SPREAD - DESIGN DOCUMENT

OLIVER KENNEDY, YANIF AHMAD, CHRISTOPH KOCH

1. High Level Flow

2. Control Layer

The control layer acts as a coordinator for the distributed warehouse infrastructure. Its responsibilities include

- Monitoring node liveness and load
- Maintaining the Data $\rightarrow$ Node map, ie, the layout
- Managing layout decisions (ie, deciding when to alter the layout)
- Coordinating changes in layout.

Prior discussions have suggested that the control layer be included as part of the middleware layer. Because the functionality is distinct from the user interface, it has been separated into its own component. However, there are several potential mechanisms by which the control layer may be implemented:

(1) As part of the middleware layer: The middleware layer acts as a central control point.
(2) As its own entity: The central control point distributes state to the remaining components.
(3) As a distributed system running on the warehouse nodes: The nodes manage each other.

The last of these seems like the most interesting, there wouldn't be anything particularly novel that could be added in this space. It also eliminates any central point of failure... but is immensely more complex. As an initial approach to this problem approach, #2 is the most general. Nothing prevents the controller entity from running on the same hardware as the middleware layer, and abstracting it out makes later changes (possibly to a distributed infrastructure) simpler.

2.1. **API/Events.**

Ping Node. Periodically attempt to contact a node. Node selection can be either random, sequential, or some hybrid thereof. As part of the contact, the control layer will check the node's memory, CPU and bandwidth utilization. These values are stored for use in Full Layout/Node Layout. These values are also compared to the global average; if they are outliers, Node Layout may be called immediately. If the node is down, a Node Event is generated.

Register Node. As a node is activated within the data warehouse, it registers itself with the control layer. Doing so triggers a node event.
Node Event. The addition or removal of a node within the network triggers a node event. This forces a rebalancing; Node Layout is called immediately. In either case, the layout is selected by the layout heuristic. See below.
Load Layout Heuristic. As part of its functionality, the compiler generates a Layout Heuristic; a (simple) program that generates a layout, given current load, per-map sizes, traffic patterns, and other live state. As part of the install process (and any time the datacube inputs are changed), a heuristic is loaded into the control layer. This starts a periodic process that calls Precompute Layout All immediately, and then again, periodically. This periodicity is determined by the heuristic after every run.

The heuristic include a list of maps required by the compiler. This function is responsible for creating those maps. As a desired feature, Load Layout Heuristic should figure out if there's any reasonable way to migrate to the new set of maps. However, for now the maps will simply be erased on load.

Additionally, the heuristic specifies a replica quota for each map. The control layer will ensure that partitions of this map have at least this many replicas at all times.
Full Layout. This method is called periodically to optimize the current layout given the live state of the network. This method invokes the layout heuristic, and obtains from it a set of proposed changes from the following list:

- Split Partition(Map, Radix, {DstLeft}, {DstRight}): Given Map, split the partition that contains Radix on either side of Radix. The newly created partitions are installed on the nodes in the sets DstLeft and DstRight respectively; If either of these values is NULL, the node(s) currently storing the parent partition is/are used instead.
- Merge Partition(Map, Radix, {Dst}): Given Map, coalesce the partitions on either side of Radix. The coalesced partition is installed onto the nodes in Dst. If Dst is NULL, half (randomly selected) of the nodes hosting the parent partitions will be used.
- Replicate(Map, Index, {Node}): Install partition Index of Map onto Node.
- Lazy Replicate(Map, Index, {Node}): Use spare bandwidth/cpu cycles to install partition Index of Map onto Node. The replica created by this operation is created piecewise as bandwidth permits.
- Delete(Map, Index, {Node}): Remove partition Index of Map from Node. This may be applied to lazy replicas.
- Lazy Move(Map, Index, {Src}, {Dst}): Similar to Lazy Replicate, except that once the replicas have been created on Dst, they are deleted from Src.

The operations issued by the heuristic are recorded, but not executed until after the heuristic completes. Prior to execution, the control layer performs simple validation on the layout resulting from these commands. The main purpose of this is to ensure that each partition has met its replica quota. If the quota has not been met, the control layer automatically replicates the partition, assigning new replicas to the nodes with the lowest load. When doing so, it accounts for estimated load under the new layout. Although Lazy Replicate() processes in progress do not count towards a partition's replica quota (until they finish), they are upgraded to normal Replicate() processes if necessary.

Once the operations have been validated and corrected if necessary, Apply Layout is called to manifest the new layout on the warehouse nodes

Node Layout. As an alternative to the (potentially very computationally intensive) full heuristic, a node localized version is also generated by the compiler. This node-localized version is used both to shunt load away from a node (either due to over-utilization, or due to node failure), or to shunt load towards a node (due to under-utilization, perhaps due to a recent hardware addition). Once the operations have been validated, Apply Layout is called.

