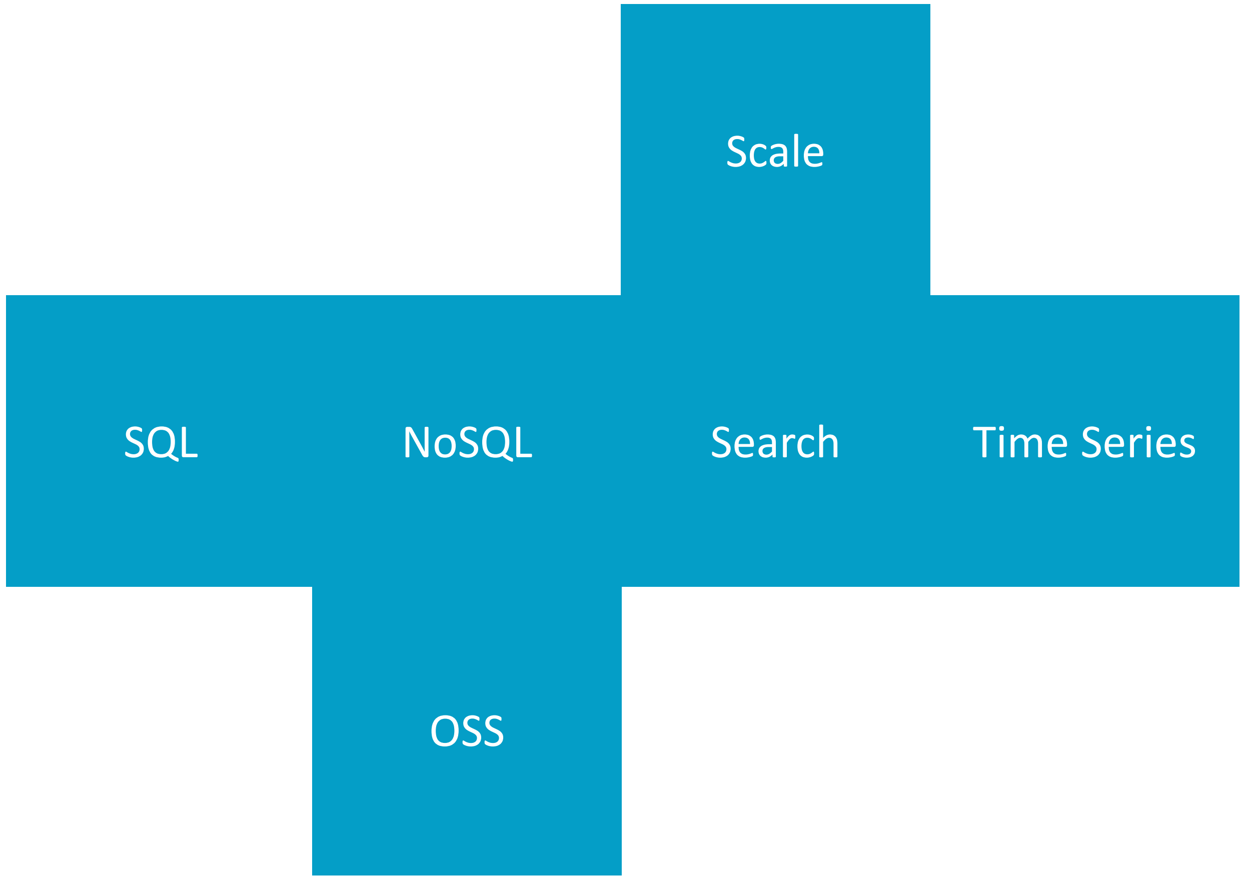
Distributed Join Algorithms in CrateDB

How We Made Distributed Joins 23 Thousand Times Faster

Marija Selakovic
Developer Advocate
marija@crate.io

CRATE.IO

CrateDB



- Since 2014: https://github.com/crate/crate
- The **distributed** database built for data-intensive **analytics solutions**
- **PostgreSQL interface**
- Based on **Apache Lucene**
- Open Source under Apache License v2.0

