Workload management

Motivation

This document introduces the basic design of workload management for Hive, esp. as applicable to LLAP. The primary motivation is to allow users to apply flexible cluster policies that are aware of Hive query execution specifics, can apply below process level, and are otherwise not tied to coarse-grained YARN scheduling and commands.
In particular:

- Allocating pool Tez sessions to different users/use cases and moving them between users/use cases.
- Allocating LLAP resources (e.g. executor threads) flexibly to queries belonging to different users/use cases.
- Adding guardrails based on counters to Hive queries that would either kill them a constraint is violated, or change their resource allocation.
- Allowing all of the above policies to be managed by service administrator, switched without requiring cluster restart, etc.

Data model and overview

Resource plans

Resource plan is a fully self-contained workload management configuration. There are several reasons to have configurations that are fully self-contained:

- Atomic switching, ability to switch back to an old configuration.
- Ability to change configuration on schedule, for different business needs.
- Versioning and auditing (later phases?).

The resource plan includes the entire workload management configuration that is described below (the pools, mappings, etc.)
Only one resource plan is active at a time on the cluster.

Changing resource plans. The resource plans have 3 possible states - disabled, enabled, and active on the cluster. One can only modify a disabled resource plan, and only activate an enabled resource plan; this is intended to prevent issues with cron jobs, etc. activating plans in the middle of the change commands. It also creates a convenient model to be able to add automatic versioning, and also synchronization points if we need better concurrent plan editing in the future (currently, we are not going to handle the cases where e.g. 2 admins unwittingly edit the plan in parallel), by redefining what enable/disable commands do.
Any newly created plan would be disabled by default, as it is assumed the admin is about to modify it.
When enabling the resource plan, some validation would be performed that it makes sense (e.g. AM counts and percentages add up, etc.).
Active plan running on the cluster would need to be cloned to be modified. We may add functionality for convenience that clones it behind the scenes if you disable an active plan, but that may be confusing to the users if it only happens for active plans. In future, we may implement atomic semantics for disable/enable commands.

Automatic versioning (phase 2)
Given that we have convenient save points for resource plans (when one is enabled and validated, it's in some consistent usable state), we may introduce automatic version history. It will start out just using automatically generated, disabled copies with dated, versioned names. Users will be able to apply old plans just like they do with regular ones.

The resource plan has:

- A name.
- (Optional) Maximum cluster query parallelism (number of AMs). This can be set by admin to ensure at validation time that the total number of AMs in all the pools does not exceed the reasonable number for the cluster.
- Default pool for queries, if the mapping is not found.

The following commands are available:

- List resource plans.
- Describe a resource plan (various levels – at least two, a full dump and a short description of the pools).
- Create an empty resource plan with a name.
- Create a resource plan cloning another resource plan.
- Drop the resource plan. An active plan cannot be dropped.
- Alter the resource plan.
- Enable/disable the resource plan.
- Apply the resource plan to the cluster (instead of the currently active plan). The plan is validated at apply time (to ensure that AM counts, allocation percentages, etc. add up). See below about the implementation. An option to "enable and apply" in one command (off by default).
- Replace the resource plan. Targets an existing plan, with the new one taking its place in any schedules, etc. present. The old plan can be deleted, or renamed to indicate it's an old version. If the plan being replaced is active, the new plan is also applied to the cluster.

Applying a new resource plan. When a new resource plan is applied to the cluster, it is not guaranteed to be compatible with an old resource plan in any way (e.g. the pools can all be different). However, in most cases we expect minor changes that should not affect the queries.

There are several ways to handle this. Initially, we will determine the changes to the plan that would cause queries to be killed, on pool level. For example, deleting a queue would cause queries in it to be killed; but e.g. changing resource allocations wouldn't. The fields below are annotated to indicate the semantics.

We will also provide a non-default option to drain queries with a timeout. We have found in rolling update scenarios that draining queries is not viable because they can take a long time, and killing queries is generally undesirable, so this will only be there for administrative convenience.

During the time the resource plan is being applied (note that it doesn't include waiting for queries, so it's not a large window), the commands to switch the plan coming from other sources will fail.

When the new resource plan is applied, all the rules (see below) in the new resource plan will apply to the running queries, potentially resulting in additional changes to the queries if the trigger definitions have changed.

Commands for entities inside a resource plan. Since the active plan cannot be modified, every command (e.g. adding a pool, a rule, etc.) must mention the name of the resource plan to be modified. We could alternatively support specifying the plan being modified for the following commands (similar to "use" command for databases), but this may be confusing and error-prone. Explicitly specifying the plan may also help avoid errors in complex scenarios, e.g. schedule-based plans.

