivery semantics by default. An event may be delivered multiple times only in the case of failures in the communication channel between the relay and the client, or in case of a hard failure in the consumer application. Consumers therefore need to be idempotent in the application of the delivered events or maintain transactional commit semantics aligned with the source.

3.2 External Clock and Pull Model

The overarching design principle of Databus is that it is simply a lossless transporter of changes that have been committed upstream. Each change set is annotated with a monotonically increasing system change number (SCN). This number is assigned by the data source and typically system specific. For example, Oracle attaches an SCN to each commit, while MySQL's commits can be identified uniquely through their offset in the binary log (binlog). As these changes flow through the ecosystem, derived state gets created by consumers and is associated back to the change stream through this number. This is important not just for auditing, but also to recover from failures and resume processing without missing any changes. The system therefore is externally clocked. All parts of the Databus infrastructure track the lineage of data records and the progress of consumers using only the SCN of the external system. Physical offsets are only used as optimizations in internal transport but are never used as source of truth.

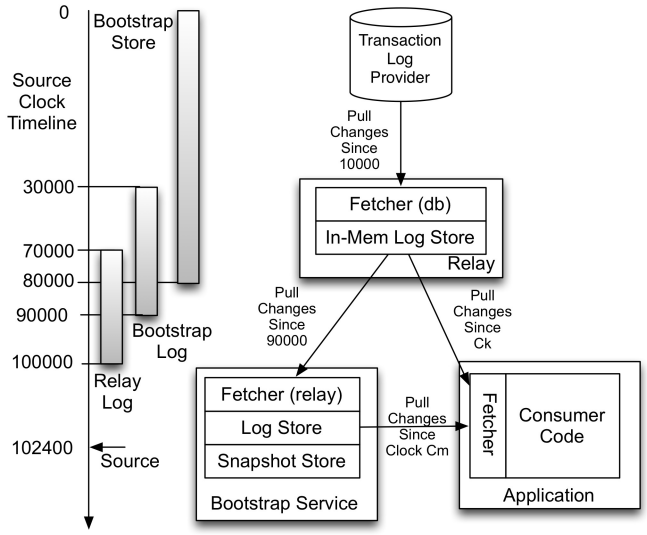

Figure 3: Pull Model and Source Clock

There are many advantages of favoring pull over push in this particular context. We need to support a large number of consumers and allow them to consume from any arbitrary point in time. The most natural place to keep state of how much of the SCN timeline has been really processed is the consumer, because that is where the processing is taking place. Keeping state at the consumer naturally leads to a pull-based data transfer model. The process can be entirely governed by the consumers; at any moment, they may decide to rollback to a previous point. Thus, all Databus components have to be stateless.

Further, the requirements for not introducing new points of failures and source consistency preservation led us to adopt a pull data transfer model for the DB fetcher. In a hypothetical push-based model, failures in pushing the changes from the DB fetcher to the relay will either lead to aborted transactions (violating the first requirement) or changes that are lost by Databus (violating the second requirement).

Figure 3 shows the interactions between the different fetcher components and the source clock across the relay, bootstrap log and snapshot store. In this example, the source has generated changes until SCN 102400, the relay has an in-memory buffer that covers all changes from 70000 to 100000, the bootstrap service's persistent log covers all changes from 30000 to 90000, and the bootstrap service's snapshot store covers all changes from 0 to 80000. Depending on where the consumer is currently at, it will pull from either the relay or the bootstrap service.

In addition to supporting simple restart and recovery semantics, this design also makes it very easy to integrate with non-Databus based data flows. For example, an application can take a dump of an Oracle database, perform arbitrary processing on that dump, and then seamlessly tap into the Databus stream to get all changes that have happened to the database without missing a single change. The only requirement is that the original Oracle dump should be stamped with the same clock (Oracle's SCN system) that the Oracle Databus fetcher uses to pull changes out from Oracle.

3.3 Source Data Extract

Databus has been designed to support extraction from different data sources. This flexibility is achieved by allowing different fetchers to be developed and plugged into the pipeline. The fetcher is typically an embedded thread that gets run from within a relay and must follow a few semantic constraints. As described above, each change set must be annotated with a monotonically increasing clock value that we refer to as the system change number (SCN) of the change set. This mapping from change set to SCN is immutable and assigned at commit time by the data source, therefore Databus can run multiple independent fetchers following the same timeline of changes without any need for synchronization. The fetcher is initialized with an SCN on startup, and must start pulling changes that are newer than this SCN from the data source. This in turn adds a "rewindability" requirement on the data source. The data source must be able to keep enough history in its change log or the fetcher must be written in a way that supports going back to pull from an arbitrary point in time. In practice, this requirement does not add any extra complexity on the source, but depending on the implementation, it can have a performance impact on the online writes that are happening on the data store. The way the Oracle fetcher is written,

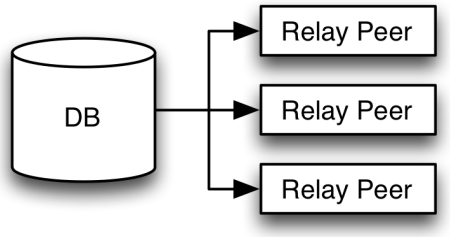

Figure 4: Independent Relay Deployment

it can go back all the way to time zero, but since it queries the in-database tables, these queries get progressively expensive and impact the OLTP workload on the system. The MySQL fetcher mines the binary log directly, and can therefore rewind back to as much time as the storage on the MySQL machine will allow to retain without any performance penalty on the OLTP workload. We talk in detail about the implementation of these two fetchers in Section 4. Overall, this centralization of complexity onto the fetcher component leads to very simple persistence and failure-recovery protocols downstream. At the same time, fault-tolerance and high-availability in the relay tier and the presence of the snapshot store reduces the likelihood of having to go back in time on the data source, thus keeping the fetcher overhead low.

3.4 The Relay

The Databus relay hosts a fetcher, a transient log and an HTTP server within a single process. As described earlier, the fetcher is a pluggable entity and can be used to fetch changes from a source or from another relay. The pluggability allows us to arrange relays in fan-out tree configurations for scaling out.

The changes extracted by the fetcher are serialized into a binary format that is independent of the data source. They are then grouped together within transaction window boundaries, annotated with the clock value or SCN associated with the transaction and buffered in the transient log. These changes are handed out to consumers when they request them. The consumers request changes based on the source clock, asking for all changes since SCN N where N is the last SCN for which changes have been processed at the consumer.

The relay does not maintain consumer-related state. Each consumer application progress is tracked through a consumer checkpoint maintained by the subscription client and passed on each pull request. Since all relays follow the same change stream timeline (determined by the commit order in the data source), the checkpoint is portable across relays.

