Cumulus semester paper

Aleksandar Vitorovic
Dept. of Computer Science
EPFL
aleksandar.vitorovic@epfl.ch

ABSTRACT

This paper presents issues related to my semester project in the Prof. Koch's DATA lab.

Keywords

Incremental View Maintenance, OLAP updates, Parallelization

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., [14, 9]).

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 [6, 12], 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 [5, 4]. 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 distributed key-value stores (maps) – 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.

This paper brings SQL expressiveness along with efficiency od distributed key-value store. SQL is translated to a trigger-based vector processing language, called M3, using DBToaster[1], an aggressive recursive incremental view maintenance 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, extremely efficient message passing C++ programs that incrementally maintain materialized views of the queries. Using Incremental View Maintenance [11], the complexity of query response times can be substantially reduced. Yet, by applying incremental view maintenance recursively, the complexity of query response times is trimmed down to constant time, for a limited set of 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 [7]) 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]. This is proven in the Koch's Algebra paper [10], in the case when polynomial amount of hardware and space is available.

M3 programs are interpreted within our runtime system, Cumulus. The Cumulus is a thin execution layer that lives above generic key-value stores and implements the message passing protocol. It can efficiently process a wide range of incremental view queries. The query results are stored in the distributed key-value store itself, as ordinary data.

Cumulus is a multiversion timestamp-based concurrency control system running in a shared-nothing main-memory environment. The data is distributed among processing nodes, and an user may submit an OLAP update just as the data was local. Out-of-order execution of updates are fully supported, and if there are violated dependencies, additional messages, called corrective update messages, are sent.

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.

[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.

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 a little more than a distributed key-value store.

Next section present system overview and requirements, in the context of M3 programs generated by DBToaster. Section 3 presents the the message passing protocol guiding reader through existing data dependencies and logical components in the system, further explaining consistency achieved and possible optimizations. In Section 4 the experiments performed and their results are explained, while Section 5 contains related work. Section 6 presents possible future improvements of the system, while Section 7 coalesces all the ideas grasped from the work.

2. SYSTEM OVERVIEW & REQUIREMENTS

A user would like efficiency of C programs, while declarative nature of SQL. All data will be stored in main memory in order to achieve better performance. The gap will be shrink by introducing M3 compiler which will compile OLAP database updates to C language constructs [2]. M3 compiler will generate triggers to cover all OLAP database updates in the system. Only insert and delete triggers are generated, since updates on tuples could be implemented as series of deletes followed by appropriate inserts. A trigger is coupled with trigger arguments, which are variable/value pairs used for representing parameters from particular OLAP update. In further text OLAP update and trigger instantiation will be used interchangeably.

EXAMPLE 2.1. An insert trigger and its possible instantiation (OLAP update):

```
ON +R(A,B){
   m1+=m2[A];
   m2[A]+=m3[B];
}
ON +R(10,20){
   m1+=m2[10];
   m2[10]+=m3[20];
}
```

Data structures. The only data structures used by transition programs are maps. A map contains a unique name and a pair {key, value}. Multikeyed pairs are supported as well. A map reference contains only map name and key(s). A map collection consists of a unique map name and the pairs from whole map domain.

A trigger consists of multiple statements which perform operations on maps. An M3 program reads from the right-hand side maps, and write to the left-hand side map of the statement. The map name and the trigger arguments specifies map to be accessed. If a variable used in a statement in a trigger is not defined in the trigger arguments, whole map domain participates in the statement. On the right-hand side of the statement the only allowed operation is *, while between left and right side only += is permitted. Each statement adds a *delta* to the left-hand side map. If there is need for adding multiple terms, it could be adjusted by utilizing multiple statements with the same left-hand side map. Both read and write are permitted to perform on non-existing maps. In that case, it is assumed initial value was zero.

M3 compiler generates only acyclic map data dependencies flow graph programs. In runtime, execution of M3 programs must provide the following guaranties:

- OLAP updates execute atomically with respect to each other. The effect has to be the same as whole OLAP update sequence was processed sequentially.
- Statements within same OLAP update use right-hand side maps from the state of the system right before current OLAP update start executing.

Update stream (i.e. ticker feeds, sensors, static files, relational databases, or even other M3 programs) will continuously inject triggers instantiated with arguments. The aim of our system is to process those OLAP updates and add deltas to the appropriate maps.

Since there is large amount of data, maps should be distributed among many processing nodes. The only way processing nodes could communicate is via sending and receiving messages. In order to achieve constant execution time complexity and support embarrassing parallelism, which is our goal, a designed protocol of sending/receiving messages has to avoid complex map locking schemes and support map updates that do not interact with each other to proceed simultaneously.

Also, our system has to be tolerant to reordering of messages which occurs in large networks. We assume the network supports pairwise in-order-delivery, but different processing nodes may observe different order of messages arrived from the multiple processing nodes. Our system may only rely on the property that every sent message will eventually arrive on its destination; messages could not be lost in the network.

