Impala Column Masking and Row Filtering Design

Column Masking Behavior

A column masking policy is something like "only show last 4 chars of the phone column to user X when reading ‘customer’ table". When user X reads the phone column, the value woule be something like "xxxxx6789" instead of the real value "123456789". See more at https://cwiki.apache.org/confluence/display/RANGER/Row-level+filtering+and+column-masking+using+Apache+Ranger+policies+in+Apache+Hive

The behavior is clear when the query is simple. However, there are two different behaviors when the query contains subqueries. The key part is where we should perform the masking. The two different behaviors at performing masking:

(a) Mask columns inside the query when read from the masked table (i.e. at the innermost query block of the masked table). Then predicates (including JOIN conjunctions) will perform on masked values.

(b) Mask columns at the select list of the outermost query block. Then predicates (including JOIN conjunctions) will perform on original values.

To be specifit, consider these two queries:

(1) subquery contains predicates on **unmasked** value

```
SELECT concat(name, phone) FROM (
  SELECT name, phone FROM customer WHERE phone =
'123456789'
) t;
```

(2) subquery contains predicates on **masked** value

```
SELECT concat(name, phone) FROM (
  SELECT name, phone FROM customer WHERE phone =
'xxxxx6789'
) t;
```

Let's say there's actually one row in table 'customer' satisfying phone = '123456789' and the corresponding ‘name’ is “Bob”. When user X runs the queries, the results of the two different behaviors are:

(a) Query1 returns nothing. Query2 returns one result: "Bobxxxxx6789".
(b) Query1 returns one result: "Bobxxxxx6789". Query2 returns nothing.

The pros for behavior (a) is no security leak. Because user X can't guess whether there are some customers with phone number '123456789'.
The pros for behavior (b) is users don't need to rewrite their existing queries after admin applies column masking policies.

Hive chooses behavior (a) and Impala will choose behavior (a) too. See more discussions in the mailing-list thread:
https://lists.apache.org/thread.html/93c4a6a7f20c88e9472067074c9746b99f0a47880aaed1417b9771ac%40%3Cdev.impala.apache.org%3E

Row Filtering Behavior

A row filtering policy is something like “only show rows satisfying state=’CA’ to user X when reading ‘customer’ table”. When user X uses the ‘customer’ table, only rows satisfying the policy expressions will be read out. For more examples, see the official definition:
https://cwiki.apache.org/confluence/display/RANGER/Row-level+filtering+and+column-masking+using+Apache+Ranger+policies+in+Apache+Hive

Same as the column masking behaviors, there could be two behaviors about where we perform the predicates. The difference happens when those predicates can’t be pushed down due to query semantic.

To be specific, consider the following query:

```
SELECT o.id, c.id, c.name
FROM orders o
LEFT JOIN customer c ON o.c_id = c.id
WHERE c.id IS NULL
```

Let’s say there’s only one row filtering policy for “only show rows satisfying state=’CA’ to user X when reading ‘customer’ table”.

For behavior (a), the policy predicate “state=’CA’” is evaluated when reading table ‘customer’. The results would be

```
SELECT o.id, c.id, c.name
FROM orders o
LEFT JOIN (SELECT * FROM customer WHERE state = 'CA') c
  ON o.c_id = c.id
WHERE c.id IS NULL
```

For behavior (b), the policy predicate “state=’CA’” is evaluated at the outermost query block. The results would be

```
SELECT o.id, c.id, c.name
FROM orders o
LEFT JOIN customer c ON o.c_id = c.id
WHERE c.id IS NULL AND c.state = 'CA'
```

Predate “c.state = ‘CA’” can’t be pushed down since it performs on values of the nullable side of a LEFT JOIN. So the two queries produce different results.

It’s obvious that behavior (a) is what we actually want. We should perform the row filtering predicates when the target table is read. That’s also Hive’s behavior.

Ranger Column Mask Types

There are 8 types for doing data masking in Ranger, as shown in the following tables. Transformer means how to replace the column with an expression. CUSTOM type is used for users to customize the transformer.

