ACM SIGMOD 2015 PROGRAMMING CONTEST

– CSTRINGS –

Lambros Petrou – ***University of Oxford (MSc)***

George Koumettou – ***Royal Holloway University of London (MSc)***

Marios Mintzis – ***University College London (MSc)***

►*All graduated from* ***University of Cyprus*** *(2014) – BSc. Computer Science*◄

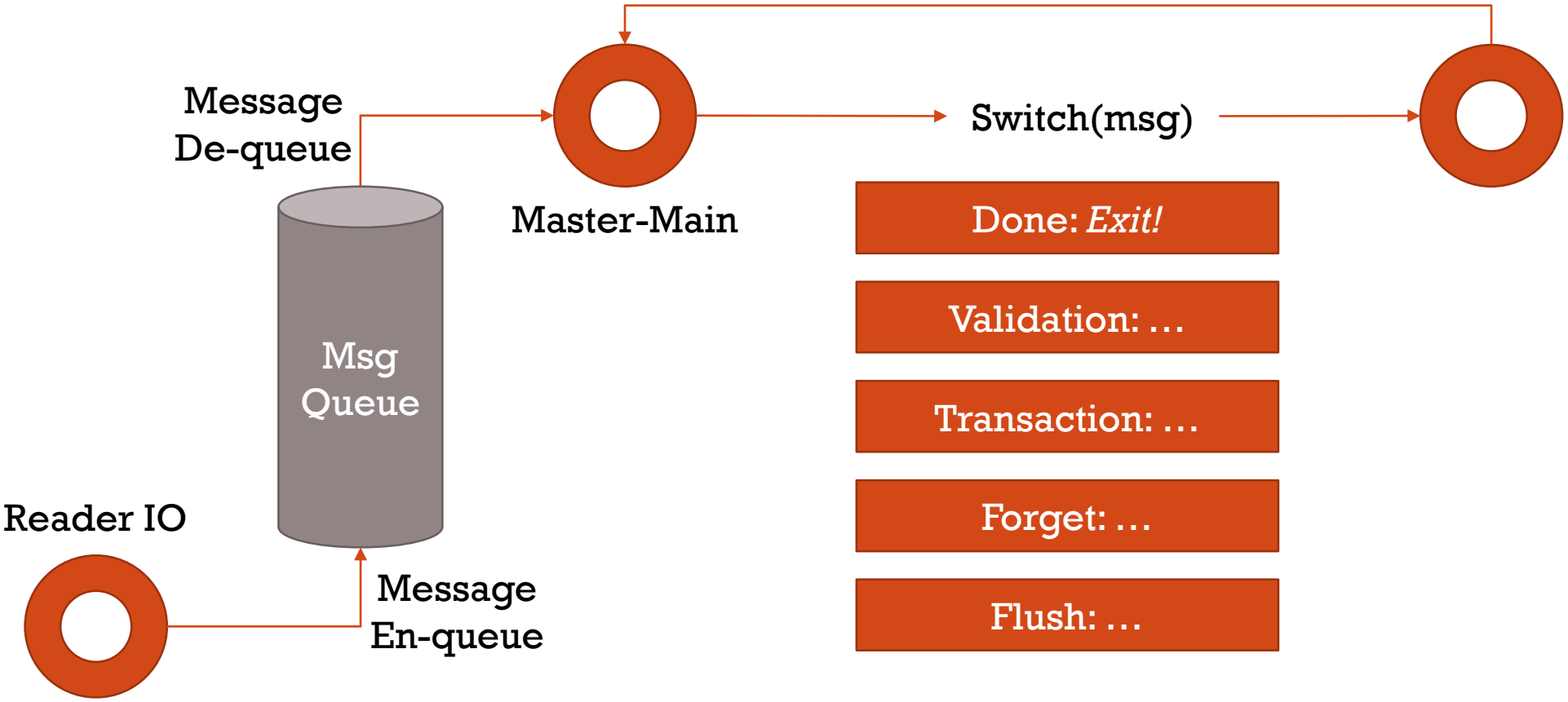
CHALLENGE

- Implement a validation system that processes validation requests against a continuously modified relational database.
- Validation only over a specific database instance (range of transactions).
- Each validation request contains a set of Conjunctive Queries extended with mathematical and logical binary operators (≥≤ < > = ≠).
- A validation is considered to be *conflicting* if and only if at least one of the CQs evaluates to true (the result set of the Conjunctive Query is not empty).

