August 23, 2017

Update Hive Tables the Easy Way Part 2

by Carter Shanklin

Thank you for reading part 1 of a 2 part series for how to update Hive Tables the easy way. This is part 2 of the series.

MANAGING SLOWLY CHANGING DIMENSIONS

In Part 1, we showed how easy it is update data in Hive using SQL MERGE, UPDATE and DELETE. Let's take things up a notch and look at strategies in Hive for managing slowly-changing dimensions (SCDs), which give you the ability to analyze data's entire evolution over time.

In data warehousing, slowly-changing dimensions (SCDs) capture data that changes at irregular and unpredictable intervals. There are several common approaches for managing SCDs, corresponding to different business needs. For example you may want to track full history in a customer dimension table, allowing you to track the evolution of a customer over time. In other cases you don't care about history but need an easy way to synchronize reporting systems with source operational databases.

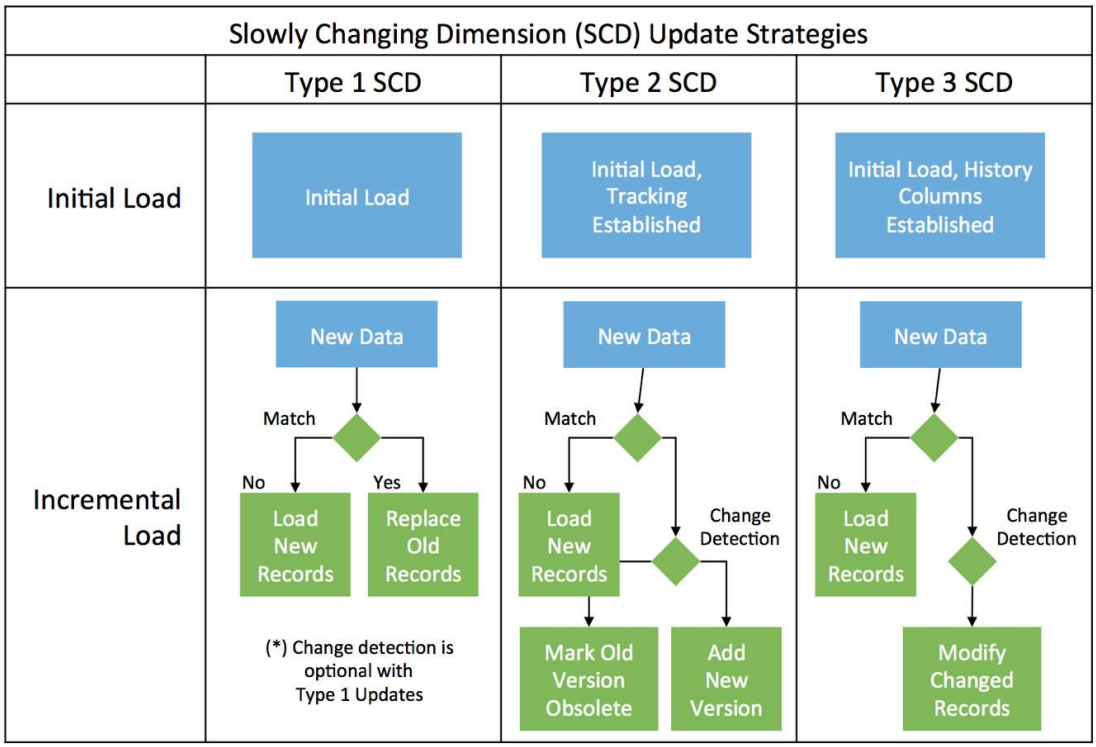
The most common SCD update strategies are:

> Type 1: Overwrite old data with new data. The advantage of this approach is that it is extremely simple, and is used any time you want an easy to synchronize reporting systems with operational systems. The disadvantage is you lose history any time you do an update.
>
> Type 2: Add new rows with version history. The advantage of this approach is that it allows you to track full history. The disadvantage is that your dimension tables grow

Type 3: Add new rows and manage limited version history. The advantage of Type 3 is that you get some version history, but the dimension tables remain at the same size as the source system. You also won't need to create additional reporting views. The disadvantage is you get limited version history, usually only covering the most recent 2 or 3 changes.

