[Design Doc] Liquid Clustering

Author: Jiaheng Tang
Date: Nov 2, 2023

Motivation

This design doc proposes Liquid clustering, a new, flexible, and incremental clustering mechanism for Delta. Users no longer need to specify the clustering columns like OPTIMIZE ZORDER BY requires; instead, clustering columns are specified during table creation and persisted in table metadata. Liquid also allows users to change the clustering columns to adapt to the workload. After changing the clustering columns, existing data is not re-clustered, and only newly ingested data is clustered by new clustering columns.

Partitioning/clustering is a common technique to manage datasets to reduce unnecessary data processing. Hive-style partitioning and ZORDER clustering are existing solutions in Delta, but both have limitations.

Hive-style partitioning

Hive-style partitioning clusters data such that every file contains exactly one distinct combination of partition column values. Although Hive-style partitioning works well if tuned correctly, there are limitations:

- Since partition values are physical boundaries for files, Hive-style partitioning by high cardinality columns will create many small files that cannot be combined, resulting in poor scan performance.
- In Spark/Delta, once a table is partitioned, its partition strategy cannot be changed, thus being unable to adapt to new use cases such as query pattern changes, etc.

ZORDER clustering

ZORDER is a multi-dimensional clustering technique used in Delta. The `OPTIMIZE ZORDER BY` command applies ZORDER clustering and improves the performance of queries that utilize `ZORDER BY` columns in their predicates. However, it has the following limitations:

- `OPTIMIZE ZORDER BY` is an operation that rewrites all data in the table, resulting in high write amplification. Also, no partial results are saved when execution fails.
- `ZORDER BY` columns are not persisted and the user is required to remember the previous `ZORDER BY` columns, often causing user errors.

Terminology

- **Liquid clustering**: A clustering technique that utilizes the Hilbert curve and ZCube to support incremental clustering.
- **Write amplification**: Signifies rewriting the same rows multiple times.

Requirements

Functional requirements

MUST:

- User Surface
    - Users can define clustering columns in any order when creating Liquid tables.
    - Users can change or remove clustering columns on existing Liquid tables via `ALTER TABLE CLUSTER BY`.

    - Users can specify up to 4 clustering columns by default.
    - Users can rename clustering columns on existing Liquid tables if column mapping is enabled.
    - Users can manually invoke liquid clustering via the `OPTIMIZE` command.
    - Clustering columns are exposed to users via the `DESCRIBE DETAIL` command.
    - Users can create Liquid tables using `CREATE TABLE LIKE`.
- Clustering
    - Liquid clustering must cluster ingested data incrementally.
    - Liquid clustering must be applied with the current clustering columns.
    - Liquid clustering should produce good file sizes, and it should respect the target file size set for OPTIMIZE.
    - After altering clustering columns, Liquid clustering should be applied with new clustering columns only to newly ingested data, not to the existing data before altering.
- Other
    - Must prevent the older Delta writers or external writers that are not aware of tables with Liquid clustering from running `OPTIMIZE ZORDER BY` on tables with Liquid clustering.

User surface

Liquid clustering introduces the following new SQL syntaxes:

SQL Syntax	Description

`CREATE TABLE <table>`
 	
`USING delta`
 	
`CLUSTER BY (<col1>, <col2>, …)`
	Creates a Delta table such that the columns specified with `CLUSTER BY` are used as Liquid clustering columns.

`ALTER TABLE <table>`
 	
`CLUSTER BY (<col1>, <col2>, …)`
	Alters the Liquid clustering columns. Data ingested after altering and data not yet clustered is clustered with new clustering columns.

`ALTER TABLE <table>`
 	
`CLUSTER BY NONE`
	Removes the Liquid clustering columns. No clustering is performed for future ingestions.

`OPTIMIZE <table>`	Triggers Liquid clustering. Unlike the existing `OPTIMIZE` semantics - which is compacting files, `OPTIMIZE` on Liquid tables triggers compaction if there are no clustering columns set, or clustering otherwise. `OPTIMIZE ZORDER BY` is not allowed for clustered tables.

Liquid clustering introduces the following new DeltaTable APIs:

API name	Description
`clusterBy(colNames: String*): DeltaTableBuilder`	Creates a Delta table such that the columns specified with `CLUSTER BY` are used as Liquid clustering columns.

