Data Engineering Design Patterns

Bartosz Konieczny

Published by O'Reilly Media, Inc.

Chapter 8. Data Storage Design Patterns

Have you ever waited for a query or job results longer than two minutes while working in a big data environment? Many of you will probably answer yes, and some of you may have even waited more than 10 minutes. This time factor is an important aspect in our data engineering work. The faster a query or job runs, the earlier we'll get the response and hopefully, the cheaper it will cost to get it.

You can optimize this time factor in two ways. First, you can add more compute resources, which is a relatively quick and easy method without any extra organizational steps. However, it's also a retroactive step that you might need to perform under pressure, for example, after users start to complain about reading latency.

The second way to optimize is by taking preemptive action that relies on a wise data organization with the data storage design patterns covered in this chapter. This well-thought-out organization should improve execution time and provide feedback earlier.

In this chapter, you'll first discover two partitioning strategies that help reduce the volume of data to process and also enable the implementation of some of the idempotency design patterns presented in Chapter 4, such as the Fast Metadata Cleaner pattern. Unfortunately, partitioning only works well for low-cardinality values (i.e., when you don't have a lot of different occurrences for a given attribute). For high-cardinality values, you may need more local optimization strategies, such as bucketing and sorting, which are presented as the second family of data storage patterns.

Besides organizing the data, there are other approaches to improving the user experience that you will see next in this chapter. They include the following:

- Leveraging the metadata layer to avoid unnecessary data-related operations
- Running costly operations only once, hence materializing them for subsequent readers
- Simplifying the data preparation step by avoiding costly listing operations

Finally, you'll also see two data representation approaches. The Normalizer approach favors data consistency, and the Denormalizer approach trades consistency for better execution time.

If you are impatient to see how to put data storage strategies into action, let's move on to the partitioning patterns!

Partitioning

When you define your storage layer's layout, the first question you'll need to answer is, what are the best ways to divide the dataset to make it easily accessible? The answer consists of two patterns that are responsible for horizontal and vertical organization.

Pattern: Horizontal Partitioner

Among these approaches to data organization, horizontal organization is probably the most commonly used due to the simplicity of its implementation and its long-term popularity since the early days of data engineering.

Problem

You created a batch job that computes rolling aggregates for the previous four days. It ran fine for a few months, but when more data began arriving in your storage layer, the job's performance declined. The biggest issue you spotted is increased execution time for the filtering operation to ignore records older than four days.

To mitigate the problem temporarily, you added more compute power to the job's cluster. However, that increased your costs. You have to find a better approach that will keep the cost as low as it was in the beginning and reduce the execution time despite new ingested data.

Solution

The rolling aggregation from the problem statement is an example of incremental data processing that uses only a portion of the whole dataset. It's a perfect condition in which to use the Horizontal Partitioner pattern and balance execution time with costs.

The solution requires identifying a partitioning attribute, which is also known as a *distribution key*. The data ingestion process or the data store will later use this attribute to save the dataset to a physically isolated storage space for each partitioning value.

Time-based partitions are popular and illustrate the horizontal parameter. As in our problem statement, they define time boundaries for the data processing step, letting you query the relevant information in a fast and cheap manner. In that context, the time attribute can come from either of the following:

The job execution context

In this situation, the partitioning relies on the job's execution time, and the partition value will be the same for all records. For example, for a job executed on 2024-12-31, all records will land in the same partition corresponding to the run date.

The dataset

In this case, the partitioning logic reasons in terms of event time. Due to the late data phenomena described in "Late Data", the partitioned dataset may contain values for different partitions.

Despite their popularity, time properties are not the only possibilities for partitioning keys. You can also use business keys, such as customer ID, partner ID, or the customer's geographical region. You can go even further and create nested partitioning schemas, for example, by combining time- and business-based attributes. Example 8-1 shows a nested

partitioning based on event time partitions and the user country attribute on top of a file system storage

Example 8-1. Dataset partitioned by event time and country attributes

```
visits/
└── 2024
    └── 05
        └── 05
            ├── france
            ├── india
            ├── poland
            └── usa
```

You can set partitions in a declarative way (i.e., while you create a table). That's the case with Databricks' or GCP BigQuery's CREATE TABLE ... PARTITIONED BY statement. In this approach, the data producer doesn't need to know anything about the underlying partitioning, and it could skip defining the partition value during the data ingestion. This flexibility doesn't exist in the opposite mode, where the partitioning logic comes from the data producer. An example here is Apache Spark with the partitionBy method, which creates partitions from an existing column that itself can be the result of a more or less complex computation. You can use the same dynamic logic in Apache Kafka, where you can customize the partitioning logic by creating your own partitioner class.

In addition to creating the partitions, some data stores also manage partition metadata, including the last update time, the number of rows, and even the creation time. This kind of information is available on GCP BigQuery from the INFORMATION_SCHEMA.PARTITIONS view, on Databricks as part of the output for the DESCRIBE TABLE EXTENDED command, and even on Apache Iceberg with the partitions view (SELECT * FROM a_catalog.a_namespace.a_table.partitions).

In addition to optimizing data retrieval, horizontal partitioning acts as an important component of idempotency. The Fast Metadata Cleaner pattern is one example of how to leverage partitioning to enable idempotent pipelines.

Consequences

Paradoxically, the biggest drawback of the Horizontal Partitioner is...horizontal partitions and, more specifically, their static character.

Granularity and metadata overhead

A partition is a physical location storing similar entities sharing the same value for one attribute. Consequently, having too many partitions will have a negative impact on the database.

To help you understand this better, let's take a look at an example of a visits dataset from our case study. If our website is visited by one million unique users daily and we partition

the dataset by username, the Horizontal Partitioner will create one million partitions. This will result in slow partitions listing operations and many small files to read (cf. the small files problem described in "Pattern: Compactor").

For this reason, a good rule of thumb is to use low-cardinality attributes, which are attributes with few distinct values. Using the event time rounded to the nearest hour or day is a great example of this because typically, you get one day or one hour, and thus one partition, for a bunch of records. On the other hand, using the ID number for IoT devices will result in thousands of small partitions. For them, a better choice is to rely on the Bucket design pattern described later in this chapter.

Skew

You may be thinking that horizontal partitioning guarantees even data distribution, but that's not always true.

Skewed partitions can often be a source of latency issues. A good example here is the microbatch stream processing model, which incrementally processes small batches of records. It processes these small batches in a blocking manner (i.e., the next microbatch can't run as long as the previous batch isn't completed).

If one partition in the microbatch is unbalanced, the unbalanced partition will determine the duration of the microbatch. Put differently, it'll block shorter partitions from being processed early as they will have to wait for the unbalanced partition to complete before moving on. To mitigate this issue, you can apply a backpressure mechanism that will store all extraneous records from the skewed partition in a separate buffer and process them only in the next microbatch. Figure 8-1 shows this mechanism with an extra backpressure buffer considered as an optional data source.

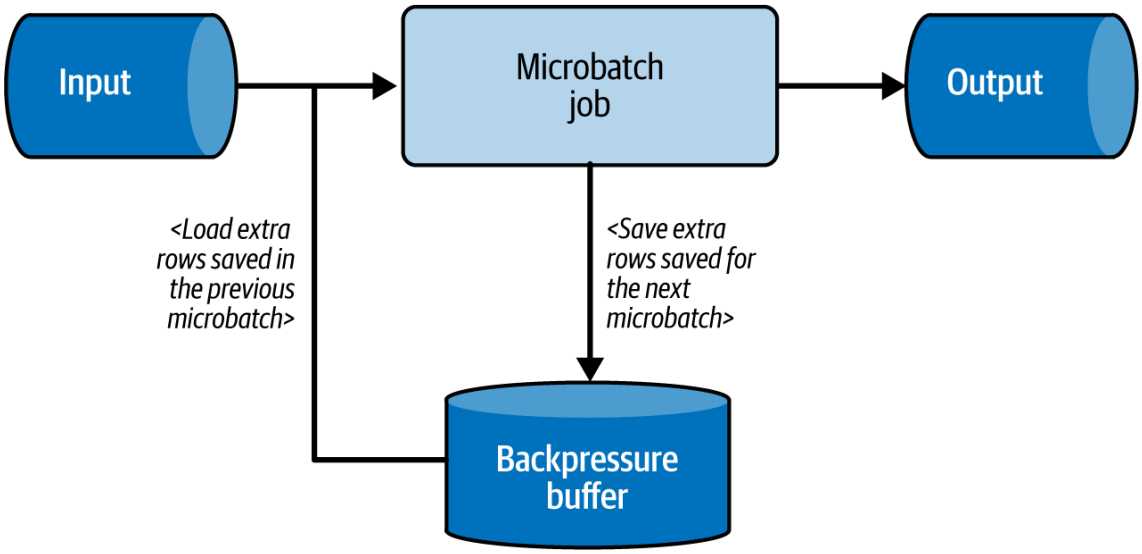


Figure 8-1. Data skew handling for a horizontally partitioned streaming broker

The backpressure buffer will increase overall data delivery latency to the skewed partition as the task will deliver buffered input later. However, this approach guarantees the other tasks can run in close to real time.

Mutability

Changing a partition key is difficult. It requires moving all already written data to a different location, which is costly and time-consuming.

Thankfully, some data stores may handle this mutability problem a bit better than others. For example, Apache Iceberg supports changing the partitioning schema at any moment. However, this operation works only at the metadata layer (i.e., it doesn't move the files to the new partition). Consequently, the partitioning storage remains unchanged for the old records, and the new organization applies only to the records created after the partition evolution.

Horizontal Partitioning Versus Sharding

Sharding consists of splitting a dataset into multiple machines, and it involves physical data division. Horizontal partitioning, although it also divides a dataset into multiple locations, doesn't require data movement across machines. Therefore, sharding is a special type of horizontal partitioning based on the physical (i.e., hardware) layer.

Examples

Let's first discover the Horizontal Partitioner pattern with Apache Spark. This data processing framework has a built-in method called partitionBy that natively splits the written dataset into partitions. Example 8-2 shows an example of this applied to the change_date column.

Example 8-2. Horizontal partitioning with Apache Spark creating granular partitioning columns

```
partitioned_users = (input_users
 .withColumn('year', functions.year('change_date'))
 .withColumn('month', functions.month('change_date'))
 .withColumn('day', functions.day('change_date'))
 .withColumn('hour', functions.hour('change_date')))

(partitioned_users.write.mode('overwrite').format('delta')
 .partitionBy('year', 'month', 'day', 'hour').save(output_dir))
```

After executing this code, the job will create a dataset partitioned by year/month/day/hour, making possible many access patterns that combine the values present in the partitioning path.

The solution is slightly different for Apache Kafka, where you can implement a custom partitioning logic with a custom partitioner. Example 8-3, which is written in Java due to partitioner implementation constraints, shows an example of a custom partitioning logic writing the records to partition 0 or 1, depending on the record's key.

Example 8-3. Custom Apache Kafka partitioner

```
public class RangePartitioner implements Partitioner {
```

```
 private static final int DEFAULT_PARTITION = 1;
 private final static Map<String, Integer> RANGES_PER_PARTITIONS = new HashMap<>();
 static {
  RANGES_PER_PARTITIONS.put("A", 0);
  RANGES_PER_PARTITIONS.put("B", 0);
 }

 @Override
 public int partition(String topic, Object key, byte[] keyBytes,
  Object value, byte[] valueBytes, Cluster cluster) {
   String keyAsString = key.toString();
   return RANGES_PER_PARTITIONS.getOrDefault(keyAsString, DEFAULT_PARTITION);
 }
// ...
```

To declare your custom partitioner, you need to reference the created class in the partitioner.class property (see Example 8-4).

