Cumulus : Combining Periodic and Eventual Consistency (in the cloud)

Oliver Kennedy
Cornell University
okennedy@cs.cornell.edu

Yanif Ahmad
Cornell University
yanif@cs.cornell.edu

Christoph Koch
Cornell University
ckoch@cs.cornell.edu

ABSTRACT

1. INTRODUCTION

Recent years have seen the beginning of a paradigm shift in data management research from incrementally improving decades-old database technology to questioning established architectures and creating fundamentally different, more lightweight systems that are often domain-specific (e.g., [**?**, **?**]). Part of the impetus for this change was given by potential users such as scientists and the builders of large-scale Web sites and services such as Google, Amazon, and Ebay, who have a need for data management systems but have found current databases not to scale to their needs. One can observe a trend to disregard database contributions in these communities [**?**, **?**], and to build lightweight systems based on robust technologies mostly pioneered by the operating systems and distributed systems communities, such as large scale file systems, key-value stores, and map-reduce [**?**, **?**]. Further impetus has resulted from the current need to develop data management technology for multicore and cloud computing.

There is a recent tendency among pundits outside the database community to contest the need for powerful queries, and to think of key-value stores – with only the power to look up data by keys – as (much more efficient) database query engines. However, expressive query languages such as SQL do not cease to have important applications and a substantial user base. Alas, we do not know how to process SQL queries on updateable data using a system as lightweight as a key-value store.

This paper contributes a fundamental and versatile building block for enabling new, more lightweight and nimble data processing systems based on SQL aggregation queries. Cumulus is a thin execution layer that lives above generic key-value stores and can efficiently process a range of incremental view queries. We believe that our contribution constitutes an important step towards achieving the apparent contradiction in terms of executing complex aggregation queries on updateable data using little more than a key-value store.

At the heart of our approach is M3, a trigger-based vector processing language. M3 can either be written directly or produced using DBToaster[], an aggressive recursive incremental view maintenence compiler. In most traditional database query processors, the basic building blocks of queries are large-grained operators such as joins. Conversely, a large class of SQL aggregation queries can be compiled down to very simple message passing programs that incrementally maintain materialized views of the queries.

These message passing programs keep a hierarchy of map data structures (which may be served out of a key-value store) up to date. They can share computation in the case that multiple aggregation queries (e.g., a data cube [**?**]) need to be maintained. Most importantly, though, these message passing programs can be massively parallelized to the degree that the updating of each single result aggregate value can be done in constant time on normal off-the-shelf computers[1].

Cumulus parallelizes computation using a novel mixed-consistency runtime. This runtime is based around an eventual consistency processing model that provides low-latency approximate results. However, unlike most eventual consistency systems, Cumulus' runtime periodically generates consistent snapshots as a side effect of its normal operation. Consequently, the same runtime and data may be used simultaneously for applications requiring low-latency as well as applications requiring strong consistency.

Our main technical contributions are as follows:

- We present M3, a massively parallizable language for message passing programs that can be used to incrementally maintain SQL aggregation queries.
- We present Cumulus, a lightweight realtime exact online aggregation system that integrates seamlessly with generic key-value stores.
- We describe three different approaches to ensuring the consistency of the data: Centralized Target Aggregation, Centralized Source Aggregation, and Eventually and Periodically Consistent Source Aggregation. We deomonstrate how the last of these techniques simultaneously produces low-latency eventually consistent results as well as periodic consistent snapshots.
- We show evidence for the scalability of our approach by examining the performance of Cumulus on examples

[1]Note that since there are usually many more aggregate values to maintain than there are processors, this does not mean that each update is processed in constant time.

drawn from the TPC-H benchmark[]

2. THE M3 UPDATE CALCULUS

Cumulus is built around M3, a trigger-based language closely related to Relational Calculus. The basic datatype in M3 is the map, a mapping from vectors of keys to numeric values, or entries. An M3 program consists of a set of triggers of the form

$$\texttt{on <R>}(\vec{x}\vec{y})\ \{\ s_1;\ \ldots;\ s_k\ \}$$

where `<R>` is an event parametrized by a set of *event variables* $\vec{x}\vec{y}$. For example, for an M3 program that incrementally computes the results of a SQL query, an event would be a user inserting a row into a relation. Each event triggers the evaluation of a list of statements s_i each of which updates zero or more entries in a single map.

