SIGMOD Programming Contest 2018
Team Dataurus

T. Belmpas, D. Chalatsis, O. Gkini, G. Kamaras, G. Michas, C. Svingos
{sdi1400119, sdi1400219, sdi1400036, sdi1400058, sdi1400109, csvingos}@di.uoa.gr

Εθνικό και Καποδιστριακό ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
National and Kapodistrian UNIVERSITY OF ATHENS

Problem

Task: The task is to **evaluate batches of join queries** on a set of pre-defined relations. Each join query specifies a set of relations, (equality) join predicates, and selections (aggregations). The maximum number of relations that a query joins is four. The challenge is to execute the queries as fast as possible without (much) prior indexing. For this reason we only have 1 second for preprocessing.

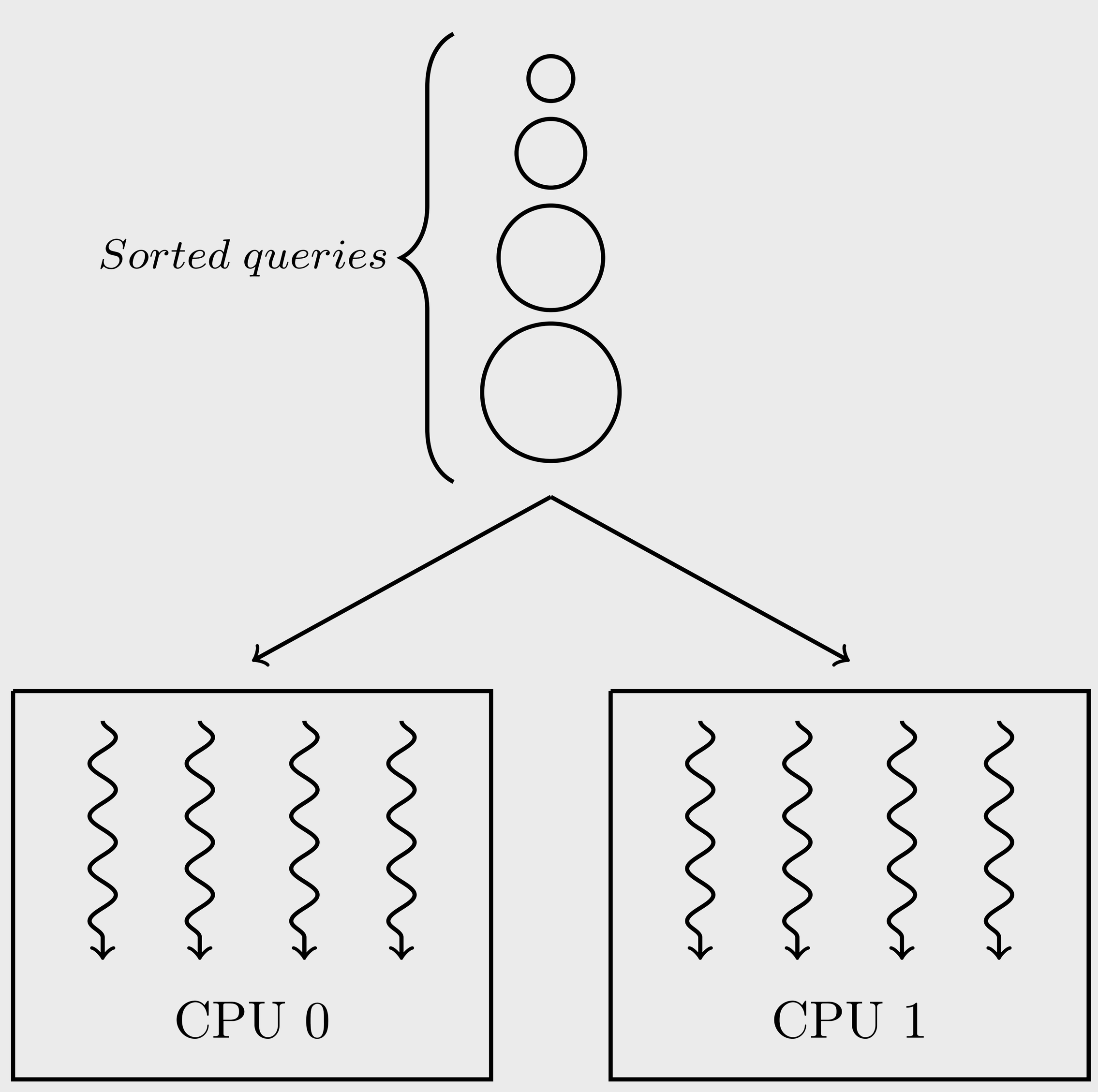
Hardware: 2x Intel Xeon E5-2660 v2 CPU (2.20 GHz, 3.00 GHz turbo) - 20 Cores / 40 Hyperthreads - 256 GB RAM

1. Preprocessing

- Data is stored in **column-store** fashion for cache efficiency.
- Parse the input data in parallel and collect **statistics** per column, such as
  - minimum element value
  - maximum element value
  - number of tuples
  - number of distinct elements
  - spread of values

3. Query Scheduling

- A **global queue** is used to distribute the queries into two NUMA regions.
- **Sort the queries** of a batch in descending order (heavy first) based on the estimations (cost) of the optimizer and push to the queue.
- As soon as a cpu finishes the execution of a query, the next one in the queue is assigned to it immediately.
- **Clone** the dataset on both NUMA regions for better memory bandwidth utilization.