Example 8-4. Customizing horizontal partitioning at the producer level

```
Properties props = new Properties();
// ...
props.put("partitioner.class", "com.waitingforcode.RangePartitioner");
```

Keep It Simple!

Keep in mind that any code increases complexity. That's why it's always good to favor simplicity and add code (and thus complexity) only when necessary. As a result, most of the time, you will stick to the default partitioners in Apache Kafka.

In addition to Apache Spark and Apache Kafka, Horizontal Partitioner is present in relational databases. Example 8-5 shows an example of a PostgreSQL table storing website visits. Each partition is responsible for keeping visits from a different day.

Example 8-5. Range partitioning logic for date times in PostgreSQL

```
CREATE TABLE visits_all (
 visit_id CHAR(36) NOT NULL,
 event_time TIMESTAMP NOT NULL,
 user_id  TEXT NOT NULL,
```

```
  page VARCHAR(20) NULL,
  PRIMARY KEY(visit_id, event_time)
) PARTITION BY RANGE(event_time);

CREATE TABLE visits_all_20231124 PARTITION OF visits_all
FOR VALUES FROM('2023-11-24 00:00:00') TO ('2023-11-24 23:59:59')

CREATE TABLE visits_all_20231125 PARTITION OF visits_all
FOR VALUES FROM('2023-11-25 00:00:00') TO ('2023-11-25 23:59:59')
```

Pattern: Vertical Partitioner

As you've seen, the Horizontal Partitioner pattern processes whole rows each time. The next partitioning pattern is its alternative because it divides each row and writes the separate parts to different places, such as tables or files.

For Storage and Security

The Vertical Partitioner pattern presented in Chapter 7 is a specialization of vertical partitioning applied to security. The Vertical Partitioner presented in this chapter is a specialization dedicated to data storage.

Problem

In one of your pipelines, you track user visits to your website. The visits dataset has two categories of attributes: mutable ones that change at each visit (such as visit time or visited page) and immutable ones that remain the same throughout the visit (like IP address). You're looking for a way to avoid duplicating the immutable information and store it only once for each visit.

Solution

Having two types of attributes like in our problem statement is the perfect condition to use the Vertical Partitioner pattern.

The implementation begins with data classification, where you need to put related attributes together. For the announced problem statement, you would divide the attributes into the mutable and immutable groups. In addition to those groups, you need to identify an attribute that you're going to use to combine them if needed. In our example, it'll be the visit ID.

Once this specification step is completed, your data processing job will write the grouped attributes for each row into dedicated locations, such as tables in a data store or directories in a file system.

In addition to optimizing storage costs, the Vertical Partitioner pattern brings flexibility. Because a row is now divided, you can easily apply different data retention or data access policies to it. That would be more challenging to put in place if the row were kept undivided.

To sum up, the difference with the Horizontal Partitioner pattern comes from the partitioning heuristic. As demonstrated in Figure 8-2, the horizontal approach applies the partitioning rule to a whole row by moving it fully to a different location. On the other hand, the vertical logic splits a row and writes it to different locations.

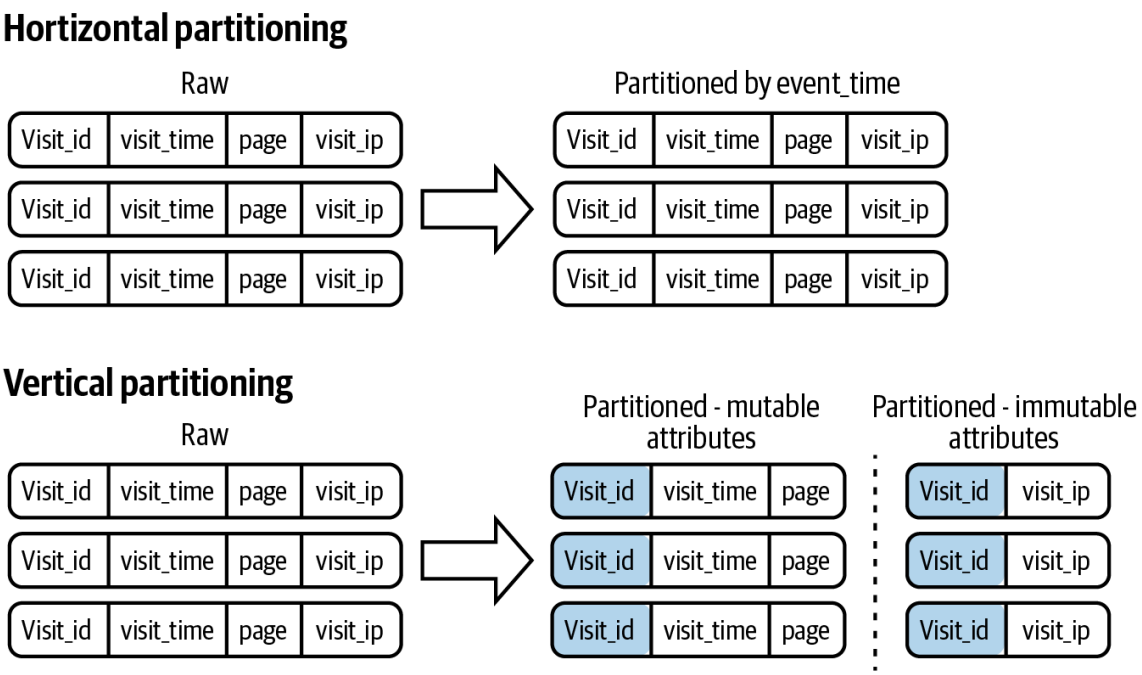


Figure 8-2. A visit row partitioned horizontally and vertically (note that the dashed line separates two partitioning locations, and the filled boxes represent unique IDs used to recombine the divided row)

Consequences

However, the Vertical Partitioner pattern has some logical implications in the following areas.

Domain split

Since each row is split apart, there may be logically related attributes that are stored in two separate places. It may not be easy to find them, and good documentation support for your end user will be key.

Querying

This drawback results from the domain split. As each row is separated, it gets harder to get the full picture than in a horizontally partitioned dataset. To mitigate this issue, you can expose the data from a view combining all tables for the vertically partitioned entity (for example, with the Dataset Materializer pattern).

Data producer

In addition to the consumers, the Vertical Partitioner impacts the producers, who from now on can't simply take a row and write it elsewhere. Instead, producers need to implement the row division logic and consequently perform multiple writes at a potentially higher network communication cost.

Examples

Let's begin this section with an Apache Spark example that extracts the user and technical visit context into two different tables. Although this task sounds easy, you must remember to call the persist() function so that the input dataset doesn't get read twice. Later, you need to build both tables by using drop() and select() functions to, respectively, remove and select columns. Example 8-6 shows this logic implemented.

Example 8-6. Vertical Partitioner in Apache Spark

```
visits = spark_session.read.schema(visit_schema).json(input_location)

visits.persist()


visits_without_user_technical_context = (visits.drop('user_id')

 .withColumn('context', F.col('context').dropFields('user'))

 .withColumn('context', F.col('context').dropFields('technical')))

visits_without_user_technical_context.write.format('delta').save(output_dir)


(visits.selectExpr('visit_id', 'context.user.*', 'user_id').dropDuplicates()

.write.format('delta').save(get_delta_users_table_dir()))


(visits.selectExpr('visit_id', 'context.technical.*').dropDuplicates()

.write.format('delta').save(get_delta_technical_table_dir()))


visits.unpersist()
```

When it comes to a SQL-based implementation, let's see what commands can help you implement the pattern in PostgreSQL. The first command uses the INSERT INTO…SELECT FROM operation. Here, instead of declaring each row to insert explicitly, you delegate this declaration task to the dynamic SELECT query. Example 8-7 shows this in action.

Example 8-7. Inserting technical visit context with INSERT INTO…SELECT FROM

```
INSERT INTO dedp.technical (visit_id, browser, browser_version, ...)

 (SELECT DISTINCT visit_id, context->'technical'->>'browser',

   context->'technical'->>'browser_version', ...

 FROM dedp.visits_all);
```

Also, you can use a different approach that creates the vertically partitioned table from a SELECT statement. This is commonly known as a CREATE TABLE AS SELECT (CTAS) construction, an example of which is presented in Example 8-8.

Example 8-8. CTAS construction for the technical context of a vertically partitioned visit

```
CREATE TABLE dedp.technical_select AS (SELECT DISTINCT
 visit_id, context->'technical'->>'browser' AS browser,
 context->'technical'->>'browser_version' AS browser_version, ...
 FROM dedp.visits_all;
```

Records Organization

Partitioning is often the first step in organizing data. But as you've seen, it's rather rudimentary as it moves either full or partial records to different locations. Moreover, you can't use it on all attributes. For example, you've seen that high-cardinality values are not well suited to horizontal partitioning. The next category of patterns goes one step further because it applies some smart optimizations for records colocation, addressing, among other things, the cardinality issues of the Horizontal Partitioner pattern.

Pattern: Bucket

If, for whatever reason, you need to improve access to a column with high cardinality, such as a unique user ID, there is hope. Instead of colocating rows in the same storage space with partitioning, you can colocate groups of rows. That's an oversimplified definition of what the next pattern does.

Problem

The dataset you're modeling has a business attribute that is frequently used in queries as part of the predicate. Initially, you wanted to use this attribute as a partitioning column, but its cardinality is too high. It would result in too many partitions that at some point could reach your data store metadata limits. As 80% of operations rely on this high-cardinality attribute, you still want to optimize storage, but at the moment, you don't know how.

Solution

The fact that you've got a high-cardinality column that is often involved in queries is a good reason to use the Bucket pattern. Although on the surface it also stores records in a dedicated location, unlike Horizontal Partitioner, it colocates different values in the same storage area.

As for the two partitioning patterns, the Bucket pattern's implementation starts with the data analysis step that defines the column(s) to use for bucketing. If a dataset is already partitioned with the horizontal or vertical approach, you can consider these attributes as a kind of secondary set of grouping keys (the partition key being the primary key), which are more commonly known as bucket columns.

Next, you might also need to set the number of buckets you want to create. The number depends on your bucket key's cardinality. If the cardinality is really high, it means you have a lot of unique values. A higher number would mean more smaller buckets, while a lower number would create fewer bigger buckets. This dependency comes from the grouping formula that applies a modular hashing so that the bucket number for each key is computed as hash(key) % buckets number.

Grouping records enables two optimization techniques for consumers:

Bucket pruning

Whenever a bucket column is used as a predicate in the query, the query execution engine can directly use the bucketing algorithm and eliminate all buckets without the required keys. This may cause a significant performance boost for all filtering operations.

Network exchange (shuffle) elimination

This applies to JOIN operations using the identical bucketing configuration on both sides of the join. That way, the query runner can leverage the buckets to directly load correlated records from each dataset to the same join process, thus combining them without the network exchange you discovered while you were reading about the Distributed Aggregator pattern. Figure 8-3 illustrates this optimization strategy.

