What have I learned from doing Merge-On-Read PoC

In the document [1], we've discussed the high-level design about the three solutions: Eager (Copy On Write), Lazy with Natural Keys (Merge On Read), Lazy with Synthetic Keys (Merge On Read). The three of them have their different pros & cons. I think we may need to abstract the general Iceberg interfaces and framework, so that the implementations can be pluggable and can be adapted to different scenarios.

Here, we would focus on merge-on-read ways, also called *lazy with natural keys (NRI)* or *synthetic keys (SRI)*. For most user cases in big data areas, we don't have the unique key(s) in a given data set which can be defined as the natural keys, so we have to choose the *lazy with synthetic keys* implementation. While for some other cases described in mail list[4], such as syncing the row-level binlog into Iceberg table, it may be better to use the NRI ways, because almost all of the RDBMS users have defined their primary/unique keys in their tables, the primary/unique keys can be used as the natural keys, and we can just write the unique keys into iceberg's differential files without pre-fetching the row identifier as the SRI way, finally the ingest latency will be quite better. So I think the merge-on-read update/delete solution should try to support both NRI and SRI.

In passed days, I've written some demo (PR is here [5]) to do the merge-on-read PoC, and I think the merge-on-read work can be divided into five parts:

- Define the table format specs:
  - Columns of insert data files and delete diff files.
  - Schema of DataFile/Manifest.
  - SortOrder spec
  - Unique key spec.
- Implement the read path inside Iceberg
  - How to find the delete differencial files when iterating the metadata iterator lazily
  - Prune the insert data files and delete differential files
  - How to JOIN the insert data file with its delete diff files efficiently
- Implement the write path inside Iceberg
  - Write the data file & differential file
  - Construct the manifest & snapshot and commit.
  - How to handle the CDC changes.
- Integrate the iceberg read&write interfaces with spark or flink.
  - How to define the suitable Iceberg API for compute engine
  - Rewrite the Update/Delete SQL to SELECT for getting all the row keys, then write them into relative delete differential files.
  - Update/Delete SQL translate into physical plan, concat plan with iceberg.
- Compaction
  - Minor/Major,

    - When/Who/How triggered ?
    - Minimize the transaction conflicts with the normal write conflicts.
    - Don't change the logical table data, shouldn't be consumed by downstream consumers.

I didn't write any codes to verify the last two things (spark integration & compaction), so will mainly focus on the first three parts.

Table Format Specs

Q1. Columns of insert data files and delete differential files.

Let's take the example in UT, assume we have the table (Actually I used this table in my demo unit test):

```
private static final Schema SCHEMA = new Schema(
      required(1, "id", Types.LongType.get()),
      required(2, "data", Types.StringType.get())
  );
```

As the upsert/delete document[1] says, for Lazy SRI we will need two extra columns: ***file_id*** and ***row_offset***, which identify those rows. The first thought in my mind is: we don't have to define the file_id and row_offset explicitly in insert/append data files because the rows are written in parquet or avro format one by one and its offset is starting from 0 to N-1. Soon I found that the parquet didn't provide the direct API to return the implicit row index (0~N-1) when searching the given column values, but it provided an API to get the matched row's row group and pages etc(also can read the given row group by offsets). Anyway making the row_offset to be implicit should be possible but I don't struggle with it in the demo, I defined an explicit row_offset column in the underlying data file.