On the one hand, the above simplifies the relay's implementation and the consumer fail-over from one relay to another. On the other hand, it causes an additional problem that the relay does not know if a given change set has been processed by all interested consumers. We therefore institute a time or size-based retention policy at the relay tier to age out old change sets. In practice, we provision the relays sufficiently using memory-mapped buffers to ensure that most consumers can consume the whole stream directly from the relay even while encountering minor hiccups, planned or unplanned downtime. The bootstrap service described in Section 3.5 is the insurance policy in case of unforeseen downtime or if full data set reprocessing is required. We next describe the deployment models that are used in setting up relay clusters.

3.4.1 Relay Cluster Deployment

Typical Databus deployments consist of a cluster of relay servers that pull change streams from multiple data sources. Each relay server can connect to multiple data source servers and host the change stream from each server in separate buffers in the same relay. The relays are also set up in such a way that the change stream from every data source is available in multiple relays, for fault-tolerance and for consumer scaling. There are two configurations the relays are typically deployed.

In one deployment model as shown in Figure 4, all the relay servers hosting a stream connect to the stream's data source directly. Each relay server is assigned a subset of all the streams being hosted by the relay cluster. The relay connects to the specified data source server and pulls the change streams. When a relay fails, the surviving relays continue pulling the change streams independent of the failed relay. If the configured redundancy factor is R, this model provides 100% availability of the streams at very low latency as long as all the R relays that connect to the same data source server do not fail at the same time. This however comes as the cost of increased load on the data source server since there are R relays that pull the same change stream.

To reduce the load on the source data source server, an alternative deployment model for relays as shown in Figure 5, is the Leader-Follower model. In this model, for every data source server, one relay is designated to be the leader while R-1 are

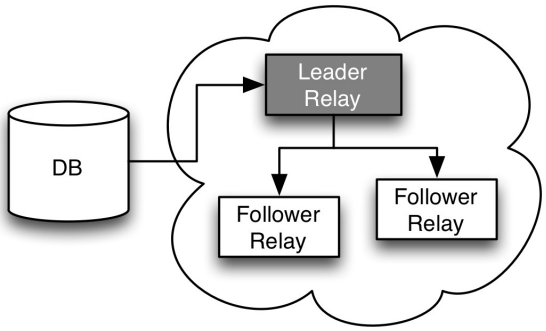

Figure 5: Leader-Follower Relay Deployment

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 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 data source server 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 the data source server are not available to the consumers.

To expand the capacity of the relay cluster, new relay servers can be added. When this happens, a new assignment of data source servers to relay servers is generated so that some streams are transferred from the old relay servers to the new relay servers. The new relay servers then connect to the data source servers and start pulling the change streams. They can optionally copy the existing change streams from the old relay servers before connecting to the data source servers. This management of the relay cluster is built on top of a generic cluster management framework built at LinkedIn called Helix.

The assignment of data sources to relays is made available to Databus consumers in the form of a routing table so that the clients can discover the location of the streams they wish to consume. When the assignment changes due to relay failures or relay cluster rebalancing, this routing table is automatically updated and the consumers switch to the new servers transparently.

3.5 The Bootstrap Service

As we have described previously, consumers typically subscribe to changes from the relay, which maintains an in-memory log store. Occasionally, there are situations in which the consumers might fall significantly behind in their processing. This usually happens because of consumer failures, which cause them to be offline for an extended period of time. In other cases, new consumers are launched and they need to bootstrap their initial state before consuming the change log from the relay.

Possible approaches for dealing with these situations are going back to the source OLTP database and storing extended logs at the relays. The first approach is not acceptable since it leads to greatly increased load on the database that is serving online traffic. Besides, getting a consistent snapshot of all the rows in the OLTP database by running a long running query is very difficult. Storing the log at the relay for extended periods of time is not always viable since if the consumer has fallen behind a lot, consuming every change event is likely to be slow and unnecessary. It is much more efficient to catch up using a snapshot store which is a compacted representation of the changes i.e. only the latest state of every affected row needs to be consumed.

Databus implements this functionality using a Bootstrap Service. As shown in Figure 2, the bootstrap service consists of three components:

- a *bootstrap database*: This has two parts. One is a persistent log store that maintains the change log for an extended time. The other is a snapshot store of the data that represents a view of the database at a given point in time.
- a *bootstrap producer*: This is really just a fetcher that subscribes to the change log from the relay and writes it to the log store in the bootstrap database.
- and a *bootstrap applier*: This is just another fetcher that pulls from the log store and periodically merges the changes into the snapshot.

In the above design, the combination of both a snapshot store and a log store is what ensures the ability of the Bootstrap

Service to provide consistent snapshots to the consumer applications from arbitrary points in the change stream. It is important to note that the snapshot store itself is not sufficient to achieve this goal as explained below.

Getting a consistent read of the snapshot by locking the snapshot store is not practical. For big data sets, it may take many hours for a consumer application to read and process the snapshot. If there are multiple consumers trying to bootstrap, the application of new changes to the snapshot store may be suspended indefinitely. Further, if the entire snapshot is read in one humongous batch, a small processing error may require restarting of the entire bootstrapping process. Instead, the Bootstrap Service needs to allow the consumer application to bootstrap the data in manageable batch sizes while new changes are applied to the snapshot store. Thus at the end of reading the snapshot, it may not be consistent – the consumer application may have been observed some of the new changes while it may have missed others. To ensure consistency, the Bootstrap Service has to replay all the changes that have happened from the point when the snapshot read started. Thus, the need for a log store to buffer such changes.

Further, splitting the responsibilities between the relay fetcher and the log store fetcher ensures that the Bootstrap Service has enough write throughput to keep up with the data source: appending to the log store is much cheaper than building the snapshot store. Peaks in the write traffic are handled gracefully by letting the snapshot store lag slightly behind the newest changes.

The full bootstrapping algorithm with support for bootstrapping of multiple sources is described below.

Algorithm 1 Bootstrap Consumption

bootstrap(*sources*)
  startScn = current scn of the bootstrap db
  for $i = 0$ to *sources*.length **do**
Require: Source S_j $(j < i)$ is consistent as of *startScn*
    {Begin Snapshot phase}
    Get all rows from S_i where $rowScn < startScn$
    {Can be done in batches}
    $targetScn$ = max SCN of rows in S_i {Begin Catchup phase}
    for all source Sj such that $j \leq i$ **do**
      Get all rows from S_j log store from startScn until targetScn
    end for
    $startScn = targetScn$
  end for

3.6 Event Model and Consumer API

There are two versions of the consumer API, one that is callback driven and another that is iterator-based. At a high-level, there are eight main methods on the Databus callback API.

