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
koch@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., [9, 6]).

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 [4, 8], 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 [3, 2]. 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 continue to act as an important intermediary between users and more complex data-processing tasks. 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 wide range of incremental view queries. The query results are stored in the key-value store itself, as ordinary data.

Cumulus is built around a novel hybrid consistency execution framework. Like most cloud data management systems, Cumulus achieves low latencies by retreating to the realm of eventual consistency. However, unlike most eventual consistency systems, Cumulus' runtime periodically generates consistent snapshots as a side effect of its normal operation. The result is a system that can be employed side-by-side by both applications that require low latency and applications that require strong consistency.

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 by hand or translated from SQL with DBToaster[1], 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 [5]) 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].

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 managing the consistency of query results: Centralized Target Aggregation, Centralized Source Aggregation, and Hybrid Consistency Source Aggregation. We demonstrate how the last of these techniques simultaneously pro-

[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.

duces 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 drawn from the TPC-H benchmark[10]

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 lookups parametrized by variables. A simplified version of M3 that will be used in this paper is presented in Figure 1

Two types of variables appear in M3 statements: event variables $\vec{x}, \vec{y}$ as defined above, and *loop variables* $\vec{z}$. 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 restricted M3, this aggregate can be computed incrementally by maintaining an additional map with one fewer 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 restricted M3, joins are computed incrementally using event variables. Note that this restriction results in each loop variable's domain being defined by exactly one map.
- *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 [7], 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, l.part
         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, l.part
```

This query can be implemented in streaming form as the following M3 program[3]*.*

```
ON +LINEITEMS(OID,PRICE,PART) DO {
 q[PART,NATION] += m1[OID,NATION] * PRICE;
 m2[CID,PART]   += m3[CID,OID] * PRICE;
 m4[OID,PART]   += PRICE }
ON +ORDERS(CID,OID,DISCOUNT) DO {
 q[PART,NATION] += DISCOUNT * m4[OID,PART] *
                   m5[CID,NATION];
 m1[OID,NATION] += DISCOUNT * m5[CID,NATION];
 m2[CID,PART]   += DISCOUNT * m4[OID,PART];
 m3[CID,OID]    += DISCOUNT }
ON +CUSTOMERS(CID,NATION):
 q[PART,NATION] += m2[CID,PART];
 m1[OID,NATION] += m3[CID,OID];
 m5[CID,NATION] += 1 }
```

Here, the map `q` represents the query result, while other maps represent intermediate subqueries. For example, `m2` represents

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

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.

[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.

[3]Only insertion triggers are shown. Deletion triggers are identical except for a multiplier of -1 applied to each right-hand side.

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

Figure 1: A restricted subset of M3

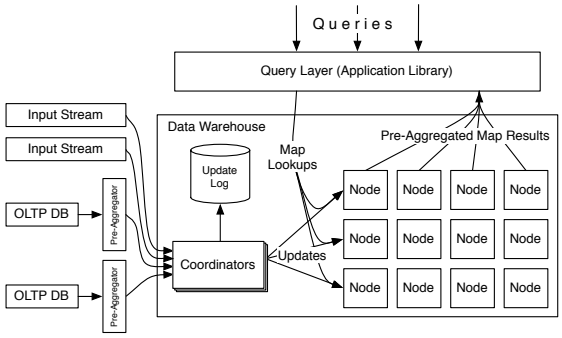

Figure 2: The Cumulus Architecture

3. CENTRALIZED CONSISTENCY

As all statements in M3 are effectively vector operations over maps, the language is extremely amenable to distribution. Cumulus distributes both data and processing over a cluster of *nodes*, each using BerkleyDB(TODO: Cite??) as a local shared-nothing key-value store for map entries[4].

All data in Cumulus is stored as part of a map - either as the value of a map entry, or as one of the entry's keys. A map is horizontally partitioned into one or more *map segments* along any or all of its keys, and each segment is stored on one or more nodes in the cluster.

Nodes interact with the outside world via one or more *coordinators*. Event sources (eg., ticker feeds, sensors, static files, relational databases, or even other M3 programs) send events to the coordinator(s). The coordinator optionally appends the event to a redo log, tags it with a timestamp, and dispatches it to the cluster of nodes for processing. The overall design of Cumulus is illustrated in Figure 2.

External applications access map nodes directly, using a mapping of map segments to nodes, or *layout* obtained from the coordinator. With the layout, the application fetches desired map values directly from nodes storing the values. Cumulus provides library methods for doing so, described in Figure 3. The API allows the client to indicate that a sum, min, or max aggregate is required, so that nodes can

[4]Though Cumulus uses the secondary indexing and iteration capabilities of BerkleyDB, both features can be built into or atop any key-value store.

READ_SLICE(map, key_1, key_2, ...)
Read the current version of a slice of a map; Null keys are treated as wildcards. All matching nonzero entries are returned.

READ_SUM(map, key_1, key_2, ...)
READ_MIN(map, key_1, key_2, ...)
READ_MAX(map, key_1, key_2, ...)
As READ_SLICE but preaggregate values at the node if possible. Null keys are treated as wildcards. The sum, min, max, etc... of nonzero matching entries is returned.

READ_STABLE(map, agg_type, key_1, key_2, ...)
As READ_SLICE, but guarantees a self-consistent result when using eventual consistency (See Section 4.1). Returns the most recent consistent snapshot of all matching nonzero entries with optional pre-aggregation.

Figure 3: Cumulus' Query API

pre-aggregate matching values locally to reduce messaging overhead.

3.1 Distributing M3

We consider two approaches to distributing data processing and data storage across nodes in the cluster in order to minimize network overheads.

1. Co-locate the processing of an M3 statement with the node storing the left-hand side map - that is, the map being updated. We refer to this as *target aggregation*.

2. Co-locate the right-hand side maps of an M3 statement on the same node that processes the statement. We refer to this as *source aggregation*.

As we will show, these questions are also intricately linked to the primary challenge of distributing M3: ensuring serializably consistent execution.

Throughout the following sections, we will assume that each node is fully aware of the data layout. The generation and dynamic management of layouts is beyond the scope of this paper.

3.2 Target Aggregation

In target aggregation, M3 statements are evaluated at the node that will store the results. Data appearing on the right-hand side of the statement is sent to this node for delta computation.

EXAMPLE 3.1. *Consider the following M3 statement with maps* `m1` *and* `m5`*, each partitioned only on attribute N.*