Another thing is about the ***file_id***, should it be a logical column or physical column in delete differential files ? If a given delete differential file can only belong to one data file, then we can just remember the insert data file's file_id in metadata, and it will be a logical column. Otherwise it should be a physical column in the differential file. We can compare the two solutions in the following table (let's say the first solution as 1-1, the second one as 1-N):

	1-1	1-N
Small files	Produce many delete diff files because each matched data file	Will have some, but less than 1-1.

	will have at least one delete dif file in a write transaction.	
Extra column in delete file.	No need to add the physical file_id column, only keep it in metadata.	Will have a physical file_id column in the delete differential files.
Read Path	For each data file, it will have an exclusive delete diff file set, mark them as <ins, del0, ..,delN>>. We K-merge sort them based on the sorted row_offset in SRI. For NRI we need the nested-loop JOIN.	For each data file, it will have a shared delete diff file set (means share some delete files with other data files). K-merge sort them based on the sorted <file_id, row_offset> pairs in SRI. For NRI, we need the nested-loop JOIN.
Cost to find the overlap delete diff files for a given data file.	Low cost. because only need to filter in the manifest files, and the result file set is exactly the delta changes we're searching.	Low cost, but the delta file set we filter from the manifest files will have some false positive samples, Does it matter ?
Compaction	May more frequent compaction to control the file count.	Less frequent compaction.
Write concurrent	File-level concurrent.	Task-level ? Say if a task has 3 data files, and can write their delete markers into one delete diff file (sorted by row_offset or unique keys) serially.

(I chose the 1-1 solution in my PoC demo because it's easy to implement). While in fact the 1-1 solution wouldn't work for the equality-deletes because we won't scan the given data file and don't know the exact row id(s) of update/delete rows, finally we don't know how to generate the delete differential files for a given data file. The 1-N solution would work for both SRI and NRI: SRI is easy to understand, for NRI we just write the delete differential file which contains a column for each equality predicate, finally the delete differential file would overlap with multiple data files, it's 1-N.

So I think we can define the columns of insert/append data files and delete differential files as following:

1. For Lazy SRI, if the iceberg table has (id, data) columns, then the insert data file should also only have (id, data) columns, the row_offset column is a logical and virtual column. The delete diff file only has the row_offset column. The row_offset field is sorted in both insert data file and delete differential file naturally, we can still do the K-merge sort based

on the ordered row_offset even if compute engines don't sort any column values in the Iceberg table when writing data into iceberg files. NOTICE:

   a. We may need to use O(N) cost to find the row_offset(s) from data files to dump them into delete differential files, if we don't sort any column value in data files.
   b. If we sort a column in data files, and the UPDATE/DELETE query contains the column value, then we will only need O(logN) cost to find the row_offset(s).

2. For Lazy NRI, insert data files will have (id, data) columns, the delete diff files's columns depend one the UPDATE/DELETE query. If we have a table with columns (a,b,c,d), and the equality-delete query performs like: UPDATE table SET (0,1,2,3) WHERE a=0 AND d=3, then we have two solutions:
   a. The delete differential files will have columns (a,d). For each update query, we have to provide all the correct column values to SET, otherwise some of the insert data files will have part of columns inside them.
   b. Another method is putting the matching columns and setting columns together in delete diff files, we will have 6 columns in delete differential files for the UPDATE query (a,d, a,b,c,d), we can consider the first (a,d) pair as matching columns, the following tuple (a,b,c,d) as setting columns. In the read path, we could use (a,d) as the JOIN condition between insert data files and delete differential files to get all the existing rows. For this case, we don't have to provide all the correct column values to SET, say we can execute the query like: UPDATE table SET a=1 WHERE a=0 AND d=3.

   As different delete differential files may have different column combinations, for example, the data files have (a,b,c,d) columns, while some delete diff files have column (a) and others have (c,d) columns, so it's hard to keep the differential files to be sorted by the same key(s) as the insert data files did. We would have to do the nested-loop JOIN between insert data files and delete differential files. Of course, if we could hash the updates/deletes into different buckets and each bucket only has a very small deletes set, then we could cache them into HashSet and use the HASH-JOIN algorithm, as Ryan Blue commented. In my opinion, the size of updates/deletes is unpredictable, we'd better not expect the change set to be small. For example, sinking change logs into an iceberg table, the change logs is an endless stream, which would produce lots of change logs, it may not be suitable for the current design. We can discuss this further.