- *onStartDataEventSequence*: the start of a sequence of data events from an events consistency window.
- *onStartSource*: the start of data events from the same Databus source (e.g. Oracle table).
- *onDataEvent*: a data change event for the current Databus source.
- *onEndSource*: the end of data change events from the same Databus source.
- *onEndDataEventSequence*: the end of a sequence of data events with the same SCN.
- *onCheckpoint*: a hint from the Databus client library that it wants to mark the point in the stream identified by the SCN as a recovery point
- *onRollback*: Databus has detected a recoverable error while processing the current event consistency window and it will rollback to the last successful checkpoint.
- *onError*: Databus has detected a unrecoverable error and it will stop processing the event stream.

.

The above callbacks denote the important points in the stream of Databus change events. A typical sequence of callbacks follows the pattern below.

```
onStartDataEventSequence(startSCN)
    onStartSource(Table1)
```

```
        onDataEvent(Table1.event1)
                ...
        onDataEvent(Table1.eventN)
    onEndSource(Table1)
    onStartSource(Table2)
        onDataEvent(Table2.event1)
              ...
        onDataEvent(Table2.eventM)
    onEndSource(Table2)
        ...
onEndDataEventSequence(endSCN)
```

Intuitively, the Databus client communicates with the consumer: "Here is the next batch of changes in the watched tables (sources). The changes are broken down by tables. Here are the changes in the first table, then the changes to the next table, etc. All the changes represent the delta from the previous consistent state of the database to the following consistent state."

The contract on all of the callbacks is that the processing code can return a result code denoting a successful processing of the callback, recoverable or unrecoverable error. Failures to process the callback within the allocated time budget or throwing an exception, result in a recoverable error. In cases of recoverable errors, the client library will rollback to the last successful checkpoint and replay the callbacks from that point.

The offloading of state-keeping responsibility from the consumer simplifies the consumer recovery. The consumer or a newly spawned consumer can rewind back to the last known good checkpoint. For instance, if the consumer is stateful, they just need to tie the state that they are keeping with the checkpoint of the stream. On failure, the new consumer can read the state and the checkpoint associated with it. With this, it may start consuming from that point. If the stream consumption is idempotent, then the checkpoint can be maintained lazily as well.

3.7 Partitioned Stream Consumption

Databus supports independent partitioning of both the data source and the consumers. This allows independent scaling of the data source tier from the processing tier. For example, we have monolithic data sources which emit a single stream, but are consumed by partitioned search indexes. Conversely, there are partitioned data stores which are consumed as a single logical stream by downstream consumers. The partitioning cardinality can change over time on either side without affecting the other side. In the case of partitioned data sources, Databus currently enforces transactional semantics only at the partition level, and provides in-order and at-least once guarantees of delivery per source partition.

There are three primary categories of partitioning scenarios:

1. *Single consumer*: A consumer that is subscribing to the change stream from a logical database must be able to do so independently of the physical partitioning of the database. This is supported by just doing a merge of the streams emanating from the source database partitions.

2. *Partition-aware consumer*: The consumer may have affinity to certain partitions of the stream and want to process only events that affect those partitions. This form of partitioning is implemented through server-side filters and includes mod-based, range-based and hash-based partitioning. This filtering is natively supported in the Relay and the Bootstrap Service. The consumers specify the partitioning scheme to be used as part of their registration in the Subscription Client. The Subscription Client then includes it as part of the pull requests to the Relay or the Bootstrap Service. The Bootstrap Service may apply further optimization like predicate push-down so that only the necessary events are read from the persistent log or snapshot stores.

3. *Consumer groups*: The change stream may be too fast for a single consumer to process and the processing needs to be scaled out across multiple physical consumers who act as a logical group. This is implemented by using a generic cluster management framework called Helix. The partitions of the database are assigned to the consumers in the same group so that every partition has exactly R consumers in the group assigned to it and the partitions are evenly distributed among the consumers. When any consumer fails, Helix rebalances the assignment of partitions by moving the partitions assigned to the failed consumer to the surviving consumers. If the existing consumers in the group are not able to keep up with the stream, additional consumers might be added to the group to scale the processing. In this case, the assignment of partitions is rebalanced so that some partitions from each existing consumer are moved to the new consumers.

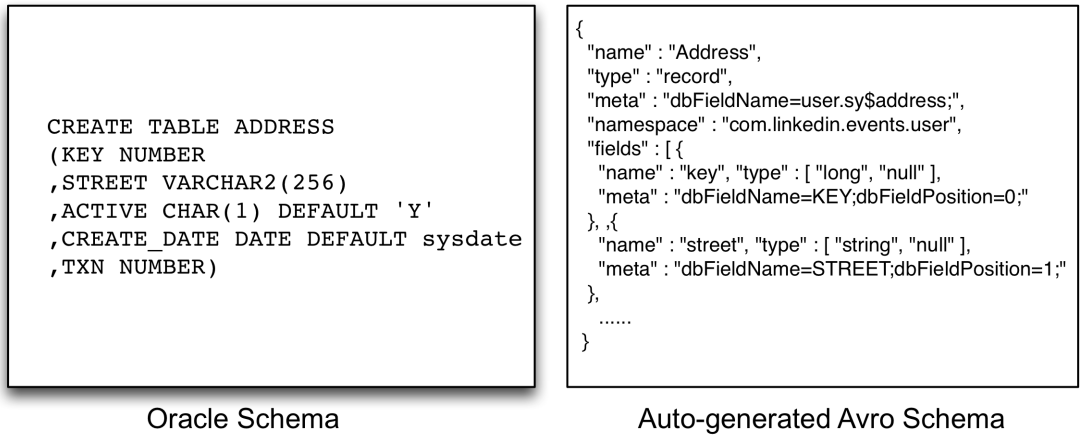

Figure 6: Oracle table mapped to Avro schema

3.8 Metadata

Databus uses Avro for serialization of events and supports schema versioning and evolution. This provides an isolation layer that protects consumers from upstream changes in the schema.

The fetchers infer the Avro schema by inspecting the source tables. Figure 6 shows a simple Oracle table mapped to an Avro schema. There are some intricacies when mapping data-types, e.g. when you have a column defined as NUMBER in Oracle, which one of int, long, float do you use when referring to the column in Avro? Once a schema is inferred, the relay is notified about it. When the fetcher generates new events from this table, it serializes the events with the Avro schema. When a table evolves, the schema for it changes and Databus attaches a newer version identifier to it. All new events are now serialized with this new schema version identifier.

Databus ensures that changes to the source schema do not affect consumers and they can upgrade at their own cadence. The Databus client library performs automatic schema conversion using the standard Avro schema resolution rules [2] . We do not require reserialization of all older events generated since the beginning of time when the schema evolves because all versions of the schema are kept around forever. The schema resolution to an older version of the event schema is performed dynamically at the callback to the consumer.

