

Sorting and GroupingSorting



Copyright

© Postgres Professional, 2019–2024

Authors Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov

Photo by: Oleg Bartunov (Phu monastery, Bhrikuti summit, Nepal)

Use of course materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

Feedback

Please send your feedback, comments and suggestions to:

edu@postgrespro.ru

Disclaimer

In no event shall Postgres Professional company be liable for any damages or loss, including loss of profits, that arise from direct or indirect, special or incidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

Retrieving Sorted Data

In-memory sorting

External sort

Incremental sort

Sorting in Parallel Execution Plans

Sorting During Index Construction

Window functions involving sorting

Index Access (B-Tree)

returns a sorted row set

Sequential Access (Seq Scan)

returns an unordered set of rows

An additional sorting step is required

As previously stated, index access (using a B-tree) enables the immediate retrieval of a row set sorted by the indexing keys.

But how do you handle it when sorting is needed on fields without an index? In this case, the server must first retrieve the data from the table using a sequential scan and then perform an additional sorting step.

Retrieving Sorted Data

Index access automatically returns rows sorted by the indexed column:

```
=> EXPLAIN (costs off)
SELECT * FROM flights
ORDER BY flight_id;
```

QUERY PLAN

```
-----
Index Scan using flights_pkey on flights
(1 row)
```

But if you ask the server to sort data by a column that lacks an index, it will require two separate steps: retrieving the data and sorting it.

```
=> EXPLAIN (costs off)
SELECT * FROM flights
ORDER BY status;
```

QUERY PLAN

```
-----
Sort
  Sort Key: status
  -> Seq Scan on flights
(3 rows)
```

Next, we'll explore how sorting works within the Sort node.

Quick sort (quicksort)

Top-N heapsort Top-N Heap Sort (Partial Heapsort)

when only a portion of the values is needed

In an ideal scenario, the set of rows to be sorted entirely fits within the memory constrained by the `work_mem` parameter. In this case, all rows are sorted using the Quick sort (quicksort) algorithm, and the result is passed up to the parent node.

If you need to retrieve just the first few rows of a sorted set (when using the `LIMIT` clause), you don't need to sort the entire set. In this case, the server can use partial heap sort (top-N heapsort), which operates in random access memory.

In-Memory Sorting

The planner has several sorting methods. In the following example, quicksort is employed (Sort Method: quicksort). The memory used is shown in the same line:

```
=> EXPLAIN (analyze, timing off, summary off)
SELECT *
FROM seats
ORDER BY seat_no;
```

QUERY PLAN

```
-----
-
Sort  (cost=90.93..94.28 rows=1339 width=15) (actual rows=1339 loops=1)
  Sort Key: seat_no
  Sort Method: quicksort  Memory: 101kB
  -> Seq Scan on seats  (cost=0.00..21.39 rows=1339 width=15) (actual rows=1339 loops=1)
(4 rows)
```

To begin outputting data, the Sort node must fully sort the row set. This means the initial cost estimate for the node includes the full cost of reading the table. The time complexity of quicksort is generally $O(M \log M)$, where M represents the number of rows in the input set.

If the number of rows is limited, the planner can switch to partial sorting—essentially, instead of fully sorting all rows, it retrieves the top 100 minimum values here.

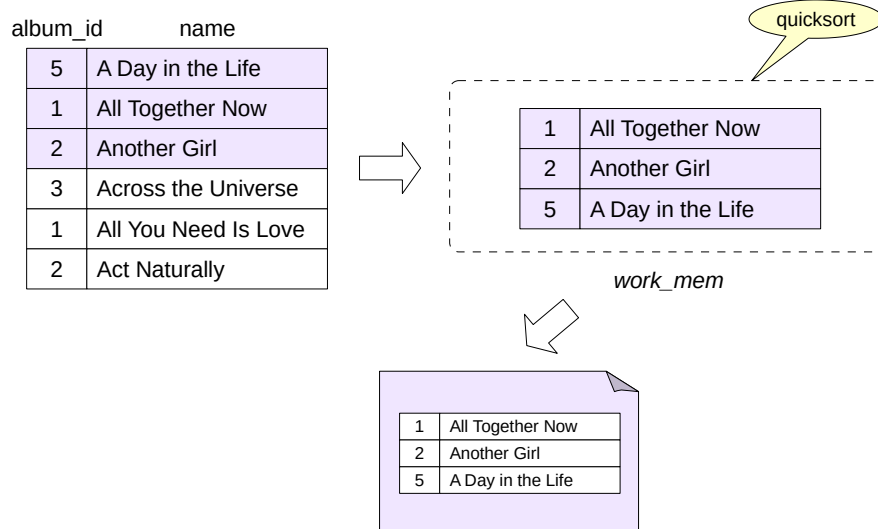
```
=> EXPLAIN (analyze, timing off, summary off)
SELECT *
FROM seats
ORDER BY seat_no
LIMIT 100;
```