Delay is another network characteristic which may significantly degrade overall performance. Only one message sent to all other nodes in a 10.000 processing node system will effect in delay for several seconds. Therefore, it is extremely important to build protocol based on the lowest possible number of exchanged messages.

Different queries will have different materialized view and will operate on different maps, supporting parallelism inherently. Our goal is to parallelize OLAP updates relevant to one complex SQL query a user is interested in. That way, not just throughput, but also latency could enhance.

We generally assume that the data is persistent in the OLTP databases. However, our system is also able to log update streams and replay it from the log later to recreate the data warehouse in main memory. We could also log the map states using techniques as surveyed in [13] for faster recovery, but this is currently not done.

3. SYSTEM DESIGN

3.1 Data Dependencies

In this section, we will present types of dependencies, and how they could be violated, while Section 3.3 presents the structures used for avoiding dependency violations.

There exist three types of dependencies corresponding to different statements `STMT1` and `STMT2`:

TS	OLAP update	Statement
1	ON +R(1,4)	m1 += m2[1]
		m2[1] += m3[4]
2	ON +R(1,5)	m1 += m2[1]
		m2[1] += m3[5]

Figure 1: W-R on m2[1] for Example 2.1.

- `Write-Read`: `STMT2` is later in an update stream and reads from a map that `STMT1` writes to
- `Read-Write`: `STMT2` is later in an update stream and writes to a map that `STMT1` reads from
- `Write-Write`: Both `STMT1` and `STMT2` update the same map

Write-Write dependency violation may occur even within a single OLAP update. Write-Read and Read-Write dependencies within the same OLAP update could not be violated since, according to M3 program guaranties, no map read will return the value from the current OLAP update. Therefore, we imply logical order between statements, based on Timestamps of the OLAP update a statement belongs to. In case of a single update stream, Timestamp (TS) is a implicitly defined unique serial number of a OLAP update in that update stream. For multiple update streams, some Total Order has to be introduced. We could introduce any total order as long as all OLAP updates inside one update stream preserve original order. To conduct unique global TS, we may combine a TS from particular update stream with ID of the update stream. The update stream ID may be generated utilizing the Standard Renaming Technique [3]. Introduced Total Order always exist, since map data dependencies are acyclic.

Consider an example shown in Figure 1. For the introduced total order, a Write-Read dependency exists on m2[1] between the second statement in the first OLAP update and the first statement in the second OLAP update.

Example 3.1. Consider the following insert triggers and a sequence of OLAP updates:

```
ON +R(A){
   m3[C]+=m1[A]*m2[C];
}
ON +S(B){
   m2[B]+=1;
}

TS1: ON +R(1)
TS2: ON +S(2)
```

Since the variable C in the `+R(A)` trigger is not present in the trigger arguments, loop over whole m3 domain is implied. If statements from `+R(A)` and `+R(B)` are performed on different processing nodes, the OLAP update from TS2 may complete before the OLAP update from TS1 read m2[2] value. Thus, Read-Write dependencies are violated.

In addition, an OLAP update should be processed atomically. The result of OLAP update has to be same as all statements inside a OLAP update were executed at the same point in the time. In other words, all OLAP updates have to be executed in some total order.

Example 3.2. Atomicity could be violated in

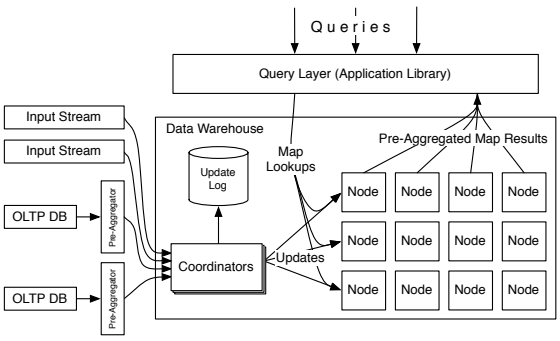

Figure 2: System Architecture

```
ON +S(A){
   m1[A]+=m2[A]
}
ON +T(B){
   m2[B]+=m1[B];
}
```

when next update sequence emerge:

```
S(1)
T(1)
```

Necessary conditions are:

- For the update sequence, statements from `+S(A)` and `+T(A)` are performed on different processing nodes.
- Both OLAP updates synchronously read map on the right side then synchronously write to map on the left side.

Note that atomicity can not be violated by writing to a single map since and all processing on a single processing node is done sequentially.

Even if atomicity is preserved, imposed total order may differ from actual OLAP update execution order. For example, if imposed total order was `S(1), T(1)` and actual OLAP update execution order was `T(1), S(1)`, both the Read-Write and the Write-Read dependencies are violated.

3.2 System components

Overall system architecture is presented in Figure 2. There are two logical components in the system. *Switches* maintains information about map partitioning and, by sending messages, delegate map updates from update streams to a number of data storage and query processing nodes, or *Data Warehouse (DW) Nodes*. The DW Nodes store maps and perform computations on them. The *client* sends queries which will probe the system and return a requested set of maps. Separate messages have to be sent between a pair of processing nodes, since multicast is not efficient, and in cloud computing not supported at all.