4. IMPLEMENTATION NOTES

4.1 Oracle Adapter

Oracle provides replication support between Oracle databases using DataGuard. Additionally, there are commercial products like GoldenGate (from Oracle) that make the change log from Oracle available to external applications. In the absence of available open-source technology to solve this problem, a company has two options. It can either license the commercial solution with the associated fees and feature availability or try to develop and support its own solution. At the time LinkedIn was evaluating change capture technologies, GoldenGate lacked required features like BLOB/CLOB support. LinkedIn has implemented an Oracle change-capture mechanism based on triggers which we describe next.

A simple approach to get the change log from Oracle is to have a timestamp column with every row. A trigger on the table updates the timestamp column with the current time on an insert or update to the row as shown in Figure 7. The adapter then issues a query to the database to get all the changed rows.

This however has a problem. Timestamps in this mechanism are set at the time of the change to the row, not at transaction commit. Long running transactions might have rows that changed long before the transaction finally commits. Thus, this query will miss changes from the database. For example in Figure 8, tx2 commits before tx1 but t2 > t1. If the query happens between the two commits, lastTimeStamp is t2 and tx1 is missed. We can try some padding to reread rows that changed since $lastTimeStamp - n$ seconds but this is very error prone.

Oracle 10g and later versions provide a feature that provides the ora_rowscn pseudo column which contains the internal Oracle clock (SCN - system change number) at transaction commit time. By default, ora_rowscn is available at the block granularity but tables can be created with an option to provide ora_rowscn at the row granularity. We can now query the database to fetch all rows that changed since the last rowscn but unfortunately ora_rowscn is not an indexable column. To

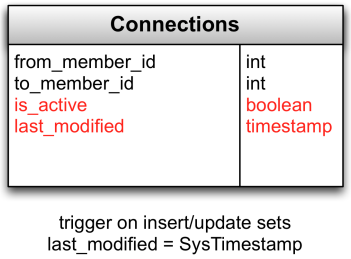

Figure 7: Timestamp based CDC attempt

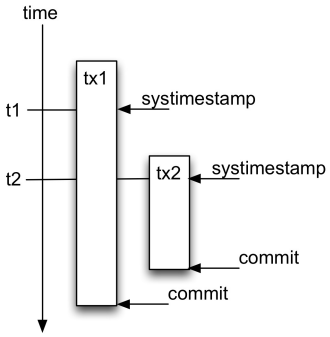

Figure 8: Timestamp based CDC: commit reordering

get around this problem, we add a regular column scn to the table and create an index on the column. The default value of scn is set to infinity. After commit, the ora_rowscn for the affected rows is set. Every so often, we run a statement to update the scn column.

```
update T set scn = ora_rowscn
where scn = infinity;
```

The query to select the changed rows since lastScn now becomes

```
select * from T
where scn > lastScn
AND ora_rowscn > lastScn;
```

This works to get changes from a single table. However, transaction can span multiple tables in a database and these changes need to be transported to the consumer while preserving the transaction boundary. To solve this, we add a per database table TxLog that has the indexed scn column. We add a txn column to all the other tables that we wish to get changes from. We have a trigger than allocates txn from a sequence and adds an entry to the TxLog table on every transaction commit as shown in Figure 9.

Changes can now be pulled using the following query

```
select src.* from T src, TxLog
```

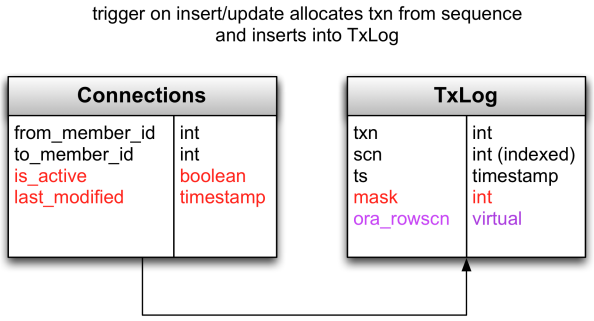

Figure 9: Trigger based CDC

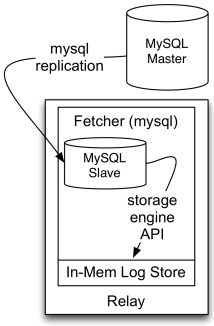

Figure 10: MySQL Adapter

```
where scn > lastScn
AND ora_rowscn > lastScn
AND src.txn = TxLog.txn;
```

The trigger-based approach used by the Oracle adapter has a couple of drawbacks. Firstly, it can miss intermediate changes to rows because it is only guaranteed to return the latest state of every changed row. This is not a correctness problem, but if possible, it is desirable to surface every change made to the row to the consumer. Secondly, triggers and the associated tables that they update cause additional load in terms of reads and writes on the source database.

4.2 MySQL Adapter

One way to address the drawbacks of the trigger-based approach is to interact directly with the transaction log, since that does not take up resources on the source database. Oracle and MySQL both have binary logs that contain the log of changes as they are applied to the database. However, it is fragile to mine these logs and reverse-engineer the structure, because there is no guarantee that the format will be stable across multiple versions. In the case of MySQL though, it is possible to tap into the Storage Engine API. This is a stable interface that has been used to build many commercial and open-source storage engines for MySQL.

However, MySQL replication works only between MySQL databases and does not make the change log available to external applications. The pluggable storage engine layer allows MySQL replication to be setup between MySQL databases that might use different storage engines. Thus, a MySQL server using InnoDB storage engine can replicate to MySQL server using the MyISAM storage engine. MySQL replication takes care of the protocol between master and slave, handling restarts across failures, parsing of the binary log and then calling the appropriate insert, update, delete statements on the slave through the storage engine API. Databus uses this feature of MySQL and obtains the change log from the MySQL master into a custom light-weight storage engine RPL_DBUS that writes to the in-memory log. This architecture is shown in Figure 10. The relay manages the local slave MySQL instance and uses MySQL admin commands to connect to the MySQL master.

4.3 Relay Internals

As described earlier, the relay hosts a transient log store for low-latency serving of the change capture stream. When designing and implementing the relay, we tried to achieve the following goals:

- *Low latency* and *high throughput* for consumer pull requests
- *Low variance* for the latency of consumer pull requests
- *Scalability* to a large number of consumers
- *Multi-tenancy* of multiple change streams on the same relay

To achieve the above goals, we made some interesting design decisions.

- *Wire-format change buffer*: The change data is stored in memory using the on-wire serialization format in contiguous long-lived memory blocks outside the Java heap. This has two major benefits. First, it enables the use of high-throughput bulk-write operations to the network when serving pull requests. Second, it allows us to keep the change data in contiguous long-lived memory blocks outside the Java heap, thus, minimizing the GC pressure (low variance).