Type	Name	Description	Transformer
MASK	Redact	Replace lowercase with 'x', uppercase with 'X', digits with '0'	mask({col})
MASK_SHOW_LAST_4	Partial mask: show last 4	Show last 4 characters; replace rest with 'x'	mask_show_last_n({col}, 4, 'x', 'x', 'x', -1, '1')
MASK_SHOW_FIRST_4	Partial mask: show first 4	Show first 4 characters; replace rest with 'x'	mask_show_first_n({col}, 4, 'x', 'x', 'x', -1, '1')
MASK_HASH	Hash	Hash the value	mask_hash({col})
MASK_NULL	Nullify	Replace with NULL	
MASK_NONE	Unmasked (retain original value)	No masking	
MASK_DATE_SHOW_YEAR	Date: show only year	Date: show only year	mask({col}, 'x', 'x', 'x', -1, '1', 1, 0, -1)
CUSTOM	Custom	Custom	

The definitions are at https://github.com/apache/ranger/blob/release-ranger-2.0.0/agents-common/src/main/resources/service-defs/ranger-servicedef-hive.json#L349-L405

The mask functions (mask, mask_show_first_n, mask_show_last_n, etc) are GenericUDFs in Hive. Currently Impala doesn't support GenericUDFs (IMPALA-7877, IMPALA-9271). We need our own implementation for these functions (IMPALA-9010).

Hive Implementation Investigation

Hive supports column masking and row filtering at HIVE-13125. It rewrites the query at semantic analysis phase and replace the table with an inline view doing the masking.

For instance, Hive transforms query

```
SELECT c_id, c_name, o_cid
FROM customer c JOIN orders ON c_id = o_cid
```

into

```
SELECT c_id, c_name, o_cid FROM (
      SELECT mask1(c_id) AS c_id, mask2(c_name) AS c_name
      FROM customer
      WHERE row_filtering_predicates
) c
JOIN orders
ON c_id = o_cid
```

Here 'customer' is the table with column masking and row filtering policies. Hive replaces the TableRef with a subquery in the semantic analysis phase. The replacement is done at **AST level**. Some entry points for the codes:

- SemanticAnalyzer#rewriteASTWithMaskAndFilter
- SemanticAnalyzer#walkASTMarkTABREF
- TableMask#create

Note that TableMask introduced in HIVE-13125 supports both column masking and row filtering policies, since all the masked expressions and row filters are put in an inline view for table masking.

Impala Column Masking Design

Our first step is to support column masking policies. Based on our behavior design, we should do the transformation at the innermost query using the masked table. There's a design in the old design doc about doing expression rewrite for column masking. Comparing to the table mask solution, doing table mask as Hive's implementation is better since we can easily support row filtering in the future (by just adding row filtering expressions into the inline view of table masking).

We can do the transformation just like we resolve a view. When a SQL string is parsed into AST by the SqlScanner, we call analyze() at the root of the AST for semantic analysis. It will recursively call analyze() for the sub nodes. The transformation can be done at this phase. To be specific, at

```
FromClause#analyze() -> Analyzer#resolveTableRef()
```

we can replace the resolved InlineViewRef or BaseTableRef with an InlineViewRef representing the table mask subquery.

To better introduce this part, we need some background about view resolution.

How Impala resolves a view

When all required tables/views metadata are loaded, Impala resolves views in the `analyze()` phase of the AST. The AST is generated by the SqlScanner. All tables/views are represented as `TableRef`s. But we don't know their exact types (table/view) without resolving them using the metadata.

The resolution starts in QueryStmt#analyze and finally goes into Analyzer#resolveTableRef:

```
SelectStmt.analyze(Analyzer)
  SelectAnalyzer in SelectStmt.analyze()
    FromClause.analyze(Analyzer)
      Analyzer.resolveTableRef(TableRef)
```

`Analyzer#resolveTableRef` will return a resolved TableRef which could be `BaseTableRef`, `InlineViewRef` and `CollectionTableRef`. Non-inline views (aka. catalog views) will be represented as `InlineViewRef`. FromClause.analyze() calls `TableRef.analyze()` for each resolved TableRef. For views, it recursively analyzes the subquery (QueryStmt). This acts like AST rewrite since a TableRef node for a view is replaced with an InlineViewRef containing a QueryStmt node. The analysis continues in the InlineViewRef. Don't need to re-analyze the AST from root.