DATA ANALYSIS & OBSERVATIONS

- Cardinality of relations differs greatly
- Transactions that affected a few tuples and transactions that affected thousands
- Conjunctive Queries related
  - Very small percentage (< 0.1%) were tautology (assuming the relation was not empty)
  - A lot of queries were invalid (no DB instance satisfies it) – **requires pruning**
  - More than 90% of the queries had at least 1 (ONE) predicate with equality operator (=)
  - A lot of duplication among the predicates of the queries
- A very small minority of the validation requests were actually conflicting
- Almost half of the columns among all relations were used in the validations but most of the attention was around the low-index columns (0-primary, 1, 2, 3, 4…)

WORKFLOW


Message
De-queue
Master-Main
Switch(msg)
Done: *Exit!*
Validation: …
Transaction: …
Forget: …
Flush: …
Msg
Queue
Reader IO
Message
En-queue

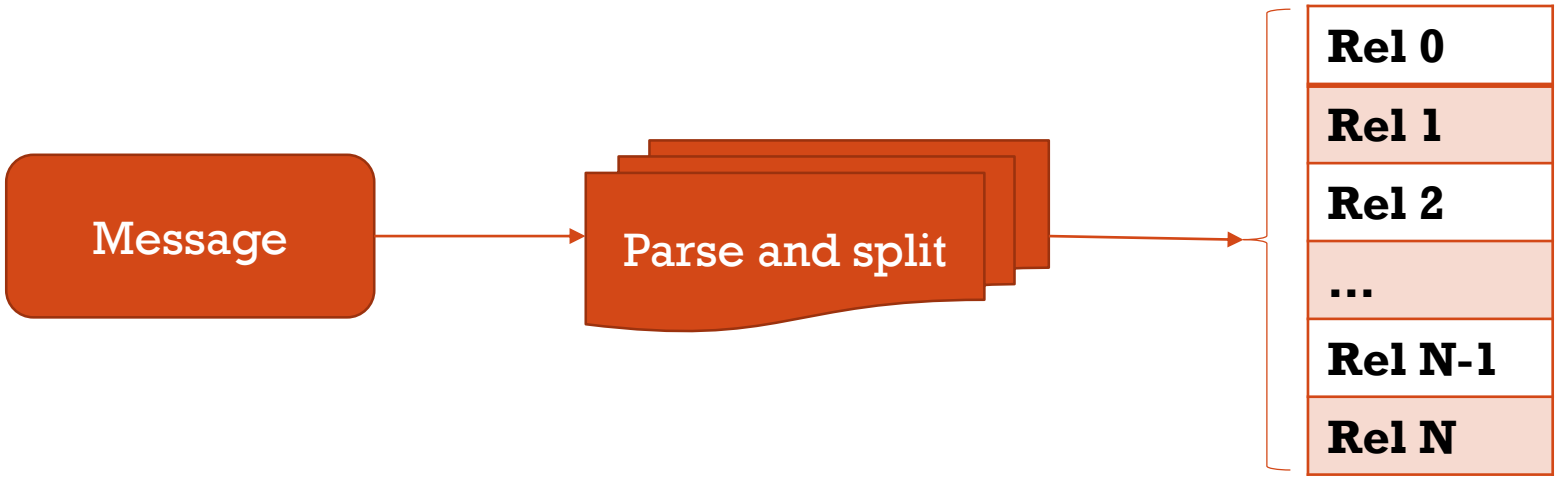
MESSAGE = VALIDATION

- Just insert the validation message into the Pending Validations queue without any processing at this point

Pending Validations

MESSAGE = TRANSACTION

- Partially parse the transaction message up to the point to recognize the relations that it modifies.
- Copy the respective part of the message and append it in Pending Transactions for its relation accordingly



Pending Transactions
(1 queue per relation)

MESSAGE = FORGET / FLUSH

- FLUSH
  - Master thread does some initializations and starts the thread-pool
  - Main validation processing by thread-pool (*described later*)
  - Print out the results up to the validation requested

