horizontal lineImproving ACID performance in Apache Hive

# 

# 

# OVERVIEW

This project proposes to extend and improve the performance of ACID in Apache Hive. Currently, Hive implements ACID transactional semantics through data stored in a base file and a set of delta files. During a typical write operation, the delta files record additional INSERT, UPDATE & DELETE events made against the original data stored in the base file. These delta files are then used to impose a snapshot-based isolation scheme when reading back the data from Hive. During a typical read, the events from the appropriate delta files are collapsed with the base file to a create a consistent snapshot of the data. To keep the number of delta files from growing indefinitely, minor and major compactions are performed at regular intervals that collapse multiple delta files into a single delta file and merge delta files with the base file respectively.

The proposal is to enable predicate push-down (PPD) to delta files by splitting an UPDATE event into a combination of a DELETE+INSERT event. Essentially, what it means is that, whenever a row is updated in Hive, we would generate a delete event for the original version of the row and we would create a new insert event that inserts the new version of the updated row. This new feature is being termed as ‘*split-update*’.

The Hive ORC file format offers a nice SARG (search-argument) feature that can be used to quickly look up rows with specified values inside an ORC file. This feature is heavily used to speed-up non-ACID Hive queries that specify a predicate, by pushing the predicate condition to ORC files in terms of SARG. However, pushing predicate down to delta files in case of ACID tables may lead to incorrect results, when the delta files consist of any update/delete events. (PPD stills work for delta files consisting only of insert events). This limits a large number of ACID queries from taking advantage of PPD. However, when split-update is enabled, we will have only two kinds of delta files- the insert\_deltas & the delete\_deltas. The insert\_deltas will have the user data, while the delete\_deltas will just have the primary key of the rows that got deleted. We can then easily push PPDs to all insert\_deltas effectively giving a large performance boost.

The second advantage of this kind of splitting is that we can now effectively also split the delta files across mappers. Earlier this was prohibitively expensive because to create the final version of a row from the delta files, a mapper would be required to read all the delta files that might contain that row. However, when split-update is enabled, to create the final version of the row, we just need to know whether that row got deleted or not. And since the delete\_deltas can be highly compressed, they are small enough to be shipped around the cluster. This enables many-fold parallelism for ACID leading to huge performance gain that we have been seeing in our experiments.

The third advantage of this kind of splitting is that it now also enables efficient vectorization of ACID tables. Earlier, to create vectorized row batches in ACID, we would do the merging one row at a time and then add it incrementally to a vectorized row batch. So, even though the ORC readers would be reading in batches, the whole vectorization process was bottlenecked because of this *stitching* mechanism. However, when we just have the insert\_deltas, we can read the vectorized batches directly from the underlying ORC file and optimize deletes through in-memory lookup. As pointed out, the delete\_deltas can be highly compressed and in our experiments, it has been proved that even with a million deletes per bucket the entire size of the delete\_deltas would be still in the order of just MBs- small enough to be loaded into memory. Of course, there is a fallback strategy that just sort-merges the delete\_deltas when they could not be loaded into memory.

# GOALS

1. Enable predicate push-down to delta files when executing ACID queries (HIVE-14035)
2. Enable insert\_deltas to be split across mappers when reading ACID tables (HIVE-14035)
3. Enable versioning for ACID that can support rolling upgrades/downgrades (HIVE-14035)
4. Enable bucket pruning for ACID tables. (HIVE-14199)
5. Enable efficient vectorization for ACID tables. (HIVE-14233)

# SPECIFICATIONS

## Split-update: Split an update event into combination of delete & insert

* The main idea behind ‘split-update’ is to split an update event into a combination of a delete and an insert event (U = D + I ). That is, every time an update operation is applied on a record, it is split into a combination of delete event for the original record followed by an insert event for the newly updated value of the record.
* Now, how to store this delete part corresponding to an update event? We create two files for every update query- one recording all the inserts and the other recording all the corresponding deletes. The ACID directory structure has been modified to store all the delete\_deltas into a separate directory that starts with the prefix ‘delete\_delta\_x\_y’, where *x* and *y* would the usual transaction ids.
* Note that this kind of splitting will not have any affect on the previous sort-merge algorithm for finding the correct version of the row. The sort-merge algorithm will open the readers on both the insert\_deltas and the delete\_deltas, but this time there will no update events. Every time an inserted row has a corresponding match with the deleted record entry, it will be just skipped- like in the previous sort-merge algorithm.
* PPD can now be enabled for all the insert\_deltas, because in all cases, there will ever be only one version of the row. A row can only deleted and hence, pushing PPDs to insert\_deltas will have no side

