

Join MethodsHash Join



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.

Sequential hash join: single- and two-pass

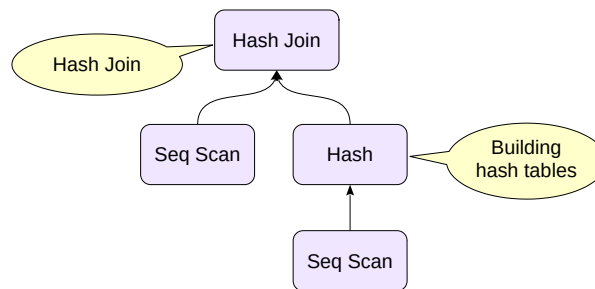
Computational complexity

Parallel hash join: single- and two-pass

Hash Join

A hash table is constructed from the rows of one set, and the rows of the other set are matched against it.

Equi-joins only



The main idea of hashing is discussed in the "Types of Indexes" and "Grouping" sections.

Hashing is used for joining as follows. Initially, a hash table is constructed based on one of the datasets within the Hash node. Then, in the Hash Join node, the rows from the other dataset are compared to the built table.

Similar to a hash index, a hash join can only function with an equality condition: the hash function does not maintain the order of values.

Let's look at an example of how the join works.

One-pass Join

Used when the hash table fits into available memory

Building the Hash Table

```
SELECT a.title, s.name FROM albums a JOIN songs s ON  
a.id = s.album_id;
```

id	title	year
3	Let It Be	1970
1	Yellow Submarine	1969
6	Abbey Road	1969
4	The Beatles	1968

inner set



	hash code	id	title
00	010001 00	4	The Beatles
01	101000 01	3	Let It Be
	100010 01	6	Abbey Road
10			
11	110001 11	1	Yellow Submarine

$work_mem \times hash_mem_multiplier$

5

The first step involves constructing a hash table in memory.

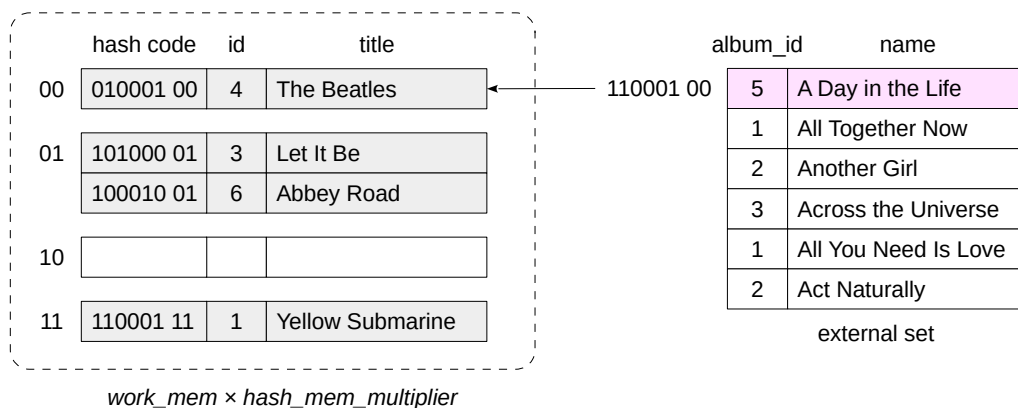
The rows of the first (inner) set are processed sequentially, with a hash function computed for each row based on the values of the fields involved in the join condition (in our example, these are numeric ID values).

The computed hash code and all fields involved in the join condition or used in the query are stored in the hash table bucket.

The size of the hash table in memory is constrained by $work_mem \times hash_mem_multiplier$. Optimal performance is achieved when the entire hash table fits within this memory allocation. Therefore, the planner typically selects the smaller of the two row sets as the inner one. It's also a good idea to avoid including unnecessary fields in the query, such as the asterisk, to prevent the hash table from being overloaded with extra data.

Row Matching

```
SELECT a.title, s.name FROM albums a JOIN songs s ON  
a.id = s.album_id;
```



6

During the second phase, we read the second set of rows in sequence. While reading, we compute the hash function from the fields' values involved in the join condition. If a row is found in the matching bucket of the hash table.

- 1) with a matching hash code
 - 2) and with field values that satisfy the join condition
- then we found a match

Simply checking the hash code isn't enough. First, not all join conditions listed in the query are taken into account during a hash join operation (as only equijoins are supported). Secondly, collisions may occur where different values result in the same hash code (even though the probability is low, it still exists).

In our example, the first row has no match.

Row Matching

```
SELECT a.title, s.name FROM albums a JOIN songs s ON  
a.id = s.album_id;
```

	hash code	id	title
00	010001 00	4	The Beatles
01	101000 01	3	Let It Be
	100010 01	6	Abbey Road
10			
11	110001 11	1	Yellow Submarine

album_id	name
5	A Day in the Life
1	All Together Now
2	Another Girl
3	Across the Universe
1	All You Need Is Love
2	Act Naturally

110001 11

The second row of the second set yields a match that can be passed up to the higher-level node in the query plan: ('Yellow Submarine', 'All Together Now').

