**Task**
- Compute a set of validations, each producing a boolean
- A validation is a disjunction of queries and evaluated over a range of transactions
- A query is a conjunction of predicates over table records
- A predicate is a comparison operation (eq, leq, <, <=, >, >=) of a table column value against a constant

**Mathematical Formulation**

\[
\text{validation}(d) = \bigvee_{r=\text{from}} \bigwedge_{q=0}^{\text{to}} \bigwedge_{p=0}^\text{p} \text{operation of predicate } v.q.p \text{ column of predicate } v.q.p
\]

**Strategy**
- Analyze data statistics
- Come up with a reasonable algorithm
- Parallelize to multiple threads
- Win!

**Statistics**
- Tables and columns are used highly non-uniformly!
- Selectivity of equality 1e-6 for most columns
- Some columns have many duplicates and low selectivity, can wreck hash table access
- Probability of query match: ~1e-3
- Probability of validation match: ~4%
- Since tiny number of queries match, most work is spent in evaluating non-matching predicates

**Data Structures and Memory Allocation**
- Primary key is stored in a B-tree (STX)
- Log-oriented storage for transaction records. New records are only appended and old records are purged lazily
- Link fields for index traversal are embedded into records

**Query Plans**
To accommodate highly varying predicate selectivity, two query execution strategies:
- Index lookups for high selectivity predicates
- Full scans for low selectivity predicates
- Pretty much like in a DBMS
- Plan selection depends on column selectivity (number of unique values)

**Index Lookups**
- Hash table w/o collision resolution
- All historic values with the same hash value are linked
- No distinction between collisions and older values
- Dynamically-expanding hash table
- Validation is required

**Scans**
- Typically, used for queries w/o equality
- All transaction records are scanned backwards starting from the most recent
- Non-equality scans are accelerated with “aggregate records” for every N transaction, storing min and max for each column

**Three-Stage Multithreading**
- Stage 1 (single thread): input parsing and distribution between tables
- Stage 2 (thread per table): building transaction history and primary key, initial stage of query evaluation
- Stage 3 (many threads): final evaluation over immutable input: following index links, scanning record ranges
- Unfortunately, due to high skew, degree of parallelism was low (~3)

**Final Remarks**
- C++ 11, ~2200 loc, excluding third-party code
- Final performance was limited by cache misses (mostly by index lookups)
- Technical implementation details are as important as algorithms

**Third-Party Code Used**
- Boost intrusive lists (work scheduler for multiple threads)