HIVE-19715 - Consolidated and flexible API for fetching partition metadata from HMS

Current fetch partition APIs

1. list<Partition> get_partitions(1:string db_name, 2:string tbl_name, 3:i16 max_parts=-1)

Gets list of Partition objects given the db and tbl_name. Has a check for max number of partitions that can be requested. If the number of partitions requested > limit a MetaException is thrown. By default all the partitions are fetched.

This API supports directSQL

2. list<Partition> get_partitions_with_auth(1:string db_name, 2:string tbl_name, 3:i16 max_parts=-1, 4: string user_name, 5: list<string> group_names)

This is equivalent to API in (1) except it also fetches the partition privileges information. Specifically, it queries and retrieves all the user, group and roles which have permissions on each of the returned partitions by internally querying MPartitionPrivilege objects (internally maps to PART_PRIVS) table. This API loops through all the partitions and calls multiple queries on each partition to fetch the partition privilege information. Note that this API is different than above API functionally only when the table property PARTITION_LEVEL_PRIVILEGE is set.

This API does not support directSQL.

3. list<PartitionSpec> get_partitions_pspec(1:string db_name, 2:string tbl_name, 3:i32 max_parts=-1)

This API is very interesting because it de-duplicates StorageDescriptor (SD) objects of the partition objects and location information for all the partitions. In order to de-duplicate the SDs it groups the partitions into two categories one which are stored under table location and another which are stored at a different location. For all the partitions which are stored under the table location, it sends only one copy of each unique SD along with the list of partitions which are stripped of their SD. The location field of the partition is also trimmed down. Each partition only provides the relative location w.r.t table location.

For the partitions which are not under table location, there is no deduplication done and they are all lumped into a object called PartitionListComposingSpec.

This API applies only when the table property **hive.hcatalog.partition.spec.grouping.enabled** is set to true so potentially was added specifically for some hcatalog use-case.

4. list<string> get_partition_names(1:string db_name, 2:string tbl_name, 3:i16 max_parts=-1)

Returns the partition names (key=value lists) for the given table and database. Ideally should be fast since it only queries PART_NAME columns from the PARTITIONS joined by table and database name. This method does not support directSQL.

5. PartitionValuesResponse get_partition_values(1:PartitionValuesRequest request)

Returns the list of partition values given the list of partition keys passed from the request object. Note that the request list of keys could be subset of total number of partition keys of the table and the API fetches only the values of the requested partition keys. In order to fetch the values only, it queries the partition names and then applies a substring operation using the given key in the request. In addition to retrieving only the partition values, this API also has the ability to pass the JDO filter strings in the request object. This filter string is parsed using the grammar defined in filter.g to construct a JDO filter string which can be pushed down to the database. Interestingly, the string value of the filter string is used to convert to ExpressionTree form. The get_partitions_by_expr API (see 11 below) adds a layer on top of this filter string using ExprNodeGenericFuncDesc passed around a deserialized byte array.

The request object also can pass distinct flag and the ascending/descending flag for the ordering of the results. One weird thing about this API is when the fetch partitionNames by filter doesn't work it does a fetch full partition objects by the same filter and then extracts the partitionNames from the result. It also seems to be a doing a unnecessary sort operation when it uses fetch partition using filter because fetch partition API anyways returns ascending order. In order to cover for that it does a reverse sort operation which doesn't take into account that fetch partition names API by filter already has a descending order by clause in there.

6. list<Partition> get_partitions_ps(1:string db_name 2:string tbl_name, 3:list<string> part_vals, 4:i16 max_parts=-1)

Returns a list of partitions with the matching partition part_vals. It internally uses (7) below with the null values for user_name and group_names. Internally it issues a JDOQL to query partition name from MPartition objects which match the constructed regex string using the given part_vals. This method does not support directSQL

7. list<Partition> get_partitions_ps_with_auth(1:string db_name, 2:string tbl_name, 3:list<string> part_vals, 4:i16 max_parts=-1, 5: string user_name, 6: list<string> group_names)

Same as (6) but with the addition auth information. The auth information (user_name, group_names) is used to get the partition privilege information as well for the given user and group_names. Check (2) to understand what partition privilege information consists of.

