Streamlining Data Warehousing through Compilation

1. INTRODUCTION

Consider a standard corporate datacube infrastructure. At one end, lie one or more OLTP databases. These databases manage corporate data in a standard relational infrastructure that permit easy updates to minor components. Periodically, a large query extracts the relational contents of these databases for conversion into a datacube for use by corporate analysts.

Periodic scraping is inefficient. Not only does each run duplicate effort expended by prior runs, the necessity of batching these updates ensures that analysts are working with stale versions of the data. To date, work on incremental datacube maintenance relies on discarding data and potential dimensions of analysis to achieve sufficient performance characteristics.

In this paper, we present SpreadDB[1], a system that constructs and maintains datacubes in realtime. SpreadDB applies set of novel compilation and optimization techniques to datacube construction queries of arbitrary complexity. The result is not only a highly optimized execution plan for translating relational database updates into cube deltas, but a *distribution plan* that maximizes communication efficiency within the warehouse.

In developing SpreadDB, we also make several observations about current corporate datacube infrastructures:

1. Disk-based corporate datacubes are stored on vastly more drives than required in order to minimize lookup latency.
2. RAM has become cheap, plentiful, and low-latency.
3. The process of constructing datacubes is very amenable to parallelization.

Rather than maintaining its datacube on an array of disks, SpreadDB constructs the entire datacube in-memory across a network of machines. The minimal functionality required from each node in this network not only minimizes the cost of those nodes, but allows them to be constructed without the use of error-prone, heat-sensitive disk drives.

[1]The name "SpreadDB" has been anonymized for double-blind reviewing

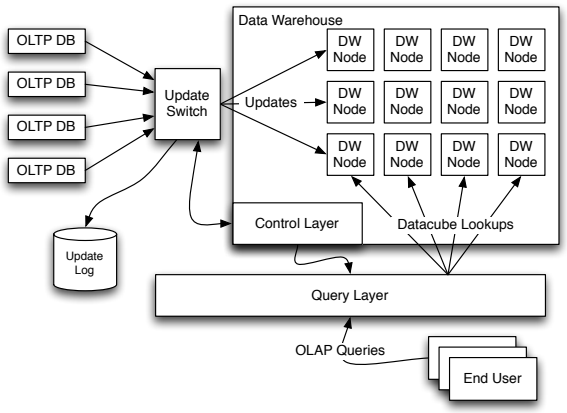

Figure 1: SperadDB's architecture.

2. ARCHITECTURE

SpreadDB consists of three components: a runtime, a *Query Layer*, and a compiler. A diagram of this architecture is presented in Figure 1. The warehousing runtime accepts relation updates at a coordinator node called the *Switch*, and distributes their data throughout an array of data warehouse storage nodes, or *DW Nodes*. Though this paper focuses on a runtime with a centralized switch, we discuss a distributed switch implementation in Section 4.

Adjacent to the runtime is the *Query Layer*, a component that acts as an intermediary between the end user and the dw nodes. The query layer accepts roll-up and drill-down queries, translates them into the corresponding set of data warehouse lookups, and executes those queries on the warehouse.

SpreadDB's final component is a compiler that guides the behavior of the other three components. Given an arbitrary SQL query, the SpreadDB compiler produces a set of update rules, or *Triggers*. Each trigger is invoked at the Switch when one of the query's input tables is modified, and begins the process of applying the appropriate changes within the warehouse.

Changes applied by a trigger are a form of delta-encoding. SpreadDB maintains a set of *Maps* on the DW Nodes to

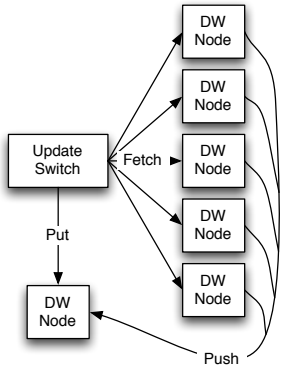

Figure 2: Information flow during one map update.

minimize the work required to process any individual update to an input table. Every trigger incrementally updates the appropriate map, potentially using data from other maps. The compilation and construction of triggers is discussed in further detail in Section 3. In short, every trigger consists of one or more *Map Update Rules* that take the following form:

$$+R(Params) \ : \ Map[Key, Key, \ldots] \mathrel{+}= Expression$$

Here, the Trigger is fired when the tuple ($Params$) is inserted into Table R. When this happens, the target Map is modified by the corresponding algebraic $Expression$ of constants, references to other maps, and variables bound to the input parameters. Map references are indexed by one or more Keys, which may be constants or variables, but not references to other maps. Thus, the total latency for expression evaluation on a quiescent system is limited to at most two network hops.