$$m[\vec{x}\vec{z}]\ \texttt{+=}\ t(\vec{x}\vec{y}\vec{z})$$

Statements are comprised of a reference to a map entry on the left-hand side, and a right-hand side consisting of a term: a monomial algebraic expression over constants, variables, and map entry references. The simplified version of M3 used in this paper is presented in Figure 1

Two types of variables appear in M3 statements: event variables as defined above, and *loop variables*. Loop variables draw their domains from the right-hand side maps that they parametrize. For example, consider the trigger:

```
ON R(A) { m1[B] += m2[A,B] }
```

When Event `R(A)` occurs, every value `m1[B]` will be incremented by the corresponding `m2[A,B]`[2]. In effect, this is a multidimensional vector operation where the *slice* of map `m2` with first parameter equal to `A` is extracted and used to increment `m1`. Because the right hand side must be a monomial, only values of `B` for which `m2[A,B]` is nonzero can affect `m1`. These values form the *domain* of loop variable `B`. Note that the domains of two loop variables may be linked if they appear as parameters to the same map.

M3 provides two consistency guarantees for programs:

- Triggers execute atomically with respect to each other.
- Statements within a trigger execute in order.

These guarantees are equivalent to serializable consistency, where each event constitutes a transaction.

2.1 Restricted M3

For the purposes of this paper, we restrict ourselves to a limited, fully incremental version of M3.

- *All loop variables appear on both sides of a statement.* The natural implication of a loop variable appearing only on the right-hand side is as a sum aggregate over the variable. In the restricted M3, this aggregate can be computed incrementally by maintaining an additional map with one less key.
- *All loop variables appear as a parameter to exactly one right-hand side map.* A loop variable that appears in two right-hand side maps is effectively a join between the maps. In the restricted M3, joins are computed incrementally using event variables.
- *Terms do not contain conditions.* Unrestricted M3 supports conditioned non-incremental aggregates.

Distributing unrestricted M3 is beyond the scope of this paper. As shown in [], this restricted form of M3 remains powerful enough to express any SQL count or sum aggregate query without inequalities.

EXAMPLE 2.1. *Consider the following query over a TPC-H-like schema.*

```
SELECT   c.nation, SUM(o.discount * l.price)
FROM     CUSTOMERS c, ORDERS o, LINEITEMS l
WHERE    c.cid = o.cid AND o.oid = l.oid
GROUP BY c.nation
```

This query can be implemented in streaming form as the following M3 program

```
ON Insert_LINEITEMS(OID, PRICE) DO {
 q[NATION] += m1[OID, NATION] * PRICE;
 m2[CID]   += m3[CID, OID] * PRICE;
 m4[OID]   += PRICE }
ON Insert_ORDERS(CID,OID,DISCOUNT) DO {
 q[NATION]      += DISCOUNT * m4[OID] * m5[CID,NATION];
 m1[OID,NATION] += DISCOUNT * m5[CID,NATION];
 m2[CID]        += DISCOUNT * m4[OID];
 m3[CID,OID]    += DISCOUNT }
ON Insert_CUSTOMERS(CID,NATION) DO {
 q[NATION]      += m2[CID];
 m1[OID,NATION] += m3[CID,OID];
 m5[CID,NATION] += 1}
```

with deletion triggers that are identical except for a multiplier of -1. Here, the map `q` represents the query result, while other maps represent intermediate subqueries. For example, `m2` represents

```
SELECT   o.cid, SUM(o.discount * l.price)
FROM     ORDERS o, LINEITEMS l
WHERE    o.oid = l.oid
GROUP BY o.cid
```

To see how loop variables are not required for an incremental join, note the three lines that update the result map `q`. Each line incrementally updates the full join result by joining only with the tuple being inserted.

