HBase Lattice Quick Start

Dmitriy Lyubimov
dlyubimov at apache dot org

Contents

1 What it is

HBase Lattice is an attempt at a big data BI solution. Namely, it is an attempt at implementation of HBase-based incremental OLAP-ish cubes.

Like some MOLAP and ROLAP solutions, HBase-Lattice copes with aggregate queries by prebuilding certain cuboids in a cube lattice model. Hence, the name.

Support of OLAP features (in MDX sense) is fairly limited, but being a big data solution, the project does not limit itself just to traditional OLAP feature set either. E.g. there's a way to run and consume HBL query as a locality-aware distributed MapReduce job, thus enabling fast export/processing of the cube data. Similarly, cube update is also a distributed job. Bottom line, I think it is good to be fairly open-minded in scope definition within a big data environment in the sense that some things in a standard are difficult to achieve and can be thrown away while some "nonstandard" features may become quite powerful enablers in the world of Big Data. I think this is approximately the same reason why the Hive project doesn't follow the exact set of SQL standard either.

Querying data is available as an API, or HBL query dialect and at the moment such clients are available for Java, MapReduce and R environments.

Notations.

Tentative or experimental material
Comment
~~Tentative deletion~~

2 Motivation

- **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 Notable differentiating facts

Compiler is a Pig codegen. The compiler component generates a Pig script at runtime based on current specification of the model.

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

It is possible to script out some additional distributed processing before (or in the same time as) passing the data to compilation.

Incremental data addition to the cube (aka "compilation"). Low latency for cube update cycles. (as often as compiler Pig job can be complete).

Custom HBase filtering for querying. A special custom hbase filter is used to allow to skip over the rows we are not really interested in during composite key scan, so the scan iterations keep going over mostly relevant facts only and can skip over significant portion of the data between first and last scan rows.

Four different ways to query the cube data. Querying is available via:

- Declarative Java client
- HBL query language dialect
- MapReduce distributed query (input format) for batch consumption
- R package for running hbl queries in R.

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. Optimizer selects most optimal (from its point of view) aggregation to use for a query automatically. Aggregate queries can use a range cuboids but the best performance is achieved whenever grouping fits the declared cuboid aggregating dimensions.

API to define new dimensions and aggregate functions. Addition of custom dimension or measure type, or aggregate functions, is fairly easy and straightforward.

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.

Note on how to run Example1.java: Compile project with maven so that example job file is built (it will be used by Hadoop at the backend). Then open example1.java in Eclipse and choose "Run As" → "Java application". For out-of-Eclipse experience, you need to run it from the "sample" maven module directory so the hadoop job file is found.

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 the dependency of the HBL. If compiled and installed from sources (see *hblr/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 :