Row Matching

```
SELECT a.title, s.name FROM albums a JOIN songs s ON  
a.id = s.album_id;
```

	hash code	id	title
00	010001 00	4	The Beatles
01	101000 01	3	Let It Be
	100010 01	6	Abbey Road
10			
11	110001 11	1	Yellow Submarine

111101 10

album_id	name
5	A Day in the Life
1	All Together Now
2	Another Girl
3	Across the Universe
1	All You Need Is Love
2	Act Naturally

No match exists for the third row (the corresponding hash table bucket is empty)

Row Matching

```
SELECT a.title, s.name FROM albums a JOIN songs s ON  
a.id = s.album_id;
```

	hash code	id	title
00	010001 00	4	The Beatles
01	101000 01	3	Let It Be
	100010 01	6	Abbey Road
10			
11	110001 11	1	Yellow Submarine

album_id	name
5	A Day in the Life
1	All Together Now
2	Another Girl
3	Across the Universe
1	All You Need Is Love
2	Act Naturally

101000 01

The fourth match is ("Let It Be", "Across the Universe").

Note that the hash table bucket contains two rows from the first set, and both must be examined.

Row Matching

```
SELECT a.title, s.name FROM albums a JOIN songs s ON  
a.id = s.album_id;
```

	hash code	id	title
00	010001 00	4	The Beatles
01	101000 01	3	Let It Be
	100010 01	6	Abbey Road
10			
11	110001 11	1	Yellow Submarine

album_id	name
5	A Day in the Life
1	All Together Now
2	Another Girl
3	Across the Universe
1	All You Need Is Love
2	Act Naturally

110001 11

10

The fifth row corresponds to ("Yellow Submarine", "All You Need Is Love").

Row Matching

```
SELECT a.title, s.name FROM albums a JOIN songs s ON  
a.id = s.album_id;
```

	hash code	id	title
00	010001 00	4	The Beatles
01	101000 01	3	Let It Be
	100010 01	6	Abbey Road
10			
11	110001 11	1	Yellow Submarine

album_id	name
5	A Day in the Life
1	All Together Now
2	Another Girl
3	Across the Universe
1	All You Need Is Love
2	Act Naturally

111101 10

11

No match exists for the sixth row. The connection process is complete.
The algorithm's source code is located in the file
src/backend/executor/nodeHashjoin.c

One-Pass Hash Join

When selectivity is low, the optimizer tends to choose a hash join:

```
=> EXPLAIN (costs off)
SELECT *
FROM aircrafts a
JOIN seats s ON a.aircraft_code = s.aircraft_code;
```

QUERY PLAN

```
-----
Hash Join
  Hash Cond: (s.aircraft_code = ml.aircraft_code)
    -> Seq Scan on seats s
    -> Hash
        -> Seq Scan on aircrafts_data ml
(5 rows)
```

The Hash Join node begins by accessing its Hash child node. This child node retrieves all rows from its own child (here, a Seq Scan) and builds a hash table.

Then the Hash Join node accesses the second child node and joins the rows, gradually returning the results.

Let's see how this query is executed

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT *
FROM aircrafts a
JOIN seats s ON a.aircraft_code = s.aircraft_code;
```

QUERY PLAN

```
-----
Hash Join (actual rows=1339 loops=1)
  Hash Cond: (s.aircraft_code = ml.aircraft_code)
    -> Seq Scan on seats s (actual rows=1339 loops=1)
    -> Hash (actual rows=9 loops=1)
        Buckets: 1024 Batches: 1 Memory Usage: 9kB
        -> Seq Scan on aircrafts_data ml (actual rows=9 loops=1)
(6 rows)
```

The hash table was able to fit into memory (Batches: 1). The Buckets parameter indicates the number of buckets in the hash table, while Memory Usage shows the volume of used random access memory.

Note that the hash table was built from the smaller set of rows.

Hash Join has several variations, including the previously mentioned Left, Semi, and Anti, as well as Right and Full for right and full joins respectively:

```
=> EXPLAIN (costs off)
SELECT *
FROM aircrafts a
FULL JOIN seats s ON a.aircraft_code = s.aircraft_code;
```

QUERY PLAN

```
-----
Hash Full Join
  Hash Cond: (s.aircraft_code = ml.aircraft_code)
    -> Seq Scan on seats s
    -> Hash
        -> Seq Scan on aircrafts_data ml
(5 rows)
```