Pools

Pools belong to resource plans; as with YARN queues, they can be hierarchical.
Each pool has:

Field	Comment	Diff semantics
Name		Pool queries killed (queue doesn't exist in new plan).
Parent	Optional	Pool queries killed (queue doesn't exist in new plan).
Guaranteed resource allocation	Percentage of the cluster for top level, or a parent pool.	No effect.
Query parallelism	Number of AMs	Decrease: kill all queries (for now; we could also scale down). Increase: no effect.
Sharing policy	Resource allocation policy for multiple queries (e.g. FCFS, or fair)	No effect.
Properties	Extensible, or non-essential, configuration; similar to table properties	No effect (properties are optional, and if they weren't we'd add them as fields),

Note - rules/triggers are stored separately, see below.

Note #2 - because the changes to pools are significantly less frequent than their usage, we will store the full pool path in the database, and recompute these on edits, instead of iterating the parents to get at a hierarchical pool that we want to use at use time.

The total query parallelism of all pools cannot exceed the one for resource plan (if specified); also if HS2 can examine YARN queue capacity, we may validate to ensure the total parallelism can actually be achieved on the cluster.

Parent-child relations and resource allocation

The query parallelisms for pools are absolute numbers, whereas the resource allocations are fractions of a parent pool or the cluster. Therefore:

- The query parallelism numbers for all pools are independent, i.e. if a q1 has 5 AMs, and q1.q2 has 2, the total number of AMs is 7.
- The resource allocation of the parent pool is split between the child pools. If the user desires to run queries in both leaf and branch pools (which we assume is not a typical use case), the capacity must be left for the parent pool (e.g. if q1 has 50% of the cluster, and q1.q2 has 100% of the parent, q1 has no allocation; if q1 has 50% of the cluster, and q1.q2 has 80% of the parent, q1 would still have 20% of itself (or 10% of the cluster). We can optionally disallow queries in branch pools (by default?)
- If the resource allocation of the sibling pools adds up to more than 100%, we have 2 options – normalize it (the default; a warning would be displayed), or error out during validation when the plan is enabled or activated.

Properties may include (all of these are optional for the first cut)

- Whether to pre-launch AMs - some pools, e.g. the ones used for ETL, or "dumping" pools for the rules that move queries, may want to start AMs lazily on demand (or only get AMs from other pools when the rules trigger).
- Priority - by default, all the pools will have the same priority. Optionally, priorities can be used on each node to choose a task to run (or preempt) in case of oversubscription; priorities do not cause task preemption.
- Maximum speculative execution per query (queries can start unlimited speculative, non guaranteed tasks in the duck scheduling design; this can be limited to a fraction of the cluster).
- Maximum resource allocation per query (e.g. if query parallelism is 4, and the pool has 40% of the cluster, each query will get 10% of the cluster; however, as queries start, the latter queries will have a small delay as they get resources currently used by earlier queries; this can be avoided by capping maximum resource allocation somewhat).

- AM count enforcement grace period (AMs can get oversubscribed due to a query being moved, see the Rules section).
- Guaranteed resource enforcement grace period (some queries with guaranteed allocation may be able to afford to wait before killing a speculative task).
- Other ideas.

The following commands are available:

- List all the pools.
- Describe a pool (always produces a full description with rules, etc.)
- Create a pool. A name involving parent pools should be allowed, but all the parent pools must exist (since their allocations, etc. must be specified).
- Alter a pool (change name, allocation, parent, parallelism, properties).
- Drop a pool (for convenience, it should be possible to drop "with redistribute" – redistributing the pool's allocation between siblings and parent). If the pool no longer exists when applying a new plan, the queries in this pool are terminated.
- List the queries in a pool and current allocations and trigger values for each.
- Any trigger actions (see the Rules section below), e.g. killing a query.

Referring to pools in commands

The pools can be referred to by their full name in the tree (e.g. "parent.child.leaf"), or a short name (e.g. "leaf"). However, if multiple pools with the same name exist (e.g. "parent.leaf1" and "parent2.leaf1"), the short name should produce an error (ambiguous pool name).

Query to pool (or non-WM) mappings

The queries will be mapped to pools based on mapping rules. The mapping rules will be based primarily on the user name; it may also be based on other data later.

Mapping to unmanaged (non-WM) execution

There will be a special type of mapping that will cause the query to not use WM and instead execute in a separate (pool or non-pool) session. We will store pool name as null for such mapping. That way, a single HS2 can be used for both WM-managed and unmanaged queries.

Groups

As a next step after user-based mapping, we may add group support based on LDAP/UserGroupInformation. Later, we may integrate with Ranger to get user groups.