```
ON +ORDERS(C,O,D) DO { m1[O,N] += D * m5[C,N] }
```

When event `+ORDERS` *occurs, each node storing a segment of* `m5` *will send each value in its segment to the node storing the corresponding segment of* `m1`*. If the partition boundaries of* `m1` *and* `m5` *are identical, each* `m5` *node will send its entire segment to exactly one* `m1` *node. This process is illustrated in Figure 4a. Note that If the same node stores each pair of corresponding* `m1` *and* `m5` *partitions, no actual data transmission occurs.*

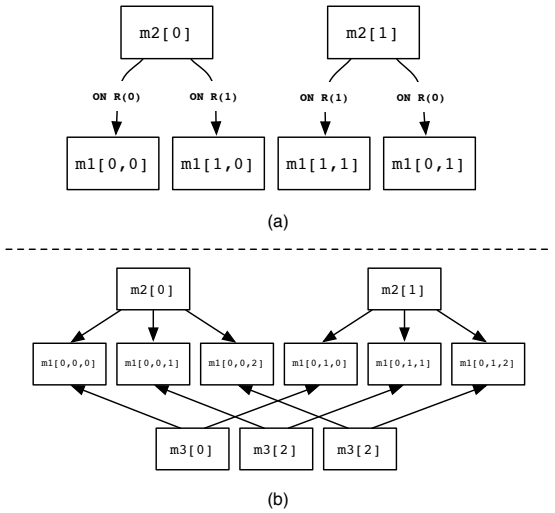

Figure 4: Messages sent during evaluation of a single M3 statement in target aggregation, assuming a uniform partitioning scheme. (a) The statement `on R(A) : m1[A,B] += m2[B]` (b) The statement `on R(A) : m1[A,B,C] += m2[B]*m3[C]`

Even if `m5` is also partitioned over `O`, each `+ORDERS` event updates only one map segment per `B` value; A node storing a segment of `m5` is still only required to send the segment to one `m1` node. However, there are situations where multicast is required.

EXAMPLE 3.2. *Consider the following statement:*

