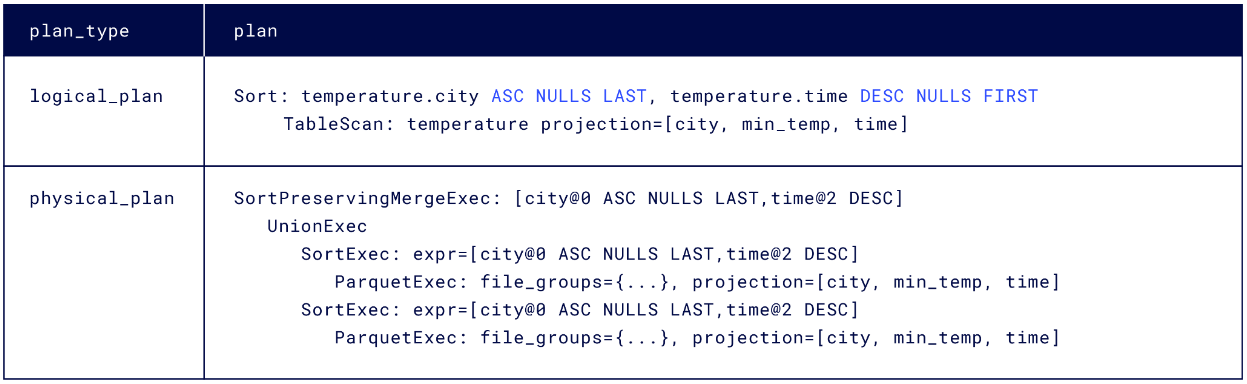
How to read InfluxDB 3.0 Query Plans

This blog post explains how to read a query plan in InfluxDB 3.0 and requires basic knowledge of InfluxDB 3.0 System Architecture.

InfluxDB 3.0 supports two query languages: SQL and InfluxQL. The database executes a query written in either SQL or InfluxQL according to the instructions of a **query plan**. To see the plan without running the query, add the keyword `EXPLAIN` in front of your query as follows:

```
<pre><code class="language-sql">EXPLAIN
SELECT   city, min_temp, time
FROM     temperature
ORDER BY city ASC, time DESC;
</code></pre>
```

The output will look like this:



plan_type	plan
logical_plan	Sort: temperature.city ASC NULLS LAST, temperature.time DESC NULLS FIRST TableScan: temperature projection=[city, min_temp, time]
physical_plan	SortPreservingMergeExec: [city@0 ASC NULLS LAST,time@2 DESC] UnionExec SortExec: expr=[city@0 ASC NULLS LAST,time@2 DESC] ParquetExec: file_groups={...}, projection=[city, min_temp, time] SortExec: expr=[city@0 ASC NULLS LAST,time@2 DESC] ParquetExec: file_groups={...}, projection=[city, min_temp, time]

```
<br>
```

Figure 1: A simplified output of a query plan

There are two types of plans: the logical plan and the physical plan.

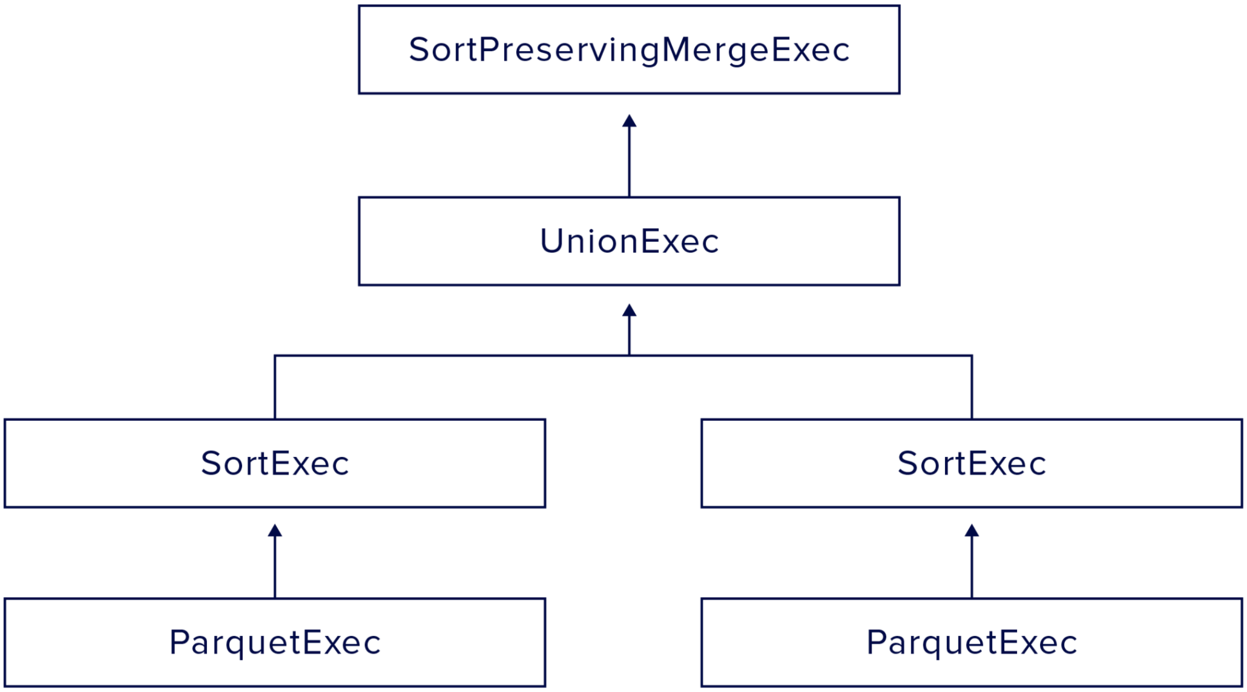
Logical Plan: This is a plan generated for a specific SQL or InfluxQL query without knowledge of the underlying data organization or the cluster configuration. Because InfluxDB 3.0 is built on top of DataFusion, a logical plan is very similar to what you would see with any data format or storage in DataFusion.

