Efficient Query Processing with Optimistically Compressed Hash Tables & Strings in the USSR

Tim Gubner[*], Viktor Leis[^], Peter Boncz[*]

[*] CWI Database Architectures

[^] Friedrich-Schiller-Universität Jena

Motivation

Hash tables frequently used in analytical queries

Crucial for overall performance

But (large) HTs bottlenecked by main memory bandwidth

What can we do about it?

Motivation

Orthogonal approaches:

- Optimize access:
  - Partitioning
  - Prefetching
- Increase fill-rate:
  - Cuckoo*, Robin Hood^ hashing
  - Virtually eliminate empty rows (Concise HT)°
- <u>Shrink the table itself</u>

* R. Pagh, F. Rodler: Cuckoo Hashing

^ P. Celis: Robin Hood Hashing

° R. Barber, G. Lohman, I. Pandis et al: Memory-efficient Hash Joins

Shrinking Hash Tables

100 MiB, magically shrink by 10x:

a) Downsize your computer
b) Increase query throughput

Shrinking Hash Tables

100 MiB, magically shrink by 10x:

a) Downsize your computer
b) Increase query throughput

Bonus:

HT 10 MiB, fits into L3/LLC cache

Improved runtime

Shrinking Hash Tables

100 MiB, magically shrink by 10x:

a) Downsize your computer
b) Increase query throughput

Bonus:

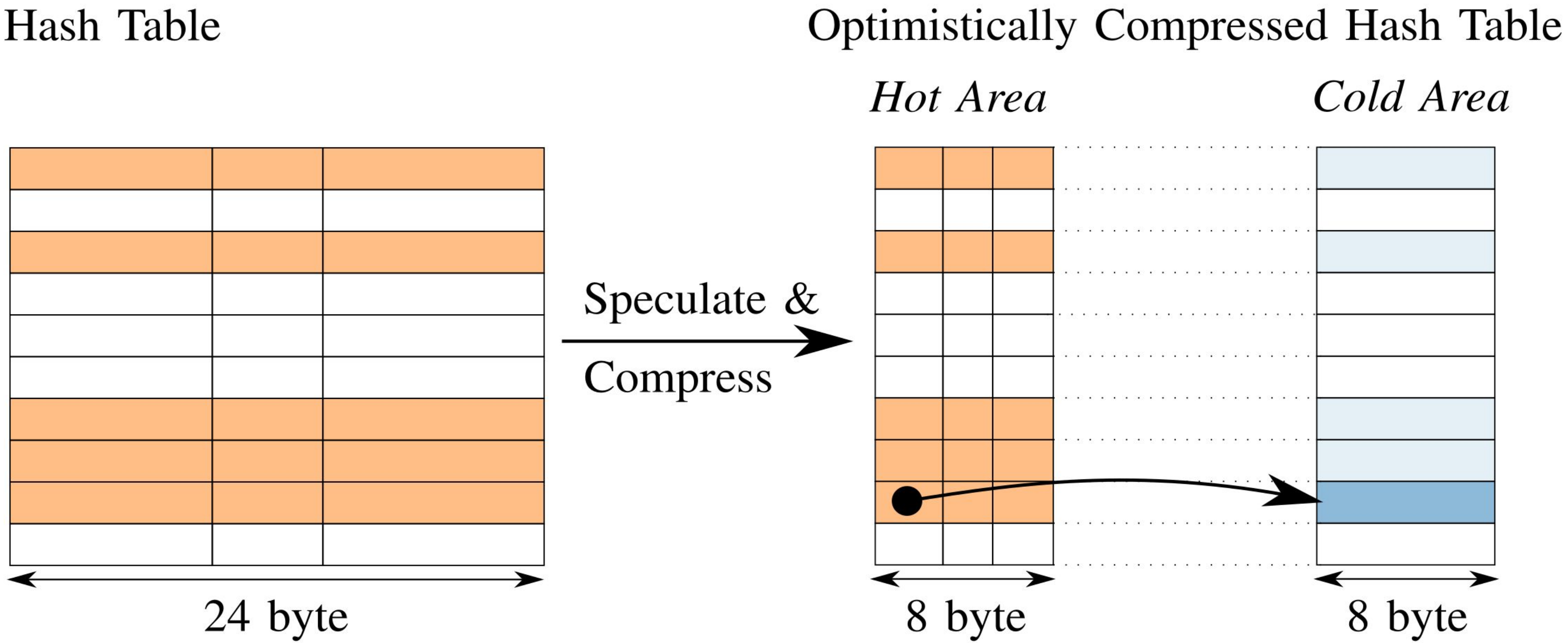
HT 10 MiB, fits into L3/LLC cache

Improved runtime

Better Latency & Throughput

Approach

Overview


Hash Table
Optimistically Compressed Hash Table
Hot Area
Cold Area
Speculate &
Compress
24 byte
8 byte
8 byte