Flexible mappings

It's not clear if there's a standard way to supply additional information, e.g. application name, to JDBC. set/getClientInfo API is not supported by Hive driver and may not be

used consistently. Therefore, to add flexibility to the mappings, we will allow the users to supply an argument to the JDBC connection similar to Hive configuration settings to use a particular pool. Note that this (or any other client supplied information like setClientInfo) is not secure and is a convenience API that should be coupled with user-based enforcement; therefore we will check that a mapping exists allowing this user to use this pool.

The mapping has:

- Type – initially, “user” or "group" (see above).
- Entity name – e.g. the user name.
- Full pool name. No short names will be allowed here due to how error prone that would be.
- Priority. In case multiple rules apply, the first one will take precedence; that is mostly relevant for multiple types of rules (e.g. all the queries from “oozie” group go to “lazy” pool, unless sent by user “sershe”, in which case they go to “high-priority” pool). Additionally, this makes it possible to map a user to multiple pools, allowing him to submit queries explicitly to any one of them.

Note that mappings do not have a name; that would be cumbersome given that typically there would be many mappings, also altering a mapping is a rare operation. Rules must be unique based on {type, entity name, pool name}.

Changes in mappings do not affect existing queries (i.e. queries keep running).

The following commands are available:

- List all mappings (optional filters by type and entity name, or pool name).
- Create a mapping.
- Drop a mapping (a type, entity name, and pool name must be specified).
- Alter the mapping priority.

Pool rules (triggers and actions)

Rules can be created to perform any actions an admin can perform on queries, based on triggers.
The following actions are initially available:

- Move a query to a different pool. The typical setup would be to have a low latency pool, and a high latency pool. A rule would move a query from a low latency pool to a high latency pool.
- “Pause” a query (remove all guaranteed allocation from it; speculative execution only). This is another option to deal with slow queries in BI pools without having a separate pool setup.

- Kill a query (no retries).

The following triggers are initially available:

- Total query runtime (elapsed timer at hive level has to be added).
- Total task runtime (slot time; the combined execution time of all the tasks).
- Bytes read from FileSystem-(by all tasks). Scheme of the filesystem has to specified explicitly. HDFS_READ_BYTES, S3A_READ_BYTES. Ignoring scheme will assume local file system.
- Bytes written to a FileSystem (by all tasks). Scheme of the filesystem has to specified explicitly. HDFS_READ_WRITTEN, S3A_READ_WRITTEN. Ignoring scheme will assume local file system.
- Shuffle bytes per query.

The rule has:

- Name.
- Trigger expression (initially, only one value is supported. also will only support >= expressions as most cases just sets a limit).
- An action (may include parameters – e.g. the destination pool name).
- Scope (Query, Session, Global) - optional (initial implementation will only support Query level scope)

There's also a separate mapping of rules to pools (many to many).

Changes in rules do not affect existing queries (i.e. queries keep running). Note that any new rules would still naturally apply to old queries when HS2 checks the queries for compliance, and that may cause queries to be killed; however, that is not a part of the plan update process.

The following commands are available:

- List all rules (optional filter by pool they are attached to).
- Create a rule.
- Alter a rule (by name).
- Attach a rule to a pool.
- Detach a rule from a pool.
- Drop a rule (with an option to detach from all the pools).

Global/Default Triggers (not attached to Pool/Resource plan). These triggers will get applied only for Tez execution engine; they will still be a part of resource plan to ensure atomic edits and application, but they will not be attached to a WM pool. Global/Default triggers will only allow "Kill Query" action.

Moving queries and enforcing query parallelism

Moving an AM into another pool may cause this pool to exceed its query parallelism. We will track the original pool the query comes from. The AM will effectively be "on loan" from that pool. If the destination pool is over capacity, the AM will be killed when the source pool needs another AM and is itself at capacity (subject to an optional grace period).

If multiple rules move the query multiple times, we will still only keep track of the original pool from which the moves have started, since at kill time, it's as if the query was moved from the original to the final pool.

If multiple queries have been moved by a rule to a different pool(s), we will kill them in order of a simple heuristic, e.g. least execution time first, to waste less work; however, at a given decision point, only the queries that actually oversubscribe their current pools can be killed. E.g. if pools q1, q2, q3 have parallelism of 2, 1, and 1 (4 AMs total), and the state is q1 [a, b], q2 [], q3 [c]; a and b are moved: q1 [], q2 [a], q3 [c, b], only q3 is oversubscribed; therefore if 2 more queries are submitted to q1, "b" will be killed regardless of runtime/etc.

Note on schedules

Administrators may want to specify a schedule to switch resource plans (i.e. the cluster-wide configurations, see below) on the cluster at a given time. Initially, we are not going to implement schedules inside the system, the users may rely on cron and similar tools to create them.

