Partitioning Effects in a Sharding Database System

By Nga Tran, staff software engineer, InfluxData

In a previous post, I described a sharding system to scale throughput and performance for query and ingest workloads. This post introduces another common technique, **partitioning**, that adds further effects in performance and management for a sharding database. This post also describes how to handle partitions efficiently for both query and ingest workloads, and how to manage cold (old) partitions where the read requirements are quite different from the hot (recent) partitions.

Sharding vs Partitioning

Sharding is a way to split data in a distributed database system. Data in each shard does not have to share resources such as CPU or memory, and can be read or written in parallel. Figure 1 is an example of a sharding database. Sales data of 50 states of a country is split into four shards, each containing data of twelve or thirteen states. By assigning a query node to each shard, a job that reads all 50 states can be split between these four nodes running in parallel and will be performed four times faster compared to the setup that reads all 50 states by one node. More information about shards and their scaling effects on ingest and query workloads can be found in the previous post.

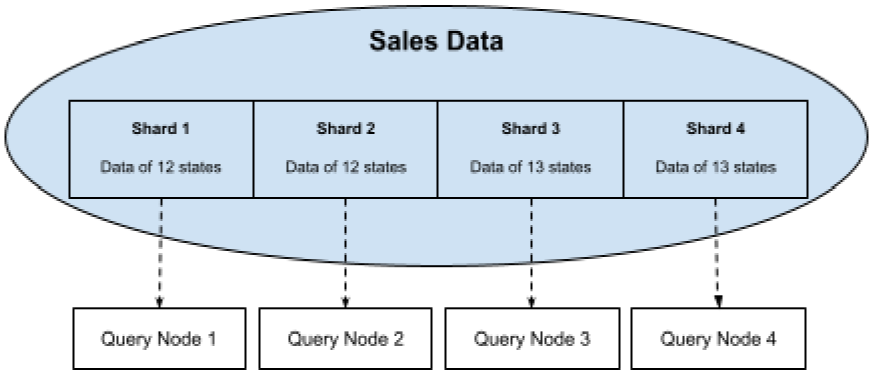

Figure 1: Sales Data is split into 4 shards, each is assigned to a Query Node

Partitioning is a way to split data within each shard into non-overlapped partitions for further parallel handling, reducing the reading of unnecessary data, and efficiently implementing data retention policies. In Figure 2, data of each shard is partitioned by sales day. If we need to get a report on sales of one specific day such as May 1st, 2022, the query nodes only need to read data of their corresponding partitions of 2022.05.01. The rest of this post will focus on the effects of partitioning and on how to handle and manage them efficiently for both query and ingest workloads on both hot and cold data.

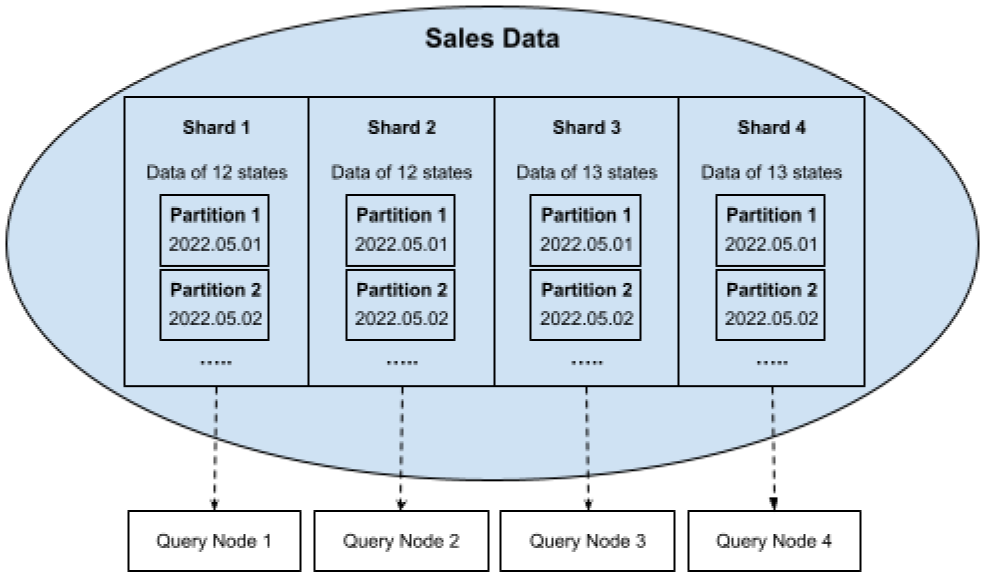

Figure 2: Sales data of each shard is further split into non-overlapped day partitions

Partitioning Effects

The three most common benefits of data partitioning are data pruning, intra-node parallelism, and fast deletion.