Compression

Requirements:

- Very lightweight
- Support random access

Domain-Guided Prefix Suppression:

- Variant of bit-packing/null suppression
- Lightweight: Handful bitwise operations
- Fast random access: Rows independently compressed & word-aligned
- Fast equality comparisons on compressed data

Optimistic Splitting

- Decrease *effective* memory footprint
- Decompose HT into:

Hot HT:

- Frequently accessed
- Cache-resident
- Aggregates:

    SUM: sub-sums fit smaller data types

- Frequent strings

Cold HT:

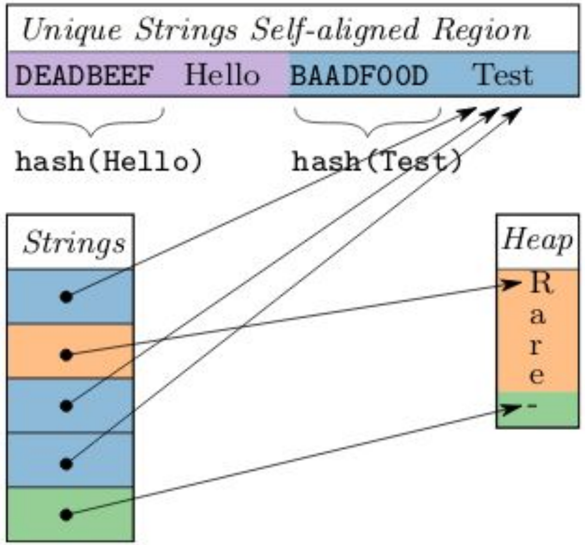
- Rarely accessed
- Main memory
- Aggregates:

    SUM: stores full SUM or overflow counter

- In-frequent strings

Strings in the USSR

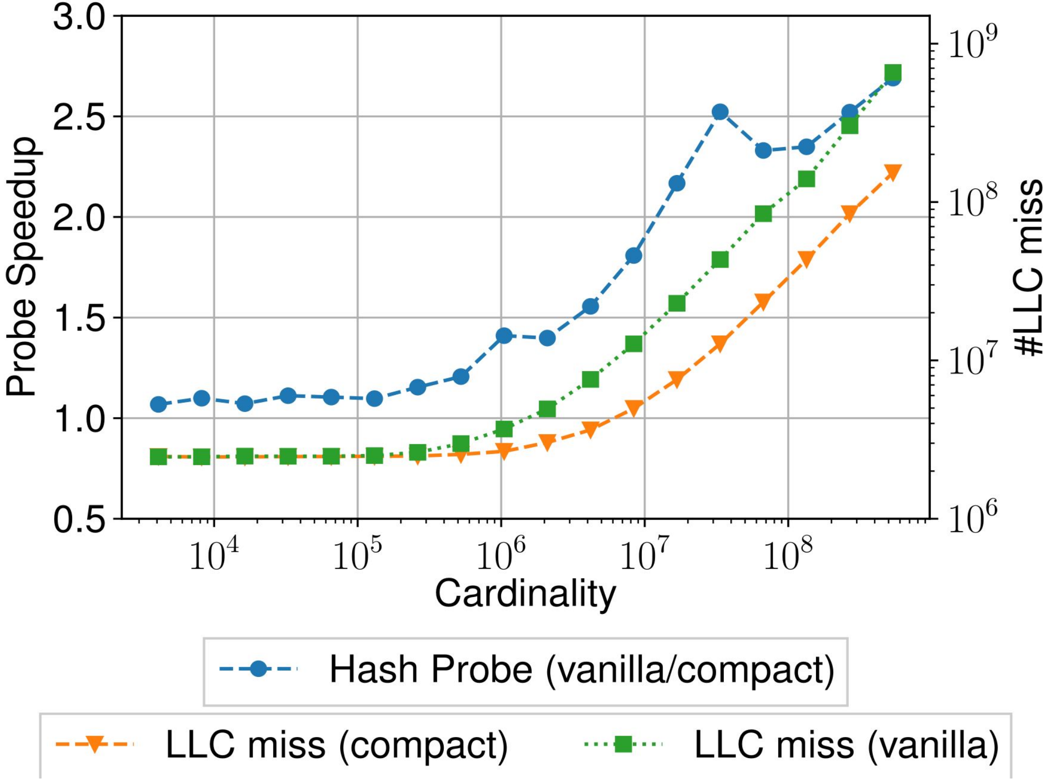
- Assumption: Many strings repeat
- Exploit dictionary-compression, **but**:
    - Global dictionaries come with huge challenges (updates, synchronization)
    - Per-block dictionaries need translation[*]
- Unique Strings Self-aligned Region (USSR):
    - *Query-wide* dictionary
    - Limited size (cache resident)
    - Only holds *frequent* strings
    - Built during scan: Exploit dictionary compression
    - *Easy to retrofit* into existing engines