QUERY PLAN

```
-----
-----
Limit  (cost=72.57..72.82 rows=100 width=15) (actual rows=100 loops=1)
  -> Sort  (cost=72.57..75.91 rows=1339 width=15) (actual rows=100 loops=1)
        Sort Key: seat_no
        Sort Method: top-N heapsort  Memory: 32kB
        -> Seq Scan on seats  (cost=0.00..21.39 rows=1339 width=15) (actual rows=1339
loops=1)
(5 rows)
```

Notice that the query's cost decreased and sorting required less memory. Generally, the top-N heapsort algorithm has lower complexity than quicksort— $O(M \log N)$.

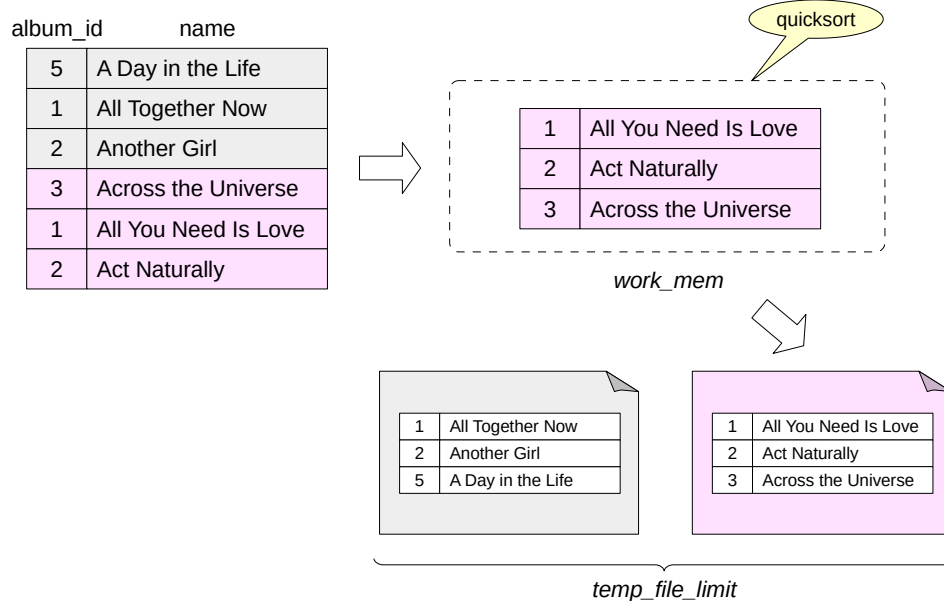
External sort



If the row set is too large, it won't fit into memory all at once. In such cases, an external sort is used.

The row set is loaded into memory while there's capacity, then sorted and written to a temporary file.

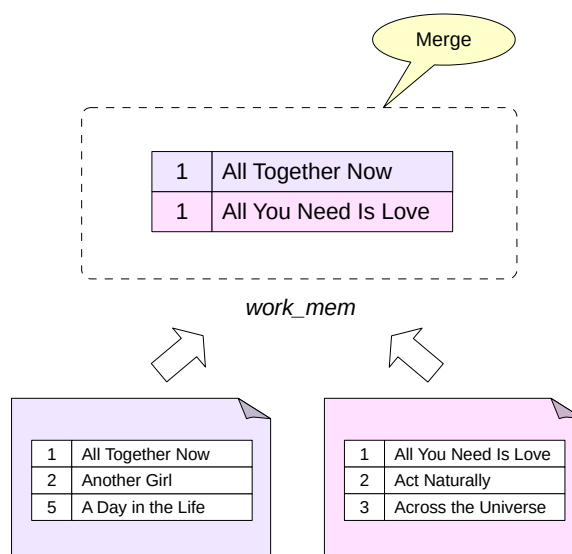
External sort



This procedure is repeated as needed to write all the data to files, each sorted separately.