   Now we prefer the solution(a). Concluded the limits of equality-deletes here so that the users can decide whether it is suitable for their cases, assume the table test with (a,b,c,d) columns:

   - The UPDATE should fill all the columns to be set. the query `update test set (1,2,3,4) where a=0` is supported, while `update test set a=1,b=2 where a=0` isn't supported.
   - The WHERE conditions of UPDATE/DELETE have to be equality predictions connected by OR, AND, IN (NOT is not allowed). such as: a=1 and b IN (2,3), a=1 OR b=23, a=1 AND b=23.

- The equality-deletes should be small so that we can JOIN/HASH-JOIN efficiently, otherwise it will be an inefficient nest-loop JOIN.

Q2. Schema of the DataFile/Manifest schema.

The next thing is designing necessary fields in the DataFile/Manifest schema. In my mind, both insert data files and delete differential files are DataFile although they have different parquet/avro schemas (partition spec should be the same, the delete differential file should be put under the partition of its data file). So I add two fields in DataFile schema (we can discuss whether there is a better solution):

```
optional(134, "file_type", Types.IntegerType.get()),
optional(135, "sequence_number", LongType.get()),
```

- file_type: it could be 0, 1, 2:
  - **0** means the data file is an insert/append data file, not a delete differential file.
  - **1** means the data file is a delete differential file, with <file_id, row_offset> pairs inside it.
  - **2** means the data file is a delete differential file, with several equality columns of the iceberg table inside it.
- sequence_number: each write transaction will increment the sequence number. Say if we have a data file *<insert-data-file-0, sequence-number-0>*, the next transaction which deletes a few rows in *insert-data-file-0* would write a delete differential file *<delete-diff-file-0*, *sequence-number-0* + 1> . Only the delete differential files whose sequence number is larger than *sequence-number-0* may need to JOIN with the *insert-data-file-0*.

The manifest schema should add the sequence_number field, but I'm not sure whether the snapshot should add sequence_number or not (may need to consider this carefully). One important thing we need to notice is: we need to sort the manifest file by the sequence number from newest to oldest in the manifest file list, I will explain the reason in the next *Read Path* section.

As Ryan Blue commented, we could keep a separate set of delete manifests. In that case, we would annotate whether a manifest is a delete manifest or data manifest in the manifest list. Then we could process all delete manifests first. This strategy has some advantages if the set of delete files is small.

Q3. SortOrder specification

As we'll discuss in the following CDC part, we should provide the option to let users define their sorted keys in the Iceberg table if necessary.

We have the pull request for PoC and the issue to track it, Need to push it merge into the trunk.

Q4. Unique key specification

Talked with the guys from NetEase, they have the requirement about the unique keys. They have a lot of RDBMS data with a primary key , and want to sink to an iceberg data lake. But they don't like the iceberg data lake breaking the uniqueness if other data sources also sink fews data into it. so they hope the iceberg can support the unique keys.

Need to discuss which semantics is suitable:

1. Upsert semantis: the later key will always overwrite the previous existing key, similar to the HBase.
2. Reject if existed: The insert would be executed as two steps: a. Get key; b. Insert key if key does not exist, otherwise just reject the insert operation.

If we have an unique key in the Iceberg table, then we don't have to write all the column values of equality predicates into delete differential files. We can just write the unique keys into delete differential files, with sorted or unsorted. There're the following advantages:

1. Only write the unique key column into delete diff files. Save lots of storage.
2. We could JOIN the data files with delete files by unique key, don't have to compare each column values, save lots of CPU.
3. The unique key can be sorted, so that we can depend on the merge-sort JOIN.

Seems the upsert is the easiest way: make the insert as two operations: 1. delete ; 2. insert. then apply them to the whole table. won't be too hard if we've implemented the Merge-On-Read I guess, it can be a separate issue to discuss rather than the Merge-On-Read feature.

Read Path

Q1. How to find the delete differential files for a given data file ?