How to do table masking in analyze phase

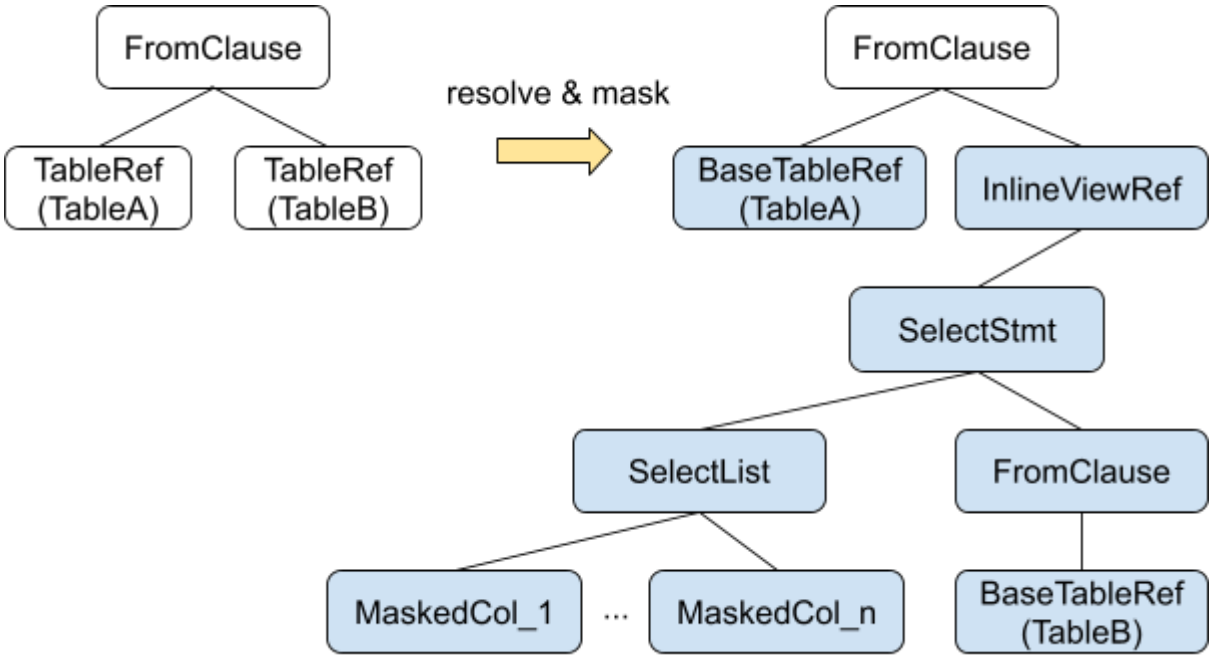
In `Analyzer.resolveTableRef`, after a TableRef is resolved we know exactly what table/view it refers to. Then we can check whether it needs to be masked: create column access requests for all the required columns on this table, and use Ranger plugin to get the column masking policies. If any of the columns need masking, we will wrap the table/view with a table masking inline view. Codes for getting the column masking policies:

```
RangerAccessResourceImpl resource = new RangerImpalaResourceBuilder()
    .database(dbName)
    .table(tableName)
    .column(columnName)
    .build();
RangerAccessRequest req = new RangerAccessRequestImpl(resource,
    SELECT_ACCESS_TYPE, user.getShortName(), getUserGroups(user));
RangerAccessResult accessResult = plugin_.evalDataMaskPolicies(req, null);
String maskType = accessResult.getMaskType();
String maskedValue = accessResult.getMaskedValue();
```