5. Join Operator

- Use **Parallel Radix Hash Join**[a] to compute the join result
  - The two input relations R and S are divided into **partitions**.
    - Each thread receives a chunk of data and computes a **histogram** over the input data, so that the exact output size is known for each thread and each partition.
    - Each thread pre-computes the **exclusive location** where it writes its output. Finally, all threads perform their partitioning.
  - Each thread takes a set of R and S partitions. A separate **hash table** is created for each R partition (assuming R is the smaller relation). Each of these hash tables fit into the CPU cache.
  - During the final probe phase, S partitions are scanned and the respective hash table is **probed** for matching tuples.

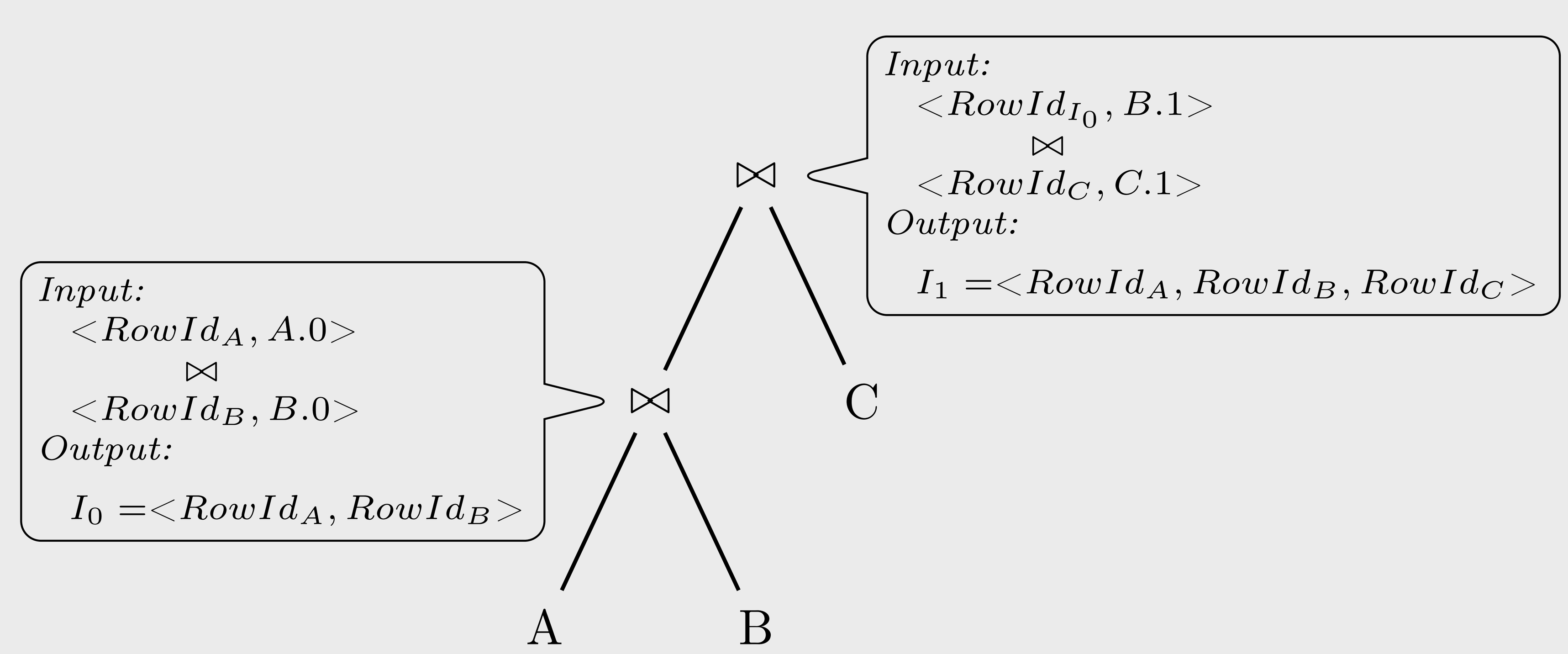
[a] source: ETHZ Systems Group's work on Parallel & Distributed Joins at https://www.systems.ethz.ch/node/334

2. Query Optimization

- Assumptions
  - All attribute values are **uniformly distributed**.
  - The values of all attributes are drawn **independently**.
- Join re-ordering
  - Join tree enumeration using a bottom up **dynamic algorithm** approach.
  - Consider only **left deep** binary join trees.
  - **Cardinality estimation** for both sides of the join predicate with statistical formulas - taking into consideration foreign key relationships between the two columns.
  - Use as cost function the **sum** of sizes of intermediate results.

4. Plan Execution

- Intermediate results are materialized into RAM to evaluate next-level operations.
- **Filters** are performed first in the tree hierarchy to eliminate as many tuples as possible.
- Each column (suppose from relations A and B) on both sides of the join predicate is represented as a vector of <**RowId, Value**> pairs. These vectors are then passed as input to the join operator.
- The join operation results in a vector of pairs $<RowId_A, RowId_B>$, where A and B are the joined relations.
- This result is then transformed into a vector of **values** that correspond to the **row ids** of all relations whose join predicates have already been calculated.
- To calculate the final aggregate functions we scan the vector of row ids and probe the initial tables.
- Let's consider the query $A.0 = B.0$ & $B.1 = C.1$



6. Caching of indexes

- The Radix-Join algorithm requires a preprocessing phase that divides a relation into partitions (build phase). This phase is the most time consuming in comparison with the other radix hash join phases.
- Every time we apply the build phase to a column of an initial table, we **store** the result for subsequent use.

Conclusion

- Memory's bandwidth seems to be the bottleneck for in memory database systems.
- Occam's Razor: The simplest solution is always the best.