HBase Metastore Prototype

Introduction

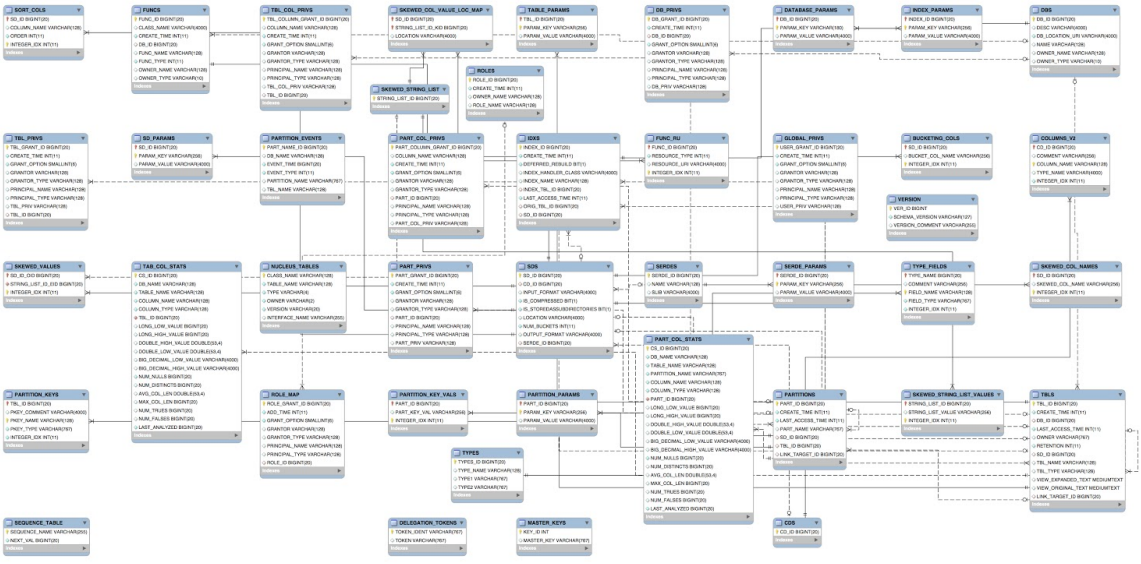
The purpose of this project is to explore replacing Hive's use of DataNucleus (an open source ORM layer) and various RDBMSs to store metadata with HBase.

Why?

Hive is moving towards sub-second response in queries. However, for even mid-sized queries fetching of metadata can take over a second. In a recent test done at Hortonworks we found that fetching metadata (including statistics) for TPC-DS query 27 (which has ~1800 partitions to fetch) took 1.5 seconds.

The current size of the Hive metastore fits easily into most RDBMSs. And given that only clients (not task machines) are allowed to connect to the metastore[1] handling the number of simultaneous connections is not usually an issue either. And RDBMSs are fast. Why do we need HBase?

For several reasons. One, as is well know, there is a significant impedance mismatch between objects and relational stores. Breaking objects into relations involves creating tortured schemas. This is the EER diagram for the Hive 1.0.0 metastore.



It has 43 tables, and something as simple as looking up a partition requires seven or more separate queries (or a seven way join). Obviously we could hand code a better schema, but it would involve denormalizing much of the data. And it would also require us to handle the

[1] However, see below for more on this.

various differences between the five different databases currently supported by Hive. As the ACID project has shown, handling the database differences by hand is painful for developers.

Also, currently issues like scalability, high availability, fault tolerance, and disaster recovery have to be handled by the end user. This is not too bad, as at least HA, FT, and DR solutions are available for most database options, and certainly the commercial ones. But in my experience they are not easy to use.

De-normalized data, high thoughput, scalability, high availability…, these are exactly the areas where big data solutions such as HBase shine.

Requirements and Goals

We would like to significantly reduce the time taken to fetch metadata. For the purposes of this requirement we will seek to be able to fetch all metadata (including statistics) for TPC-DS query 27 in under 200 milliseconds.

The system must be able to support millions of partitions per table, including statistics for those partitions.

There must not be any regression in terms of security or high availability.