8. list<string> get_partition_names_ps(1:string db_name, 2:string tbl_name, 3:list<string> part_vals, 4:i16 max_parts=-1)

This API is internally same as get_partitions_ps except instead of returning the full partition objects it returns the partition name objects only.

9. list<Partition> get_partitions_by_filter(1:string db_name 2:string tbl_name, 3:string filter, 4:i16 max_parts=-1)

This API returns a list of partitions given a db name, table name and a JDO filter string which can pushed down to the database. It also supports DirectSQL. This is the same filter implementation used in get_partition_values (5).

10. list<PartitionSpec> get_part_specs_by_filter(1:string db_name 2:string tbl_name, 3:string filter, 4:i32 max_parts=-1)

This API is same as get_partitions_pspec (2) except that it takes in a filter string as well. It internally uses get_partitions_by_filter from (9) and then deduplicates using the same logic as described in get_partitions_pspec (3)

11. PartitionsByExprResult get_partitions_by_expr(1:PartitionsByExprRequest req)

This is a different way of filtering the partitions based on expressions. In this case the request provides the tablename, dbname and a byte[]. This byte[] is generated using kryo by serialization of the ExprNodeGenericFuncDesc which is defined in ql. There are 2 known issues of using this mechanism to pass the filters to metastore. First, the ExprNodeGenericFuncDesc in defined in ql module which is used to describe a function node of an expression tree used during query execution. The jars which contain these classes are present in hive-exec which should be not present when using standalone-metastore. In a standalone-metastore this particular API does not work if hive-exec jars are not present in the classpath. This API is specifically designed for hive and will only work for hive. Secondly, it also can break backwards compatibility easily because of the fact that the serialized and deserialized expressions of the byte[] will be dependent on the hive-exec jar versions on the HS2 and metastore side. So in case the versions are different, it is possible that this API can break for random use-cases where some UDF is serialized in an unexpected way during the filter generation. However, this basically builds on top of (9) and (5) which also accept filter strings using ExpressionTree. So the transformation flow of the expression is as following:

ExprNodeGenericFuncDesc -> byte[] -> send over thrift -> ExprNodeGenericFuncDesc -> toString() -> ExpressionTree -> JDO/DirectSQL filter strings
Implementation vise it calls into the same internal objectstore API used in get_partition_names to retrieve all the partition names for the given database and table name. It then using PartitionPruner to prune all the partition names which don't satisfy the given expression.

12. i32 get_num_partitions_by_filter(1:string db_name 2:string tbl_name 3:string filter)

Same as get_partitions_by_filter except gets a count of the partitions for the given filter.

13. list<Partition> get_partitions_by_names(1:string db_name 2:string tbl_name 3:list<string> names)

Gets the partitions given the names. Uses directSQL.

14. Partition get_partition(1:string db_name, 2:string tbl_name, 3:list<string> part_vals)

Gets the partition object given the dbname, tablename and its values

15. Partition get_partition_with_auth(1:string db_name, 2:string tbl_name, 3:list<string> part_vals, 4: string user_name, 5: list<string> group_names)

Same as get_partition (14) but gets the partition privileges as well.

16. Partition get_partition_by_name(1:string db_name 2:string tbl_name, 3:string part_name)

Gets the partition object given dbname, tablename and the partition name (key1=value1/key2=value2)

Related JIRAs

HIVE-7223 attempted to do something related to deduplication of the results. Specifically it adds the PartitionSpec object which keeps track of common Storage Descriptor object to all the partitions, the rootLocation and all the partition objects are lightly loaded (without sd and relative location instead of full path) which can be reused for returning partition information of this API. While this API deduplicates the data on wire so that payload size is much smaller, it still issues a complete fetch of full partition object so in terms of runtime performance it is no better than the current fetch partitions API on the server side.

This API is also limited to the table which have property hive.hcatalog.partition.spec.grouping.enabled is set. We can however use this same principle in general for the response of the generic fetch partitions API.

