

Impala Transparent Query Retries

[Project Summary](#)

[Problem Statement](#)

[Non-Goals](#)

[Design](#)

[High Level Architecture](#)

[Proposed Design](#)

[Proposed Implementation](#)

[Supportability](#)

Project Summary

Problem Statement

Currently, if the Impala Coordinator or any Executors run into errors during planning, query startup or query execution, Impala will fail the entire query. It would improve user experience to transparently retry the query for some transient, recoverable errors.

Non-Goals

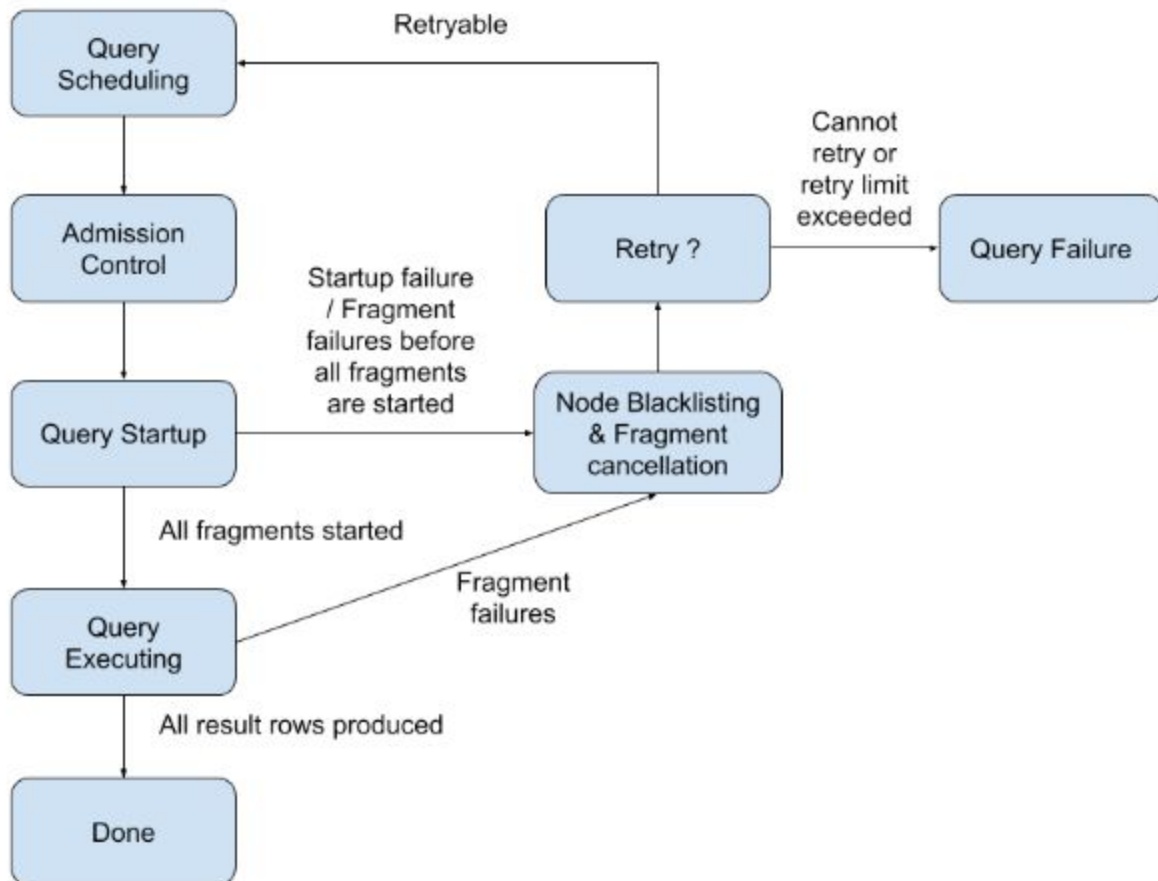
The goal of this feature is **not** to provide fine-grained retries of individual query fragments (e.g. what MapReduce, Spark, LLAP, etc. do). The focus is on retrying the entire query (or all query fragments).