```
=> EXPLAIN (costs off)
SELECT *
FROM seats s
RIGHT JOIN aircrafts a ON a.aircraft_code = s.aircraft_code;
```

QUERY PLAN

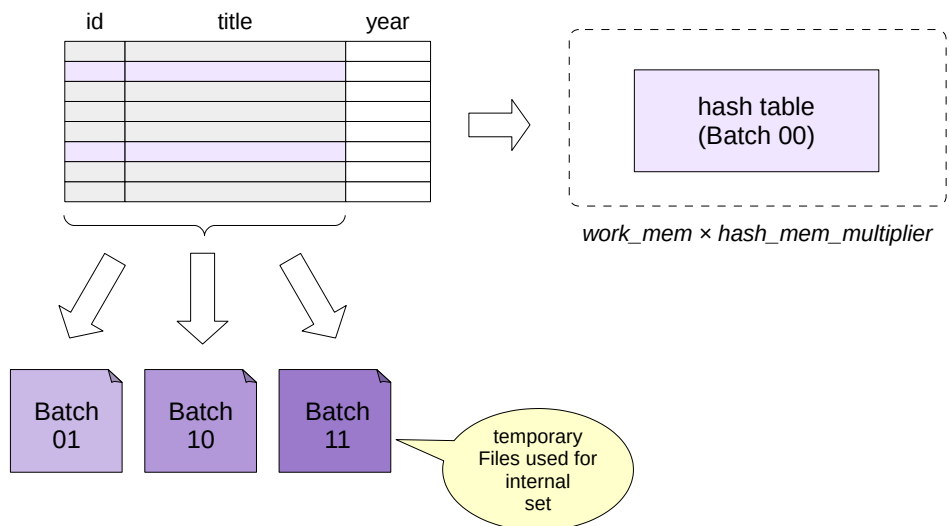
```
-----  
Hash Right Join  
  Hash Cond: (s.aircraft_code = ml.aircraft_code)  
    -> Seq Scan on seats s  
    -> Hash  
          -> Seq Scan on aircrafts_data ml  
(5 rows)
```

Note that the hash table is always built from the smaller set of rows, regardless of the order of the tables in the join.

Two-pass Join

Used when the hash table cannot fit into main memory: data sets are divided into batches and joined sequentially.

Building the Hash Table



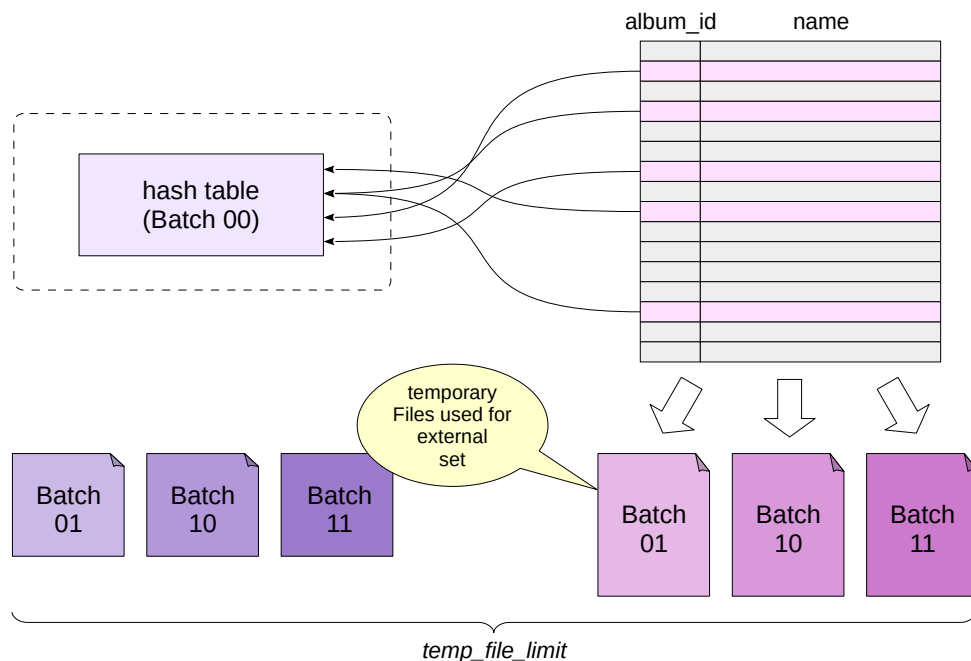
If the hash table exceeds the memory limit set by $work_mem \times hash_mem_multiplier$, the initial (internal) set of rows is divided into separate packages. A certain number of hash code bits are used to distribute the data into packages, so the number of packages is always a power of two. Ideally, each package would contain roughly the same number of rows, but if row values are repeated, skew can occur.

During query planning, the minimum number of packages required is calculated in advance to ensure that the hash table for each package fits into memory. This number remains unchanged even if the optimizer made errors in its estimates, but can be dynamically adjusted when necessary.

The hash table for the first package stays in memory, while rows belonging to other packages are written to disk as temporary files — each package in its own file.

The figure shows four packages.

Join - Package 1



15

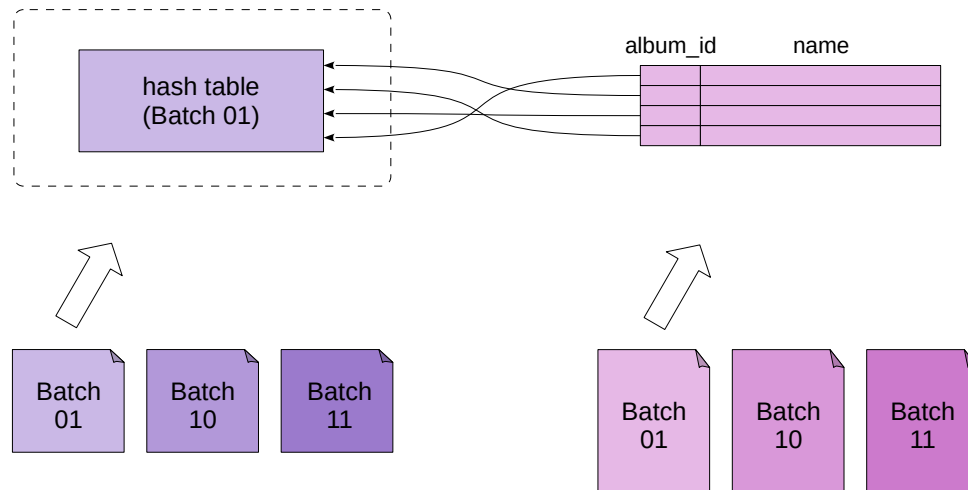
Next, the second (external) set of strings is processed. If the string belongs to the first Package, it is matched against the hash table that contains the first Package. There's no need to match the string against other packages — they can't contain a match, as the hash codes are guaranteed to differ.