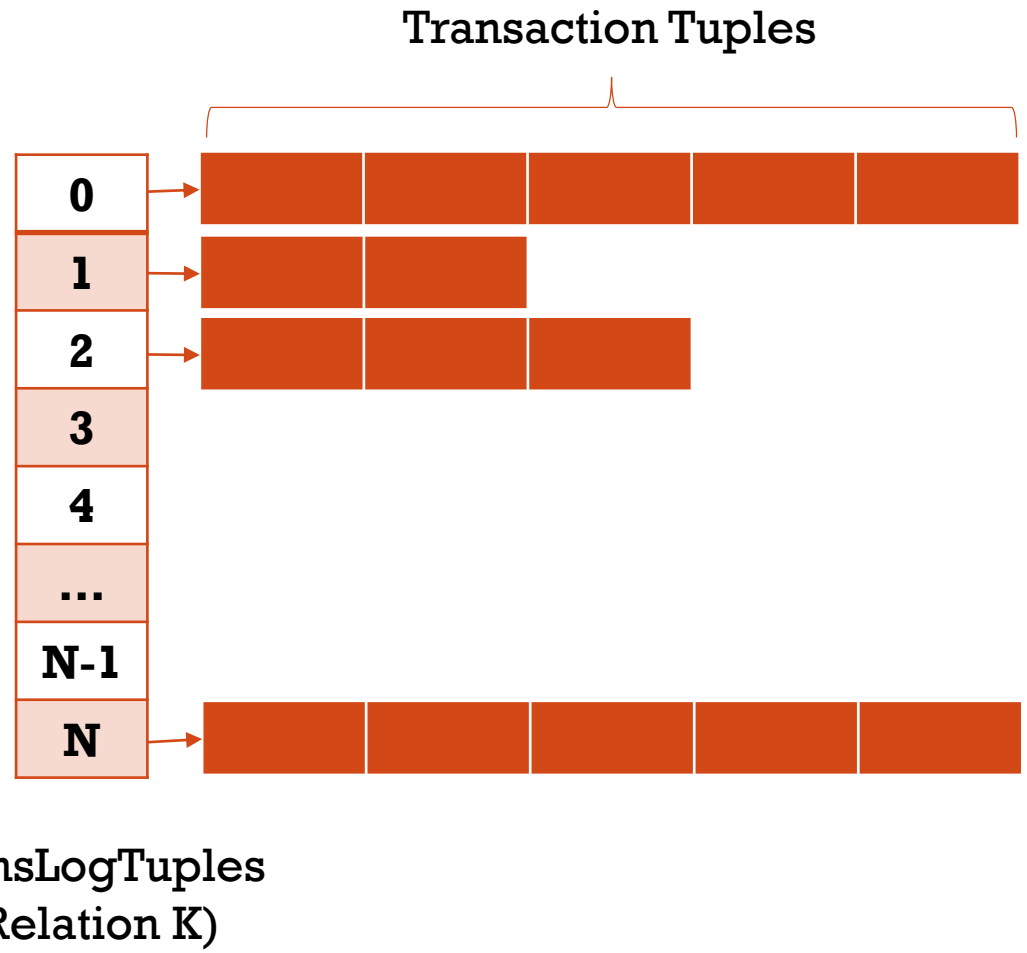
- FORGET
  - Master thread does some initializations and starts the thread-pool
  - Main validation processing by thread-pool (*described later*)
  - Master thread does some cleanup if necessary

MAIN PROCESSING

- The main processing (transaction processing + validation evaluation) is handled by our custom thread-pool (as many threads as cores)
- The execution has 5 steps, which are going to be explained in the rest of the presentation

MAIN PROCESSING – STEP 1

- Concurrency on Relations
- Each relation is being processed by a single thread
- For each transaction in the Pending Transactions queue we create the tuples it modifies and associate them with their transaction ID
- Tuples are created for inserts AND deletes since for us the treatment is the same
- Tuples are stored ordered by the transaction ID inside each relation