Physical Plan: This is a plan generated from a query's corresponding logical plan plus the cluster configuration (e.g., number of CPUs) and underlying data organization (e.g., number of files, the layout of data in the files, etc.) information. The physical plan is specific to your data and InfluxDB cluster configuration. If you load the same data to different clusters with different configurations, the same query may generate different physical query plans. Similarly, running the same query on the same cluster at different times can have a different plan depending on your data at that time.

Understanding a query plan can help explain why the query is slow. For example, if the plan shows that your query reads many files, you can add more filters to reduce the amount of data it needs to read or modify your cluster configuration/design to create fewer but larger files. This document focuses on how to read a query plan. Techniques for making a query run faster depend on the reason(s) it is slow and are beyond the scope of this blog post.

A query plan is a tree

A query plan is an upside-down tree and should be read from the bottom up. In tree format, we can represent the physical plan of Figure 1 in the following way:



<br>

Figure 2: The tree structure of physical plan in Figure 1

The name of each node in the tree ends with `Exec` to indicate an `ExecutionPlan` that processes, transforms, and sends data to the next level of the tree. First, two `ParquetExec` nodes read Parquet files in parallel, and each node outputs a stream of data to its corresponding `SortExec` node. The `SortExc` nodes are responsible for sorting the data in `city` ascending and `time` descending. The UnionExec node combines the sorted outputs from the two `SortExec` nodes, which are then (sort) merged by the `SortPreservingMergeExec` node to return the sorted data.

How to understand a large query plan

A large query plan may look intimidating, but if you follow these steps, you can quickly understand what the plan does.

1. As always, read from the bottom up, one `Exec` node at a time.
2. Understand the job of each `Exec` node. Most of this information is available in the DataFusion Physical Plan documentation or directly from its repo. The `ExecutionPlans` that are not in the DataFusion docs are InfluxDB specific—more information is available in this InfluxDB repo.
3. Recall what the input data of the `Exec` node looks like and how large/small it may be.
4. Consider how much data that `Exec` node may send out and what it would look like.

Using these steps, you can estimate how much work a plan has to do. However, the `explain` command shows you the plan without executing it. If you want to know exactly how long it takes a plan and each of its ExecutionPlan to execute, you need other tools.

Tools that show the exact runtime for each ExecutionPlan

1. Run `EXPLAIN ANALYZE`, to print out an ‘explain plan’ (see Figure 1) annotated with execution counters and information such as runtime and rows produced.
2. There are other tools, such as distributed tracing with Jaeger, which we will describe in a future post.

More information for debugging

If the plan has to read many files, the `` `EXPLAIN` `` report will not show all of them. To see all files, use `` `EXPLAIN VERBOSE` ``. Like `EXPLAIN`, `EXPLAIN VERBOSE` does not run the query and won't tell you the runtime. Instead, you get all information omitted from the `` `EXPLAIN` `` report and all intermediate physical plans that the InfluxDB 3.0 querier and DataFusion generate before returning the final physical plan. This is very helpful for debugging because you can see when the plan adds or removes an ExecutionPlan and what InfluxDB and DataFusion are doing to optimize your query.

