Cassandra CDC

Table of Content

Introduction

As Cassandra becomes a more common choice of database for microservices at WePay, we want to be able to transfer data from Cassandra to other databases/services/search engines for analytics, such as BigQuery or ElasticSearch. Currently, there is a mechanism to transfer data to BigQuery via Airflow, however there are a few problems with this batch process:

- **It is expensive**. The offline batch processing is a full load which means it rewrites the entire db
- **It is slow**. As data in Cassandra grows, it eventually won't be feasible to run the batch job hourly or even daily
- **It is not generic**. The operator is specific for BQ. If we want to transfer the data to other sinks, we need to build a new operator per sink

Since Cassandra CDC JIRA, there is a cdc feature added to Cassandra. Whenever the commit log is flushed, it is also copied to `cdc_raw` directory, which then can be processed by a CL reader and converted into change messages for downstream consumption.

The following compares the pros/cons of parsing CDC files with other alternatives:

Solutions	**Pros**	**Cons**
C* Driver Double-Write	* It is fairly simple to reason about and to implement: can be done with a custom client that wraps the existing C* driver and extends the feature to double-write to Kafka	* It at least doubles the the latency of each write * It's challenging to guarantee all or nothing: data is either written to both C* and Kafka, or neither * It is difficult to always make sure application code will use the correct driver * Additional code needs to be written for each new table by the application developer, thus increases burden to their development cycle and is error prone
Trigger	* It is fairly simple to reason about and to implement -- just need to implement the built-in C* trigger functionality	* It is not ideal to bring CDC logic to application code * It does not scale well as the write throughput increases * Additional code needs to be written for each new table by

		the application developer, thus increases burden to their development cycle and is error prone * It is known to be buggy
Parsing CDC CommitLog	* It is decoupled from db write operation and does not add latency on each write * It can be easily turned on/off with no code change required * It does not require any knowledge/work from the application developer	* It requires more code complexity than double-writing or trigger

Requirement

To be able to feed data from Cassandra to Kafka in near real-time so downstream consumers can take actions on changed data.

What Cassandra CDC Service will provide:

- Messages arrive in Kafka with < 5 minutes delay
- Messages contain the full row of data **after** the change
- Messages are in order **at best effort**
- Messages are deduped **at best effort**
- The ability to bootstrap a table
- The ability to handle schema evolution
- The ability to handle messages correctly when we scale C* nodes up/down

What Cassandra CDC Service does not provide:

- Messages contain the full row of data **before** the change
- Messages for Range delete
- Messages for TTL deletion
- Messages for changes to static columns
- Messages for changes to materialized views
- Messages for every single mutation
- Messages for light-weight transaction
- Messages for batch writes

Proposed Architecture

Unlike MySQL, Cassandra mutations contain only modified columns. So in order to get the full row of data, there are two approaches:

(1) **Stateful Processing:** maintain the state of the entire db. As we read each mutation, apply the mutation to the old state to get the new state (i.e. via Kafka Stream, Flink)

(2) **Stateless Processing:** for each mutation received, issue a select query from the local node to get the entire row's data.

Stateless processing means we eliminate the storage and operational overhead of having devops to maintain a separate DB (i.e. RocksDB), the tradeoff here is increasing code complexity and memory pressure. The remaining doc discusses the stateless processing mechanism.

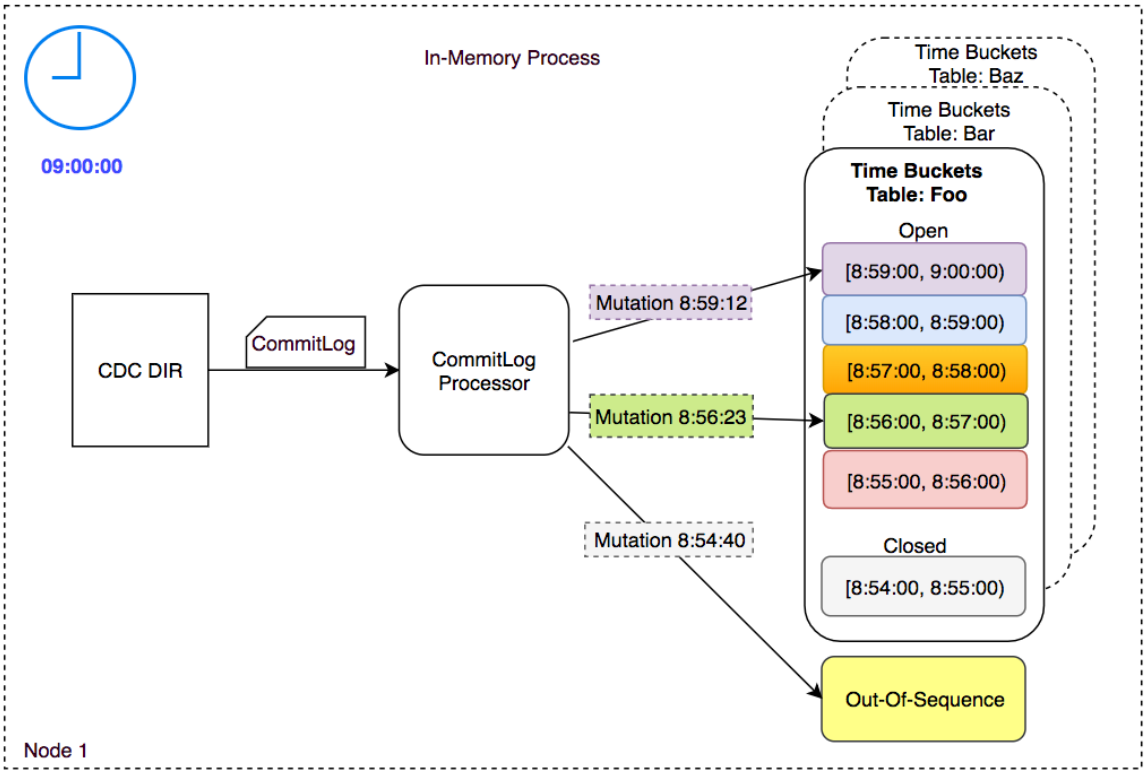
CDC Service Components

The CDC Service contains the following components:

- **CommitLogProcessor -** a task that parses the commit log and extract useful information (primary key, crc, timestamp, operation type). It is also responsible for querying the local node and sending changed messages to Starts as soon as CDC service is started.
- **OutOfSequenceProcessor** - a task that polls `cdc.out_of_sequence` table and runs query against the C* cluster. Starts as soon as CDC service is started
- **SchemaUpdater** - a task that queries the schema periodically and sends updated schema updates to Kafka. Starts as soon as CDC service is started.
- **SSTableProcessor** - a temporary task responsible for reading result set from the select * during bootstrap. There will be one bootstrap reader per table that is bootstrapping. Starts when `cdcservice bootstrap` command is triggered.
- **cdc_processor.yaml** - config file for the cdc service

CommitLogProcessor

Processing Mutation



1. A CommitLog processor lives inside each Cassandra node. It watches the `cdc_raw` directory and look for changes in the active CommitLog. When it detects a new change, it processes it immediately.