Recall that the total size of session temporary files (excluding temporary tables) is constrained by the value of the `temp_file_limit` parameter.

External sort



9

Subsequently, multiple files (two or more) are merged while maintaining the line order.

Merging doesn't require much memory; you just need to keep one line from each file (as shown in the example on the slide). The minimum (maximum) is selected from these lines and returned as part of the output, with a new line read from the same file to replace it. In practice, lines are read in batches to speed up input/output, rather than one at a time.

When there isn't enough RAM to merge all files at once, the process begins by merging a subset of files and saving the result to a temporary file. This is then merged with other temporary files, and the process continues iteratively.

We won't go into all the details of the sorting implementation here; you can find them in the file `src/backend/utils/sort/tuplesort.c`.

The evolution of sorting techniques in PostgreSQL is thoroughly covered in Gregory Stark's presentation «Sorting Through The Ages»: https://wiki.postgresql.org/images/5/59/Sorting_through_the_ages.pdf with a synchronized Russian translation of the talk available at <https://pgconf.ru/talk/1587768>

External Sorting External Sort

Example execution plan using external sorting (Sort Method: external merge):

```
=> EXPLAIN (analyze, buffers, timing off, summary off)
SELECT *
FROM flights
ORDER BY scheduled_departure;
```

QUERY PLAN

```
-----
Sort (cost=31883.96..32421.12 rows=214867 width=63) (actual rows=214867 loops=1)
  Sort Key: scheduled_departure
  Sort Method: external merge  Disk: 17112kB
  Buffers: shared hit=3 read=2624, temp read=2139 written=2145
  -> Seq Scan on flights (cost=0.00..4772.67 rows=214867 width=63) (actual rows=214867
loops=1)
    Buffers: shared read=2624
Planning:
  Buffers: shared hit=10 read=1
(8 rows)
```

Note that the Sort node reads and writes temporary data (temp reads and writes).

Let's raise the work_mem value:

```
=> SET work_mem = '48MB';
```

SET

```
=> EXPLAIN (analyze, buffers, timing off, summary off)
SELECT *
FROM flights
ORDER BY scheduled_departure;
```

QUERY PLAN

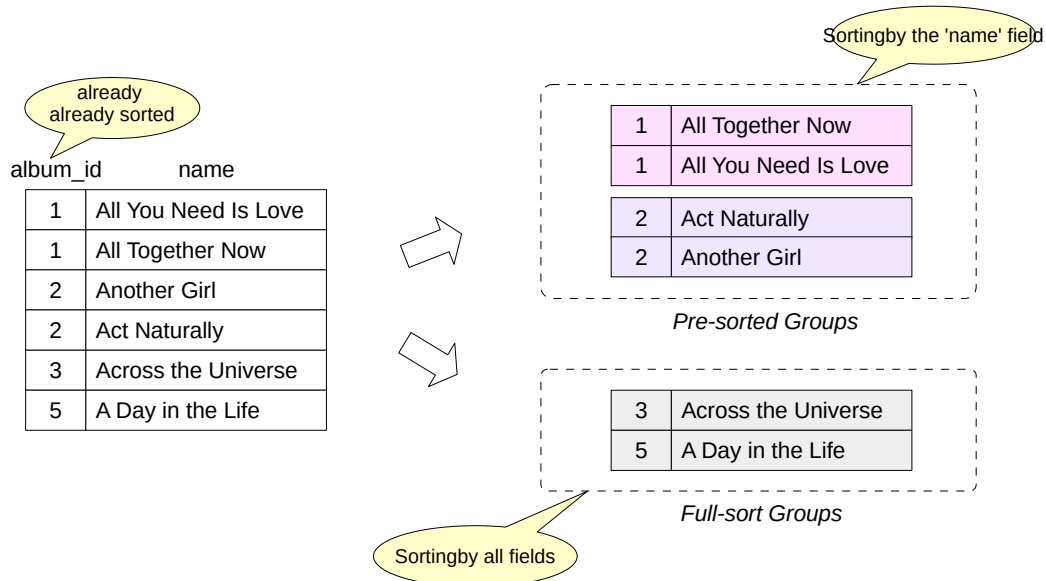
```
-----
Sort (cost=23802.46..24339.62 rows=214867 width=63) (actual rows=214867 loops=1)
  Sort Key: scheduled_departure
  Sort Method: quicksort  Memory: 26161kB
  Buffers: shared hit=2624
  -> Seq Scan on flights (cost=0.00..4772.67 rows=214867 width=63) (actual rows=214867
loops=1)
    Buffers: shared hit=2624
(6 rows)
```