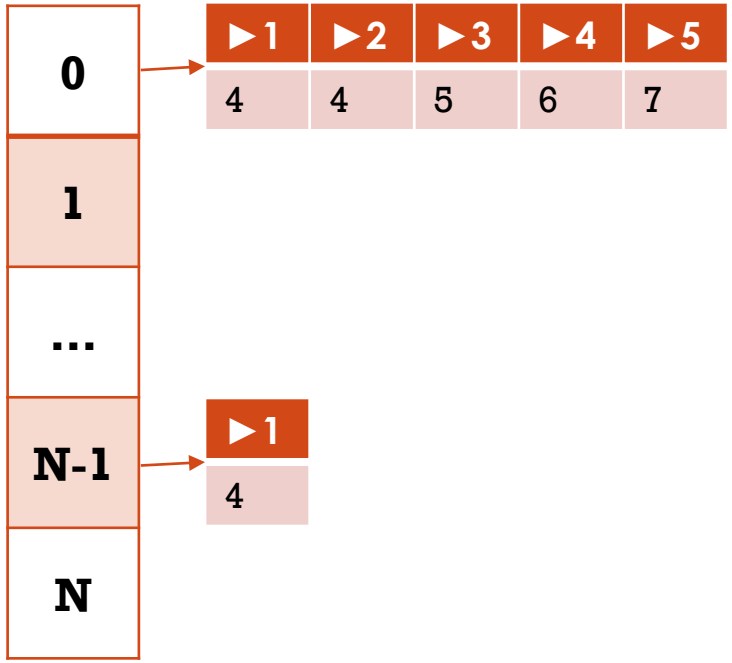


MAIN PROCESSING – STEP 2

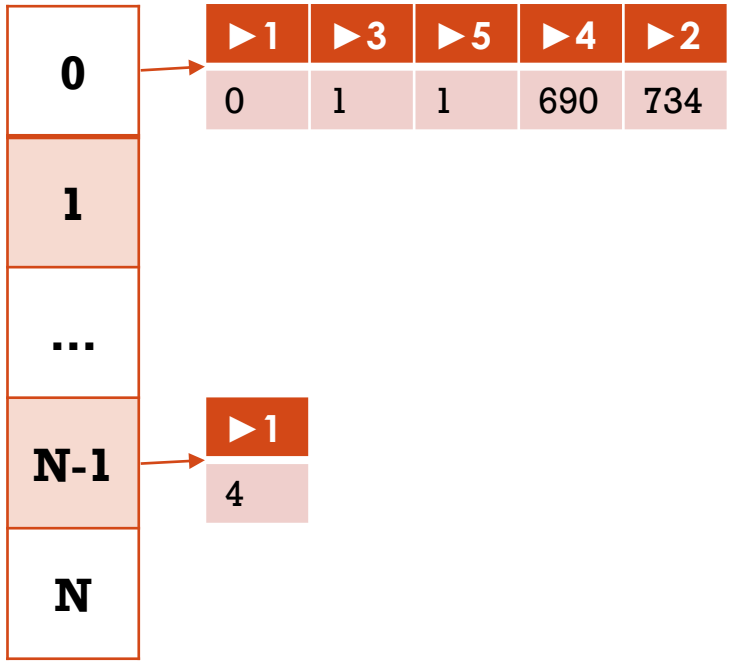
- Concurrency on Relations & Columns
- Process concurrently all the columns among all relations (1 thread takes 1 column)
- Update the Column-Index for each column with the tuples created by the transactions in the current batch (from step 1)
- Our Column-Index groups the tuples by transaction ID and for each transaction the tuples are sorted by their value in the corresponding column
- We use this property heavily during the evaluation of the validation requests
  - Sorted by transaction ID – each validation specifies a range of transactions
  - Sorted by value – since 90% of the queries had equality (=) operator, we can get all the tuples with the queried value with a single binary search

MAIN PROCESSING – STEP 2 (2)

- The same tuple can be in different position inside its transaction at different columns of the same relation since they are sorted by their value in each specific column