Finally, some Update Rules may modify an entire cross section of a map. Such updates are expressed via unbound variables in the update rule, that is, variables not bound to one of the input parameters. These loop parameters occur both as a target key and within the update expression. Update rules behave as if they were a set of parallel updates; every update in the set corresponds to one valuation of all loop variables selected from their contextually identified domains.

2.1 Anatomy of an Update

A map update begins at the Switch when an input table is modified. Each input table is associated with one or more triggers, each requiring a write to some range of map values and zero or more reads from different maps. For each trigger, the Switch identifies and sends a PUT message to each DW Node managing a map being written to. Additionally, the Switch identifies all DW Nodes managing maps that will be read from and sends a FETCH message to them, requesting the desired map values. The nodes receiving a FETCH request perform the appropriate reads and send their responses to the PUT nodes in a PUSH message. Finally, upon receipt of all necessary PUSH messages, the PUT nodes compute the update expression and modify the affected maps. An illustration of this process is presented in Figure 2.

1: $Msgs \leftarrow \emptyset$
2: **for all** Update $U \in$ Trigger **do**
3: Region $Reg = \pi_{U.target.loop\_vars}\,(index(U.target\_map))$
4: **for all** Partition $\{P | P \in U.target\_map \wedge (P \cap Reg \neq \emptyset)\}$ **do**
5: $Reads = \emptyset$
6: **for all** Ref $R \in get\_map\_refs(U.expression)$ **do**
7: $RReg \leftarrow \pi_{R.loop\_vars} P$
8: $RPart \leftarrow \{RP | RP \in R.map \wedge (RP \cap RReg \neq \emptyset)\}$
9: $Reads \leftarrow Reads \cup \{(ReadP.node, R)\}$
10: **end for**
11: $Reads \leftarrow group\_by\_node(Reads)$
12: $Msgs \leftarrow put(U, P, Reads.size)$
13: **for all** (Node N, {Ref R}) $\in Reads$ **do**
14: // PUSH results to $P.node$
15: $Msgs \leftarrow fetch(N, \{R\}, P.node)$
16: **end for**
17: **end for**
18: **end for**

Figure 3: The Switch's trigger dispatch algorithm

2.1.1 Trigger Dispatch

Maps are partitioned along dimensional axes when they grow beyond the capacity of a single node, akin to the partitioning done in grid files[1]. To streamline the dispatch of messages to component nodes, the switch pre-generates a spatial index for each template, similar to the grid file directory. Every entry in the spatial index contains a set of PUT and FETCH messages. When the template is triggered, the tuple's values are used to index into the spatial index and the corresponding messages are parametrized and sent. The algorithm for generating the spatial index is presented in Figure 3.

Looping updates require the Switch to match corresponding read and write partitions. The correspondence is obtained by identifying intersections between partitions of the target map that are affected by the update, and those of each map in the update expression. This is equivalent to a join over components of the spatial index stored at the Switch. Loop-free updates are a special case of this, where only one partition is required from each map. The trigger dispatch algorithm is shown in Figure 3.

2.1.2 Get Collation

FETCH responses, or PUSH messages for an update are sent to the node managing the partition being updated. Having received the number of FETCH messages sent by the switch with the PUT message, the destination node can buffer PUSH messages until all have arrived. At this point, if the update is loop-free, the destination node simply uses the contents of the PUSH messages to evaluate the update expression.

If the update requires a loop, the destination node must do some processing. The node first generates a set of tables, one for each map reference in the update expression. Arriving map values populate tables that correspond to any map reference matching the value's keys, where loop variables act as wildcards. When all values have been received, the destination node computes the natural join of all of the generated tables, effectively producing one update value for every assigned value in the domain of all of the loop vari-

```
1: Given: Update U
2: for all Ref R ∈ U.expression do
3:    Inputs ← Inputs ∪ (R, ∅)
4: end for
5: for all (Ref InR, Value V) ∈ ∪(msg_PUSH) do
6:    for all (R, Table) ∈ Inputs do
7:       if check_match(InR, R) then
8:          Table ← Table ∪ (InR.keys, V)
9:       end if
10:   end for
11: end for
12: for all (K, {V}) ∈ JOIN (Keys, Val) ∈ Inputs do
13:   Target = bind_vars(U.target ← Keys)
14:   apply(Target, bind_vars(U.expression ← {V}))
15: end for
```

Figure 4: The DW Node's collation algorithm

ables. The loop-free update is a special case of this where each generated table contains only one row. The collation algorithm is shown in Figure 4

2.2 Consistency Model

SpreadDB's update delta-encoding uses data already precomputed in the warehouse to perform updates. This dependency requires that all reads see a consistent snapshot of the warehouse at the completion of the previous update. Consistency requires that there be an ordering over all updates. As the clearinghouse for updates, the Switch presents an ideal point for generating this ordering.