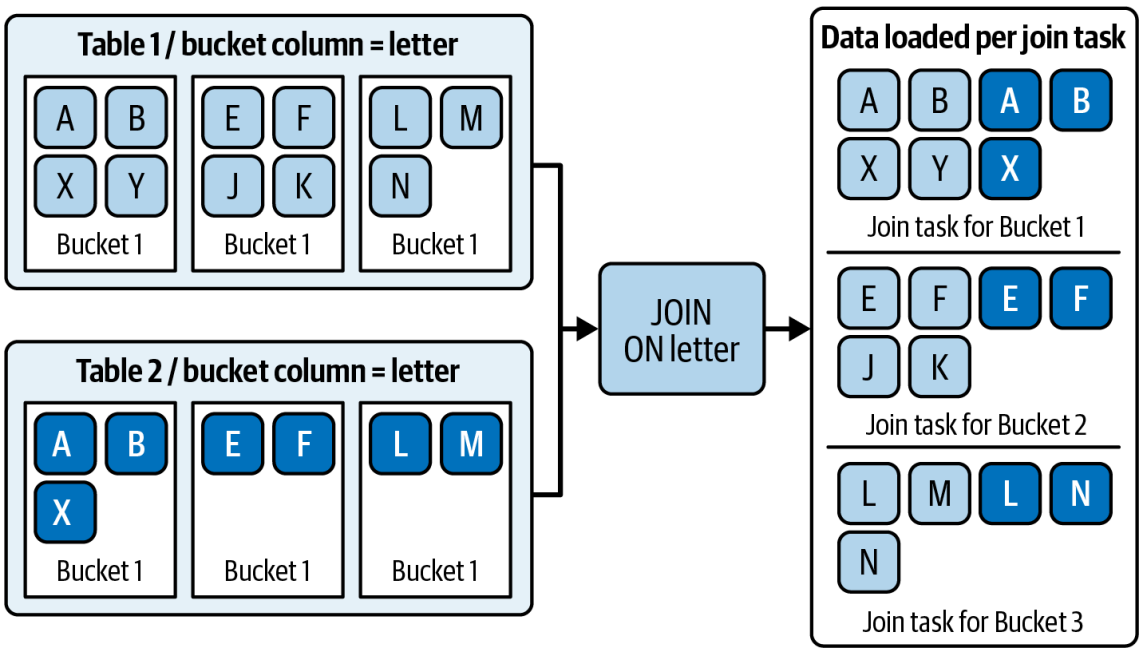


Figure 8-3. Distributed join without shuffle on top of identically bucketed tables

Historically, the bucketing feature was made popular by Apache Hive, but since then, it has been integrated into modern data solutions, including Apache Spark and AWS Athena.

Consequences

Yet again, the data is static, and that's one of the biggest issues with the Bucket pattern.

Mutability

The bucketing schema is immutable. Technically, it's possible to modify it by either changing the column or bucket size, but that's a costly operation requiring backfilling the dataset.

Bucket size

The Bucket pattern requires setting the bucket size. Unfortunately, finding the right size is challenging if you expect to get more data in the future. If you rely on the current data volume, in the future, you'll create big buckets. On the other hand, if you try to predict the number, there's no guarantee that your prediction will be accurate, and in the meantime, the writers may create more buckets than necessary. Both techniques are acceptable ways to mitigate the problem, but as you can see, they both have some gotchas.

Examples

Amazon Athena is a serverless query service implementing the Bucket pattern at the logical level. Put differently, it doesn't write any data. Instead, it only applies the existing bucketing logic to the tables already stored on S3. For that reason, if you issue an INSERT INTO query into a bucketed table, you will get an error.

To configure a table as a bucketed table, you have to define the bucket columns in the CLUSTERED BY statement, plus set the bucketing format. In Example 8-9, the visits table is bucketed by the user_id column in the Apache Spark format.

Example 8-9. Bucketing configuration in AWS Athena

```
CREATE EXTERNAL TABLE visits (...) ...

CLUSTERED BY (`user_id`) INTO 50 BUCKETS

TBLPROPERTIES ('bucketing_format' = 'spark')
```

Apache Spark creates a bucketed table by calling the bucketBy function, which applies the modulo-based algorithm mentioned in the implementation section to the bucket columns (see Example 8-10).

Example 8-10. Bucketing in Apache Spark

```
input_dataset.write.bucketBy(50, 'user_id').saveAsTable(table_name)
```

Pattern: Sorter

Colocating groups of records in buckets is not the only storage optimization technique. Another technique that helps eliminate data blocks that are irrelevant to queries relies on data storage order.

Problem

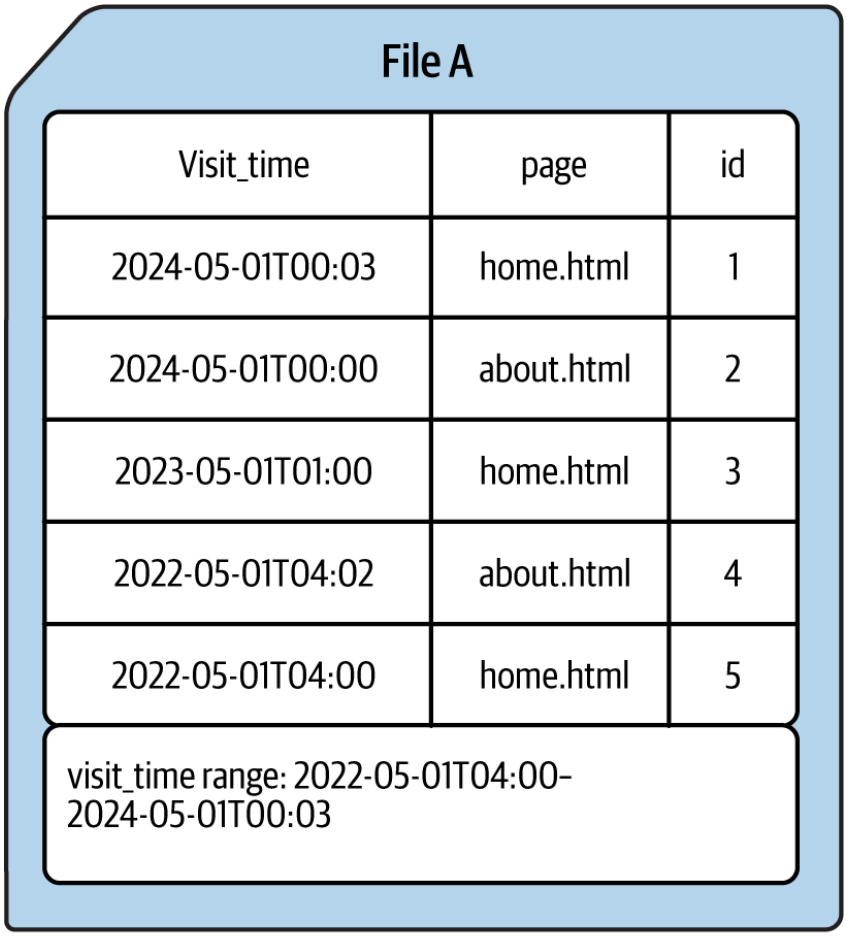
You decided to store data in weekly tables to leverage the Fast Metadata Cleaner pattern. Although it made your daily maintenance task less painful, it didn't improve the query execution time. You don't want to change this idempotency strategy, but at the same time, you would like to reduce data access latency. For that reason, you're looking for a solution that could speed up query execution. The good news is that you know the types of users' queries. Most of them will filter or sort by the event time column.

Solution

Knowing which column or columns are commonly used in sorting or filtering is a good way to implement the Sorter pattern to optimize data access.

You start the implementation by identifying the sorting column or columns. Next, you have to declare the sorting column(s) in the table's creation query. Thereafter, the database will take care of organizing the written rows according to the defined order.

Thanks to the sorted storage, any query targeting the sorting column(s) will be able to skip irrelevant data blocks, very often thanks to the metadata information associated with each of them. Figure 8-4 illustrates this optimization, and if you need more details, you'll find them in "Pattern: Metadata Enhancer".


File A
Visit_time
page
id
2024-05-01T00:03
home.html
1
2024-05-01T00:00
about.html
2
2023-05-01T01:00
home.html
3
2022-05-01T04:02
about.html
4
2022-05-01T04:00
home.html
5
visit_time range: 2022-05-01T04:00–
2024-05-01T00:03

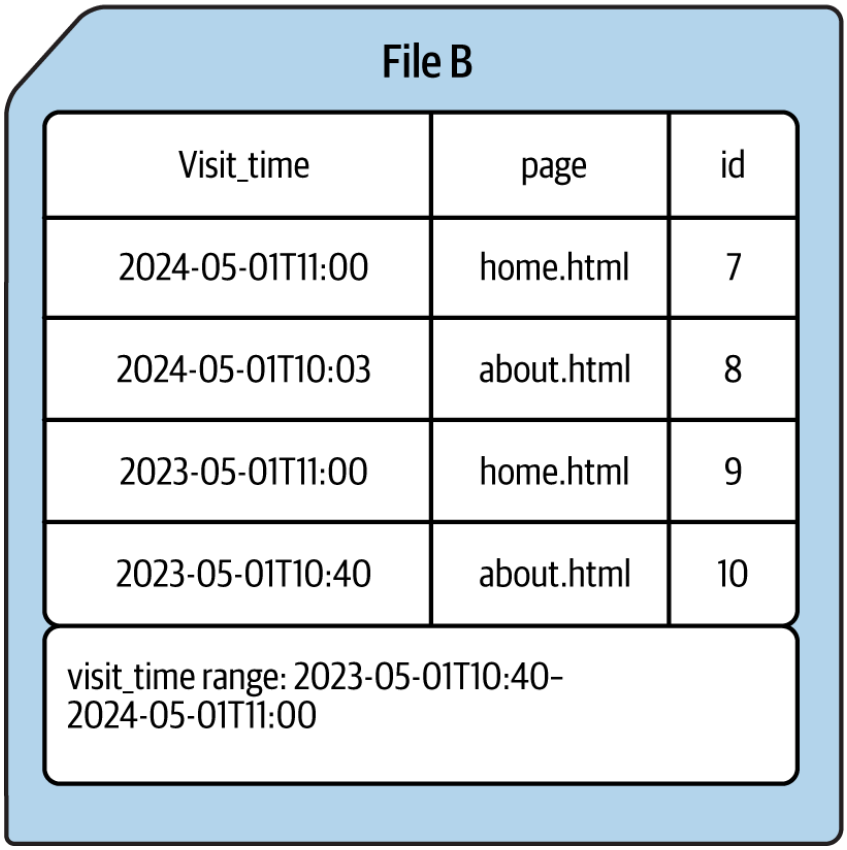

File B
Visit_time
page
id
2024-05-01T11:00
home.html
7
2024-05-01T10:03
about.html
8
2023-05-01T11:00
home.html
9
2023-05-01T10:40
about.html
10
visit_time range: 2023-05-01T10:40–
2024-05-01T11:00

Figure 8-4. Metadata information for data skipping (note that if a query targets visit_time within one of the ranges, the query engine can avoid processing one of the files)

Curved sorts is a variant of the classical top-to-bottom sorting algorithm, where the results are sorted vertically. A popular example of this, especially thanks to recent advances in the table file formats space, is Z-order. Instead of lexicographical order, this method colocates rows from x-dimensional space.

Explaining the Z-order algorithm in detail is out of scope of this book,[1] but fortunately for you, table file formats like Apache Iceberg and Delta Lake implement it natively. However, it's important to understand why Z-order works better than lexicographical order for multiple columns. To help you grasp this, let's analyze how both methods store a dataset sorted by columns x and y, as shown in Figure 8-5. As you will notice, Z-order reduces the number of data blocks to read thanks to a different, curved data organization layout.

