Compactor: A Hidden Database Server that Handles Post-Ingestion and Pre-Query Workloads

By Nga Tran & Paul Dix, InfluxData

The demand for high volumes of data has increased the need for databases that can handle both data ingestion and querying with the lowest possible latency (aka high performance). To meet this demand, database designs have shifted to prioritize minimal work during ingestion and querying, with other tasks being performed in the background as post-ingestion and pre-query.

This article will describe those tasks and how to run them in a completely different server to avoid sharing resources (CPU and memory) with servers that handle data loading and reading.

Tasks of Post-Ingestion and Pre-Query

Depending on the designs and offerings of a database, the tasks that can proceed after the completion of data ingestion and before the start of data reading can be different. In this post, we describe three most common tasks: data file merging, delete application, and data deduplication.

Data File Merging

Query performance is always an important goal of most databases and, thus, their data needs to be well organized such as sorted and encoded (aka compressed) or indexed. Since query processing can handle encoded data without decoding it, and the less I/O a query needs to read the faster it runs, it is a clear benefit if a large amount of data is encoded into a few large files. In a traditional database, the process that organizes data into large files is performed during load time by merging ingesting data with existing data. The sorting and encoding or indexing are also needed during this data organization; hence, for the rest of this article the sort, encode and index go hand in hand with the file merge operation.

On the other hand, fast ingestion has become more and more critical to handle large and continuous flow of coming data and near real-time queries. To support fast performance for both data ingesting and querying, at load time, the newly ingested data is not merged with the existing but stored in a small file (or small chunk in memory in the case of a database that only supports in-memory data). The file merge is performed in the background as a post-ingestion and pre-query task. A variation of LSM tree (log-structured merge-tree) technique is usually used to merge them. With this technique, the small file that stores the newly ingested data should be organized (e.g. sorted and encoded) the same as other existing data files but since it is a small set of data, the process to sort and encode that file is trivial. The reason to have all files organized the same will be explained in the section Data Compaction below.

Refer to this data partitioning article for examples of data-merging benefits.

Delete Application

Similarly, the process of data deletion and update needs the data to be reorganized and takes time, especially for large historical datasets. To avoid this cost, data is not actually deleted when a delete is issued but a tombstone is added into the system to 'mark' the data as 'soft deleted'. The actual delete is called 'hard delete' and will be done in the background.

Updating data is often implemented as a delete followed by an insert, and hence, its process and background tasks will be the ones of the data ingestion and deletion.

Data Deduplication

Time series databases such as InfluxDB accept ingesting the same data more than once but then apply deduplication to return non-duplicate results. Specific examples of deduplication applications can be found in this deduplication article. Like the process of data file merging and deletion, the deduplication will need to reorganize data and is the right task for performing in the background.

Data Compaction

The background tasks of post-ingestion and pre-query are well known as data compaction because its output very often contains less data and is more compressed. The compaction is a background loop to find data it wants to compact and perform the compaction. However, because there are many tasks of compaction as described above and they usually touch the same data set, the compaction performs all of the tasks in the same query plan that scans data, finds rows to delete and deduplicate, and then encodes and indexes them as needed.

Figure 1 shows a query plan that compacts two files. A query plan in the database is usually executed in a streaming/pipelining fashion from bottom up, and each box in the figure represents an execution operator. First, data of each file is scanned concurrently. Then tombstones are applied to filter deleted data. Next, the data is sorted on the *primary key (aka deduplication key)* which is a set of columns before going through the deduplication step that applies a merge algorithm to eliminate duplicates on the *primary key*. The output is then encoded and indexed if needed and stored back in one compacted file. When the compacted data is stored, the metadata of File 1 and File 2 stored in the database catalog can be updated to point to the newly compacted data file instead and then File 1 and File 2 can be safely removed. The task to remove files after they are compacted is usually performed by the database's garbage collector which is beyond the scope of this article.