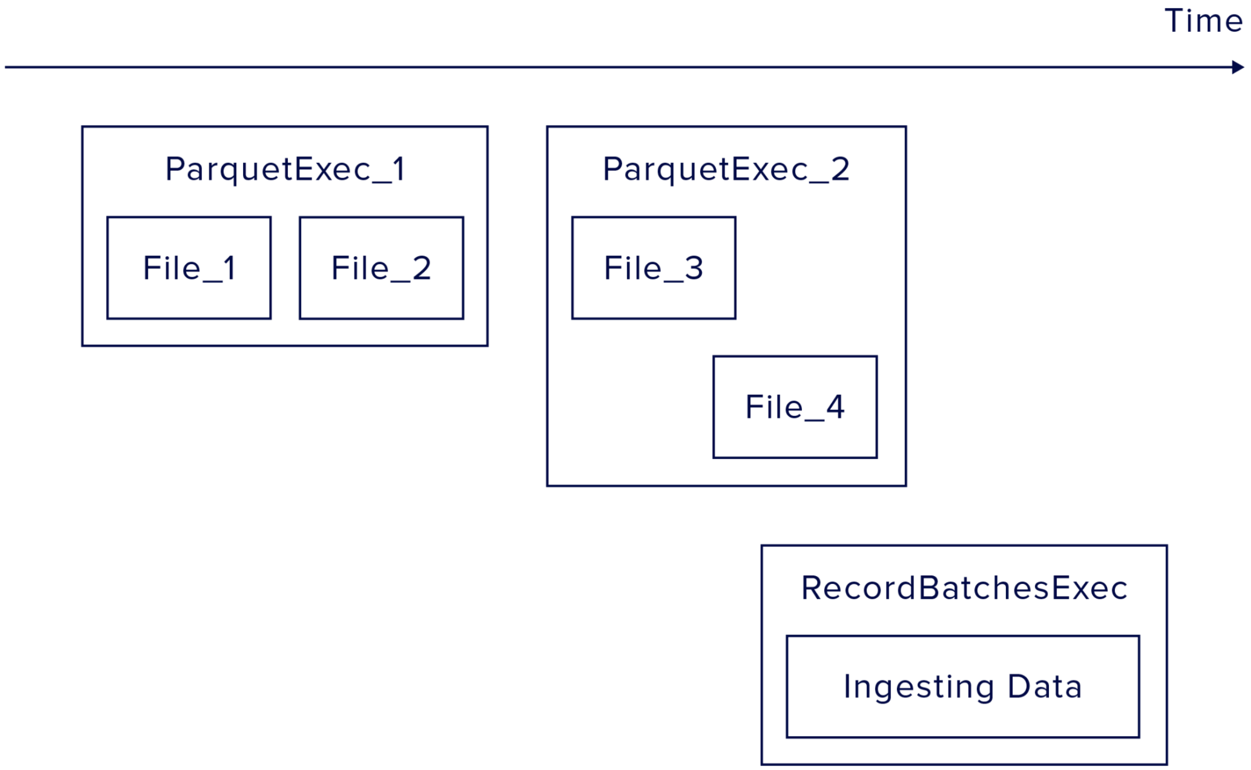
Example of a typical plan for leading-edge data

Let's delve into an example that covers typical ExecutionPlans as well as InfluxDB-specific ones on leading-edge data.

Data organization

To make it easier to explain the plan below, Figure 3 shows the data organization that the plan reads. Once you get used to reading query plans, you can figure this out from the plan itself. Some details to note:

- There may be more data in the system. This is just the data the query reads after applying the predicate of the query to prune out-of-bounds partitions.
- Recently received data is being ingested and isn't yet persisted. In the plan, the `` `RecordBatchesExec` `` represents data from the ingester not yet persisted to Parquet files.
- Four Parquet files are retrieved from storage and are represented by two `` `ParquetExec` nodes `` containing two files each:
  - In the first node, two files, `` `file_1` `` and `` `file_2` ``, do not overlap in **time** with any other files and do not have any duplicated data. Data within a file never has duplicates, so deduplication is never necessary for non-overlapped files.
  - In the second node, two files, `` `file_3` `` and `` `file_4` ``, overlap with each other and with the ingesting data represented by the `` `RecordBatchesExec` ``.