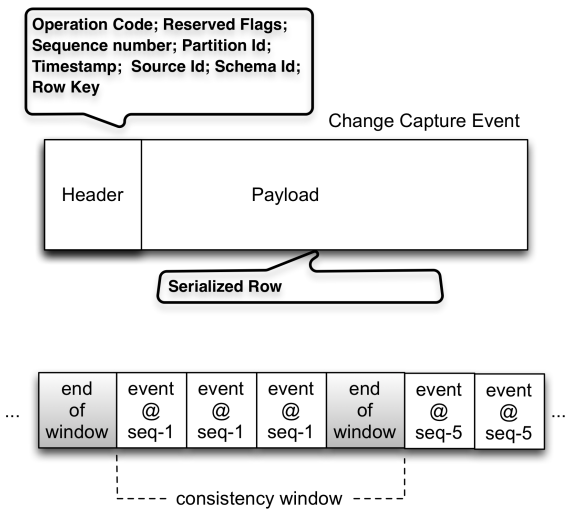

Figure 11: Change Capture Window Format

- *Space-bound change buffer*: In addition to the above wire-format change buffer design, we wanted to support setting limits on how much space the buffer could use. This naturally led us to build it as a circular buffer with the changes pre-serialized and inlined. This supports very fast insert performance and also supports low latency range scans which are required for servicing the consumer pull requests. This also means we can easily estimate and control the memory utilization for the change data storage, thus, enabling multi-tenancy.

- *Space-bound indexing*: Consumer pull requests perform range scans over the change data from the SCN of the last fetched change event. To allow efficient serving of such requests, we maintain an SCN index. The index is implemented as a skip-list on top of the change buffer. As in the case of the change buffer, the index uses off-heap bounded memory for performance and multi-tenancy. This design also allows us to trade off some additional scan latency for reduced memory usage.

- *Fine-grained locking*: We use range-based locking to maximize read-write throughput for non-overlapping reads and writes. Writes acquire an exclusive lock on the region of the buffer to be over-written which is near the head of the buffer (the oldest events). Reads acquire a non-exclusive lock on a region from their starting read point to the tail of the buffer. Contention can occur only if a consumer is lagging behind and is trying to read older events. In practice, we hardly see any contention in our production workloads because readers and writers are pretty much in lock step most of the time. Further, read locks are generally short-lived (see below) so lagging readers cannot block the writer for a long time. If such readers further request events that have been overwritten, they will be redirected to the Bootstrap Service and will not affect the relay performance.

- *Filtering*: We support server-side filtering by brute-force scanning the changes and streaming out only the events that match the subscriber's filter pattern. With the use of memory-mapped buffers, we avoid double-copying between user-space and file-system, and yet retain the ability to filter out data when we stream it out. The on-wire format of the change buffer incurs some CPU cost when applying the filter. In practice, we have found that the benefits of the chosen buffer implementation outweigh the costs as we are rarely bottle-necked on CPU utilization.

- *No consumer state*: As described in Section 3.4, the relay does not maintain any per-consumer state. All information (e.g. the consumer checkpoint) needed to process a pull request is passed as part of the request. Most of the object allocation is associated with processing short-lived HTTP requests.

- *Short-lived pull requests*: All pull requests contain a limit on the size of the returned data. The subscription client requests only as much data as it can immediately buffer. The relay enforces aggressive timeouts when servicing pull requests. Because of this design decision and the off-heap storage of the change events and indexes, the relay has a trivial promotion rate to the old generation in the Java heap. This, coupled with the use of a CMS garbage collector, allows the relay to avoid long GC pauses and maintain low latency and low-variance response times.

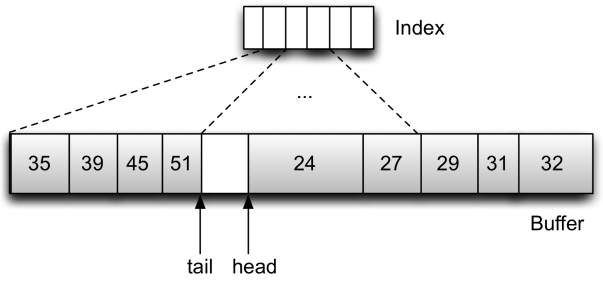

Figure 12: Buffer and Index Data Structures

Figure 11 shows the organization of a single consistency window in the change buffer. We decided to mark explicitly the end of consistency windows in the change stream. First, this allows the consumer to differentiate between the case of empty pull response because of no updates and the case of empty pull response because of no events matching the pull criteria. The former will return no changes; the latter will return the markers for the scanned windows. Thus, the subscription client can avoid processing the same change events on subsequent pulls. Second, end-of-window markers allow us to pass additional meta data (like checksums) about the windows.

Figure 12 summarizes the data structures used to implement the log store.

The consumer pull request is serviced through the *getSince* method in the Relay. The primary input parameters into this call are the consumer's checkpoint, the list of tables they are interested in and any subscription filters that they want to apply additionally on the changes. We first determine the scan offset to begin the scan by consulting the index. We then acquire a read range lock from the scan offset to the tail of the buffer. We then iterate through the buffer streaming out any events that match the filter. The stopping condition is either reaching the tail or hitting the maximum size limit set by the consumer.

4.4 Subscription Client Internals

The Databus subscription client is the glue between the Databus infrastructure (relays, bootstrap service) and the Application (business logic in the consumer). The client is responsible for connecting to the appropriate relay and bootstrap service clusters, keeping track of progress in the Databus event stream, switching over automatically between the Relays and Bootstrap service when necessary, and performing conversions between the schema used for serialization of the payload and the schema expected by the consumer.

The client runs a fetcher thread that is pulling continuously from the relay over a persistent HTTP connection and a dispatcher thread that fires callbacks into a pool of worker threads. There is local flow control between the fetcher and the dispatcher to ensure a steady stream of Databus events to the consumer. Two forms of multi-threaded processing are supported. The first type allows multi-threaded processing within a consistency window only. This ensures that consistency semantics are maintained at the destination. The second type allows multi-threaded processing without regard to the transaction window boundaries. This provides higher throughput at the consumer at the cost of relaxed consistency. The client maintains state of where it is in the sequence timeline through a customizable CheckpointPersister. By default, a checkpoint is persisted to disk for every successfully-processed consistency window. Applications that need very close control of the checkpoint may implement their own storage and restore for checkpoints to tie it to their processing state. For example, search indexes will often commit the checkpoint as meta-data along with the index files themselves so that they share the same fate.

5. EXPERIMENTS

In this section, we present our findings from a set of performance experiments that we ran to test the scalability characteristics of Databus.