We will work on the DataFrameWriter API when `clusterBy()` is available in OSS Spark (WIP).

Proposal Sketch

Better clustering with Hilbert curves

Today, Delta supports `OPTIMIZE ZORDER BY` command that utilizes the Z-Order curve (design doc) for data clustering. We propose to use Hilbert curve, a continuous fractal space-filling curve as a multi-dimensional clustering technique for Liquid, which significantly improves data skipping over ZORDER.

Similar to how ZORDER works, Hilbert curve maps multidimensional data onto a 1D space by fitting them on the curve. This preserves the locality very well meaning points that are close in the 1D space should also be close in multi-dimensional space. We can leverage this property to do efficient data skipping. Take the following example of a simple two-column table, with 64 distinct records. Each dotted rectangle represents a single file, and each file contains 4 records.

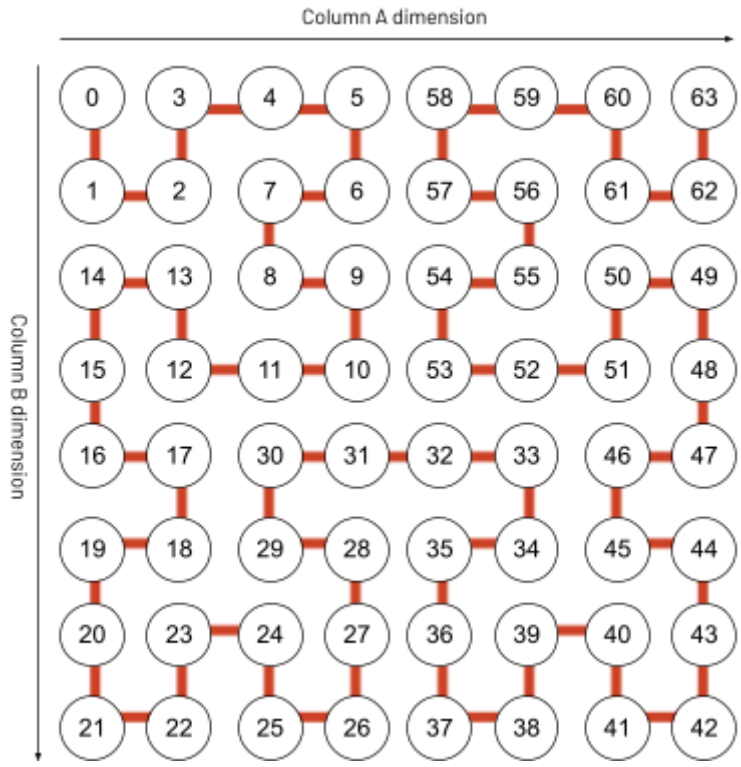

Fig 1: Adjacent points on the curve always have distance = 1

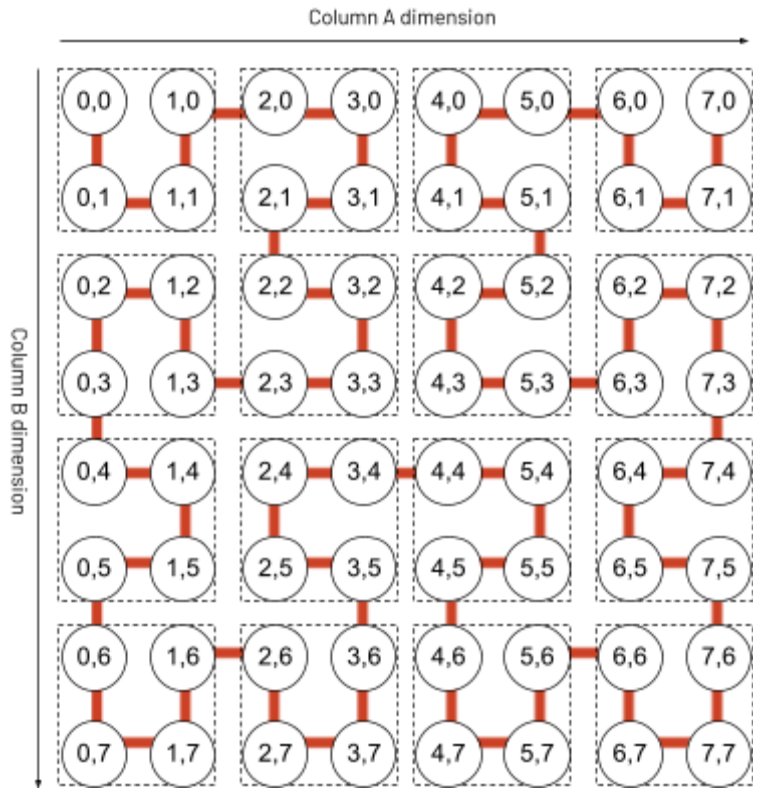

Fig 2: Partition the data by points on the curve and pack them into files

In this example, we first translate the two columns A and B into a numeric range of [0, 7] by range partitioning the value distribution. The Hilbert curve gives us a nice property that adjacent points *on the curve* always have a distance of 1. To make use of this, we partition the data by the points of the curve, and then pack the nearby points into good-size files. This means each file contains points that are close to each other on the curve, which means they'll have close min/max ranges for each of the clustering dimensions.