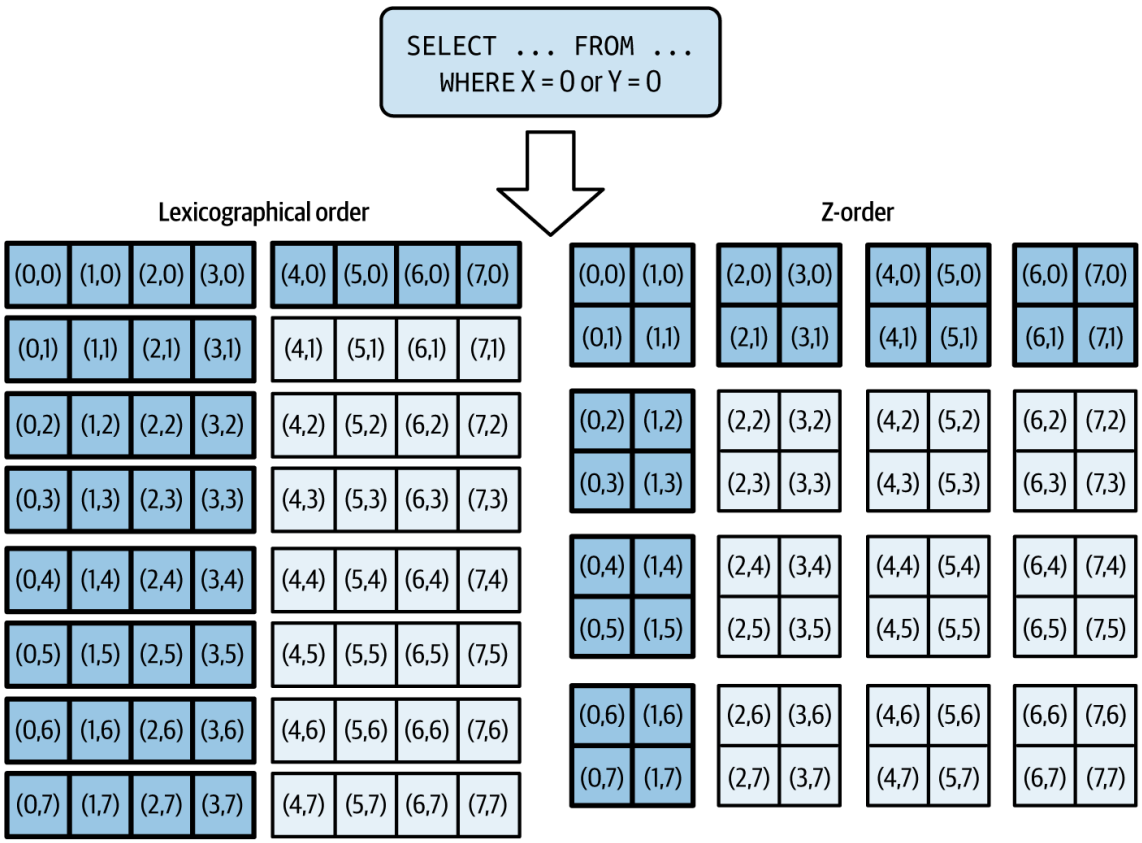


Figure 8-5. Data blocks read for a predicate and two-column sorted datasets (note that lexicographical order reads nine data blocks, while Z-order only reads seven)

Z-order became famous with Delta Lake and Apache Iceberg, but it has been around for longer. Among other data stores, Amazon Redshift provides a Z-order-like sort implementation based on Z-curves with the *interleaved sort keys* feature. Classical sorting is present in data warehouses such as GCP BigQuery and Snowflake via clustered tables.

Sorting Versus Clustering

Z-order is also referenced in the context of clustering due to colocating related records in the same files. However, it does this by effectively sorting data on disk, like a lexicographical sort would do. For that reason, Z-order is classified here as an example of the Sorter pattern.

Consequences

A presorted dataset has a positive impact on the reader's performance. However, it negatively impacts the writer.

Unsorted segments

Sorting may not always be an instantaneous activity. This means that whenever you write new records, there will be some unsorted blocks that will not benefit from the Sorter pattern's optimizations. To mitigate the issue, you may need to schedule the sorting actions in the data writing job or outside of it. Keep in mind that integrating the sorting action with the data writing process will impact the execution time.

Composite sort keys

When you use composite sort keys in the lexicographical order method, keep in mind that the queries should always reference the sorting columns preceding the one(s) you're targeting. Otherwise, despite sort declaration, the query engine will still need to iterate over most of the data blocks. Let's illustrate that with a simple case of a sort key composed of a visit time and a page ID column. Figure 8-6 points out the rows involved when the query targets both columns, and the rows impacted by the read when the query filters only on the page ID.

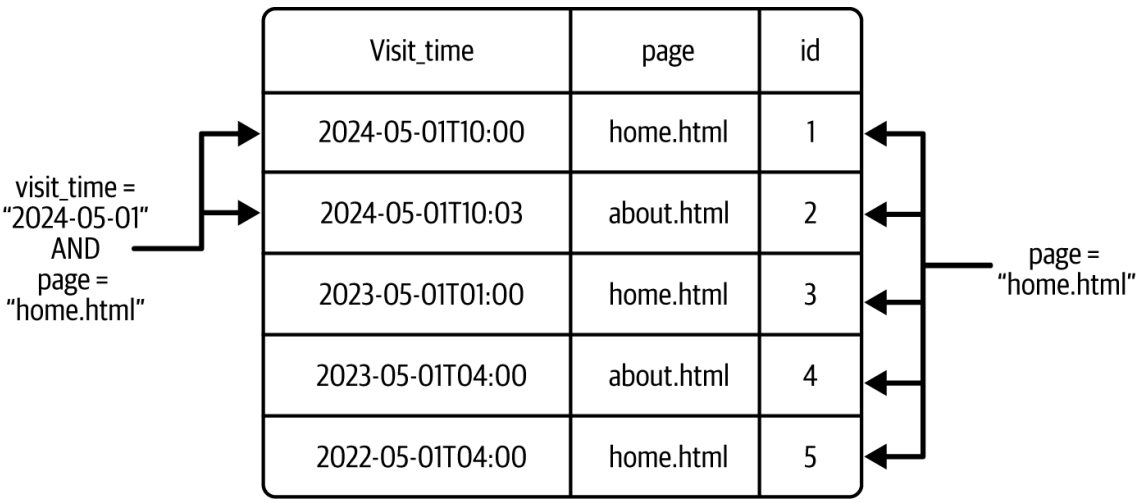


Figure 8-6. A table sorted by visit_time and page, and rows impacted by querying both columns (left side) or only one of the columns (right side)

Mutability

Although it's often possible to change the sorting keys after creating them, you must be aware that the operation may need to sort the entire table. Depending on the table's size, this can be costly.

Examples

Let's start this section with a cloud example. GCP BigQuery implements the Sorter pattern via clustered tables. A clustered table requires the declaration of the sorting columns as part of the CLUSTER BY statement (see Example 8-11).

Example 8-11. Clustered table for visit_id **and page columns in BigQuery**

```
CREATE TABLE `dedp.visits.raw_visits`

PARTITION BY DATE(event_time)

CLUSTER BY visit_id, page
```

Although the clustered table will improve performance when it comes to targeting the visit_id and page columns, it will not help that much if you only need to filter on the page column. Curved sorts solve this issue. Let's see how by using a Delta Lake Z-order compaction. Creating a Z-order-compacted table requires calling the optimized API with the columns that should be used to create this curved distribution. Example 8-12 shows this initialization step.

Example 8-12. Z-order compaction with Delta Lake for the visit_id **and page columns**

```
DeltaTable.forPath(spark, output_dir)

 .optimize().executeZOrderBy(['visit_id', 'page'])
```

As a result, Delta Lake will compact data files to better organize the records inside the rewritten files.

Read Performance Optimization

The patterns from this section extend the data organization techniques presented so far to optimize data access.

Pattern: Metadata Enhancer

The first technique you can leverage to optimize reading performance uses metadata. This is one of the reasons why columnar file formats such as Apache Parquet have been viewed as disruptive changes in the data engineering field for many years.

Problem

You partitioned your JSON dataset horizontally by event time, hoping to reduce the execution time of batch jobs. And it worked! However, your company then hired new data analysts who are also working on the same partitioned dataset but are targeting only a small subset of rows from one partition.

Since the partitions are big, data analysts complain about the query execution latency and increased cloud bills as they're relying on a pay-as-you-go querying service. After the first analysis, you find out that the data analysts' queries always load the full dataset and only later apply the filtering logic. You would like to reverse these two operations and apply the filtering logic before loading the dataset into the query engine.

Solution

An easy way to optimize the query execution time and cost is to skip all irrelevant data files before loading them for processing. That's where the Metadata Enhancer pattern comes into play.

The implementation consists of collecting and persisting statistics about the stored records in a file or database. Since we mentioned the files in the problem statement, let's discover this integration first.

The Metadata Enhancer implementation for files applies to columnar file formats such as Apache Parquet, in which each data file contains a footer with additional metadata. As per its name, the columnar file format stores a column in each file. The statistics are local to the file (i.e., they describe only the values from the file). Figure 8-7 shows a simplified version for an age column with attached statistics.

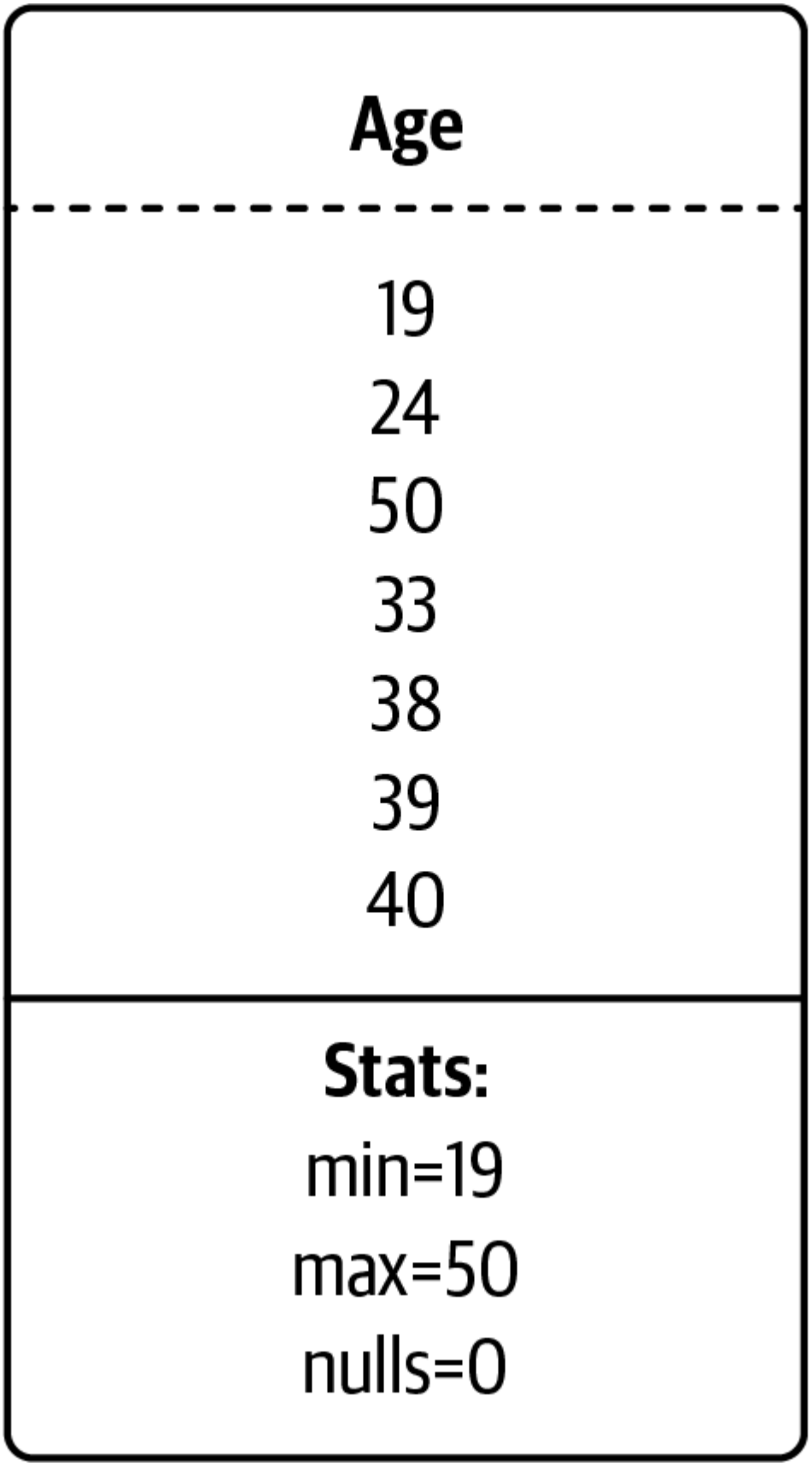


Figure 8-7. A simplified example of the metadata footer with a data summary for the stored records in an Apache Parquet file

As you've likely noticed, the footer includes a range of possible values that are automatically computed while a file is created by the data producer. Now, when a user queries the age column as part of the predicate (for example, SELECT … FROM table WHERE age > 50), the query execution engine can simply verify in the metadata footer whether the requested age is included in the file. Since the footer is smaller than the data block, the filtering operation relies on a reduced dataset and consequently is much faster than opening a larger portion of the data to analyze each entry separately. That said, there is still the overhead of reading all the footers to know where the relevant records are, but the overhead is incomparably smaller than for reading all the data files.