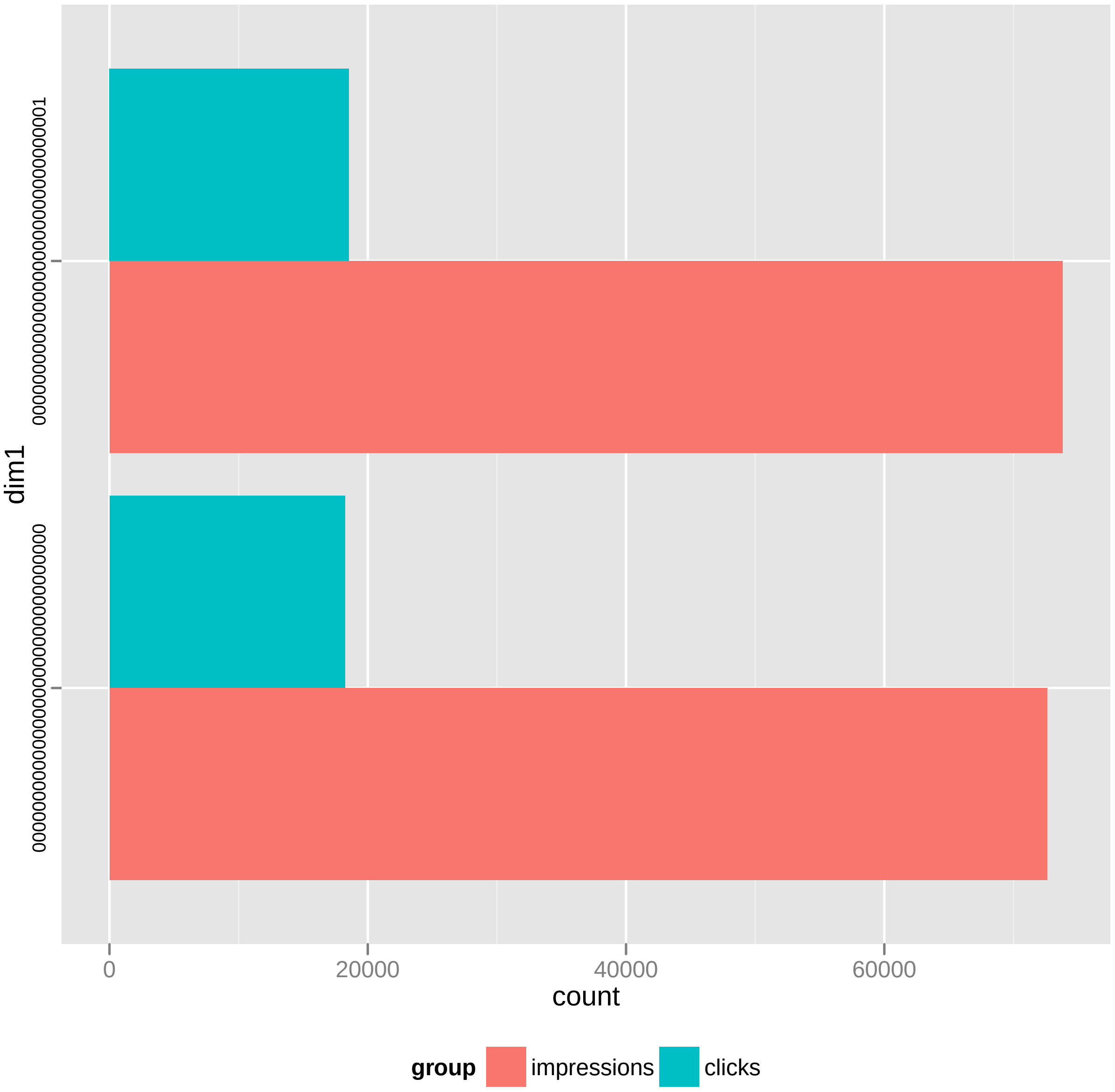
```
library(hblr)
q <- hbl.HblQuery$new(c("select dim1, COUNT(impCnt) ", "as impCnt from Example1 ",
    "where impressionTime in [?,?),", "dim1 in [?] ", "group by dim1"))
timerange <- strptime(c("2011/9/1", "2011/11/1"), "%Y/%m/%d")
q$setParameter(0, timerange[1])
q$setParameter(1, timerange[2])
q$setParameter(2, "1")
system.time(dframe <- q$execute())

##    user  system elapsed
##   1.412   0.044   0.755

dframe

##                                dim1 impCnt
## 1 00000000000000000000000000000001  14760
```

Note the use of half-open slice specification for time slices. This is a recommended way to slice time series data. Since time dimension is not really continuous but is transformed into a time hierarchy with lowest granularity of 1 hour, the slice spec of [“2011/9/1”, “2011/11/1”] really means range from “2011/9/1 00:00:00 UTC” to “2011/11/1 01:00:00 UTC” i.e. includes 1 more hour than the obvious intent here. To fix the situation, a half-open slice spec “[?,?)” is used.

[*] as of hbl-0.2.0-SNAPSHOT, ecoadapters-0.4.0-SNAPSHOT

Another example on the test data(lifetime):

```
library(ggplot2)

dframe <- hbl.HblQuery$new(c("select dim1, SUM(impCnt) as impCnt, ", "SUM(click) as clickCnt ",
    "from Example1 group by dim1"))$execute()
dframe

##                                dim1 impCnt clickCnt
## 1 00000000000000000000000000000000  72600    18235
## 2 00000000000000000000000000000001  73800    18551



# rearrange
dframe <- rbind(data.frame(dim1 = dframe$dim1, count = dframe$impCnt, group = "impressions"),
    data.frame(dim1 = dframe$dim1, count = dframe$clickCnt, group = "clicks"))

# flipped barplot with dodge by event category
ggplot(dframe, aes(x = dim1, y = count, fill = group)) + geom_bar(stat = "identity",
    position = "dodge") + coord_flip() + opts(axis.text.y = theme_text(angle = 90,
    size = 8), legend.position = "bottom")
```



Another test in the R sample code produces time series histogram over entire range of simulated data (which in Example1 simulates only few days in each month, progressively simulating increased impression log entries day-to-day) spanning several months worth of impression counts (figure 1)