Column-Index
Column 1



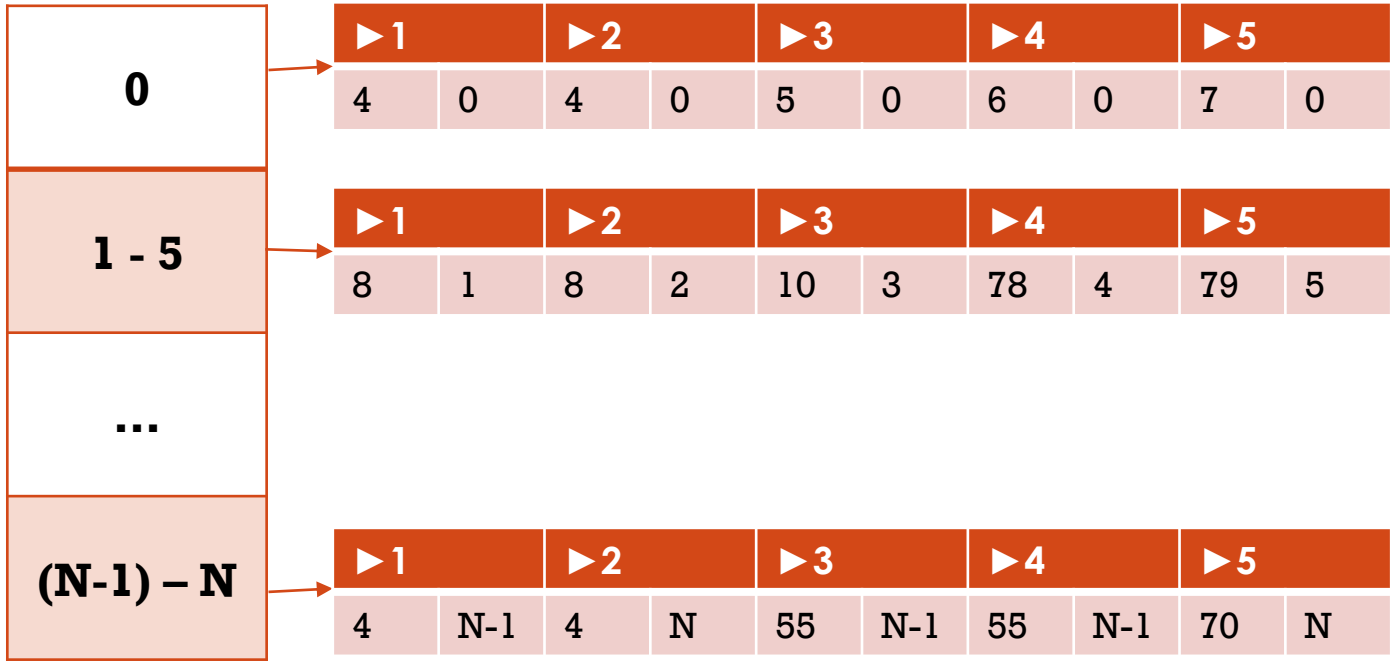
Column-Index
Column 2

MAIN PROCESSING – STEP 2 (3)

- An optimization to mitigate the great difference among the number of tuples each transaction had is that instead of having one entry for each transaction in our index we now have buckets of transactions
- As before the tuples in each bucket are sorted by value and in case of ties we sort by the transaction ID
- The Column-Index had two rules as to when to create a new bucket
  - ***Tuples threshold***: create a new bucket if the tuples inside the current one exceed our threshold. This helps to avoid having too many tuples which will make sorting very slow.
  - ***Transactions threshold***: create a new bucket if the number of different transactions in the current one are more than our threshold. This will ensure that we will not have to skip a lot of tuples outside the requested range
    - Remember that we do a binary search in each bucket to get the tuples with the proper value and then we process only those that also fall within the transaction range
    - Easy to skip those not needed since they are sorted by transaction ID two

MAIN PROCESSING – STEP 2 (4)

- Each bucket sorts its tuples by value on that column and then by transaction ID
- You can have buckets with tuples from 1 transaction (TPL_THRES) or with more (TRANS_THRES)



Column-Index
Column 1

MAIN PROCESSING – STEP 3

- Concurrency on Pending Validations
- We process all pending validation requests concurrently
- For each request we parse its set of Conjunctive queries
  - Check for validity & de-duplication (query: col-X == 5 AND col-X > 5, ***invalid***)
  - If valid, sort its predicates - ***!important***
    - Equality (=) operators first
    - Lowest columns first (especially column 0 which is primary key => less results in binary search)
- For each column *K* we keep an inverted index that contains all the queries that have as 1st predicate an equality operation using the column *K*
  - Queries are sorted by value inside each column