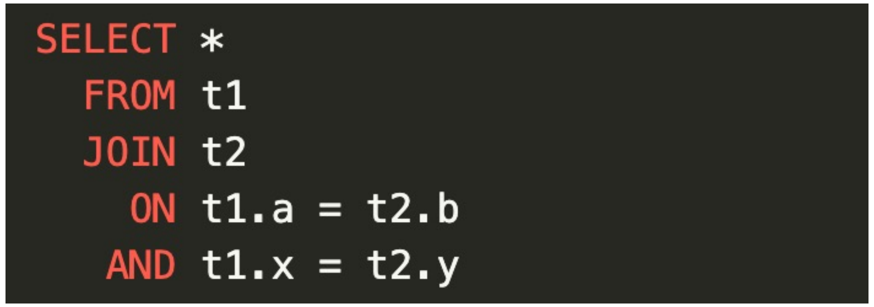
Motivation

- CrateDB analyzes and indexes data at scale: TB or even PB of data
- Allows efficient query over large datasets
- SQL compatibility – developers rely on JOIN operators when querying multiple tables
- Distributed architecture poses challenges for efficient JOIN operators
- Default algorithm: nested loop join algorithm with some optimizations for distributed query execution
- **This talk**: focus on *equi-join* and the performance improvements with distributed block hash join algorithm

Joins in CrateDB

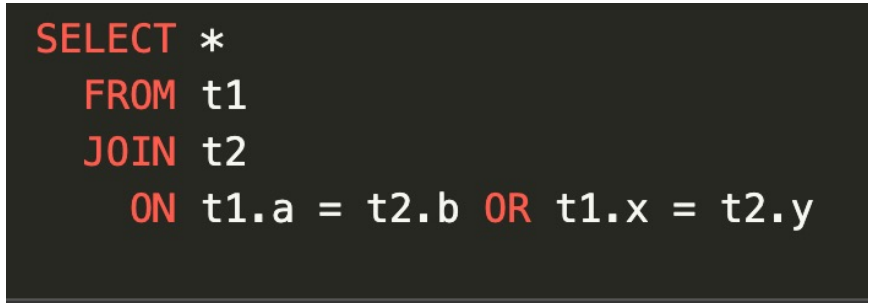
- CrateDB supports:
  - Cross joins
  - Inner joins
  - Outer joins
- *Equi-join*: type of inner join with the condition that has one or more multiple equality operators chained together with `AND` operator
- Nested loop algorithm comes with a high-performance cost
- Quadratic time complexity: `O(M*N)`
- `M,N` – number of rows of the two tables joined

Condition that belongs to equi-join:



```
SELECT *
  FROM t1
  JOIN t2
    ON t1.a = t2.b
   AND t1.x = t2.y
```

Condition that does not belong to equi-join:



```
SELECT *
  FROM t1
  JOIN t2
    ON t1.a = t2.b OR t1.x = t2.y
```

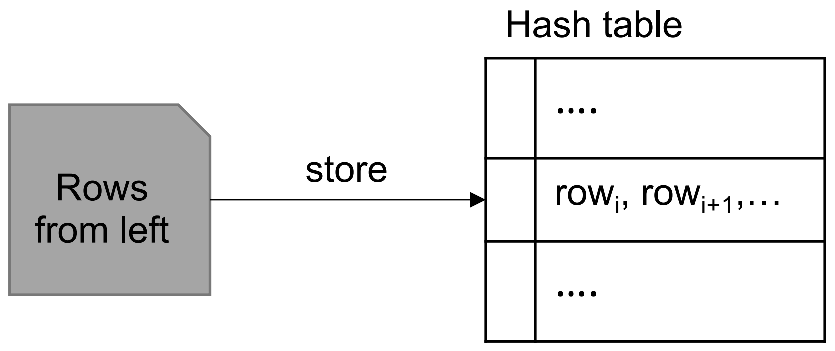
Hash Join Algorithm

- The common method for processing equi joins in relational databases
- Consists of two phases: build phase and probe phase
- Build phase: scan the smaller table and store hash values of join attributes in the hash table
- Probe phase: each row in the other table is probed against the hash table
- Time complexity: **`O(M+N)`**
- Hash join algorithm is a good choice is the smaller table **fits into memory**