2. For RF = 3, we assign one primary node, secondary node, and tertiary node for each token range. For example:
   N1 = {primary: A-J, secondary: K-T, tertiary: U-Z}
   N2 = {primary: K-T, secondary: U-Z, tertiary: A-J}
   N3 = {primary: U-Z, secondary: A-J, tertiary: K-T}
   The priority are arbitrary but consistent. It is used for deduplication such that only data from the primary node will be sent to Kafka.

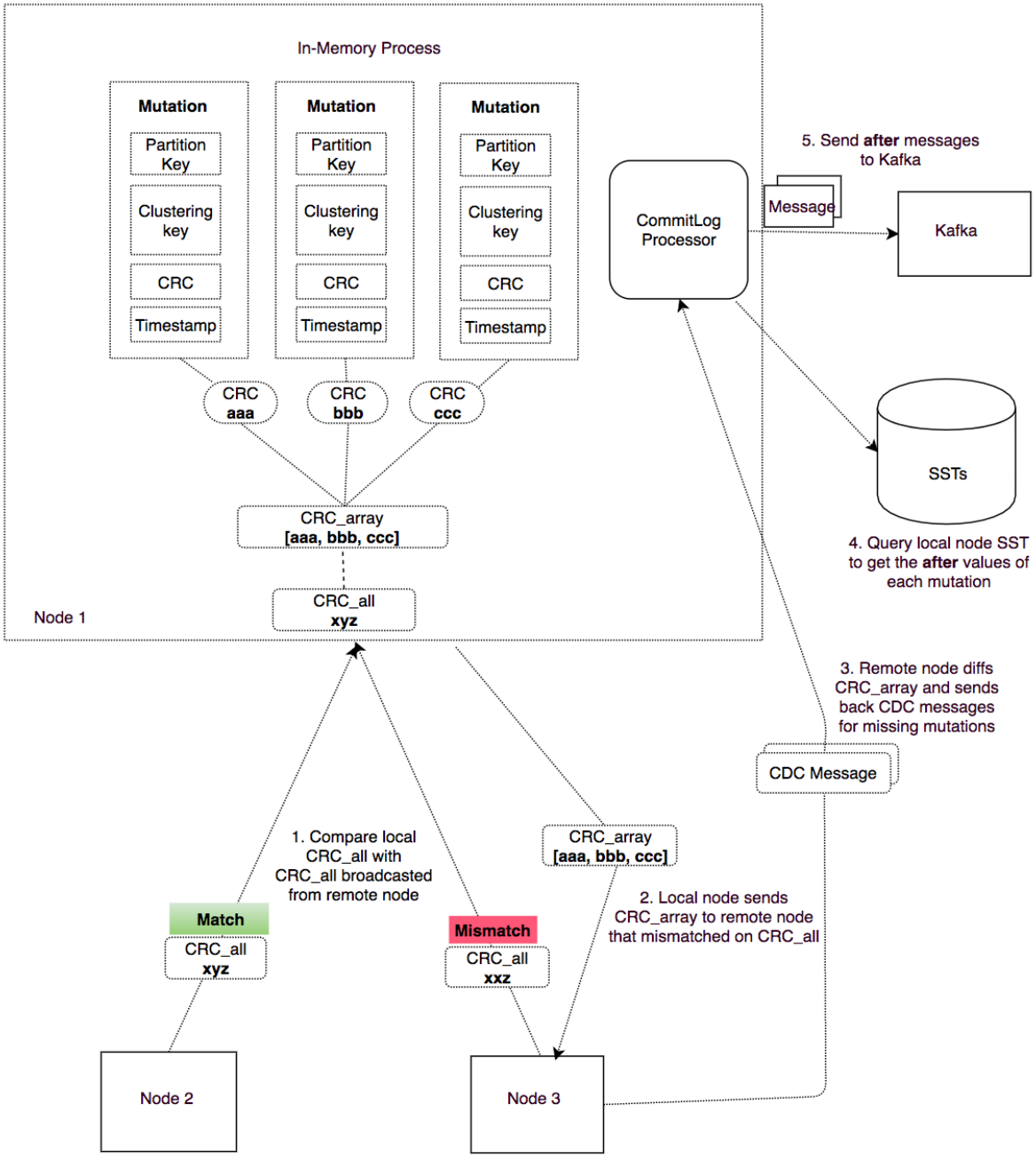
3. When processing a mutation from CommitLog, the CommitLogProcessor will first identify the table and token range the mutation belongs to. Then based on its timestamp, it will sort the mutation into corresponding time bucket in-memory. There are 5 active time bucket for each table, with each bucket representing one minute. So there is a total of 5-minute window that accepts mutations.

4. At the start of each minute, the oldest active bucket is closed and no more mutations can be added to it, instead they must be send to the out-of-sequence table (only partition key, clustering key, and timestamps are stored in the `cdc.out_of_sequence` table, which are sufficient for lookup later).

5. Each time a window is closed, the last closed mutation offset is updated in a separate `<commitlog_name>_cdc.offset` file. In the case the node goes down, when it comes back up it will start processing mutations from the recorded offset.

6. A new active bucket will be created to account for the latest time bucket.

Processing Closed Time Bucket



1. A closed bucket contains all mutations for the specific table and token range, which in theory should be identical to the bucket corresponding to the same table and token range on the other nodes. We process each mutation in the bucket and extract its partition keys, clustering keys, CRC and timestamp and store them in-memory.

2. Take the CRC of each mutation, sort them, and then take a CRC of all CRCs. Communicate this CRC as well as the array of individual CRCs to all nodes.
3. For each primary token range, dedupe the in-memory primary keys. Then query local node for each PK and convert the result set into a change message to be consumed by Kafka broker later.
4. For each non-primary token range:
    a. Compare with CRC communicated from primary node for this token range
    b. If the CRC does not match, compare the CRC array and identify all local CRCs that are missing in the primary node for this token range. Query the local node to get the missing rows, and communicate the missing rows back to primary node.
5. For each primary token range:
    a. Compare with CRC communicated from non-primary nodes for this token range
    b. If the CRC does not match, it means the local node is missing some mutations. Since only the primary node is responsible for sending change messages to Kafka, we want to make sure primary node contains all the changes. The primary node waits for the missing rows from non-primary node, and timeout if it doesn't get any updates.
    c. Once it receives responses from remote nodes, it compares local row to remote row. If they are different, use the latest timestamp to resolve conflicts.
6. For each primary token range, send all change messages to Kafka.