INSERT and DDL queries will not be retried. It is possible retries for the SELECT portion of a INSERT / CTAS query will be retried in a future milestone, but for now that is out of scope.

The first milestone focuses on retrying queries that would otherwise fail due to cluster membership changes. Specifically, node failures that cause changes in the cluster membership (currently the Coordinator cancels all queries running on a node if it detects that the node is no longer part of the cluster) and node blacklisting (the Coordinator blacklists a node because it detects a problem with that node - can't execute RPCs against the node). It is not focused on retrying general errors (e.g. any frontend errors, MemLimitExceeded exceptions, etc.).

Design

High Level Architecture



The above diagram shows the query lifecycle with retries after the planning phase. In particular, an augmentation to the existing model is a stage in which we may consider pushing the query through scheduler and admission control again after hitting a failure. The “Retry?” box needs to decide if the query is retryable (e.g. DML/INSERT queries may not be retryable), the error is recoverable, and whether we have exceeded the maximum number of retries allowed.

Proposed Design

The proposed design is to augment the Coordinator code (ImpalaServer, ClientRequestState, Coordinator, QueryState, etc.) to support automatically retrying a query that fails due to any cluster membership related errors. The Status object will include new information about whether a query is retryable or not, as well as more structured information about what caused the failure (e.g. a RPC to node “x” caused the failure). If the Coordinator ever sees a retryable status it will

update the cluster membership (e.g. add a node to the blacklist), cancel and unregister the running the query, and then launch a retry of the query. The retry will skip fe/ planning and re-use the TExecRequest from the original query. Creation / running of retried queries will be handled by a dedicated threadpool.

The rest of this section outlines the major design points in more detail.

- **Classifying Errors**

- The goal here is to classify certain errors as “retryable” (e.g. cluster membership shows can be retried, a SQL parser error is typically not retryable)
- Modify Status and introduce a concept of Status “categories”, initially the only category will be RETRYABLE
- A Status marked as RETRYABLE should trigger a query retry
- Modify the RPC failure Statuses to be well-formed; e.g. objects that contain relevant information about the error, such as source and destination host + port
 - This is necessary when a query fails because an RPC to a target host fails (because that host has been killed)
 - This extra information is necessary so that the Coordinator node can determine the target host, and add it to the blacklist

- **Node Blacklisting**

- Queries that fail and add a node to the blacklist, should be retried
- The current blacklisting implementation adds a node to the blacklist if an Exec RPC to it fails (presumably this means that something is wrong with the impalad so we shouldn't include in when running queries)
- So if a query fails, and then decides to add a node to the blacklist, the query should be retried, rather than completely failing
- This should be as simple as adding the RETRYABLE flag to the Status returned after a node is added to the blacklist

- **Frontend Retries**

- Only retry backend fragment failures, don't retry any frontend failures
 - Reasoning: The initial milestone is to handle query failures due to changes in cluster membership (blacklisting, node failures, etc.)
- It should actually be straightforward to add in frontend retries later, the fe/ call is relatively isolated “exec_env_->frontend()->GetExecRequest”
- Avoiding re-running the fe/ saves time when re-executing a query
- Re-running the fe/ code is not necessary to handle cluster membership changes

- **Query Cancellation**

- When retrying a query, make sure the original query has been cancelled before re-running the query
 - Reasoning: Logic is simpler and less error prone; we can add optimizations in later that trigger retries more eagerly, if needed