Now that all rows fit in memory, the planner opted for a more efficient quicksort.

```
=> RESET work_mem;
```

RESET

Incremental Sort



11

Let's suppose you want to sort a set by keys $K_1 \dots K_m \dots K_n$, and it's known that the set is already sorted by some of the first keys $K_1 \dots K_m$. If the dataset is already sorted by the first m keys, there's no need to re-sort the entire dataset. Instead, you can split the set into groups, with each group containing only rows with the same $K_1 \dots K_m$ values (arranged consecutively) And then sort each group by the remaining keys $K_{m+1} \dots K_n$. This approach is referred to as incremental sorting .

The algorithm processes relatively large row groups separately while merging smaller groups and fully sorting them. As the data sets being sorted become smaller, the memory requirements also decrease. However, increasing the number of groups leads to higher overhead.

Sets can be handled either in RAM or on disk when there's not enough work_mem available.

Incremental sorting enables delivering results as soon as the first group is processed, without waiting for the entire dataset to be sorted.

Disable incremental sorting by using the enable_incremental_sort parameter.

Incremental Sorting Incremental Sort

Incremental Sorting can use both in-memory and external sorting.

To illustrate, let's create an index on the bookings table:

```
=> CREATE INDEX ON bookings(total_amount);
```

CREATE INDEX

Let's take a look at an example of Incremental Sorting:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT *
FROM bookings
ORDER BY total_amount, book_date;
```

QUERY PLAN

```
-----
-----
Incremental Sort (actual rows=2111110 loops=1)
  Sort Key: total_amount, book_date
  Presorted Key: total_amount
  Full-sort Groups: 2823  Sort Method: quicksort  Average Memory: 28kB  Peak Memory: 28kB
  Pre-sorted Groups: 2624  Sort Method: quicksort  Average Memory: 2193kB  Peak Memory:
2263kB
  -> Index Scan using bookings_total_amount_idx on bookings (actual rows=2111110
loops=1)
(6 rows)
```

Here, the data retrieved from the bookings table using the newly created bookings_total_amount_idx index is already sorted by the total_amount column (Presorted Key), so the remaining task is to sort the rows by the book_date column.

The Pre-sorted Groups row refers to large groups that were partially sorted by the book_date column, while the Full-sort Groups row indicates small groups that were merged and fully sorted. In this example, all groups fit into the allocated memory, so quicksort was used.

Let's reduce work_mem and rerun the query:

```
=> SET work_mem = '128 kB';
```

SET

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT *
FROM bookings
ORDER BY total_amount, book_date;
```

QUERY PLAN

```
-----
-----
Incremental Sort (actual rows=2111110 loops=1)
  Sort Key: total_amount, book_date
  Presorted Key: total_amount
  Full-sort Groups: 2823  Sort Method: quicksort  Average Memory: 28kB  Peak Memory: 28kB
  Pre-sorted Groups: 2624  Sort Methods: quicksort, external merge  Average Memory: 0kB
Peak Memory: 45kB  Average Disk: 1047kB  Peak Disk: 1088kB
  -> Index Scan using bookings_total_amount_idx on bookings (actual rows=2111110
loops=1)
(6 rows)
```