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

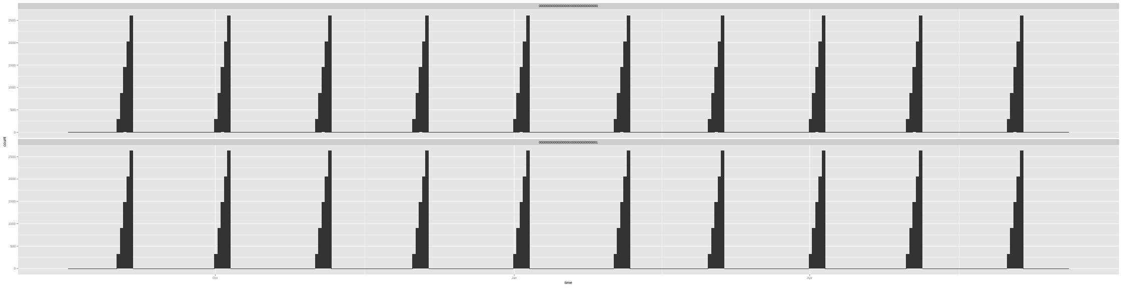


Figure 1: R histogram of impression counts in simulated logs spanning several months worth of data simulated by *Example1.java*.
The binning queries time and plotting time will not depend on actual count of impressions happened during that period since the cube already does data weighting for us.

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 Pig numerical types 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. Slice parameters and fact streams can now also have it as strings encoded in hex (equivalent to output of `hex()` function in *MySQL*). The fact stream compiler 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). Care should be taken to ensure uniform distribution of queries for this dimension if it is used first in a cuboid to scale up hbase query volume.

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 preamble 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[†]. `IrregularSampleMeasure` type is useful when aggregate function wants to handle time series without binning limitations assumed by time dimension hierarchies. Still, such function has to adhere to combine and (optionally) complement contracts to be able to merge its state.

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 *preamble* 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 Preamble script in sample module.

[†] 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.

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

Preamble is also an opportunity to massage fact stream data a little bit before handing off to compiler. Preamble 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.

4.3.1 Fact stream Pig schema requirements

The requirements for HBL_INPUT relation must have a *Pig schema* and such schema must satisfy the following:

- 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).
- Fact stream items should have Pig types supported by their respective dimension and measure types.

TODO: command line utility to run compiler bean to generate the pig script into a file. Perhaps same for R hblr package.

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 Querying

Within the Java world, there are 3 different ways to query HBL:

- `AggregateQuery` which is a declarative way to add slicing and grouping predicates (not unlike Hibernate's Query).
- `PreparedAggregateQuery:` the HBL query language to add slicing and grouping predicates, as well as inline application of aggregate functions. This is quite similar to JDBC's PreparedQuery (and perhaps there's not much rationale why JDBC could not be eventually used except JDBC is much more demanding in terms of metadata).

[§] *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.*

- `HblInputFormat` which is a way to run HBL query in a locality-sensitive distributed way as well as continue working on HBL results in a MapReduce environment.

There’s also an R package available to run HBL Queries and transform the results into an R data frames, which is basically a wrapper around `PreparedAggregateQuery` API along with some data transfer mechanism from Java to R. This is described in brief detail in §4.1.3.

4.5.1 Declarative predicates (∝`AggregateQuery`)

See Example1.java for examples of uses of `AggregateQuery.` Java query api is used to specify slicinig and grouping predicates (not unlike in Hibernate’s Query API).

4.5.2 Querying with a prepared query (∝`PreparedAggregateQuery`)

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 suitable cuboid present and built (see §6.1.)
- 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.

4.5.3 Running location-sensitive query as a MapReduce job ($\propto$`HblInputFormat`)

One of the most common goals of having multidimensional database is to have an exploratory OLAP capability. However, sometimes it is useful to run a bulk query and dump certain projection of the cube (or work on it) as a batch. For example, to dump average performance metrics for all accounts in the system for the last month. Such query may produce a significant amount of work both on HBase-Lattice side and client side that works with the results and thus may benefit enormously from a locality-sensitive worker distribution. Since HBase-Lattice queries consist of HBase scans, they can be defined as a MapReduce job that tries to optimize scan splits so that results of multidimensional query are compiled from HBase region data located on the same node as a MapReduce worker task.

HBase-Lattice project has now support for that in a form of `HblInputFormat`.[¶] See `MRExample1Query` for an example how to run a simple distributed HBL prepared query.[‖] The total result of the query is equivalent tpo concatenating results of all map tasks. Extend `HblMapper` abstract mapper for the mapper task.

[¶] Pig's *LoadFunc* work is still underway at this moment.