Apply Layout. Apply Layout does two things. Firstly, it applies a series of layout commands created by the layout heuristic to the underlying warehouse nodes. Secondly, it ensures that the layout map is kept properly updated on both Middleware and Switch.

Each high level command from the layout heuristic is broken down into a sequence of low level commands for each node:

- Split Partition(Node, Map, Radix): Split the partition and store a copy on the local device.
- Merge Partition(Node, Map, Index): Merge two partitions living on the local host.
- Load Partition(Node, Map, Index, {Src/Priority}, Version): Register for updates to the specified partition and transfer the contents of a specified version of the partition from a given host.
- Delete Partition(Node, Map, Index): Remove a partition from the specified host.
- Join(): Block until all prior commands have been completed.

Commands are executed asynchronously; In most cases, the command is driven by the node to which it is applied. For example, when loading partition data into a node, the node is provided with a list of sources hosting the partition data, each with a (high/low) priority (depending on resources available to the node in question). The destination node does a piecewise data transfer from the other nodes, breaking the partition into chunks and requesting a chunk at a time from each of its sources.

Apply Layout also applies the relevant changes to the middleware and switch. This requires that the layout be distributed to all middleware nodes and the switch. As a point of interest, while consistency isn't a particularly huge issue (since we're replicating), we can still achieve it trivially. Split and Merge partition both apply to nodes that host both partitions in question, thus they are (effectively) a no-op with respect to the layout. With respect to Delete Partition, the actual deletion can be delayed until the layout changes have been pushed and no further requests are pending.

Load Partition is the only one where things get a little tricky. The partition needs to start receiving updates to the data before it starts loading, but can't respond to queries until after it finishes. Thus, we split the switch layout update into a Get and a Put component. Prior to Load Partition being evaluated, Apply Layout ensures that the switch has registered the node with the switch's put layout. After loading completes, Apply Layout updates the switch's Get layout and the middleware. There are some versioning issues as well, but these are described in the LoadPartition section.

2.2. **Datastructures.**
Layout Heuristic.
Layout.
Node List.
Pending Commands.

3. Middleware

The middleware layer provides an interface by which users may query the warehouse. It receives updates about the layout from the control layer, and presents a one-function interface to the rest of the world: Query. Because of this limited functionality, we can actually have many middleware boxes (as appropriate to the load being placed on them).

3.1. **API.**
Query({Col/Val}. Every query requires evaluation of one or more Get commands on a set of map partitions. How this happens is something that needs to be worked out. There are two factors of interest with respect to the Get commands. Firstly, load balancing. Each map of interest is replicated on a set of nodes. In the early implementations, we should just be able to send load to a random replica. Later implementations might coordinate with the control layer to do smarter load balancing.

The second issue is versioning. The tentative approach proposed here was to have the switch maintain and advertise a version map to the middleware layer. Updates to this version map are batched at least at the granularity of an update, so they remain consistent. Thus, every request is a request for the latest version in the version map. Note that in this context, we only need a version number for every warehouse node. Thus, we can use an abbreviated form of the version map, where we assign each node a version number that is the maximum of the versions of all maps stored on that node.

There's a better way to do this, but let's see if the simple approach works.
ReceiveLayoutUpdate.
ReceiveVersionMapUpdate.

3.2. **Datastructures.**
Layout.
VersionMap.

4. Switch

The switch acts as a sprayer for database updates. It maintains three datastructures: a layout obtained from the control layer, directives generated by the compiler, and a version table that it maintains by itself.

Each update triggers the execution of a set of directives, where every directive is a Put command that modifies an existing map by the result of an algebraic expression defined in terms of constants, values from the triggering update, and the result of any number of Get expressions. These operations are implemented within the warehouse. A message is sent to a random Put replica which sends Get requests to the relevant nodes. The Get nodes send their results to the Put node where the value is computed and the result forwarded to the other replicas. Once the value is fully replicated, the Put node responds to the switch.

Note that this makes it possible to have a (potentially very large) number of updates in-flight within the warehouse at any given time. Consistency is addressed by having the switch maintain a version map, matching each partition with a version number. Every update is sequentially assigned a version number. The switch keeps track of which partitions are updated by a particular update and places the update's version number in the version map.

Before issuing the Get requests for a directive, the switch first queries the version map to identify the latest versions of the relevant partitions. These version numbers are sent with the Get request. This requires that nodes store a small window of older versions of the map until it can be guaranteed that there are no further get requests for that version in flight. Thus, as part of the version map, the system records the set of versions of each partition being accessed.

Prior to issuing a Put, the system ensures that either no instructions in-flight are reading from that value, or forces the Put to save the last version of the map. In the latter case, the switch sends a Commit message to the affected Put node when the relevant Gets complete (that is, when it hears back from the Put node tasked with issuing the Gets).

