HBase Lattice Quick Start

Dmitriy Lyubimov
dlyubimov@apache.org

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 some 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 over any period of time and whatever other slice specifications we can make very quickly with a single hbase table and a very limited amount of iterations in a scan. (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 means there's 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, as soon as incremental compiler iteration is complete. Once compiled, the data remains continuously available unless thrown out by HBase during compaction given specified projection TTL parameter.
- **Keep stuff within same ecosystem.** Another motivation is being able to do things within the same resource space of HDFS and HBASE one already invested to. While it is definitely try to use other tools out there with same, if not greater, success, (MongoDB comes to mind), those tools would perhaps require their own distinct environment (resources) and perhaps bulk data transfer and import.

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

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. All projection data is right there, in a cuboid table. This provides 2 major benefits:

- Low query TTLBs. If we are hitting cuboid with precompiled aggregate results, we only need to scan a handful of items per request.

- 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 be compiled 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 a 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).

In either case, a special custom hbase filter is used to allow to skip over the rows we are not really interested in, so the scan iterations are kept going over mostly relevant facts only.

4 Quick Howto

4.1 Specifying a model.

Model is specified by composing a bunch of java classes representing cube, cuboids, hierarchies, dimensions and measures. Instead of writing some java code wiring this composition up, it is also to use a declarative approach for model definition. We use YAML for declarative model definition (see file `example.yaml` in the *sample* module of the project for an example of a declarative model defintion).

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. It accepts java type `byte[]` and its Pig equivalent (in context of compilation).

- `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. The continuous member data type for this dimension (when not expressed with a hierarchy member) is `GregorianCalendar`, or `Long` expressing number of milliseconds since epoch (in context of projection compilation in pig).

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

Cube compilation is done via incremental Pig script dynamically generated by compilter component (see sample module for example of the compilation). With the current approach, compiler doesn't support any input adapters, so it cannot read any standard fact stream sources on its own. Instead, it relies on a fragment of the script that reads input into a predefined Pig relation, to be supplied. This Pig-scripted fragment is called "preambula" and expected to be supplied via Spring Resource specification. * The compiler expects fact stream to be put in a predefined Pig relation (HBL_INPUT by default).

The requirements for HBL_INPUT relation produced by preambula is as follows:

- It must have all defined dimensions. Dimension names used must be the same as in model description. Dimension Pig types depend on the dimension class.
- It must have at least one measure fact (currently, of either long or double type only). The measure is recognized by having the same name as in model description. The scope of measures may be reduced by using exclude/include api on the compiler bean (see sample module for an example). By default, all measures are expected. Using measure scope reduction allows to easily compile in multiple fact streams containing different measures and potentially originating in different sources (for as long as all dimensions can be inferred for each of them).

*Perhaps this only dependency on Spring is bad and it is worth considering getting rid of this abstraction; but developing a project-specific resource abstraction is probably just as equally bad.

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)

measure-name = ID / '?'  ; id rules or
substitution via a parameter

cube-name = ID / '?'  ; id rules or
substitution via a parameter

alias-name = ID / '?'  ; id rules or
substitution via a parameter

function-name = ID / '?'  ; id rules or
substitution via a parameter

dimension-name = ID / '?'  ; id rules or
substitution via a parameter

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.

Aggregating over multidimensional hyperplane (a plane perpendicular to an axis and going thru a specific point on that axis) 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 optimizations for hierarchies are not in yet. (but there's a working prototype).

Poor selection of aggregate functions

Poor selection of hierarchy and dimension types

Poor selection of measure types

Is there a clever way of supporting some of HAVING conditions?