## Creating an ACID table with split-update:

* The syntax to create an ACID table that has split-update enabled is as follows:

CREATE TABLE acidtbl (a INT, b STRING) CLUSTERED BY (a) INTO 2 BUCKETS STORED AS ORC TBLPROPERTIES ('transactional'='true', **'transactional\_properties'='default'**);

* Setting the table property **transactional\_properties** with the value ‘**default**’ will turn on the split-update for an ACID table. The keyword **transactional\_properties** allows us to add some sort of versioning to ACID tables. Currently it supports only two values: ‘**default**’ and ‘**legacy**’, the ‘legacy’ value turning on the old ACID behavior.

## Converting a non-ACID table to ACID table with split-update:

* Currently, it is also possible to start with a non-ACID table and convert it to an ACID table with split-update turned on. The syntax would be as:

ALTER TABLE nonacidtbl SET TBLPROPERTIES ('transactional'='true', **'transactional\_properties'='default'**);

## ‘transactional\_properties’ cannot be reset once an ACID table is created:

* Once an ACID table is created either via CREATE TABLE statement or by altering an existing non-ACID table, it is not possible to reset the the **‘transactional\_properties’** for that ACID table. This is because the **transactional\_properties** changes the directory layout for the ACID table.

## ACID split-update explained through illustrations:

* The following sequence of operations illustrate how an ACID directory layout will look like once split-update is enabled:
* **CREATE TABLE acidtbl (a INT, b INT) CLUSTERED BY (a) INTO 1 BUCKETS STORED AS ORC TBLPROPERTIES ('transactional'='true', 'transactional\_properties'='default');**

This creates an ACID table with split-update enabled when transactional\_properties is set to default.

* **INSERT INTO acidtbl (a,b) VALUES(1,2), (3,4);**

**/warehouse/acidtbl/delta\_0000001\_0000001\_0000/bucket\_00000**

{"operation":0,"originalTransaction":1,"bucket":0,"rowId":0,"currentTransaction":1,"row":{"\_col0":3,"\_col1":4}}

{"operation":0,"originalTransaction":1,"bucket":0,"rowId":1,"currentTransaction":1,"row":{"\_col0":1,"\_col1":2}}

An insert statement only creates an insert\_delta which is just same as the regular delta, we had in the previous version of the ACID.

* **UPDATE acidtbl SET b = -4 WHERE a = 3;**

**/warehouse/acidtbl/delta\_0000001\_0000001\_0000/bucket\_00000**

{"operation":0,"originalTransaction":1,"bucket":0,"rowId":0,"currentTransaction":1,"row":{"\_col0":3,"\_col1":4}}

{"operation":0,"originalTransaction":1,"bucket":0,"rowId":1,"currentTransaction":1,"row":{"\_col0":1,"\_col1":2}}

**/warehouse/acidtbl/delta\_0000002\_0000002\_0000/bucket\_00000**

{"operation":0,"originalTransaction":2,"bucket":0,"rowId":0,"currentTransaction":2,"row":{"\_col1":3,"\_col2":-4}}

**/warehouse/acidtbl/delete\_delta\_0000002\_0000002\_0000/bucket\_00000**

{"operation":2,"originalTransaction":1,"bucket":0,"rowId":0,"currentTransaction":2,"row":null}

An update statement is more interesting because it now creates two kinds of deltas- a regular insert delta and a delete\_delta. Note the directory structure- how the delete events were written to a delete delta in an altogether different directory. Also note that the updated row got inserted with a new primary key. This is the effect of split-update.