Note: the way the nodes communicate with each other will be writing and polling via a shared cassandra table in version 1 of CDC Service for ease of implementation.

Change Messages

The Cassandra change event is largely modeled after Debezium change event. Each change event contains a kafka topic key and a kafka topic value, and each of which has a schema and a payload. The `schema` section contains the schema of what is in the payload portion.The `payload` section contain the actual values of the change message.

When the processor detects a schema change, it sends a schema change event to a central `serverName` topic. Cassandra commit log does not contain change event for DDLs, so in order to get an updated schema, the schema for each table will be checked on an interval.

On the other hand, data change are directly a transformation of the data read from Cassandra commit log. The data change message are sent to the corresponding topic in the form of `serverName.keyspaceName.tableName`.

Schema Change Key

The key payload should have a single field `keyspace`.

Example:

```
{
	“schema”: {
		“type”: “struct”,
		“name”: “class.path.to.cassandra.processor.SchemaChangeKey”,
		“optional”: false,
		“fields”: [
		{
			“field”: “keyspace”,
			“type”: “string”,
			“optional”: false
		}
		]
	},
	“payload”: {
		“keyspace”: “foo”,
	}
}
```

Schema Change Value

The value contains not only the `keyspace`, but also the `ddl` and the `source` metadata. The `source` metadata includes:

- `version`: version of the cdc service used
- `hostname`: hostname the ddl change is discovered
- `ts_ms`: timestamp of when this mutation is generated
- `snapshot`: boolean representing if this event is part of bootstrap

Example:

```
{
	“schema”: {
		“type”: “struct”,
		“name”: “class.path.to.cassandra.processor.SchemaChangeValue”,
		“optional”: false,
		“fields”: [
		{
			“field”: “keyspace”,
			“type”: “string”,
			“optional”: false
		},
		{
			“field”: “ddl”,
			“type”: “string”,
			“optional”: false
		},
		{
			“field”: “source”,
			“Name”: “class.path.to.cassandra.processor.Source”,
			“type”: “struct”,
```

```
                        “optional”: false
                        “fields”: [
                        {
                                “type”: “string”,
                                “optional”: false,
                                “field”: “version”
                        },
                        {
                                “type”: “string”,
                                “optional”: false,
                                “field”: “hostname”
                        },
                        {
                                “type”: “int64”,
                                “optional”: false,
                                “field”: “ts_ms”
                        },
                        {
                                “type”: “boolean”,
                                “optional”: false,
                                “field”: “snapshot”
                        },
                        ]
                },
                ]
        },
        “payload”: {
                “keyspace”: “foo”,
                “ddl”: “CREATE TABLE abc (id INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
firstname VARCHAR(512), lastname VARCHAR(512);”,
                “source”: {
                        “version”: “1.2.3”,
                        “hostname”: “server”,
                        “ts_ms”: 1453452345,
                        “snapshot” : true
                }
        }
}
```

Data Change Key

The key payload should have a single field which represents the kafka key.

Example:

```
{
        “schema”: {
                “type”: “struct”,
                “name”: “class.path.to.cassandra.processor.Value”,
                “optional”: false,
                “fields”: [
                {
                        “field”: “id”,
                        “type”: “int32”,
                        “optional”: false,
                },
```

```
            ],
        }
        "payload": {
            "id": 1001
        }
}
```

Data Change Value

The value contains an envelope structure with the following fields:

- `op`: type of operation: `c` for create, `u` for update, `d` for delete, `r` for read (in case of snapshot)
- `source`: a structure which describe the source metadata for the event
  - `version`: version of the CDC Service used
  - `hostname`: hostname the commit log is from
  - `file`: name of the commit log file
  - `pos`: position in the commit log file
  - `snapshot`: boolean representing if this event is part of bootstrap
  - `ts_ms`: timestamp of when this mutation is generated
  - `query`: cql query that triggered the mutation
- `ts_ms`: timestamp at which the event is processed by the CommitLog Service
- `after`: if present contains the state of the row after the event occurred. In the case of a deletion, the after value will be emption

Example:

```
{
        "schema": {
            "type": "struct",
            "optional": false,
            "name": "server.keyspace.table.Envelope",
            "version": 1,
            "fields": [
            {
                "field": "op",
                "type": "string",
                "optional": false
            },
            {
                "field": "source",
                "type": "struct",
                "name": "class.to.source",
                "optional": false,
                "fields": [
                {
                    "type": "string",
                    "optional": false,
                    "field": "version"
                },
                {
                    "type": "string",
                    "optional": false,
```

```
                        "field": "hostname"
                },
                {
                        "type": "string",
                        "optional": false,
                        "field": "file"
                },
                {
                        "type": "int32",
                        "optional": false,
                        "field": "pos"
                },
                {
                        "type": "int64",
                        "optional": false,
                        "field": "ts_ms"
                },
                {
                        "type": "boolean",
                        "optional": false,
                        "field": "snapshot"
                },
                ]
        },
        {
                "field": "ts_ms",
                "type": "int64",
                "optional": true
        },
        {
                "field": "after",
                "type": "struct":,
                "name": "server.keyspace.table.Value",
                "optional": true,
                "fields": [
                {
                        "type": "int32",
                        "optional": false,
                        "field": "id"
                },
                {
                        "type": "string",
                        "optional": false,
                        "field": "firstname"
                },
                {
                        "type": "string",
                        "optional": false,
                        "field": "lastname"
                }
},
"payload": {
        "op": "c",
        "source": {
                "version": "1.2.3",
                "hostname": "server",
                "file": "CommitLog-6-1532485837032.log",
```

```
                        "pos": 123,
                        "ts_ms": 1453452345,
                        "snapshot" : false
                },
                "ts_ms": 1292138901489,
                "after": {
                        "id": 1001,
                        "firstname": "John",
                        "lastname": "Bar
                }
        }
}
```

Data Type Conversions

The following table describes how the cdc service maps each Cassandra data types to an Avro type:

Cassandra Data Type	Java Type	Avro Type
ascii	String	string
bigint	java.math.BigInteger	long
blob	ByteBuffer	bytes
boolean	Boolean	boolean
counter	Integer	int
date	Integer	int
decimal	java.math.BigDecimal	double
double	java.lang.Double	double
float	java.lang.Float	float
frozen		bytes
inet	String	string
int	Integer	int
list	List	array
map	Map	map
set	Set	array

smallint	Integer	int
text	String	string
time	Long	long
timestamp	Long	long
timeuuid	UUID	bytes
tinyint	Integer	int
tuple		
uuid	UUID	bytes
varchar	String	string
varint	java.math.BigInteger	long

OutOfSequenceProcessor

OutOfSequenceProcessor will poll mutations from `cdc.out_of_sequence` table in batch to look for work. If the table is not empty, it will:

1. filter only the mutations that are considered primary to the local node
2. dedupe mutations based on primary key
3. for each mutation:
    a. quorum read the to get the full row data
    b. compare max timestamp from the out-of-sequence table against the result set
    c. if max timestamp from the table is older than max timestamp from result set, sends the change message to Kafka
    d. delete the row from the out-of-sequence table

SchemaUpdater

C* schema evolution is handled by a background SchemaUpdater task in each node. We use a config variable `schema_updater_poll_interval_ms`, which defaults to 10 seconds. When the task starts, it fetches and caches the latest avro schema for each table. Then at the start of each interval, it compares the schema generated from the db against the cache. If they are

different, then the processor will update the Schema Registry with the new table schema and refreshes the cache.