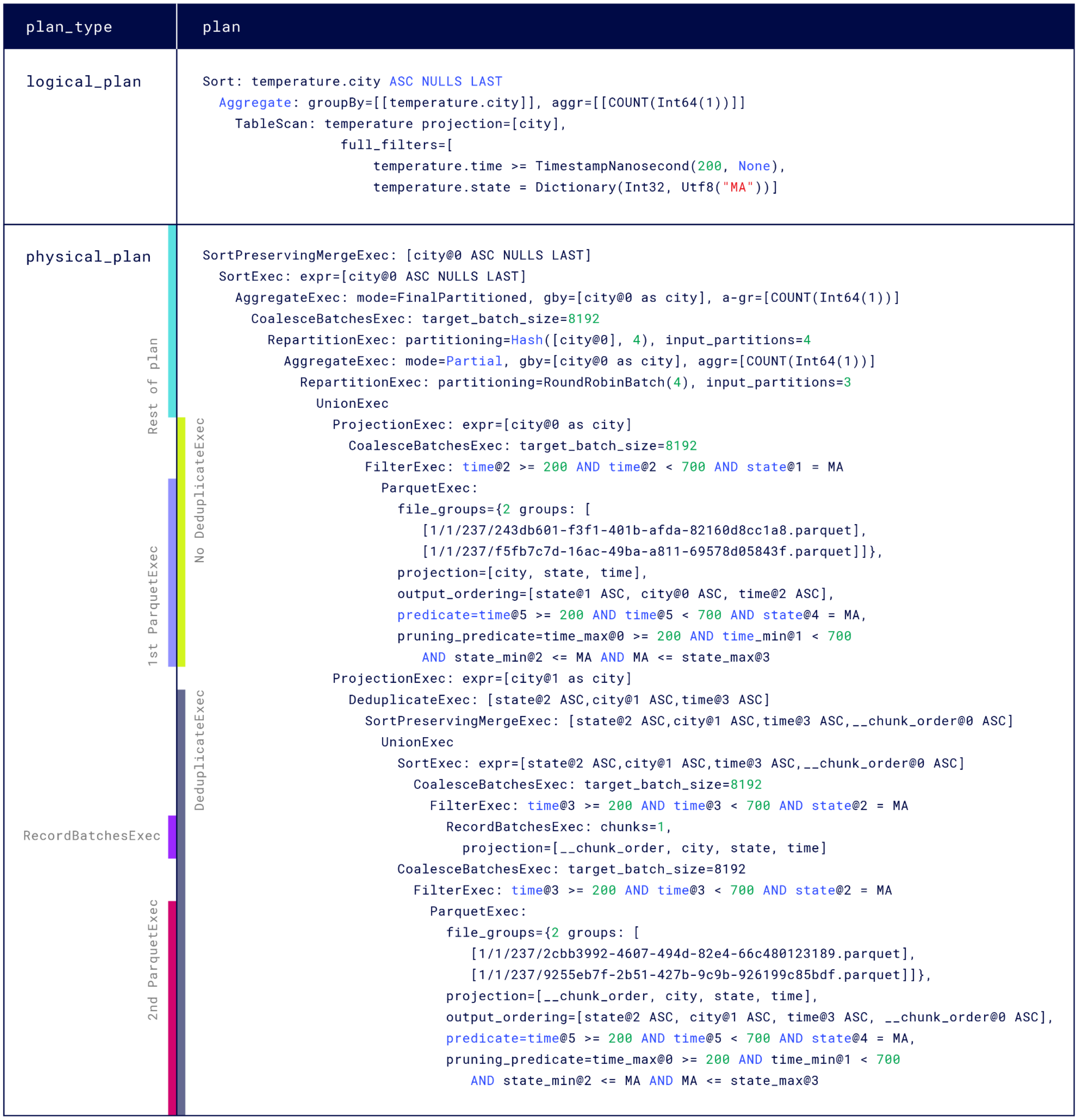


<br>

Figure 3: Data of the query plan in Figure 4

Query and query plan

```
<pre><code class="language-sql">EXPLAIN
SELECT city, count(1)
FROM   temperature
WHERE  time >= to_timestamp(200) AND time < to_timestamp(700)
       AND state = 'MA'
GROUP BY city
ORDER BY city ASC;
</code></pre>
```



plan_type | plan

logical_plan

```
Sort: temperature.city ASC NULLS LAST
  Aggregate: groupBy=[[temperature.city]], aggr=[[COUNT(Int64(1))]]
    TableScan: temperature projection=[city],
                 full_filters=[
                     temperature.time >= TimestampNanosecond(200, None),
                     temperature.state = Dictionary(Int32, Utf8("MA"))]
```

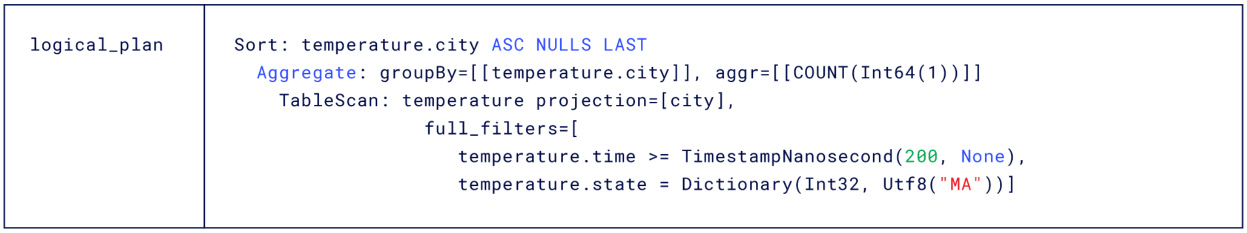
physical_plan

```
SortPreservingMergeExec: [city@0 ASC NULLS LAST]
  SortExec: expr=[city@0 ASC NULLS LAST]
    AggregateExec: mode=FinalPartitioned, gby=[city@0 as city], a-gr=[COUNT(Int64(1))]
      CoalesceBatchesExec: target_batch_size=8192
        RepartitionExec: partitioning=Hash([city@0], 4), input_partitions=4
          AggregateExec: mode=Partial, gby=[city@0 as city], aggr=[COUNT(Int64(1))]
            RepartitionExec: partitioning=RoundRobinBatch(4), input_partitions=3
              UnionExec
                ProjectionExec: expr=[city@0 as city]
                  CoalesceBatchesExec: target_batch_size=8192
                    FilterExec: time@2 >= 200 AND time@2 < 700 AND state@1 = MA
                      ParquetExec:
                        file_groups={2 groups: [
                           [1/1/237/243db601-f3f1-401b-afda-82160d8cc1a8.parquet],
                           [1/1/237/f5fb7c7d-16ac-49ba-a811-69578d05843f.parquet]]},
                        projection=[city, state, time],
                        output_ordering=[state@1 ASC, city@0 ASC, time@2 ASC],
                        predicate=time@5 >= 200 AND time@5 < 700 AND state@4 = MA,
                        pruning_predicate=time_max@0 >= 200 AND time_min@1 < 700
                           AND state_min@2 <= MA AND MA <= state_max@3
                ProjectionExec: expr=[city@1 as city]
                  DeduplicateExec: [state@2 ASC,city@1 ASC,time@3 ASC]
                    SortPreservingMergeExec: [state@2 ASC,city@1 ASC,time@3 ASC,__chunk_order@0 ASC]
                      UnionExec
                        SortExec: expr=[state@2 ASC,city@1 ASC,time@3 ASC,__chunk_order@0 ASC]
                          CoalesceBatchesExec: target_batch_size=8192
                            FilterExec: time@3 >= 200 AND time@3 < 700 AND state@2 = MA
                              RecordBatchesExec: chunks=1,
                                projection=[__chunk_order, city, state, time]
                        CoalesceBatchesExec: target_batch_size=8192
                          FilterExec: time@3 >= 200 AND time@3 < 700 AND state@2 = MA
                            ParquetExec:
                              file_groups={2 groups: [
                                 [1/1/237/2cbb3992-4607-494d-82e4-66c480123189.parquet],
                                 [1/1/237/9255eb7f-2b51-427b-9c9b-926199c85bdf.parquet]]},
                              projection=[__chunk_order, city, state, time],
                              output_ordering=[state@2 ASC, city@1 ASC, time@3 ASC, __chunk_order@0 ASC],
                              predicate=time@5 >= 200 AND time@5 < 700 AND state@4 = MA,
                              pruning_predicate=time_max@0 >= 200 AND time_min@1 < 700
                                 AND state_min@2 <= MA AND MA <= state_max@3
```