Since Apache Parquet is the storage format used by table file formats, the pattern is automatically available on Delta Lake, Apache Iceberg, and Apache Hudi. But in addition to the Apache Parquet statistics, these formats store additional metadata in the commit log that can optimize readers. Some of this metadata consists of numbers, such as the number of rows created in the commit, the minimum and maximum values per column, and even the number of NULLs in a given column. That way, users can pretty quickly perform queries counting the number of elements or filtering on nonexistent values.

But files are not the only place where the Metadata Enhancer applies. You can find the same kind of statistics for tables in relational databases and data warehouses. The statistics in that context will often be located in a separate table that the query planner will leverage to create the most efficient execution plans.

Consequences

Although it's hard to find drawbacks for the Metadata Enhancer, there is a little one related to the cost of this additional layer.

Overhead

When it comes to columnar file formats, building statistics at writing time is an extra operation the writing job must perform. It can slightly impact the processing time because for each processed column, the job must keep the configured stats.

Additionally, for relational databases and data warehouses, the data store must keep the statistics up-to-date. Otherwise, the execution plan might be far from the most optimal one. To address this issue when statistics are out of date, you can run a command that's responsible for refreshing them.

Out-of-date statistics

Even though statistics are updated automatically for relational databases and data warehouses, the update process may not be immediate. Often, its execution is controlled by certain thresholds, such as the number of rows that have been modified since the last update. Consequently, if your table undergoes small changes from time to time that don't reach the thresholds, the statistics can become out of date over time.

To mitigate this issue, you can refresh the statistics manually with commands like ANALYZE TABLE. But keep in mind that this might add temporary read overhead on the database to process the table and generate updated statistics.

Examples

To start this section, let's look at the most basic example with Apache Parquet. The writing step in Apache Spark requires using an appropriate data writer (see Example 8-13).

Example 8-13. Writing an Apache Parquet file

```
input_dataset.write.mode('overwrite').parquet(path=get_parquet_dir())
```

Statistics are created for you under the hood. You can see their content by running a Docker command like the one in Example 8-14.

Example 8-14. Apache Parquet metadata analyzer command

```
docker run --rm -v "./output-parquet:/tmp/parquet"
 hangxie/parquet-tools:v1.20.7  meta
 /tmp/parquet/part-00001-3c52ae6f-aeea-4364-aac3-7fc69d63e898-
c000.snappy.parquet
```

The output should print statistics for each column. Example 8-15 shows this for the ID column.

Example 8-15. Apache Parquet statistics for the login column

```
"NumRowGroups": 1, {"PathInSchema": ["Id"], "Type": "BYTE_ARRAY",
 "Encodings": ["PLAIN", "RLE", "BIT_PACKED"],"CompressedSize": 180463,
 "UncompressedSize": 200035,"NumValues": 5000,
 "NullCount": 0, "MaxValue": "fffbe4f8-8d88-43d2-a9a5-54bf536de75b",
 "MinValue": "0018e1dc-1b80-4410-92f6-5261d2dadf35",
 "CompressionCodec": "SNAPPY"}
```

Delta Lake adds an extra layer on top of the Apache Parquet metadata. This layer is present in the commit logs and also contains metadata to accelerate data processing operations. As for Parquet, you don't need to generate those values explicitly. That's done by the data processing framework under the hood. As a result, you should receive entries like those in Example 8-16.

Example 8-16. Statistics in the Delta Lake commit log

```
{"commitInfo":{"timestamp":1716954694590,"operation":"WRITE",
"operationMetrics":{"numFiles":"1",
"numOutputRows":"6100",
"numOutputBytes":"50437"}," ...}
{"add":{"path":"part-...-c000.snappy.parquet, "size":50437,
"stats":"{
 \"numRecords\":6100,
 \"minValues\":
```

{\"type\":\"galaxy\",\"full_name\":\"APPLE iPhone 11 (White, 64 GB)\",
\"version\":\"Android 10\"},
\"maxValues\":
{\"type\":\"mac\",\"full_name\":\"Yoga 7i (14\\\" Intel) 2 in 1 Lapto\",
\"version\":\"v17169535721658688\"},
\"nullCount\":{\"type\":0,\"full_name\":0,\"version\":0}"}}

Pattern: Dataset Materializer

Costly operations pose another challenge to improving data access. If you need to write a query that involves some shuffle and CPU-intensive transformations, and if you need to run the same query over and over again, performance may suffer. Surprisingly, you could benefit from data duplication to improve data reading performance.

Problem

You wanted to simplify the process of querying multiple partitioned tables of the same dataset to get the past three weeks of data. You created a view, but consumers weren't fully satisfied. They complained about latency, and because the view runs the underlying query each time, you can see their point. However, you want to solve this issue and provide them with a better-performing single point of data access.

Solution

When the computation of results is slow, the simplest solution is to avoid the problem by materializing the data. That's what the Dataset Materializer pattern does.

The implementation starts by identifying the datasets that should be materialized. After the identification step, you need to implement the materialization. Typically, this will involve querying the data with the appropriate SELECT statements and maybe combining multiple datasets with a UNION or JOIN operation. Then, the created query is later used to materialize the dataset as a materialized view or a table in your database.

Which of the materialized view and table techniques should you chose? The biggest difference between them is related to refreshes. Manual refreshes of materialized views are possible, but modern data warehousing solutions support automatic refreshes under some criteria as well. For example, Amazon Redshift supports this feature via an AUTO REFRESH YES option defined in the CREATE MATERIALIZED VIEW statement. However, the refresh isn't meant to be run immediately after you change the underlying tables. Its execution depends on the current workload on the database or the size of the data to refresh. Therefore, the logic, albeit automated, is less predictable. Besides Redshift, materialized views are available in other data warehousing solutions, including GCP BigQuery, Databricks, and Snowflake.

On the other hand, when you use a table as the storage for the materialized dataset, you'll be responsible for refreshes, without the possibility of leveraging any automatic refresh feature. In exchange for this extra work, you get extra flexibility as the table may benefit from other storage optimization techniques—including partitions, buckets, and sorting—which

may not be available for a materialized view. All this gives you more work to do but also provides more operational flexibility and optimization techniques.

Consequences

While you may be thinking that refreshing is not an issue, I have bad news. It may be.

Refresh cost

As you can imagine, whenever you need to refresh the view, you need to rerun the creation query. If this setup query is costly, perhaps because of the data volume or the type of operations, it'll impact the resources of your database, including the ones available for regular users interacting with other tables.

To overcome this issue, you can use an incremental refresh (i.e., integrate only the most recent changes into the view). This fits perfectly into insert-only workloads where historical data doesn't change and the refresh only appends the new records.

Modern data warehousing solutions support incremental refreshes out of the box. That's the case with Databricks and GCP BigQuery. However, their incremental refreshes don't support all SQL operations, and sometimes, they will still refresh the whole dataset.

Data access

Because the materialized dataset combines multiple tables, it may be challenging to apply consistent data management, including retention or access configuration. Typically, if a user doesn't have access to one of the building tables, you should continue to deny access to the view, or you should implement one of the options in the Fine-Grained Accessor pattern, if possible.

Data storage overhead

Materialization does indeed optimize access, but it trades optimization for storage. If storage is a concern for you, you may opt for a mixed implementation of the Dataset Materializer pattern, in which only some of the view's datasets get materialized and the others live as regular, recomputable parts.

Examples

GCP BigQuery is a cloud-managed data warehouse that not only supports materialized views but also let you configure an automatic refresh of them. Example 8-17 shows a materialized view that's refreshed every 15 minutes.

Example 8-17. Query creating an automatically refreshed materialized view in BigQuery

```
CREATE MATERIALIZED VIEW dedp.visits.visits_enriched

OPTIONS (enable_refresh = true, refresh_interval_minutes = 15)

AS SELECT...
```

There's one thing to notice, though: automatic refreshes are rarely guaranteed to run just after you modify the base table. This is also true for BigQuery, which should run the refresh within five minutes of the change. But if there is not enough capacity, the refresh will be delayed.

That's why as an alternative, you can use a manually refreshed materialized view. PostgreSQL provides a REFRESH MATERIALIZED VIEW command that integrates new data into the view, as shown in Example 8-18.

Example 8-18. Refreshing a materialized view with PostgreSQL

```
REFRESH MATERIALIZED VIEW dedp.windowed_visits WITH DATA;
```

As for the incremental version of the Dataset Materializer pattern, let's analyze how to integrate new visit counts. First, the input table has an insertion_time column that corresponds to the writing time of each row. The idea is to use this column to query only the rows added after the previous execution and combine the result with the existing dataset. As you can see already, the solution combines the Incremental Loader pattern with the Merger pattern.

Example 8-19 shows the SQL query executed at each run to update the number of overall visits per user by combining existing counts with new records.

Example 8-19. Incremental version of the Dataset Materializer pattern

```
MERGE INTO dedp.visits_counter AS target
USING (
 -- 2024-11-09T03:27:32 is the time after the previous insertion_time
 SELECT user_id, COUNT(*) AS visits FROM dedp.visits
 WHERE insertion_time > '2024-11-09T03:27:32' GROUP BY user_id
) AS input
ON target.user_id = input.user_id
WHEN MATCHED THEN UPDATE SET count = count + input.visits
WHEN NOT MATCHED THEN INSERT (user_id, count) VALUES (input.user_id, input.visits)
```

Pattern: Manifest

The last read access performance challenge concerns data listing, which can be slow, especially for object stores with many files because this will result in many API calls. Even though you can try to mitigate this issue by parallelizing the listing operation, there is a better way.

Problem

You have created an Apache Parquet dataset in your object store. Your batch jobs are now performing very well, and their decreased execution time has also reduced your cloud bill. As a result, your company has asked you to create a data warehouse layer. One of the requirements is the exposition of this Apache Parquet dataset to the data analysts team. Unfortunately, when you did your first tests, the execution time was not as good as for the batch job producer. You found out that the slowest operation lists the files to load from your object store, and you would like to avoid this costly step.

