AccumuloIntegration

Hive Accumulo Integration

Overview

Apache Accumulo is a sorted, distributed key-value store based on the Google BigTable paper. The API methods that Accumulo provides are in terms of Keys and Values which present the highest level of flexibility in reading and writing data; however, higher-level query abstractions are typically an exercise left to the user. Leveraging Apache Hive as a SQL interface to Accumulo complements its existing high-throughput batch access and low-latency random lookups.

Implementation

The initial implementation was added to Hive 0.14 in HIVE-7068 and is designed to work with Accumulo 1.6.x. There are two main components which make up the implementation: the AccumuloStorageHandler and the AccumuloPredicateHandler. The AccumuloStorageHandler is a StorageHandler implementation. The primary roles of this class are to manage the mapping of Hive table to Accumulo table and configures Hive queries. The AccumuloPredicateHandler is used push down filter operations to the Accumulo for more efficient reduction of data.

Accumulo Configuration

The only additional Accumulo configuration necessary is the inclusion of the hive-`accumulo-handler.jar`, provided as a part of the Hive distribution, to be included in the Accumulo server classpath. This can be accomplished a variety of ways: copying/symlink the jar into `$ACCUMULO_HOME/lib` or `$ACCUMULO_HOME/lib/ext` or include the path to the jar in `general.classpaths` in accumulo-site.xml. Be sure to restart the Accumulo tabletservers if the jar is added to the classpath in a non-dynamic fashion (using `$ACCUMULO_HOME/lib` or `general.classpaths` in accumulo-site.xml).

Usage

To issue queries against Accumulo using Hive, four parameters must be provided by the Hive configuration:

Connection Parameters
accumulo.instance.name
accumulo.zookeepers
accumulo.user.name
accumulo.user.pass

For those familiar with Accumulo, these four configurations are the normal configuration values necessary to connect to Accumulo: the Accumulo instance name, the ZooKeeper quorum (comma-separated list of hosts), and Accumulo username and password. The easiest way to provide these values is by using the `-hiveconf` option to the `hive` command. It is expected that the Accumulo user provided either has the ability to create new tables, or that the Hive queries will only be accessing existing Accumulo tables.

```
hive -hiveconf accumulo.instance.name=accumulo -hiveconf accumulo.zookeepers=localhost -hiveconf accumulo.user.
name=hive -hiveconf accumulo.user.pass=hive
```

To access Accumulo tables, a Hive table must be created using the `CREATE` command with the `STORED BY` clause. If the `EXTERNAL` keyword is omitted from the `CREATE` call, the lifecycle of the Accumulo table is tied to the lifetime of the Hive table: if the Hive table is deleted, so is the Accumulo table. This is the default case. Providing the `EXTERNAL` keyword will create a Hive table that references an Accumulo table but will not remove the underlying Accumulo table if the Hive table is dropped.

Each Hive row maps to a set of Accumulo keys with the same row ID. One column in the Hive row is designated as a "special" column which is used as the Accumulo row ID. All other Hive columns in the row have some mapping to Accumulo column (column family and qualifier) where the Hive column value is placed in the Accumulo value.

```
CREATE TABLE accumulo_table(rowid STRING, name STRING, age INT, weight DOUBLE, height INT)
STORED BY 'org.apache.hadoop.hive.accumulo.AccumuloStorageHandler'
WITH SERDEPROPERTIES('accumulo.columns.mapping' = ':rowid,person:name,person:age,person:weight,person:height');
```

In the above statement, normal Hive column name and type pairs are provided as is the case with normal create table statements. The full AccumuloStorageHandler class name is provided to inform Hive that Accumulo will back this Hive table. A number of properties can be provided to configure the AccumuloStorageHandler via SERDEPROPERTIES or TBLPROPERTIES. The most important property is "accumulo.columns.mapping" which controls how the Hive columns map to Accumulo columns. In this case, the "row" Hive column is used to populate the Accumulo row ID component of the Accumulo Key, while the other Hive columns (name, age, weight and height) are all columns within the Accumulo row.