Figure 4: A typical query plan of leading-edge (most recent) data. Note: The colors in the left column correspond to the figures below.

Reading logical plan

The logical plan in Figure 5 shows that the table scan occurs first and that the query predicates then filters the data. Next, the plan aggregates the data to compute the count of the number of rows per city. Finally, the plan sorts and returns the data.



logical_plan

```
Sort: temperature.city ASC NULLS LAST
  Aggregate: groupBy=[[temperature.city]], aggr=[[COUNT(Int64(1))]]
    TableScan: temperature projection=[city],
                 full_filters=[
                    temperature.time >= TimestampNanosecond(200, None),
                    temperature.state = Dictionary(Int32, Utf8("MA"))]
```

<br>

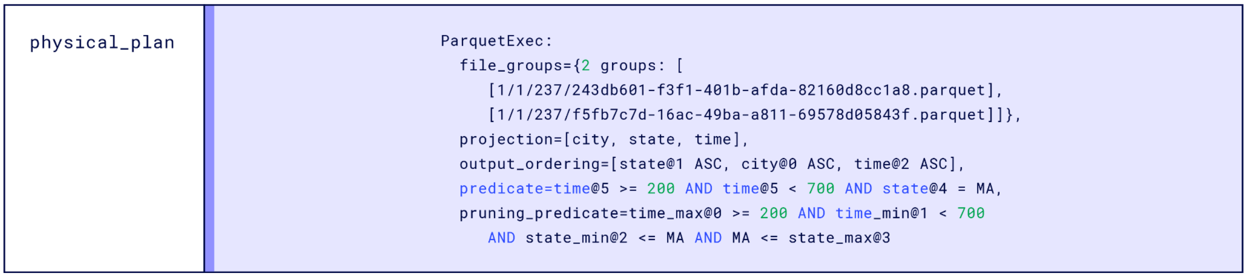
Figure 5: Logical plan from Figure 4

Reading physical plan

Let us begin reading from the bottom up. The bottom or leaf nodes are always either `ParquetExec` or `RecordBatchExec`. There are three of them in this plan, so let's go over them one by one.

The three bottom leaves consist of two `ParquetExec` nodes and one `RecordBatchesExec` node.

First `ParqetExec`



physical_plan

```
ParquetExec:
  file_groups={2 groups: [
     [1/1/237/243db601-f3f1-401b-afda-82160d8cc1a8.parquet],
     [1/1/237/f5fb7c7d-16ac-49ba-a811-69578d05843f.parquet]]},
  projection=[city, state, time],
  output_ordering=[state@1 ASC, city@0 ASC, time@2 ASC],
  predicate=time@5 >= 200 AND time@5 < 700 AND state@4 = MA,
  pruning_predicate=time_max@0 >= 200 AND time_min@1 < 700
     AND state_min@2 <= MA AND MA <= state_max@3
```

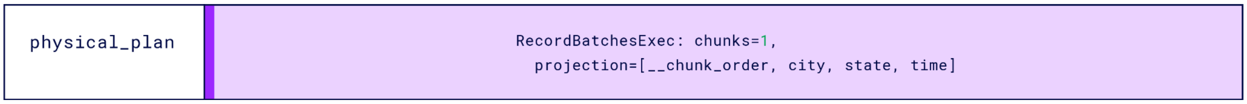
<br>

Figure 6: First ParquetExec

- This `ParquetExec` includes two groups of files. Each group can contain one or many files, but in this example, there is one file in each group. The node executes the groups in parallel and reads the files in each group sequentially. So, in this example, the two files are read in parallel.
- `1/1/237/2cbb3992-4607-494d-82e4-66c480123189.parquet`: this is the path of the file in object storage. It is in the structure `db_id/table_id/partition_hash_id/uuid_of_the_file.parquet`, and each segment, respectively, tells us:
  - Which database and table are queried

  - Which partition the file belongs to (you can count how many partitions this query reads)
  - Which file it is
- `projection=[__chunk_order, city, state, time]`: there are many columns in this table, but the node only reads these four. The `__chunk_order` column is an artificial column the InfluxDB code generates to keep the chunks/files ordered for deduplication.
- `output_ordering=[state@2 ASC, city@1 ASC, time@3 ASC, __chunk_order@0 ASC]`: this `ParquetExec` node will sort its output on `state ASC, city ASC, time ASC, __chunk_order ASC`. InfluxDB automatically sorts Parquet files when storing them to improve storage compression and query efficiency.
- `predicate=time@5 >= 200 AND time@5 < 700 AND state@4 = MA`: This is a filter in the query used for data pruning.
- `pruning_predicate=time_max@0 >= 200 AND time_min@1 < 700 AND state_min@2 <= MA AND MA <= state_max@3`: this is the actual pruning predicate transformed from the predicate above. It is used to filter files outside that predicate. At this time (Dec 2023), InfluxDB 3.0 only filters files based on `time`. Note that this predicate is for pruning **files from the chosen partitions**.