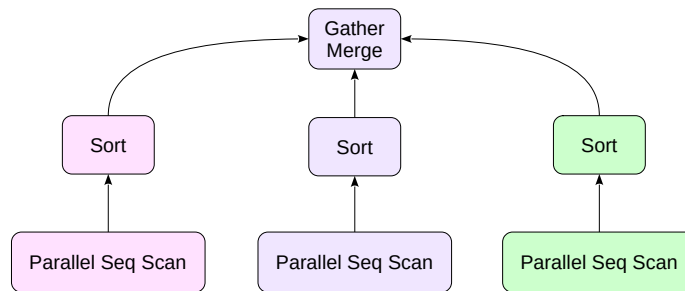
Now, with insufficient memory, some large groups had to undergo external sorting using temporary files (Pre-sorted Groups ... external merge).

```
=> RESET work_mem;
```

RESET

The Gather Merge node preserves the sorted order.

выполняет слияние данных, поступающих от дочерних узлов



Sorting can be part of parallel execution plans. Each worker process sorts its portion of the data and sends the sorted results to a parent node, which combines them into a single set.

But the Gather node isn't suitable for this purpose because it outputs results in the order they are received from the worker processes.

Therefore, such plans use a Gather Merge node to maintain the sorted order of incoming rows. To achieve this, it uses a merge algorithm to combine multiple sorted sets into a single set.

In parallel execution plans

A query that sorts a large table on an unindexed column can run in parallel:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT *
FROM bookings
ORDER BY book_date;
```

QUERY PLAN

```
-----
Gather Merge (actual rows=2111110 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  -> Sort (actual rows=703703 loops=3)
        Sort Key: book_date
        Sort Method: external merge  Disk: 23488kB
        Worker 0:  Sort Method: external merge  Disk: 23024kB
        Worker 1:  Sort Method: external merge  Disk: 21720kB
        -> Parallel Seq Scan on bookings (actual rows=703703 loops=3)
(9 rows)
```

Here, each process reads and sorts its portion of the table, and then the sorted data sets are combined when passed to the leader process, preserving the order.

Building the B-Tree

Sorting is employed

All rows are sorted first.
Then the rows are gathered into leaf index pages
The pointers are collected into next-level pages.
and so on until the root is reached

Can run in parallel

max_parallel_maintenance_workers

Constraint

maintenance_work_mem, as the operation is not frequent

When building an index (specifically a B-tree), the server would add records to an empty index one at a time, processing the table's rows in sequence. However, this approach is highly inefficient.

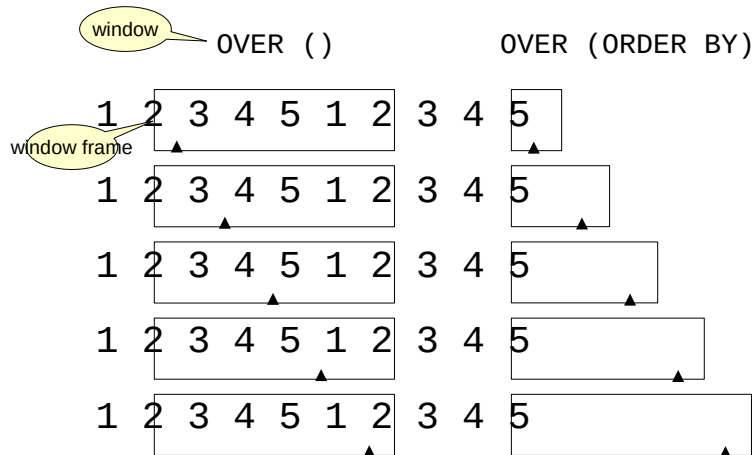
Therefore, sorting is used when creating or rebuilding indexes: all table rows are sorted and then organized into leaf index pages. The upper levels of the tree, consisting of references to elements from lower-level pages, are then constructed until only one page remains — this becomes the root of the tree.

Sorting is done in the same way as described earlier. However, the memory limit is not determined by *work_mem* but by *maintenance_work_mem*, as the index creation process isn't too frequent and it's beneficial to allocate more memory for it.

Index building can be done in parallel. Number of worker processes the number of worker processes is determined similarly to parallel queries (depending on the table size), but is limited by the *max_parallel_maintenance_worker* parameter.

Window functions

The window defines the set of rows to be aggregated for each row.
In the sorted set, the frame boundaries can shift.



16

Using sorting in window functions has unique characteristics.

Unlike ordinary aggregate functions, window functions process every row in the set without altering its size. In the `OVER` clause, after the function name, a window is specified that defines the set of rows processed by the window function for each row in the dataset. Such a sample is referred to as a window frame.

If the window is specified as `OVER()`, the window frame is the same for each row and includes all rows in the dataset.

Window Functions

Let's examine the query plan for a query that computes the running total of bookings:

```
=> EXPLAIN SELECT *, sum(total_amount) OVER (ORDER BY book_date)
FROM bookings;
```

QUERY PLAN

```
-----
WindowAgg (cost=342915.67..379860.10 rows=2111110 width=53)
  -> Sort (cost=342915.67..348193.45 rows=2111110 width=21)
      Sort Key: book_date
      -> Seq Scan on bookings (cost=0.00..34558.10 rows=2111110 width=21)
(4 rows)
```

The window function is executed in the WindowAgg node, which receives sorted data from the Sort child node.

Adding other window functions that use the same row order (not necessarily with a matching window) and the ORDER BY clause avoids unnecessary sorting:

```
=> EXPLAIN SELECT *,
sum(total_amount) OVER (ORDER BY book_date),
avg(total_amount) OVER (ORDER BY book_date ROWS BETWEEN 3 PRECEDING AND 3 FOLLOWING)
FROM bookings
ORDER BY book_date;
```

QUERY PLAN

```
-----
WindowAgg (cost=342915.67..411526.75 rows=2111110 width=85)
  -> WindowAgg (cost=342915.67..379860.10 rows=2111110 width=53)
      -> Sort (cost=342915.67..348193.45 rows=2111110 width=21)
          Sort Key: book_date
          -> Seq Scan on bookings (cost=0.00..34558.10 rows=2111110 width=21)
(5 rows)
```

Here, there are two WindowAgg nodes (each handling its own window) but a single Sort node. The query cost increased, but only marginally.

Of course, window functions requiring different row orders force the server to re-sort the data:

```
=> EXPLAIN SELECT *,
sum(total_amount) OVER (ORDER BY book_date),
count(*) OVER (ORDER BY book_ref)
FROM bookings;
```

QUERY PLAN

```
-----
WindowAgg (cost=422743.30..459687.73 rows=2111110 width=61)
  -> Sort (cost=422743.30..428021.08 rows=2111110 width=29)
      Sort Key: book_date
      -> WindowAgg (cost=0.43..99951.73 rows=2111110 width=29)
          -> Index Scan using bookings_pkey on bookings (cost=0.43..68285.08
rows=2111110 width=21)
(5 rows)
```

In this example, the lower WindowAgg node (corresponding to the count function) receives an ordered set of rows through an index, while the upper WindowAgg node (corresponding to the sum function) has the Sort node re-sort the rows.

Sorting in window functions can employ any of the methods discussed above. For example, quicksort:

```
=> EXPLAIN (analyze, buffers, timing off, summary off)
SELECT *, count(*) OVER (ORDER BY seat_no)
FROM seats;
```

QUERY PLAN

```
-----  
WindowAgg (cost=90.93..114.36 rows=1339 width=23) (actual rows=1339 loops=1)  
  Buffers: shared read=8  
    -> Sort (cost=90.93..94.28 rows=1339 width=15) (actual rows=1339 loops=1)  
        Sort Key: seat_no  
        Sort Method: quicksort Memory: 101kB  
        Buffers: shared read=8  
        -> Seq Scan on seats (cost=0.00..21.39 rows=1339 width=15) (actual rows=1339  
loops=1)  
            Buffers: shared read=8  
(8 rows)
```

Or a foreign key:

