ViewDev

Hive Views

Use Cases

Views (http://issues.apache.org/jira/browse/HIVE-972) are a standard DBMS feature and their uses are well understood. A typical use case might be to create an interface layer with a consistent entity/attribute naming scheme on top of an existing set of inconsistently named tables, without having to cause disruption due to direct modification of the tables. More advanced use cases would involve predefined filters, joins, aggregations, etc for simplifying query construction by end users, as well as sharing common definitions within ETL pipelines.

Scope

At a minimum, we want to

- add queryable view support at the SQL language level (specifics of the scoping are under discussion in the Issues section below)
  - updatable views will not be supported (see the Updatable Views proposal)
- make sure views and their definitions show up anywhere tables can currently be enumerated/searched/described
- where relevant, provide additional metadata to allow views to be distinguished from tables

Beyond this, we may want to

- expose metadata about view definitions and dependencies (at table-level or column-level) in a way that makes them consumable by metadata-driven tools

Syntax

```
CREATE VIEW [IF NOT EXISTS] view_name [(column_name [COMMENT column_comment], ...) ]
[COMMENT table_comment]
AS SELECT ...

DROP VIEW view_name
```

Implementation Sketch

The basics of view implementation are very easy due to the fact that Hive already supports subselects in the FROM clause.

- For **CREATE VIEW v AS view-def-select**, we extend SemanticAnalyzer to behave similarly to **CREATE TABLE t AS select**, except that we don't actually execute the query (we stop after plan generation). It's necessary to perform all of plan generation (even though we're not actually going to execute the plan) since currently some validations such as type compatibility-checking are only performed during plan generation. After successful validation, the text of the view is saved in the metastore (the simplest approach snips out the text from the parser's token stream, but this approach introduces problems described in the issues section below).
- For **select ... from view-reference**, we detect the view reference in SemanticAnalyzer.getMetaData, load the text of its definition from the metastore, parse it back into an AST, prepare a QBExpr to hold it, and then plug this into the referencing query's QB, resulting in a tree equivalent to **select ... from (view-def-select)**; plan generation can then be carried out on the combined tree.

Issues

Some of these are related to functionality/scope; others are related to implementation approaches. Opinions are welcome on all of them.

Stored View Definition

In SQL:200n, a view definition is supposed to be frozen at the time it is created, so that if the view is defined as select * from t, where t is a table with two columns a and b, then later requests to select * from the view should return just columns a and b, even if a new column c is later added to the table. This is implemented correctly by most DBMS products.

There are similar issues with other kinds of references in the view definition; for example, if a table or function name can be qualified, then the reference should be bound at the time the view is created.

Implementing this typically requires expanding the view definition into an explicit form rather than storing the original view definition text directly. Doing this could require adding "unparse" support to the AST model (to be applied after object name resolution takes place), something which is not currently present (and which is also useful to have available in general).

However, storing both the expanded form and the original view definition text as well can also be useful for both DESCRIBE readability as well as functionality (see later section on ALTER VIEW v RECOMPILE).

Update 7-Jan-2010: Rather than adding full-blown unparse support to the AST model, I'm taking a parser-dependent shortcut. ANTLR's TokenRewriteStream provides a way to substitute text for token subsequences from the original token stream and then regenerate a transformed version of the parsed text. So, during column resolution, we map an expression such as "t.*" to replacement text "t.c1, t.c2, t.c3". Then once all columns have been resolved, we regenerate the view definition using these mapped replacements. Likewise, an unqualified column reference such as "c" gets replaced with the qualified reference "t.c". The rest of the parsed text remains unchanged.

This approach will break if we ever need to perform more drastic (AST-based) rewrites as part of view expansion in the future.

Metastore Modeling

The metastore model will need to be augmented in order to allow view definitions to be saved. An important issue to be resolved is whether to model this via inheritance, or just shoehorn views in as a special kind of table.

With an inheritance model, views and base tables would share a common base class (here called ColumnSet following the convention in the Common Warehouse Metamodel for lack of a better term):

For a view, most of the storage descriptor (everything other than the column names and types) would be irrelevant, so this model could be further refined with such discriminations.

View names and table names share the same namespace with respect to uniqueness (i.e. you can't have a table and a view with the same name), so the name key uniqueness would need to be specified at the base class level.

Alternately, if we choose to avoid inheritance, then we could just add a new viewText attribute to the existing Table class (leaving it null for base tables):

(Storing the view definition as a table property may not work since property values are limited to VARCHAR(767), and view definitions may be much longer than that, so we'll need to use a LOB.)

