2015 SIGMOD Programming Contest

Team _CUT_HERE

서울대학교 컴퓨터공학부
Seoul National University
Dept. of Computer Science and Engineering

KAIST School of Computing

Eunjin Song, Seoung Gook Sohn, Wonha Ryu
eunjin@csap.snu.ac.kr sgsohn@dbs.snu.ac.kr wonha.ryu@gmail.com

Task Overview

- Given processed transactions on relations
- Concurrent validation requests on the transactions
  - e.g. (Rel. 1: $Q_1 \wedge Q_2 \wedge Q_3$) $\vee$ (Rel. 2: $Q_4 \wedge Q_5$)
  - A Query consists of
    - Column index
    - Operator (==, >, >=, <, <=, !=)
    - Value (64-bit integer)
- Evaluate the validation queries ASAP

Our Approach

Dataset Feature Analysis + **Efficient Implementation**

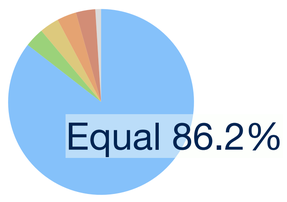
Dataset Features

Validation results:

```
00010000000000000000000000
00000100100010010000000000
00000000000000100000000000
00000000001000000000000...
```

Most validations fail

Operator type:



Most operators are equal

Transaction size:

...
T_n Insert rows: 4
Delete rows: 4
T_{n+1} Insert rows: 1
Delete rows: 2
T_{n+2} Insert rows: 1
Delete rows: 1
T_{n+3} Insert rows: 2380
Delete rows: 1643
T_{n+4} Insert rows: 1
Delete rows: 3
...

Substantial small-sized transactions and exceptional large-sized transactions

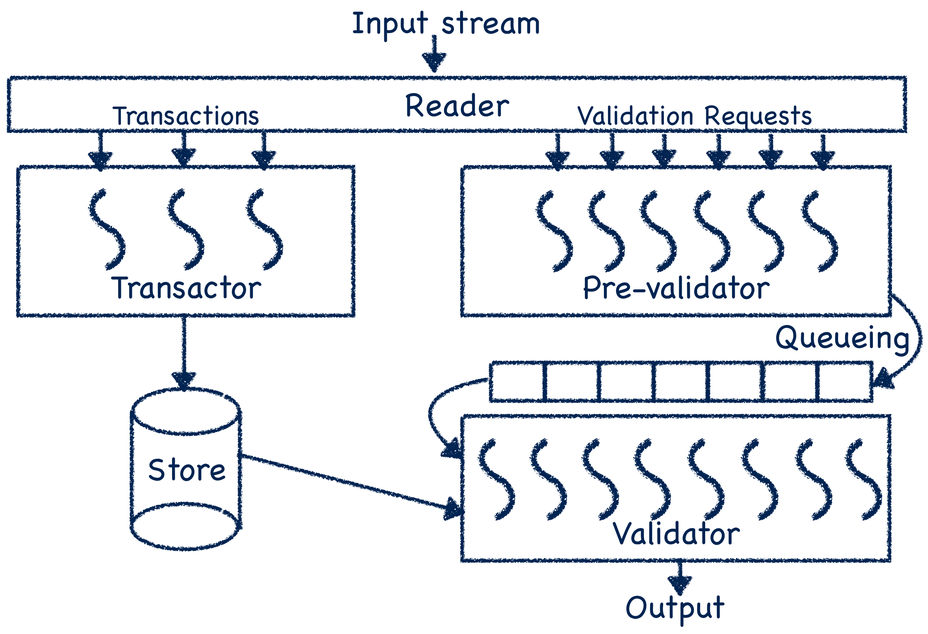
Dist. of column values:

C_1 (dense)	C_2 (sparse)
3074	8896128387
3077	6811549715
3075	7782881939
3077	5819204839

$C_1 \stackrel{?}{=} 3076$ $C_2 \stackrel{?}{=} 5739273029$

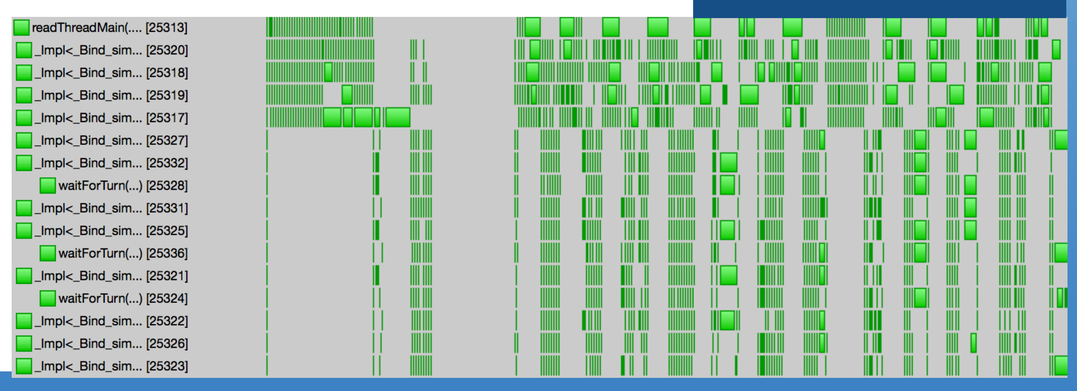
C_2 has lower prob. of conflicting with equal

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.