Data Pruning

A database system may contain several years of data but most queries only need to read recent data (e.g ”How many orders have been placed in the last 3 days”). By partitioning data into non-overlapped partitions, illustrated in Figure 2, it can be easy to skip entire out-of-bound partitions and only read and process relevant and very small sets of data to return results quickly.

Intra-Node Parallelism

Multi-threading processing and streaming data is critical in a database system to fully use available CPU and memory and obtain the best performance possible. Partitioning data into small partitions makes it easier to implement a multi-threaded engine that executes one thread per partition. For each partition, more threads can be spawned to handle data within that partition. Knowing partition statistics such as size and row count will help allocate the optimal amount of CPU and memory for specific partitions.

Fast Data Deletion

Many organizations only keep recent data (e.g. data of the last three months) and want to remove old data ASAP. By partitioning data on non-overlapped time, removing old partitions can be made as simple as deleting files without the need to reorganize data and interrupt other query or ingest activities. If all data must be kept, a section later in this post will describe how to manage recent and old data differently to ensure the systems provide great performance in all cases.

Storing and Managing Partitions

Optimizing for Query Workload

Since a partition contains a quite small set of data, we do not want to store a partition in many smaller files (or chunks in the case of in-memory database). A partition should be presented by one or a few files to:

- Reduce I/Os while reading data for executing a query.
- Improve data encoding/compression which in turn lowers the storage cost and, more importantly, improves query execution speed due to reading less data. There are other read benefits going with encoding such as sort but they are beyond the scope of this post.

Optimizing for Ingest Workload

Naive Ingestion

To keep data of a partition in a file for the benefits of reading optimization stated above, every time a set of data is ingested, it must be parsed and split into the right partitions, then merged into the existing file of its corresponding partition, as demonstrated in Figure 3. The process of merging new data with existing data often takes time because of expensive I/O and the cost of mixing and encoding data of the partition. This will lead to long latency for:

- Responding back to the client that the data is successfully ingested.
- Querying the newly ingested data as it is not available in the storage yet.

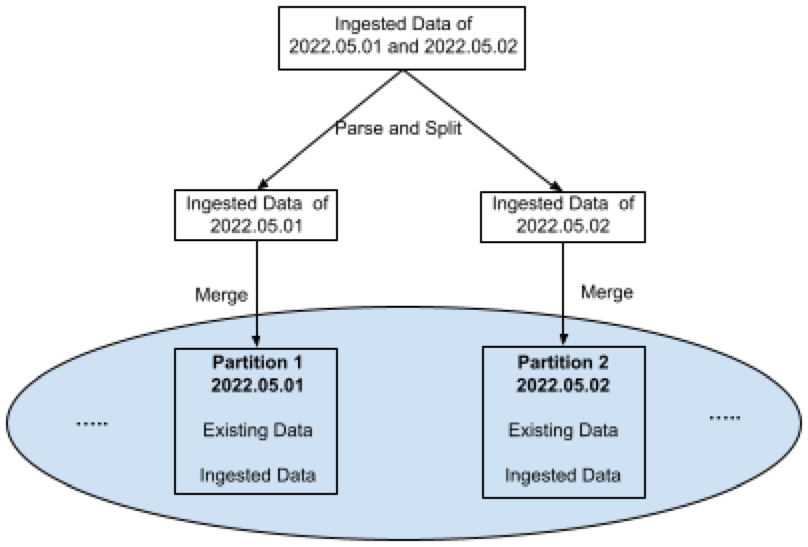

Figure 3: Naive Ingestion in which new data is merged into the same file of the existing one immediately

Low Latency Ingestion

To keep the latency of each ingestion low, we can split the process into two steps: ingestion and compaction.

Ingestion

During the ingestion step, ingested data is split and written to its own file shown in Figure 4. It is not merged with the existing data of the partition. Right after the ingested data is successfully durable, the ingested client will get a success signal and the newly ingested file is available for querying. If the ingest rate is high, there will be many small files accumulated in a partition illustrated in Figure 5. At this stage, a query that needs data from a partition must read all files of that partition. It is not ideal but the compaction step below will keep this stage temporarily short.