For the above schema in the "accumulo_table", we could envision a single row in the table:

```
hive> select * from accumulo_table;
row1        Steve        32        200        72
```

The above record would be serialized into Accumulo Key-Value pairs in the following manner given the declared accumulo.columns.mapping:

```
user@accumulo accumulo_table> scan
row1        person:age []        32
row1        person:height []        72
row1        person:name []        Steve
row1        person:weight []        200
```

The power of the column mapping is that multiple Hive tables with differing column mappings can interact with the same Accumulo table and produce different results. When columns are excluded, the performance of Hive queries can be improved through the use of Accumulo locality groups to filter out unwanted data at the server-side.

Column Mapping

The column mapping string is comma-separated list of encoded values whose offset corresponds to the Hive schema for the table. The order of the columns in the Hive schema can be arbitrary as long as the elements in the column mapping align to the intended Hive column. For those familiar with Accumulo, each element in the column mapping string resembles a column_family:column_qualifier; however, there are a few different variants that allow for different control.

1. A single column
   a. This places the value for the Hive column into the Accumulo value with the given column family and column qualifier.
2. A column qualifier map
   a. A column family is provided and a column qualifier prefix of any length is allowed, follow by an asterisk.
   b. The Hive column type is expected to be a Map, the key of the Hive map is appended to the column qualifier prefix
   c. The value of the Hive map is placed in the Accumulo value.
3. The rowid
   a. Controls which Hive column is used as the Accumulo rowid.
   b. Exactly one ":rowid" element must exist in each column mapping
   c. ":rowid" is case insensitive (:rowID is equivalent to :rowId)

Additionally, a serialization option can be provided to each element in the column mapping which will control how the value is serialized. Currently, the options are:

- 'binary' or 'b'
- 'string' or 's'

These are set by including a pound sign ('#') after the column mapping element with either the long or short serialization value. The default serialization is 'string'. For example, for the value 10, "person:age#s" is synonymous with the "person:age" and would serialize the value as the literal string "10". If "person:age#b" was used instead, the value would be serialized as four bytes: \x00\x00\x00\xA0.

Indexing

Starting in Hive 3.0.0 with HIVE-15795, indexing support has been added to Accumulo-backed Hive tables. Indexing works by using another Accumulo table to store the field value mapping to rowId of the data table. The index table is automatically populated on record insertion via Hive.

Using index tables greatly improve performance of non-rowId predicate queries by eliminating full table scans. Indexing works for both internally and externally managed tables using either the Tez or Map Reduce query engines. The following options control indexing behavior.

Option Name	Description
accumulo.indextable.name	**(Required) The name of the index table in Accumulo.**
accumulo.indexed. columns	(Optional) A comma separated list of hive columns to index, or * which indexes all columns (default: *)
accumulo.index.rows.max	(Optional) The maximum number of predicate values to scan from the index for each search predicate (default: 20000) *See this note about this value*
accumulo.index.scanner	(Optional) The index scanner implementation. (default: org.apache.hadoop.hive.accumulo. AccumuloDefaultIndexScanner)

The indexes are stored in the index table using the following format:

```
rowId = [field value in data table]
```

```
column_family = [field column family in data table] + ‘_’ + [field column quantifier in data table]
```

```
column_quantifier = [field rowId in data table]
```

```
visibility = [field visibility in data table]
```

```
value = [empty byte array]
```

When using a string encoded table, the indexed field value is encoded using Accumulo Lexicoder methods for numeric types. Otherwise, values are encoding using native binary encoding. This information will allow applications to insert data and index values into Accumulo outside of Hive but still require high performance queries from within Hive.

It is important to note when inserting data and indexes outside of Hive it is important to update both tables within the same unit of work. If the Hive query does not find indexes matches for the any of the query predicates, the query will short circuit and return empty results without searching the data table.

If the search predicate matches more entries than defined by the option `accumulo.index.rows.max` (default 20000), the index search results will be abandoned and the query will fall back to a full scan of the data table with predicate filtering. Remember using large values for this option or having very large data table rowId values may require increasing hive memory to prevent memory errors.