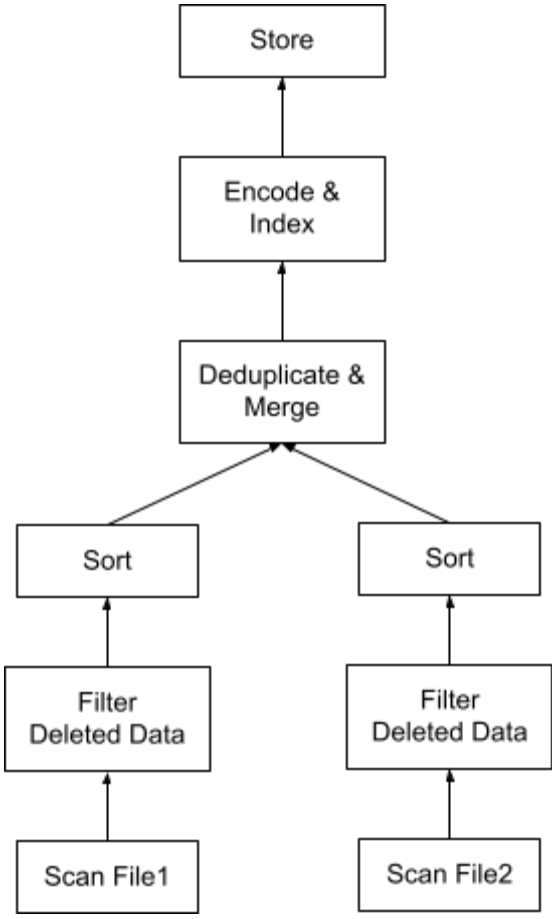

Figure 1: The process of compacting two files

Even though the compaction plan in Figure 1 combines all three tasks in one scan of the data and avoids reading the same set of data three times, the plan operators such as filter and sort are still not cheap. Let us see whether we can avoid or optimize these operators further.

Optimized Compaction Plan

Figure 2 shows the optimized plan of the plan in Figure 1 with two major changes:

1. The operator *`Filter Deleted Data`* is pushed into the *`Scan`* operator which is an effective predicate-push-down way to filter data while scanning.
2. We no longer need the *`Sort`* operator because the input data files are already sorted on the *primary key* during data ingestion. The `Deduplicate & Merge` operator is implemented to keep its output data sorted on the same key as its inputs. Thus, the compacting data is also sorted on the primary key for future compaction if needed.

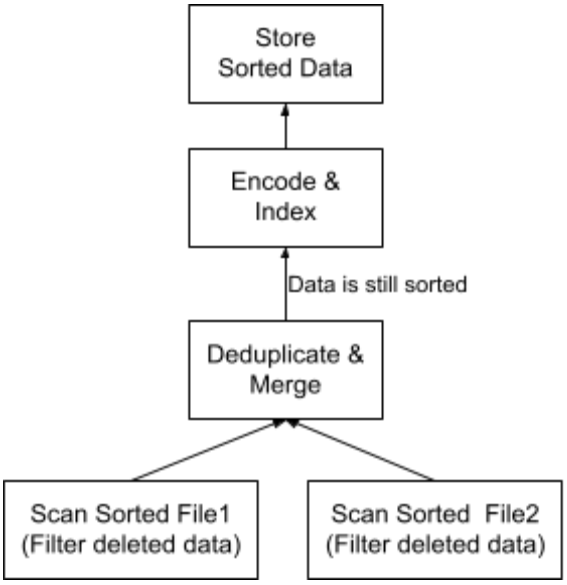

Figure 2: Optimized process of compacting two sorted files

Note that, if the two input files contain data of different columns which are common on some databases such as InfluxDB, we need to keep their sort order compatible to avoid doing resort. For example, the primary key contains columns `a, b, c, d` but File 1 only includes columns `a, c, d` (and other columns that are not a part of the primary key) and sorted on `a, c, d`. If data of File 2 is ingested after File 1 and includes columns `a, b, c, d`, its sort order has to be compatible with File 1's sort order `a, c, d` which means column `b` can be placed anywhere in the sort order but `c` must be after `a` and `d` must be after `c`. For implementation consistency, the new column, `b`, can always be added as the last column in the sort order and, thus, the sort order of File 2 would be `a, c, d, b`.

Another reason to keep the data sorted is that in column-stored format such as Parquet and ORC, the encoding goes well with sorted data. For the common RLE encoding, the lower the cardinality (aka the number of distinct values), the better the encoding. Hence, choosing the lower-cardinality columns to come first in the sort order of the primary key will not only help compress data more on disk but more importantly help the query plan to execute faster because the data is kept encoded during execution as described in the late materialization strategy paper.

Compaction Levels

To avoid the expensive deduplication operation, we want to manage the data files in a way that we know whether they potentially contain duplicates with data in other files or not. This can be done by using the technique of data overlapping. To simplify the examples of the rest of this article, we assume the data sets are time series in which data overlapping means their data overlap on time. However, the overlap technique can be defined on non-time series data, too.