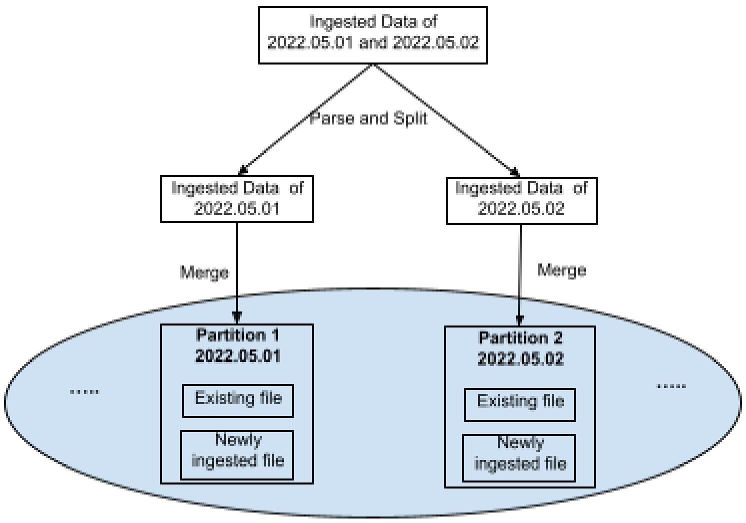

Figure 4: Newly ingested data is written into a new file

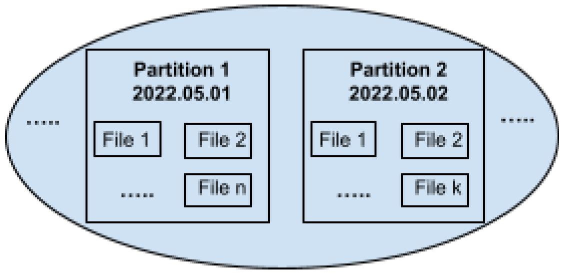

Figure 5: Many files in a partition for high ingest workload

Compaction

Compaction is a process to merge files of a partition into one or a few files for better query performance and compression. Figure 6 demonstrates that all files in partition 2022.05.01 are merged into one file while all files of partition 2022.05.02 are merged into two files each smaller than 100MB. The decision on how often to compact and what the largest size of compacted files can be different for various systems, but the common goal is to keep the query performance high by reducing I/Os (aka the number of files) and having the files large enough to effectively compress.

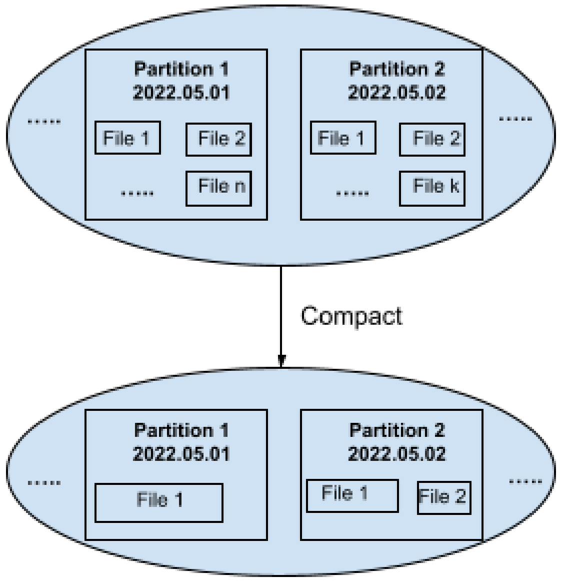

Figure 6: Compacting several files of a partition into one or few files

Hot vs Cold Partitions

Partitions that are queried a lot are considered **hot partitions**, while the ones that are rarely read are called **cold partition**s. In databases, hot partitions are usually the partitions of recent data such as recent sales dates, and the older a partition is the less likely it will be read. Moreover, when the data gets old, it is usually queried by **a larger chunk** such as by month or even by year. Here are a few examples to **ambiguously** categorize hot (data of the current week), less hot (data from previous weeks but in the current month), cold (data from previous months but in the current year), more cold (data of last year and older).