```
=> EXPLAIN (analyze, buffers, timing off, summary off)  
SELECT *, sum(total_amount) OVER (ORDER BY book_date)  
FROM bookings;
```

QUERY PLAN

```
-----  
WindowAgg (cost=342915.67..379860.10 rows=2111110 width=53) (actual rows=2111110  
loops=1)  
  Buffers: shared hit=13447, temp read=16491 written=16528  
    -> Sort (cost=342915.67..348193.45 rows=2111110 width=21) (actual rows=2111110  
loops=1)  
        Sort Key: book_date  
        Sort Method: external merge Disk: 65984kB  
        Buffers: shared hit=13447, temp read=16491 written=16528  
        -> Seq Scan on bookings (cost=0.00..34558.10 rows=2111110 width=21) (actual  
rows=2111110 loops=1)  
            Buffers: shared hit=13447  
(8 rows)
```

Sorting is used during query execution and for building B-trees

There are various ways to implement sorting

- in-memory
- external (uses temporary files)
- incremental

Indexes can help avoid sorting

1. What execution plan will be selected for the following query?
`SELECT * FROM flights`
Will the execution plan change if we increase `work_mem` to 32 MB?
2. Create an index on the `passenger_name` and `passenger_id` columns of the `tickets` table. Did this operation require a temporary file?

2. 2. Turn on logging for temporary file usage by setting the `log_temp_files` parameter to zero.

1. Query Execution with Sorting

Let's run the query with the default parameter values:

```
=> EXPLAIN (analyze, buffers, costs off, timing off)
SELECT *
FROM flights
ORDER BY scheduled_departure;

               QUERY PLAN
-----
Sort (actual rows=214867 loops=1)
  Sort Key: scheduled_departure
  Sort Method: external merge  Disk: 17112kB
  Buffers: shared hit=3 read=2624, temp read=2139 written=2145
  -> Seq Scan on flights (actual rows=214867 loops=1)
      Buffers: shared read=2624
Planning:
  Buffers: shared hit=104 read=25 dirtied=6
Planning Time: 5.171 ms
Execution Time: 307.688 ms
(10 rows)
```

The server performed an external sort (Sort Method: external merge).

Pay attention to the number of pages and I/O type (Buffers): the temp read and written values indicate that the server used temporary files.

Run the query multiple times — you'll notice that the server consistently runs out of memory.

We'll increase the `work_mem` parameter to 32 MB:

```
=> SET work_mem = '32 MB';
```

SET

Increasing `work_mem` allows the server to use more random access memory for sorting.

Run the query again and compare the execution plans

```
=> EXPLAIN (analyze, buffers, costs off, timing off)
SELECT *
FROM flights
ORDER BY scheduled_departure;

               QUERY PLAN
-----
Sort (actual rows=214867 loops=1)
  Sort Key: scheduled_departure
  Sort Method: quicksort  Memory: 26161kB
  Buffers: shared hit=2624
  -> Seq Scan on flights (actual rows=214867 loops=1)
      Buffers: shared hit=2624
Planning Time: 0.076 ms
Execution Time: 166.535 ms
(8 rows)
```

The server has switched to in-memory sorting (Sort Method: quicksort), with the temp read and written fields no longer appearing in the Buffers line.

2. Building an Index

Enable temporary file logging:

```
=> SET log_temp_files = 0;
```

SET

Current `maintenance_work_mem` setting:

```
=> SHOW maintenance_work_mem;
```

```
maintenance_work_mem
-----
64MB
(1 row)
```

Creating an Index:

```
=> \timing on
```

Timing is on.

```
=> CREATE INDEX ON tickets(passenger_name, passenger_id);
```

CREATE INDEX

Time: 30767.411 ms (00:30.767)

A temporary file was required:

```
student$ tail -n 2 /var/log/postgresql/postgresql-16-main.log
```

```
2025-10-18 23:00:22.995 MSK [123782] postgres@demo LOG:  temporary file: path
"base/pgsql_tmp/pgsql_tmp123782.0.fileset/0.0", size 64184320
```

```
2025-10-18 23:00:22.995 MSK [123782] postgres@demo STATEMENT:  CREATE INDEX ON
tickets(passenger_name, passenger_id);
```