HBase Lattice Quick Start

Dmitriy Lyubimov
dlyubimov@apache.org

Contents

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**. HBase is not really an OLAP service 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 a day to process which we want to cope with by parallelizing the compilation with the help of MapReduce. We also want to scale storage and querying capacity along with it horizontally on demand.
- **Low query TTLB** (especially on Time Series data). Our goal was to answer queries over any period of time and whatever other slice specifications very quickly with a single hbase table access and a very limited amount of iterations in a scan. (TTLB <~ 1ms for the scanning code itself, +whatever overhead of currently configured HBase cache, + 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. Thus, cube data is not an immutable entity. 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 to be able to do things within the same resource space of HDFS and HBASE one has already invested in. While there are definitely other tools out there to try with the same, if not greater, capabilities (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.

MapReduce compiler allows to do distributed aggregation of the data before making updates to hbased-stored cuboids.

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 Overview of the dev workflow

The flow is as follows:

1. Define cube model.
2. Deploy/update model to HBL system table in HBase.
3. Generate compiler script using compiler bean
4. Run incremental compiler cycle on a next portion of a fact stream.
5. Repeat step 4 as many times as needed.
6. At any moment after (2) querying the cube is enabled.

Step 1-2 may be repeated as many times as needed. Step 2 requires script update (step 3) in order for the changes to take effect. The changes committed in step (2) are effective immediately for step 6 activities (almost; hbl client actually caches the models client side after a first query to a cube. So Hbl client interface must be re-created if reused in the client. Perhaps we could implement api to reset the model cache and force the reload).

Two fundamental approaches to implement steps 2-4 can be taken: Command line interface approach and embedded approach.

4.1.1 Command line approach for the workflow

TODO: write command line wrappers for steps 2-4

4.1.2 Embedded approach.

See Example1.java in sample module for the full cycle of steps 2-6. That's actually how we use HBase Lattice: a 100% embedded client application that handles scheduling of incremental compilations, schema updates and code generation.

4.1.3 R approach to the workflow.

Starting with hbl-0.2.0, R package 'hblr' is available to perform all workflow tasks except for running incremental cube compilation (step 3 and 4 in §4.1).

Package "hblr" is using R5 reference classes to describe query and HBL admin classes.

Package "hblr" requires two packages: "rJava" and "ecor". The latter is a non-standard package from ecoadapters project which is dependency of the HBL. If compiled and installed from sources (see *ecor/install.sh* script for an example how to compile and install binary R package from sources) then another package, "roxygen2" and its dependencies are required in order to generate R help files from the source annotations.

Deploying/updating cube model from an R script. The following is an example of deploying/updating Example1 model using R script. `HblAdmin` R5 class is defining HBL admin functions.

```
# deploy/update HBL cube model from file
hblAdmin <- hbl.HblAdmin$new(model.file.name=

    "~/projects/github/hbase-lattice/sample/src/main/resources/example1.yaml")
hblAdmin$deployCube()
```

Running prepared hbl query from an R script. "hblr" package provides ability to run a prepared hbl query in an R script and convert result to a dataframe. `HblQuery` is the R5 class defining prepared HBL query api. The following is an example how to query "example1" compiled data for particular date/time interval and dimension :

```
library(hblr)
q <- hbl.HblQuery$new(c("select dim1, COUNT(impCnt) ",
    "as impCnt from Example1 ",
    "where impressionTime in [?,?) ",
    ",dim1 in [?] ",
    "group by dim1"));
```

```
timerange <- strptime(c("2011/9/1", "2011/11/1"),
    "%Y/%m/%d")
q$setParameter(0,timerange[1])
q$setParameter(1,timerange[2])
q$setParameter(2,"1")
system.time(dframe <- q$execute())
dframe


                                dim1 impCnt
1 00000000000000000000000000000001  14760
```

There is not much more details at this point other than in this example. Further and up-to-date details could be found in R help system for the *hblr* package.

4.2 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 `example1.yaml` in the *sample* module of the project for an example of a declarative model defintion). We employ *SnakeYaml* project for the purposes of constructing java model objects, see how *SnakeYaml* works with classes and constructors for details of what's going on there.

Once model is defined, it needs to be deployed with HblAdmin (see Example1.java in sample module for the example of embedded approach and javadoc for the `HblAdmin` class).

TODO: create command line utility wrapping HblAdmin functionality for model deployments/updates.

4.2.1 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). Now also accepts any numbers and converts them to key using Big Endian conversion. Results returned from queries are always of `byte[]` type.

`HexDimension` is initialized with the value length. The fact stream will accept all values shorter or equal in length to the one specified in model.

- `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).
- `Utf8CharDimension`. This is almost the same as `HexDimension` but expects string type and encodes it in Utf-8 encoding (meaning collation rules of encoded form of Utf8 strings in HBase). It can optionally quietly allow truncation if fact length exceeds defined storage length (see constructor parameters for details).