Solution

To overcome a repeated listing operation problem, it's better to list files only once or not at all if the data producer can record filenames beforehand. That's the premise of the Manifest pattern.

Table file formats such as Delta Lake, Apache Iceberg, and Apache Hudi are the first implementations of the pattern. They write the list of files created within the given transaction to the commit log stored in the metadata location. That way, when a reader needs to access the data files, it can simply get them from the commit files, without performing any listing of the underlying storage. In the context of the Manifest pattern, these commit logfiles act as manifest files, meaning files providing all necessary and important information about the data.

The alternatives to automatically managed manifests are manually created manifests that may require a prior listing operation. They can be particularly useful if many different readers operate on the same dataset, for example, as part of the Fan-Out patterns in Chapter 6.

In addition to their utility in data reading, manifests can play a crucial role in writing. Amazon Redshift uses a manifest file in the COPY command that loads new data into a table. For each loading operation, you can define a different manifest file with a dedicated list of files to upload. This materialization can be incredibly helpful in implementing idempotent pipelines, like the ones in Chapter 4. A similar implementation exists for the Storage Transfer Service on GCP. This offering relies on manifest listings to copy files from other cloud stores to GCS.

Consequences

As you can see, the pattern offers efficiency and optimization, but there are trade-offs with complexity and overall size.

Complexity

If you need to add the manifest creation step, you'll add some extra complexity to the execution flow. However, manifest creation is a rather simple operation consisting of listing recently written files. Having this extra complexity in the pipeline should be easier to accept than running a slow and unpredictable listing action many times.

Size

Manifests can grow really big. That's particularly apparent if the input location has many small files or if the data producer is a continuous streaming job. In that case, it's common to see manifests of several gigabytes in size. Some of the implementations may have a maximum size limit for a manifest file or a retention configuration for the entries present in the file.

The size issue was present in the early days of Apache Spark Structured Streaming. When you were using the framework to write files, in addition to creating new files in the output location, the job was adding their names to a manifest file. Over time, the manifests were continuously growing, and sometimes the jobs couldn't even restart because the manifests were too big to restore. Since then, the issue has been fixed (see SPARK-27188).

Examples

Let's see how the Manifest pattern can enable two different technologies to work together. The goal of the first example is to create a so-called external table for a Delta Lake dataset in BigQuery. To start, you have to generate the manifest file from Delta Lake. The operation is just a matter of calling a generate function (see Example 8-20).

Example 8-20. Generating a manifest file in Delta Lake

```
devices_table = DeltaTable.forPath(spark_session, DemoConfiguration.DEVICES_TABLE)

devices_table.generate('symlink_format_manifest')
```

The generated manifest contains all files used by the most recent version of the Delta Lake table (aka snapshot). The next thing to do is to reference this file as part of the external table creation statement (see Example 8-21).

Example 8-21. External table creation with a Delta Lake manifest file

```
CREATE EXTERNAL TABLE IF NOT EXISTS `dedp.visits.devices`

...

OPTIONS (

  hive_partition_uri_prefix = "gc://devices",

  uris = ['gc://devices/_symlink_format_manifest/*/manifest'],

  file_set_spec_type = 'NEW_LINE_DELIMITED_MANIFEST',

  format="PARQUET");
```

Another use case of the Manifest pattern occurs in Redshift, which can enforce the idempotency of the COPY command with the list of files to load to the table. If before loading the files, you create the manifest and associate it with the job's execution, you'll be able to use the same manifest file for any replayed job's runs. Example 8-22 shows an example of the operation using a manifest file composed of two required data files.

Example 8-22. Manifest for data loading in Amazon Redshift

```
COPY customer

FROM 's3://devices/manifest_20250601_1031'

...

MANIFEST;

# manifest_20250601_1031

{"entries": [

 {"url":"s3://devices/dataset_1","mandatory":true},

 {"url":"s3://devices/dataset_2","mandatory":true}]}
```

Data Representation

Data storage is not only about organizing storage or optimizing read performance. Both are crucial steps to make a dataset useful, but they're missing one piece: data representation,

which answers the crucial question of what attributes will be stored together and thus what tables you're going to create.

Pattern: Normalizer

The first data representation pattern favors decoupling, which is great for keeping a dataset consistent by not duplicating the information.

Problem

You defined a data model for the visit events from Figure 1-1. A colleague pointed out some data duplication. In fact, your visits table stores event-driven attributes, such as visit time and visited page, but also immutable attributes, such as device name, operating system name, and version. The immutable attributes are repeated for each visit row, leading to increased storage and slow update operations whenever these attributes are modified.

You were asked to review your design and propose a model that addresses the issues of data repetition and slow updates.

Solution

In the context of our problem, the separation can be understood as normalization since we try to reduce repetition by representing each piece of information only once. From there comes the name of the next pattern, the Normalizer pattern.

The Normalizer has two possible implementations called normal forms (NF) and the snowflake schema. Despite their different names and technical details, the two share the same high-level design process that follows these steps:

1. Defining the business entities. First, you establish a list of terms involved in your data model. For the sake of our website visits example, the terms could include *visits*, *devices*, *browsers*, and *link referrals*.
2. Describing the business entities. Then you define the attributes of each entity. For example, if the entity you are describing is a browser, its attributes could include the browser name and version.
3. Defining the relationships among the business entities. Finally, you define the dependencies between the business entities. For example, a visit would depend on the availability of a browser, while a browser would depend on the availability of an operating system.

As for the specific implementations, let's start with the NF-based approach. It's widely used in transactional workloads that mainly involve writing operations occurring at a fast pace. The NF design helps eliminate data quality–related issues by reducing duplicates and, consequently, the volume of data to write. The model respects the following forms:

The first NF

After you apply this form, the columns of your table should have nonrepeating atomic values, and each row should be uniquely identifiable by a primary key.

The second NF

Here, each column must depend only on the primary key. In other words, a nonprimary key attribute must be described by all primary key attributes.

The third NF

This form guarantees that there are no transitive dependencies between nonprimary attributes. In other words, all nonprimary columns should depend only on the primary key.

More Normal Forms

Even though there are more NFs, the three we just explained are the most commonly used. Knowing them should be enough to implement the Normalizer pattern.

To help you understand these NFs better, let's take a look at three examples, each of which shows a broken version of one of the forms:

1. The first NF. Table 8-1 contains repeating attributes in the comments column. To normalize this table, you should extract each comment to a dedicated games_comments table.

Name (primary key)	Comments
Puzzle Tour	["...", "..."]
Runner	["..."]

Table 8-1. The first NF, broken

2. The second NF. Here, you have a table with a composite primary key. As you can see in Table 8-2, our games table has a primary key (PK) composed of the name and platform. According to the second NF, all other columns should fully depend on both key columns. However, that's not the case for the platform language, as it only depends on the game platform. To normalize this table, we should create a new table in which we store the platform and the platform language.

Name (PK)	Platform (PK)	Release year	Platform language
Puzzle Tour	iOS	2023	Swift
Puzzle Tour	Android	2024	Kotlin
Runner	Android	2024	Kotlin

Name (PK)	Platform (PK)	Release year	Platform language

Table 8-2. The second NF, broken

3. The third NF. To help you understand the last form, let's stay with our gaming example. Table 8-3 shows the schema breaking the third NF because of a transitive dependency. This dependency comes from the studio country column as its value doesn't depend on the game's name but on the studio's name. To fix that, we should create a new studios table and put all related information inside it.

Name (PK)	Studio	Studio country
Puzzle Tour	Studio A	Italy
Runner	Studio B	Portugal

Table 8-3. The third NF, broken

The NF model was the first implementation of the Normalizer pattern. Even though this model is often used in transactional workloads of relational databases, it also has a dimensional model variant that's present in analytical workloads. In a nutshell, a *dimensional model* is a design composed of one fact table and multiple dimension tables. A *fact table* represents an observation, such as an ecommerce order or a website visit, while a *dimension table* describes the observation by providing extra context, such as product information for the order or the browser configuration for the visit. Those are the shortest definitions possible, but if you need to know more about fact tables and dimension tables, there is plenty of great reference material.[2]

One of the dimensional models, and (by the way) another implementation of the Normalizer pattern, is the *snowflake model*. In this model, the fact table is described by multiple dimension tables that in turn can be described by other dimension tables. Figure 8-8 shows a central fact table decorated with dimensions.

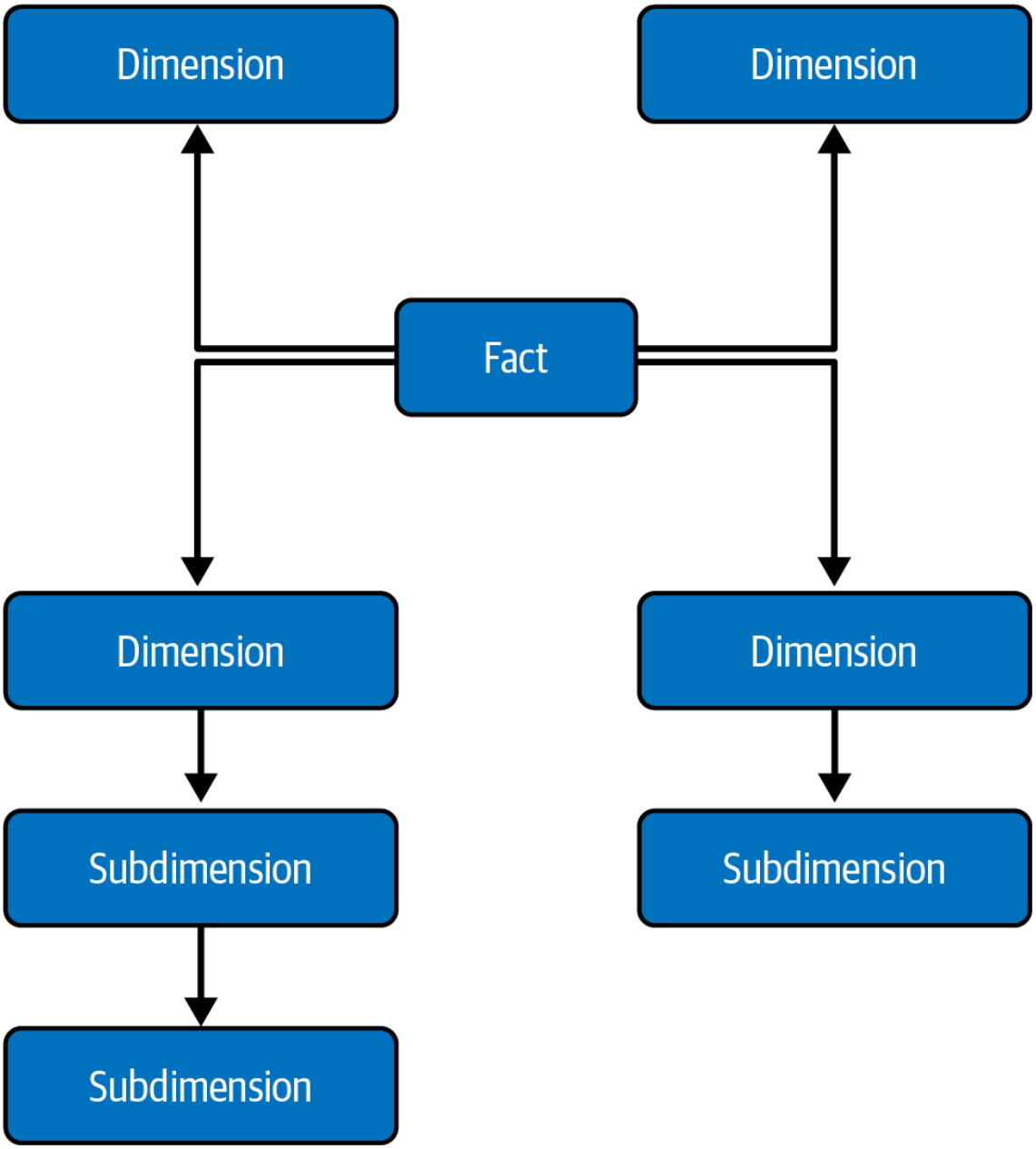


Figure 8-8. Snowflake model where dimension tables are described by other dimensions

As you can see, one dimension like the date can be normalized into subdimensions, such as quarters or months. The snowflake model tends to move attributes repeated multiple times to a dedicated subdimension table.