If the string belongs to another package, it is written to disk — again, each package is stored in its own temporary file.

Therefore, with N packages, a total of $2(N-1)$ files are used.

Note that the use of temporary files on disk is limited by the `temp_file_limit` parameter, which sets the total disk space limit for the session. (Temporary table buffers are excluded from this limit.) (Буферы временных таблиц в это ограничение не входят.)

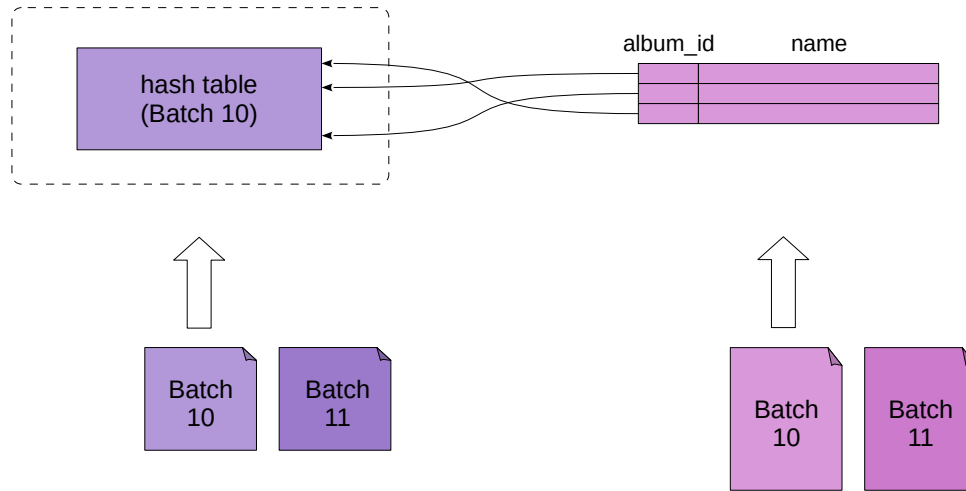
Matching: Package 2



Next, all packages are processed in turn, starting with the second. The internal set's rows are loaded into a hash table from a temporary file, followed by reading the external set's rows from another temporary file and matching them against the hash table.

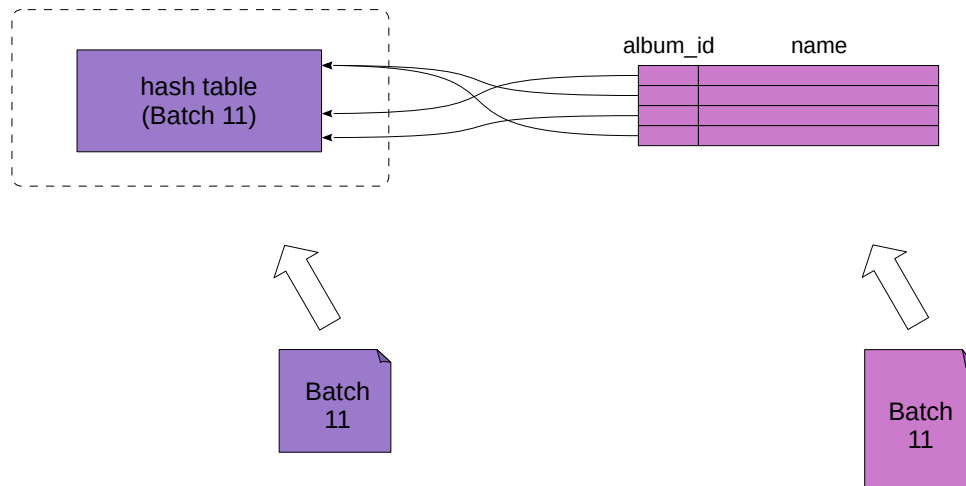
The procedure is repeated for all remaining packages, which amount to $N-1$. The figure illustrates the connection for the second package (01).

Join - Package 3



The figure illustrates the connection for the third package (10).

Mapping: Package 4



18

After processing the final package, the connection is closed and temporary files are freed.

Therefore, when insufficient RAM is available, the join algorithm operates in two passes: each packet (except the first) must be written to disk and then read back. Of course, this impacts the connection's efficiency. To avoid the inefficiency, make sure that:

- Only the actually needed fields were included in the hash table (the query author's responsibility).
- The hash table is built for the smaller set (responsibility of the planner).

Two-Pass Hash Join

Now let's use large tables and display the I/O statistics:

```
=> EXPLAIN (analyze, buffers, costs off, timing off, summary off)
SELECT *
FROM bookings b
JOIN tickets t ON b.book_ref = t.book_ref;
```

QUERY PLAN

```
-----
Hash Join (actual rows=2949857 loops=1)
  Hash Cond: (t.book_ref = b.book_ref)
  Buffers: shared read=62862, temp read=54218 written=54218
  -> Seq Scan on tickets t (actual rows=2949857 loops=1)
      Buffers: shared read=49415
  -> Hash (actual rows=2111110 loops=1)
      Buckets: 131072 Batches: 32 Memory Usage: 4551kB
      Buffers: shared read=13447, temp written=10701
      -> Seq Scan on bookings b (actual rows=2111110 loops=1)
          Buffers: shared read=13447
Planning:
  Buffers: shared hit=88 read=15 dirtied=5
(12 rows)
```

The hash table no longer fits into memory, so a Two-Pass Hash Join is used. This required 32 batches (Batches).

As shown, the Hash node writes batches to temporary files (temp written), while the Hash Join node both reads and writes (temp read and written).

Let's compare it to the same query that selects only one field:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT b.book_ref
FROM bookings b
JOIN tickets t ON b.book_ref = t.book_ref;
```

QUERY PLAN

```
-----
Hash Join (actual rows=2949857 loops=1)
  Hash Cond: (t.book_ref = b.book_ref)
  -> Seq Scan on tickets t (actual rows=2949857 loops=1)
  -> Hash (actual rows=2111110 loops=1)
      Buckets: 262144 Batches: 16 Memory Usage: 7057kB
      -> Seq Scan on bookings b (actual rows=2111110 loops=1)
(6 rows)
```

Memory usage dropped since the hash table now contains only one field (instead of three), and the number of batches halved.

$\sim N + M$, где

N and M represent the number of rows in the first and second data sets.

Initial overhead for building a hash table

Efficient for a large number of rows

The overall complexity of a hash join is proportional to the combined number of rows in both data sets. Thus, hash joins are significantly more efficient than nested loops when handling large datasets.

However, since a hash table must be built to start the join, the nested loop is more efficient for small row counts.

Hash joins (combined with full table scans) are typically used in OLAP queries that require processing a large number of rows, where throughput is prioritized over response time.

Hash Join Cost

```
=> EXPLAIN SELECT *  
FROM tickets t  
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no;
```

QUERY PLAN

```
-----  
Hash Join (cost=161879.43..498586.96 rows=8391960 width=136)  
Hash Cond: (tf.ticket_no = t.ticket_no)  
-> Seq Scan on ticket_flights tf (cost=0.00..153852.60 rows=8391960 width=32)  
-> Hash (cost=78913.86..78913.86 rows=2949886 width=104)  
-> Seq Scan on tickets t (cost=0.00..78913.86 rows=2949886 width=104)  
(5 rows)
```

The initial cost of the Hash Join node consists of the following costs:

- retrieving the entire first dataset (in this case, tickets);
- The hash table must be built before the join can begin.

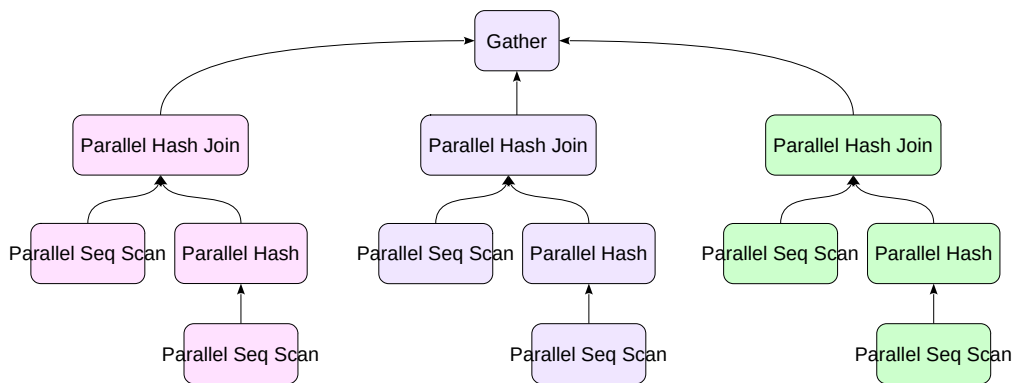
It's worth noting that the Hash node does not account for the cost of building the hash table.

The total cost includes the following additional costs in addition to the initial cost:

- fetching the full second data set (in this case, flights);
- hash table lookup;
- disk accesses when more than one batch is expected

Key takeaway: The cost of a hash join scales with $N + M$, where N and M represent the number of rows in the data sets being joined. When N and M are large, this approach is much more efficient than the product of N and M in nested loop joins.

Efficient algorithm: parallel hash table construction and parallel matching



Unlike other join methods, hash join not only supports parallel execution plans but also has a distinct efficient algorithm. This algorithm enables parallel execution of both join stages: building the hash table from the first (inner) row set and matching the second (outer) row set against it.

The parallel hash join capability is governed by the `enable_parallel_hash` parameter; by default, this parameter is enabled.

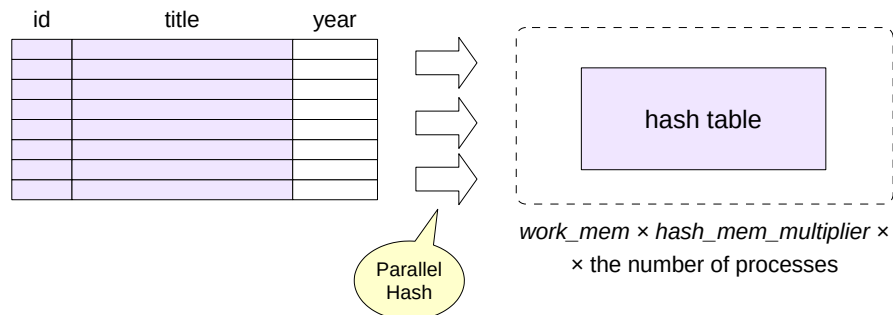
Similar to the sequential algorithm, the parallel version has two approaches: single-pass when sufficient random access memory is available and two-pass.