Why is the Z-curve worse than Hilbert?
Remember the nice property of the Hilbert curve (see Fig 1)? The Z-curve doesn't have that property. Z-curve's adjacent points don't always have distance = 1 and it has large jumps in the curve. Those jumps translate into nonlinear (and potentially huge) increases in bounding box size for sub-ranges of the curve. For instance, let's look at the same orange line of length 6. For Z-curve, the bounding box shown in Fig 3 covers the entire space! For the Hilbert curve, a range of this length could only cover half the space (as shown in Fig 4), because it could cover only two adjacent quadrants. This means for Z-Order, potentially some files will have min/max range equal to the full range, and data skipping can't skip these files.

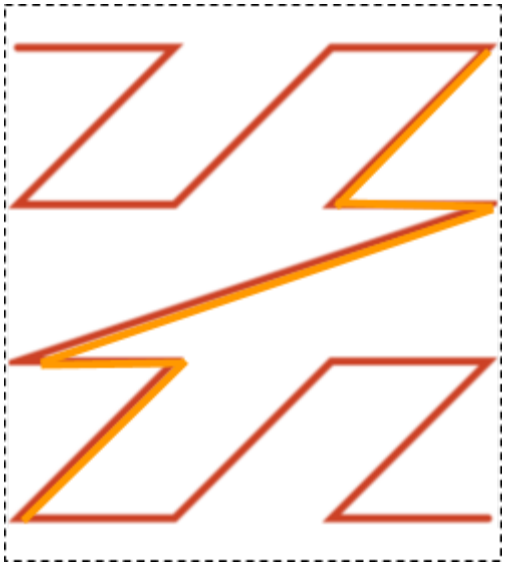

Fig 3: Z-curve: The bounding box for the orange line spans the entire space

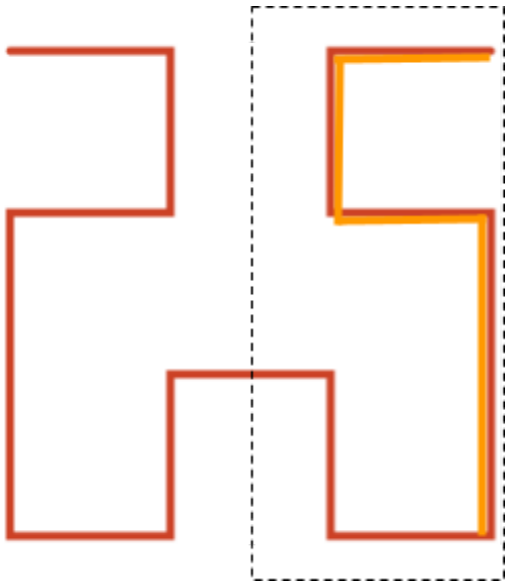

Fig 4: Hilbert Curve: The bounding box for the orange line spans only half the space

Incremental clustering using ZCube

Currently, OPTIMIZE ZORDER BY requires rewriting all data, even if no new files have been added since the last time. That makes it very expensive to run it on a large table. Also, when it fails we lose all the progress, and the next run will need to start from scratch.

We propose to introduce incremental clustering capability with Liquid, which allows users to run OPTIMIZE without rewriting all data. OPTIMIZE will also be completed in batches,

where each batch will produce a single OPTIMIZE commit, so that not all progress is lost when something goes wrong.

