Solution by Team Embryo (Third Place) for the SIGMOD Programming Contest 2025

Hangrui Zhou, Yiming Qiao, Shaoxuan Tang

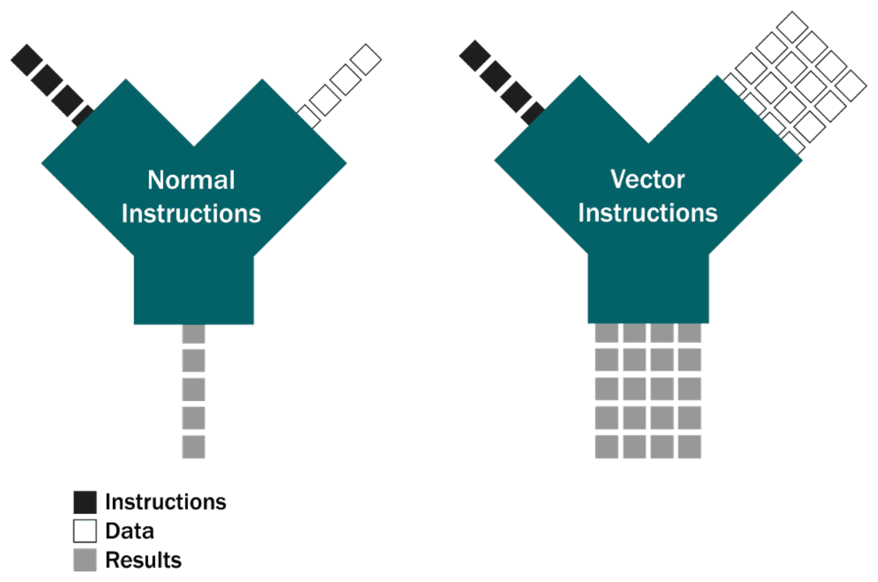
Task: Implement an In-memory Join Pipeline Executor and Operators

- Build a **high-performance join pipeline executor** that processes in-memory base tables using a given join plan.
- The solution must demonstrate strong **end-to-end performance** across **heterogeneous hardware platforms**, including Intel, AMD, ARM, and Power.
- Implementations can leverage **multi-threading, SIMD, join reordering, Bloom filters**, and **other hardware-aware optimizations** – but must use the **same codebase** for all platforms.
- The final output is a fully materialized in-memory result table, and only the total runtime is measured.

Solution: Compact, Clean, and Competitive — Just 2300 Lines of Code

- **Vectorized & Push-Based Execution**: Efficient tuple processing with high CPU utilization and cache locality.
- **Custom Hash Table**: Inspired by DuckDB, using salt-based hashing and lightweight Bloom filters for fast key matching.
- **Min-Max Filter Pushdown**: Early filtering for range conditions to skip irrelevant data efficiently.
- **Data Chunk Compaction**: Reduce data movement and processing cost by compacting sparse vectors [Qiao, Zhang].
- **Lock-Free Multi-threading**: Fast and contention-free parallelism across pipeline stages.
- **Lean Memory Management**: Minimal allocation overhead with simple, allocator-aware design.

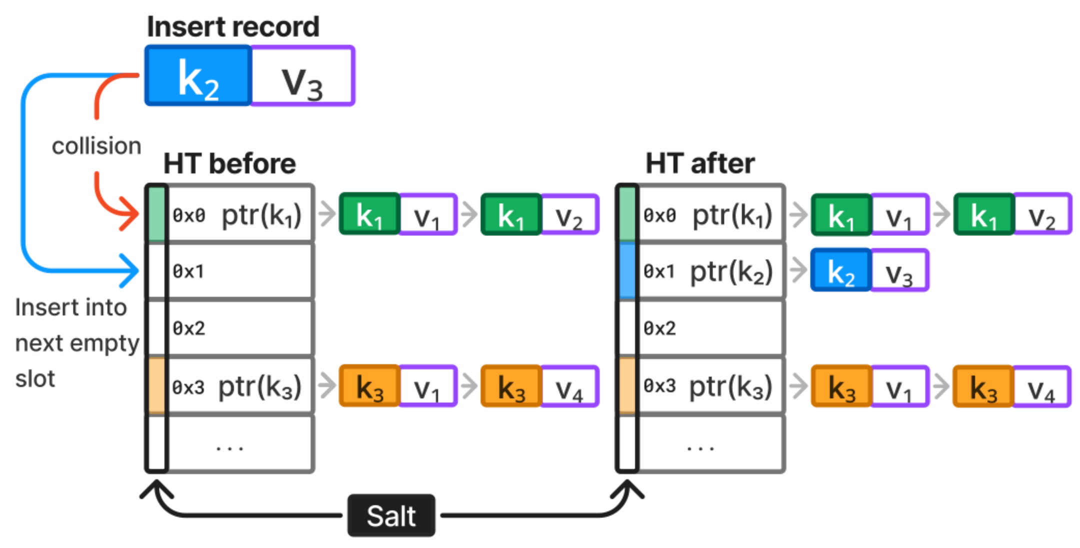
Vectorized & Push-Based Execution



- Processes data in batches to reduce per-tuple overhead.
- Minimizes function calls by operating on entire vectors. Well-suited to modern hardware, leveraging wide SIMD units.