Using only one switch simplify system design since arriving OLAP updates could be easily serialized. Single point entry in the system inherently generate order between OLAP updates without using TS avoiding violation of data dependencies when introduced total order differs from actual OLAP update execution order. On the other hand, single-switch system implies some serious performance limitations. Firstly, the switch will be bottleneck, so the system will not

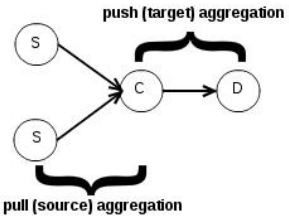

Figure 3: Push and pull aggregation

scale. Secondly, a single point of failure property will exist in the system by design. Therefore, our system will contain several switches, i.e. one switch per update stream.

Each statement in a OLAP update employs three classes of actors:

- Source nodes: The nodes hosting the right-hand side maps.
- Computation nodes: The nodes where statements are evaluated.
- Destination nodes: The nodes hosting the left-hand side map.

Assume we have a statement m+=m1[a]*m2[b]. We need to identify the sources (at most two DW Nodes), to compute the result and store the result to the destination. Those actors are logical entities: it is not necessary (and in fact, typically detrimental) for the actors to be on separate physical nodes within the cluster. The system may employ different mappings of logical DW Nodes distribution (source, computation, destination) across physical hardware in the cluster for each invocation of each statement.

Destination-computation. Co-locate the processing of an M3 statement (the computation nodes) with the DW Node storing the left-hand side map (the destination node). Each source node transmits all relevant maps to the destination node. Upon arrival, the destination node evaluates the statement and stores the result.

Source-computation. Instead of processing map updates at the DW Node where the result will be stored (the destination node), precompilation allows our system to ensure that all data necessary to process a statement is co-located. Rather than storing the results locally, a replica of the map is created at each DW Node that reads from the map. When evaluating an map update, the computation can be performed instantaneously.

However, it is typically not possible to generate a partitioning of the data that ensures that for each statement in a M3 program, all the source nodes will be co-located. Due to replication, our system may have multiple destination nodes, so the overhead of transmitting the result to the destination nodes is introduced. While replication is typically a desirable characteristic, storage-constraint infrastructures may need to use complex partitioning schemes. Thus, our system will use destination-computation approach.

Figure 3 presents both Destination-computation and Source-computation approaches.

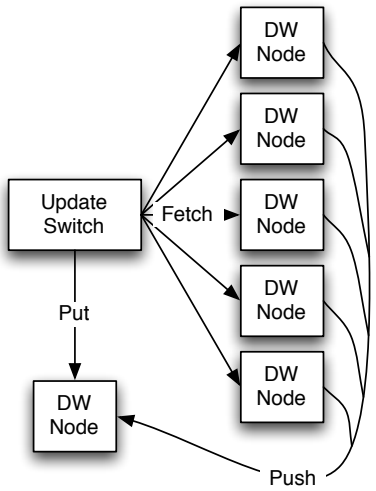

Figure 4: Messages between a switch and a DW Node

A set of messages issued by a switch for destination-computation approach is illustrated in Figure 4. FETCH messages are sent to DW Nodes that contain the right-hand side maps. In further text such DW Node will be denoted as the DW Read Node, by role they play in current statement. The DW Read Node will send PUSH message containing the data read to the DW Node that contain the left-hand side map. In further text such DW Node will be denoted as the DW Write Node, again, by role they play in current statement. Same DW Node may be the DW Write Node for one statement and the DW Read Node for some other statement. The DW Write Node will be aware of the map update to perform by receiving PUT message from a switch. For each statement, there is exactly one PUT message (since only one map can be the left-hand side) and one or more FETCH/PUSH messages (since there may exist multiple source nodes).

The content of the messages in the system will be shown on the first statement in Example 2.1. In order to reduce the network traffic to a minimum, a switch need not to send to a DW Node all the operations that will be applied on the maps. If each DW Node contains current M3 program, a switch may send to a DW Write Node only a unique ID of the statement. The ID of the statement could simply be a line number in a file which contains all statements in M3 program.

The system could coalesce all messages with the same source, destination and TS into a single one. Multiple FETCH messages per TS may be sent only if there are multiple DW Read Nodes per OLAP update, similarly, multiple PUT messages per TS may be sent only if there are multiple DW Write Nodes per OLAP update. For all types of messages, since DW Nodes are aware of current M3 program and its triggers, a unique ID of trigger, instead of a unique ID of statement, is sent. Thus, the message traffic is significantly reduced. In addition, single message per Timestamp forbids outside world to see Timestamp-inconsistent version of the map (i.e. two same left-hand side maps are updated withing single OLAP update and result of the first one is visible to the outside world).

A switch will send a FETCH message to each DW Read Node containing some of the right-hand maps (in our exam-

ple to a single DW Read Node containing m2[10]) with the following encapsulated information:

- Trigger ID
- Trigger arguments relevant to the DW Read Node
- TS of the OLAP update
- A reference to a DW Write Node (in our example the DW Node containing m1)

A DW Read Node can extract information which maps to send from Trigger ID. A DW Read Node will check all the right-hand side maps from the OLAP update, and send each of them which is stored on current DW Read Node in a single PUSH message to the all relevant DW Write Nodes.