From both NFs and the snowflake model, you can see that the most important goal of the Normalizer pattern is to prioritize data consistency over any eventual performance optimizations. This means easier updates since any given update will be immediately reflected in all related tables.

Consequences

Complexity is one of the biggest drawbacks here, but that's the price you have to pay if you need to keep the data consistent.

Query cost

The Normalizer pattern favors data split into multiple places. That translates into relying often on JOIN operations for querying data. Unfortunately, joins can be costly in a distributed environment as they involve exchanging data across the network.

Even though that's the price you have to pay for better data consistency, there are technical solutions that can reduce network traffic, such as colocating smaller dimension or entity tables with bigger ones, so that the joining remains local to the node.

Another mitigation technique consists of using the *broadcast* mode (i.e., sending these smaller tables to all compute nodes to avoid other, usually more expensive data distribution methods).

Broadcasting Big Tables

The easiest approach to broadcasting a big table is to reduce its size by applying filters. If that's not possible, you can eventually try to configure your data processing layer to broadcast tables of larger size. In Apache Spark, you can control that part with the spark.sql.autoBroadcastJoinThreshold property.

Archival

The next challenge comes from archival needs. A dimension or entity table can be time sensitive. For example, our product table may have different prices over the years, and from your query layer, you may want to find out what the price was on a specific date.

You can easily mitigate this issue with the SCD techniques we introduced while exploring the Static Joiner pattern in Chapter 5.

Examples

First, you're going to see the NF. Figure 8-9 demonstrates how to create tables for a visit event from our use case dataset. As you can see, there are many tables you can create.

There are some important things to notice here. First, browsers and devices attributes are on dedicated tables. They can't be part of the visits_context table since the browser or device doesn't depend on the visit. They're different entities that can be shared across different visits. Second, the records on the visits_contexts table are not on the visits table because putting them there would involve repeating groups and thus break the first NF.

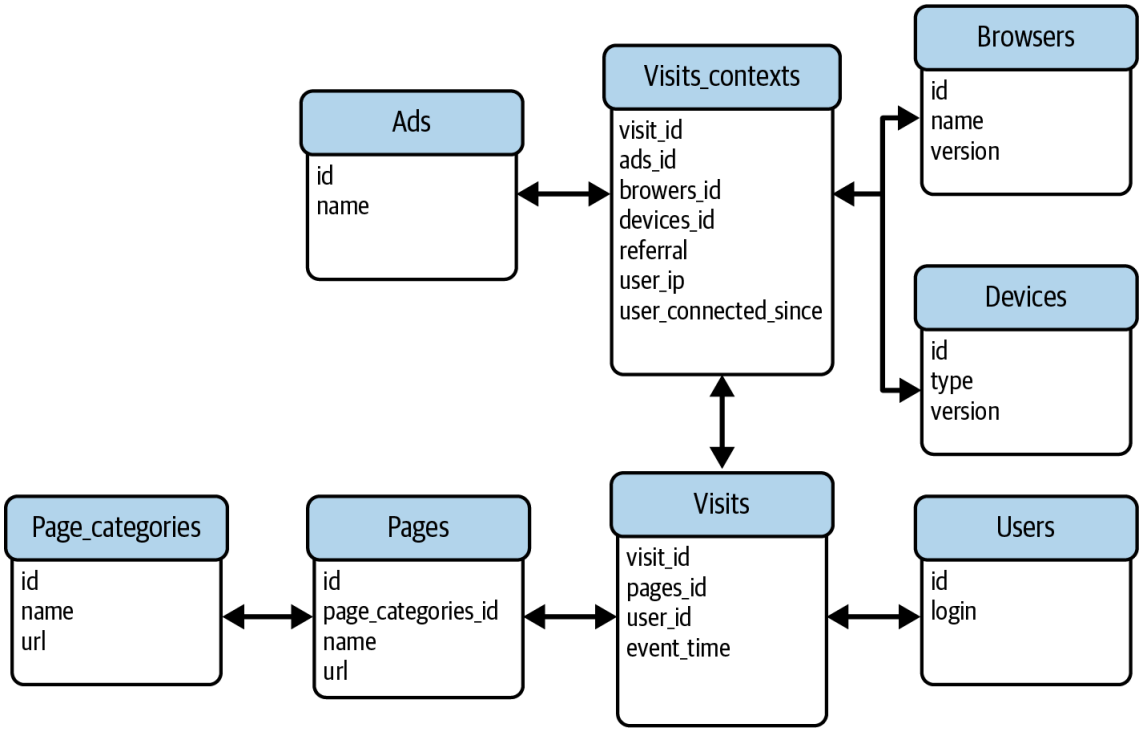


Figure 8-9. Visits in normalized form. The visit_id, pages_id, user_id, and event_time columns compose the primary key of a visit.

As you can see from the diagram, if you want to get a full picture of a visit, your query will be verbose. Example 8-23 shows this overhead for Apache Spark and Delta Lake.

Example 8-23. Joining normalized datasets

```
context = (visits_context
 .join(ads, visits_context.ads_id == ads.id, 'left_outer').drop('id')
 .join(browser, visits_context.browsers_id == browser.id, 'left_outer').drop('id')
 .join(device, visits_context.devices_id == device.id, 'left_outer').drop('id'))


page_with_category = (pages.withColumnRenamed('id', 'page_id')
  .join(categories, pages.page_categories_id == categories.id, 'left_outer')
   .drop('id').withColumnRenamed('page_id', 'id'))


full_visit = (visits
 .join(context, visits.visit_id_event == context.visit_id, 'left_outer')
 .drop('visit_id_event')
 .join(users, visits.users_id == users.id, 'left_outer').drop('id')
 .join(page_with_category, visits.pages_id == page_with_category.id, 'left_outer')
```

```
 .drop('id').withColumnRenamed('visit_id', 'id')
)
```

From that, you can deduce that the fully normalized datasets are not easily queryable and may not perform well on big datasets due to the number of joins. You'll encounter the same problem while designing a snowflake model for our visits use case. Overall, to get the full picture of a visit, the model still requires a lot of joins, like the ones in Example 8-24 for combining dates and pages.

Example 8-24. Querying overhead for a simplified snowflake schema for visits

```
page_w_category = dim_page.join(dim_page_category,
 dim_page.dim_page_category_id == dim_page_category.page_category_id,
 'left_outer')
date_with_month_and_quarter = (dim_date
 .join(dim_date_month, dim_date.dim_month_id == dim_date_month.month_id,
  'left_outer')
 .join(dim_date_quarter, dim_date.dim_quarter_id == dim_date_quarter.quarter_id,
  'left_outer'))


full_visit = (fact_visit
 .join(page_w_category, fact_visit.dim_page_id == page_w_category.page_id,
  'left_outer')
 .join(date_with_month_and_quarter
  fact_visit.dim_date_id == date_with_month_and_quarter.date_id,
  'left_outer')
)
```

Pattern: Denormalizer

Knowing that joins can be costly, a simple optimization technique is to reduce or avoid them. Unfortunately, that causes side effects that you'll learn more about in a few minutes, after discovering the next pattern.

Problem

You were called to help a company that implemented a relational model on top of their data warehouse storage for analytics. They didn't notice any issues in the first few months, as the data volume was low. But then their product became incredibly successful, and their data analytics department started to complain about the query execution time.

You performed your preliminary analysis and learned that the most expensive solution involves joining all eight of the tables involved in 80% of queries. Thanks to your previous experience, you have an idea how to create a better solution for the issue.

Solution

The stated problem is a typical scenario where the Denormalizer pattern can help. Unlike the Normalizer, it tends to reduce and even eliminate all joins from the query.

The elimination approach consists of flattening values from all joined tables into a single row so that there is no need to exchange data across the network. How? There are two different ways to do it:

Just as regular columns

Here, each column from the joined tables is copied as is. A user can access them directly as top-level columns from a SELECT statement.

As nested structures

In this approach, all rows from the joined tables can be put into one column in the target table. Typically, you will rely on the STRUCT type that's available in modern data stores to represent complex types. As a result, the user will need to access the attributes of this column instead of accessing the column directly.

An example of the first implementation could be storing the visits from our use case alongside referential datasets, such as users and devices, in the same table. This design approach is also known as One Big Table, and you can see it in Table 8-4.

visit_id	user_id	user_name	device_id	device_full_name	visit_time	visited_page
1	409	user ABC	10000	local computer	2024-07-01T09:00:00Z	home.html

Table 8-4. Denormalized visits table

In the second approach, the Denormalizer pattern reduces the number of joins by flattening related tables. This is typically the usage in dimensional models with the *star schema*. Although the star schema also uses the fact and dimension tables, unlike the snowflake schema, it doesn't accept nested dimensions. Put differently, dimensions describing other dimensions are now present in the highest-level dimension table (Figure 8-10).

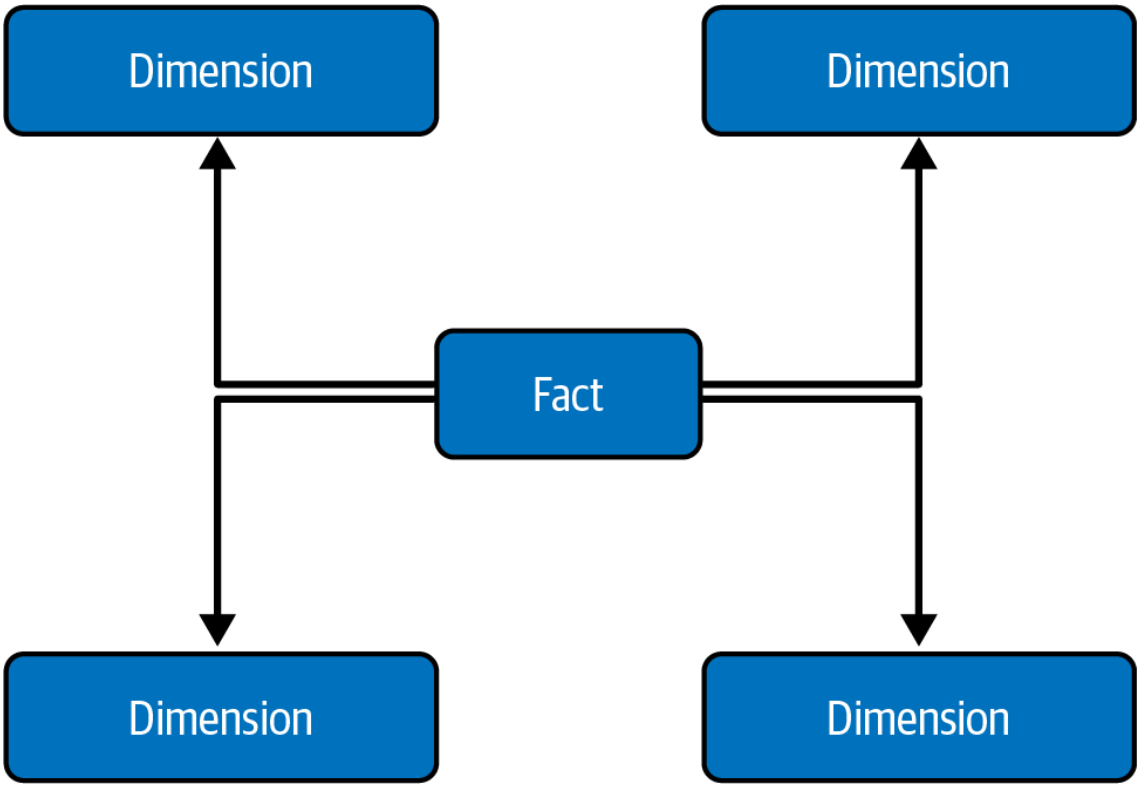


Figure 8-10. Star model with only one dimension table level

With the Denormalizer pattern, the query cost is significantly lower, thanks to the reduced network traffic needs.

