SPREAD: STREAMLINING DATA WAREHOUSES WITH TOAST

OLIVER KENNEDY, YANIF AHMAD, CHRISTOPH KOCH

1. Introduction

Consider a standard corporate datacube infrastructure. At one end, lies 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 runs to extract the relational contents of these databases and convert them into a datacube for use by corporate analysts.

This 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. Prior work on incremental datacube maintenance exists, but relies on discarding data and potential dimensions of analysis to achieve sufficient performance characteristics.

In this paper, we address these concerns with the System for Parellized Rapid Event-Appending to Datacubes. SPREAD is designed based on the following observations:

(1) Corporate datacubes are stored on vastly more drives than required in order to minimize latency.
(2) Memory is cheap, plentiful, and fast.
(3) Construction of datacubes is amenable to parallelization.

2. Toasting Datacubes

How do TOAST datastructures differ from Datacubes? Can we re-create the missing components of a Datacube from TOAST datastructures? Can we evaluate OLAP queries directly on TOAST datastructures.

What kind of code does the compiler need to generate? What is expected of the runtime?

3. Architecture

3.1. **Partitioning.**

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?

3.2. **Consistency.**

Some discussion here already. The consensus for now seems to be:

Every update is assigned a version number by the switch. The maps (specifically, the partitions) that are being modified and read by each update are known. The switch maintains a list of modification times for each partition out in the warehouse. Prior to dispatching a request to the warehouse, it first looks up the most recent modification time

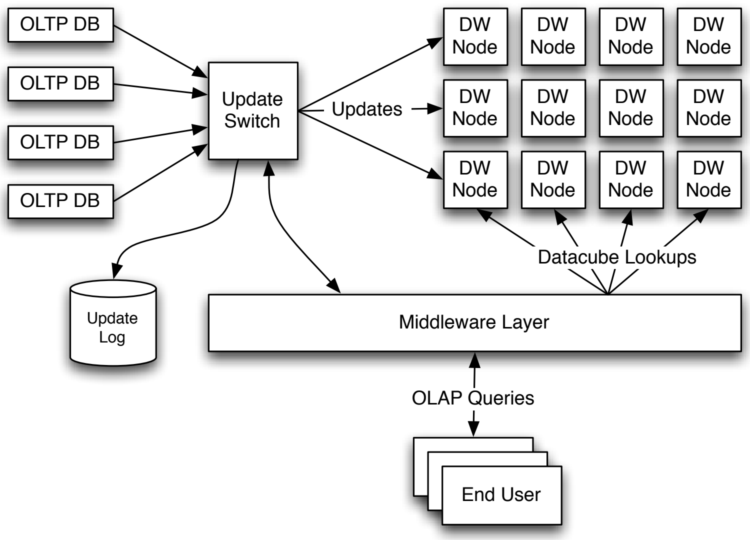

Figure 1. The SPREAD architecture.

- Middleware
  - EvaluateQuery(Query)
  - PushLayout(NodeEvent)
- Warehouse Nodes
  - Get(Map, {Key}, Version)
  - Put(Map, {Key}, Version, SafeVersion)
- Switch
  - DispatchToHandler(TableUpdate)

Figure 2. The SPREAD API

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?