5.1 Experimental setup

We ran our experiments on two types of machines:

- Relays and client machines - 12 core 2.56GHz Intel Xeon machines with 48 GB RAM, 1TB SATA RAID 1+0 and 1 Gbps Ethernet
- Bootstrap servers - 12 core 2.40GHz Intel Xeon machines with 48 GB RAM, 800GB 15K SAS RAID 1+0 and 1 Gbps Ethernet

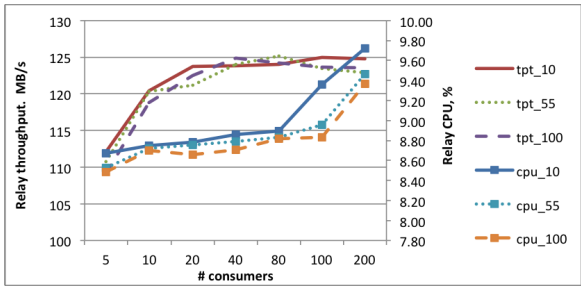

Figure 13: Relay throughput scalability depending on poll interval

There were three main parameters that we varied:

- Event rate - the rate at which change events are being fetched into the relay buffer
- Number of consumers - the number of applications that consume events from the relays and bootstrap service
- Consumer poll interval - the frequency with which consumers poll the relays for new events

For all experiments, we used moderately-sized events with a size of 2.5KB.
The metrics we measured were

- Throughput - the number of events or bytes per second that the relay can push out to the consumers
- E2E event latency - the time it takes for an event to propagate from the relay to the consumers

5.2 Relay

Figure 13 shows the first set of experiments where we measured the maximum relay throughput and CPU utilization with different sets of consumers. We used a relay with already pre-filled buffers and measured the maximum speed at which multiple consumers can read from the relay. We varied the number of consumers and the consumer poll interval. The latter parameter allowed us to test the impact of polling less frequently but in bigger batches of events. We expected that less frequent but bigger batches would allow the relay to support larger number of consumers. The data series prefixed with "tpt" show the relay throughput in MB/s (e.g. "tpt_10" shows the throughput for consumers using 10ms poll interval) with throughput being plotted on the left-hand side "y" axis. The data series prefixed with "cpu" show the relay CPU utilization percentage (e.g. "cpu_55" shows the throughput for consumers using 55 ms poll interval) with the CPU utilization being plotted on the right-hand side "y" axis.

The experiments confirmed our hypothesis. With low poll intervals and fast processing consumers (as the ones that we used for our experiments), we can quickly saturate network bandwidth. We notice that once we get closer to network saturation, the relays do not scale linearly: doubling the number of consumers from 5 to 10 causes an increase of throughput of only 9% (from 110MB/s to 120MB/s). We attribute this to networking overhead. The number of transmitted packets peaked at around 82 thousand packets/s even for 5 consumers and it did not increase with the number of consumers. Thus, increased number of consumers just leads to slightly better utilization of those packets.

Increasing the poll interval has only a small impact on the relay in terms of CPU utilization for small number of consumers. With increased number of consumers the effect becomes more pronounced although still the CPU utilization is fairly low – less than 10%. Overall, the experiments show that network bandwidth becomes a bottleneck much before the CPU. Thus, trading CPU for network bandwidth on the relay side can be expected to pay off. This is further explored in the experiments below.

The second set of experiments aimed at testing how writes to the relay buffer affected throughput. For this set of experiments, we gradually increased the update rate (100 update/s, 1000 updates/s, 2000 update/s) at the data source and measured the consumption throughput (in updates/s) and E2E latency (in milliseconds) at the consumers. Our goal was to detect when bottlenecks on the relay cause the consumers to fall behind the stream of updates from the data sources. We fixed the consumer poll interval at 10ms. Therefore, the consumers can expect on average 5 ms latency due to the poll frequency.

Figure 14 shows the rate at which the consumers were receiving the events. For an event rate of 100 updates/s, the consumption rate stays flat at 100 updates/s, i.e. the consumers are keeping up with the update rate. For event rates of 1000 and 2000 updates/s, we observe two "knees" in the consumption rate when the number of consumers goes above 40 and 20

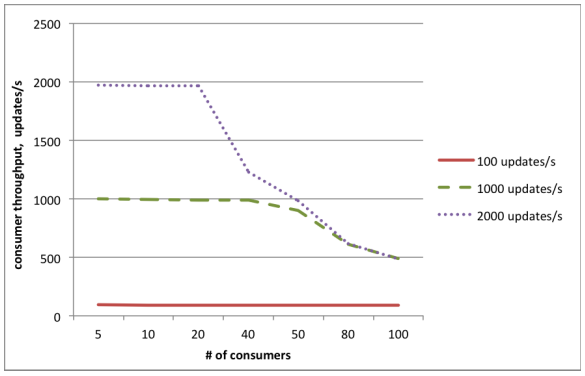

Figure 14: Throughput at consumers when varying update rate

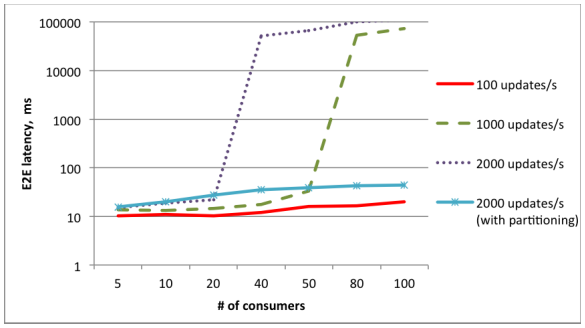

Figure 15: Latency to consumers

consumers respectively. These are the points at which the consumers are no longer able to keep up with the update stream. This behavior can again be explained by network bandwidth saturation: 40 consumers at 1000 updates/s utilize 100MB/s, close to the maximum network bandwidth.

We observe a similar pattern when studying the E2E event latency on Figure 15. At 100 updates/s, the E2E latency (including the poll interval) increases from 10ms to 20ms as the number of consumers grew. For 1000 and 2000 updates/s, we observe similar "knees" as the previous graph at 40 and 20 consumers. Up until those thresholds the E2E latency stays between 15 and 20ms and increases without recovery after that.

For the experiment on Figure 15, we also tested the effect of consumer partitioning using server-side filtering; each consumer only subscribes to a portion of the change stream. We wanted to remove the outbound network bandwidth bottleneck and see if the relay can scale to more consumers. The results are shown in the "2000 updates/s (with partitioning)" data series. For the experiment, we used a simple partitioning scheme where the key space is uniformly partitioned among all available consumers (a fairly typical scenario). In this case, the required outbound bandwidth is equal to the incoming data bandwidth and does not vary with the number of consumers. Although the E2E latency went up to a range of 15 to 45ms due to the additional server-side processing, the rate of increase in E2E latency is very gradual. Thus, we can conclude that the relay can scale to hundreds of consumers if the client application can tolerate slightly higher E2E latency.

5.3 Bootstrap service

We decided to setup our bootstrap performance experiments differently than our relay performance experiments. Firstly, the E2E latency metric used in the latter is not meaningful since the bootstrap service by definition serves older change records. Secondly, our initial bootstrap experiments showed that bootstrap servers can easily saturate the network bandwidth when bootstrapping multiple consumers, similarly to the relay experiments.