* J.-G. Lee et al.: Joins on Encoded and Partitioned Data

Experiments

Faster HashJoin Probe



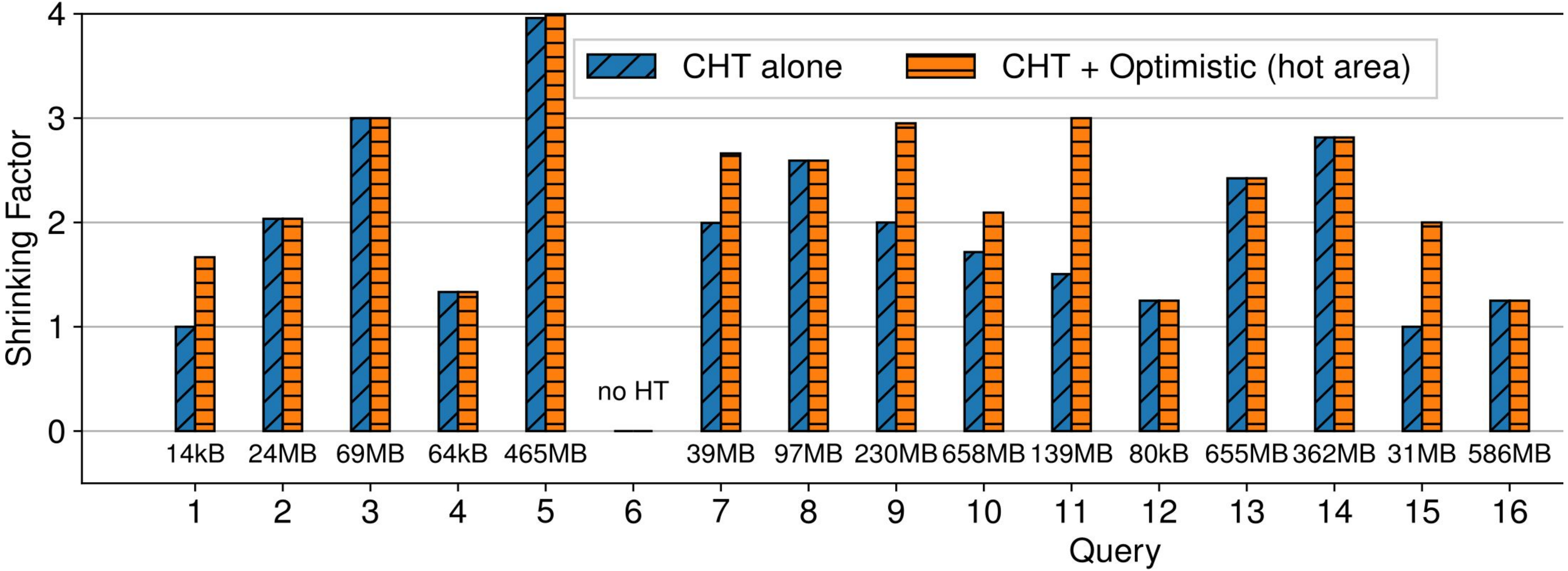
4 keys [0...1000] á 64 bit, 4 payloads [0...10] á 64 bit

Faster GroupBy on strings

- 10 unique strings
- Speedup S(x) over strings with varying length

Length	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**
S(Query)	*1*	*2*	*1*	*2*	*3*	*3*	*5*	*10*	*22*
S(==)	2	2	2	3	3	4	10	20	50
S(Hash)	4	4	4	6	10	15	20	37	80

Smaller Hash Tables in TPC-H



Faster Real-World Workloads (Public BI*)

- String heavy^
- “CommonGovernment” workbook:

Query	1	2	3	4	5	6
Speedup	*2.1*	*1.4*	*2.2*	*1.4*	*1.3*	*1.0*
USSR size (kB)	1.8	0.5	2.0	0.3	66.1	512.0
Rejection Ratio (%)	0.0	0.0	0.0	0.0	0.0	18.3

* https://github.com/cwida/public_bi_benchmark
^ Adrian Vogelsgesang et al.: Get Real: How Benchmarks Fail to Represent the Real World

Summary

Hash tables can be made smaller via:

- Compression
- Optimistic Splitting
- Unique String Self-aligned Region (USSR)

Results in:

(a) Faster runtime
(b) Less memory footprint
(c) Composable (combinable with Cuckoo hashing, Concise HT, etc.)

Improved runtime:

- 50% on TPC-H
- 2x on Public BI (real workload)
- 22x GroupBy on strings
- 2.5x Hash Join probe

Improved memory footprint:

- 4x TPC-H (working set)
- 2x TPC-H (total)