Query	**Category**
● Number of orders of Apple products so far today ● Number of orders of yesterday's sales ● Sales revenue so far this week	Hot
● Last week's sales revenue	Less hot
● Last month's sales revenue of Samsung products	Cold
● Total sales revenue of last year	More cold

To reduce the hot and cold ambiguity, we need to answer these questions:

1. How to quantify hot, less hot, cold, more cold, and even more and more cold?
2. How to have fewer I/O in the case of reading cold data? We do not want to read 365 files, one representing a day partition of data, in case we need to get last year sales revenue.

Hierarchical Partitioning

Hierarchical partitioning provides answers for the questions above and is illustrated in Figure 7. Data of each day of the current week is stored in its own partition. Older weeks of the current month are partitioned by week. Older months in the current year are partitioned by month. Even older data is partitioned by year.

This model can be relaxed by defining an **active partition** in place of the **current date partition**. All partitions after the active one will be partitioned by date; data before that that will follow the same mechanism. This allows the system to keep as many small recent partitions as necessary. Even though all examples in this post partition data by time, non-time partitioning will work the same as long as you can define expressions for a partition and their hierarchy.

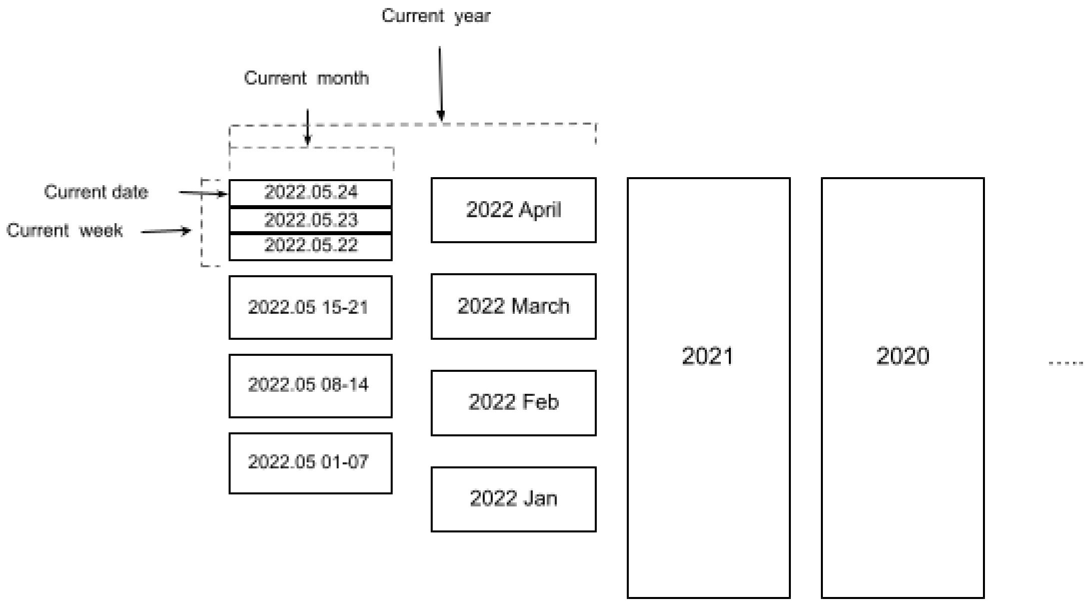

Figure 7: Hierarchical Partitioning

This partitioning mechanism reduces the number of partitions in the system, keeping it easier to manage, and reducing the number of partitions that need to be read when querying larger and older chunks.

The **query** process for this hierarchical partitioning is the same as the non-hierarchical one as it will apply the same pruning strategy to read the only relevant partitions. The **ingestion** and **compaction** processes will be a bit more complicated to organize the partitions in their defined hierarchy.

Aggregate Partitioning

Many organizations do not want to keep old data but instead their aggregations such as number of orders and total sales of every product every month. This can be supported by aggregating data and partitioning them by month. However, since the **aggregate partitions** store aggregated data, their schema will be different from the non-aggregated ones which will lead to extra work for ingesting and querying. There are different ways to manage this cold and aggregated data, but these are large topics for a future post.

Nga Tran is a staff software engineer at InfluxData, and a member of the IOx team, which is building the next-generation time series storage engine for InfluxDB. Before InfluxData, Nga had been with Vertica Analytic DBMS for over a decade. She was one of the key engineers who built the query optimizer for Vertica, and later, ran Vertica's engineering team.