3. CENTRALIZED EVALUATION

M3's use of maps and vector-based operations makes it extremely amenable to distribution. In particular key-value stores are a natural instantiation of the map datatype. Consequently Cumulus is built over a cluster of nodes, each running a shared-nothing key-value store. These key-value stores support multi-dimensional keys and iteration over partial-keys; in effect secondary indices.

The nodes interact with the outside world via one or more coordinator nodes. Event sources such as ticker feeds, sensors, static files, or even relational databases feed the coordinator(s) events. A coordinator then appends the event to a persistent, local log before tagging it with a timestamp and dispatching it for processing into the cluster. Data is

[2]Because all operations in M3 are increments, default values are required for each map entry. In the simplified version of M3 that we describe here, the default value is always 0.

$$
\begin{array}{rcl}
PROGRAM & := & TRIGGER; TRIGGER; \ldots \\
TRIGGER & := & \textbf{ON } \langle EVENT \rangle \ (evt\_var_1, evt\_var_2, \ldots) \\
 & & \textbf{DO } \{STMT, STMT, \ldots\} \\
STMT & := & map_{target}[VAL_1, \ldots] \mathrel{+}= EXPR \\
EXPR & := & map_k[VAL_a, \ldots] \mid VAL \mid \\
 & & EXPR * EXPR \mid \\
 & & \textbf{IF } EXPR \, CMP \, EXPR \\
 & & \textbf{THEN } EXPR \textbf{ ELSE 0} \\
VAL & := & evt\_var_n \mid loop\_var_m \mid constant \\
CMP & := & = \mid < \mid <=
\end{array}
$$

Figure 1: A restricted subset of M3

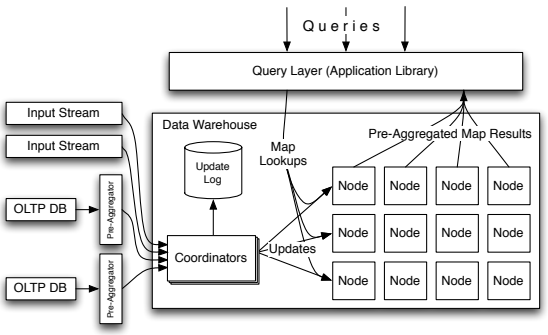

Figure 2: The Cumulus Architecture

horizontally partitioned across nodes by splitting the key-space into multiple disjoint regions. Each node stores one or more map-segments.

External applications access the map nodes directly. Using a partitioning scheme, or *layout* obtained from the coordinator, the application is able to fetch desired map values directly from the node or nodes storing the values. If appropriate, the node may do some preprocessing on the map values (eg: computing partial sum/count aggregates) before sending the response. This design is illustrated in Figure **??**.

The layout of the data across the cluster suggests two natural schemes for distributing update processing: co-locating map-update processing with the map being updated - target aggregation, or co-locating data required for a map-update with the processing node - source aggregation. As we will show, processing distribution is also intricately connected to the primary challenge of distributing M3: ensuring serializably consistent processing of data in M3 programs. We assume that each node is fully aware of how map segments are distributed across the cluster. This assumption and the details of data distribution are discussed further in Section 5.

3.1 Target Aggregation

In target aggregation, data appearing on the right-hand side of a computation is aggregated at the node that will store the corresponding left-hand-side. For example, consider the following M3 statement with maps m1 and m2,

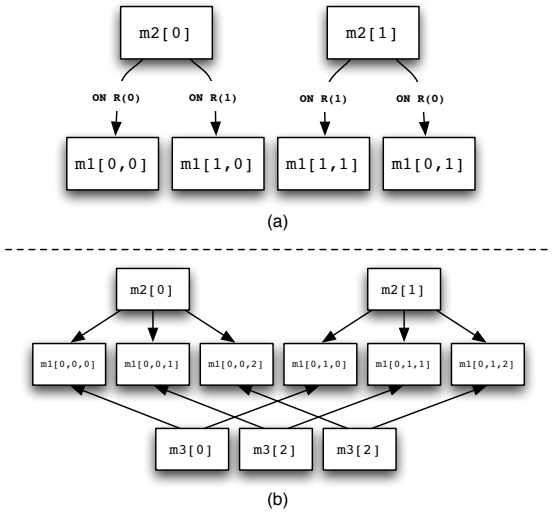

Figure 3: Messages sent during evaluation of a single M3 statement in target aggregation, assuming a uniform partitioning scheme. (a) The statement `on R(A) : m1[A,B] += m2[B]` (b) The statement `on R(A) : m1[A,B,C] += m2[B]*m3[C]`