Let's start with the single-pass approach.

Parallel, single pass

Processes use a shared hash table

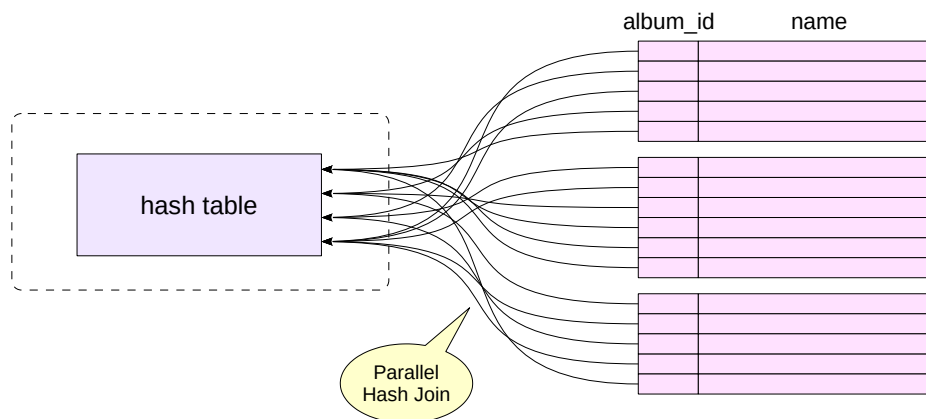
Building the Hash Table



The single-pass algorithm is employed when the hash table can fit into the total memory allocated to all processes involved in the join, which means the hash table's size is constrained by $work_mem \times hash_mem_multiplier \times \text{the number of processes}$.

Processes read the first set of rows in parallel (e.g., using the Parallel Seq Scan node) and build a shared hash table in shared memory, which all processes can access.

Row Matching



Once the hash table is fully built, worker processes begin parallel scanning of the second dataset, comparing the rows they read against the shared hash table. Each process only examines a subset of the data using the hash table.

Single-Pass Parallel Hash Join

Increase the memory allocated to the process for building the hash table.