Design

Considering the overall set of different get_partition APIs above a common theme seems to be that clients want to extract full or partial information from the Partition objects which are filtered by some criteria. Most basic criteria is the filter by table and database name, but many APIs have more complex filtering mechanisms which can be pushed down to the database. Additionally, some API have a limit parameter which relies on clients to set a “good” value of number of partitions to fetch in request. This could be problematic because clients rarely know what is a good value to set for requesting the number of partitions. Most cases the the limit is set as -1 which means returning all the matching partitions. This could cause performance problems for metastore when there are large tables with hundreds of thousands of partitions. A good API should determine the good number of partitions to be sent to the client and provide clients a mechanism to request for rest of partitions in subsequent request. This kind of pagination is tricky to implement since Thrift does not natively provide support for pagination. The transaction semantics of pagination are not clear in the sense two batches of fetch partitions in the same requests happen in separate transactions on the server side so in theory it can return inconsistent results. This is in general not a problem for hive because it acquires locks which reading tables and partitions but these locks are specific to hive and other applications do not understand how to read/acquire them.

Projection Fields

There are couple of approaches to implement fetching sub-set of partition fields.

Approach 1

The first approach is to mark all the fields of Partition as optional. The fields which are changed from “required -> optional” is technically wire-compatible so it will not break existing APIs. In order to truly support selective fetching of all the nested fields we would need to mark the fields of the nested structures like StorageDescriptor as optional too. For instance if some application wants to fetch all the partition locations for a table, we will need to mark location field of Storage Descriptor as optional too in addition to marking parameters field as optional in Partition. While the advantages of this partially filled partition is that clients can request only the information they care about, the disadvantage is the increased burden on all the other write APIs (add_partition, alter_partition) to make sure that the required fields set on the server-side. This is currently done by Thrift for free on the client-side. This can also be confusing to clients in the sense it can fetch a partially filled partition but if they have to modify its state they need to send a fully-filled partition object which they do not have.

Approach 2

The second approach is to define new Thrift objects which replicate what Partition have but instead mark all the fields as optional. The only advantage over approach 1 of this approach is that there is no need for the write side API validation since Thrift makes sure that Partition objects will always have the required objects as set. This approach also has the same disadvantages as listed in approach 1. Additionally, you have a maintenance overhead to keep up with the Partition thrift definition.

Approach 3

The third approach meets the above two approaches in the middle. It leverages existing Thrift PartitionSpec objects. PartitionSpec is defined as follows:

```
struct PartitionSpec {
 1: string dbName,
 2: string tableName,
 3: string rootPath,
 4: optional PartitionSpecWithSharedSD sharedSDPartitionSpec,
 5: optional PartitionListComposingSpec partitionList,
 6: optional string catName
}

struct PartitionWithoutSD {
 1: list<string> values // string value is converted to appropriate partition key type
 2: i32          createTime,
 3: i32          lastAccessTime,
 4: string       relativePath,
 5: map<string, string> parameters,
 6: optional PrincipalPrivilegeSet privileges
}

struct PartitionSpecWithSharedSD {
 1: list<PartitionWithoutSD> partitions,
 2: StorageDescriptor sd,
}

struct PartitionListComposingSpec {
 1: list<Partition> partitions
}
```

In case of PartitionSpec the StorageDescriptor is deduplicated such that for all the Partitions which share the same StorageDescriptor it only sends the common information once in PartitionSpecWithSharedSD object. The path object is also efficient in the sense if all the partitions are within the table directory, it stores one copy of top level directory (rootPath) and all the partitions send the relativePath with respect to the rootPath. For non-standard partitions

(which have different storage descriptor and/or location) it sends the fully-filled partition object. This API optimizes for the most common case and is no-worse than the current APIs for the less-common case.

One further optimization which can be done in existing PartitionSpec is to mark its fields as optional. Since PartitionSpec is used only in a few existing APIs the validation scope for write-side APIs is reduced. Also, since this structure is created to have a “light-weight” partition objects it makes sense to make it even more lighter. This can be done by marking all the fields of PartitionWithoutSD as optional. This will help with the case of read-only clients requiring only subset of fields while still keeping the scope of changes limited to only the APIs which use PartitionSpec APIs.