A DW Read Node sends a PUSH message to the DW Write Node with the following encapsulated information:

- Trigger ID
- TS of the OLAP update
- A list of the right-hand side maps from the DW Read Node (in our example m2[10])

A switch will send a PUT message to the DW Write Node with the following encapsulated information:

- Trigger ID
- Trigger arguments relevant to the DW Write Node
- TS of the OLAP update

A DW Write Node could match a PUSH message with a corresponding PUT message by TS. A DW Write Node could not complete processing a PUT message until the PUSH messages from all corresponding DW Read Nodes arrived. For each statement within a trigger, the DW Write Node will compute the right-hand side of a statement and add computed value to the left-hand side map. (Only "+=" operation could be applied to the left-hand side map). Note that even non-existing maps must be initialized to zero and sent; the DW Write Node needs to hear from all DW Read Nodes in order to proceed with its write.

The DW Node utilizes the *ReceiveLog* and a list of boolean flags per statement for matching PUT and PUSH messages. The ReceiveLog, consisting of pairs {map, TS}, is responsible for storing all maps from the PUSH messages. When a PUT message arrives, the DW Write Node will initialize the lists of boolean flags for each statement in the OLAP update the PUT message refers to. Each arrived PUSH message will append ReceiveLog and set appropriate boolean flag for the corresponding PUT message. (It may happen to have multiple boolean flags across OLAP update if the arrived map is appearing in multiple right-hand side maps.) If some PUSH message arrives before the corresponding PUT message, a list of boolean flags may be initialized, utilizing Trigger ID from the message. From arrival of the PUT message to the time when all necessary boolean flags are set, processing the PUT message is suspended, so other messages may arrive (including awaited PUSH messages). Until completing PUT message processing, no messages with higher TS is processed, in order to prevent data dependency violation.

Upon all necessary boolean flags are set for a statement, map update is performed, using accumulator. Since the only operation on the right-hand side is multiplication, the accumulator will set its initial value to 1. Then, the value from accumulator is multiplied with each the right-hand side map, and again stored in accumulator, until all of them is processed. Finally, += operation between the left-hand side map and accumulator is performed.

A client could obtain some query answer from our system by sending queries to a switch. A switch will issue a QUERY message encoded, among with TS and the reference to the switch a client is communicating with. After receiving the maps from the DW Nodes, the switch will submit them to the client. Alternatively, a client may access the DW Nodes directly, using a map partitioning information obtained from a switch.

If we assume partitioning is static, information about map distribution could be sent to all DW Nodes before any OLAP update arrived. Thus, the FETCH messages need not to contain relevant DW Write Node information, providing additional message traffic reduction.

3.3 Corrective updates

In this subsection, steps taken upon dependency violation is presented.

As already mentioned, Introduced Total Order may not reflect actual OLAP update execution order. A DW Node may first receive a message from a switch with a greater TS and potentially violate dependencies. A DW Node itself cannot be aware whether some OLAP update will arrive later due to processing on a slow switch and/or network delays, or will never arrive since it is irrelevant to our DW Node (no maps stored in our DW Node in particular TS).

Generally, our system always uses an optimistic approach, continue whenever possible, and if some dependencies are violated, employ the *corrective update* mechanism. Since increasing number of switches and/or update streams yields more messages in the system, there is higher probability for delays and reordering of messages, violated data dependencies and finally, corrective update messages.

Our system could recognize dependency violation by combining information about maps sent and received along with their TS. For that purpose, we will take a look at a simple statement mw+=mr. Assume the DW Read Node contains mr, while the DW Write Node contains mw. All information about sent map are presented in the mr(TSx, TSy) notation. There are three types of dependencies, we will tackle them separately:

- `Read-Write`: If at the DW Write Node, after applying write in TSy, arrive a FETCH message referring to mw with TSz such that $z < y$, the DW Write Node will send some previous version of mw.
- `Write-Write`: If at the DW Write Node, after applying Write on TSy, arrive a PUT message referring to mw with TSz such that $z < y$, both versions of mw could still be needed, depending of future arrival of messages and their TS. Since there is only one PUT message per DW Write Node in a OLAP update, Write-Write dependencies could not be violated within the single Timestamp.
- `Write-Read`: If at the DW Read Node, after sending mr, arrive a PUT message referring to mr with TSz such that $z < y$, Write-Read dependency is violated.

At the time of sending mr, the DW Read Node can not be aware whether the sent map is consistent, or the PUT message with TSz will arrive later due to the network delays. Thus, the DW Read Node optimistically send the most recent version of the map prior to TSy. However, when the PUT message with TSz arrives, the DW Read Node has to send the delta applied to mr in TSz. The message containing the delta is denoted as the *corrective update* message.

Since whole algebra is incremental, writes to the same map can be applied in any order, as long as others receive correct version (or a delta in the Write-Read case).