* **DELETE FROM acidtbl WHERE b > 0;**

**/warehouse/acidtbl/delta\_0000001\_0000001\_0000/bucket\_00000**

{"operation":0,"originalTransaction":1,"bucket":0,"rowId":0,"currentTransaction":1,"row":{"\_col0":3,"\_col1":4}}

{"operation":0,"originalTransaction":1,"bucket":0,"rowId":1,"currentTransaction":1,"row":{"\_col0":1,"\_col1":2}}

**/warehouse/acidtbl/delta\_0000002\_0000002\_0000/bucket\_00000**

{"operation":0,"originalTransaction":2,"bucket":0,"rowId":0,"currentTransaction":2,"row":{"\_col1":3,"\_col2":-4}}

**/warehouse/acidtbl/delete\_delta\_0000002\_0000002\_0000/bucket\_00000**

{"operation":2,"originalTransaction":1,"bucket":0,"rowId":0,"currentTransaction":2,"row":null}

**/warehouse/acidtbl/delete\_delta\_0000003\_0000003\_0000/bucket\_00000**

{"operation":2,"originalTransaction":1,"bucket":0,"rowId":1,"currentTransaction":3,"row":null}

Again, a delete statement *ONLY* creates a delete\_delta and no insert\_deltas. This is one of the optimizations of the split-update.

* **ALTER TABLE acidtbl COMPACT ‘MINOR’**

**/warehouse/acidtbl/delta\_0000001\_0000001\_0000/bucket\_00000**

{"operation":0,"originalTransaction":1,"bucket":0,"rowId":0,"currentTransaction":1,"row":{"\_col0":3,"\_col1":4}}

{"operation":0,"originalTransaction":1,"bucket":0,"rowId":1,"currentTransaction":1,"row":{"\_col0":1,"\_col1":2}}

**/warehouse/acidtbl/delta\_0000002\_0000002\_0000/bucket\_00000**

{"operation":0,"originalTransaction":2,"bucket":0,"rowId":0,"currentTransaction":2,"row":{"\_col1":3,"\_col2":-4}}

**/warehouse/acidtbl/delete\_delta\_0000002\_0000002\_0000/bucket\_00000**

{"operation":2,"originalTransaction":1,"bucket":0,"rowId":0,"currentTransaction":2,"row":null}

**/warehouse/acidtbl/delete\_delta\_0000003\_0000003\_0000/bucket\_00000**

{"operation":2,"originalTransaction":1,"bucket":0,"rowId":1,"currentTransaction":3,"row":null}

**/warehouse/acidtbl/delta\_0000001\_0000003/bucket\_00000**

{"operation":0,"originalTransaction":1,"bucket":0,"rowId":0,"currentTransaction":1,"row":{"a":3,"b":4}}

{"operation":0,"originalTransaction":1,"bucket":0,"rowId":1,"currentTransaction":1,"row":{"a":1,"b":2}}

{"operation":0,"originalTransaction":2,"bucket":0,"rowId":0,"currentTransaction":2,"row":{"a":3,"b":-4}}

**/warehouse/acidtbl/delete\_delta\_0000001\_0000003/bucket\_00000**

{"operation":2,"originalTransaction":1,"bucket":0,"rowId":0,"currentTransaction":2,"row":null}

{"operation":2,"originalTransaction":1,"bucket":0,"rowId":1,"currentTransaction":3,"row":null}

A minor compaction will minor compact both the insert and the delete deltas. Currently it will produce the txn ids in the same range for both the insert and delete deltas, irrespective of how many insert\_deltas and delete\_deltas were there for that range. For example, in this case once the minor compaction had decided that it had to compact for txn range 1-3 it produced both **delta\_0000001\_0000003** and **delete\_delta\_0000001\_0000003**  irrespective of the fact that the delete\_delta\_01\_01 and delta\_03\_03 did not exist. Even if there were no delete\_delta, the minor compaction would still produce the file **delete\_delta\_0000001\_0000003** . This is a simplifying behavior for now and may be changed in the future.

* **ALTER TABLE acidtbl COMPACT ‘MAJOR’**
* **Run cleaner**

**/warehouse/acidtbl/base\_0000003/bucket\_00000**

{"operation":0,"originalTransaction":2,"bucket":0,"rowId":0,"currentTransaction":2,"row":{"a":3,"b":-4}}

{"operation":0,"originalTransaction":4,"bucket":0,"rowId":0,"currentTransaction":4,"row":{"a":5,"b":6}}

Major compaction works exactly same as the previous version.

## 