Pagination [Work in progress]

In case of thousands of partitions it is reasonable to have some kind of default pagination mechanism for the response so that we don’t overwhelm the client and server by the limiting number of objects returned. One basic pagination scheme could be that request object could include offset and limit integer values which represent the offset and the number of objects from that offset in the returned list of partitions. Eg. If there are 10k partitions for a table and the client can only process 1000 at a time, client can issue multiple requests with <offset,limit> pairs like <0,1000>, <1000, 2000>, <2000,3000> etc.

As described above the issue with this approach is most of the times clients don’t know what is a good number of partitions to request in each batch. They would almost always request all the partitions in one batch which could cause issues on the server side. Instead the metastore server should decide what is the number of partitions it can send for each request. This limit can be either configured via hive-site.xml or can be determined dynamically in the runtime (using error statistics, load metrics etc). For the first version we can just rely on a configured limit and improve this limit to auto-scale dynamically depending on the load factor on the metastore server in the future. In such a scheme the response of the request should indicate that there are more partitions to come and client should send another request with the correct offset value.

Such server side implementation of pagination can be done using following approaches:

Approach 1

JDO Query provides setRange() API to limit the number of rows returned. The offset and limit fields from the request object can be directly passed down to the database using setRange(offset, offset+limit) so that only subset of the results are returned. Internally in the directSQL implementation this will work by fetching PART_IDs in the given rowid range (offset, offset+limit) and then return the partitions corresponding to the selected PART_IDs. The advantage of this approach is we don’t need to introduce any new fields which would mean it would seamlessly start working for existing clients.

In order to support transactionality of get_partitions across multiple fetch calls for each batch, we can use a modification stamp (UUID/number) stored as a table property. The get_partitions request provides the optional value. When client provides such a UUID string, server will check the current UUID of the table object and serves the batch of partitions if it matches. If it does not match, server throws an exception and informs the client to retry the operation for the beginning of the transaction (get all partitions from offset 0). This also means that the modification stamp on the table property has to be updated (generate new UUID and update) in the table property for all the alter_table and alter_partition API calls.

Approach 2

If pagination needs to be supported with transactional semantics the server can cache the results temporarily while the client requests the results in batches. In this approach, HMS server will cache the entire list of PartitionSpec objects and return the client with a unique identifier and count of the results. Clients can then issue multiple calls to fetch the results until all the results are fetched.

For example, in this approach the client can send the request with desired number of partition batch size. If the batch size in the requested is greater than the result size all the results are sent in the response. If however, the results are greater than batch size requested, the response will include a pending count which is still unfetched. Client can send another request with the same unique id to request subsequent batches of the results.

This achieves transactionality of get_partitions API call but could potentially increase memory pressure on the server if clients take long time to process batches. The server response can include a timeout value to indicate how much more time the results will be cached until they expire automatically. Additionally, the results could be stored as weak references so that they could be GCed if required when HMS is under memory pressure.

API Spec

The new API should use request and response structs so that newer fields can be added in the future. This helps with overall extensibility of the API. There are many other APIs with “get_partitions” in their name, so it would be preferable to have the same name.

```
GetPartitionsResponse get_partitions(GetPartitionsRequest request)

GetPartitionsRequest {
  PartitionProjectionSpec projectionSpec
  PartitionFilterSpec filterSpec
  optional PaginationToken token
}

PartitionProjectionSpec {
```

```
  list<string> fieldList; // dot separated strings. Eg sd.location, serdeInfo.name. Empty list will mean all the fields
  list<string> paramList; //keys for the params to be either included or excluded
  bool includeParamList; //if true paramList acts as a inclusion list else acts exclusion list
}

PartitionFilterSpec {
  string dbName,
  string tblName,
  optional bool withAuth,
  optional string user,
  optional  list<string> groupNames,
  optional enum PartitionFilterMode,
  optional list<string> filters //used as list of partitionNames or list of values or expressions depending on mode
}

PaginationToken {
 byte[] token
}

enum PartitionFilterMode {
  BY_NAMES, //filter by names
  BY_VALUES, //filter by values
  BY_EXPR //filter by expression
}

GetPartitionsResponse {
 list<PartitionSpec>
 i32 countReturned //number of partitions returned
 i32 count //total count of partitions for the given filterSpec in the request
 optional PaginationToken token //to be sent back by the client in case the countReturned does not match with count
}
```