Dimension types are assumed to be discrete. If dimension is computed over a continuous value in the fact stream (such as time), it has to be mapped into a discrete one perhaps by means of using a hierarchy (e.g. current `SimpleTimeHourHierarchy` copes with continuous nature of time by setting up discrete member values as hour-long buckets and thus implicitly converts every time fact from the fact stream into [YEAR-MONTH].[DATE-HOUR] form. Obviously, we can make granularity finer and finer so that eventually it may go all the way down to a millisecond, and still be able to optimize queries with additive and complement scans.

Handling NULL values for dimensions in the fact stream. At this time, compiler will not accept NULL values for dimensions because representation of NULL values is not currently supported while representing dimension values as a part of a composite hbase key. If it is desirable to accept quietly NULL values as dimensional values in fact stream, such NULLs may be transformed into a reserved non-null value in the compiler's preambula script. Alternatively, some dimension implementations (`HexDimension` and `Utf8CharDimension`) allow specifying such automatic NULL2non-null substitution as a parameter of their constructors.

4.2.2 Supported measure types

The measure type per se is currently not limited to a particular type. It can be any type. However, aggregate functions will only support a particular type. If the fact is not of the type an aggregate function would expect during compilation, it will be considered NULL without any additional noise.

Most of the aggregate functions like SUM(), COUNT() are supporting facts of a numerical type only. In the *yaml* model this is defined as `NumericMeasure`.

Currently, compiler also recognizes pig tuples in a form of (x, t) tuples where x is a numeric fact and $t \in \mathbb{N}_1$ long type representing time of the sample. Such facts are converted to an instance of `IrregularSample` class and subsequently are used with exponentially weighted averaging and rate functions. To define an irregular fact sampling with time datum, `IrregularSampleMeasure` must be used in the model*.

Handling NULL values for measures in the fact stream. Measure values could be NULL in the fact stream. It may affect behavior of certain aggregate functions similar to their definition in Pig and SQL. E.g. if slice did not contain non-NULL facts for a measure, SUM() will produce NULL but COUNT() will produce 0.

4.3 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 Pig relation with a predefined name ("HBL_INPUT" by default). See Preambula script in sample module.

If the script is being run directly in Pig's executable, preambula perhaps should also register hbl jar as additional classpath jar with `register()`. ‡

Preambula is also an opportunity to massage fact stream data a little bit before handing off to compiler. Preambula doesn't suit to host complex preprocessors though because of the codegen'd nature of the final script. If complex data preprocessing is required, it probably better be done by the logging application or a separate preprocessing MR job.

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

*See example for the usage details.

†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. Besides, it facilitates wiring compiler bean up using *Spring*, which is what we do.

‡ *TODO: create maven build for default hadoop job jar for standalone pig application. Example module builds its own hadoop job jar that includes hbl and the pig and uses PigContext api to communicate with Grunt mechanics which is the way we run it, i.e. 100% embedded way.*

- 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).

TODO: command line utility to run compiler bean to generate the pig script into a file.

4.4 Deploying HBL custom HBase filters

Since custom filters are used for querying, one jar (hbl-0.1.2.jar as of the time of this writing) should be deployed to region servers (perhaps with a rolling restart afterwards). With CDH distribution it turns out it is enought just to drop hbl.jar into $HBASE_HOME/lib folder at the region servers. Only querying part depends on this, incremental compiler does not depend on this.

4.5 Query API

TODO

4.5.1 Supported aggregate functions at this time.

- SUM(). This function returns `Double` in queries.
- COUNT(). This function returns `Long` in queries.
- Exponentially (or, rather, Canny function) weighted average for facts with time-based sampling (x, t) as a custom function in the model. This function returns instance of `OnlineCannyAvgSummarizer` containing the weighted sum state. It additionally can be used to derive a biased binomial estimate on a slice[§].
- Exponentially (or, rather, Canny function) weighted rate for facts with time-based sampling (count, t) as a custom function (see Example and doc)

[§]see metrics doc and example.

- SUM_SQ() sum of squares
- AVG()
- SD() standard deviation (todo: add sample deviation)
- VAR() standard variance (todo: add sample variance)
- MIN()
- MAX()

Api allows to add new custom functions easily, it's just all we needed at the moment.

4.6 Querying with a prepared query

TODO: write a command line shell (perhaps akin to HBase shell) to enable ad-hoc query runs.

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 semantics. 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 [?]`' (*degenerate* dimension interval).

Aggregate functions may return `NULL` if a measure group had been empty (or consisted only of `NULL` measure values). This semantics is consistent with `SUM()` and some other aggregate functions semantics in SQL and Pig. As a corner case, a measure group might have been empty if reduced measure scope was applied during compilation. Reduced scope fact stream basically is equivalent to a full fact stream having all facts for the excluded measures as `NULL`.

Query limitations.