We need to open up communicating with the metastore from the task machines. Part of this is to meet the requirements of LLAP, which will need to communicate directly with the metastore in order to check authorization of a user to access data. But it will also open up other options, such as caching index blocks of storage formats (such as ORC's file footers) so that the execution tasks do not need to repeatedly read these blocks.

We need to reduce the memory footprint of the Hive client. Queries that fetch large numbers of partitions require hundreds of megabytes or even gigabytes of memory because the representation of partition information is extremely verbose, with each partition containing several hundred K of data that is identical to every other partition.

It is not our goal to remove the RDBMS option for a metastore. At least for a while, many users will continue to prefer this to using HBase.

Architecture

All metadata operations in Hive are done through the `RawStore` interface. Currently this is implemented by `ObjectStore`, which then communicates with DataNucleus. The proposal here is to provide a new implementation of `RawStore, HBaseStore` which will handle communication with HBase.

HBase Layout

The data will be stored in nine tables. The key for each table is chosen to match how the metastore accesses that table. The names are taken from the existing Hive metastore schema.

Name	Key	Column Families	Comments
DBS	dbname	c	databases
TBLS	dbname:tablename	c,s	tables
PARTITIONS	dbname:tablename:partval:[partval:...]	c,s	
SDS	hash (see below)	c	storage descriptors
FUNCS	dbname:funcname	c	UDFs
ROLES	rolename	c	
VERSION	TBD	c	Hive version information
GLOBAL	TBD	c	privilege information
IDXS	TBD	c	indices

Column family ‘c’ (for catalog) is where the object is serialized and stored in a single column named ‘cat’. Privilege information will most likely be stored in a separate column in the same column family. TBLS and PARTITIONS have a second column family ‘s’ for stats. This column family has a column for each column in the table or partition for which statistics have been calculated.

Of the nine tables in the schema, one does not related directly to a user facing object, SDS. Storage descriptors record how a table or partition is stored and what its schema is. Currently, every partition in a table has its own storage descriptor. This allows Hive to make changes to a table without restating old data. However, it also means that in a select from a table with 2000 partitions, the schema, input format, serde library, etc. are stored 2000 times in the query plan, and in the database. This is ludicrous.

To overcome this, this prototype has been written to factor out common storage descriptors and store them only once. Non-common information (partition location and parameters) are pulled out of the storage descriptor and stored with the table or partition, then the common portions (schema, input/output format, serde information, etc.) are stored in the separate SDS table. The key to this table is an MD5 hash of the value. This hash is stored with the partition

or table. This way when a storage descriptor truly changes (e.g. a column is added to the table, or the storage format is changed) a new hash will be produced and a new row entered in the SDS table. But otherwise creating new partitions will just create references to existing rows in SDS rather than new rows. Similarly, in memory the classes have been modified to not duplicate this information.

Serialization

For the moment serialization is done by hand. We did not use the existing thrift objects because we are relocating certain pieces of data (see the previous section). Doing this by hand is brittle as it will break whenever someone adds a new field to an existing thrift object. It will also make it hard to be backward compatible. We need to use new thrift objects, protocol buffers, or some other technology that will handle the compatibility issues for us gracefully. We also need to find a way to have this break obviously and quickly when new fields are added to the thrift objects to avoid developer errors.

Caching

In the course of planning a single query execution Hive often requests the same data multiple times. Catalog objects cannot be cached across queries without building a cache invalidation mechanism. But they can safely be cached for the duration of a query, as a user should see the same version of an object throughout the query. To this end caches have been built for tables, partitions, and storage descriptors. Early tests suggest that by far the biggest win is in caching the storage descriptors, since they are often the same for all partitions in a query.

Statistics are also accessed frequently. Unlike catalog objects, statistics can be cached across queries because slightly stale statistics will not usually doom the planning for the query. To avoid excessively stale statistics, each entry in the cache will be given a time to live (ttl) after which it will not be used. This type of caching is of interest for HiveServer2, where users could share the statistics cache. It seems better to cache aggregated stats rather than individual partitions as fetching and aggregating stats for all partitions in a query may take a long time. At what level or levels to aggregate and cache them is an open question.