Instead, we focused on the more interesting aspect of the bootstrap service: the ability to serve compacted deltas of events from the snapshot store versus replaying all updates from the log store. On the one hand, serving from the log store utilizes very efficient sequential disk reads but has to return all change records. On the other hand, serving from the snapshot store allows the bootstrap service to stream out fewer records (because only the latest change record for a given key is returned) at the cost of a less efficient disk access pattern.

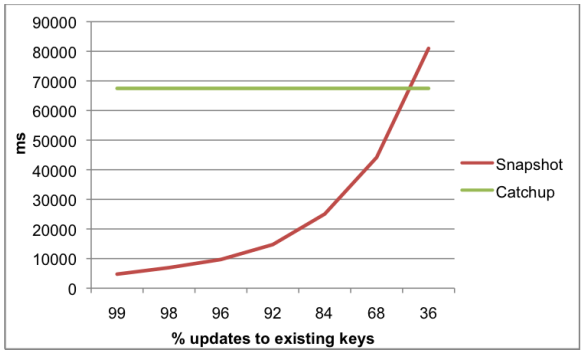

Figure 16: Time to bootstrap 1 hour of changes using a Snapshot delta or Catch-up log

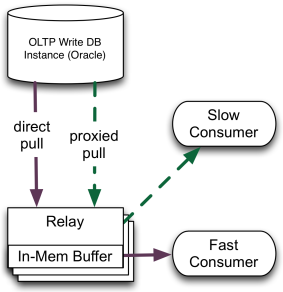

Figure 17: LinkedIn: Databus Architecture circa 2007

The goal of our experiments was to find the break even point for the two access patterns. This helps us tune the retention policy in the bootstrap service and to implement an optimization in the bootstrap service where the snapshot phase can be bypassed if it is more beneficial to serve the bootstrap data entirely from the bootstrap log store. Thus, we introduced a new parameter: the ratio between updates to existing keys (including deletes) versus the total number of change records. This parameter models the benefit of returning only the latest version of the value for a given key.

The experiments compared the time (in milliseconds) to do a full bootstrap on a set of test databases for which we varied the above ratio. We tested databases where the ratio varied from around 33% to 99%. The results are presented on Figure 16.

Our experiments show that with maintaining the appropriate index structures on the snapshot store, returning compacted deltas from the snapshot store is very efficient. The break-even point is when about half of the change records contain updates to existing keys. In practice, we actively monitor the above ratio in production and use the results from these experiments to tune our bootstrap services. Since the ratio changes slowly, the configuration is manual. In the future, we plan to make the bootstrap service auto-tunable.

6. EXPERIENCE IN PRODUCTION

Databus has been in production at LinkedIn since its early days. It was originally developed to keep the graph engine in sync with the primary Oracle database. The original architecture [3] is shown in Figure 17. It provided the consistency semantics that we needed and basic support for table-level subscription. The implementation included only relays and acted as a caching layer for the change stream from the database. Relays were constantly monitoring for new changes in the source database and buffering those in memory. If a consumer was lagging behind and its pull request could not be served from the in-memory buffer, the relay transparently created a proxied pull connection to the source database and fetched all changes needed by the consumer.

With growth in traffic, number of consumers and the complexity of use cases, the original implementation started showing some scalability and operability limitations. This resulted in a significant redesign and rewrite of Databus. In this section, we look at how we fared and lessons learned from operating the system in the last two years.

6.1 The Good

6.1.1 Source Isolation

In the original implementation of Databus, when a consumer fell too far behind, it would get proxied through to the source database. Our experience has shown that there are many reasons why clients often fall behind by a lot in unexpected ways. The most common cases happen when clients bootstrap themselves with state from data in offline systems like Hadoop, and then come online and have to catch-up a week's worth of data. Another set of cases arise due to software bugs. There have been cases where "bad" or unprocessable data has been written to the database or the consumer logic had bugs that made it choke on a particular event. Since the Databus framework provides in-order delivery guarantees, it retries some number of times and eventually pauses waiting for manual intervention. A third category of reasons are bursts or spikes of activity on the primary datasets. Downstream consumers which are typically provisioned for consuming the steady flow of events during normal operation, are unable to keep up with bursts of data and start falling behind. As explained in the Oracle fetcher implementation, the further the consumers fall behind, the more expensive the pull queries get. Thus, a problem on the consumer side gets translated to a problem on the source database. When we added the Bootstrap database to the Databus architecture, we took away the capability of the consumer to impact the source database in this way. Now, catch-up queries from consumers that are very far behind are served off of the bootstrap database that is isolated and optimized for this purpose. In this way, we've managed to reduce load on the source databases enormously while being able to keep up with the client demands easily. We routinely see clients seamlessly connecting to the bootstrap service, sometimes on a daily basis but just catching up quietly without raising any alarm bells.

6.1.2 Common Data Format

The original implementation of Databus used hand-written Java classes to represent the table rows, and serialized them using Java serialization. This created two problems. Firstly, every time the table's schema was changed, someone would have to hand-edit the Java class; secondly, because the Java serialization of that object was not backwards compatible with the previous version of the object, all downstream consumers would need to get upgraded to pick up the new class definition. The workaround for this problem was to create a new view every time the schema was changed, essentially creating one view per schema version on the database, and one copy of the event per version. As consumers evolved to picking up the new versions of the classes, the old views could be retired. In practice, consumers rarely had incentive to upgrade unless they needed the extra fields, thus the old views tended to stay around forever. The utilization of the relay buffer would worsen because each event would get serialized multiple times for each view version. In our latest changes, we moved to Avro, got rid of the multiple versions and this gave us an immediate three-fold performance win in terms of read and write load on the source database as well as utilization of the relay buffer.

6.1.3 Rich subscription support

At LinkedIn we see a wide variety of consumers that are themselves partition-aware. For example, our distributed search system has many hundreds of nodes and each node only indexes a fraction of the complete data set. Often, different consumers want different partitioning functions or axes, and it is an organizational challenge to force everyone to agree to the same partitioning model. For example, our search engine uses range-based partitioning, while our relevance engine uses mod-based partitioning. Earlier, all the individual machines would pull the entire stream and filter it client-side by dropping the events that they were not interested in. When we added server-side filtering to Databus, we allowed consumers to specify their filtering function while subscribing to the sources. This has resulted in huge network savings which decreased the bandwidth requirements by 40 times.

6.2 The Bad

We haven't solved all our problems yet. There are a few open challenges that we are working on addressing.