- **Query Lifecycle**

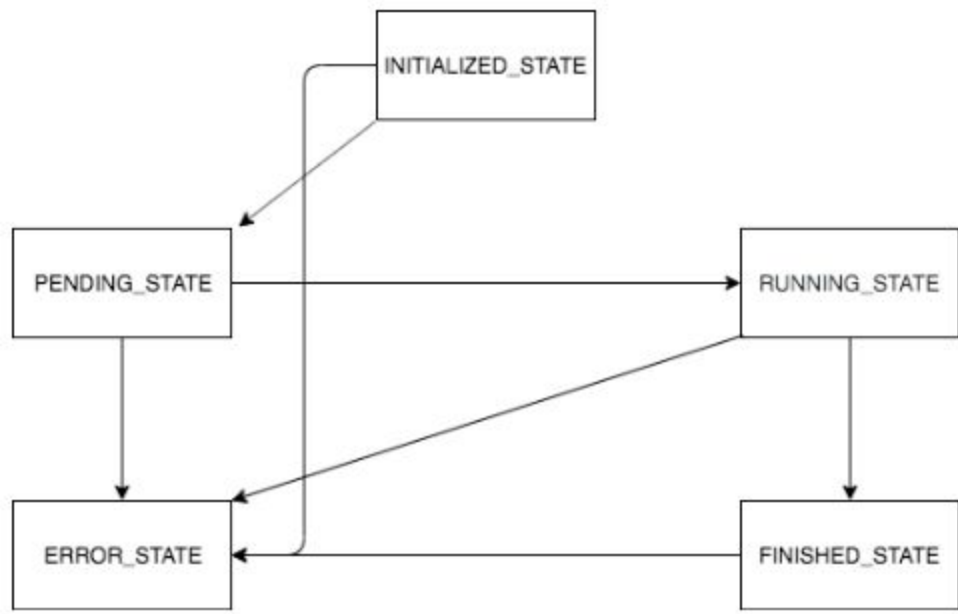
- A retried query should be a completely new query, from the perspective of the ClientRequestState and Coordinator
 - Reasoning: Implementation is much simpler, the ClientRequestState / Coordinator code assumes a query is run exactly once
 - This has some advantages w.r.t to runtime profiles, the runtime profiles of the failed queries will show up normally in the Web UI
- This means that the failed query should be cancelled, closed, unregistered, etc. and the retried query should be a completely new query
 - This means that retried queries have their own query id
- The TExecRequest from the first attempted query can be reused for the retried queries (thus avoiding re-running the fe/ for the query)
- **Runtime Profiles**
 - A new runtime profile is used for each query attempt
 - A retry marker (with a list of previously attempted query ids) will be included in the retried runtime profiles
 - The advantage is that the query profiles of the failed query attempts are easily available
 - Even if a query is retried, it is important to understand why it failed in the first place and exposing the RP of the query profiles help achieve that
 - The profiles are available using the standard mechanism - via the Web UI and the profile log since the failed query has a unique query id that is not shared with the retried query
- **State Management**
 - Two new internal states will be introduced: RETRYING and RETRIED
 - RETRIED is a terminal state, indicating that a query was successfully retried
 - A query can fail with a new state called RETRIED, which indicates that the query failed and was retried
 - The new states will show up in the Runtime Profiles and the Web UI, but won't be queryable by the Beeswax / HS2 APIs (since neither protocol has a concept of retrying queries)
- **Beeswax / HS2 State**
 - The state machine of the Beeswax and HS2 protocols will need to change, specifically, additional state transitions will now be possible
 - For HS2, queries will be able to transition from the PENDING_STATE or RUNNING_STATE state back to the INITIALIZED_STATE state
 - For Beeswax, queries will be able to transition from the COMPILED or RUNNING state back to the CREATED state
 - More details on this are in the implementation section below
- **Result Spooling**
 - If query retries are enabled, calls to PlanRootSink::GetNext will not return until all results have been spooled
 - Some additional work will be required to disable query retries if the BufferedPlanRootSink can not buffer all query results