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

SET

Let's run the aggregation query:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT count(*)
FROM bookings b
JOIN tickets t ON b.book_ref = t.book_ref;
```

QUERY PLAN

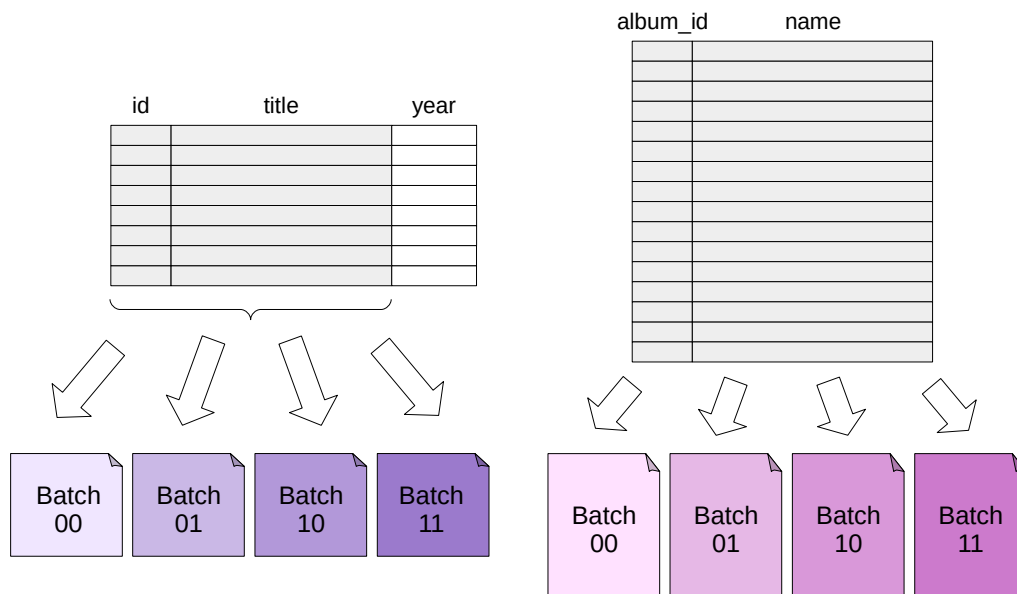
```
-----
--
Finalize Aggregate (actual rows=1 loops=1)
  -> Gather (actual rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Partial Aggregate (actual rows=1 loops=3)
              -> Parallel Hash Join (actual rows=983286 loops=3)
                    Hash Cond: (t.book_ref = b.book_ref)
                    -> Parallel Seq Scan on tickets t (actual rows=983286 loops=3)
                    -> Parallel Hash (actual rows=703703 loops=3)
                          Buckets: 4194304 Batches: 1 Memory Usage: 115392kB
                          -> Parallel Seq Scan on bookings b (actual rows=703703
loops=3)
(11 rows)
```

Note the memory usage (Memory Usage): the amount exceeds the limit set for a single worker process (64MB = work_mem × hash_mem_multiplier), but the hash table fits into the shared memory of three processes. Therefore, a single-pass join is performed (Batches: 1).

Two parallel passes

Row sets are split into batches that are then processed in parallel by worker processes.

Splitting into Batches



28

A hash table may not fit within the memory limit determined by $\text{work_mem} \times \text{hash_mem_multiplier} \times \text{number of processes}$, and this may become apparent during the join execution. In this case, a two-pass algorithm is employed, which differs significantly from both the two-pass sequential and the single-pass parallel approaches.

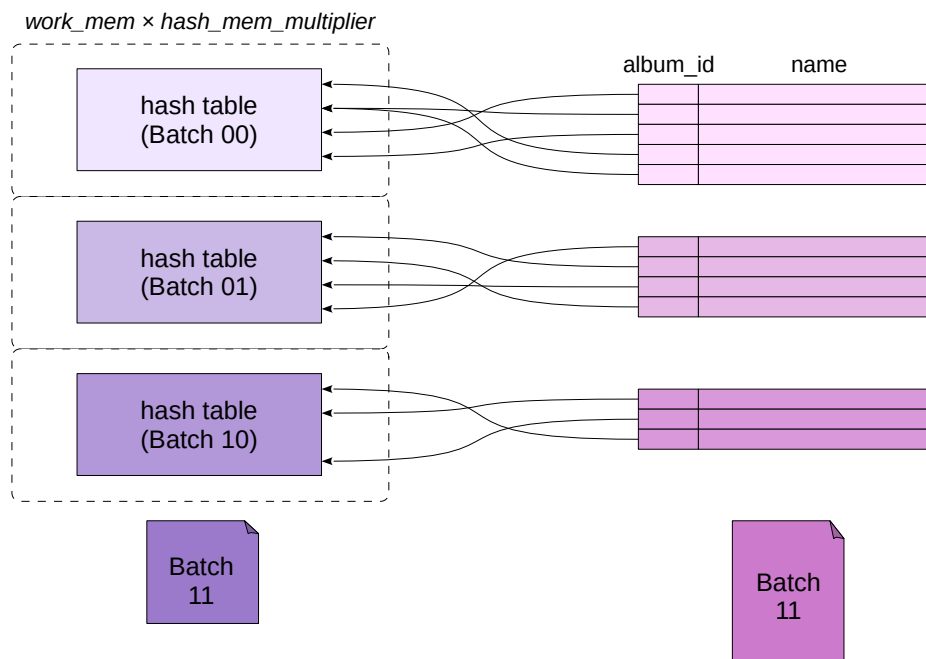
First, worker processes read the first dataset in parallel, split it into batches, and write the batches to temporary files. The first batch is also written to the file; the hash table is not constructed in memory.

Note that each process writes rows to all temporary files, with the writes synchronized.

Next, worker processes read the second dataset in parallel, splitting it into batches and writing them to temporary files.

As a result, $2N$ files are written to disk when there are N batches.

Row Matching



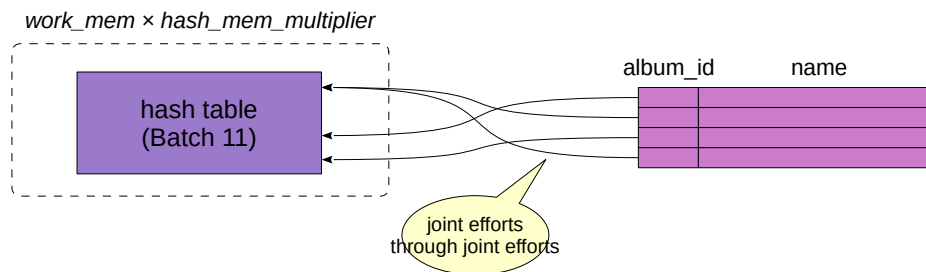
Each worker process then selects one batch.

The process loads the first dataset from the selected batch into the hash table in memory. In this algorithm, each process has its own hash table sized at $work_mem \times hash_mem_multiplier \times r$, but these tables are stored in shared memory, allowing all worker processes to access each table.

Once the hash table is populated, the process reads the second dataset from the selected batch and matches the rows.

Once a process finishes processing a batch, it selects the next unprocessed one.

Row Matching



If there are no unprocessed batches left, the process teams up with another process to help it finish up its batch. They can do that because all the hash tables reside in shared memory.

Using multiple hash tables is more efficient than a single large one: it simplifies coordination and reduces synchronization overhead.

Two-Pass Parallel Hash Join

Now, let's decrease the memory allocation:

```
=> SET work_mem = '16MB';
```

```
SET
```

Let's run the same aggregation query:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT count(*)
FROM bookings b
JOIN tickets t ON b.book_ref = t.book_ref;
```

QUERY PLAN

```
-----
--
Finalize Aggregate (actual rows=1 loops=1)
  -> Gather (actual rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Partial Aggregate (actual rows=1 loops=3)
              -> Parallel Hash Join (actual rows=983286 loops=3)
                    Hash Cond: (t.book_ref = b.book_ref)
                    -> Parallel Seq Scan on tickets t (actual rows=983286 loops=3)
                    -> Parallel Hash (actual rows=703703 loops=3)
                          Buckets: 1048576 Batches: 4 Memory Usage: 28896kB
                          -> Parallel Seq Scan on bookings b (actual rows=703703
loops=3)
(11 rows)
```

The join is now a two-pass join with four batches.