Incremental clustering is built around the concept of ZCubes. A **ZCube** is a group of files produced by the same OPTIMIZE job. Since we only want to rewrite fresh/unoptimized files, we distinguish between already optimized files (that are part of some ZCubes) from unoptimized files using the `ZCUBE_ID` tag in AddFile. We'll generate a unique ZCUBE_ID using UUID for each new ZCube.

There are a few strategies to pick which files to cluster, and more details can be found here.

We'll introduce two configs to provide flexibility for controlling which files to consider for clustering:

- **MIN_CUBE_SIZE**: ZCube size for which new data will no longer be merged with it during incremental OPTIMIZE. Defaults to 100 GB.
- **TARGET_CUBE_SIZE**: target size of the ZCubes we will create. This is not a hard max; we will continue adding files to a ZCube until their combined size exceeds this value. This value must be greater than or equal to MIN_CUBE_SIZE. Defaults to 150 GB.

We call any ZCubes with a size less than *MIN_CUBE_SIZE* a **partial ZCube**. All new files will be considered for OPTIMIZE but we also consider any existing partial ZCubes. Once a partial ZCube accumulates enough data and crosses the *MIN_CUBE_SIZE* threshold, it becomes a **stable ZCube**, at which point we'll no longer consider it for rewriting.

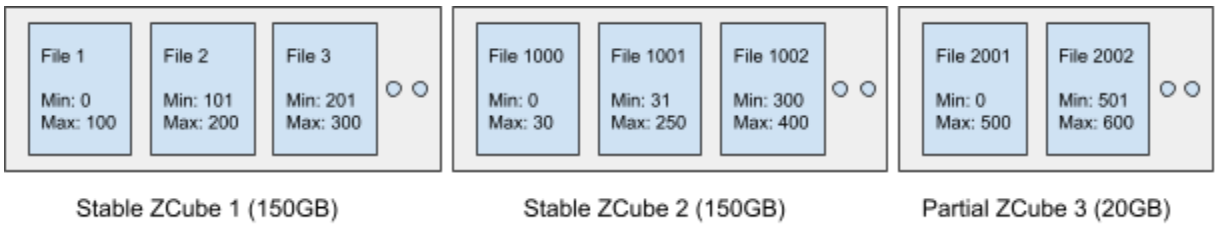

Fig 5: ZCube-based incremental clustering

Stable ZCubes may become partial again if DML operations delete too many files, at which point files for partial ZCubes will be considered for clustering.

