Innumerable Hot Data partitioning and de-partition of large distributed data set

Nirmit Jain
jainnirmit8@gmail.com

Context

In this paper, we primarily refer to large distributed datasets maintained within a data lake, which are either directly queried by end consumers or accessed by multiple consumers with distinct query patterns—some oriented toward analytical workloads, while others are more transactional in nature.

In this paper, we will explore strategies for partitioning data in a way that minimizes downtime, ensures faster performance, and enables a fully autonomous process.

In this paper de partitioning is not detailed but it is the exact opposite process of partition of large datasets. So just to reduce the amount of context it is not being explicitly detailed here.

What is a large distributed data set in a data lake?

A **large distributed data set** in a **data lake** refers to a collection of vast and varied data that is, **Massive in size** (potentially terabyte or more) and is **Located within a data lake architecture** (typically built on cloud storage systems like Amazon S3, Azure Data Lake Storage, or Hadoop Distributed File System (HDFS))

Example:

Imagine a company that stores clickstream logs, social media feeds, sensor data, and customer transaction records. This data, distributed across a cloud data lake like AWS S3 and processed with tools like Apache Spark or Presto, forms a **large distributed data set**.

What is data partitioning?

Data partitioning is a fundamental strategy in data lakehouse architectures, designed to optimize performance, scalability, and manageability.

- Data partitioning is a technique used to divide a large dataset into smaller pieces, called partitions.
- These partitions are typically stored separately, often on different storage devices or within different tables.

- Partitioning helps manage large datasets by allowing for parallel processing and improved resource utilization.

What are the benefits of Data partition?

There is a great blog by Onehouse for understanding this better follow this link

Most query engines leverage **partition pruning** techniques to limit the data search to only those relevant partitions matching conditions specified in the query instead of processing all the partitions. These are achieved via **filters and predicate push-downs** during query analysis and planning phases. Query engines depend on such techniques to devise an optimal plan to execute the query instead of having to scan the entire table data. Partition-pruned executions can be orders of magnitude faster than executing the same query on a non-partitioned table with the same columns and data definitions.

When queries target partition columns or specific values, predicate pushdown enables the query engine to fetch only the relevant partitions. This significantly reduces query costs by minimizing compute usage, network overhead, and API calls to external systems—ultimately lowering system load and promoting a more efficient, harmonious system.

What are the disadvantages of Data partitioning?

Data filters and predicate pushdown are highly effective when **partitioning is thoughtfully designed** around actual use cases and query patterns. However, if partitioning is misaligned with how the data is queried, it can become a liability—driving up query costs, increasing data scans, and degrading performance.

For instance, consider an employee directory dataset that includes global employment history, with fields like company type and field of work. If the data is partitioned by field of work but most queries filter by company type, predicate pushdown won't apply. As a result, the engine must scan unnecessary data, leading to increased compute usage, higher costs, and slower query execution.

Data partitioning can also introduce challenges such as **data skew**, especially when executing complex analytical operations like joins and group-bys. When partitions are unevenly sized, certain operations become bottlenecks, making it difficult to optimize performance. To address this, **additional strategies like salting, hash-based partitioning**, or custom data redistribution often need to be employed—**ironically adding complexity to a system originally partitioned to reduce query cost and improve efficiency**.

So should we focus on how we choose the right partition column?

The answer isn't straightforward—real-world applications rarely follow a fixed flow or consistent query patterns. Business needs evolve, data grows in unpredictable ways, and the questions we ask of that data continually change.
Before we even ask, ***"What should the partitioning column be?"***, we should first ask, ***"When does partitioning truly become necessary?"***. Understanding the point at which partitioning adds value is critical to making informed and adaptive design choices.

When does partitioning truly become necessary?

Partitioning is generally not recommended for low-velocity, small-volume datasets—particularly those under 10 GB—unless read patterns justify it.

Key factors to consider:

- Data Volume: If the dataset is only a few gigabytes, most modern query engines or in-memory databases can process it efficiently without the need for partitioning.