Hash joins require preparation

A hash table must be built

Efficient for large samples

It also supports parallel joins

It depends on the join order

The inner relation should be smaller than the outer relation, to reduce the hash table's size.

Only equijoins are supported.

The 'greater than' and 'less than' operators are not applicable to hash codes.

Unlike nested loop joins, hash joins require setup, such as building a hash table. Until the hash table is built, no result rows can be produced.

Hash joins, however, are efficient with large data volumes. Both row sets are read sequentially and only once (twice if there's insufficient RAM).

A limitation of hash joins is that they only support equijoins. The issue is that hash values can only be compared for equality, and the 'greater than' and 'less than' operations are not applicable.

1. Write a query to show occupied seats in the cabin for all flights. Which join strategy did the planner select? Check whether there was sufficient RAM to allocate the hash tables.
2. Modify the query to display only the total count of occupied seats. How did the query plan change? Why didn't the planner use the same plan for the previous query?
3. Examine the query plan to determine the order of operations.

1. To do this, simply join the flights table with the boarding_passes table.
3. Such a query needs to join three tables: tickets (tickets), ticket_flights (ticket_flights), and flights (flights).

1. List of Used Data Pages

```
=> EXPLAIN SELECT f.flight_no, bp.seat_no
FROM flights f
JOIN boarding_passes bp ON bp.flight_id = f.flight_id;
```

QUERY PLAN

```
-----
Hash Join (cost=8508.51..229819.73 rows=7925688 width=10)
  Hash Cond: (bp.flight_id = f.flight_id)
    -> Seq Scan on boarding_passes bp (cost=0.00..137535.88 rows=7925688 width=7)
    -> Hash (cost=4772.67..4772.67 rows=214867 width=11)
        -> Seq Scan on flights f (cost=0.00..4772.67 rows=214867 width=11)
(5 rows)
```

Hash join was used

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT f.flight_no, bp.seat_no
FROM flights f
JOIN boarding_passes bp ON bp.flight_id = f.flight_id;
```

QUERY PLAN

```
-----
Hash Join (actual rows=7925812 loops=1)
  Hash Cond: (bp.flight_id = f.flight_id)
    -> Seq Scan on boarding_passes bp (actual rows=7925812 loops=1)
    -> Hash (actual rows=214867 loops=1)
        Buckets: 262144 Batches: 2 Memory Usage: 6676kB
        -> Seq Scan on flights f (actual rows=214867 loops=1)
(6 rows)
```

The hash table couldn't fit entirely in memory, requiring two chunks.

2. Number of Used Data Pages

```
=> EXPLAIN SELECT count(*)
FROM flights f
JOIN boarding_passes bp ON bp.flight_id = f.flight_id;
```

QUERY PLAN

```
-----
Finalize Aggregate (cost=114695.55..114695.56 rows=1 width=8)
  -> Gather (cost=114695.34..114695.55 rows=2 width=8)
      Workers Planned: 2
      -> Partial Aggregate (cost=113695.34..113695.35 rows=1 width=8)
          -> Parallel Hash Join (cost=5467.82..105439.41 rows=3302370 width=0)
              Hash Cond: (bp.flight_id = f.flight_id)
              -> Parallel Seq Scan on boarding_passes bp (cost=0.00..91302.70
rows=3302370 width=4)
              -> Parallel Hash (cost=3887.92..3887.92 rows=126392 width=4)
                  -> Parallel Seq Scan on flights f (cost=0.00..3887.92
rows=126392 width=4)
(9 rows)
```

The query planner opted for a parallel plan here. In the previous query, this wasn't cost-effective due to the high data movement between processes, but in this case, only a single value is being transferred.

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT count(*)
FROM flights f
JOIN boarding_passes bp ON bp.flight_id = f.flight_id;
```

QUERY PLAN

```
-----  
-----  
Finalize Aggregate (actual rows=1 loops=1)  
  -> Gather (actual rows=3 loops=1)  
        Workers Planned: 2  
        Workers Launched: 2  
        -> Partial Aggregate (actual rows=1 loops=3)  
              -> Parallel Hash Join (actual rows=2641937 loops=3)  
                    Hash Cond: (bp.flight_id = f.flight_id)  
                    -> Parallel Seq Scan on boarding_passes bp (actual rows=2641937  
loops=3)  
                          -> Parallel Hash (actual rows=71622 loops=3)  
                                Buckets: 262144 Batches: 1 Memory Usage: 10496kB  
                                -> Parallel Seq Scan on flights f (actual rows=71622 loops=3)  
(11 rows)
```

Note the loops column in the nodes above and below the Gather node — it reflects the actual number of processes involved in executing the query.

3. Passengers and Flight Numbers

```
=> EXPLAIN (costs off)  
SELECT t.passenger_name, f.flight_no  
FROM tickets t  
      JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no  
      JOIN flights f ON f.flight_id = tf.flight_id;
```

QUERY PLAN

```
-----  
Hash Join  
  Hash Cond: (tf.flight_id = f.flight_id)  
  -> Hash Join  
        Hash Cond: (tf.ticket_no = t.ticket_no)  
        -> Seq Scan on ticket_flights tf  
        -> Hash  
              -> Seq Scan on tickets t  
  -> Hash  
        -> Seq Scan on flights f  
(9 rows)
```

The join between tickets and ticket_flights is performed first, with the hash table constructed from the tickets table.

Then, flights are joined with the result of the first join, with the hash table built from the flights table.