```
ON +ORDERS(C,O,D) DO { q[P,N] += D * m4[O,P]
                                  * m5[C,N] }
```

Let `q` have 10 partitions in a grid pattern, with the domain of `P` split in half and `N` split into 5 parts. Let `m4` and `m5` use the same partitioning scheme (with 2 and 5 partitions, respectively). In this instance, when an `ORDERS` event occurs, each `m4` node sends its segment to each of the corresponding 5 `q` nodes, while each `m5` node will send its segment to each of the corresponding 2 `q` nodes. This process is illustrated in Figure 4b.

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

```
ON +ORDERS(C,O,D) DO { m1[O,N] += D * m5[C,N] }
```

In this instance, only the slice `m5'[N] = m5[C,N]` is sent.

3.3 One Coordinator

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 5.

1: **for all** Event Variable V in Statement S **do**
2: $partitions[V] = 1$;
3: **for all** Map $M \in S$ 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: $partitions[V] = \text{compute\_lcm}(partitions[V], C)$
7: **end for**
8: **end for**
9: **for** $P \in \mathbb{N}^{|partitions|}$ with $0 \leq P_V < partitions[V]$ **do**
10: $Directive[P] = \{\}$
11: **for all** Map $M \in \text{right\_hand\_side}(S)$ **do**
12: **for all** Event Variable V in M **do**
13: $C = \#$ of partitions of M on the V axis.
14: $slice[V] = C \cdot \frac{P}{partitions[V]}$
15: **end for**
16: {Note that the slice may have multiple owners}
17: $Dir[P] += \{\textbf{read } M \textbf{ from } \text{owner}(M, slice)\}$
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: $slice[V] = C \cdot \frac{P}{partitions[V]}$
23: **end for**
24: $Dir[P] += \{\textbf{write } M \textbf{ to } \text{owner}(M, slice)\}$
25: **end for**
26: **return** Dir

Figure 5: The directive table construction algorithm.

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.

3.4 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.

When discussing source aggregation, we assume without loss of generality that each node processes updates for exactly one map, and refer to maps and nodes interchangeably. Where the distinction is relevant, maps are indicated with `fixed width font`.

4. DECENTRALIZED CONSISTENCY

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 6.

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

ON Event $R(\vec{X})$ with version V

for all Statement S_i Triggered By $R(\vec{X})$ **do**
  $target$ = target_map(S_i)
  δ = compute(S_i, $\vec{X}$, V)
  if $\delta \neq \emptyset$ **then**
    send_push($target$, $\vec{X}$, δ, $V \circ i$)
  end if
  $history[V \circ i] = \{S_i, \vec{X}, \delta\}$
end for

ON Push($target$, $\vec{X}$, δ, V)

for all Key $\vec{E} \in \delta$ **do**
  sorted_insert($target[\vec{E}]$, V, δ) {Map entries are stored as a list of deltas, sorted by version}
end for
for all $\{S, \vec{X}', \delta'\} \in history[V']$ with $V' > V$ **do**
  if S read from $target$ **then**
    δ_{new} = compute(S, $\vec{X}'$, V') - δ
    if $\delta_{new} \neq \emptyset$ **then**
      send_push(target_map(S), $\vec{X}'$, δ_{new}, V')
    end if
  end if
end for

ON compute(S, $\vec{X}$, V)

$slice = \{[] \rightarrow 1\}$
{We assume there is at least one term. Terms may be numerics (treated as $\{[] \rightarrow \#\}$) or map accesses.}
for all Term $T_i[\vec{Y}] \in$ terms(S) **do**
  {$\times$ and $\bowtie$ are the cross product and natural join}
  $term\_slice[\vec{Y}] = \sum_{\vec{Y} \bowtie \vec{X}; V' \leq V} T_i[\vec{Y}][V']$
  $slice = slice \times term\_slice$
end for
δ = project($slice$, keys(target_map(S)))
return filter(delete $[*] \rightarrow 0$, δ)

Figure 6: The Eventual Consistency Update Algorithm (simplified).

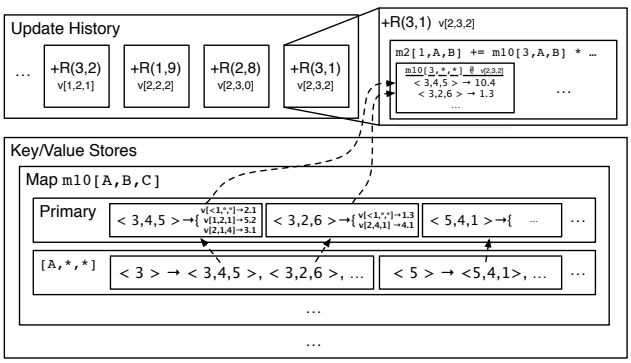

Figure 7: Datastructures used at each node. The update history stores updates since the last garbage collection together with a cache of precomputed map values at a particular version number. One key-value store is used for each map. If the key-value store does not support secondary indices, the same effect may be achieved by using an additional store for each secondary index required. The key-value store maintains one delta for each version in which a map entry entry was modified.