each partitioned only on attribute B.

```
on R(A) : m1[A,B] += m2[B]
```

When event R(A) occurs, each node storing a segment of m2 will send each value in its segment to the node storing the corresponding segment of m1. If the partition boundaries of m2 and m1 are identical, each m2 node will send its entire segment to exactly one m1 node. This process is illustrated in Figure 2a. Note that If the same node stores corresponding partitions of m2 and m1, no actual data transmission occurs.

Even if m1 is also partitioned over A, each event R(A) updates only one map segment per B value; A node storing a segment of m2 is still only required to send the segment to one m1 node. However, there are situations where multicast is required. Consider the following statement:

```
on R(A) : m1[A,B,C] += m2[B]*m3[C]
```

Let m1 have 10 partitions in a grid pattern, with 2 partitions in the B dimension and 5 in C, and the same partitioning scheme applied to m2 and m3 (with 2 and 5 partitions, respectively). In this instance, when an event R(A) occurs, each m2 node sends its segment to each of the corresponding 5 m1 nodes, while each m3 node will send its segment to each of the corresponding 2 m1 nodes. This process is illustrated in Figure 2b.

If a right-hand side map contains an event variable, only a subset, or slice of the map is sent. For example:

```
on R(A) : m1[B] += m2[A,B]
```

In this instance, only the slice `m2'[B] = m2[A,B]` is sent.

3.2 Centralized Consistency

```
1: for all Event Variable V in Statement S do
2:   partitions[V] = 1;
3:   for all Map M ∈ S with V as a key do
4:     {Run once for each occurrence of V in S as a key}
5:     C = # of partitions of M on the V axis.
6:     partitions[V] = compute_lcm(partitions[V], C)
7:   end for
8: end for
9: for P ∈ ℕ^|partitions| with 0 ≤ P_V < partitions[V] do
10:  Directive[P] = {}
11:  for all Map M ∈ right_hand_side(S) do
12:    for all Event Variable V in M do
13:      C = # of partitions of M on the V axis.
14:      slice[V] = C · P / partitions[V]
15:    end for
16:    {Note that the slice may have multiple owners}
17:    Dir[P]+ = {read M from owner(M, slice)}
18:  end for
19:  M = target_map(S)
20:  for all Event Variable V in M do
21:    C = # of partitions of M on the V axis.
22:    slice[V] = C · P / partitions[V]
23:  end for
24:  Dir[P]+ = {write M to owner(M, slice)}
25: end for
26: return Dir
```

Figure 4: The directive table construction algorithm.

The simplest way to provide M3's serializable execution guarantees is to route all events through a centralized coordinator. Though centralizing creates a scalability bottleneck, the central coordinator's task is straightforward event serialization and dissemination. The task of dissemination is further simplified by building a distribution tree over the nodes themselves.

As a query is loaded, each node precomputes a table of directives from the M3 program and the data layout. This table indicates for each class of event (where class is defined by the partitioning scheme), which of the node's map segments will be modified (ie, those that appear on the left-hand side of a triggered statement), and which segments will be read from (right-hand side). The construction algorithm is shown in Figure 3.

The coordinator first assigns a version number to each event. Then, when a node receives an event from the distribution tree, it consults the directive table for the appropriate read and write operations to perform.

Write operations are deferred until all prior writes on the map have been completed, and all slices appearing on the right hand side of the update statement have been received.

Similarly, read operations block until all write operations on the map pending at the time of the event are complete. Upon completion of the last prior write, the source node gathers the required slices, and sends them to the corresponding destination nodes. Note that the even empty slices must be sent; the destination node needs to hear from all source nodes in order to proceed with its write.