This blog shows how to manage SCDs in Apache Hive using Hive's new MERGE capability introduced in HDP 2.6. All of the examples here are captured in a GitHub repository for easy reproduction on your Hadoop cluster. Since there are so many variations for managing SCDs, it's a good idea to refer to standard literature, for example The Data Warehouse Toolkit, for additional ideas and approaches.

HIVE- OVERVIEW OF SCD STRATEGIES



Hive Overview of SCD Strategies

GETTING STARTED: COMMON ELEMENTS

All of these examples start with staged data which is loaded as an external table, then copied into a Hive managed table which can be used as a merge target. A second external table, representing a second full dump from an operational system is also loaded as another external table. Both of the external tables have the same format: a CSV file consisting of IDs, Names, Emails and States. The initial data load has 1,000 records. The second data load has

records. It is up to the various merge strategies to capture both these new and changed records. If you want to follow along, all data and scripts are on the GitHub repository.

TYPE 1 SCD

Since Type 1 updates don't track history we can import data into our managed table in exactly the same format as the staged data. Here's a sample of our managed table.

ID	Name	Email	State
1	Dr. Iza Gerhold	loragutkowski@yahoo.com	NJ
2	Katharyn Goyette DVM	sadiewunsch@hotmail.com	VI
3	Nikolas Tromp	danniekemmer@yahoo.com	VI
4	Ms. Shawnna Gerlach DVM	reichelson@hotmail.com	MP

Type 1 Initial Managed Table

The Merge SQL Code for Type 1 updates is extremely simple, if the record matches, update it; if not, add it.

```
merge into

 contacts_target

using

 contacts_update_stage as stage

on

 stage.id = contacts_target.id

when matched then

 update set name = stage.name, email = stage.email, state = stage.state

when not matched then
```

Let's see what this does for a particular record that changes, Record 93:

ID	Name	Email	State
93	Tosha Parisian	arline72@hotmail.com	IL

ID 93 Before Type 1 Merge

ID	Name	Email	State
93	Tosha Parisian	junie45@price.info	CA

ID 93 After Type 1 Merge

The important things to emphasize here is that all inserts and updates are done in a single pass with full atomicity and isolation to upstream SQL queries, plus automated rollback if failures occur. Guaranteeing all these properties with legacy SQL-on-Hadoop approaches is so difficult that hardly anyone has put them into practice, but Hive's MERGE makes it trivial.

TYPE 2 SCD

Type 2 updates allow full version history and tracking by way of extra fields that track the current status of records. In this example we will add start and end dates to each record. If the end date is null, the record is current. Again, check out the GitHub for details of how to stage data in.

Type 2 Initial Managed Table

ID	Name	Email	State	ValidFrom	ValidTo
1	Dr. Iza Gerhold	loragutkowski@yahoo.com	NJ	2017-01-01	null
2	Katharyn Goyette DVM	sadiewunsch@hotmail.com	VI	2017-01-01	null
3	Nikolas Tromp	danniekemmer@yahoo.com	VI	2017-01-01	null
4	Ms. Shawnna Gerlach DVM	reichelson@hotmail.com	MP	2017-01-01	null

Type 2 Initial Managed Table

We'll use a single-pass Type 2 SCD which completely isolates concurrent readers against in-flight updates, meaning that for changes we want to update the existing record to mark it obsolete and insert a net new record which will be the current record

```
merge into contacts_target
using (
 — The base staging data.
 select
contacts_update_stage.id as join_key,
contacts_update_stage.* from contacts_update_stage
 union all
— Generate an extra row for changed records.
 — The null join_key forces records down the insert path.
 select
  null, contacts_update_stage.*
 from
  contacts_update_stage join contacts_target
  on contacts_update_stage.id = contacts_target.id
 where
  ( contacts_update_stage.email <> contacts_target.email
    or contacts_update_stage.state <> contacts_target.state )
  and contacts_target.valid_to is null
) sub
on sub.join_key = contacts_target.id
when matched
```

```
 then update set valid_to = current_date()

when not matched

 then insert

 values (sub.id, sub.name, sub.email, sub.state, current_date(), null);
```

The key thing to recognize is the using clause will output 2 records for each updated row. One of these records will have a null join key (so will become an insert) and one has a valid join key (so will become an update). If you read Part 1 in this series you'll see this code is similar to the code we used to move records across partitions, except using an update rather than a delete.