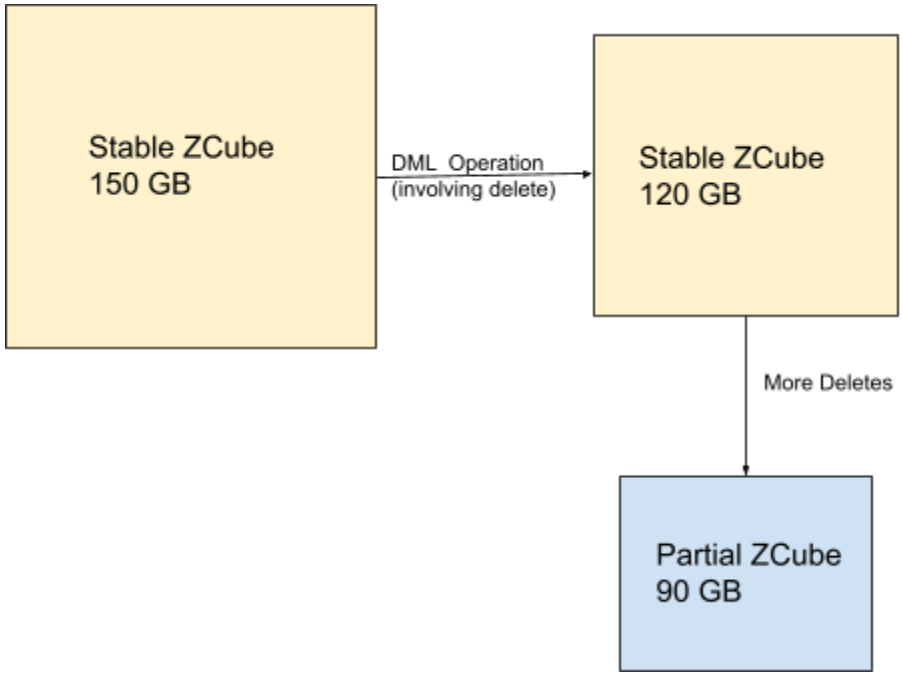

Fig 6: Stable ZCube dropping into partial status after deletes

One special case is when users change the clustering columns, we want to keep the old data untouched, and from that point on only cluster the new data using the new clustering columns. Therefore, we will persist the clustering columns in AddFile using the `ZCUBE_ZORDER_BY` tag to indicate which clustering columns these files are clustered to. When picking candidate files, we'll filter out files with a different set of clustering columns.