3.3 Source Aggregation

Instead of processing updates at the node where the result will be stored, precompilation allows Cumulus to ensure that all data necessary to process a statement is co-located. In this scheme, each map segment is stilly processed at a specific node. Rather than storing the results locally, a replica of the map segment is created at each node that reads from the segment.

Consistency is maintained in the same way as in Target Aggregation. When an event occurs, each node identifies which local maps might be updated as a consequence. As before, writes are deferred until all input maps have been updated. Also, as before, writes are always sent across the wire, even if no updates actually occur.

Without loss of generality, when discussing source aggregation, we assume that each node processes updates for exactly one map.

4. DECENTRALIZED EVALUATION

Both target and source aggregation attempt to enforce serializable consistency. This introduce delays into update processing, and also requires the use of "empty" messages. Worse still, the centralized coordinator creates a single point of failure and a scalability bottleneck.

The M3 language affords Cumulus a critical benefit: map data is changed only by offset. Thus, the messaging overhead of an undo operation is equivalent to that of a write. Cumulus exploits the low undo overhead in order to achieve a processing model that simultaneously generates both low-latency eventual consistency outputs and periodic consistent snapshots.

4.1 Eventual Consistency

In the eventual consistency approach, there are multiple coordinators, each with a numeric identifier. Coordinators maintain a loosely synchronized clock between themselves, and use it to define epoch boundaries. Synchronized epoch transitions are not required. However, tighter inter-coordinator epoch transitions result in a more efficient system. This is discussed further below.

By decentralizing, each coordinator is freed to do additional work to minimize network usage. Each coordinator maintains a dispatch table, similar to the directive table used by the nodes under centralized consistency. The table maps partitions of each event's parameter space to the set of nodes required to process the partition. When an event occurs, only these nodes are notified.

Each event is sent to a coordinator, where it is assigned a version identifier consisting of the three-tuple {`VersionID, CoordinatorID, EpochID`}. Here, the version identifier is a unique identifier, monotonically increasing at the coordinator within a given epoch. This three-tuple is enough to create a total ordering over all version identifiers by sorting by epoch, coordinator, and then version. An ordering over map updates can be created by extending the three-tuple with the ordering of the M3 statements.

Nodes follow the Source Aggregation approach, but instead of deferring writes, send them as soon as possible. For improved efficiency, this includes a small buffer period. Writes are recorded in a short-term log, along with the event and statement that produced them. The value of the delta is computed immediately based on the source maps' current values, excluding deltas tagged with later versions. The resultant delta is sent immediately.

When a node receives a map delta, it identifies all earlier

updates in its history. It then determines which of these updates are affected by the incoming delta, computes the new delta, and sends the difference between the deltas. Pseudocode for this process is presented in Figure 4.

The values stored in a map may not be correct at any given moment. For example, consider the partial M3 program:

```
ON R(A,B): m1[] += m2[A]
           m2[A] += m3[B]
```

This fragment of the program is deployed with no map segmentation on a 2 node system. Node 1 stores m2 and processes updates to m1, while Node 2 stores m3 and processes updates to m2.

In this scenario, two subsequent R events with identical A parameters are dispatched by the same coordinator nearly simultaneously. Both nodes process the necessary updates to m1 and m2 respectively. However, while node 1 is processing the second event, it does not have the value of m2 updated by the first event. The result is a value of m1 that is incorrect. However, when the updated value of m2 is received, Node 1 can overwrite the corresponding version of m1.

This process maintains an eventually consistent version of all maps. The value of the maps may not be consistent at any given moment, but the error is limited to values that have not fully propagated through the system. Thus, if the system quiesces, maps will return to consistency.

4.2 Data Dependencies

Event E_2 is dependent on event E_1 if E_2 has a later timestamp, and reads reads from a map entry that E_1 writes to[3]. Concretely, E_2 is dependent on E_1 if a non-zero entry appears in the intersection between a slice appearing on the right-hand side of a statement issued by E_2, and a left-hand slice from E_1.

