Live long and process (#LLAP)

sershe, sseth, hagleitn, 2014-08-27.

Table of Contents

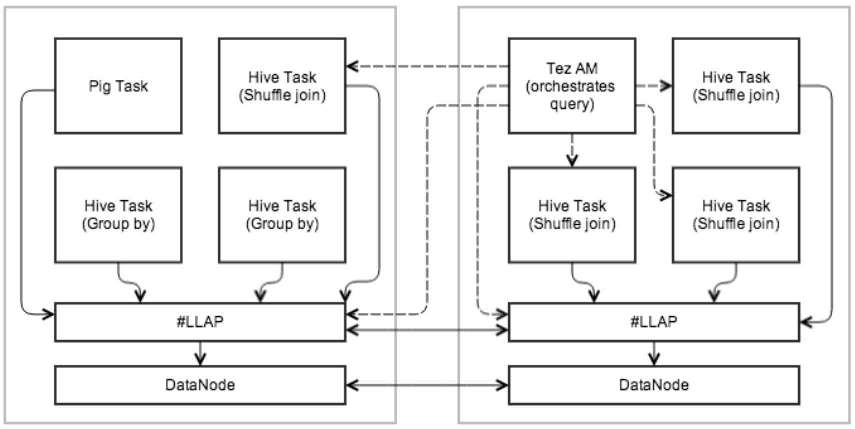
Overview

Hive has become significantly faster thanks to various features and improvements that were built by the community over the past two years. Keeping the momentum, here are some examples of what we think will take us to the next level: asynchronous spindle-aware IO, pre-fetching and caching of column chunks, and multi-threaded JIT-friendly operator pipelines.

In order to achieve this we are proposing a hybrid execution model which consists of a long-lived daemon replacing direct interactions with the HDFS DataNode and a tightly integrated DAG-based framework. Functionality such as caching, pre-fetching, some query processing and access control will move into the daemon. Small/short queries can be largely processed by this daemon directly, while any heavy lifting will be performed in standard YARN containers.

Similar to the DataNode, LLAP daemons can be used by other applications as well, especially if a relational view on the data is preferred over file-centric processing. We're thus planning to open the daemon up through optional APIs (e.g.: InputFormat) that can be leveraged by other data processing frameworks as a building block.

Last, but not least, fine-grained column-level access control -- a key requirement for mainstream adoption of Hive -- fits nicely into this model.



Example of execution with #LLAP. Tez AM orchestrates overall execution. Initial stage of query is pushed into #LLAP, large shuffle is performed in their own containers. Multiple queries and applications can access #LLAP concurrently.

Persistent daemon

To facilitate caching, JIT optimization and to eliminate most of the startup costs, we will run a daemon on the worker nodes on the cluster. The daemon will handle I/O, caching, and query fragment execution.

- **These nodes will be stateless.** Any request to an #LLAP node will contain the data location and metadata. It will process local and remote locations; locality will be the caller's responsibility (YARN).
- **Recovery/resiliency.** Failure and recovery is simplified because any data node can still be used to process any fragment of the input data. The Tez AM can thus simply rerun failed fragments on the cluster.
- **Communication between nodes.** #LLAP nodes will be able to share data (e.g., fetching partitions, broadcasting fragments). This will be realized with the same mechanisms used today in Tez.

Working within existing execution model

#LLAP will work within existing, process-based Hive execution to preserve the scalability and versatility of Hive. It will not replace the existing execution model but enhance it.

- **The daemons are optional.** Hive will continue to work without them and will also be able to bypass them even if they are deployed and operational. Feature parity with regard to language features will be maintained.
- **External orchestration and execution engines.** #LLAP is not an execution engine (like MR or Tez). Overall execution will be scheduled and monitored by existing Hive execution engine such as Tez; transparently over both #LLAP nodes, as well as regular containers. Obviously, #LLAP level of support will depend on each individual execution engine (starting with Tez). MapReduce support is not planned, but other engines may be added later. Other frameworks like Pig will also have the choice of using #LLAP daemons.
- **Partial execution.** The result of the work performed by an #LLAP daemon can either form part of the result of Hive query, or be passed on to external Hive tasks, depending on the query.
- **Resource management.** YARN will remain responsible for the management and allocation of resources. The YARN container delegation model will be used for users to transfer allocated resources to #LLAP. To avoid the limitations of JVM memory settings, we will keep cached data, as well as large buffers for processing (e.g., group by, joins), off-heap. This way, the daemon can use a small amount of memory, and additional resources (i.e., CPU and memory) will be assigned based on workload.

Query fragment execution

For partial execution as described above, #LLAP nodes will execute “query fragments” such as filters, projections, data transformations, partial aggregates, sorting, bucketing, hash joins/semi-joins, etc. Only Hive code and blessed UDFs will be accepted in #LLAP. No code will be localized and executed on the fly. This is done for stability and security reasons.

- **Parallel execution.** The node will allow parallel execution for multiple query fragments from different queries and sessions.
- **Interface.** Users can access #LLAP nodes directly via client API. They will be able to specify relational transformations and read data via record-oriented streams.

I/O

The daemon will off-load I/O and transformation from compressed format to separate threads. The data will be passed on to execution as it becomes ready, so the previous batches can be processed while the next ones are being prepared. The data will be passed to execution in a simple RLE-encoded columnar format that is ready for vectorized processing; this will also be the caching format, and intends to minimize copying between I/O, cache, and execution.

- **Multiple file formats.** I/O and caching depend on some knowledge of the underlying file format (especially if it is to be done efficiently). Therefore, similar to Vectorization work, different file formats will be supported through plugins specific to each format (starting with ORC). Additionally, a generic, less-efficient plugin *may* be added that supports any Hive input format. The plugins have to maintain metadata and transform the raw data to column chunks.
- **Predicates and bloom filters.** SARGs and bloom filters will be pushed down to storage layer, if they are supported.

Caching

The daemon will cache metadata for input files, as well as the data. The metadata and index information can be cached even for data that is not currently cached. Metadata will be stored in process in Java objects; cached data will be stored in the format described in the I/O section, and kept off-heap (see Resource management).

- **Eviction policy.** The eviction policy will be tuned for analytical workloads with frequent (partial) table-scans. Initially, a simple policy like LRFU will be used. The policy will be pluggable.
- **Caching granularity.** Column-chunks will be the unit of data in the cache. This achieves a compromise between low-overhead processing and storage efficiency. The granularity of the chunks depends on particular file format and execution engine (Vectorized Row Batch size, ORC stripe, etc.).

Workload Management

YARN will be used to obtain resources for different workloads. Once resources (CPU, memory, etc) have been obtained from YARN for a specific workload, the execution engine can choose to delegate these resources to #LLAP, or to launch Hive executors in separate processes. Resource enforcement via YARN has the advantage of ensuring that nodes do not get overloaded, either by #LLAP or by other containers. The daemons themselves will be under YARN's control.

Acid

#LLAP will be aware of transactions. The merging of delta files to produce a certain state of the tables will be performed before the data is placed in cache.

Multiple versions are possible and the request will specify which version is to be used. This has the benefit of doing the merge async and only once for cached data, thus avoiding the hit on the operator pipeline.

Security

#LLAP servers are a natural place to enforce access control at a more fine-grained level than “per file”. Since the daemons know which columns and records are processed, policies on these objects can be enforced. This is not intended to replace the current mechanisms, but rather to enhance and open them up to other applications as well.