When implementing the merge-on-read reading path, **the first thing** I encountered is: How to find the delete differential files for each given insert data file efficiently ?

The Iceberg table format has implemented the scalable and efficient metadata. I see the manifest group will dispatch the manifest file filtering task to several threads and each manifest file will iterate the manifest entries lazily (It's unfriendly to put all those manifest entries into memory because of the huge number of parquet/avro in real data-lake production). As we said above, for each insert data file with *sequence-number-0,* we have to find all the delete differential files whose sequence_number is larger than *sequence-number-0* and interval overlap with the data file.

The problem is we're iterating the manifest entries lazily and concurrently, we couldn't get the complete delete differential file set when we are iterating the current manifest entry (can consider it as an insert data file).

If we're iterating the manifest entries lazily from newest sequence number to oldest and at some point we meet the manifest entry whose insert data file is file0, then actually we've seen all the delete differential files whose sequence number is larger than file0's sequence number. We can use a temporary SortedMap to cache the delete differential files, and once encountered an insert data file then we can remove its delete differential files from cache. so the cache won't be too large I guess.

Q2. Prune the insert data files and delete differential files

<TODO>

Q3. How to JOIN the insert data file with its delete diff files efficiently

When doing the merge-on-read read, it will be quite efficient if the original insert data files are sorted by the row keys (logical row_offset in SRI or physical unique keys in NRI). Of course, the prerequisite is: the underlying file formats should provide the efficient reader API such as reader.seekBefore(row_identity).

I provide the pseudo-code here (could still be optimized by heap I think):

```
Iterable mergeSort(DataFile insertFile, List<DataFile> deleteFiles) {
    iterable = Iterables.transform(insertFile,  dataRecord ->{
         isDeleted = false;
          for(deleteFile : deleteFiles){
            If (deleteFile.isEmpty()) continue;
            curRecord = deleteFile.seekBefore(dataRecord);
            if(curRecord == dataRecord){
                // DeleteMarker, should skip this record.
                isDeleted = true;
            }
          }
          return isDeleted ? null : dataRecord;
    });
    return iterable.filter(notNull);
}
```

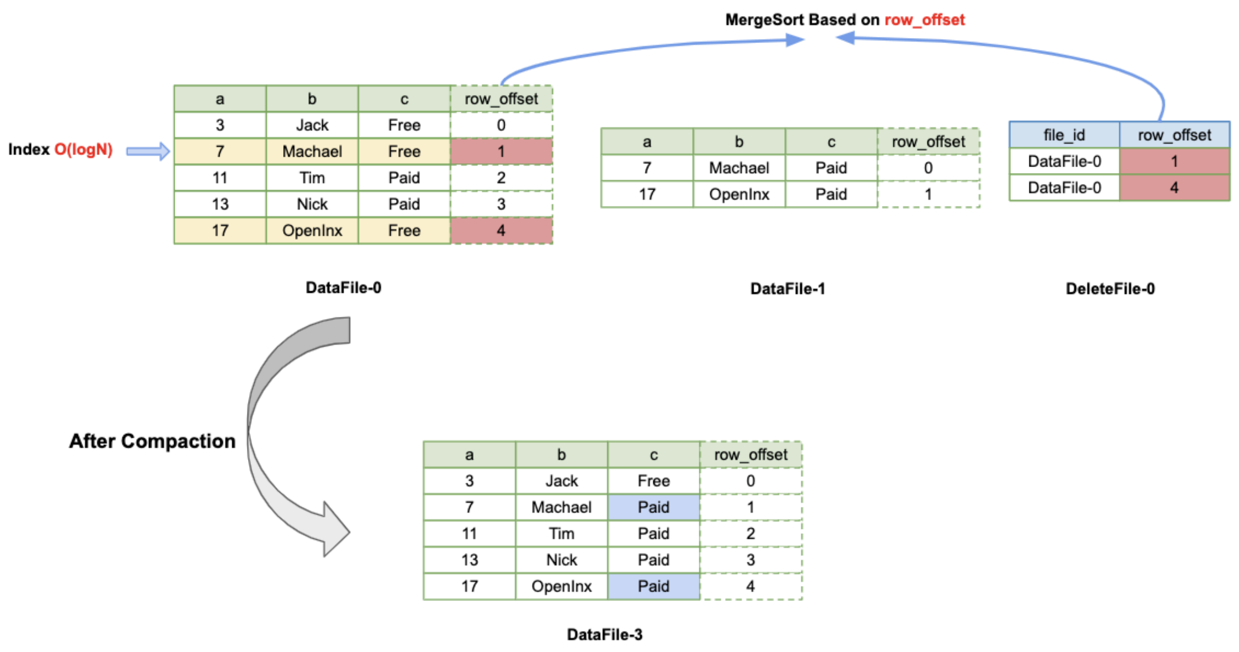
In another case: if records in the insert data files are out of order, we can only do an inefficient JOIN to merge the insert data files with delete diff files.