Comparison of the two approaches:

	Inheritance Model	**Flat Model**
JDO Support	Need to investigate how well inheritance works for our purposes	Nothing special
Metadata queries from existing code/tools	Existing queries for tables will NOT include views in results; those that need to will have to be modified to reference base class instead	Existing queries for tables WILL include views in results; those that are not supposed to will need to filter them out
Metastore upgrade on deployment	Need to test carefully to make sure introducing inheritance doesn't corrupt existing metastore instances	Nothing special, just adding a new attribute

Update 30-Dec-2009: Based on a design review meeting, we're going to go with the flat model. Prasad pointed out that in the future, for materialized views, we may need the view definition to be tracked at the partition level as well, so that when we change the view definition, we don't have to discard existing materialized partitions if the new view result can be derived from the old one. So it may make sense to add the view definition as a new attribute of StorageDescriptor (since that is already present at both table and partition level).

Update 20-Jan-2010: After further discussion with Prasad, we decided to put the view definition on the table object instead; for details, see discussion in HIVE-972. Also, per HIVE-1068, we added an attribute to store the type (view, managed table, external table) for each table descriptor.

Dependency Tracking

It's necessary to track dependencies from a view to objects it references in the metastore:

- tables: this is mandatory if we want DROP TABLE to be able to correctly CASCADE/RESTRICT to a referencing view
- other views: same as tables
- columns: this is optional (useful for lineage inspection, but not required for implementing SQL features)
- temporary functions: we should disallow these at view creation unless we also want a concept of temporary view (or if it's OK for the referencing view to become invalid whenever the volatile function registry gets cleared)
- any other objects? (e.g. udt's coming in as part of http://issues.apache.org/jira/browse/HIVE-779)

(Note that MySQL doesn't actually implement CASCADE/RESTRICT: it just ignores the keyword and drops the table unconditionally, leaving the view dangling.)

Metastore object id's can be used for dependency modeling in order to avoid the need to update dependency records when an object is renamed. However, we'll need to decide what kinds of objects can participate in dependencies. For example, if we restrict it to just tables and views (and assuming we don't introduce inheritance for views), then we can use a model like the one below, in which the dependencies are tracked as (supplier,consumer) table pairs. (In this model, the TableDependency class is acting as an intersection table for implementing a many-to-many relationship between suppliers and consumers).

However, if later we want to introduce persistent functions, or track column dependencies, this model will be insufficient, and we may need to introduce inheritance, with a DependencyParticipant base class from which tables, columns, functions etc all derive. (Again, need to verify that JDO inheritance will actually support what we want here.)

Update 30-Dec-2009: Based on a design review meeting, we'll start with the bare-minimum MySQL approach (with no metastore support for dependency tracking), then if time allows, add dependency analysis and storage, followed by CASCADE support. See HIVE-1073 and HIVE-1074.

Dependency Invalidation

What happens when an object is modified underneath a view? For example, suppose a view references a table's column, and then ALTER TABLE is used to drop or replace that column. Note that if the column's datatype changes, the view definition may remain meaningful, but the view's schema may need to be updated to match. Here are two possible options:

- **Strict**: prevent operations which would invalidate or change the view in any way (and optionally to provide a CASCADE flag which requests that such views be dropped automatically). This is the approach taken by SQL:200n.
- **Lenient**: allow the update to proceed (and maybe warn the user of the impact), potentially leaving the view in an invalid state. Later, when an invalid view definition is referenced, throw a validation exception for the referencing query. This is the approach taken by MySQL. In the case of datatype changes, derived column datatypes already stored in metastore for referencing views would become stale until those views were recreated.

Note that besides table modifications, other operations such as CREATE OR REPLACE VIEW have similar issues (since views can reference other views). The lenient approach provides a reasonable solution for the related issue of external tables whose schemas may be dynamic (not sure if we currently support this).

Update 30-Dec-2009: Based on a design review meeting, we'll start with the lenient approach, without any support for marking objects invalid in the metastore, then if time allows, follow up with strict support and possibly metastore support for tracking object validity. See HIVE-1077.

View Modification

In SQL:200n, there's no standard way to update a view definition. MySQL supports both

- **CREATE OR REPLACE VIEW v AS new-view-def-select**
- **ALTER VIEW v AS new-view-def-select**

Note that supporting view modification requires detection of cyclic view definitions, which should be invalid. Whether this detection is carried out at the time of view modification versus reference is dependent on the strict versus lenient approaches to dependency invalidation described above.

Update 30-Dec-2009: Based on a design review meeting, we'll start with an Oracle-style ALTER VIEW v RECOMPILE, which can be used to revalidate a view definition, as well as to re-expand the original definition for clauses such as select *. Then if time allows, we'll follow up with CREATE OR REPLACE VIEW support. (The latter is less important since we're going with the lenient invalidation model, making DROP and re-CREATE possible without having to deal with downstream dependencies.) See HIVE-1077 and HIVE-1078.