Ref: MonetDB/X100: Hyper-Pipelining Query Execution. CIDR’05.

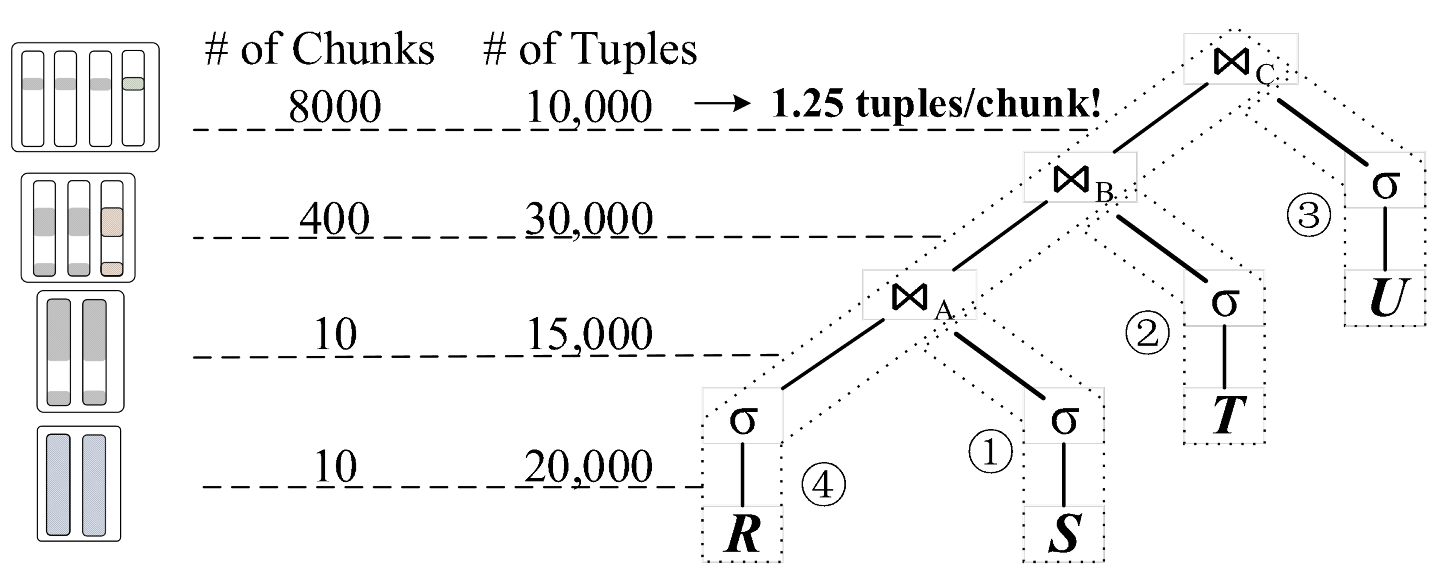
Salty Hash Table



- Combines linear probing and key-based chaining to handle collisions effectively. On collision, inserts into the next free slot and links it to the same-key chain.

Ref: Adaptive Factorization Using Linear-Chained Hash Tables. CIDR’25.

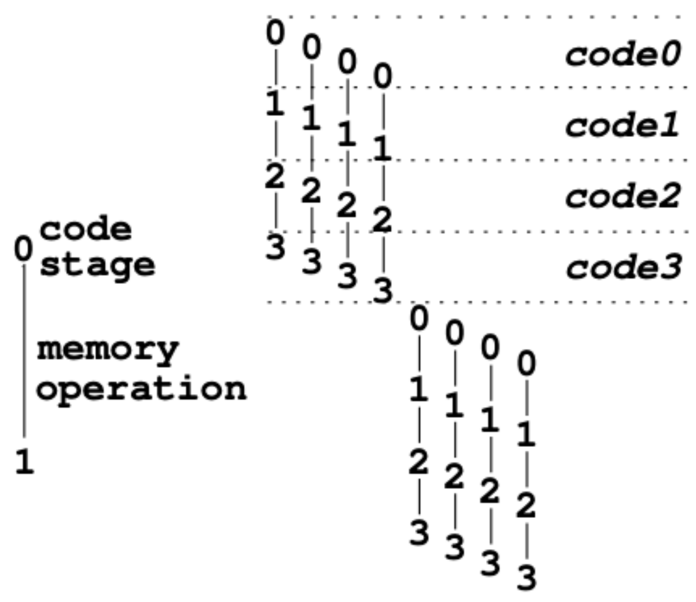
Data Chunk Compaction



- **Filters and joins** often produce Small chunks.
- It leads to:
  ✗ High per-chunk overhead; ✗ Poor SIMD/cache utilization.
- Solution: Logical Compaction.

Ref: Data Chunk Compaction in Vectorized Execution. SIGMOD’25.

Implicit SIMD Optimization



- **We do not use explicit SIMD intrinsics** (e.g., AVX, NEON).
- Instead, our **vectorized and tight loop structure** allows modern compilers to auto-vectorize.
- Delivers performance gains **without sacrificing portability**.
- Tight loop also aligns with Group Prefetch.

Ref: Improving Hash Join Performance through Prefetching. ICDE’04.

Contact: Hangrui Zhou · 596082417@qq.com