The goal is is to ensure that a processing node never has to wait to read a map, therefore multiple versions of each map is maintained, each with a TS. Therefore, the *WriteLog* structure, consisting of pairs {map, TS}, is exploited. Thus, Write-Write dependencies could not be violated, while violating Read-Write dependencies is avoided by reading the most recent version of the map prior to the Read TS. For preserving Write-Read dependencies, the WriteLog is also used, but a delta from versions of the map is extracted. On the DW Read Node side, upon receiving a PUT message referring to map mr with TSz, corrective updates will be sent to each DW Write Node which used map mr as right-hand side map for their computation in TSy such that $z < y$. The *SendLog* structure will maintain pairs {map reference, reference to DW Write Node, TSy}, so our system could iterate over this structure exploring the entries fulfilling the condition.

Note that the pairs from the ReceiveLog and the WriteLog are the same. The only difference is that the WriteLog maintains history of the maps on the current DW Node, while the ReceiveLog maintains history of the maps from the other DW Nodes. In addition, the SendLog will be utilized even if the right-hand side map and the right-hand side map are on the same DW Write Node, since Write-Read dependencies may be violated even on a single DW Node (i.e. messages regarding different TS arrived from different switches out-of-order).

Corrective update message is similar to a PUSH message, but with different TS. It consist of:

- TSx of the last right-hand side map update on the DW Read Node (last mr update in our example)
- A delta of the right-hand side map (the delta of mr in our example)

Note that the PUSH message contains TSy, while TSx is utilized for corrective update messages. PUSH message refers to exactly one Timestamp on the DW Write Node, while corrective update has to be applied on each OLAP update which logically succeeds TSx and use map from corrective update message.

Upon arrival, all types of messages on a DW Node, including FETCH, PUSH, PUT and corrective messages, are stored in the same prioritized buffer by their TS. Thus, the frequency of corrective updates will be amortized.

At the time of processing of corrective update message on the DW Write Node, the following steps have to be taken:

(1) Update all the mr in the ReceiveLog which logically succeeds TSx.

(2) For each map update performed on current DW Write Node that logically succeeds TSx, if mr was used as the right-hand side map, apply map update to each affected pair {mw, TSy} in the WriteLog structure. In order to support this, DW Node log all OLAP updates (TS, trigger and trigger arguments) in which it performed any write operation in the OperationLog. The result of a multiplication between the delta on the arrived map and the values of all the remaining right-hand side maps from the ReceiveLog, is added to affected mw. If the delta is applied to a pair {mw, TSy}, it has to be applied to each mw in the WriteLog structure succeeding TSy. This is due to implicit Write-Read dependencies between any two += operations with the same left-hand side map.

(3) Corrective updates could be recursive, since the DW Write Node could propagate incorrect mw further to the DW Write Nodes that used mw as the right-hand side map. For each map reference in the SendLog concerning mw which logically succeeds TSy, corrective update mechanism is applied recursively, but on the {mw, TSy} pair.

The recursive mechanism will be more efficient if all mw maps on the local DW Node is firstly updated.

Since corrective update refers to a map which already exist in the SendLog, the corrective updates will never create new entry in the SendLog. Furthermore, there is no need for maintaining complete history of maps in the SendLog, yet only to have highest TSy for each pair {map reference, reference to DW Write Node}. A value y may only increase, and if for some arrived corrective update with TSz such that $z < y$, it will hold after y increased. The SendLog structure do not store information about last map update for which corrective update was sent, yet for the highest TS of sent map.

We will show now that corrective update mechanism terminates. A finite number of PUT messages will arrive out of order to a DW Read Node. On a DW Write Node, a finite number of corrective messages will arrive and a finite number of recursive corrective updates will be sent. Since data flow graphs are always acyclic, recursive corrective updates are bounded by the maximum path length in the data flow graph. Due to the fact that some total order was introduced, corrective updates will eventually terminate. Overall, this process terminates and all maps will eventually achieve correct value.

On Figure 5, an enhanced example of out of order execution is presented. Messages in the same line are simultaneous, and arrive before messages from next line. Their logical ordering is specified in parenthesis in the figure. The DW Read Node stores map mr, which is used as the right-hand side map in computations of mw on the DW Write Node. Both PUT and FETCH messages on the DW Read Node corresponds to mr, while PUT messages on the DW Write Node corresponds to mw. For the PUT messages on the DW Read Node there is no corresponding FETCH messages, since mr+=1. Assume PUT(TS10) on the DW Read Node comes before TS7 on both DW Nodes, as depicted in the figure. The WriteLog of the DW Read Node is employed, and the most recent mr with TS less than 7 is sent (in this case value from TS2). The same value will be sent for PUT(TS9). However, for PUT(TS11), mr value from TS10 will be sent. In our example PUT(TS6) may arrive

DW Read Node	DW Write Node
PUT(TS2)	
PUT(TS10)	
FETCH(TS7)	PUT(TS7)
FETCH(TS7)	PUT(TS9)
FETCH(TS7)	PUT(TS11)
PUT(TS6)	

Figure 5: Possible arrival of messages from a switch to the DW Nodes.

later, so it is time to exploit the SendLog structure. Corrective update will be sent to all DW Nodes which demanded mr with TS greater than 6 (in this case including DW Write Node). Note that the DW Write Node could not be aware whether PUT(TS6) will arrive later, or it is irrelevant to the DW Write Node and will never arrive.