Write Path

Q1. Write the data file & differential file

> Solution.1: Lazy With SRI

When executing SQL like: `update user set c='Paid' where a in ('7','17')` . we expect that the compute engines will rewrite it to a SELECT and find all the row ids, and then will dump them into several insert data files and delete differential files.



DataFile-0

a	b	c	row_offset
3	Jack	Free	0
7	Machael	Free	1
11	Tim	Paid	2
13	Nick	Paid	3
17	OpenInx	Free	4

DataFile-1

a	b	c	row_offset
7	Machael	Paid	0
17	OpenInx	Paid	1

DeleteFile-0

file_id	row_offset
DataFile-0	1
DataFile-0	4

DataFile-3

a	b	c	row_offset
3	Jack	Free	0
7	Machael	Paid	1
11	Tim	Paid	2
13	Nick	Paid	3
17	OpenInx	Paid	4

As we said above, the (file_id, row_offset) columns don't have to be explicit in the insert data file, but have to be explicit columns in delete differential files.

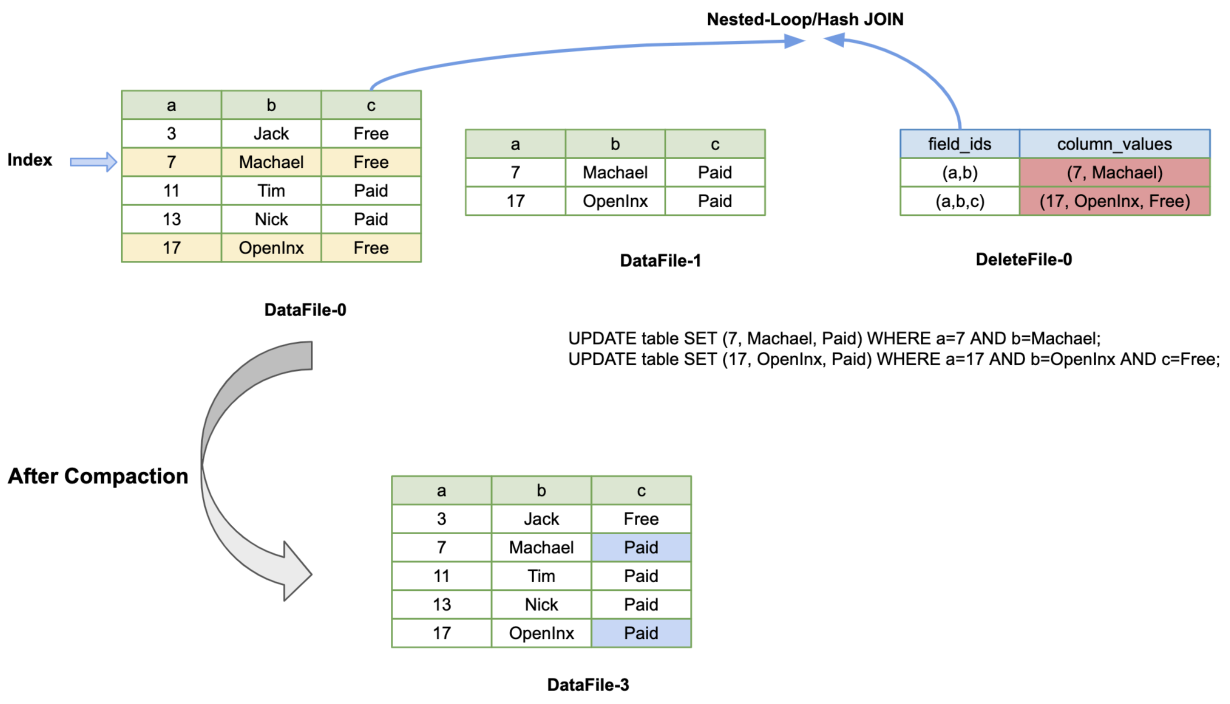
When executing the query, we will do:

Step.1: Prune the scan query into iceberg partitions and find the relative data file set.

Step.2: For each computing task, it will have few data files, let's mark them as <df0, df1,.., dfn>. Inside the task, we should sort the array by file name. Then for each data file, we locate the row_offset by scanning/seeking it. Finally it will generate the sorted <file_id, row_offset> pairs. We can implement this lazily, Mark it as *SRI_Iterator*. NOTICE: here we will generate an insert data file and a delete differential file for each task. Inside the task, we don't have to shuffle to keep the pairs sorted.
Step.3: Iterate the *SRI_Iterator* and write the <file_id, row_offset> pairs into a delete differential file.

In Step.2, if the data file is sorted by other columns (depends on SortOrder specification), then we may locate the given rows in O(logN).

> Solution.2: Lazy with NRI (Or equality-deletes)



DataFile-0

a	b	c
3	Jack	Free
7	Machael	Free
11	Tim	Paid
13	Nick	Paid
17	OpenInx	Free

DataFile-1

a	b	c
7	Machael	Paid
17	OpenInx	Paid

DeleteFile-0

field_ids	column_values
(a,b)	(7, Machael)
(a,b,c)	(17, OpenInx, Free)

UPDATE table SET (7, Machael, Paid) WHERE a=7 AND b=Machael;
UPDATE table SET (17, OpenInx, Paid) WHERE a=17 AND b=OpenInx AND c=Free;

DataFile-3

a	b	c
3	Jack	Free
7	Machael	Paid
11	Tim	Paid
13	Nick	Paid
17	OpenInx	Paid

For a table with columns (a,b,c) (the field_ids of a,b,c are 0,1,2) , we may have the UPDATE query:

```
UPDATE table SET (7, Machael, Paid) WHERE a=7 AND b=Machael;
UPDATE table SET (17, OpenInx, Paid) WHERE a=17 AND b=OpenInx AND c=Free;
```

For the delete differential file, we will have two columns : 1. field_ids ; 2. column values. so the row-delete equality predicates can be encoded as following:

```
<list(0,1), list(7, Machael)>
<list(0,1,2), list(17, OpenInx, Free)>
```

for the field_ids, we can optimize it by bit map, say it will encode as following (the fields_ids will be a bit_map, the values would be a list<binary>):

```
<binary(110), list(7, Machael)>
<binary(111), list(17, OpenInx, Free)>
```

When executing the query, we will do:
Step.1: Dispatch the UPDATE/DELETE query to the partition and tasks.
Step.2: Each task will parse the equality predicates and the full columns of a row, then encode them into a row of delete differential file, finally append it into file. It will do the same thing for insert data files.