If a table/view contains column masking policies for the current user, we wrap the resolved TableRef (could be BaseTableRef or InlineViewRef) with a subquery with column maskings and return an InlineViewRef as a whole. Here are the codes about this wrapping:

```
List<Column> columns = resolvedPath.getRootTable().getColumnsInHiveOrder();
List<SelectListItem> items = Lists.newArrayListWithCapacity(columns.size());
for (Column col: columns) {
  // TODO: only add materialized columns to avoid introducing new privilege
  //  requirements (IMPALA-9223)
  items.add(new SelectListItem(
      tableMask.createColumnMask(col.getName(), col.getType()),
      col.getName()));
}
SelectList selectList = new SelectList(items);
FromClause fromClause = new FromClause(Lists.newArrayList(tableRef));
SelectStmt tableMaskStmt = new SelectStmt(selectList, fromClause,
    null, null, null, null, null);
InlineViewRef viewRef = new InlineViewRef(/*alias*/ null, tableMaskStmt,
    (TableSampleClause) null);
tableRef.migratePropertiesToTableMaskingView(viewRef);
return viewRef;
```

The above codes construct an AST node (InlineViewRef) wrapping the original table/view. It will be used to replace the original TableRef in the AST:



In the future if we want to support Ranger row filtering, we just need to add predicates into the WhereClause of this InlineViewRef (Let's call it TableMaskingView).

How to deal with alias

Alias of the original table/view should be copied to the TableMaskingView. Including the resolved alias (table name and fully qualified table name) which doesn't exist in the query.

Here're some examples. Let's say table 'default.customer' should mask its 'id' column to 'mask1(id)', and 'name' column to 'mask2(name)'.

	AST of the original query	Will be rewritten to AST of this

1	`select c.id, c.name`
	`from customer c;`
		`select c.id, c.name from (`
		`  select mask1(id) as id,`
		`  mask2(name) as name`
		`  from customer c) c;`

2	`Select c.* from customer.c;`	`Select c.* from (`
		`  Select mask1(id) as id,`
		`  mask2(name) as name`
		`  From customer c) c;`

3	`Select id, name`
	`from customer;`
		`Select id, name from (`
		`  Select mask1(id) as id,`
		`  mask2(name) as name`
		`  From customer) // view aliases_ = ['customer', 'default.customer']`

4	`select default.customer.id,`
	`  customer.name`
	`from customer;`
		`select default.customer.id,`
		`  customer.name from (`
		`  select mask1(id) as id,`
		`  mask2(name) as name`
		`  from customer) // view aliases_ = ['customer', 'default.customer']`

The queries on the right side just represent the AST. They can't be compiled by SqlScanner of course (and we don't need this).

In example #1 and #2, alias name 'c' is used by the subquery and table 'customer' inside the subquery. It's fine since they belong to different `Analyzer`s - subquery has another `Analyzer` corresponding to a new query block. The alias is used to resolve `SlotRef`s in the scope inside an `Analyzer`:

- `Analyzer.aliasMap_` maps aliases to the TupleDescriptor.
- `Analyzer.resolvePath() -> getTupleDescPaths() -> getDescriptor()` uses it to find the TupleDescriptor.

In example #3 and #4, there are no explicit aliases in the original query. So the aliases_ of table 'customer' will be ['customer', 'default.customer'] after it's resolved. These aliases should be copied to the subquery as well, so `SlotRef`s can be resolved correctly.

Apart from aliases, other fields of the resolved TableRef should be migrated (not copied) to the TableMaskingView:

- onClause_
- usingColNames_ // The columns used in 'tableA join tableB using (xxx)'
- joinOp_
- joinHints_
- tableHints_

Unmasking in Reset

Each AST node has a reset() function clearing the analyzed results. In general, all results set while calling analyze() should be reset(cleared). As we perform table masking in the analyze() phase, we need to perform unmasking in reset(). To be specific, we should unmask the TableRef in FromClause#reset(). It's used when we clone the InlineViewRef of local views. reset() is called inside clone().