[‖] right now limitation for prepared parameters is that they are serialized as string values only thru MR configuration so substitution of parameters is limited to cases where string values are supported by the substitution use case. In particular, I think the date/hour hierarchy currently may be somewhat a problem.

4.5.4 Querying HBL from R

This has been outlined as a part of §4.1.3.

4.5.5 Supported standard aggregate functions

- SUM(). This function returns `Double` in queries. As per conventions in Pig and SQL, this function will return NULL for empty groups (groups with no single fact encountered). Formally: $\mathrm{sum}\left(X\right)=\sum_{i=1}^{N}x_{i}$.
- COUNT(). This function returns `Long` in queries. For empty groups, returns 0L. Formaly: $\mathrm{count}\left(X\right)=N$.
- Exponentially (or, rather, Canny function) weighted average for facts with time-based irregular 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 irregular sampling (count,t) as a custom function (see Example and doc)
- SUM_SQ() sum of squares: $\mathrm{sumsq}\left(X\right)=\sum_{i=1}^{N}x_{i}^{2}$
- AVG() mean: $\mu=\frac{1}{N}\sum_{i=1}^{N}x_{i}$
- SD() standard deviation: $\sigma=\sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(x_{i}-\mu\right)^{2}}$. (todo: add sample deviation with Bessel correction; but see variance for a workaround formula).
- VAR() standard variance: $\mathrm{var}\left(X\right)=\frac{1}{N}\sum_{i=1}^{N}\left(x_{i}-\mu\right)^{2}$. (todo: add sample variance with Bessel correction, but as workaround, Bessel correction can be obtained now by client performing $N\leftarrow\mathrm{count}\left(X\right)$; $\mathrm{var}_{B}\left(X\right)=\frac{N}{N-1}\cdot\mathrm{var}\left(X\right)$.)
- MIN() Returns NULL for empty groups. Formally: $f\left(X\right)=\min_{i\in[1,N]}\left(x_{i}\right)$.
- MAX() Returns NULL for empty groups. Formally: $f\left(X\right)=\max_{i\in[1,N]}\left(x_{i}\right)$

4.5.6 User defined aggregate functions

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

5 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.

** see metrics doc and example.

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 $\mathbf{add}(r_1, r_2)$ 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 $\mathbf{complement}(r_1, r_2)$ 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

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

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.

6 Optimization and known deficiencies to watch for

Most, if not all, of HBL optimization actually revolves around choosing the right set of available cuboids. Choice of cuboids is encompassed by a set of frequently-run (or potentially run) queries. This and other optimization/known deficiency topics are discussed here.

6.1 Suitable cuboid

If a query slices by set of dimensions S and groups by set of dimensions G then a suitable cuboid with dimensions C is such that $S \cup G \in C$ and $G \cup S'_d$ dimensions are in the lead positions of C (in any order). Here, $S_d \subset S$ is subset of slicing dimensions where degenerate slicing interval is used (i.e. closed degenerate interval such as $[a, a]$); and $S'_d \subset S_d$ is any subset of S_d.

For example, if we have a query:

```
select A, COUNT(m1) as m1Count from Cube
where B in [?]  group by A
```

then any of cuboids with dimensions (A,B); (B,A); (A,C,B) or (B,A,C) etc. are suitable to run the query (albeit they will provide different performance; optimizer will try to choose an “optimal” one). Cuboids (B,C,A); (C,A,B) or (C,B,A) will not be “suitable”.

For a similar query:

```
select A, COUNT(m1) as m1Count from Cube
where B in [?], C in [?,?)  group by A
```

examples of “suitable” cuboids are (A,C,B); (A,B,C); (B,A,C); (B,A,D,C) or (A,D,C,B) etc. Examples of non-suitable cuboids are (B,C,A); (C,A,B); (C,B,A) etc.

Currently, such seemingly elaborate criteria of “suitability” stems from a requirement to produce query results “inline” with a set of scan merges. In-memory result accumulation and re-sorting is prohibited. On-disk “spills” of the results are also being avoided. This ensures fastest response and minimal memory requiremnts on the client side. On the other hand, we want to capture all possibly usable cases, which also complicates suitability critera definition a little bit.

6.2 Queries over shorter cuboids normally run faster

If an often-run query groups by A and B, it can be served by any of cuboids with dimensions (A,B), (A,B,C), (A,B,C,D) but will ran fastest with (A,B). Optimizer will make the right choice but only if (A,B) is actually available.

6.3 Slicing vs. grouping

If an often-run query groups by A and B and slices by A, it can be served by both cuboids with dimensions of (A, B,...) and (B, A, ...) but the former will run faster. Currently optimizer does not support discerning between those cases (mostly because it is unusual to have both) so it is recommended to make only one of those choices available.

