Technical Perspective: Optimistically Compressed Hash Tables & Strings in the USSR

Marcin Zukowski
Snowflake Inc.
marcin.zukowski@snowflake.com

Hash tables are possibly the single most researched element of the database query processing layers. There are many good reasons for that. They are critical for some key operations like joins and aggregation, and as such are one of the largest contributors to the overall query performance. Their efficiency is heavily impacted by variations of workloads, hardware and implementation, leading to many research opportunities. At the same time, they are sufficiently small and local in scope, allowing a starting researcher, or even a student, to understand them and contribute novel ideas. And benchmark them... Oh, the benchmarks... :)

This paper by Tim Gubner, Viktor Leis and Peter Boncz addresses less frequently researched aspects of hash tables, in particular string processing, and presents some useful real-system implementation ideas. It improves the main aspects of the (in-memory) hash table performance, CPU operations and memory accesses, with 3 techniques.

Domain-Guided Prefix Compression (DGPC), a combination of FOR coding and bit packing, is used to reduce the memory footprint. While similar methods have been proposed at the scan level, authors discuss how the range metadata can be propagated up and used in the higher layers of the query. This technique, while not discussed much in literature, is used by some systems and allows various optimizations. For example, the discussed idea of using this info to avoid the overflow checks is beneficial for both interpreted and compiled systems. In the case of this paper, DGPC significantly improves the memory usage and hence cache efficiency, with negligible processing overhead. Unfortunately, for optimal performance, the implementation complexity is far from trivial, esp. in a not-compiled engine.

Optimistic Splitting proposes dividing the per-row data stored in the hash table into a frequently accessed hot-set, and rarely accessed cold-set. The authors present a few concrete ideas for such a split, for example overflows in aggregations. This idea can be generalized to other techniques that pull information from later stages to earlier-accessed data structures. I am sure we will see additional ideas in this space in the future. Importantly, techniques like this can have a *negative* effect on performance, so triggering them in a safe and robust manner is very important.

The final technique, **Unique Strings Self-aligned Region (USSR)**, besides having a cool name, provides a simple but powerful mechanism that can improve string processing performance without the complexity of maintaining a full, global string dictionary. USSR does all that **without changing the data format** (assuming strings are kept as pointers). The USSR-aware operations can easily and quickly determine which strings belong to the USSR Data Region and optimize for them. This can dramatically simplify the complexity of implementing this in an existing system. At the same time, the price paid here is that USSR is **partial** and, while beneficial in many cases, in some situations it will not work.

If we treat the self-aligned block as the main building idea behind USSR, the implementation from this paper can be seen as one of the many choices from a broad design space:

- All strings stored in USSR are unique. An implementation without this can still provide some benefits
- A single USSR structure is used for all columns. With, e.g., per-attribute structures, some additional optimizations are possible (e.g., narrower codes for low cardinality key columns, useful for DGPC).
- The paper tightly couples the Data Region and the supporting Linear Hash Table structure, but other options could be used for the latter structure.
- Data Region size is fixed at 512KB. It is possible to envision different sizes, or even dynamic sizing.
- USSR stores strings 8-byte aligned and preceded by hash values. Both of these reduce space utilization and are not strictly needed.

USSR presents some additional opportunities, common to dictionary-compressed strings, that are not discussed in the paper. For example, USSR-strings can use faster serialization methods than other, non-persistent, strings. Another opportunity is fast memoization of various operations for USSR-strings during processing.

While simple in many aspects, USSR also introduces some implementation challenges. For example, some database operations (e.g., serializing data to disk or network) might not produce original string pointers, removing the ability to benefit from USSR-strings further in the query plan. Additionally, maintaining the USSR encoding can be tricky in parallel and, especially, distributed systems. While all these challenges can be addressed, they can significantly add to the implementation complexity.

In summary, if you are a database researcher or, especially, a database system engineer, *do* study this paper. Problems discussed here are real, the improvement ideas are valuable not only for hash table processing, and some presented techniques are (relatively) easy to add to an existing system.