Hash Join Algorithm (1)

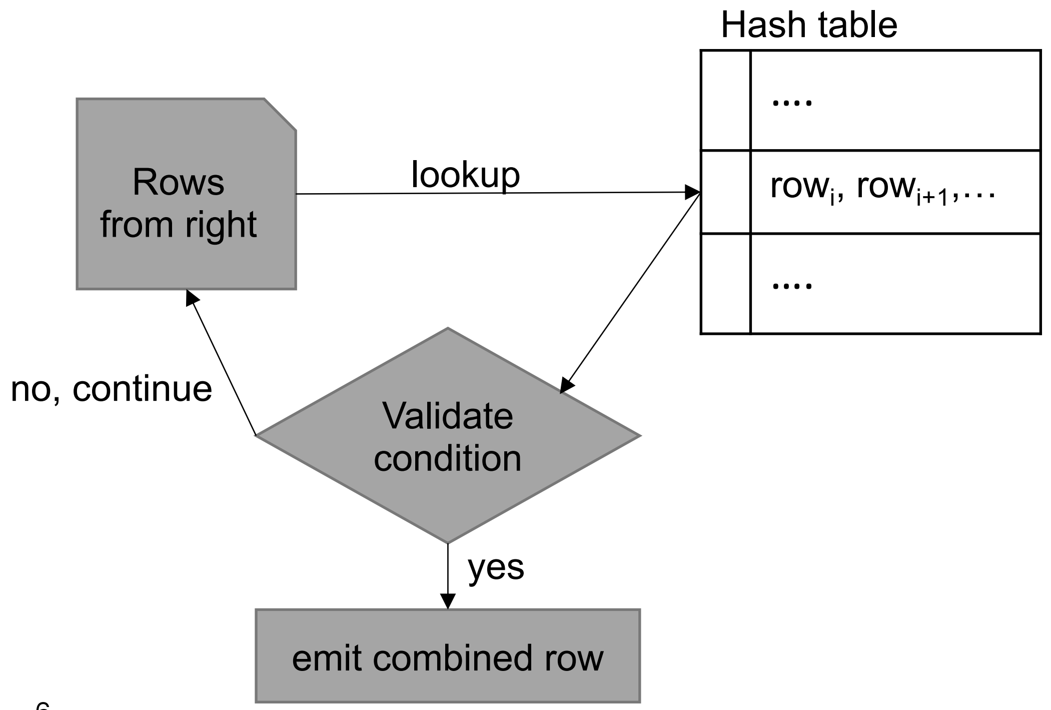
Build phase

1. For each row in the left table, calculate the hash and insert the row into the hash table.
2. Use chain hash table to avoid collisions.



Probe phase

1. For each row in the right table, calculate the hash and look it up in the hash table.
2. If no entry is found, then skip that row.
3. If an entry is found, validate the join condition for each row in the list. If the join condition is true, emit the row.