Multiple version of the same map has to be preserved in the WriteLog structure, but a DW Node may not always send the most recent version of the data. In literature this multiversioning is regarded as the *1-copy serializability* [8].

EXAMPLE 3.3. Recursive corrective updates is illustrated in the following trigger:

```
ON +S(A){
   m1[A]+=m2[A]*m3[A]
}
```

when next update sequence emerge, as seen by DW Node containing m1[1] map:

```
TS2: S(1)
TS3: S(1)
TS1: CorrectiveMessage(m2[1])
```

The delta is extracted from the corrective update message. Then, the OLAP update from TS2 is examined, since m2[1] is on the right-hand side. The result of multiplication of the delta and the pair {m3[1], TS2} from the ReceiveLog is added to m1[1] in TS2, and TS3 as well due to implicit Write-Read dependency on m1[1]. The OLAP update from TS3 is now glanced. Similarly, the result of multiplication of the delta and the pair{m3[1], TS3} from the ReceiveLog is added to m1[1] in TS3.

Now we will see that corrective update mechanism works the same way for atomicity violations. In Example 3.2 assume m1 is stored on DW Node 1 and m2 is stored on DW Node 2 and introduced total order of OLAP updates is `S(1), T(1)`. Figure 6 depicts two cases, with and without violated atomicity, both exploiting corrective updates (shown in red). In Figure 6(a) a non-linearizable OLAP update execution is presented. OLAP updates which should go one after another are actually executed at approximately same time. DW Node 1 will send m1, last updated at TS0, to be used in TS2 on DW Node 2. DW Node 2 will send m2, last updated at TS0, to be used in TS1 on DW Node 1. Upon receiving a PUSH message from DW Node 2, DW Node 1 will realize m1 is changed at TS1, but version from TS0 was previously sent. Corrective update take place and delta(m1) is sent to DW Node 2. After executing corrective updates on DW Node 2, the effect will be the same as OLAP updates were executed atomically in introduced total order. In Figure 6(b) violated order is presented. Actual OLAP update

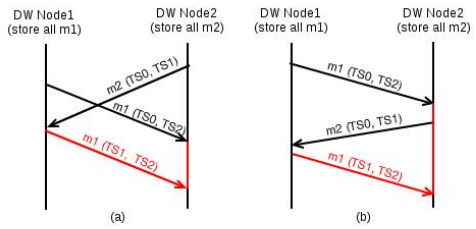

Figure 6: Atomicity(a) and Order(b) Violations.

execution order is `T(1), S(1)`. DW Node 1 will send m1 to be used in TS2 on DW Node 2. The most recent m1 is from TS0. Afterwards, DW Node 2 will send m2 to DW Node 1 to be used in TS1. The most recent m2 with TS less than 1 is from TS0. Now, DW Node 1 realizes it has to send corrective update to DW Node 2, since we now have m1 from TS1, and it is used on DW Node 2 in TS2. Note that in Figure 6 both TS are shown above line due to better understanding of messages. Still, only the TS of the map last update is actually sent.

Two conclusions are inflicted:

- Corrective updates are always sent from DW Node which contains maps that should be written earlier in introduced Total Order.
- When the introduced total order differs from actual OLAP update execution order, no matter whether atomicity is violated, corrective update mechanism works the same way.

3.4 Consistency

Our system 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 system. In order to provide consistent version of all maps, snapshots may be employed at certain points in time, without using a centralized point of synchronization.

We already introduced total order, so our system may use a background task to periodically identify the furthest point in the update streams that has been fully propagated. For all maps, the values set in the most recent TS which logically precedes furthest point TS will be used as consistent version. Periods between two consistent version of all maps are called *epochs*. Though this process is slow, it does not interfere with any processing nodes normal operations, and can be performed infrequently, for example once every few seconds.

The process may begin by sending an EPOCH END from a switch to all other switches. All switches are aware of the DW Nodes they communicated with, so they will send an EPOCH END to each of them. Each DW Node has to provide the last processed TS, but corrective updates has to be taken into account.

DW Nodes must ensure that no further map deltas (corrective updates), regarding a OLAP update which logically precedes the last processed TS, is received after sending the furthest point TS. Using the data flow graph constructed from the WriteLog and the SendLog, a DW Node could determine the DW Nodes the deltas were sent to. Because this

graph is acyclic, deltas could be flushed along it. The process will start from maps/DW Nodes which depends only on input, not on other maps/DW Nodes, and then it will propagate down the graph. After the propagation reaches all the maps on a DW Node, the DW Node will send EPOCH READY message to the switch containing the last processed TS.