`RecordBatchesExec`

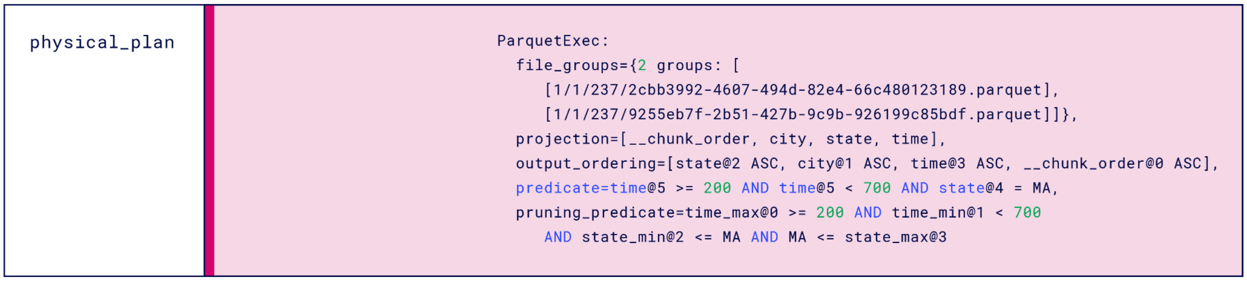


physical_plan	RecordBatchesExec: chunks=1, projection=[__chunk_order, city, state, time]

Figure 7: RecordBatchesExec

Data from the ingester can be in many chunks, but often, as in this example, there is only one. This node only sends data from four columns to the output, like the `ParquetExec` node. We call the action of **filtering columns** a **projection pushdown**. It thus has the name `projection` in the query plan.

Second `ParquetExec`



```
physical_plan    ParquetExec:
                   file_groups={2 groups: [
                      [1/1/237/2cbb3992-4607-494d-82e4-66c480123189.parquet],
                      [1/1/237/9255eb7f-2b51-427b-9c9b-926199c85bdf.parquet]]},
                   projection=[__chunk_order, city, state, time],
                   output_ordering=[state@2 ASC, city@1 ASC, time@3 ASC, __chunk_order@0 ASC],
                   predicate=time@5 >= 200 AND time@5 < 700 AND state@4 = MA,
                   pruning_predicate=time_max@0 >= 200 AND time_min@1 < 700
                      AND state_min@2 <= MA AND MA <= state_max@3
```

<br>

Figure 8: Second ParquetExec

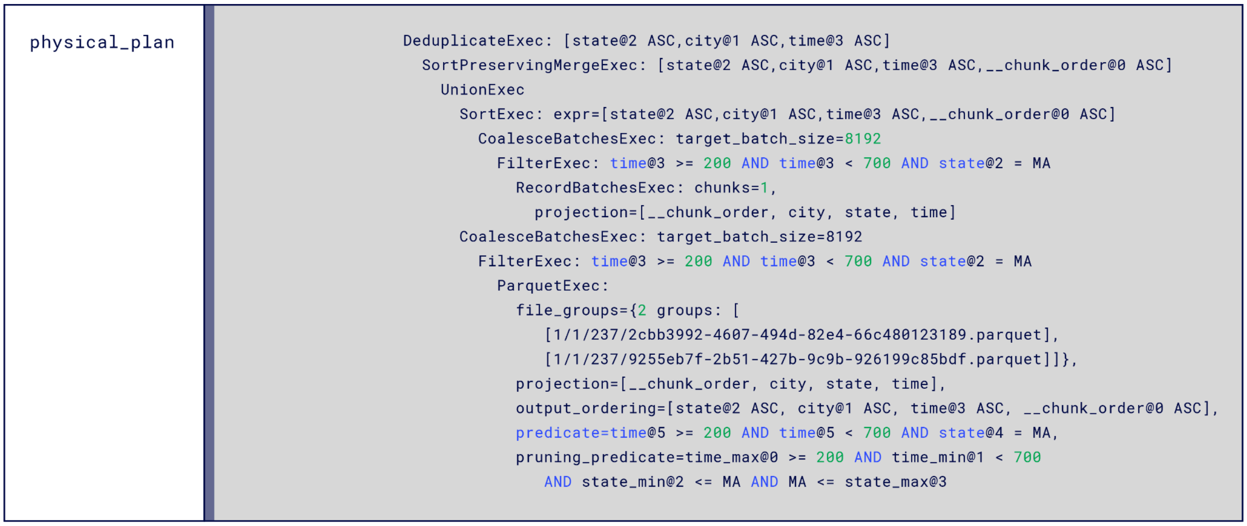
Reading the second `ParquetExec` node is similar to the one above. Note that the files in both `ParquetExec` nodes belong to the same partition (*237*).

