# Team: *CStrings*


**Lambros Petrou**
lambrospetrou@gmail.com
***University of Oxford***

**George Koumettou**
gkoume01@gmail.com
***Royal Holloway University of London***

**Marios Mintzis**
mariosmintzis@hotmail.com
***University College London***

►All members graduated from ***University of Cyprus*** in 2014 – BSc. Computer Science ◄


## ACM SIGMOD 2015 Programming Contest - The challenge

Implement a validation system that processes validation requests against a continuously modified relational database, but only over a specific database instance (range of transactions). Each validation request contains a set of Boolean Conjunctive Queries extended with mathematical and logical binary operators (≠ ≤ ≥ < > =).

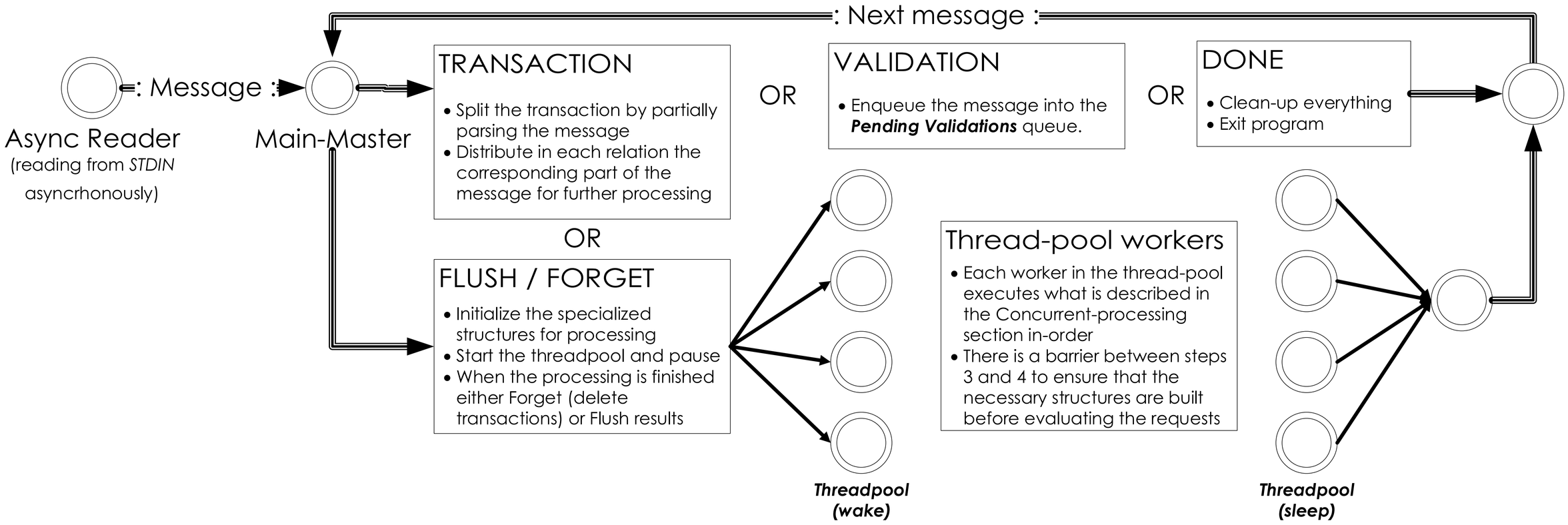
***Simplified Example***

**Schema: R1**(X, Y, Z), **R2**(K, L)
**Validation Request: [**Transactions: 17 – 2000**] { Q1=** *R1*: (X = 5) ^ (Y ≥ 3746), **Q2=** *R2*: (K ≠ 1000) **}**

This validation should only be checked against transactions 17 to 2000 and it is evaluated as ***CONFLICT*** if and only if either Q1 or Q2 is true. Q1 is true if and only if there exists a tuple in relation *R1* that has *X-value* 5 and *Y-value* larger than 3746, whereas Q2 is true if and only if there exists a tuple in relation *R2* with *K-value* different than 1000.
***Note*** that only tuples modified (inserted or deleted) in the specified range of transactions (17 to 2000) should be considered.

## Workflow

Just a brief overview of the system's work-flow throughout the whole execution.




## Concurrent processing

- ***Step 1 – Concurrency on Relations***
  process all the transactions for each relation in order and create the inserted/deleted tuples as needed
- ***Step 2 – Concurrency on Relations & Columns***
  update our custom Column Index for each column
- ***Step 3 – Concurrency on Validations pending***
  parse the pending validation messages and distribute each query (that contains == condition) of each validation to the corresponding relation (heavy query pre-processing and pruning in this stage too to eliminate invalid queries and to sort the predicates inside each query such that the equality (==) conditions are first)
- ***Step 4 – Concurrency on Relations & Columns***
  evaluate all the equality queries of each column (step 3) using the column index (step 2)
- ***Step 5 – Concurrency on Validations pending***
  evaluate the remaining queries in queue (without equality operators) if their validation request has not been already marked as *CONFLICT*

## Key data-structures

- ***Inverted Index for validation queries (step 3)***

  Each relation has an index for each of its columns to hold all the queries among all the pending validations that have as their 1st predicate an equality operator of that specific column. The queries are inserted in this index ***AFTER*** being pre-processed & pruned.

  This allowed us to evaluate all the queries of a specific column in sequence, thus leading to a better cache usage (significant speedup if single-thread) since only a single column index was used for thousands of queries.

- ***Column-based Index (step 2)***

  We designed a custom **Column-based Index** for the transactions and the tuples they deleted/inserted. Each column had buckets of transactions sorted by the transaction Ids they contained in order to allow retrieval of only the transactions specified in each validation range.
  Additionally, each bucket was filled until either:
  a) the **number of transactions** contained exceeded our threshold
  b) the **number of tuples** contained exceeded our threshold
  As a result we always had roughly balanced buckets with just enough tuples to make processing faster. The tuples inside each bucket were sorted by their value in that specific column, therefore during the evaluation (*step 4*) with just a single binary search we could get all the tuples that had the value we want.

## Query pre-processing

- ***Sort and unique all the predicates***
- **Check validity of query**
  (col-X == 5 ***AND*** col-X > 5) => ***invalid!!!***
- **Make ranges stricter or remove them entirely**
  (col-X == 5 ***AND*** col-X >= 5) => (col-X == 5)
- ***Make sure that the predicates with equality (==) operators are the first to be processed***

## Other Tricks

- **Custom thread-pool** and concurrent processing over a sequence of elements with **atomics** instead of locks
- Auto-vectorized loops
- **Branch annotations** with compiler instrinsics
- **Cache-line fitting** of data and **aligned allocator** for std::vector in some cases