MAIN PROCESSING – STEP 4

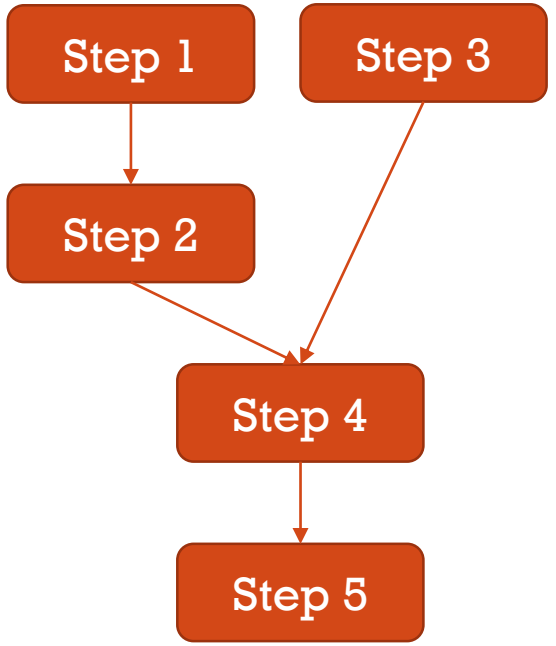
- Concurrency on Relations & Columns
- This is the main step for the evaluation of the pending validation requests
- All the columns among all relations are being processed concurrently, and for each one we evaluate all its queries inside the Query Inverted-Index (step 3)
- Major benefit of this step is that we have good cache-usage since the same Column-Index (step 2) instance is being used to evaluate thousands of queries
  - Significant speedup in serial execution (170% speedup) compared to evaluating each query in the order of its validation request (different column each time)
  - Not so much speedup in concurrent execution (mainly due to 8 threads working at the same time in different columns)
    ►Optimization: All threads should try to execute queries from the same column

MAIN PROCESSING – STEP 5

- Concurrency on Pending Validations
- This is the last step of the evaluation
- Evaluate the few queries remaining in case their validation has not been already marked as conflicting.
- Again, we use the Column-Index and a single binary search to get the tuples we want inside the transaction range
- The difference now is that the operators we have are ≥≤ < > ≠ therefore we have to process a range of tuple-values (*std::lower_bound* & *std::upper_bound*)

MAIN PROCESSING – CONCURRENCY

- We avoided locking (apart from step 3) using atomics
- Initially each step of the processing was followed by a barrier
- Our metrics showed that a single relation was accountable for the 95% of the execution time for steps 1 and 2 therefore we had to introduce some overlap
- Additionally, step 2 can only be done on the columns that belong to relations that have already finished step 1.
- Therefore in our final execution order the following are processed concurrently in this order:
  - Step 1 ►Step 3 ►Step 2 (we hoped that the bottleneck of step 1 would finish by the time step 3 was finished)
- This way we managed to also hide part of step 3 execution time – which was significant



Initial Execution
1 ►2 ►3 ►4 ► 5

Better Execution
1 ►2, 3 ►4, 5

Final Execution
1, 3, 2 ►4, 5

(► = barrier)

WHAT COULD BE IMPROVED

- We noticed that more overlap could be achieved in main execution with step 4 too. This would give us a lot of speedup since often most of the threads were stalled at the barrier before step 4 waiting for a single thread to finish step 2 (Relation 3)
- Compression in Column-Index since the tuples are sorted by value and there were a lot of duplicated values in many columns
  (Relation 3 – column 4 had only 3 unique values but thousands of tuples)
- We hardly used any SIMD code (apart from some auto-vectorizable loops)
- Update Column-Index on request – incrementally.
  We implemented it BUT
  - before Large dataset was released
  - before we switched to Column-Index with buckets

  So we do not know if it would benefit our final solution in the bigger datasets

TOOLS & LIBRARIES

- Intel® VTune™ Amplifier 2015
  - Concurrency analysis
  - Advanced Hotspots
- Concurrent queues (moodycamel + custom-made)
- B-tree (https://code.google.com/p/cpp-btree/)
- Agner subroutine library (http://agner.org/optimize/#asmlib)
- Extensive usage of C++11 templates & STL

THANK YOU – ABORT();