With this approach, a Kafka message may be using a stale schema for up to `schema_updater_poll_interval_ms` amount of time.

- If the new schema adds a column, then the message will contain fields that the schema doesn't, in which case the field is temporarily ignored.
- If the new schema removes a column, then the new message will be missing field which is in the schema, in which case the field is temporarily a null value.

SSTableProcessor

SSTableProcessor is used to bootstrap a new table, a separate `bootstrap` Kafka stream is created to buffer mutations until the bootstrapping task is completed. Steps are:

1. During the bootstrapping process, disallow:
    a. new nodes to be added
    b. existing nodes to be removed
    c. any schema change to the table being bootstrapped
2. Quorum write an entry to `cdc.bootstrap` table indicating that bootstrap for the table is in process.
3. Enable cdc for the new table, this will start CommitLog processing for mutations in this table. Note that change messages for these mutations will be buffered in the `bootstrap` Kafka stream instead.
4. Run a `select *` query against the table. For each row in the result set, convert to change message and send to the normal `cdc` stream. If the table is too large to be done in a single query, run the command one token range at a time.
5. Once all results from `select *` are drained, start polling from `bootstrap` stream and transferring them to to `cdc` stream.
6. Once all change messages in the `bootstrap` stream have been drained, update `cdc.bootstrap` table indicating that bootstrap for the table has completed. This will switch back to sending change messages to `cdc` stream.

CDC_Processor.yaml (WIP)

`Commitlog_processor_window_count` – the number of windows to maintain in-memory (defaults to 5)

`commitlog_processor_window_interval_ms` - the per window duration in milliseconds (defaults to 60000)

`schema_updater_poll_interval_ms` - frequency of table schema update in milliseconds (defaults to 10000)

`out_of_sequence_processor_poll_interval_ms` - frequency of checking cdc.out_of_seqeuence table to poll for work (defaults to 10000)

Handling Commissioning/Decommissioning of Node

When a new node is being added to the cluster, CommitLog Processor waits until bootstrap completion, then begins to process the change log. Because the new node will double-write updates until point of completion, waiting for bootstrap completion will reduce the number of duplicates being send to kafka stream.

When an existing node is being removed from the cluster, the node will no longer be handling reads, and writes will be double-written to existing nodes that is reassigned. Immediately stop reading from node at the start of bootstrap that is being decommissioned to reduce double-writes.

Handling Failures

Cassandra CDC Service is a distributed system that captures all changes from different Cassandra nodes. It is definitive that one or more components in the CDC pipeline will fail at some point.

Configuration And Startup Error

The CommitLog Service could fail upon startup if:

- One or more configurations are invalid
- The CommitLogProcessor/SchemaUpdater cannot establish connection to C* cluster
- The CommitLogProcessor/SchemaUpdater cannot establish connection to Kafka broker

Commit logs will not be processed if there is a startup error, it is up to the admin to turn off cdc until the problem is resolved.

Cassandra Node Is Down

The CommitLog Service will attempt to query the local node, and in the case it fails, it will sleep for some time and then retry in a loop. The CommitLog Service will resume as soon as the Cassandra node is back up. Since the CommitLog is not being updated on that node, disk space pressure is not a concern.

The Schema Updater will successfully query the cluster, so as long as number of nodes alive is at least quorum, it should continue to work. In the case where it's less than quorum, it will sleep for some time and then retry in a loop until successful.

Mutation Out-of-Sequence During Parsing

If the CommitLog Processor finds a mutation out of the parse window (> 5 minutes ago), it will write the mutation to `cdc.out_of_sequence` table in Cassandra and move on. The OutOfSequenceProcessor will pick it up and process it separately.

CommitLog Processor Failed During Parsing

If the CommitLogProcessor is unable to parse the content of a mutation. It will write the mutation to a `cdc.processor_error` table in Cassandra and moves on to the next mutation. The fix will be manually done offline and the error rows will be deleted. Since all data from the

mutation will be preserved in the `cdc.processor_error` table, the commit log can be deleted once fully parsed.

// todo: figure out how to address failure and remove rows in error table once fixed

CDC Service Down

If the CDC service failed, then we run into the risk that the CommitLog quickly fill up in the C* instance. If the disk used exceeds `cdc_free_space_in_mb`, it could prevent writes from happening. This is not acceptable since we do not want failure of CDC processing to affect production services. If the `cdc_free_space_in_mb` falls below a certain threshold, the commit log file will be send to GCS and deleted and will be required to be processed separately.

When CommitLog Service comes back online, it will look for the commit log offset it last sent to Kafka and restart from there. Note that this will likely temporarily add a lot of workload to the OutOfSequenceProcessor.

Kafka Becomes Unavailable

Kafka broker could be down from time to time. When this happens, kafka messages will be temporarily send to `cdc.kafka_buffer` table. Then the CommitLog Service will periodically check the liveliness of the Kafka broker. Once it is healthy again, the CommitLog starts to drain the db by sending the messages to Kafka. Once the DB is empty, it will then switch back to directly sending messages to Kafka.

Proposed CLI Interface

The initial version of the CDC Service has a simple CLI interface with the following features:

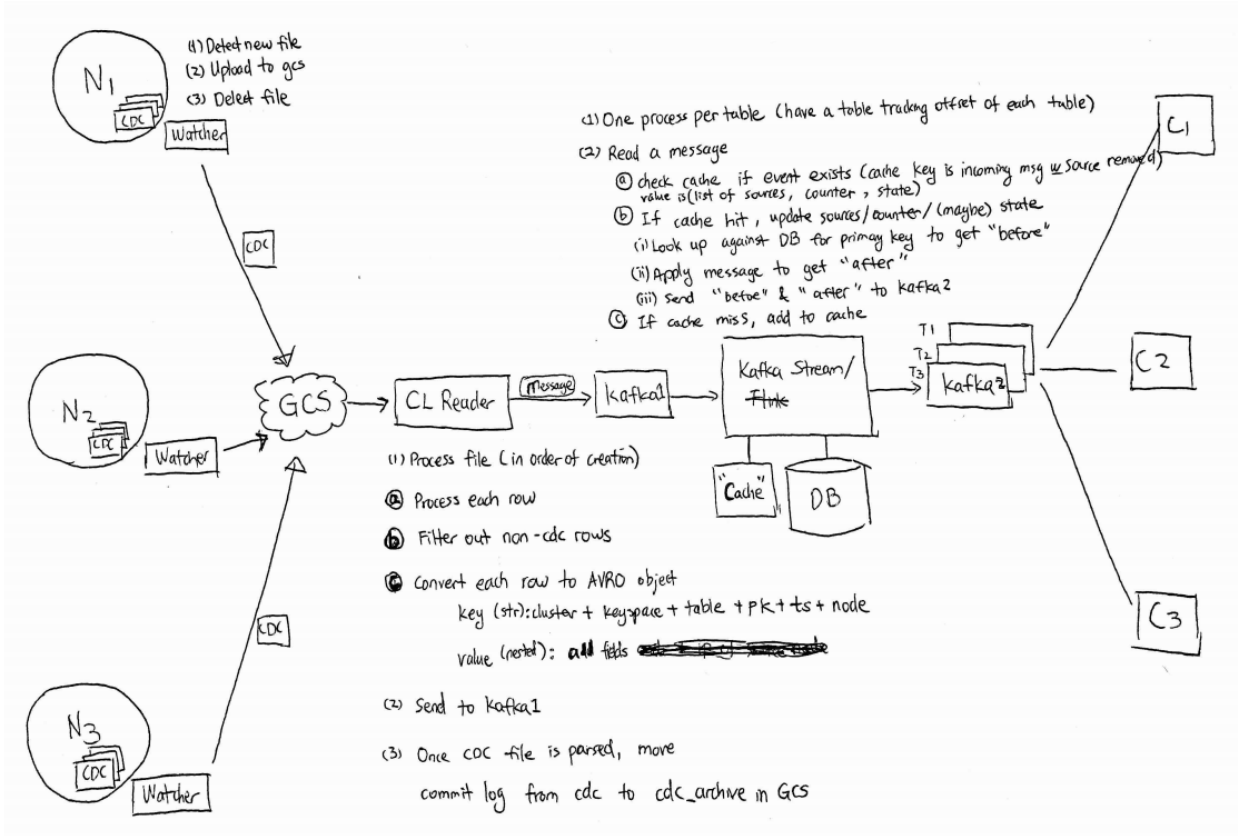
```
// start the CDC service
cdcservice start --conf <PATH_TO_CDC_PROCESSOR.YAML>

// stop the CDC service
cdcservice stop

// bootstrapping a table
cdcservice bootstrap --table TABLE_NAME
```

Additional Notes

Proposed Architecture (old, stateful processing)



CDC Directory Watcher - the cdc directory watcher resides on every Cassandra node and is responsible for detecting new cdc files and sending them to GCS (modify the cdc file name so it includes the address of the node for later use).

GCS - GCS bucket contains a cdc directory and a cdc_archive directory. When files are uploaded, they are in the CDC directory. When they finish processing, they get dumped into the cdc_archieve directory.

CommitLog Reader - the CommitLog Reader is responsible for processing the CDC files. Since commit log contains both CDC data *and* non-CDC data. It will process each mutation, and filter only for CDC-enabled mutations. It will then convert the Cassandra mutation object into a AVRO (or some other format that kafka accepts) object, where they key contains information about node + cluster + keyspace + table + primary key + client timestamp (all fields should have the same timestamp in a given mutation), and the value contains all the parsed field key/value from the mutation. It will send this information to Kafka. And once it's complete it will move the CommitLog from the cdc directory to cdc_archive directory in GCS. If the reader fails at any point, it can always find the file with the earliest creation timestamp, and reprocess it.

Kafka1 - There is one kafka topic only. It is okay to do compaction here, since each message is a unique cdc event from a specific node (assuming it's not possible for a single node to make two consecutive updates on the same key at the same time).

Kafka Stream/Flink Framework - There would be a k/v on disk tracking each table and its offset. Need to know the number of replicas for each keyspace. When a new message arrives, checks against the cache. The cache key is the incoming message (with source removed), and the cache value is the list of sources, counter, and state. If the message is in the cache, update its counter/source/(maybe) state if quorum is reached. Once it reaches quorum, mark state as 'ready'. look up against the RocksDB for the primary key to get the "before" value, then apply the message to the "before" to get the "after", store the "after" to RocksDB, and then send the "before" and "after" to the second Kafka. (We may want one process per table and do it in kubernetes)

Kafka2 - This broker contains only the before and after of valid messages. All consumers of this Cassandra CDC will read from this kafka broker.

Areas of Concern

Using the CDC feature provided by Cassandra, the following issues needs to be addressed:

CDC dir full causing production write to fail

- If size of cdc_raw directory > cdc_free_space_in_mb, then Cassandra will actually stop allowing writes to happen. This is very dangerous. To prevent this:
    - If we process CL in the node, we want to make the processing as quick as possible, and have mechanism in place if cdc disk space runs low.
    - Otherwise, we could always copy the CL to GCS and process asynchronously, and delete the CL immediately from the node.

CDC file contain both CDC data and non-CDC data; non-CDC data should be filtered out

- Each cassandra node has one active CommitLog file at any given time. This CommitLog file is copied to cdc_raw directory when it's flushed. Since every mutation is being written to the CommitLog, non-CDC-enabled logs should be filtered out when processing CDC files.

Missing before/after data: CDC file doesn't contain full row, unlike mysql binlog, it only contains mutation of columns that changed

- Need to get data for the full row **before**/**after** the change
- Getting **before** is not easy since by the time we read the mutation, reading C* would return the state after the change
- Getting **after** is also not easy since by the time we read the mutation, multiple writes could have happened to the same key, reading C* would return the state not immediately after the change
- Only way to get around this, is if we have another db that apply these mutations to generate the table. This gives us ability to call the db immediately before/after the mutation to get the full row. We could use a C* cluster for this, or a different db, either way it will require significant additional storage space.

All CDC files will contain RF copies of each mutation and need to be deduped

- Internally, cassandra uses Last Write Win to solve conflict. This means a quorum write that "failed" due to timeout will eventually get replicated to all nodes.
- This means for CDC, all mutations should be processed regardless of quorum and we persist those with the latest timestamp just like Cassandra
- Alternative option is to use a separate cluster with RF = 1 just to handle CDC. However there are some issues:
    - If node is down temporarily, the cdc would be temporarily halted
    - If node is down/destroyed permanently, could lose data due to hinted handoff

Ordering: mutations are not necessarily written in order

- Mutations could arrive to nodes in different order ([1, 2] in node one, [2, 1] in node two)
- Even where data are in order, since they are replicated to each node, it's possible to see row flipping back and forth as one replica might lag another (this may be okay since we know C* is eventual consistency) (timestamp could help)
- To handle ordering, we can use client timestamp. We'd have to compare the highest timestamp that has already been processed against the timestamp of the newly arrived record.