Example 1.

```
    select A,B, SUM(M) from cube where B in [?] group by A,B
```

is best optimized by a presence of cuboid with dimensions (B,A) (unless B is a non-uniform key like time).

Example 2.

```
    select A,B, SUM(M) from cube where C in [?,?] group by A,B
```

is best optimized by a presence of one of cuboids with dimensions (A,B,C) or (B,A,C).[††] Further considerations include whether A or B more uniformly distributed to cater to better query load distribution among region servers (but not running time per se) – see below.

[††] Actually, in case where slicing by C is using a degenerate interval $[a, a]$ then the best execution plan would perhaps arguably be over a cuboid (C,A,B) or (C,B,A). Pushing dimension slicing over degenerate interval up the cuboid key chain over grouping dimension is now supported as of most recent 0.2.x release tag.

6.4 Uniformity of key distribution

Per HBase tactics, it is important to maintain uniform key distribution (at least within certain range) in order to provide uniform query distribution as well. In hbl, cuboid dimensions are used as cuboid composite keys. Therefore, it is important to have a dimension with more or less uniform member distribution in a certain range in the lead position of a cuboid (typically, a hash or random key value). By the same logic, it is almost never useful to put time related hierarchies into the lead position unless the nature of application provides uniform interest over the entire time period analyzed.

Additional instruments include traditional HBase techniques such as query hot spot detection and elimination (perhaps by inducing a forced split on a cuboid table).

6.5 Long dimension values

Per HBase schema design, it is quite deficient to use big keys and small values because the row and column names are stored (and even communicated over the wire) within each cell. In HB-L, a cell content is a measure with a number of aggregated states of such a measure and the key is the dimensional slice for which the aggregates are precompiled. Hence, this HBase imposed deficiency (or perhaps “feature”, it’s not quite clear to me at this point), translates into deficiencies when long dimensional values are used with HBase-Lattice. To add insult to injury, HBL dimensional values are stored as a zero-padded fixed length type (similarly to char(N) type in databases) to ensure proper composite key parsing and collation rules for efficient custom filtering.

That means that if dimesion type is declared to have a length of 3000, the same amount of bytes is used to store the dimension value in a cuboid record (before compression) and it is also multiplied by number of cells (i.e. measures). Also, hierarchies generate one distinct key per level (e.g. time-hour hierarchy generates one lifetime key, one mohthly key, and one hour-of-day key) so several rows may be generated per just a row of input. This makes the case of long dimension values quite bad in a geometric proportion.

There currently no remedy for this in HBase-Lattice. A workaround is to use hashes instead of values for long documents that are desired to be used as a dimension. The implication of this is also lack of practically useful collate semantics of such values for the purposes of querying by interval other than a degenerate interval ([?]) since the byte-order hbase collation is imposed over hashes rather than actual dimension values. Current thought is to incorporate that workaround into HBL compiler by means of mantaining membership tables and dimension member properties and using so-called ’blob’ dimensions. However, resolving hashes into actual values yanked from a member table would result in one additional *get* per each returned blob key, imposing additional latency overhead. To some degree, this can be somewhat helped by MultiGet facitlity.

6.6 Dimensions with large membership count

If the membership count is large and incremental cycles are short, it may turn out that incremental data chunks aggregate poorly accross cuboids and you may find that compilation generates hbase traffic that is proportional to the number of facts. The sign of this is “slow” running reducers of the first MR job of the compiler cycle.

Adding more reducer tasks in this case will not help much because the bottleneck is hbase traffic i.e. I/O from reducer’s point of view, and it probably will make things worse by creating a possiblity of overloading HBase in some cases. My educated and anecdotally verified guess about # of reducers is about 120% of the total number of region servers in the system, provided lead cuboid keys are uniformly distributed.

The remedy to that is one of

- consider using better aggregating cuboids, or

- increase the period of compilation so the data aggregates perhaps better, or
- scale out HBase itself to be able to handle increased query traffic (combining existing counters with the added group counts), or
- a combination of the above.

It is also possible to split same cube incremental compilation cycle into several cycles with different periods. I.e. it is possible to compile cuboids that aggregate better with more frequency than those that require to accumulate more data (i.e. time) to aggregate better, in several distinct compiler runs using compiler groups. The data will be "eventually" consistent among cuboids, although it would take different time to get results into cube. Another thing to think about in this case is to use coarser time hierarchies for cuboids that compile less often (unless finer granularity is still important even that availability exceeds it by a respectful margin).

7 TODOs and FIXME

7.1 Assorted issues

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?