Each switch will compute the minimum TS received from corresponding DW Nodes and forward the TS within EPOCH READY to all other switches. After receiving EPOCH READY from all the other switches, a switch will compute minimum value among them. Once all switches computed the minimum value, the switches split the work of broadcasting EPOCH COMMIT containing the minimum TS value across the cluster. Once a node receives an EPOCH COMMIT message, it will never again receive an event for a prior epoch.

Our system will provide mechanism to avoid a single switch issues EPOCH END messages all the time. This is easily achieved by a token-passing system and ordering switches. Only the switch with the token can switch epochs. It waits for the duration of one period, switches, and then passes the token to the next switch.

The Log structures grow over time. To prevent unbounded memory usage, we may employ the same background task we used for consistent views to periodically truncate, or garbage collect the entries in each Log structure. The WriteLog and the SendLog are truncated at this point, and all writes before this point is coalesced into a single entry. The entries from the ReceiveLog which precedes the consistent image TS may be safely deleted. They must not be deleted earlier, since before EPOCH END the corresponding corrective updates may arrive.

Still, a DW Node may run out of memory before the switch with a token switched epochs. In order to avoid this, the DW Node may request end of current EPOCH by sending an EPOCH END REQUEST message to a switch.

As a conclusion, our system will employ a hybrid consistency model, similar to [15], in order to provide a user both a low-latency tentative data (the value from the highest TS), or a committed data (a value from the most recent snapshot of the system).

3.5 Optimizations

Bypass Optimization. Bypass refers to executing messages on a DW Node in a different order than their TS imply. This is helpful when a message cause DW Node to block and wait for some data to arrive. Thanks to the WriteLog structure, a message received from a switch may bypass an uncompleted PUT message. The PUSH, FETCH and corrective update messages could bypass an uncompleted PUT message if not referring to the same map the uncompleted PUT message is trying to write to. By this restriction, unnecessary corrective updates are avoided.

PUT Optimization. PUT messages sent from a switch to a DW Node may not be used at all. As explained in Section 3.2, by receiving a PUT message, the DW Write Node will be aware of map update that will be performed upon receiving all the right-hand side maps. However, the DW Write Node will be implicitly notified about map update from a PUSH message. If PUT messages do not exist at all in our system, the "blind bypassing" effect is achieved. By blind bypassing, messages will always bypass non-existing PUT message. Since our system could not check restrictions (do not bypass if referring to the same map), corrective updates messages have to be sent. In addition, the WriteLog structure should apply corrective updates as well. If corrective updates eradicated by absence of PUT messages do not cause more traffic than PUT messages themselves, this optimization could reduce network traffic.

PUSH Optimization. On the other side, we could also decrease traffic in network by reducing number of sent PUSH messages. If map hasn't been updated after last send to particular DW Node, there is no need to send it again. By applying this optimization huge gain is expected, especially when mostly small set of maps is updated.

The procedure for determining whether a PUSH message will be sent on a FETCH message arrival to a DW Read Node is the following. First, find the most recent TSx of the map prior to FETCH message TS. If the SendLog structure contains the most recent version of the map, a PUSH message is not sent to to the DW Write Node. For this purpose, the SendLog structure will additionally contain TSx.

The ReceiveLog structure will suffer from the following changes. Instead of TSy, an entry in the ReceiveLog structure should contain TSx (TS of statement where map got value which was sent). The DW Write Node always use the most recent version of the map prior to TSy.

The DW Write Node could not be aware whether it will receive something from DW Read Node or its current value in the ReceiveLog is the most recent one. Two possibilities may be employed:

(1) Wait for some timeout to give some time for a PUSH messages to arrive, or

(2) Optimistically continue and do corrective updates if the PUSH messages arrive.

If a switch sends multiple messages per OLAP update, the optimization will not work since from multiple PUSH messages per DW Write Node/DW Read Node pair per TS only the first one will be sent. This can be worked around by expanding the SendLog with the map values.

FETCH Optimization. It is possible to reduce size of a single FETCH message. If all DW Nodes contain information about map partitioning, a reference to a DW Write Node inside a FETCH message is redundant. However, the system has to inform all DW Nodes in case of map repartitioning.

Buffering corrective updates. The system may limit recursive corrective updates by buffering deltas for a short period of time after the corrective update message arrived, allowing multiple subsequent deltas to be aggregated together.

Map compression. Maps can apply compression techniques to address frequently repeating data.

Look ahead. When the system realizes that some patterns of accessing maps in the Log structures are repeating, prediction of future accesses may be utilized. An approach similar to cache memory may be exploited, so frequently requested maps could be faster accessed. If maps are continuously changing, out system may benefit from prediction of future map updates.

Work-space trade-off. The system could employ i-th order of materialization. Thus, our system amortizes space requirements causing additional work during OLAP update.

4. EXPERIMENTS

5. RELATED WORK

5.1 Comparison to Transactional Memory

We will compare our system to a classical Transactional Memory. Here, transactions are implicit at a OLAP update level. The order of transactions is defined with TS at the time of the entry in the system (reaching a switch). Our system will execute OLAP updates optimistically and store different versions of the data in the WriteLog structure. Instead to aborting and redoing an OLAP update, the system may perform the corrective update on the map, if Write-Read dependencies are violated. Since there is no aborts and multiversions are employed, unrecoverable schedules can not arise. Performing periodic garbage collection and consistent snapshot may be regarded as a commit.