- Data Read Pattern: Even with a modest data size, if queries frequently access the dataset may in a magnitude of 100QPD, the effective data scanned can resemble terabyte-scale workloads, making partitioning more valuable.

These challenges typically emerge at scale—when datasets grow significantly or when timely, consistent insights become critical for business operations.

Choosing an effective partitioning scheme for large distributed datasets is only practical when there's a well-defined business problem, with established query patterns and proven methods for extracting insights from the system.

If there's no clear visibility into how the system will evolve over time, how can we reliably predict an optimal partitioning strategy?

How is partitioning of large datasets typically performed, and under what circumstances does the need for partitioning arise?

The need to partition large datasets typically arises when existing data processing jobs begin to exceed resource limits—either slowing down significantly or incurring higher operational costs. These performance or cost bottlenecks often trigger discussions around introducing partitioning, restructuring the workflow, or even adopting a different technology stack to optimize efficiency and scalability

These discussions often lead to the adoption of a suitable partitioning strategy for the problematic dataset. However, this introduces a new challenge: partitioning existing data to align with the intended query patterns. Organizations then make critical decisions—either to prune or reduce historical data to minimize

backfill processing costs, or to retain all historical data, accepting the one-time expense and resource load required to reprocess it at scale.

In many cases, only a subset of the output data experiences high read frequency, while the rest remains largely untouched. Identifying these high-access partitions in advance is challenging, as it requires collecting and analyzing extensive query metadata. Without this insight, it's difficult to determine which partitions are most accessed and should be optimized.

Additionally, backfilling may be associated with unavailability of data and repartitioning underlying datasets often require rewriting existing queries, which can lead to temporary downtime or disruption in delivering insights.

"Surely there needs to be a better way of doing these thing"

Before we dive into optimal partitioning strategies, let's first understand how large distributed query engines and data processing systems operate.

How large distributed query engines and data processing systems operate?

Large distributed query engines and data processing systems are optimized for parallel data processing across multiple nodes. The majority of processing time is spent on I/O operations—reading and writing large volumes of data from distributed storage systems. Systems like Apache Spark or Presto distribute queries across multiple workers, performing computations in parallel. However, the challenge lies in minimizing I/O bottlenecks, as inefficient data shuffling or excessive disk reads can significantly slow down query execution. Effective partitioning and indexing are crucial to reduce unnecessary data access and improve performance.

When processing large volumes of data for partitioning, the operation can become highly unoptimized. This often involves extensive data shuffling, heavy disk I/O, and repeated read-write cycles. Due to technical constraints—such as limits on memory, I/O bandwidth, or compute quotas—the entire dataset cannot always be loaded or processed in one go. As a result, the system may need to read the input data multiple times, significantly increasing overhead and prolonging the partitioning process

Just-in-time partitioning of data isn't always necessary—there's no need to read and rewrite the entire dataset in a single, time-consuming process. Instead, we can perform hot partitioning of data.

What is hot data partitioning and how to perform it?

I would like to categorize partitioning strategies into cold data partitioning and hot data partitioning, based on the timing and mechanism of how the data is segmented.

- **Cold Data Partitioning** occurs **on-demand**, typically when a request for data necessitates the creation of a new partition. This method involves **spawning a separate process** dedicated to carrying out the partitioning operation. Since this process is initiated reactively, it may introduce some latency, particularly during high-load periods. Cold partitioning is generally used when the data distribution is not known beforehand or when partitioning is too resource-intensive to be performed proactively.

- **Hot Data Partitioning**, on the other hand, is a more **proactive and lightweight** approach. Instead of physically segmenting the data at the time of the request, the system **generates metadata** that logically links two or more storage locations to the same dataset. This allows the system to **simulate partitioning** without actually moving or duplicating data.