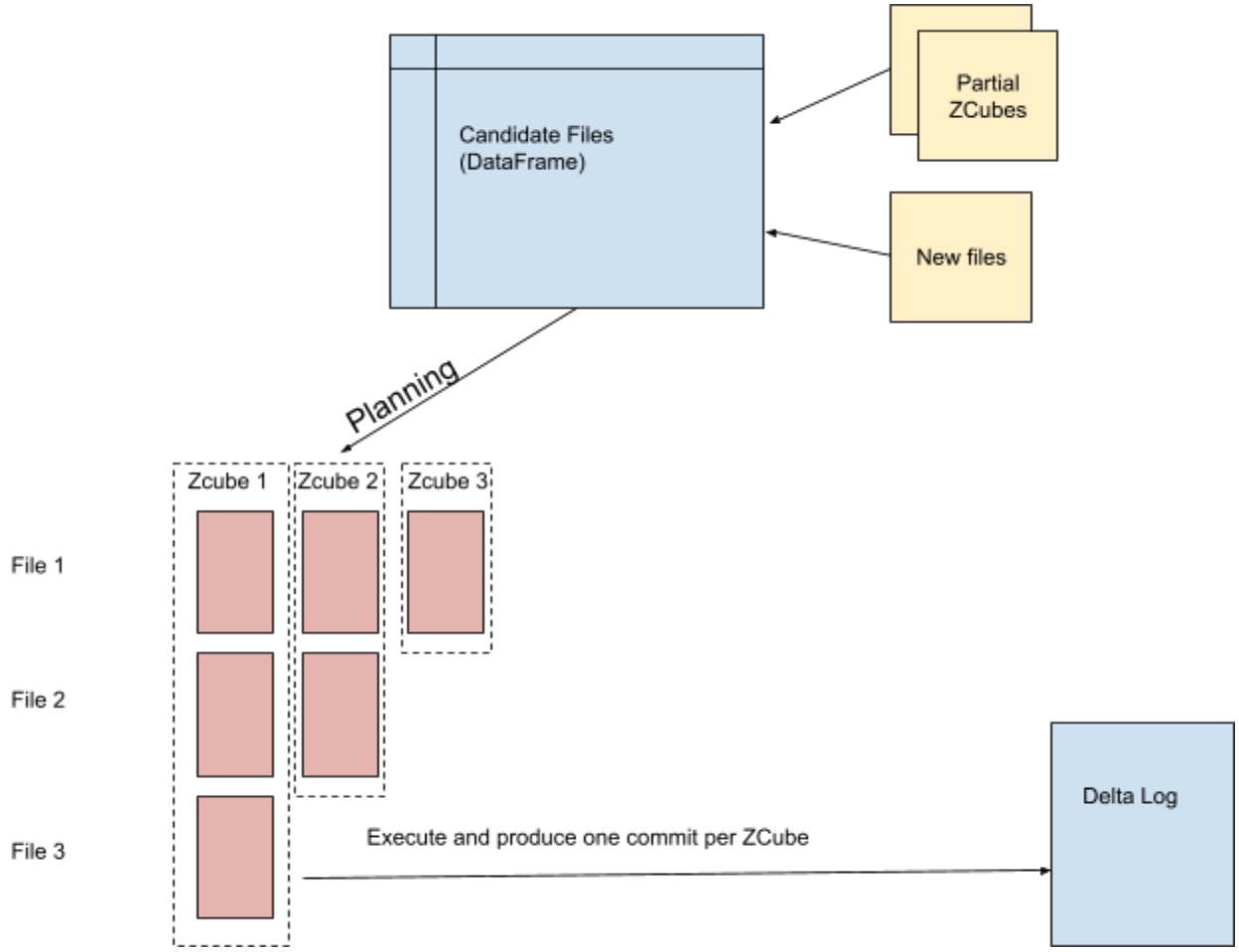

Fig 7: Flow of OPTIMIZE using ZCube

The illustration above shows the flow of `OPTIMIZE` using ZCubes. To begin, we select our candidate files, which encompass both fresh data and any partial ZCubes with the same clustering columns as our current clustering columns. Then we bin pack the files into multiple ZCubes and run the clustering algorithm on each of the ZCubes. After execution, we'll produce one commit per ZCube to make sure partial results are saved during an event of crash.

User surface

Users no longer need to specify the `ZORDER BY` columns. Instead, clustering columns are specified during table creation using the following syntax.

```
Unset
CREATE TABLE <table> USING delta CLUSTER BY (<col1>, <col2>, …)
```

Then, the clustering columns will be saved in the metadata and users can just run the `OPTIMIZE` command to trigger clustering.

Users can also run ALTER TABLE CLUSTER BY to change the clustering columns. There are two variants:

1. ALTER TABLE CLUSTER BY (col1, col2): this will update the clustering columns. Already clustered data won't be touched and new data will be clustered by the new clustering columns.
2. ALTER TABLE CLUSTER BY NONE: this will remove any clustering columns. Already clustered data won't be touched and new data will not be clustered. Instead, `OPTIMIZE` will compact the new data, similar to the existing `OPTIMIZE` which runs file compaction.

Metadata

Liquid clustering will utilize Domain Metadata (design doc) to store the clustering columns as a `DomainMetadata` action, which is a part of the Clustering table feature(PR):

```
JavaScript
{
  "domainMetadata": {
    "domain": "delta.clustering",
    "configuration": "{\"keys\":\"colA,colB\"}"
  }
}
```

Liquid clustering also supports changing, and if column mapping is enabled, renaming clustering columns. Note that we do not allow dropping clustering columns, but users can always do ALTER TABLE CLUSTER BY to move columns out of the clustering columns and then drop them. To support column mapping, we store columns' physical names in the metadata and extract the logical column names for any user-facing commands like `DESCRIBE DETAIL`.

Protocol Change

Liquid clustering leverages the `ClusteringTableFeature`, so no additional changes to the Delta protocol.

The design decision for depending on ClusteringTableFeature can be found here.

Design Decisions

Decision 1: Should we rewrite any already-optimized files?

Option 1: Always rewrite all files

This means keeping only 1 ZCube and for any OPTIMIZE run, merge the existing ZCube with unoptimized data. This is the current implementation and has a long OPTIMIZE time and large write amplification.

Option 2: Only optimize fresh data

This means never merging with existing ZCubes. However, many small ZCubes are bad for data skipping effectiveness. Although within each ZCube data skipping rate would be similar, since there's no total ordering across ZCubes, we'll need to perform data skipping for each ZCube and the scan time will be linear to the number of ZCubes.

Option 3 (Recommended): Use minimum cube size threshold

In this approach, all new files are merged but also any existing cubes with size smaller than a given threshold. This gives us lower write amplification than Option 1 while still keeping the number of ZCubes lower than Option 2, which provides better read performance than Option 2.

Decision 2: Should we introduce a Liquid-specific table feature?

Option 1: No, but depend on ClusteringTableFeature (Recommended)

By leveraging ClusteringTableFeature(PR), we can prevent older Delta Lake versions and external writers from running `OPTIMIZE ZORDER BY` on Liquid tables, which will invalidate the clustering maintained by Liquid tables.

We'll follow the ClusteringTableFeature's spec by tagging OPTIMIZE's output files with `CLUSTERED_BY` set to `liquid`.

Option 2: No

No additional table feature that external readers/writers need to support. However, writers can invalidate the clustering on Liquid tables by running OPTIMIZE ZORDER BY.

Option 3: Yes, introduce a LiquidTableFeature

This will have the same benefit as option 1, but with the additional overhead of introducing another table feature. ClusteringTableFeature is built for this exact purpose, to serve as a table feature for any kind of clustering implementation.