- There has to be a cuboid specifying all dimensions in a group clause in the leftmost positions. Hence, plan optimizer may complain if certain grouping is not possible due to lack of suitable cuboid.
- Complement scan optimizations for hierarchies are 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'.
- One has to select at least one measure aggregate in the query. Technically, there should be no reason why not support a request for dimension members satisfying *where-clause* conditions only, but the way it is currently designed, it need a least one measure to sum up in an aggregate (even if one doesn't use it).
- Unlike with MDX, there's no *optimized* way to query a dimension or hierarchy membership. Since the system is pretty dynamic, new members might appear at any time and at this point we don't keep track of distinct list of them. *It is possible to query members in a particular slice though, including the total cube,* but that would be a full table scan over shortest cuboid still.

4.7 Complement and additive scan query optimizations.

Scanning a point slice of either dimension or hierarchy is trivial.

Scanning a range of slices of a dimension is trivial as well (assuming that's the last dimension in cuboid spec).

Scanning a range over a hierarchy is less trivial. Hierarchy must support notion of `[ALL]` member aggregates to be able to produce batch. Additionally, hierarchy needs to support optimizing for complement vs. union scans. (time hierarchies come to mind as a particularly good example of benefiting from complement scans).

Here I'll develop a very simple bit of theory behind additive and complement scans.

Definition - additive scan-capable aggregate functions. Suppose we have a bunch of metrics (facts) $\mathbf{M} = \{m_1, m_2, ..., m_n\}$. We also consider and aggregate function defined over a fact set, $\text{aggr}(\mathbf{M})$, which returns a single variable. If for any two disjoint subsets $\mathbf{M}_1$ and $\mathbf{M}_2$:$\mathbf{M}_1 \cap \mathbf{M}_2 = \emptyset$ also satisfying $\mathbf{M}_1 \cup \mathbf{M}_2 = \mathbf{M}$ exists a function $\textbf{add}(r_1, r_2)$ such that

$$\text{aggr}(\mathbf{M}) = \text{add}(\text{aggr}(\mathbf{M}_1), \text{aggr}(\mathbf{M}_2)),$$

then we call function $\text{aggr}(\cdot)$ *additive scan-capable.*

Definition - complement scan-capable aggregate functions. Similarly, every request can be devised into summing scan over metrict fact set $\mathbf{M}_1$ and a complement scan over another dataset $\mathbf{M}_2 : \mathbf{M}_2 \subseteq \mathbf{M}_1$.

Suppose there's an existing aggregating function over metric set $\text{aggr}(\cdot)$ If there exists a function of two variables $\textbf{complement}(r_1, r_2)$ such that for any two fact sets $\mathbf{M}_1$ and $\mathbf{M}_2$ satisfying $\mathbf{M}_2 \subseteq \mathbf{M}_1$ the following is true

$$\begin{aligned} r &= \text{aggr}(\mathbf{M}_1 \backslash \mathbf{M}_2) \\ &= \textbf{complement}(\text{aggr}(\mathbf{M}_1), \text{aggr}(\mathbf{M}_2)), \end{aligned}$$

then we say that $\text{aggr}(\cdot)$ is *complement scan-capable.*

Obviously, sum(), count() and avg() could be represented in a way that makes them complement scan-capable.

Complement optimization specifically comes in light for time series scans that involve timezone corrections, e.g. timezone correction for a month hierarchy with additional complement scan on an hour hierarchy.

Hence, it follows that we should model a hierarchy in a way so that its implementation would be able to opimize using complement scans. We will require scan additivity of all aggregate functions, but it seems that we cannot request complement scan capability of any given aggregate function. Hence, future complement scan optmization should interrogate functions as to whether complement scan optimization is possible.

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).

Access to model elements is rudimental. Model is exchanged as yaml-serialized string within compiler, which may overload pig communication to backend wastefully if model becomes sizeable (thousands of measures or dimensions). System tables containing model is rudimental as well. We seem to employ several dozens of measures in production without noticeable problems, but obviously this is a somewhat severe limitation for a "big data" system.

Poor selection of aggregate functions. Modelling for aggregation functions and supported member types needs more thought, it is not flexible enough right now.

Poor selection of hierarchy and dimension types. Add more fine grained time hierarchy. Queries don't support specifying hierarchy members in their hierarchical inline syntas as in [ALL][2011][JAN] , but only as a continuous value. Hierarchical members can be constructed and passed in as parameters though.

Crosstab output is not formatted as tab (although equivalent data can be returned as adjacency list).

Poor selection of measure types. Currently, we are limited to numeric facts in fact stream only.

Support for explicitly unbounded intervals in queries. (I think i provisioned some code for this in optimizer and custom hbase filter, but client doesn't support these constructions per se).

Is there a clever way of supporting some of HAVING conditions without running a full table scan?

Parallel querying with region server-side preprocessors?

Support for stringified dimensions. probably needs back and forth conversion from string to hashified representation. Also, probably functionality to enumerate all distinct members of such dimensions. More ideas how to represent? Can as well represent as a fixed size type, then no hash tricks required.

Pivoting UI: how to enable them? an MDX minimum dialect (which is an effort similar to building a Mondrian or whatever tranlsator)? what minimum subset of MDX is needed for such UI? embed our own dialect into jpivot?