One of the strategies to avoid recompacting well-compacted files is defining levels for the files. Level 0 represents newly ingested small files and level 1 is for compacted, non-overlapping ones. Figure 3 shows an example of files and their levels before and after first and second rounds of compaction. Before any compaction, we have all level-0 files and they potentially overlap in time in arbitrary ways. After the first compaction, many small level-0 files are compacted into two large non-overlapped level-1 files. During first compaction, there are more small level-0 files loaded in that then kick-start a second round of compaction that only compacts the second level-1 file with the newly ingested level-0 files. With the strategy to keep level-1 files always non-overlapped, we do not need to recompact level-1 files if they do not overlap with any newly ingested level-0 files.

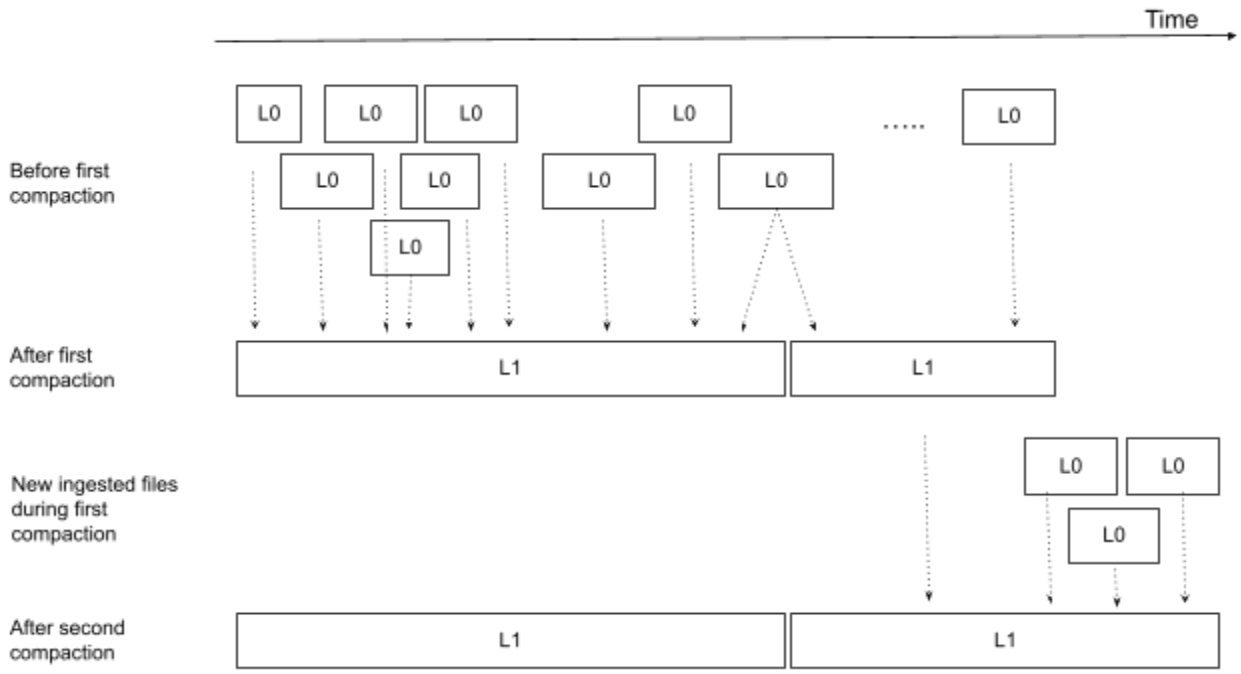

Figure 3: Ingested and compacted files after 2 times of compaction

If we want to add different levels of file size, more compaction levels (e.g. 2, 3, 4, ...) can be added. Except level-0 files that can overlap with any other files, other level files should not overlap with any files in the same level.

Even if we try to avoid deduplication as much as possible, the deduplication operator itself is expensive, especially when the primary key includes many columns that need to be kept sorted. Building fast and efficient memory multi-column sorts is critical and some common techniques to do so are described in this article: part 1 and part 2.

Data Querying

The system that supports data compaction needs to know how to handle a mixture of compacted and not-yet compacted data. Figure 4 illustrates 3 files that a query needs to read: File 1 and File 2 are level 1 and File 3 is level 0 and overlaps with File 2.