An unfulfilled dependency occurs at a node N if an event E_2 depends on event E_1, but E_1's deltas do not arrive at node N until after N has sent out the delta it produces for E_2. Unfulfilled dependencies are handled by sending out a corrective delta, as described above. However, the corrective delta may be the source of another unfulfilled dependency at the target node if the target has already processed an event with a timestamp after E_2's. We refer to such an event as a cascade.

We can examine the effects of a cascade by considering the dataflow hypergraph of an M3 program, such as the one in Figure 5. In a dataflow graph, each node represents one map and each hyperedge represents a statement. Hyperedges are labeled with the triggering event, and flow out of the source maps of a statement and into the target map. Cascades are bounded by the maximum path length in the dataflow graph.

In [], we show how to produce an M3 program for computing non-recursive SQL queries. Such M3 programs always have acyclic dataflow graphs. However, it is still possible to produce M3 programs with cycles. If a cycle occurs in the graph, As long as newer events continually arriv at nodes directly ahead of the cascade, it will generate an unbounded number of deltas. To prevent this, Cumulus buffers and aggregates deltas for a short period of time. While this does

[3]Note that since M3 statements manipulate entire maps, dependencies at the entry granularity are data-dependent

ON Event $R(\vec{X})$ with version V
1: **for all** Statement S_i Triggered By $R(\vec{X})$ **do**
2: $target$ = target_map(S_i)
3: δ = compute(S_i, $\vec{X}$, V)
4: **if** $\delta \neq \emptyset$ **then**
5: send_push($target$, $\vec{X}$, δ, $V \circ i$)
6: **end if**
7: $history[V \circ i] = \{S_i, \vec{X}, \delta\}$
8: **end for**

ON Push($target$, $\vec{X}$, δ, V)
1: **for all** Key $\vec{E} \in \delta$ **do**
2: sorted_insert($target[\vec{E}]$, V, δ) {Map entries are stored as a list of deltas, sorted by version}
3: **end for**
4: **for all** $\{S, \vec{X}', \delta'\} \in history[V']$ with $V' > V$ **do**
5: **if** S read from $target$ **then**
6: δ_{new} = compute(S, $\vec{X}'$, V') - δ
7: **if** $\delta_{new} \neq \emptyset$ **then**
8: send_push(target_map(S), $\vec{X}'$, δ_{new}, V')
9: **end if**
10: **end if**
11: **end for**

ON compute(S, $\vec{X}$, V)
1: $slice = \{[] \rightarrow 1\}$
2: {We assume there is at least one term. Terms may be numerics or map accesses. In the former case, the pseudocode treats it as $\{[] \rightarrow \#\}$}
3: **for all** Term $T_i[\vec{Y}] \in$ terms(S) **do**
4: $term\_slice[\vec{Y}] = \sum_{\vec{Y} \bowtie \vec{X}; V' \leq V} T_i[\vec{Y}][V']$
5: {Where $\bowtie$ is equality over shared keys}
6: $slice = slice \times term\_slice$
7: {This is the relational cross-product: $\{[A] \rightarrow 1, [B] \rightarrow 2\} \times \{[C] \rightarrow 2\} = \{[A, C] \rightarrow 2, [B, C] \rightarrow 4\}$}
8: **end for**
9: δ = project($slice$, keys(target_map(S)))
10: **return** filter(delete $[*] \rightarrow 0$, δ)

ON Cleanup(V)
1: **for all** Map M **do**
2: **for all** Entry $E \in M$ **do**
3: $E'[V'] = E[V']$ where $V' > V$
4: $E'[V] = \sum_{V' \leq V} E[V']$
5: replace_entry(E', M)
6: **end for**
7: **end for**
8: $history$ = filter(delete $history[V']$ if $V' < V$, $history$)

Figure 5: The Simplified Eventual Consistency Update Algorithm.