Let's see what this does to Record 93.

ID	Name	Email	State	ValidFrom	ValidTo
93	Tosha Parisian	arline72@hotmail.com	IL	2017-01-01	null

ID 93 Before Type 2 Merge

ID	Name	Email	State	ValidFrom	ValidTo
93	Tosha Parisian	arline72@hotmail.com	IL	2017-01-01	2017-07-24
93	Tosha Parisian	junie45@price.info	CA	2017-07-24	null

ID 93 After Type 2 Merge

We have simultaneously and atomically expired the first record while adding a new record with up-to-date details, allowing us to easily track full history for our dimension table.

TYPE 3 SCD

Type 2 updates are powerful, but the code is more complex than other approaches and the dimension table grows without bound, which may be too much relative to what you need. Type 3 SCDs are simpler to develop and have the same size as source dimension tables, but only offer partial history. If you only need a partial view of history, Type 3 SCDs can be a good compromise.

For this example we will only track the current value and the value from one version prior, and will track the version in the same row. Here's a sample:

ID	Name	Email	LastEmail	State	LastState
1	Dr. Iza Gerhold	loragutkowski@yahoo.com	loragutkowski@yahoo.com	NJ	NJ
2	Katharyn Goyette DVM	sadiewunsch@hotmail.com	sadiewunsch@hotmail.com	VI	VI
3	Nikolas Tromp	danniekemmer@yahoo.com	danniekemmer@yahoo.com	VI	VI
4	Ms. Shawnna Gerlach DVM	reichelson@hotmail.com	reichelson@hotmail.com	MP	MP

Type 3 Initial Managed Table

When an update comes, our task is to move the current values into the "last" value columns. Here's the code:

```
merge into
 contacts_target
using
 contacts_update_stage as stage
on stage.id = contacts_target.id
when matched and
 contacts_target.email <> stage.email
 or contacts_target.state <> stage.state — change detection
 then update set
 last_email = contacts_target.email, email = stage.email, — email history
 last_state = contacts_target.state, state = stage.state  — state history
when not matched then insert
 values (stage.id, stage.name, stage.email, stage.email,
 stage.state, stage.state);
```

You can see this code is very simple relative to Type 2, but only offers limited history. Let's

ID	Name	Email	LastEmail	State	LastState
93	Tosha Parisian	arline72@hotmail.com	arline72@hotmail.com	IL	IL

ID 93 Before Type 3 Merge

ID	Name	Email	LastEmail	State	LastState
93	Tosha Parisian	junie45@price.info	arline72@hotmail.com	CA	IL

ID 93 After Type 3 Merge

A SIMPLER CHANGE TRACKING APPROACH

If you have many fields to compare, writing change-detection logic can become cumbersome. Fortunately, Hive includes a hash UDF that makes change detection simple. The hash UDF accepts any number of arguments and returns a checksum based on the arguments. If checksums don't match, something in the row has changed, otherwise they are the same.

For an example, we'll update the Type 3 code:

```
merge into

 contacts_target

using

 contacts_update_stage as stage

on stage.id = contacts_target.id

when matched and

 hash(contacts_target.email, contacts_target.state) <>

   hash(stage.email, stage.state)

 then update set

 last_email = contacts_target.email, email = stage.email, — email history

 last_state = contacts_target.state, state = stage.state  — state history
```

```
values (stage.id, stage.name, stage.email, stage.email,

stage.state, stage.state);
```

The benefit is that the code barely changes whether we're comparing 2 fields or 20 fields.

CONCLUSION:

SCD management is an extremely import concept in data warehousing, and is a deep and rich subject with many strategies and approaches. With ACID MERGE, Hive makes it easy to manage SCDs on Hadoop. We didn't even touch on concepts like surrogate key generation and checksum-based change detection, but Hive is able to solve these problems as well. The code for all these examples is available on GitHub and we encourage you to try it for yourself on the Hortonworks Sandbox or Hortonworks Data Cloud.

Tags:

CLOUD | HIVE | BUSINESS VALUE OF HADOOP | MERGE CAPABILITY | SLOWLY CHANGING DIMENSIONS