Partitioning in data systems is often touted for its ability to enable filter pruning and predicate pushdown, both of which can improve query performance. The idea is that by knowing exactly where relevant data resides (i.e., the partition path), the system can avoid scanning unnecessary data. However, this approach has a significant limitation: if the system already knows the exact path to the data, the performance gain attributed to partitioning is minimal—you're essentially just querying data directly, which somewhat defeats the purpose of intelligent data partitioning.

This exposes a flaw in traditional partitioning approaches: they rely heavily on physical data paths, which limits their adaptability and efficiency in dynamic or real-time environments.

To address this, the concept of Hot Partitioning emerges. Rather than depending on static partition paths, Hot Partitioning relies on metadata-driven strategies. It dynamically captures and leverages metadata about how data should be partitioned, allowing the system to simulate partitioning logic on-the-fly without physically restructuring the data. This enables smarter, more robust data access patterns.

Additionally, Hot Partitioning enables the creation of multiple logical partitioning keys for the same dataset, allowing it to support diverse query patterns efficiently. This approach helps mitigate common issues such as data skew, small file proliferation, and inefficient query execution, without the need to physically duplicate or restructure the data.

How to perform Hot partitioning?

Hot partitioning of data can done in two ways:

- Logical set: We create metadata of the partition strategy and store it as metadata location the underlying files and are not altered in this strategy.
- Write on Read: Extending Logical set partitioning we enable rewriting of data on read of the data to a new location, this concept is similar to MoR tables in ACID based data lake table strategy.

Logical set

In this strategy, we collect user-defined inputs—such as the partition keys over which the dataset should be logically segmented. Based on these keys, we generate a structured metadata representation that captures partition-specific information for each key, as illustrated below.


```
[
   {
       "keyColumnName": "partition_key1",
       "keyValues": {
           "v1": [
               {
                   "relativePath": "fileA.csv",
                   "KeyrecordsInFile": 1000,
                   "totalRecords": 10000
               },
               {
                   "relativePath": "fileB.csv",
                   "KeyrecordsInFile": 1000,
                   "totalRecords": 10000
               }
           ],
           "v2": [
               {
                   "relativePath": "fileC.csv",
                   "KeyrecordsInFile": 1000,
                   "totalRecords": 10000
               },
               {
                   "relativePath": "fileB.csv",
                   "KeyrecordsInFile": 1000,
                   "totalRecords": 10000
               }
           ]
       }
   },
   {
       "keyColumnName": "partition_key2",
       "keyValues": {}
   }
]
```

Each piece of metadata is meaningful and plays a crucial role in identifying the exact paths where specific partition keys reside, thereby **minimizing the amount of data scanned during query execution**.
The fields "totalRecordsInFile" and "keyRecordsInFile" enable a short-circuiting mechanism, allowing the system to bypass unnecessary records reads. This too reduces I/O operations and accelerates the processing of input data.

Below is the pseudo algorithm to follow for implementing Logical Set:

- Write Algorithm:
    - Take partition key as input
    - Enrich data set with record file path and total record per file
    - For Each partition key perform
        - Group by over partition key and record file path and store aggregated value of record count per partition key and record file path along with max of total record in file path
    - Check if the partition metadata exists
        - If Exists perform merging with existing metadata
        - If not initialize metadata
    - The Algorithm does high lights around concurrent write, since that is mostly a widely known algorithm and has ample implementation out in the open source world.
- Read Algorithm:
    - Check if the partition metadata exists
        - If Exists
            - Push down and list of predicates and get a list of files that needs to be processed.
            - From the list generated files to process read records from the file where the keys exist.
        - If not initialize metadata read whole data
    - The Algorithm does high lights around concurrent read and write, since that is mostly a widely known algorithm and has ample implementation out in the open source world.

Drawbacks of Logical Set Hot partitioning:

While logical partitioning can significantly reduce data scans by limiting the number of files read, it does not guarantee a reduction in scanned data volume—especially in cases involving low-cardinality partition keys.

For instance, if partitioning is based on a key with only two distinct values, there's a high likelihood that most of the data will be spread across both partitions. As a result, queries targeting either value may still require reading nearly the entire dataset, leading to increased compute overhead and slower data processing.