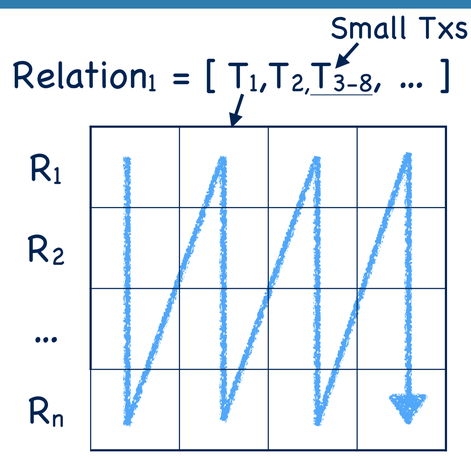
Profiling Result



Transaction

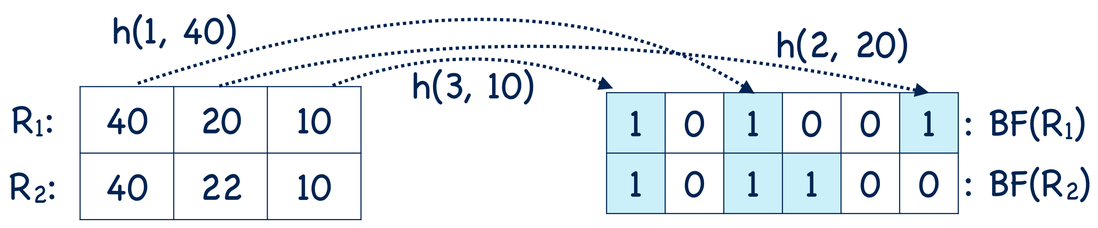
Transactions are stored as tables for each relation.

- Each relation object holds its own list of transactions
- Store in column-store fashion for cache efficiency
- Group small transactions into one transaction to reduce overhead
- Construct bloom filters for each row



Simplified Bloom Filter

- Each value is viewed as a pair of column index and value
  - [40, 20, 10] ⇒ equivalent set: {(1, 40), (2, 20), (3, 10)}
- Only a single hash function is used for performance issues
- Also stored in column-store fashion per transaction



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

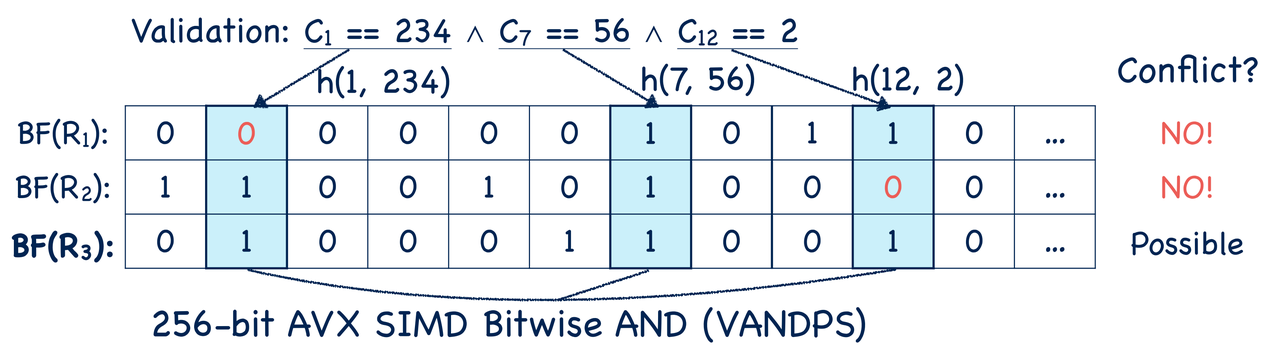
Original: $(C_1 < V_1) \wedge (C_2 == V_2) \wedge (C_3 == V_3) \wedge (C_4 >= V_4)$

Reordered: $\underset{1.}{(C_2 == V_2)} \wedge \underset{2.}{(C_3 == V_3)} \wedge \underset{3.}{(C_1 < V_1)} \wedge (C_4 >= V_4)$

Reorder queries in the following order to reject ASAP

1. Query with == operator having the widest range of values
2. Queries with == operators
3. Queries with other operators

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_1 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