Data-scanning structures

Why do we send Parquet files from the same partition to different `ParquetExec`? There are many reasons, but two major ones are:

1. To minimize the work required for deduplication by splitting the non-overlaps from the overlaps (which is the case in this example).
2. To improve parallelism by splitting the non-overlaps.

How do we know that data overlaps?



```
physical_plan    DeduplicateExec: [state@2 ASC,city@1 ASC,time@3 ASC]
                   SortPreservingMergeExec: [state@2 ASC,city@1 ASC,time@3 ASC,__chunk_order@0 ASC]
                     UnionExec
                       SortExec: expr=[state@2 ASC,city@1 ASC,time@3 ASC,__chunk_order@0 ASC]
                         CoalesceBatchesExec: target_batch_size=8192
                           FilterExec: time@3 >= 200 AND time@3 < 700 AND state@2 = MA
                             RecordBatchesExec: chunks=1,
                               projection=[__chunk_order, city, state, time]
                       CoalesceBatchesExec: target_batch_size=8192
                         FilterExec: time@3 >= 200 AND time@3 < 700 AND state@2 = MA
                           ParquetExec:
                             file_groups={2 groups: [
                                [1/1/237/2cbb3992-4607-494d-82e4-66c480123189.parquet],
                                [1/1/237/9255eb7f-2b51-427b-9c9b-926199c85bdf.parquet]]},
                             projection=[__chunk_order, city, state, time],
                             output_ordering=[state@2 ASC, city@1 ASC, time@3 ASC, __chunk_order@0 ASC],
                             predicate=time@5 >= 200 AND time@5 < 700 AND state@4 = MA,
                             pruning_predicate=time_max@0 >= 200 AND time_min@1 < 700
                                AND state_min@2 <= MA AND MA <= state_max@3
```

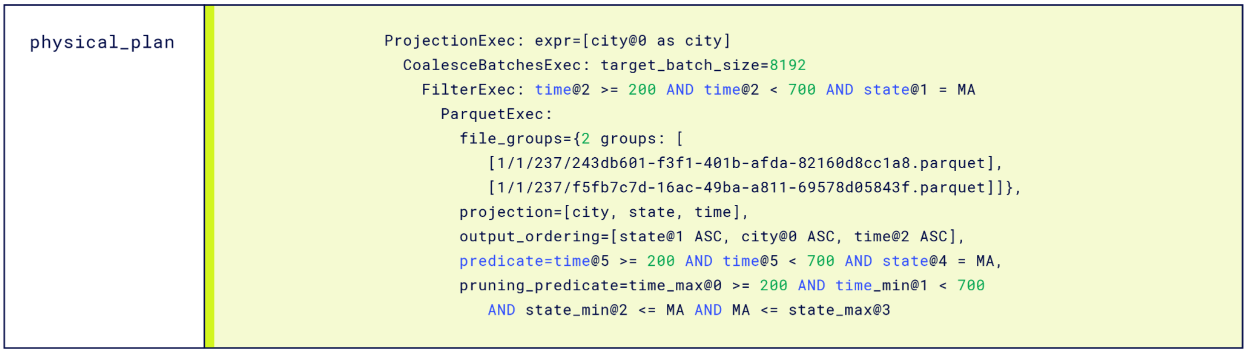
<br>

Figure 9: DeduplicationExec is a signal of overlapped data

`DeduplicationExec` in Figure 9 tells us that the preceding data (i.e., the data below it) overlaps. More specifically, data in two files overlaps and/or overlaps the data from the ingesters.

- `FilterExec: time@3 >= 200 AND time@3 < 700 AND state@2 = MA`: this is where we filter out everything that meets the conditions `time@3 >= 200 AND time@3 < 700 AND state@2 = MA`. The previous operation only prunes data when possible. It does not guarantee the pruning of all data. We need this filter to perform complete and precise filtering.
- `CoalesceBatchesExec: target_batch_size=8192` is a way to group small data into larger groups if possible. Refer to the DataFusion documentation for how it works.
- `SortExec: expr=[state@2 ASC,city@1 ASC,time@3 ASC,__chunk_order@0 ASC]`: this sorts data on `state ASC, city ASC, time ASC, __chunk_order ASC`. Note that this sort only applies to data from ingesters because data from Parquet files is already sorted in that order.
- `UnionExec` is simply a place to pull many streams together. It is fast to execute and does not merge anything.
- `SortPreservingMergeExec: [state@2 ASC,city@1 ASC,time@3 ASC,__chunk_order@0 ASC]`: this operation merges pre-sorted. When you see this, you know the data below it is already sorted and the output is in one stream.
- `DeduplicateExec: [state@2 ASC,city@1 ASC,time@3 ASC]`: this operation deduplicates sorted data strictly from one input stream. That is why you often see `SortPreservingMergeExec` under `DeduplicateExec`, but it is not required. As long as the input to `DeduplicateExec` is a single stream of sorted data, it will work correctly.