Advantages of Logical Set Hot Partitioning:

Logical set partitioning is most effective when query patterns are diverse and the partition key has high cardinality.

For example, consider a transaction table where there's a need to partition data based on customer IDs. This approach can lead to significant data skew and the generation of numerous small files, especially if the data distribution is uneven. In such cases, using highly flexible logical set partitioning is a more suitable strategy, as it avoids the pitfalls of physical partitioning while still optimizing query performance and reducing overall data scan.

Write on Read

As established, logical set hot partitioning works well for high-cardinality datasets, but proves less effective for low-cardinality scenarios. This raises the question: What approach should be taken when dealing with low-cardinality partition keys?

This is where the concept of "write-on-read" comes into play—a strategy inspired by Merge-on-Read (MoR) table designs. In MoR systems, small files are not immediately compacted during writes. Instead, compaction is deferred to read time, optimizing write performance while merging data only when there is an active consumer. This approach minimizes the impact on overall system throughput by shifting the compute burden to fewer, active read operations.

Extending this concept to low-cardinality datasets, we still begin with the initial logical set partitioning, but we augment it with selective physical partitioning. Instead of performing a full backfill partitioning job, we partition the data incrementally and in chunks, triggered by actual query access patterns. This selective processing allows:

- Efficient resource usage
- Equilibrated distribution of compute load across consumers
- And partitioning only for specific key values that are frequently accessed.

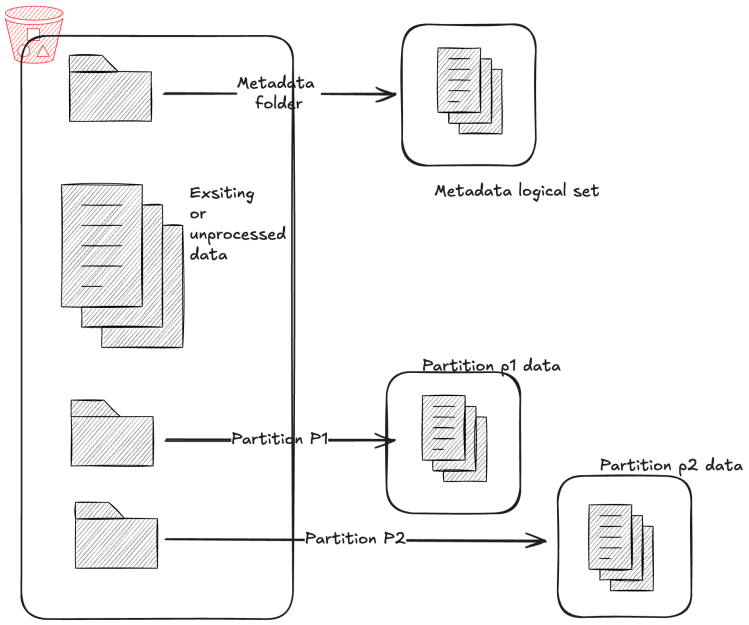
The result is a more balanced and responsive system that adapts to real-world usage without overwhelming the storage layer or incurring unnecessary compute overhead.

Below is the pseudo algorithm to follow for implementing Write on read partitioning strategy:

- Write Algorithm Initialisation:
    - Take partition key as input
    - Enrich data set with record file path and total record per file
    - For Each partition key perform
        - Group by over partition key and record file path and store aggregated value of record count per partition key and record file path along with max of total record in file path
- Write Algorithm New Input data:
    - Check if the partition key folder exists
        - If it does not exists create partition folder and write partition data

