**Task Overview**
- Given processed transactions on relations
- Concurrent validation requests on the transactions
  - e.g. \((\text{Rel. 1: } Q_1 \land Q_2) \lor (\text{Rel. 2: } Q_3 \land Q_4)\)
  - A Query consists of
    - Column index
    - Operator \((=, >, >=, <, <=, \neg=)\)
    - Value (64-bit integer)
- Evaluate the validation queries ASAP

**Our Approach**

**Dataset Feature Analysis** + **Efficient Implementation**

**Dataset Features**
- Validation results:
  - 00010000000000000000000000000000
  - 00000100100010010000000000
  - 00000000000000000000000000000000

- Operator type:
  - Most validations fail
  - Most operators are equal

- Transaction size:
  - \(T_1: \text{Insert rows: 4} \lor \text{Delete rows: 4}\)
  - \(T_2: \text{Insert rows: 1} \lor \text{Delete rows: 2}\)
  - \(T_3: \text{Delete rows: 1} \lor \text{Insert rows: 2380}\)
  - \(T_4: \text{Insert rows: 1} \lor \text{Delete rows: 3}\)

- Dist. of column values:
  - \(C_1: \text{Column index}
  - \(C_2: \text{Large-sized transactions}
  - \(C_3: \text{Small-sized transactions}

**Implementation Design Overview**
- Reader reads input messages and writes to the buffer.
- Transactors store transactions and construct bloom filters.
- Pre-validators separate and reorder validation requests by conjunctions, and push the result into the queue.
- Validators validate the requests from the queue.

**Validation Requests**
- Independently process each conjunction in validation requests
- Optimize according to dataset features:
  - Reorder queries in a conjunction
  - Check if the value of a condition is within minimum and maximum
  - Prune candidate rows using bloom filters for equal queries
  - Survived candidates are validated naively

**Reordering Queries**
- Original: \(C_1 < V_1) \land (C_2 = V_2) \land (C_3 = V_3) \land (C_4 > V_4)\)
- Reordered: \(C_2 = V_2 \land (C_1 < V_1) \land (C_3 = V_3) \land (C_4 = V_4)\)

**Pruning with Bloom Filters**
- Only utilize bloom filters for equal queries
- A row is non-conflicting if any corresponding bits of its bloom filter are 0.
  - In above example, it is guaranteed that \(R_x\) doesn’t have 234 in the \(C_1\).
- The bloom filters are stored column-wise to easily compute bitwise AND of the columns.
- SIMD bitwise AND can process one column of bloom filters of 256 rows at once.

**Performance Timeline**

- SIMD Bitwise AND
  - 256-bit AVX SIMD Bitwise AND (VANDPS)
  - Column store
  - Input buffer
  - Optimizations