A conflict arise if at least two OLAP updates:

(1) Execute statements in order not specified via introduced Total Order, or their execution overlap in time,

(2) Access same map and

(3) At least one of them performs write.

All of those is detected on per-DW Node basis, since upon receiving a PUT message the SendLog structure is examined. A conflict is resolved by using corrective updates.

When dealing with any kind of a distributed system, opacity has to be kept in mind. Until now, we discussed serializability of statements (or linearizability of OLAP updates). Opacity consists both of serializability and consistent memory view. Any side-effect due to reading values from an inconsistent map is reluctant. Typical examples for this is divide by zero or infinite loops [8]. Former is not applicable here since only addition and multiplication exist. Later is also not applicable since we do not have loops depending on indexes, only loops over whole domain exist.

6. FUTURE WORK

Rounds. We may force our total order by using rounds and enough timeouts between them. For example, after a particular write, reads depending on that write could be postponed for the next round. Still, there is no guaranty corrective updates will never occur, but it will be rare. However, this is hard to implement.

Timing assumptions. Different heuristics can be applied in our system. If a OLAP update with significantly lower TS than the DW Node is currently processing have not yet arrived, the DW Node could reasonable suppose it is irrelevant to our DW Node. This heuristic could be based on the estimated message travel time from a switch to a DW Node and the delta query level.

Batching. We only address single-tuple updates in this paper. Batching and set-at-a-time techniques have often been very successful in query processing. We believe that the compilation approach should remain unchanged – delta processing with sets of map updates does not result in code as simple as M3, while the execution of M3 triggers as generated by our current compiler can easily be batched. We can profit particularly from batching message content to send fewer messages in Cumulus runtime.

Partitioning. Map collections can contain different number of pairs. Collocating corresponding maps (via key-foreign key relationships) of different map collections on the same DW Nodes could significantly decrease network traffic. Database partitioning and co-clustering decisions are traditionally made based on a combination of:

(1) Workload statistics,

(2) Information on schema and integrity constraints and

(3) Expert insights into how database execute queries.

Ideally, such decisions should be based on a combination of (1), (2) and data flow analysis. We will tackle data flow analysis separately for one and multiple update streams. We could modify data dependencies in the system by introducing different total order.

In the case of one update stream, we can not modify introduced Total Order and TS. Analyzing an update stream in order to discover data dependencies is infeasible in the case OLAP updates arrive one at a time in the system. By the time long enough sequence of triggers is collected, computations could already be finished.

In the case of multiple update streams, we will first have to define term *switch OLAP update partitioning*. It refers to using particular algorithm for correlating OLAP updates to switches and assigning TS to them. By appropriate switch OLAP update partitioning, it is possible to make the introduced Total Order closer to the actual OLAP update execution order. That way, dependent statements will execute in secluded points in time, enough to avoid data dependency violation. As a result, this will decrease corrective updates frequency. Still, order of OLAP updates inside one update stream has to be preserved.

Using these modifications of data flow dependencies, partitioning and co-clustering decisions can be made by solving a straightforward min-cut style optimization problem.

Switch OLAP update partitioning and data partitioning are dependent on each other. If for a longer period of time different OLAP updates occur more frequently, both could be dynamically repartitioned. However, the operation is costly and should be applied very carefully.

Redundancy. Future work will also address using redundant nodes for dealing with node failures.

7. CONCLUSION

Our preliminary experiments show that the choices made in our system indeed allow us to realize the embarrassing parallelism and thus the scalability in a cluster that is promised by our approach of compiling to M3 programs. We have shown that the our system infrastructure scales linearly, both in terms of OLAP updates and client queries, with the number of nodes in the system.

The system we implemented avoids locks, but price is payed for preserving multiple map versions. The processing nodes never waits for the map, but the system compensates it using the corrective updates. We have seen that all dependency violations can be fixed by corrective updates, which will not significantly degrade performance. In most cases, corrective updates will not occur too frequently. That mechanism will enable us to elude building our implementation based on some special and expensive network features.

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

[3] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. John Wiley Interscience, March 2004.

[4] 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.

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

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

[7] 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.

[8] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, 2008.

[9] 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.

[10] C. Koch. Incremental query evaluation in a ring of databases. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems of data*, PODS '10, pages 87–98, New York, NY, USA, 2010. ACM.

[11] K. Y. Lee, J. H. Son, and M. H. Kim. Efficient incremental view maintenance in data warehouses. In *Proceedings of the tenth international conference on Information and knowledge management*, CIKM '01, pages 349–356, New York, NY, USA, 2001. ACM.

[12] 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.

[13] M. A. V. Salles, T. Cao, B. Sowell, A. J. Demers, J. Gehrke, C. Koch, and W. M. White. An evaluation of checkpoint recovery for massively multiplayer online games. *PVLDB*, 2(1):1258–1269, 2009.

[14] 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.

[15] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP*, pages 172–183, 1995.