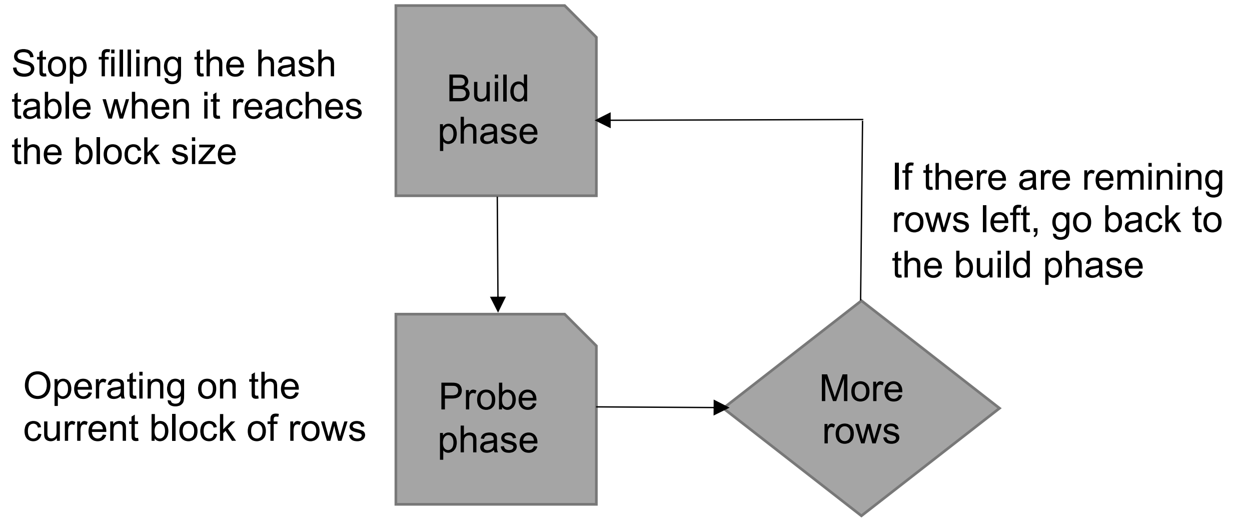
Hash Join Performance

- Five node cluster 32 GB RAM and 12 cores each
- Two tables, `t1` and `t2`
- Two join queries:
  - Matching all rows
  - Matching one-fifth of all rows
- We run each query multiple times and pick the average execution time

Query	# Rows	Nested Loop (sec)	Hash Join (sec)	Improvement
Match all	10,000	1.74110	0.02187	79x
Match one fifth	10,000	2.53793	0.01922	132x
Match all	50,000	39.58197	0.04643	852x
Match one fifth	50,000	60,97238	0.04494	1,356x
Match all	100,000	158.63308	0.07921	2,002x
Match one fifth	100,000	242,13536	0.07922	3,056x
Match all	500,000	3986.60200	0.35814	11,131x
Match one fifth	500,000	timeout	0.77291	N/A

Block Hash Join Algorithm

- Hash join limitation: one of the tables has to fit in memory
- Our solution: block hash join algorithm
- Idea: divide a large dataset into smaller chunks and work on them in isolation


Stop filling the hash table when it reaches the block size
Build phase
If there are remining rows left, go back to the build phase
Operating on the current block of rows
Probe phase
More rows

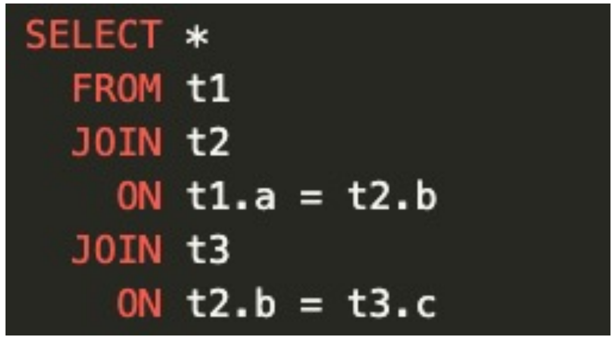
Determining a Suitable Block Size

- Calculated at the start of every query iteration
- Using **circuit breaker**[1] mechanism for setting the memory limit
  - Set up manually or by CrateDB
  - Terminates query if comes close to memory exhaustion
- CrateDB **statistical information** to estimate the size of a row in the **left table**
- The left table is split into blocks and the right table is read once for every block
- **Optimization**: size check of left and right table to switch the smaller table to the right

available memory = circuitbreaker.limit - circuitbreaker.used
max row count = available memory / estimated row size
block size = Math.min (table row count, max row count)