The Normalizer and Denormalizer patterns are not exclusive, though. If you still care about consistency, you can apply one of the Normalizer's models first and create the denormalized version on top of it for querying. To keep them in sync, you can leverage one of the sequence design patterns covered in Chapter 6.

Figure 8-11 shows an example of a workflow in which we first create a normalized snowflake schema and later use it to build a corresponding One Big Table optimized for reading. Both datasets are accessible to users, but you can also decide to hide the snowflake schema and treat it like a private reference model for the tables you expose.

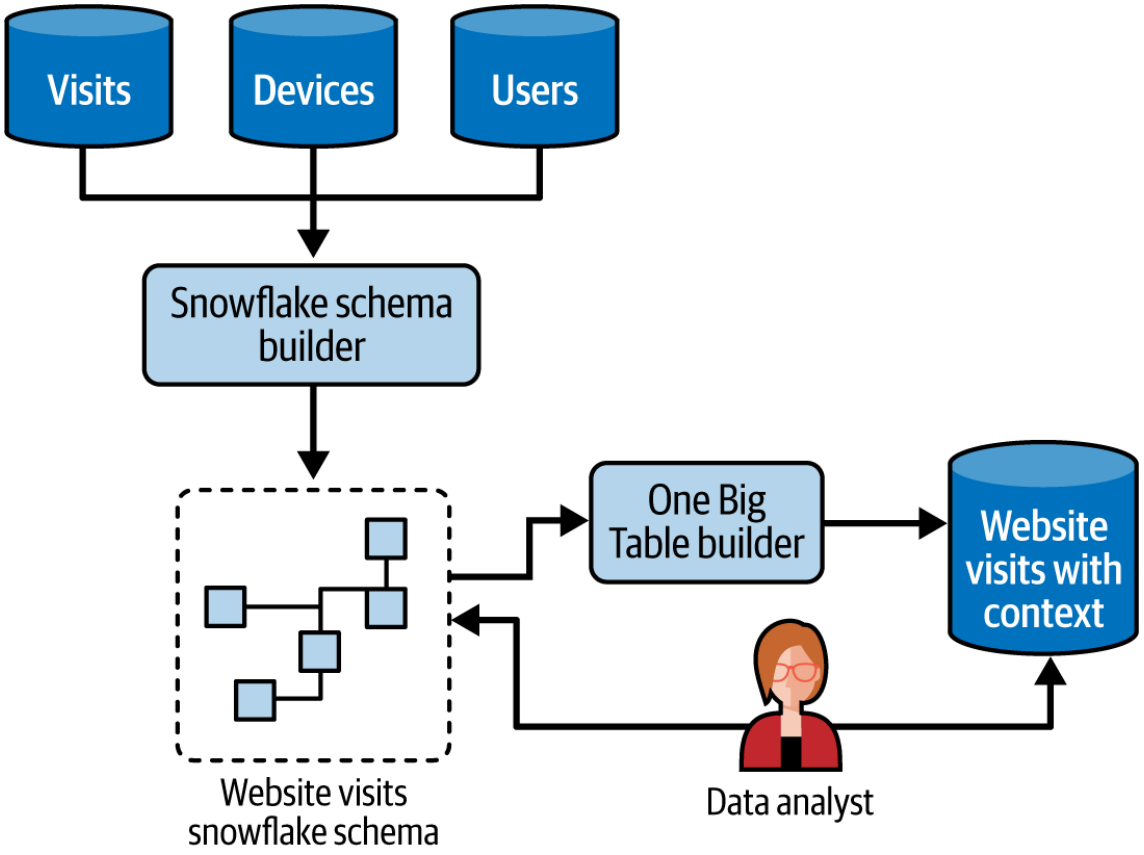


Figure 8-11. Combining the Normalizer (snowflake schema builder) and Denormalizer (One Big Table builder) design patterns, both available to the data analyst

Consequences

Even though the Denormalizer optimizes data access, it sacrifices data consistency.

Costly updates

Since all attributes are now duplicates, updating one will potentially require changing multiple rows instead of one in cases of normalized storage. This is technically feasible but will be more expensive than the normalized approach.

There is no magic solution to mitigate the issue. The only viable mitigation strategy relies on what you consider the denormalized table to be. If you consider it to be a snapshot (i.e., what your data looked like at a specific point in time), you will not need any updates. Otherwise, you may simply need to accept the fact that you need to perform a more expensive update operation to have quicker response times.

Storage

Storage is another concern. You will probably repeat the same information from the joined tables multiple times, which may end up taking up some space in your database. Fortunately, there are various encoding techniques that can reduce the storage footprint.

A popular and easy space-optimizing encoding strategy involves using a dictionary. The dictionary builds a mapping between the real values and their more compact representation, and it uses the compact values in the columns. An example of such a mapping would be transforming long string columns into integers, such as {1: "long name...",

2: "long name, next...", ...}. In addition to saving space, these techniques can improve performance. For example, the query engine may decide to check for the existence of a value by verifying the dictionary instead of reading the dataset.

One big antipattern

The One Big Table solution, despite its good intentions of flattening records and reducing query time, can end up as an antipattern if it doesn't follow any domain-oriented logic. Let's take a look at an example of the One Big Table group's attributes, such as a user's details, a list of their past orders, columns for their current visit to our website, and finally, their favorite color.

If the user's favorite color and past orders have nothing to do with the visit, One Big Table ends up as a trash bag that you put things into, but from outside, you don't really know what's inside.

How can you know when to stop while choosing attributes to combine? Your intuition about the domain knowledge should help here. If you don't know it that well, a good exercise is to try to give a name to the table. If you end up using a lot of conjunctions such as *and* or *with*, it may be a sign that you've put too many unrelated attributes together.

Examples

The first example of the Denormalizer pattern is One Big Table that combines all related elements and stores them in the same table. Creating this table can be costly, but the good news is that you pay the cost only once per update operation. All subsequent readers will take advantage of it and experience much faster operations. Example 8-25 shows the costly writing and cheap reading of a table combining visits with pages and categories.

Example 8-25. Writing and reading One Big Table

```
# writing

page_w_category = dim_page.join(dim_page_category,

 dim_page.dim_page_category_id == dim_page_category.page_category_id,

  'left_outer')

date_w_month_quarter = (dim_date

 .join(dim_date_month, dim_date.dim_month_id == dim_date_month.month_id,

  'left_outer')

 .join(dim_date_quarter, dim_date.dim_quarter_id == dim_date_quarter.quarter_id,

  'left_outer'))


full_visit = (fact_visit

 .join(page_w_category, fact_visit.dim_page_id == page_w_category.page_id,

 'left_outer')
```

```
 .join(date_w_month_quarter, fact_visit.dim_date_id == date_w_month_quarter.date_id,
  'left_outer')
)

full_visit.write.mode('overwrite').format('delta').save(get_one_big_table_dir())

# reading
visits_table = spark_session.read.format('delta').load(get_one_big_table_dir())
```

When it comes to a slightly normalized denormalization storage, the star schema, the writing step creates more tables, which also has an impact on the reading step that requires joins. That was not the case previously as all combined data was flattened. Example 8-26 shows this impact.

Example 8-26. Writing and reading for a star schema

```
# writing
page_with_category = dim_page.join(dim_page_category,
  dim_page.dim_page_category_id == dim_page_category.page_category_id,
  'left_outer').dropDuplicates()
page_with_category.write.mode('overwrite').format('delta').save(output_page)

date_with_month_and_quarter = (dim_date
 .join(dim_date_month, dim_date.dim_month_id == dim_date_month.month_id,
 'left_outer')
 .join(dim_date_quarter, dim_date.dim_quarter_id == dim_date_quarter.quarter_id,
 'left_outer')).dropDuplicates()
(date_with_month_and_quarter.write.mode('overwrite').format('delta')
  .save(output_date_dir))

visits_dataset = (spark_session.read
  .schema('visit_id STRING, event_time TIMESTAMP,  page STRING')
  .format('json').load(input_visits_dir))
fact_visit = (visits_dataset.selectExpr(
 'visit_id', 'HASH(page) AS dim_page_id',
```

```
  'HASH(TO_DATE(event_time)) AS dim_date_id',
  'DATE_FORMAT(event_time, "HH:mm:ss") AS event_time'
))
fact_visit.write.mode('overwrite').format('delta').save(output_visits_dir)

# reading
fact_visit = spark_session.read.format('delta').load(output_visits_dir)
dim_date = spark_session.read.format('delta').load(output_date_dir)
dim_page = spark_session.read.format('delta').load(output_page_dir)

full_visit = (fact_visit
  .join(dim_date, fact_visit.dim_date_id == dim_date.date_id, 'left_outer')
  .join(dim_page, [fact_visit.dim_page_id == dim_page.page_id], 'left_outer'))
```

Summary

In this chapter, you learned about data storage design patterns. The first section was dedicated to partitioning strategies. You saw two approaches, horizontal and vertical. The horizontal approach operates on whole rows and is a good candidate for low-cardinality values, such as event time values. Vertical partitioning works at the attributes level, so it splits one row into multiple parts stored in different places.

Although partitioning is a great data storage optimization strategy, it won't work well for high-cardinality values, such as last names or cities. Here, a better approach will be the Bucket pattern that groups multiple similar rows into containers called buckets. Additionally, you can leverage a Sorter to enable faster processing on top of sorted data.

The third section covered other access optimization strategies. The first of them is Metadata Enhancer, which tries to reduce the volume of data to process by filtering out irrelevant files or rows from the metadata layer. Next, you saw the Dataset Materializer pattern, which is ideal for materializing complex queries and thus optimizing the reading path by sacrificing storage. Finally, you saw the Manifest pattern, which you can use to mitigate often costly listing operations.

In the last section, you saw two data representation patterns. The first is the Normalizer pattern, which favors data consistency but involves joins. The alternative is the Denormalizer pattern, which introduces a risk of data inconsistency but completely eliminates the need for joining multiple datasets.

And now, unfortunately, I have to disappoint you yet again. Even the best-optimized storage won't be enough to guarantee that other people will use your data. You also need to provide data of the best possible quality, and that's what the next chapter will be about.

1 You can find more information about it in two O'Reilly books: *Delta Lake: The Definitive Guide* by Denny Lee, Tristen Wentling, Scott Haines, and Prashanth Babu (2024) and *Apache Iceberg: The Definitive Guide* by Tomer Shiran, Jason Hughes, and Alex Merced (2024).

2 Ralph Kimball and Margy Ross, *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*, 3rd ed. (Wiley, 2013).