A simplified incarnation of the version table is pushed out to the middleware layer periodically; the simplified version table includes only a version number for each node (the maximum of all versions hosted on that node). The period of updates is a value of interest. Frequent pushes make for less stale data, while batching could conceivably save network bandwidth. Note that the middleware layer is capable of issuing Gets as well; thus until the switch is able to push the new version table to the middleware (and guarantee that the middleware layer has no further requests in flight) no newer version may be committed.

5. Warehouse

The warehouse layer is responsible for maintaining the contents off all maps in the layout, partitioned into segments. These nodes are not responsible for keeping track of where the maps live, simply keeping track of the data and performing rudimentary computations over this data.

Map data is stored with version numbers. The version number of a data element is part of its key, albeit a part that receives special treatment. Lookups for version number 0 of any element always return 0. Lookups for unknown versions of any stored value will be deferred (in the case of asynchronous gets) or fault (in the case of synchronous gets) until the version becomes known.

This deferring behavior is required in order to deal with out-of-order lookups. It is possible (though unlikely) that a node will receive a request for a value that it has not received the corresponding put command for. Moreover, it's possible (and much more likely) that a get request might occur before the corresponding put command has retrieved all the data it requires. Even after a put request arrives, until all data has been retrieved (permitting the entry to commit), requests for the value will block or fault.

Put requests deserve special mention. While a get request is a simple key-value lookup, a put request is stores the result of an algebraic operation. Thus, a put request can include a blocking read,

5.1. **API/Events.**

ping() Respond with current status (load, memory, etc...)

get(list of Entry) returns map of Entry to float Synchronous set lookup. All entries requested will be returned, mapped to their corresponding values. If any requested entry is unavailable, an exception will be thrown. This can be for one of three reasons: The node does not maintain the corresponding partition (a legitimate bug), the node has not received the corresponding put statement, or the node has not received all of the data required by the corresponding put statement.

fetch(list of Entry, destination, identifier) Asynchronous set lookup. Equivalent to get(), with the following changes: Rather than returning a map, the map will be pushed to the provided destination via the pushget() method. Second, fetch does not fault if the node is aware of the partition that corresponds to the requested entry. If the partition is present on the node, but the specific version is not available (whether due to lack of put command or data), the response is deferred until the data becomes available.

pushget(map of Entry to value, identifier) Asynchronous set lookup response. After a fetch() command completes having obtained all required data, the specified destination will receive a pushget containing the response. The integer identifier provided as part of the fetch statement is passed through to the pushget.

put(version, template, target, list of float or Entry) Part of the process of putting a new value into the map. The target entry is updated to the specified version by means of an algebraic expression. Because all operations in the system will be more or less predefined (as they are toasted), we can save a number of bytes by not shipping around the entire algebraic equation (it also makes passing around algebra simpler, since Thrift doesn't allow recursive structures). A list of template formulae are generated by the compiler. The templates include parameters; The put method includes a set of parameters: either precise values or Entries.

Put first looks up the desired template and obtains a copy of it parametrized by the list passed to put. A new record is created and inserted into the map (appending to the list of any existing values). Put also maintains a list of pending puts.

Any parameters that can be obtained locally are obtained immediately (if they are available). Put does not trigger any remote lookups by itself. A put statement with Entries in its parameter list REQUIRES that the corresponding fetch statements have also been issued. This is desirable behavior, as it minimizes the number of hops required to complete the Put. It does however, make failure handling more challenging.

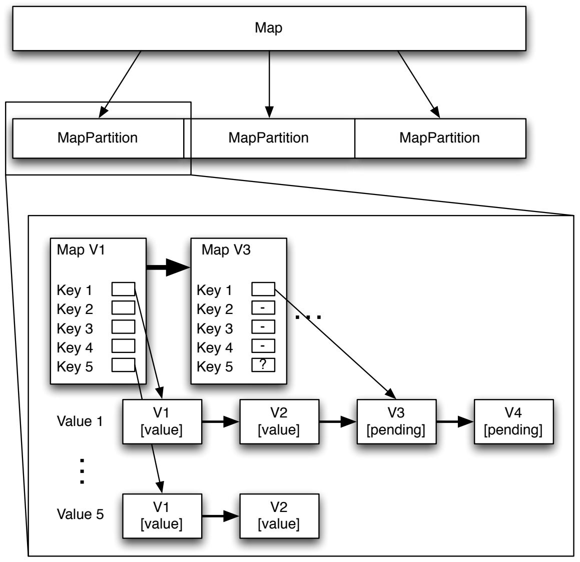

Figure 1. The Map Datastructure

When the put completes, any fetches blocked on the put will fire. Additionally, any local puts depending on the value (ie, a put for the next version of the same map entry) are allowed to complete.

5.2. **Datastructures.**

Map A linked list of maps of Entries to linked lists of CommitRecords. See Figure 1