How do we know data doesn't overlap?



physical_plan

```
ProjectionExec: expr=[city@0 as city]
  CoalesceBatchesExec: target_batch_size=8192
    FilterExec: time@2 >= 200 AND time@2 < 700 AND state@1 = MA
      ParquetExec:
        file_groups={2 groups: [
           [1/1/237/243db601-f3f1-401b-afda-82160d8cc1a8.parquet],
           [1/1/237/f5fb7c7d-16ac-49ba-a811-69578d05843f.parquet]]},
        projection=[city, state, time],
        output_ordering=[state@1 ASC, city@0 ASC, time@2 ASC],
        predicate=time@5 >= 200 AND time@5 < 700 AND state@4 = MA,
        pruning_predicate=time_max@0 >= 200 AND time_min@1 < 700
           AND state_min@2 <= MA AND MA <= state_max@3
```

<br>

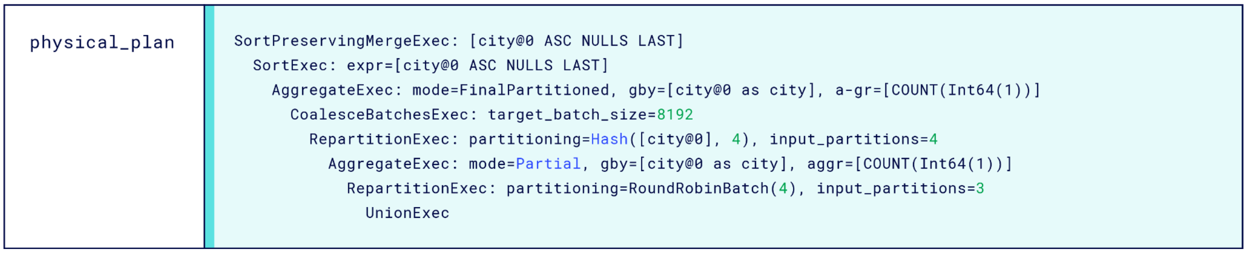
Figure 10: No `DeduplicateExec` means files do not overlap

When a `*ParquetExec*` or `*RecordBatchesExec*` branch doesn't lead to a `*DeduplicateExec*`, we know that the files handled by that `*Exec*` don't overlap.

- `*ProjectionExec: expr=[city@0 as city]*`: this filters column data and only sends out data from column `*city*`.

Other ExecutionPlans

Now let's look at the rest of the plan.



physical_plan

```
SortPreservingMergeExec: [city@0 ASC NULLS LAST]
  SortExec: expr=[city@0 ASC NULLS LAST]
    AggregateExec: mode=FinalPartitioned, gby=[city@0 as city], a-gr=[COUNT(Int64(1))]
      CoalesceBatchesExec: target_batch_size=8192
        RepartitionExec: partitioning=Hash([city@0], 4), input_partitions=4
          AggregateExec: mode=Partial, gby=[city@0 as city], aggr=[COUNT(Int64(1))]
            RepartitionExec: partitioning=RoundRobinBatch(4), input_partitions=3
              UnionExec
```

<br>

Figure 11: The rest of the plan structure

- `*UnionExec*`: unions data streams. Note that the number of output streams is the same as the number of input streams. The ExecutionPlan above is responsible for merging or splitting the streams further. This `UnionExec` is an intermediate step of the merge/split.

- `` `RepartitionExec: partitioning=RoundRobinBatch(4), input_partitions=3` ``: this splits three input streams into four output streams in a round-robin fashion. This cluster has four cores available, so this RepartitionExec partitions the data into four streams to increase parallel execution.
- `` `AggregateExec: mode=Partial, gby=[city@0 as city], aggr=[COUNT(Int64(1))]` ``: this groups data into groups that have the same values of `city`. Because there are four input streams, each stream is aggregated separately, which creates four output streams. It also means that the output data is not fully aggregated as indicated by the `` `mode=Partial` `` flag.
- `` `RepartitionExec: partitioning=Hash([city@0], 4), input_partitions=4` ``: this repartitions data on `` `hash(city)` `` into four streams so that the same city goes into the same stream.
- `` `AggregateExec: mode=FinalPartitioned, gby=[city@0 as city], aggr=[COUNT(Int64(1))]` ``: because rows for the same city are in the same stream, we only need to do the final aggregation.
- `` `SortExec: expr=[city@0 ASC NULLS LAST]` ``: sort each of the four data streams on `` `city` `` per the query request.
- `` `SortPreservingMergeExec: [city@0 ASC NULLS LAST]` ``: (sort) merge four sorted streams to return the final results.

If you see that a plan reads many files and performs deduplication on all of them, you may ask: "Do all the files overlap or not?" The answer is either yes or no, depending on the situation. Sometimes, the compactor may be behind, and if you give it some time to compact small and overlapped files, your query will read fewer files faster. If there are still a lot of files, you may want to check the workload of your compactor and add more resources as needed. There are other reasons that we deduplicate non-overlap files due to memory limitations of your querier's memory, but those are topics for a future blog post.

Conclusion

`EXPLAIN` is a way to understand how InfluxDB executes your query and why it's fast or slow. You can often rewrite your query to add more filters or remove unnecessary sorting (`` `order by` `` in the query) to make your query run faster. Other times, queries are slow because your system lacks resources. In that case, it’s time to reassess the cluster configuration or consult the InfluxDB support team.