- This is required so that partial results are not returned to the user; if a user starts reading some rows, and then the query is retried; the result set of the retried query might be different
- A query won't transition to the FINISHED state until all rows are spooled (special handling will be required for queries whose results don't fit in the allocated spool space)
- **Retry Mechanics**
 - A query can be retried a configurable number of times (perhaps by default it is retried twice), and there is a delay between each retry
 - The first retry can be immediate, the second one can be invoked after some delay (potentially configurable as well)
 - One of the retries should allow for enough time for any updates in cluster management to propagate, so that retried queries are not scheduled on dead nodes

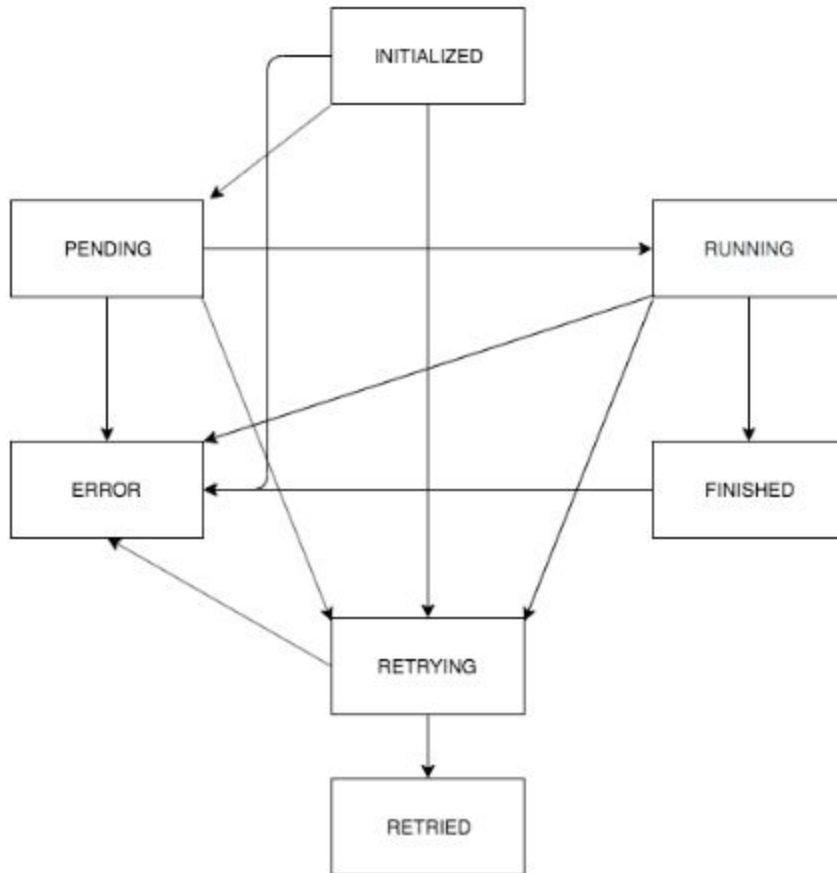
Proposed Implementation

- Most of the implementation work will be around query lifecycle management (specifically state management in the ImpalaServer and ClientRequestState)
- A new ClientRequestState (and thus a new Coordinator) will be created for each new query retry
- The ImpalaServer will call UnregisterQuery on the failed query, and RegisterQuery on the new, retried query
- **Threading:**
 - One challenge is determining what thread drives query retries (see section on Coordinator Implementation at the bottom of this doc)
 - Since most state transitions are driven by the ControlService, it would be best to drive retries via a dedicated threadpool
 - This is similar to how cancellations are scheduled (at least cancellations driven by cluster membership changes) - see ImpalaServer::CancelFromThreadPool
 - The advantage is that the kRPC threads in the ControlService do not have to drive retrying a query
 - We currently do this for some cancellation paths (cancel a query from a ControlService kRPC thread), but there is a JIRA (IMPALA-5119) to change that
- **Concurrency Issues:**
 - Since query retries will be driven by a separate thread, there are a number of concurrency issues that will need to be addressed during the implementation
 - So far, the current ClientRequestState lock_ has been sufficient to solve these issues
- **Query ID Translation:**
 - HS2 / Beeswax APIs:

- One tricky part of the implementation, is how to map an existing QueryHandle / TOperationHandle to a retried query, since clients use an operation handle (QueryHandle for Beeswax and TOperationHandle for HS2) to poll the state of a running operation
- HTTP API:
 - Unlike the Beeswax / HS2 API, no translation of query ids occurs when issuing HTTP request against the Impala server
 - This is necessary so that clients can still view the runtime profiles of both the failed and retried queries via the Web UI
- **State Management:**
 - Currently, ClientRequestState uses HS2's TOperationState for its state machine
 - The use of TOperationState will be replaced with a new state machine that includes the RETRYING and RETRIED states
 - ClientRequestState::UpdateQueryStatus will submit a "RetryWork" object to the retry thread pool if it is given a retryable Status, and then transition to the RETRYING state
 - The ClientRequestState will transition to RETRIED once the new query is registered, and ClientRequestState::Exec and WaitAsync for the new query return
 - If the retry fails, the original query will transition to the ERROR state
 - The existing ClientRequestState state machine is below:



- Unlike the previous state diagrams, this one is specific to the ClientRequestState
- The new ClientRequestState machine would be:



- Queries in the INITIALIZED, PENDING, or RUNNING phase can now transition to the RETRYING phase
- Since an attempt to retry a query can fail, queries can transition from the RETRYING phase to the ERROR phase
- Queries can only transition to the RETRIED phase if they first successfully transitioned to the RETRYING phase
- **Node Blacklisting:**
 - Nodes are currently blacklisted by calling `ClusterMembershipMgr::BlacklistExecutor(TBackendDescriptor)`
 - Currently, the only usage of this method is in `Coordinator::FinishBackendStartup` and it is called if `ControlService::ExecQueryFInstances` returns an error status
 - In order to force this call to the blacklist to trigger a retry, the Status returned by `ExecQueryFInstances` just needs to be marked as retryable
- **Query Retries:**
 - When a query is retried, the following steps need to take place:
 - Fetch the `ClientRequestState` of the failed query (it should be in the RETRYING state)
 - Cancel the query (`ClientRequestState::Cancel`)
 - Prepare the new query:
 - Call `PrepareQueryContext` to create a new query id

- Create a new ClientRequestState
 - Take the TExecRequest from the failed query and use it for the TExecRequest of the new query
 - Register the new query (ImpalaServer::RegisterQuery)
- Launch the new query
 - ClientRequestState::Exec and WaitAsync
- Mark the old query as “RETRIED”
- Unregister the old query (ImpalaServer::UnregisterQuery)

Supportability

- Logging:
 - We could consider printing a few info statements in the client logs when a retry occurs - including a link to the Runtime Profile of the new query
- Runtime Profiles:
 - Retried Query Profiles:
 - Includes a marker indicating that the query was run as a result of retrying a failed query
 - Includes the query ids of the previously failed queries
 - Includes the reasons why the previous queries failed
 - Includes some basic runtime info about the previous queries:
 - How long each failed query took to run
 - Can consider including this in the query timeline as well
 - Failed Query Profiles:
 - The Query Status will be marked as RETRIED to indicate the query failed, but was RETRIED
 - Includes the original reason why the query failed
 - Includes the query id of the new query launched
- Impala Metrics:
 - Query retry rate (the rate at which queries are retried - e.g. 3 queries are retried per second)
 - This can be further divided by retry “type” - e.g. what caused the retry
 - Potential categories would be:
 - Queries retried due to blacklisting
 - Queries retried due to failed RPCs
 - Queries retried due to statestore detection of cluster membership changes
 - This would be similar to error classification metrics (e.g. metrics that track how often queries fail due to a certain error)
- Impala Web UI:
 - It would be nice if users could view all the runtime profiles for all linked retried / failed queries via tabs on the query page

- Age out linked retried / failed queries together - e.g. there shouldn't be a situation where the failed query has been age out of the Web Server, but the retried one hasn't