Range delete: each range delete generates a single mutation in commit log record

- In the case where if it's a range delete, need to convert the range delete to one record per row so they can be applied by downstream client correctly (not possible because we can't do it in a timely fashion, cassandra compaction could already remove the delete at that point)

Anti-Entropy repair are directly repaired in SSTable and are not reflected in commit log

- Both hinted handoff and read repair will be fixed via commit log and reflected via CDC processing. On the other hand, anti-entropy repair are directly repaired in SSTable (via building a merkle tree). So commit log will not know about these changes.

TTL will get deleted during compaction but will not be reflected in commit log

- When using TTL, a record is inserted with a future local deletion time
- Commit log will not get an update on each row when it is deleted
- Therefore CDC won't be able to reflect the change from the expiration of the TTL

Schema information is not included in mutations

- Unlike mysqldb which have DDL statement (schema change) recorded in their commitlog, Cassandra commit log contain only value changes
- Need to be able to track schema update in C* and notify Schema Registry for Kafka
- Each mutation itself does not contain schema information, only the data type of the changed columns

Topology Change: handle CDC when a C* is added or removed from the cluster

- Cassandra is a distributed db, nodes can change their status while the cluster is still alive. There needs to be a way to detect node status change and react to these changes when applicable. These events include:
    - Node shutdown/boot
    - Node decommission/commission
    - Keyspace/table created/deleted

Need to be able to deduplicate data on cell level (not row-level) in SSTable when bootstrapping a new table

- Since commit log gets purged, we would have to use SSTable to handle snapshotting

- Since SSTable are grouped by C* tables, we can just filter and snapshot the SSTables for the specific table that we care about
- Since we only care about the latest value, we can just parse each node and only update the value if the timestamp is the newest. This means we may need to keep track of the highest client timestamp for each Cell, and then ignore the rows that have a smaller timestamp

Cassandra's built-in snapshotting is run locally; sufficient space needs to be available else it is risky for large tables

- C* snapshotting will first save the backup locally, which means there must be enough disk space to copy the backup, and then the file can be copied into a remote location and deleted locally.

There's a latency for data from the active commit log to be archived in the cdc_raw directory, which may cause data availability issue if nothing is written to the db for a long time.

This was solved in CASSANDRA-12148: updates not immediately available until CommitLog is flushed. The fix has been merged but not yet released.
In the past CommitLog are available in cdc_raw only when flushed. With this change, consumer will actively parse hard-links of CommitLog to get the latest data.

- C* will store offset of the highest seen CDC mutation in a separate index file per commit log in cdc_raw.
- Clients will tail this index file, delta their local last parsed offset, and parse the corresponding commit log using last parsed offset as min
- C* flags index file with an offset and DONE when the file is flushed so client knows when to clean up

There is basically a polling mechanism that got added which is suppose to also pull from CommitLog actively to get the data even if the CommitLog is not flushed.

Until then, one way to handle this is to populate dummy data to commit log when the cluster is fairly inactive.

Features Comparison Provided by Various CDC Solutions

The following evaluates the implementation complexity of various CDC solutions provided what requirements have *already* been solved or eliminated in each approach.

Requirements	Level of Importance (very important, somewhat important, Not important)	C* Driver Double-Write	Trigger	Parsing CDC CommitLog
CDC is in real time (*duh*)	**Very important**	✓	✓	✓
CDC should not have negative impact on write path	**Very important**	✗	✗	✓
CDC should be fault-tolerant	**Very important**	✗	✓	✓
CDC contains schema	**Very important**	✗	✗	✗
CDC includes “after” data from C*	**Very important**	✗	✗	✗
CDC includes “before” data from C*	**Somewhat important**	✗	✗	✗
CDC does not require additional work from application developers	**Somewhat important**	✗	✗	✓
CDC does not contain duplicates	**Somewhat important**	✓	✓	✗
CDC are ordered	**Somewhat important**	✗	✗	✗
CDC should include range delete	**Not important**	✗	✗	✗

CDC should include TTL	**Not important**	✗	✗	✗

As seen in the chart, the only substantial gain we get from C* driver double-writing or trigger is not needing to dedupe, which is a loose requirement and is something we can implement with best effort. What parsing commit log provides us are (1) asynchronous processing so it is not in the write path (2) fault tolerance so data is eventually consistent between db and cdc (3) isolation of concern so application develops do not need cdc-related domain knowledge. So we ultimately decided the commit log parsing approach is the best option over trigger and driver double-write.

FAQ

How do I enable CDC in Cassandra?

CDC could be configured on the table-level:

```
CREATE TABLE foo (a int, b text, PRIMARY KEY(a)) WITH cdc=true;
```

It could also be configured on the node level, in `cassandra.yaml`:

`cdc_enabled` : enable cdc node-wide

Other cdc configurations:

`Cdc_raw_directory:` this is the directory that the cdc log will show up in

`Cdc_free_space_in_mb:` total space to use for cdc log on disk

`Cdc_free_space_check_interval_ms:` interval to check disk space once we hit the limit

What is in a commit log segment?

- It's a single commit log file on disk
- Pre-allocated 32 MB segment file, renamed + reused after flush
- It is flushed when (1) MemTable reaches max size (2) CommitLog reaches max size (3) Manually run `nodetool flush`
- It has 2 durability modes for write:
    - **Periodic** (commitlog_sync_period_in_ms): the sync period represents how often the commit log is synchronized to disk. This means it may lose data if all replica crash.
    - **Batch** (commitlog_batch_window_in_ms): the batch window represents how often the sync happens, but unlike periodic, it guarantees sync before acknowledging writes (blocking write)
- CommitLogSegment **1:N** Sections **1:N** Mutations **1:N** PartitionUpdates **1:N** Rows **1:N** Columns **1:N** Cells *(1:N refers to mapping)*
    - A **CommitLogSegment** starts with a descriptor: version, segment id, compression dict, crc checksum, and has one or more sections
    - A (sync) **section** is just a contiguous part of a CommitLogSegment and has one or more mutations
    - A **mutation** contains one (most cases) or more (batch) partition updates
    - A **partition update** contains table metadata (and tell whether it's regular/index/view/virtual) and header (contains partition key and shared static columns and partition deletion info) and one or more rows
    - A **row** has a bunch of metadata (isStatic, hasTs, hasTTL, hasDeletion, isShadowable, complexDeletion, hasAllColumn, etc.) and zero or more columns (if it's zero, it's a range delete)
    - A **column** has a bunch of metadata (name, length, kind, etc.) and one or more cells

- A **cell** has a bunch of metadata (hasValue, isDeleted, isExpiring, useRowTs, useRowTTL) and value

Are commitlog "reader" logically separated from "writer" (so that we could use it to read mutations)?