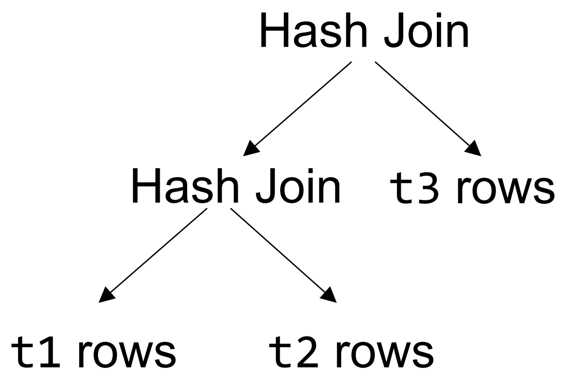
[1]https://zignar.net/2021/06/17/the-circuit-breaker-mechanism-in-cratedb/

Estimating Row Size



```
SELECT *
  FROM t1
  JOIN t2
    ON t1.a = t2.b
  JOIN t3
    ON t2.b = t3.c
```

SQL query



Logical plan

- Collect rows from `t1` and `t2`
- Perform a hash join
- Collect rows from `t3`
- Perform a hash join and produce a final rowset
- Composit of all columns from `t1`, `t2` and `t3`

For rows in `t1`, `t2` and `t3`:

- Find a corresponding shard
- Get shard size and number of rows in each shard (sys.shards table)
- Estimated row size = shard size/number of rows
- Pessimistic estimate!

Distributed Block Hash Join Algorithm

- Distribute blocks across CrateDB cluster and execute join in parallel using multiple nodes
- Our approach is inspired by grace hash join algorithm[1]
- General idea:
  - Compute hash for every row in each shard for both tables
  - Matching rows have the same hash
  - Assign each row to a node using the **modulo operator**[2]
  - After partitioning, apply the block hash join algorithm
- This method ensures that rows with matching hashes are potentially located on the same node