The existing logics in FromClause#reset() is a bit complex. To make sure we get the same results in later resolution, the unresolved tableRefs should use fully qualified paths. Otherwise, non-fully qualified paths might incorrectly match a local view. Ideally it should replace all resolved TableRef (i.e. subclass of TableRef) with unresolved TableRef. However, we don't do this for views because local views don't have fully qualified paths. **Due to this we don't unmask a TableMasking view if the underlying TableRef is a view.**

Jiras for Column Masking

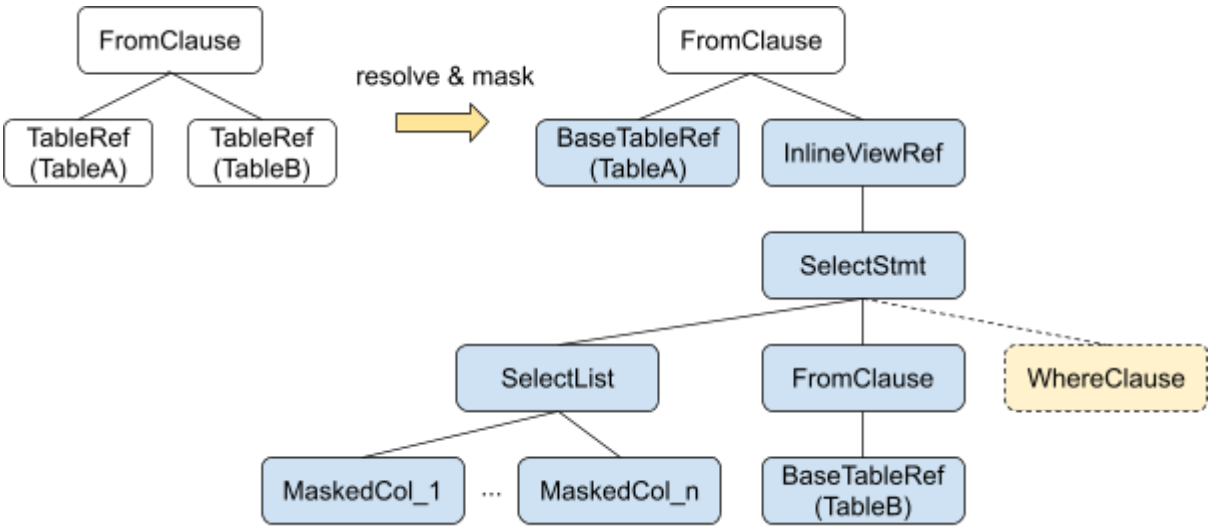
Here are JIRAs for both column masking and row filtering:

- IMPALA-9009: Core support for column masking policies.
    - Only support CUSTOM masking policies.
    - Patch under review: https://gerrit.cloudera.org/c/14894/
- IMPALA-9010: Support builtin column masking policies by adding builtin mask functions.
    - Patch under review: https://gerrit.cloudera.org/c/14963
- IMPALA-9223: Don't require privileges on unmaterialized columns
    - This is a follow-up JIRA for supporting column masking.
    - Fix the limitation of table masking views that introduces unmaterialized columns and requires additional column privileges for those columns.
    - Dev Effort estimated: 5 days

Impala Row Filtering Design

Here is a design if we plan to support row filtering policies.

We can use the table masking solution introduced above to add row filtering expressions into the table masking view when resolving TableRef:



Code logics we need to add:

- Check if there are row filtering policies on a table using Ranger plugin. (`RangerAuthorizationCheck#authorizeRowFilte`r already has this logic)
- Get string of the filter expression from `RangerAccessResult`.
- Parse the string into an AST Expr.
- Put the Expr as the WhereClause of the table masking inline view.

Special cases to cover in tests:

- Check row filtering policies on nullable side tables of outer joins.
- Check row filtering expressions containing subquery on other tables.

Jiras for Row Filtering

- IMPALA-9234: Support row filtering policies.
    - No work started on this.
    - Dev effort estimated: 5 days