We can clearly see that the equality updates/deletes won't search the row id from insert data files for a given UPDATE/DELETE query, the write will be very fast, while when we read the data we need to do the nested-loop JOIN between insert data files and delete differential files (use hash-JOIN if the data size would cache in the memory).

Q2. Construct the manifest & snapshot and commit.

The first thing is maintaining the sequence number in both manifest files and data files. Each new write commit should generate an auto-increment sequence number, so that the manifest & data files would attach it, and finally persist them in the filesystem.

When constructing the manifest file list in snapshot, we must sort the manifests by the descending sequence number, so that we can read the related delete differential files efficiently (discussed above). The MergeAppend implementation should also remember this.

In my demo, I consider the delete differential files as a normal data file although different schemas, so I did not define any extra Table API to finish the transaction commit. In the document[1], I saw it mentioned how to define a few extra Table API to commit transactions with insert/delete files. We can think further about which solution is better ?

Q3. How to handle the CDC changes.

In the mail list, some users are asking about how will the iceberg handle the CDC changes (Change Data Capture). The CDC changes may come from the upstream RDBMS/NoSQL databases, such as MySQL binlog, HBase replication entries. Let's take the MySQL row-level binlog as the example. If a table user has three columns (a,b,c), then the row-level binlog will has the following format:

```
UPDATE user set (1,2,3) WHERE a=0 AND b=0 AND c=0;
```

Each UPDATE query provides the full column matching conditions and all column values to set. It could be divided into two operations: 1. *DELETE (0,0,0)* ; 2. *INSERT (1,2,3)*. As we discussed above, we would provide the equality deletes and file/pos deletes. For the CDC cases, it would be always a continuous changing stream sinking to the iceberg table, in theory there will be endless UPDATE query which would be executed in the iceberg table. Choosing the file/pos solution is unacceptable for most of the user cases because we don't like that the write latency / throughput are quite limited by the file/pos indexing with the given column-values conditions. So better to choose **equality deletes**.

There're still some trade-offs we need to consider even if the equality-deletes. The RDBMS table usually has a key at least, which may don't have to be unique. Should we define the key in its mapping iceberg table as sorted ? If so, sorted when writing? Or during the compaction ?

Let's see the following table:

	Don't define any sorted key	Define the sorted key	
		Sort during writing	Sort during compaction
Write Efficiency (sink side)	Yes, because we can just write the column values into delete/insert files.	Impacted, need to shuffle based on the key before writing the delete/insert files.	Yes.
Read Efficiency (source side)	No, need to JOIN the delete+insert files by shuffling.	Yes, merge-sort the delete+insert files.	It's reading fast after the compaction finished, while not efficient if not finished.
End To End Ingest cost	If the insert file has N rows, and the delete file has M rows, then JOIN with cost N*M. total cost is (N+M+N*M). Of course if we sorted the files first, then JOIN with merge-sort. The cost should be the same as *sort during writing*.	Writing the insert file costs NlogN, writing data file costs MlogM. JOIN will cost N+M. Total cost is (N*logN + M*logM + N + M).	Similar with *sort during writing* if compaction finished. while if not finished then it's similar with *the unsorted key cases.*

We’d better provide an option to define whether the column is sorted or not (Anton did a good job for this. I will finish it if he didn’t have the time). Then the users can decide the option based on what they want to tradeoff. If you expect the best write throughput, then writing without sorting keys is a good choice, otherwise if you expect the best read throughput then you might want to choose the copy-on-write way with sorted keys. In general, for better end to end ingest efficiency, we’d better to choose the *sort during writing* in CDC cases.

Integrate read and write interfaces with Spark/Flink

<TODO>

Compaction

<TODO>

References

[1]. Updates/Deletes/Upserts in Iceberg
[2]. Proposal: Iceberg Merge-on-Read
[3]. Some personal views about the Apache Iceberg
[4]. Has the topic of CDC (change data capture) been considered for Iceberg? If not, should it?
[5]. https://github.com/openinx/incubator-iceberg/pull/5/files