- Yes, you can create a **CommitLogReader** and pass a custom implementation of a **commitLogReadHandler** to it.
- A **commitLogReadHandler** is an interface with `handleMutation` method which can be implemented to handle deserialized mutation object parsed from a CommitLogSegment file
- A **commitLogReplayer** is an implementation of a commitlogReadHander used by C*
    - Upon construct, the replayer will create a map of tableId and intervals of data persisted on sstable
    - Each commit log segment is then iterated in ascending order
    - Each segment is read from lowest replay offset among the replay positions read from the SSTable metadata
    - Log entry is replayed for table if position of the log entry is > ReplayPosition for table in most recent sstable metadata
    - When log replay is done, all memtables are force flushed to disk and all commitlog segments recycled
- Upon replay completion, it updates the cdc directory if `sawCDCMutation` flag is true
    - (1) a hard-link is added if not exist
    - (2) cdc index file (offset of the last data synced to disk) is added

When does commitlog replay happen?

- Currently, the replaying is called during restart of the CassandraDaemon for a C* node

What is in an sstable?

An sstable contains data from a single C* table

- **data.db** (data organized by partition. Sorted in token order.)
- **index.db** (index from partition key to position in data.db)
- **summary.db** (every 128th entry)
- **filter.db** (bloom filter of partition key in sstable)
- **compressionInfo.db** (metadata about offsets & lengths of compression chunks)
- **statistics.db** (include timestamp, tombstone, cluster key, compact, repair, compression, TTL, etc.)
- **digest_crc.db** (crc-32 digest)
- **toc.db** (table of content containing plaintext list of components for SSTable)

Why does each mutation not contain the full row?

Cassandra write path doesn't do read-before-write. Each write is a Mutation which contains only the columns that are modified, rather than the full row. The mutations are

first serialized and write into an append-only commit log. This designs allows high performance on write even ask the database scales up.

There is a timestamp for every single cell (which is made up of timestamp, key, value) and this timestamp is being used for conflict resolution. What is this timestamp source and how does conflict resolution works?

- This is a client timestamp identical for all replicas
- Timestamp is generated by the client. You cannot set this timestamp from client code, but you can view the upsert time with the CQL *writetime()* function.
- Format is long, representing utc in microseconds.
- Latest timestamp wins (LWW) when there's a conflict
- If timestamp is the same, the greater ***byte value*** wins

Does each mutation contains schema information? Or does DDL change get reflected in the Commitlog as a mutation (in either case we can do schema evolution)?

Schema change are not recorded as commit log. Mutation itself doesn't contain schema information; but mutation columncontains CQL3Type. We'd want to either periodically refresh the schema and rely on that, or use each row's column's CQL3Type to regenerate schema (confirm this).

Do all read repairs get written to CommitLog first?

There are 3 types of repairs, and the answer depends on which repair is used.

- **Hinted Handoff:** coordinators will retry by reapplying mutations and therefore will be written to the CL first
- **Read Repair:** this creates repair mutations and therefore will be written to CL first (confirm this)
- **Anti-Entropy Repair:** this directly repairs the SSTable on each node, not through CL

Why does a CDC log contain both CDC and non-CDC mutations?

The implementation of C* guarantees atomicity surrounding a mutation (either all of a Mutation or none of a Mutation will be applied). Mutations are grouped at the keyspace level. So in order to allow cdc flagging on a per-table basis. Need to have consumer be schema-aware so it can filter out non-cdc mutations on consumption.

Why didn't C* handle dedupe instead rather than leaving the burden to the downstream consumers?

According to CASSANDRA-8844, it turns out that C* did dog-fooding its db for other features (hints, batchlog) in the past, however the lesson learned from that was using C* as backing store for internal processing tend to amplify failure scenario and makes headaches a lot bigger when things go wrong. So they intentionally avoided this approach.

How would CDC handle columns with TTL?

Each cell has a TTL and a localDeletionTime. The TTL represents how many seconds left until the cell expires. The localDeletionTime represents the exact time that the cell is expiring. It is unclear how TTL is updated. If it is only updated during compaction then it may not be reflected in cdc log.

Is it possible to issue a read from the same node that the CDC file is in without dealing with network hop? This could allow us to get the "after" data without having to deal with network hops and delays (note this would not be the immediate "after" data from applying the mutation, but the latest "after")

1. VIA Cassandra Driver (in-process)

This is possible. For each driver, if we set the consistency level to LOCAL_ONE and the LB policy to `WhiteListPolicy` with only local node listed, then the driver will read local node in 99% of the case.

2. VIA QueryProcessor (out-of-process)

QueryProcessor exposes a static method `executeInternal`, which queries the local node only. But if we call it out-of-process, it would only read data once they are flushed to SSTable since the MemTable is empty in this process.

- Two options to get data not flushed to SSTable:
  (1) Read the CommitLog
  - Pro: Do not need to manage flush. MemTable flush and CommitLog flush is managed by the Cassandra node instance.
  - Con: CommitLog contains mutations only, so need to first apply all the relevant mutation in chronological order before. Need to handle the logic of combining CommitLog + SSTable data. Each read is potentially much slower since CommitLog is not sorted.

  (2) Keep track of our own MemTable
  - Pro: No extra work required when calling *executeInternal*
  - Con: Need to build our own MemTables. Need to synchronize flush (each time CommitLog is flushed, we need to explicitly flush (delete) our MemTable copy as well)

Overall, option 2 is unnecessarily complicated. Option 1 doesn't guarantee 100% but is significantly simplified.

How could we keep track of our own MemTable?

We can potentially leverage the mechanics of the **CommitLogReplayer**. The CommitLogReplayer is used when a crash occurs before its content is flushed to disk. Then the commit log is used to replay data and to rebuild the MemTable.
What we essentially do is treat each commitlog as a micro-batch. Each time a commitlog is copied over to the cdc log directory:

1. Replay the entire commit log to build the memtable
2. Iterate through the entire commit log again to get the **after** values of each mutation

3. Archive the commit log and remove it from cdc file once processed

Is it possible to query multiple primary key? This could potentially allow us to issue less queries during local read.

Can use “IN” to query more than 1 pk
I.e. select * from ks.cf where pk in (‘pk1’, pk2’);

What are the possible exception we get from the cassandra driver and how do we handle it?

NoHostAvailableException - driver can’t connect to one or more of the nodes (contact points). Data is not written.
UnavailableException - request reaches coordinator but not enough replica alive to achieve requested consistency level. Data is not written.
ReadTimeoutException - coordinator believes there are enough replicas alive to process query with requested CL, but for some reason one or more are too slow to answer and timed out. By default, driver auto retry so this error is rare.
WriteTimeoutException - no way to know whether mutation has been applied or not on the non-answering replica.
QueryValidationException - query string written is wrong and can’t be parsed.