    - For Each partition key value
        - Update Metadata and enable partition level path initialization if not already
    - The Algorithm does high lights around concurrent write, since that is mostly a widely known algorithm and has ample implementation out in the open source world.
- Read Algorithm:
    - Check if the partition metadata exists
        - If Exists
            - Push down and list of predicates and get a list of files that needs to be processed.
            - If the partition level path is initialised consider whole data partition partition along with list of files from partition metadata
            - From the list generated files to process read records from the file where the keys exist.
            - Start a new process to chunk data into partition
            - Consider a chunk of file from the list of files from partition metadata
            - Chunking can be done based on number of files or key value records, per file sort ascendingly and data size and much more
            - Once chunk of file is done and respective data is written back to partition path
            - Enable partition path initialization for all the partition key values.
            - Remove all the list of files that have been chunked from each of the partition key values
        - If metadata not initialized read whole data
    - The Algorithm does high lights around concurrent read and write, since that is mostly a widely known algorithm and has ample implementation out in the open source world.

Post Hot partitioning of data using “Write on read” the dataset will look as depicted below



In this approach, unprocessed data naturally accumulates all partition key values that have not yet been queried or accessed. This becomes a beneficial byproduct of the Write-on-Read (WoR) Hot Partitioning strategy, where unaccessed data is automatically identified without requiring explicit tracking or additional processing.

Advantages of Write on Read

Improved Write Efficiency: Defers physical partitioning, allowing faster, lightweight writes.

- On-Demand Partitioning: Data is physically organized only when accessed, reducing unnecessary computation.
- Resource Optimization: Focuses compute and storage on frequently queried data.
- Automatic Cold Data Identification: Unqueried partition keys remain unprocessed, making cold data easy to detect.
- Supports Concurrency: Designed to handle concurrent reads and writes without conflicts.
- Flexible and Adaptive: Chunking strategies can be customized based on file size, record count, or access pattern.

Disadvantages

- Initial Read Latency: First-time access to unprocessed data may incur extra processing delay.
- Metadata Overhead: Requires careful tracking of file states, partition keys, and initialization flags.
- Complex Implementation: Adds logic for chunking, metadata merging, and partition path management.

Comparison between Logical Set Hot Partitioning and Write-on-Read (WoR) Hot Partitioning:

Metric	Logical Set Hot Partitioning	Write-on-Read (WoR) Hot Partitioning
Best Use Case	High-cardinality partition keys, varied query patterns	Low-cardinality partition keys, incremental access patterns
Partitioning Type	Logical (metadata only)	Hybrid (logical + selective physical)
Data Movement	None — files remain untouched	On-demand, partial data rewriting during reads
Write Performance	Fast, as no data is rewritten	Fast initially, slows slightly during read-triggered chunking

Read Performance (First Access)	Fast if metadata is initialized; may scan large files if low cardinality	May incur latency due to chunking and writing during first access
Query Optimization	Enables filter pruning, short-circuiting on metadata	Enables filter pruning + improves layout over time via incremental rewrite
Skew and Small File Mitigation	Since data is still consolidated small files is avoided	Strong — rewrites help mitigate skew and small file issues
Storage Overhead	Minimal (no file duplication or movement)	Slightly higher — due to physical partitioning on access
Metadata Management Complexity	Moderate	High — tracks partition state, file progress, and write status
Cold Data Detection	Requires explicit tracking	Automatic — non queried data remains unprocessed
Adaptability to Query Patterns	High — can support multiple logical keys	Very high — adapts to actual access patterns over time
Compute Cost	Low — avoids physical transformation	Balanced — compute cost deferred and amortized over reads

Summary

This paper proposes dynamic partitioning strategies for large distributed datasets in data lakes to optimize performance, reduce downtime, and enable autonomous operation. Traditional cold partitioning is resource-intensive and inflexible, while hot partitioning offers adaptive, metadata-driven methods. Two hot partitioning approaches are detailed:

Logical Set Partitioning – Uses only metadata to simulate partitioning without moving data; best for high-cardinality keys and diverse queries.

Write-on-Read Partitioning – Adds selective physical partitioning triggered by queries; ideal for low-cardinality keys and incremental access patterns.

These approaches improve read efficiency, reduce compute overhead, adapt to evolving query patterns, and simplify cold data identification—making them more scalable and cost-effective for modern data lake architectures.