```
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[5]. 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.

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

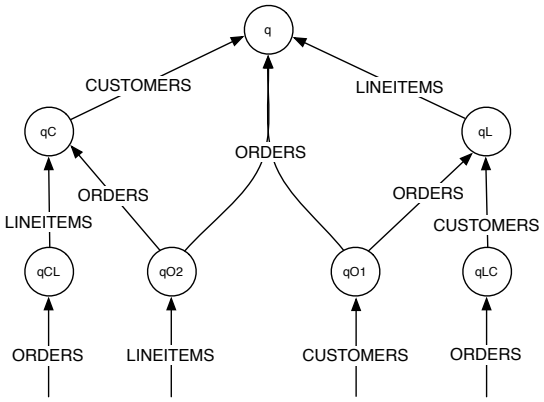

Figure 8: Dataflow graph for the M3 program representing a streaming TPC-H-style query ${}_{C.cid}G_{SUM(L.price)}(C \ldots \bowtie L \ldots \bowtie O \ldots)$

We can examine the effects of a cascade by considering the dataflow hypergraph of an M3 program, such as the one in Figure 8. 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 [7], 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.

Example 4.1. *Consider the program*

```
ON R(A): m1[A] += 1
         m2[A] += m3[A]
ON S(A): m1[A] += m2[A]
ON T(A): m3[A] += m1[A]
```

The dataflow graph for this program is presented in Figure 9. Though no single event uses or modifies the same maps twice[6], it is possible for a poorly timed event arrival order to cause each successive event to generate a progressively larger number of messages. In this example, consider the ordered event sequence `T[1],R[1],S[1]`*, causing the following messages to be dispatched:*

1. *Event* `T` *arrives at m1. m1 sends the current value of* `m1[1]` *as an delta for* `m3[1]` *to m3.*
2. *Event* `R` *arrives at m1 and m3. m1 is updated. m3 sends its current value* `m3[1]` *to m2.*
3. *The delta for Event* `T` *arrives at m3, changing the value of* `m3[1]`*. m3 must now update its delta to* `m2[1]` *and send a correction to m2.*

[6]Even if one event were to read and write to/from the same map, there is still an explicit order of execution across statements

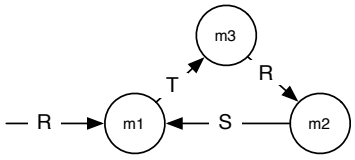

Figure 9: Dataflow graph for an M3 program with a dataflow cycle.

4. *Event* `S` *arrives at m2. m2 sends its current value* `m2[1]` *to m1.*

5. *The delta for Event* `R` *arrives at m2, changing the value of* `m2[1]`*. m2 must send a correction to m3.*

6. *The correction to the delta for Event* `R` *arrives at m2, changing the value of* `m2[1]` *again. m2 must send another correction.*

We refer to such an event as a *cascade*. If there is no cycle in the graph, the number of messages in a cascade is bounded by

$$\frac{1}{2}[max\_path\_length(dataflow\_graph)]^2$$

Otherwise, this process can continue indefinitely, increasing the number of messages with each event.

