HBase Lattice Quick Start

Dmitriy Lyubimov
dlyubimov@apache.org

October 31, 2011

1 What it is

HBase Lattice is an attempt at BI solution. Namely, it is an attempt at building HBase-based incremental OLAP cube.

I scanned surroundings and noticed at least 2 such attempts which for various reasons (for most part, maturity and staleness) did not fit our purposes.

Like MOLAP solutions, HBase-Lattice copes with aggregate queries by prebuilding certain cuboids in a cube lattice models.

2 Motivations

- **In continuation of "Cassandra is OLTP, HBase is OLAP" mantra**. Except HBase is not really OLAP out of the door. It doesn't support cube models directly. There's no query language to use. There's no predefined way to update a cube. There're no concepts of dimension, hierarchy, measure and fact streams.
- **Big underlying fact stream**. (billions, perhaps trillions of facts to process) which we wont to cope with by parallelizing the compilation with the help of MapReduce.
- **Low query TTLB** (especially on Time Series data). Our goal was to answer queries for any period of time and whatever other slice specifications we can make, in a very short period of time over single hbase table with a very short scan(s). (TTLB <~ 1ms on hbase side, assuming the tablet data is in memory, + whatever network overhead).
- **Next to realtime data availability for querying**. Use of incremental updates to cuboid projections in the lattice. No need to recompile the whole cube for the past 90 days or whatever. New fact data becomes available within single number of minutes after the fact actually happened.

3 Differentiating aspects of HBL vs. MOLAP, ROLAP and cube lattice model

No fact table, no facts kept around. We don't keep individual facts around. Unlike perhaps with some other approaches, there's no level of indirection to query the fact table for actual numbers. All projection data is right there, in a cuboid table. This provides 2 major benefits:

- Low query TTLBs. If we are hitting preaggregated cuboid in the same data, we only need to scan perhaps a handful of items in a scan.
- Don't need the space to keep all original facts. Depending on the definition of dimensions and hierarchies and the nature of incoming fact streams, the space required to keep aggregated projections may require several orders of magnitude less space than the original fact stream.

The tradeoff is obviously in that one cannot query individual fact datum. It is assumed that facts are kept somewhere else outside HBL tables (and they usually are, so no need to mandate data duplication in HBL).

What cuboids are to compile is specified manually. In the interest of keeping things simple, the model specification explicitly lists all cuboids to compile in the cube. Consequently, not all aggregated groups are avaialable for querying. Working out which cuboids to compile is similar to process where DBA tries to figure out which indices to deploy based on use patterns.

New projectsions can be added dynamically to the system. Just specify new projections, deploy the model and the compiler component will start producing new projections right away. (applying new projections over past data retroactively is not easy at this point though. Pretty much the only way to do that is to drop all existing data and re-compile all projections over the entire historical facts again).

Compiler is Pig codegen. The compiler component generates pig script at runtime based on current specification of the model. (see *sample* module for example how to run these scripts).

One of the somewhat stale projects on github used similar approach but instead of using Apache Pig, that project used python streaming MR. But the idea is very similar.

Querying the data. Data querying is available in two ways:

- an API query class (not unlike the declarative api way to construct query objects in Hibernate), and
- a simplistic query language that translates into that api calls to setup a query from reporting tools (again, not unlike HQL support in Hibernate).

4 Quick Howto

4.1 Specifying a model.

Model is specified by compositing a bunch of java classes representing cube, cuboids, hierarchies, dimensions and measures. Of course one may not want to deal writing java code wiring this up. We use YAML for declarative model definition (see file `example.yaml` in the *sample* module of the project).

4.1.1 Supported dimension types

- `HexDimension`. This class supports discrete dimensions that are fixed-length byte arrays. In hbase composite keys they are translated into ASCII Hex representation of such for the sake of simpler readability when using tools like hbase shell. Hence, the name.
  Typically, `HexDimension` is suitable to represent uniformly-distributed hash IDs or otherwise hash-referenced data.
- `SimpleTimeHourHierarchy`. This is a hierarchical dimension to convert `GregorianCalendar` and/or long values representing ms since epoch in fact streams into hierarchical descrete type [ALL].[YEAR-MONTH].[DATE-HOUR]. I.e. the lowest bucket granularity for time series data is 1 hour.

4.1.2 Supported measure types

The only types currently supported for the measures in the fact stream are double and long.

4.2 Incremental cube compilation

TODO

4.3 Query API

TODO

4.3.1 Supported aggregate functions at this time.

- SUM()
- COUNT()

4.4 Querying with a prepared query

See the example for how to prepare and use query. It is recommended to use prepared query repeatedly to save on parsing it into an AST tree. (After all, that's what prepared queries are for).

Approximate current query syntax is (see RFC-822 for the BNF syntax used):

```
'select' select-expr *(',' select-expr)
'from' cube-name [where-clause]
[group-clause]

select-expr = measure-name / aggr-function
[ 'as' alias-name ]

aggregate-function = function-name '('
measure-name ')'

where-clause = 'where' slice-spec *(','
slice-spec)

slice-spec = dimension-name 'in' ('[' /
'(') value / '?'  [ ',' ( value / '?'  )  ]
(']' / ')')

group-clause = 'group by' dimension-name
*(, dimension-name)

value = ( '\" LITERAL '\" ) / LONG / DOUBLE
```

Example:

```
select d1 as dim1, COUNT( m1 ) from Example
where d1 in [?], time in [?,?)  group by d1
```

Where-clause is essentially a slice specification. Hence specification is imposed on a dimension using opened or closed interval syntax. E.g. [1,3) is a so-called half-open interval which includes between values of 1 (including) and 3 (excluding). The limitation of the *where-clause* is that currently one cannot specify more than one slice specification for the same dimension. Semantic result of an attempt to specify multiple slices for the same dimension is currently undefined.

The dimensional hyperplane specification (a plane going thru a given point) is hence equivalent to specifying '`where dimension in [?]`'.

Query limitations.

- There has to be a cuboid specifying all dimensions in a group clause in the leftmost positions.
- Complement scan optimizations for hierarchies is not implemented in this release (only in our prototype).
- There's currently no way to run some useful analytic queries like 'select COUNT(fact), ip group by ip having COUNT(fact) > 10000'.

5 TODOs and FIXMEs

At this point there's no JDBC provider available (we don't use jdbc; we integrate custom datasources directly into our reporting tool. Therefore, creating jdbc support ranked very low on our roadmap, but assuming there's an external interest in this, it should be an easy enhancement, all components are already there for it).

Complement scan optimizatinos for hierarchies are not in yet. (but we prototyped them).

Poor selection of aggregate functions

Poor selection of hierarchy and dimension types

Poor selection of measure types