Fast Path Execution

For **select * from t**, hive supports fast-path execution (skipping Map/Reduce). Is it important for this to work for **select * from v** as well?

Update 30-Dec-2009: Based on feedback in JIRA, we'll leave this as dependent on getting the fast-path working for the underlying filters and projections.

Update 6-Dec-2010: This one is addressed by Hive's new "auto local mode" feature.

ORDER BY and LIMIT in view definition

SQL:200n prohibits ORDER BY in a view definition, since a view is supposed to be a virtual (unordered) table, not a query alias. However, many DBMS's ignore this rule; for example, MySQL allows ORDER BY, but ignores it in the case where it is superceded by an ORDER BY in the query. Should we prevent ORDER BY? This question also applies to the LIMIT clause.

Update 30-Dec-2009: Based on feedback in JIRA, ORDER BY is important as forward-looking to materialized views. LIMIT may be less important, but we should probably support it too for consistency.

Underlying Partition Dependencies

Update 30-Dec-2009: Prasad pointed out that even without supporting materialized views, it may be necessary to provide users with metadata about data dependencies between views and underlying table partitions so that users can avoid seeing inconsistent results during the window when not all partitions have been refreshed with the latest data. One option is to attempt to derive this information automatically (using an overconservative guess in cases where the dependency analysis can't be made smart enough); another is to allow view creators to declare the dependency rules in some fashion as part of the view definition. Based on a design review meeting, we will probably go with the automatic analysis approach once dependency tracking is implemented. The analysis will be performed on-demand, perhaps as part of describing the view or submitting a query job against it. Until this becomes available, users may be able to do their own analysis either via empirical lineage tools or via view->table dependency tracking metadata once it is implemented. See HIVE-1079.

Update 1-Feb-2011: For the latest on this, see PartitionedViews.

Metastore Upgrades

Since we are adding new columns to the TBLS table in the metastore schema, existing metastore deployments will need to be upgraded. There are two ways this can happen.

Automatic ALTER TABLE

If the following property is set in the Hive configuration file, JDO will notice the difference between the persistent schema and the model and ALTER the tables automatically:

```
<property>
<name>datanucleus.autoCreateSchema</name>
<value>true</value>
</property>
```

Explicit ALTER TABLE

However, if the `datanucleus.autoCreateSchema` property is set to `false`, then the ALTER statements must be executed explicitly. (An administrator may have set this property for safety in production configurations.)

In this case, execute a script such as the following against the metastore database:

```
ALTER TABLE TBLS ADD COLUMN VIEW_ORIGINAL_TEXT MEDIUMTEXT;
ALTER TABLE TBLS ADD COLUMN VIEW_EXPANDED_TEXT MEDIUMTEXT;
ALTER TABLE TBLS ADD COLUMN TBL_TYPE VARCHAR(128);
```

The syntax here is for MySQL, so you may need to adjust it (particularly for CLOB datatype).

Note that it should be safe to execute this script and continue operations BEFORE upgrading Hive; the old Hive version will simply ignore/nullify the columns it doesn't recognize.

Existing Row UPDATE

After the tables are altered, the new columns will contain NULL values for existing rows describing previously created tables. This is correct for VIEW_ORIGINAL_TEXT and VIEW_EXPANDED_TEXT (since views did not previously exist), but is incorrect for the TBL_TYPE column introduced by HIVE-1068. The new Hive code is capable of handling this (automatically filling in the correct value for the new field when a descriptor is retrieved), but it does not "fix" the stored rows. This could be an issue if in the future other tools are used to retrieve information directly from the metastore database rather than accessing the metastore API.

The script below can be used to fix existing rows after the tables have been altered. It should be run AFTER all Hive instances directly accessing the metastore database have been upgraded (otherwise new null values could slip in and remain forever). For safety, it is view-aware just in case a CREATE VIEW statement has already been executed, meaning it can be rerun any time after the upgrade.

```
UPDATE TBLS SET TBL_TYPE='MANAGED_TABLE'
WHERE VIEW_ORIGINAL_TEXT IS NULL
AND NOT EXISTS(
    SELECT * FROM TABLE_PARAMS
    WHERE TABLE_PARAMS.TBL_ID=TBLS.TBL_ID
    AND PARAM_KEY='EXTERNAL'
    AND PARAM_VALUE='TRUE'
);
UPDATE TBLS SET TBL_TYPE='EXTERNAL_TABLE'
WHERE EXISTS(
    SELECT * FROM TABLE_PARAMS
    WHERE TABLE_PARAMS.TBL_ID=TBLS.TBL_ID
    AND PARAM_KEY='EXTERNAL'
    AND PARAM_VALUE='TRUE'
);
```

For MySQL, note that the "safe updates" feature will need to be disabled since these are full-table updates.