To prevent this, Cumulus buffers and aggregates deltas for a short period of time after the event that causes them. While this does prevent a cascade, it bounds the number of messages sent by the cascade.

4.3 Coordinator Favoritism

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.

4.4 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.

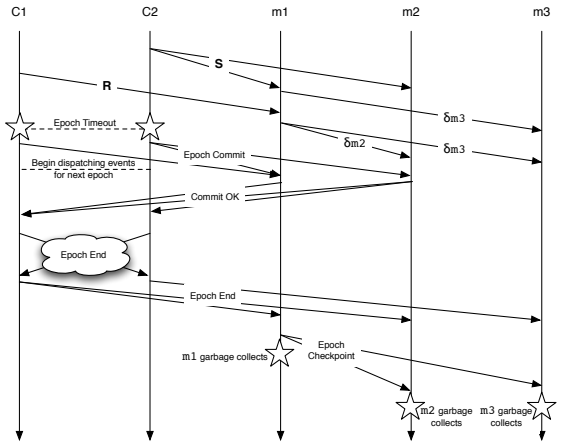

Figure 10: Timeline of Garbage Collection for one Epoch in Cumulus; The program being executed is `ON R() DO {m1[] += 1; m2[] += m1[]}; ON S() DO {m2[] += 1; m3[] += m1[]}`

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 11: Garbage Collection Algorithms

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 an `epoch commit` message to all nodes on the list. It then continues processing as normal. The garbage collection process is also illustrated in Figure 10.

When a node receives an `epoch commit`, 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 an `epoch commit` to, it broadcasts an `epoch end` to all other coordinators. Once all coordinators have sent an `epoch end`, the coordinators split the work of broadcasting an `epoch end`

across the cluster.

Once a node receives an `epoch end` message, it will never again receive an event for a prior epoch. Now, the nodes must ensure that no further map deltas will need to be propagated. Using the dataflow graph, a node can determine from where it receives deltas, and to whom it sends deltas.

If the graph is a DAG, garbage collection is simple. The node waits until it receives an epoch complete message from its coordinator, and an `epoch checkpoint` from all in edges. It then sends an `epoch checkpoint` to each out edge, and invokes the cleanup process described in Figure 11.

If the dataflow graph has cycles, as in Example 4.1, one node in the cycle is designated as the cycle leader. When this node receives an `epoch end`, it sends out a `partial checkpoint` to the next node in the cycle. The `partial checkpoint` is forwarded to subsequent nodes in the cycle. Each node uses the checkpoint message to ensure that it has received an `epoch end` and `epoch checkpoint`s from all nodes not in the cycle and that it has processed all pending messages from the prior node before forwarding the `partial checkpoint`.

If the cycle leader has not received any deltas from the prior node in the cycle by the time the `partial checkpoint` completes a full traversal, it sends a `partial checkpoint complete` to the nodes in the cycle, informing them that the cycle has completed its checkpoint. If it receives a delta in the interim, it begins the process anew.

An additional step may be introduced to deal with a node that belongs to two or more cycles. One cycle leader is designated as the primary. Before sending a `partial checkpoint complete`, the cycle leaders all block on the primary. Once all cycle leaders are ready, the primary queries all nodes that cross cycles and the nodes respond whether or not they sent a delta after sending at least one `partial checkpoint` message. If it did, the primary restarts the partial checkpointing process. Otherwise, the primary proceeds as normal.

The act of garbage collection has a very useful added 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

7. REFERENCES

[1] Y. Ahmad and C. Koch. Dbtoaster: a sql compiler for high-performance delta processing in main-memory databases. *Proc. VLDB Endow.*, 2(2):1566–1569, 2009.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[3] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[4] D. J. DeWitt and M. Stonebraker. http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/.

[5] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, March 1997.

[6] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[7] C. Koch. Incremental query evaluation in a ring of databases. In *PODS*, 2010.

[8] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD Conference*, pages 165–178, 2009.

[9] 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 *VLDB*, pages 1150–1160, 2007.

[10] Transaction Processing Performance Council. *TPC Benchmark H (Decision Support)*, revision 2.8.0 edition, 2008. http://www.tpc.org/tpch/spec/tpch2.6.0.pdf.