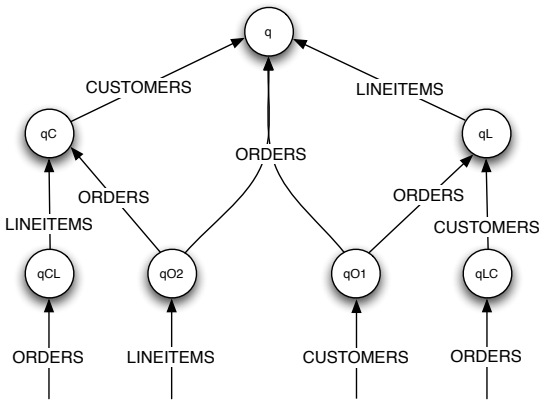

Figure 6: Dataflow graph for the M3 program representing a streaming TPC-H-style query ${}_{C.cid}G_{SUM(L.price)}(C \ldots \bowtie L \ldots \bowtie O \ldots)$

not stop the cascade, it limits the number of messages that a cascade generates.

Out of order event processing allows Cumulus to employ decentralized coordinators. Coordinators are effectively assigned a priority order. Nodes ensure consistency by consistently evaluating events in the order of their issuing coordinators, regardless of the order in which the events arrive.

This means that lower priority coordinators are much more likely to have unfulfilled dependencies with higher priority coordinators, and thus trigger a cascade. Worse, the dataflow graphs produced by compiling SQL to M3 are symmetric; there is no assignment of events to coordinators that maps lower priority coordinators to maps further down the dataflow graph.

Cumulus removes this favoritism by staggering each coordinator's epoch transitions. Each epoch is divided up into a number of periods equal to the number of coordinators. Each period, the next higher priority coordinator transitions into the next epoch. This is easily achieved by a token-passing system. Only the coordinator with the token can switch epochs. It waits for the duration of one period, switches, and passes the token to the next coordinator.

4.3 Garbage Collection

The eventual consistency algorithm uses two datastructures that at first glance appear to be both unbounded and continually growing: The update history, and map values being representated as a list of deltas. In both cases, the datastructure is unbounded because a value is stored for each version identifier that affects it.

However, the tail of both datastructures is only needed until updates with earlier version numbers have finished propagating. The update history is no longer relevant after this point. Similarly, once version V has finished propagating, prior map entries with earlier versions are used only in the aggregate form.

$$\sum_{V' \leq V} E[V']$$

Thus, it is necessary for the system to be able to discover when all updates have completed propagating through the system. Cumulus performs this discovery process periodically, once per epoch by default.

Garbage collection occurs in two phases: coordinator synchronization, and node synchronization. Each coordinator maintains a record of all nodes it sends events to since the last garbage collection. When garbage collecting, each coordinator sends a commit message to all nodes on the list. It then continues processing as normal.

When a node receives a commit message, it responds, confirming that it has received, processed, and inserted into its history all prior events from that coordinator. Once a coordinator receives confirmation from all nodes it sent a commit to, it broadcasts an epoch complete message to all other coordinators. Once all coordinators have sent an epoch complete, the coordinators collectively send each node an epoch complete message.

Once a node receives an epoch complete message, it will never again receive an event for a prior epoch. Now, the nodes must ensure that no further map deltas need to be propagated. The layout fully determines both where a node sends deltas, as well as from where it receives deltas. This dataflow graph can be used to limit the propagation of deltas. If the graph is a DAG, delta propagation may be bounded bottom-up; each node sends a map checkpoint message along all of its out edges once it has received an epoch complete message, and once it has received a map checkpoint from all of its in edges.

If the dataflow graph has cycles, one node in the cycle is designated as the cycle leader. When this node receives an epoch complete message, it sends out a partial map checkpoint to the next node in the cycle. This partial checkpoint is forwarded to subsequent nodes in the cycle, and once the message returns to the cycle leader, it is free to send a full checkpoint message.

The overhead of doing garbage collection is not without benefit. In addition to limiting the amount of historical data nodes are required to store, aggregating all deltas prior to a particular version creates a version of the map segment that is consistent with other map segments with the same version across the entire cluster.

5. PARTITIONING

6. EXPERIMENTS

Buffer size vs latency / max throughput. Intelligent buffering (ie, buffering for greater periods as we go down the dataflow dag)

Eventual+Periodic Consistency. Subquery computation