6.2.1 Oracle Fetcher performance

Our experience has shown that several factors can negatively affect the performance of the Oracle fetcher:

1. Databus sources in the form of complex join views are expensive because they have to be evaluated at fetch time.

2. Large BLOBs and CLOBs in rows can incur additional disk seeks.
3. The TxLog table grows over time which affects performance. Without time-based partitioning of the table, truncation of older rows requires maintenance with a period of write unavailability for the primary database.
4. Very high update rate can cause increased load on the SCN update job; this affects the effectiveness of the indexes on the TxLog table.
5. Very high write traffic can also lead to increased read and write contention on the TxLog table.

6.2.2 Seeding the Bootstrap DB

When launching Databus on existing tables, we have to go through the initial process of seeding the Bootstrap DB. This involves generating and loading a consistent snapshot of the source data. This can be a challenge for large data sets. Since stopping the writes to the primary store to seed the Bootstrap database is rarely an option, we either have to procure additional hardware to load a stable backup or devise an efficient restartable algorithm that can read the data out of the primary store in small chunks while guaranteeing that no updates are going to be missed and that all transactions that happen during the seeding process are fully applied at the end. We chose to use the second approach. Our production experience has shown that sources with complex joins and/or large BLOBs can negatively affect seeding performance. In some cases with complex joins, we have used dumps of the source tables and computed the joins offline (e.g. in Hadoop). With such an approach, the main challenge is ensuring that the offline join produces *exactly* the same results as if it was performed by the primary store, because the two table dumps may not be consistent with each other.

7. RELATED WORK

Change Data Capture (CDC) is not a new problem. However the environment in which it is required often determines the design trade-offs that get made in the architecture. At LinkedIn, the expectations of paying customers and the requirement to offer a premium user experience motivated us to solve change capture in a scalable manner while not sacrificing data consistency or reliability. Related work in this area falls into two categories:

- *Full featured CDC systems*: Many CDC systems such as Oracle DataGuard [5] and MySQL replication [4] are restricted to replicating changes between specific source and destination systems. Other products such as Oracle Streams [14] also make the change stream available through user APIs. Systems such as Golden Gate [6] and Tungsten Replicator [7] are capable of replicating from different sources to different destinations. But these are designed for usecases where the number of destinations is reasonably low and where consumers have a high uptime. In cases where consumer uptime cannot be guaranteed, seamless snapshotting and arbitrarily long catch-up queries must be supported. Most CDC systems are not designed for these scenarios.
- *Generic messaging systems* Messaging systems are sometimes used as transport to carry CDC data. There are some key trade-offs here. Messaging systems such as Kafka [9], ActiveMQ [1] and other JMS implementations typically provide publish-subscribe APIs. Publishers are responsible for pushing changes to the messaging system, which is then responsible for guaranteeing the fate of the messages. Thus, the messaging system has to act as a *source-of-truth* system. This leads to extra overhead for durability due to persistence, internal replication, etc. It also introduces potential for data loss in worst-case failure scenarios. In the presence of a reliable data source, this overhead is unnecessary.

8. CONCLUSION AND FUTURE WORK

In this paper, we've introduced Databus, LinkedIn's change data capture pipeline. Databus supports partitioned and non-partitioned transactional sources, very granular subscription capabilities and full re-processing of the entire data set while providing very low latencies and scaling to thousands of consumers with diverse consumption patterns. The interesting challenges we faced were mostly in the areas of:

- Low-level systems design in building a low-latency high-throughput buffer that can scale to arbitrary size while supporting deep filtering on records.
- Building an algorithm that can support consumers catching up from arbitrary points in time while maintaining a bounded amount of persistent state.
- Layering the architecture in a way that is conducive to integration with a variety of data source technologies and amenable to flexible deployment strategies.

Databus will be used as the internal replication technology for Espresso [11], our distributed data platform solution. Databus will also provide external subscribers the capability to listen to the changes happening to the base dataset. We intend to explore some interesting avenues in the future.

- *Relay-Client Protocol*: The current relay-client protocol is poll based. The client polls frequently to fetch new changes. With lots of clients polling frequently, this can lead to unnecessary resource utilization at the relay. We plan to add support for a streaming protocol, so that the client can make a request and just continue to read new changes off the response stream. This will lead to lower latencies as well as lower resource consumption.
- *GoldenGate integration*: Recent releases of Oracle GoldenGate satisfy LinkedIn's requirements for CDC from Oracle. We are working on a Databus adapter that will pull changes from Oracle while avoiding the overhead of triggers. This change will not affect downstream consumers.
- *User defined processing*: The current subscription model allows consumers to pass in pre-defined filters for the changes that they are interested in consuming. We would like to extend that to support running user-defined processing on top of the stream.
- *Change-capture for eventually consistent systems*: Our current implementation requires the source to provide a single transaction log or a set of partitioned transaction logs. Systems like Voldemort [11] do not fit into either category. It would be interesting to extend Databus to support such systems as data sources.

9. ACKNOWLEDGMENTS

First and foremost, we would like to thank our shepherd Pat Helland who provided us with extremely insightful feedback for the final version of this paper. We would also like to thank the following people for their invaluable contributions to Databus: Jean-Luc Vaillant who is the spiritual father of Databus; Mitch Stuart and Yan Pujante who built and hardened the first version of Databus; our Operations partners Agila Devi, Jill Chen, Neil Pinto, Sai Sundar, Ramana Ramakrishnan, Nishant Vyas, Kevin Krawez, Brian Kozumplik and Zachary White; Lin Qiao, Tom Quiggle, Bob Schulman and Siddharth Anand who gave us continuous feedback during the preparation of this paper and last but not the least, all Databus customers at LinkedIn who've provided us with constructive feedback always.

10. REFERENCES

[1] ActiveMQ. http://activemq.apache.org/.
[2] Avro. http://avro.apache.org.
[3] Linkedin infrastructure. QCON'2007, http://bit.ly/FQNqWI.
[4] MySQL Replication. http://dev.mysql.com/doc/refman/5.0/en/replication.html.
[5] Oracle DataGuard. http://www.oracle.com/technetwork/database/features/availability/dataguardoverview-083155.html.
[6] Oracle GoldenGate. http://www.oracle.com/technetwork/middleware/goldengate/overview/index.html.
[7] Tungsten Replicator. http://www.continuent.com/solutions/tungsten-replicator.
[8] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, UK, 1978. Springer-Verlag.
[9] J. Kreps, N. Narkhede, and J. Rao. Kafka: a distributed messaging system for log processing, 2011.
[10] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
[11] LinkedIn Data Infrastructure Team. Data infrastructure at LinkedIn. In *ICDE*, 2012.
[12] R. Ramamurthy, D. J. DeWitt, and Q. Su. A case for fractured mirrors. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 430–441. VLDB Endowment, 2002.
[13] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.
[14] L. Wong, N. S. Arora, L. Gao, T. Hoang, and J. Wu. Oracle streams: a high performance implementation for near real time asynchronous replication, 2009.