However, it is possible that in future we will implement schedules inside HS2.

Other implementation details

Note on YARN queues

WM-managed queries will not use YARN queues. The separation of the cluster will be achieved by creating a single "interactive" YARN queue; all the interactive AMs will run in that queue. That way the sizing of the "interactive" part of the cluster is clear to the user. Inside the interactive YARN queue, the capacity will be managed by Hive using "Hive pools" (see below), formerly known as Hive queues.

Multiple HS2-s

We want to be able to support multiple HS2 in a "load-balanced" configuration; basically, these would be used to allow for higher query parallelism, and fault tolerance. Ideally, workload management should work across these HS2s without depending on their number, and individual HS2s coming and going away.

The coordination applies to the following actions:

- Starting AMs of the shared interactive pool.
- Checking and acting upon the trigger-based rules.
- Maintaining a unified view of the queries in each pool as queries are ready to execute, as they finish, and also as the pools change with resource plan changes.
- Claiming a query "slot" (an AM) for a particular pool from a shared pool; returning the AM after the query completes.
- Redistributing cluster resource allocations for the existing queries based on new queries, finished queries, the new resource plan activation, and cluster configuration changes.

Active-passive option

There will be active-passive HS2 HA that is similar to all the other services (RM, etc.). Most of it is out of the scope of this document. The main thing that would need to be supported for workload management is the transfer of the AM pool, as well as the query details, to the new HS2. Resource plan details are stored in metastore DB, so they are implicitly atomic and durable.

The new HS2 will need to receive heartbeats from the AMs. To transfer heartbeats between HS2s, we will use ZK register all AMs. AMs will periodically renew their registration, the znode acting as heartbeat and also containing the counters, duck count information, etc. that HS2 needs. HS2 will contact AMs directly, as needed, to update duck counts or other settings. This model is also reusable later if we need an active-active option.

LLAP cluster changes

Workload management implementations would need to monitor for LLAP cluster configuration changes. This can be accomplished using the registry. Note that the cluster configuration will not directly inform any high level decisions w.r.t. policies, rules, resource allocation percentages, etc.; therefore, this monitoring needn't affect the high-level data model above. However, it may be necessary for administrators to change things like query parallelism to accommodate major cluster changes.

Managing AMs

The AMs in the interactive YARN queue are fungible between Hive pools, unlike the container AMs w.r.t. their multiple YARN queues. In fact, an AM doesn’t have to know which pool it belongs to; this information for now is only needed on HS2 side. For obvious reasons (YARN queue separation), HS2 will maintain separate AM pools for containers and interactive. HS2 may be able to determine the maximum possible pool size automatically at later stages.

Resource allocations – API

We have considered several implementations here, therefore it should be easy to switch to a different one in future. All of the policy, rule, etc. code will be generally applicable, and the API to the resource management bit should be terse.
In particular, given the current cluster state and the resource plan, HSI should produce a percentage allocation (as well as, optionally, things like priority, speculative execution restrictions, etc. – see pool properties) for every query. This may include new queries that were added to the cluster state, but not started yet. Workload management component can then enforce these allocations on the cluster without being aware of pools, rules, policies, etc.

Resource allocations – implementation

We have considered multiple options for resource allocation, including a global scheduler.
Our initial implementation will use a notion of "guaranteed task" (a "duck") for scheduling; a guaranteed task for each resource (currently executor slots) is created and managed in a central service (HS2); these are given to WM query AMs in proportions based on configured resource plan; AM (via LLAP) makes that number of its most important tasks guaranteed at any given time (transferring the ducks between tasks as they finish or DAG priorities change). Non-guaranteed tasks can still use resources (they are known as "speculative tasks"), however a guaranteed task can always pre-empt a speculative task. This approach can have some anomalies, however local decision making is more simple (and has much less overhead) than global. We may consider global scheduling, or a hybrid solution, in future.

Potential future work

- Scheduling coordination improvements - e.g. try to place on the least busy nodes.
- Global or hybrid scheduling.
- Better resource plan modification and versioning - atomic semantics for disable/enable commands that would work based on clones, but do the clones and checks, as well as versioning, automatically in the background, to allow parallel modification/activation/etc.
- Single inheritance for resource plans, to create multiple similar plans with small config changes without having a lot of duplication; e.g. to be used during different times of day.
- Resource plan specifications in a file (like an existing service xml config file, or w/json).

- Active-active HS2 load balancing; probably via a central service ("master HS2") managing WM, but all HS2 compiling and running queries, but only the master would give out resources, manage AM pool, apply triggers, etc. Distributed WM would be much more complicated than the electable master.