Every update is assigned a version number by the switch. The maps (specifically, the partitions) that each update modifies and reads are known. The switch maintains a list of modification times for each partition in the warehouse. With each request to the DW Nodes, the switch attaches a version

first looks up the most recent modification time for each get required by the update, and subsequently updates the modification times for each put required by the update.

When applying gets, each warehouse node first ensures that it has successfully applied all requests prior to the indicated version number. A short history of prior incarnations of each partition are also maintained in case a get arrives after a subsequent put. Each put also includes the version number of the last fully completed update. This allows the target node to discard old incarnations of the partition.

Some trickery will be required to get this table into the middleware. Perhaps the switch can periodically broadcast it over?

3. COMPILATION

3.1 Update Rules

- Translating SQL into Update Rules
- Update DAGs/Data flow graphs
- Exploiting Foreign Keys
- Cascading Map Rejection

3.2 Layout

How do we partition data (the layout)? Where do various bits of code get executed live? What kind of runtime analytics do we need to collect to manage the data layout on the fly?

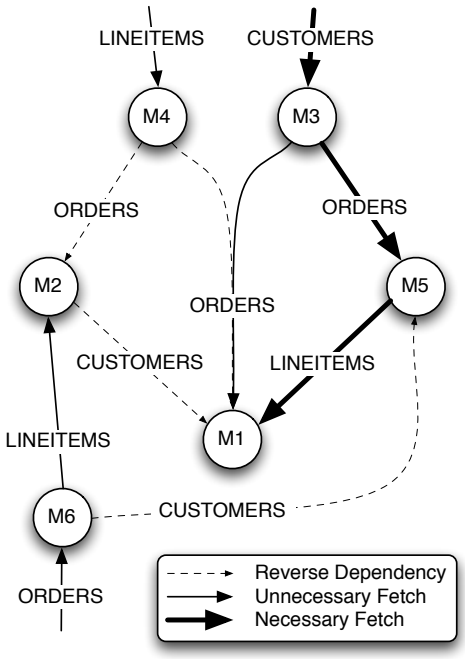

Figure 5: Data-flow graph for the example query. Light and dashed edges can be optimized out, given cascading delete foreign key constraints.

4. DISTRIBUTING THE SWITCH

The role of the switch is to provide a synchronization mechanism for the updates. Specifically, the delta encoding approach used by the update rules requires that all update rules be applied to a consistent snapshot of the maps. The current implementation of SpreadDB performs this synchronization at a single node. However, limiting the switch to only one node creates a scaling bottleneck.

Despite the cloud computing mantra of rejecting consistency, this application requires it. Complex locking protocols have poor scaling performance, so a simpler, lock-free protocol is required. We achieve this goal by introducing the notion of *pipeline scheduling*. Pipeline scheduling exploits the acyclicity of compiler-generated data-flow graphs to allow nodes to correctly interleave and process update rules with only limited network overhead and processing latency.

4.1 Pipeline Scheduling

Loose clock synchronization between nodes is assumed, and allows time to be partitioned into a sequence of numbered ticks, each lasting on the order of seconds. When a set of updates is triggered, a switch node sends tentative put requests to all participating nodes. These nodes respond with their current tick counter, and the switch forwards the maximum returned tick to all nodes.

The updates are considered to have been posted at the maximum returned tick. However processing is deferred for a number of ticks equal to the depth of the map being updated. The result is a data-flow process resembling a parallelized CPU pipeline.

During a given tick, all updates scheduled for processing are evaluated. Once all updates scheduled for the tick have

completed, the node responds to FETCH requests for the tick with PUSH messages. Recall that each edge in the data flow graph represents a FETCH request for a specific update. The difference in depth between the edge's ends is the number of ticks in advance of the PUT that the response is sent.

For example, in Figure 5 without removing any edges, $Depth(M1) = 2$, $Depth(M2) = 1$, and $Depth(M4) = 0$. Updates to map $M1$ scheduled for processing during tick 4 would receive data from map $M2$ during tick 3, and from map $M1$ during tick 2.

4.2 Gossiping Clocks

4.3 Distributed Queries

5. EXPERIMENTS

6. REFERENCES

[1] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):38–71, 1984.