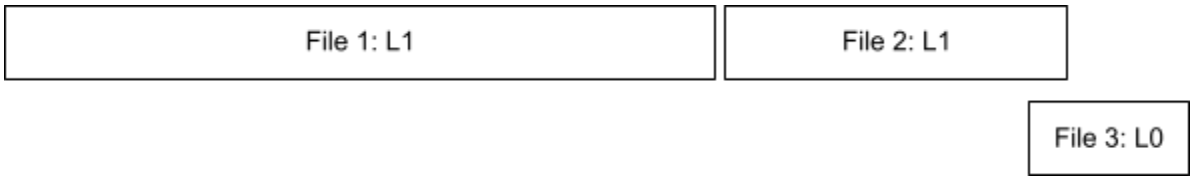

Figure 4: Three files that a query need to read

Figure 5 demonstrates a query plan that scans those three files. Since File 2 and File 3 overlap, they need to go through the `Deduplicate & Merge` operator. File 1 does not overlap with any file and only needs to be unioned with the output of the deduplication. Then all unioned data will go through the usual operators the query plan has to process. As we can see, the more compacted and non-overlapped files produced during compaction as pre-query processing, the less deduplication work the query has to perform.

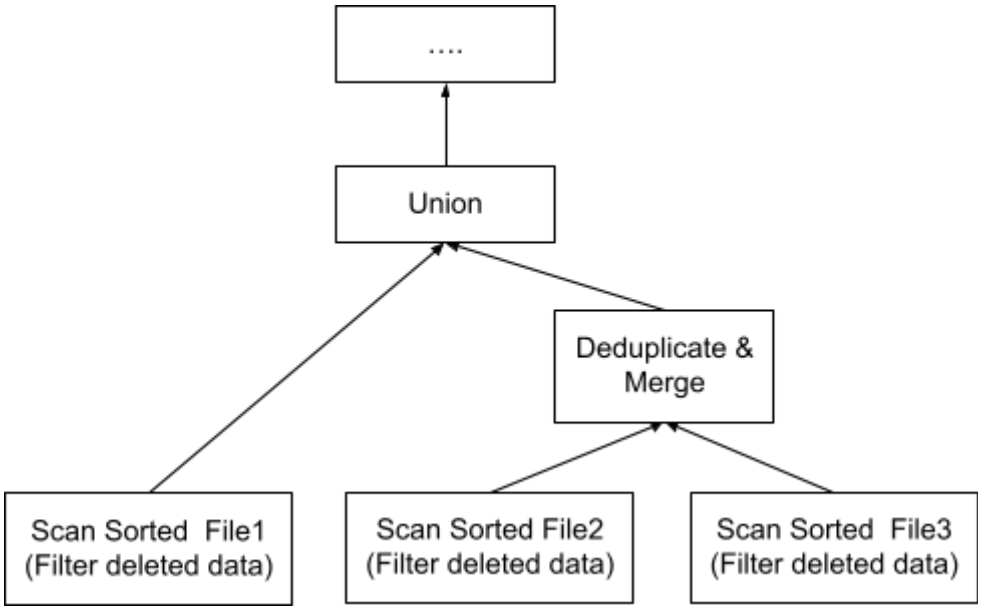

Figure 5: Query plan that reads two overlapped files and one non-overlapped one

Isolated and Hidden Compactors

Since data compaction includes only background tasks of post-ingestion and pre-query, we can perform them in a completely hidden and isolated server named *compactor*. More specifically, the increasingly common cloud object storage (aka shared storage systems) has leveraged the use of a system of multiple servers. Data ingestion, query and compaction can be processed in three respective sets of servers: integers, queriers, and compactors that do not share resources at all.

They only need to connect to the same catalog and storage, and follow the same protocol to read, write and organize data.

Because a compactor does not share resources with any other database servers, it can be implemented to handle compacting many tables (or even many partitions of a table) concurrently. In addition, if there are a lot of tables and data files to compact, several compactors can be provisioned to independently compact different tables/partitions in parallel. Furthermore, if the compaction needs a lot less resources compared to the needs for the ingestion and querying, the separation of servers will enable a system to have many ingesters and queriers to handle large ingesting workloads and queries in parallel respectively while only needing one compactor to handle all the background post-ingestion and pre-querying work. Similarly, if the compaction needs a lot more resources, a system of many compactors, one ingestor and one querier can be provisioned to meet the demand.

A well-known challenge in databases is how to manage resources of their servers: ingesters, queriers, and compactors, to utilize maximum resources (CPU and memory) while never hitting out-of-memory incidents. It is a large topic and deserves its own blog post.

In Summary

Compaction is a critical background task that enables low latency for data ingestion while keeping query performance high. The increasingly common cloud shared object storage has leveraged the importance of multiple database servers in which ingesters, queriers and compactors independently handle data loading, reading and compacting workloads respectively. For more information about the implementation of such a system, check out InfluxDB IOx. Other related techniques needed to design the system can be found in sharding and partitioning articles.