Other options

The following options are also valid to be used with SERDEPROPERTIES or TABLEPROPERTIES for further control over the actions of the AccumuloStorageHandler:

Option Name	Description
accumulo.iterator.pushdown	Should filter predicates be satisfied within Accumulo using Iterators (default: true)
accumulo.default.storage	The default storage serialization method for values (default: string)
accumulo.visibility.label	A static ColumnVisibility string to use when writing any records to Accumulo (default: empty string)
accumulo.authorizations	A comma-separated list of authorizations to use when scanning Accumulo (default: no authorizations). Note that the Accumulo user provided to connect to Accumulo must have all authorizations provided.
accumulo.composite.rowid. factory	Extension point which allows a custom class to be provided when constructing LazyObjects from the rowid without changing the ObjectInspector for the rowid column.
accumulo.composite.rowid	Extension point which allows for custom parsing of the rowid column into a LazyObject.
accumulo.table.name	Controls what Accumulo table name is used (default: the Hive table name)
accumulo.mock.instance	Use a MockAccumulo instance instead of connecting to a real instance (default: false). Useful for testing.

Examples

Override the Accumulo table name

Create a user table, consisting of some unique key for a user, a user ID, and a username. The Accumulo row ID is from the Hive column, the user ID column is written to the "f" column family and "userid" column qualifier, and the username column to the "f" column family and the "nickname" column qualifier. Instead of using the "users" Accumulo table, it is overridden in the TBLPROPERTIES to use the Accumulo table "hive_users" instead.

```
CREATE TABLE users(key int, userid int, username string)
STORED BY 'org.apache.hadoop.hive.accumulo.AccumuloStorageHandler'
WITH SERDEPROPERTIES ("accumulo.columns.mapping" = ":rowID,f:userid,f:nickname")
WITH TBLPROPERTIES ("accumulo.table.name" = "hive_users");
```

Store a Hive map with binary serialization

Using an asterisk in the column mapping string, a Hive map can be expanded from a single Accumulo Key-Value pair to multiple Key-Value pairs. The Hive Map is a parameterized type: in the below case, the key is a string, and the value integer. The default serialization is overriden from 'string' to 'binary' which means that the integers in the value of the Hive map will be stored as a series of bytes instead of the UTF-8 string representation.

```
CREATE TABLE hive_map(key int, value map<string,int>)
STORED BY 'org.apache.hadoop.hive.accumulo.AccumuloStorageHandler'
WITH SERDEPROPERTIES (
  "accumulo.columns.mapping" = ":rowID,cf:*",
  "accumulo.default.storage" = "binary"
);
```

Register an external table

Creating the Hive table with the external keyword decouples the lifecycle of the Accumulo table from that of the Hive table. Creating this table assumes that the Accumulo table "countries" already exists. This is a very useful way to use Hive to manage tables that are created and populated by some external tool (e.g. A MapReduce job). When the Hive table countries is deleted, the Accumulo table will not be deleted. Additionally, the external keyword can also be useful when creating multiple Hive tables with different options that operate on the same underlying Accumulo table.

```
CREATE EXTERNAL TABLE countries(key string, name string, country string, country_id int)
STORED BY 'org.apache.hadoop.hive.accumulo.AccumuloStorageHandler'
WITH SERDEPROPERTIES ("accumulo.columns.mapping" = ":rowID,info:name,info:country,info:country_id");
```

Create an indexed table

To take advantage of indexing, Hive uses another Accumulo table is used to create a lexicographically-sorted search term index for each field allowing for very efficient exact match and bounded range searches.

```
CREATE TABLE company_stats (
   rowid string,
   active_entry boolean,
   num_offices tinyint,
   num_personel smallint,
   total_manhours int,
   num_shareholders bigint,
   eff_rating float,
   err_rating double,
   yearly_production decimal,
   start_date date,
   address varchar(100),
   phone char(13),
   last_update timestamp )
ROW FORMAT SERDE 'org.apache.hadoop.hive.accumulo.serde.AccumuloSerDe'
STORED BY 'org.apache.hadoop.hive.accumulo.AccumuloStorageHandler'
WITH SERDEPROPERTIES (
   "accumulo.columns.mapping" = ":rowID,a:act,a:off,a:per,a:mhs,a:shs,a:eff,a:err,a:yp,a:sd,a:addr,a:ph,a:lu”,
   "accumulo.table.name"="company_stats",
   "accumulo.indextable.name"="company_stats_idx"
 );
```

Acknowledgements

I would be remiss to not mention the efforts made by Brian Femiano that were the basis for this storage handler. His initial prototype for Accumulo-Hive integration was the base for this work.