Inevitably database schema will change over time. This document discusses how we are planning to address it. In essence:

- avoid heavy changes that alter shape of large tables (e.g. by preallocating space)
- rely on partitioning to scale and run necessary schema changes fast and scalably in parallel

Data Release Catalog

Each data release is independent, therefore the database schema can easily change between data releases; we do not anticipate any non-trivial challenges here. Not only are there unlikely to be large changes to existing tables and columns, but the previous data release will always be available for people who cannot yet adjust to the new schema.

Each published data release will be frozen, i.e. to guarantee result reproducibility, changes that might impact query results will not be allowed. Allowed changes include changes such as adding new columns, changing precision (without losing information, e.g. float --> double), or adding/removing/resorting indexes.

New columns: we are planning to keep a few extra unused columns of each common type (for example, 3 FLOATs, 3 DOUBLEs, 3 INTs, 3 BIGINTs) for each large table, and use one of them when a new column is needed. (renaming a column is a trivial and instantaneous operation). Speed of filling new columns should be comparable to speed of a full table scan.

Updating columns. Speed of updating values should be comparable to speed of a single shared scan.

Deletes. If we have to delete a column, instead of deleting it, (which is expensive as it changes shape of a table), we will add it to the pool of "extra, unused columns" by renaming it.

The unsed/hidden columns will be hidden through non-materialized views. It is likely that we will end up providing multiple views, e.g. each time we make a schema change, we'd expose the changes through a new view.

We strongly encourage users to not rely on order of columns, or number of columns returns from "SELECT *" - obviously a query that expects x number of columns returned from "SELECT *" will fail if we add a new column. Introducing new view for each change will alleviate this problem.

Up-to-date Catalog

We are planning to maintain two copies: one live used by alert production pipeline, and one for user queries. The two will be synchronized daily during downtime (daytime). The up-to-date catalog will not contain the largest tables (Source and ForcedSource), the largest table will be the Object table. Because it is significantly smaller, even more complex changes can be done fast, and will complete within the ~12h downtime window. Typical scenario:

- night "X": db "A" used by alert production, db "B" used for user queries
- day "X": db "A" made available for user queries, db "B" taken offline and brought up to date (the observations from night "X" entered. Schema evolution on db "B" as needed (eg adding new column by renaming a dummy extra column from the reserved pool, etc)
- night "X+1": db "B" used by alert production, db "A" continues to be used by user queries
- day "X+1": db "B" made available for user queries, db "A" taken offline and brought up to date (schema evolution same as on db "A" the previous night, then the observations from night "X+1"

entered)

If we start running low on the extra columns in the reserved pool, we can refill it slowly during day time, a subset of chunks at a time. Since it is hidden, it it ok if some chunks have it and other don't. This is the worst case, assuming we don't have enough time during one day to add columns to all chunks.

Merging DR into Up-to-date

By the time we produce a DR catalog, the up-to-date will significantly diverge from the version used to produce DR. Merging the DR into up-to-date is non-trivial, as it will break reproducibility. [to be continued...]

Notes on Scalability

Non trivial schema changes on multi-billion row table tend to take extremely long time (measured in days). In our case, each large table will be partitioned into several thousands chunks, which makes the updates scalable and faster: not only the updates can be done in parallel on multiple machines, but each partition is small enough to be handled efficiently without running into any bottlenecks or problems with lack of buffer space.

Notes on Administration

All queries accessing the database will go through our interface, which intercepts and interprets the queries. This will allow us to put the entire system in special mode (user queries will be queued, administrative queries will continue to work). The administrative tools we are building will allow us to push the necessary updates to all nodes, and we could use the existing scheduling mechanisms to run the upgrade in the most efficient fashion. The admin tools will also allow to run necessary verifications (eg. whether the schema for all table chunks matches).

Tests

See db/tests/SchemaEvolution