See https://www.datastax.com/dev/blog/cassandra-error-handling-done-right

Is it possible that a write failed in a C* cluster, but succeeded on an individual node?

Yes. Suppose we have RF = N, and we write with consistency level of QUORUM. The write could fail, but it’s still possible that the data was successfully written to M nodes where M < N. When a quorum write fails with “WriteTimeoutException”, we don’t know if the write succeeded or failed. It’s a “partial write” and will never rollback.

When reading, quorum read will resolve the conflict with **LWW** (Last Write Win). If RF = 3, and quorum = 2, the driver will return the one with newest timestamp and do a read repair on the older node. The other older node will be repaired eventually (i.e. anti-entropy, read repair)

In summary, Cassandra use quorite write but doesn’t use true quorum read. The conflict resolution in read repair is based on timestamp. It means that a failed write can eventually overwrite existing data. We can’t confidently say data isn’t written even when a write failed. If it wrote to at least 1 node, it will be eventually successful. If it was written to none of the nodes, it will never be written (unless retried).

It is recommend to retry WriteTimeoutException, except when write operation was not idempotent (i.e. counter updates append/prepend updates on list, since they represent delta not updated value)

See https://www.datastax.com/dev/blog/how-cassandra-deals-with-replica-failure and http://mighty-titan.blogspot.com/2012/06/understanding-cassandras-consistency.html

Can read repair be turned off?

On consistency level > 1 node, if read digest don't match up, read repair is done in a blocking fashion before returning result.

But if the consistency is met and the digest result from those nodes are the same. But due to read_repair_chance repair is happening on the remaining node(s), it is done in the background after result is returned to client.

How does Datastax handle dedupe during replication?

Separate cdc mutations into distinct table. For each table, separate them into distinct token ranges. Mutations are collected in time buckets.

Assign (arbitrarily but deterministically) primary, secondary, tertiary … etc. replicas for each token range.

As mutations are proceeded from CDC file, they are stored in time-slice buckets of 1 minute worth of data.

Also keeping a stack of 5x slices in memory at a time, which means can handle up to 5 minutes out of order.

Any data proceeded > 5 minutes is put into out-of-sequence table, and treated as exceptional data which will need to be replicated from all replica nodes.

They compare the time slices using the crc of each mutation. They then sort the crcs and take a crc of all the mutation crcs and compare that timeslice crc.

Only the primary token range segment file is transmitted.

They use gossip to monitor which nodes are active and not. If a node fails, 'secondary' becomes active for that nodes 'primary' token range.

We are looking to leverage CASSANDRA-12148 for tailing CommitLog in real-time? But it won't be available until 4.0. What is the state of Cassandra 4.0 and are there any concern if we become an early adopter for it?

It is not recommended to be be an early adopter for a new release. Usually better to wait for 1 year for it to stabilize otherwise we get a lot of headache having to deal with bugs early.

What are the different options of bootstrapping a new table via SST?

1. Copyto command
    - Doesn't work well for large tables (30-50M rows okay, beyond that bad)
2. Sstabledump
    - Snapshot of sst (do not change DDL during this time)
    - Per-node basis so requires deduplication
3. "Select *" with token iteration
    - Can be done in parallel for multiple tables
    - Example: `SELECT * FROM keyspace.table WHERE TOKEN(primary_key) > value1 AND TOKEN(primary_key) < value2;`

The recommended way is 3.

There may be multiple active commit log segment at the same time. Why?

A commitlog can be active but full, in which case it will wait until it's flushed. Otherwise it will stay as active but will not be the one that's being allocated to.

What is SEDA (Staged Event-Driven Architecture) and what's its significance to Cassandra?

A design for highly concurrent internet services that support massive concurrency demands and simplify the construction of *well-conditioned* services.

- Combines aspect of threads and event-based programming models to manage concurrency, I/O, scheduling, and resource management.
- Applications are constructed as a network of stages, each with an associated incoming event queue.
- Each stage represents a building block that may be individually conditioned to load by threshold or filtering its event queue.
- Key property of well-conditioned service is *graceful degradation* -- as load exceed capacity, service maintains high throughput with linear response-time penalty that impacts all clients equally

Cassandra uses the SEDA model to implement its message service, which handles both local & remote processing.

Links

CDC-related:
Uber's CDC Slideshow
DataStax Advanced Replication (CDC Implementation)
SmartCat CDC Implementation
Will deduplication solutions work for cassandra
Streaming Data out of Monolith -- Building a highly available CDC
Using Golden Gate Capture for CDC

Tools:
Kafka-Connect-Cassandra (Landoop) and doc
SSTableTools
Aegisthus - A Bulk Data Pipeline out of Cassandra (for sstable)
Cassandra Source Connector (Landoop) (batch / incremental)
GitHub - Netflix/sstable-adaptor (spark)
Tablesnap - Uses inotify to monitor Cassandra SSTables and upload them to S3
Cassandra-Exporter
Cassandra Cluster Manager - script to easily create/destroy Cassandra cluster on Localhost

Other:
SEDA paper (stage event-driven architecture)
How is data read
How is data written
Cassandra Wiki: architecture internal
Cassandra Wiki: Hinted Handoff
Cassandra Wiki: anti-entropy architecture
Cassandra Wiki: SSTable architecture
Cassandra Wiki: CommitLog architecture
Cassandra Wiki: Distributed Delete
Cassandra SSTable Storage Format
Reaper -- automatic repair for apache cassandra
http://thelastpickle.com/blog/
Lucene based secondary indexes for Cassandra
Cassandra 2.0 Diagnostic Checklist

Meeting Notes

Notes from Carlos Rolo (Pythian)

- No materialized views
- No secondary indices & user defined type
- Single point of entry for schema
- Noisy neighbor
    - Split clusters
- Batch write during migration
    - If failed, write should be idempotent
    - Use prepared statement
- CDC (backup feature)
    - Remove logs from cdc manually
- Problem with limits
    - Cassandra doesn't like select *
- TTL typically better than deletes

Notes from John (Pythian)

- C* is good for high traffic only
- Schema change is non-trivial
- Pluggable compaction can be used to write our own compaction (TimeWindowCompactionStrategy)
- Use orchestration tool for any kind of C* operation (i.e. ansible)
- Lightweight transaction - expensive (paxos)
- Stay away from batches (does not guarantee)
- Log batch (different from batch)
    - will keep trying it (by logging into db)
    - Is not as performant as issuing two separate command
- Counters use paxos too, doesn't guarantee its correct (very expensive/slow)
- Cassandra stress is not a good stress testing tool -- it's on a per table basis
- Read/Write path on adding a node: read served on old, write served on both. Once transferred, read switched to be served on new
- Never write null data, they will automatically be treated as tombstones