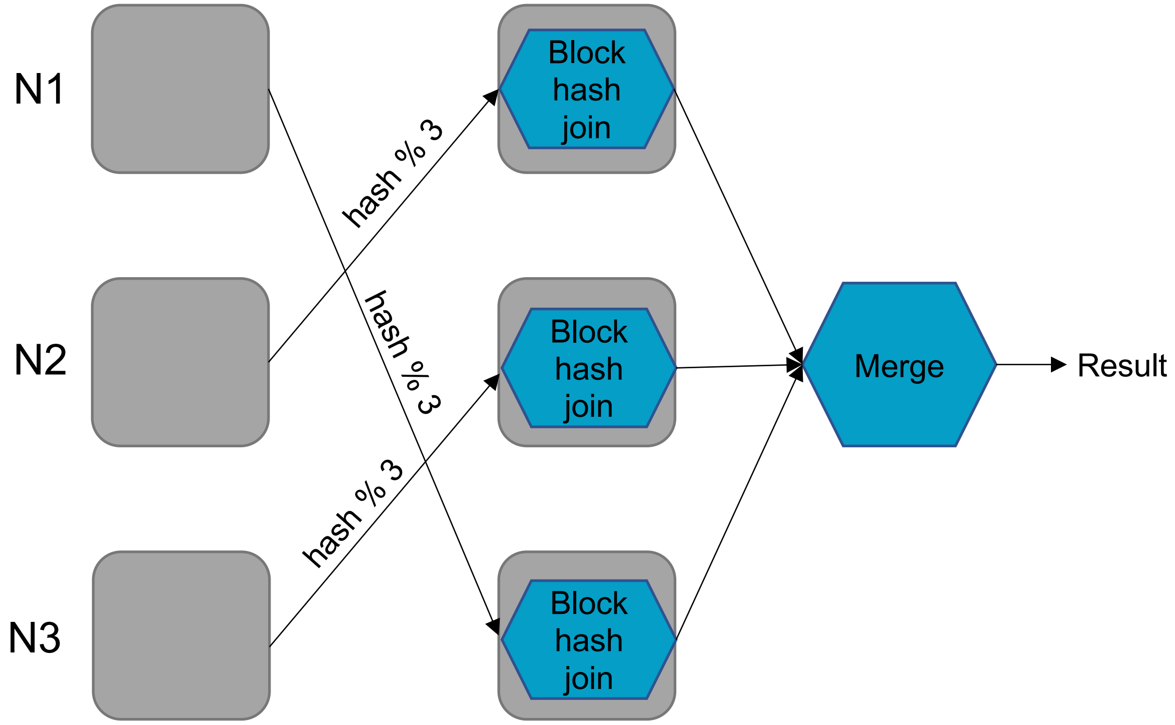
[2] node = hash value % total nodes

[1]https://en.wikipedia.org/wiki/Hash_join#Grace_hash_join

Distributed Block Hash Join: Example

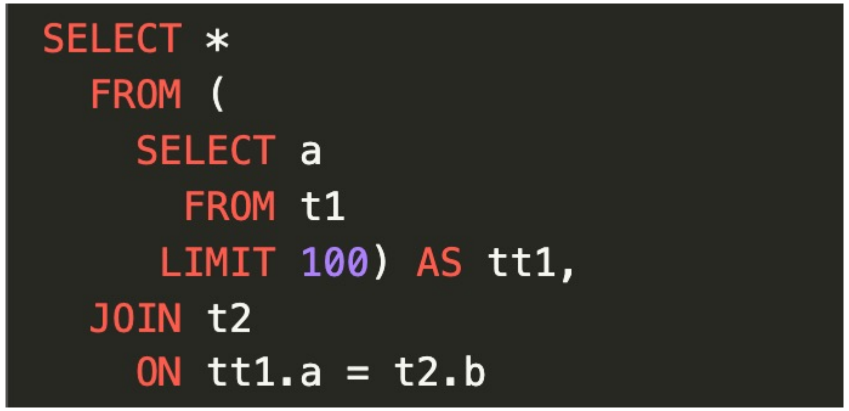
```
SELECT *
  FROM t1
  JOIN t2
    ON t1.a = t2.a + t2.b
```

- Two tables: `t1` and `t2`, distributed on three nodes
- `hash(t1.a)` and `hash(t2.a + t2.b)`
- Assign rows to a node using the modulo operator
- Assemble rows on every node (local and received)
- Run the block hash join algorithm
- Merge results



Limitations

- Use case: joining subselect with `LIMIT` or `OFFSET` clause



```
SELECT *
  FROM (
    SELECT a
      FROM t1
     LIMIT 100) AS tt1,
  JOIN t2
    ON tt1.a = t2.b
```

- Single-node block hash join performs better than distributed version
- With distributed block hash join algorithm node is only processing a subset of data
- Before distribution, the single node must fetch 100 rows from `t1`
- Out of scope for now, a single-node version of the block hash join algorithm is used

Final Benchmarks

- Two tables `t1` and `t2` with five primary shards
- CrateDB cluster consisting of five nodes
- Tested two queries with three algorithms:
  - Original nested loop algorithm
  - Single node block hash join algorithm
  - Distributed block hash join algorithm

Query	# Rows	Nested Loop (sec)	Single Node (sec)	Distributed (sec)	Improvement
Match all	10,000	1.7411	0.01587	0.01447	120x
Match one fifth	10,000	2.5379	0.01422	0.01160	218x
Match all	50,000	39.5819	0.04643	0.04155	952x
Match one fifth	50,000	60,9723	0.04194	0.03068	1,987x
Match all	100,000	158.6330	0.08921	0.06773	2,342x
Match one fifth	100,000	242,1353	0.06922	0.05169	4,683x
Match all	500,000	3986.6020	0.35814	0.17324	23,011x
Match one fifth	500,000	timeout	0.31457	0.24539	N/A

Conclusions

- Implementation of the block hash join algorithm and its modification to run in parallel on multiple nodes
- The original nested loop algorithm performs **reasonably well** for tables with up to 500k rows
- **For tables with more than 50k rows**, the distributed block hash join algorithm is significantly faster than the single node block hash join algorithm
- Distributed block hash join algorithm makes join operations **up to 23k times faster** than nested loop algorithm
- To find more, please check our GitHub repository!

https://github.com/crate/crate

THANK YOU!