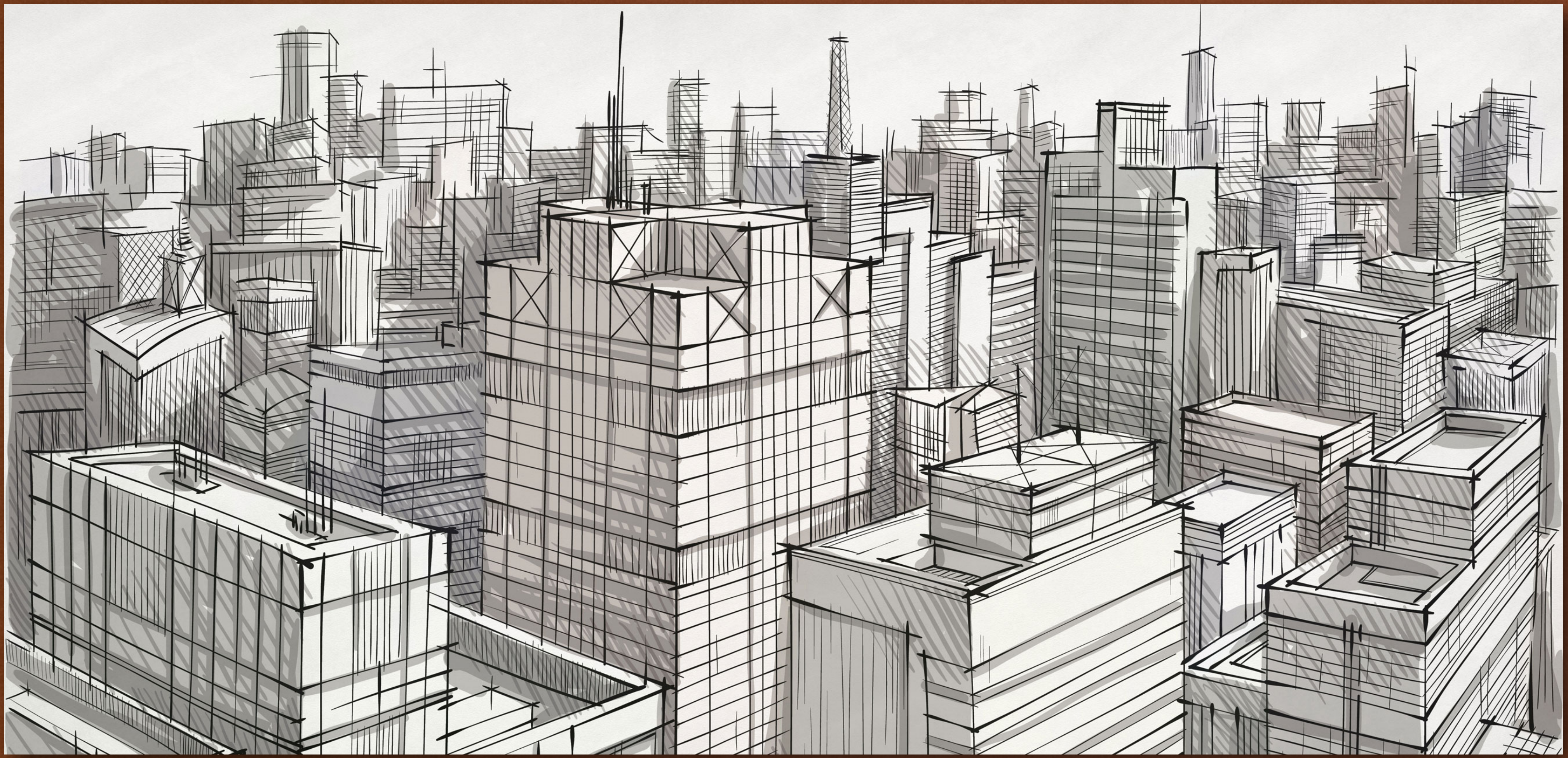
TOWARDS PRACTICAL VECTORIZED ANALYTICAL QUERY ENGINES



ORESTIS POLYCHRONIOU
AMAZON WEB SERVICES

KENNETH A. ROSS
COLUMBIA UNIVERSITY

ANALYTICAL DATABASE ENGINES

	Storage	*Execution engine*	*Unit of execution*	*Method of execution*	*Example*
Traditional database engines	Row oriented	Row oriented	Single row	Interpret	PostgreSQL
First in-memory engines	Column oriented	Column oriented	Entire column	Interpret	MonetDB
"Vectorized" execution engines	Column oriented	Column oriented	Block of tuples	Interpret	Vector(Wise)
Code generating engines	Column oriented	Row oriented	Single row	Compile	HyPer

SIMD IN DATABASES

- SIMD vectorization in databases is limited
  - Most earlier work considers <u>individual</u> operators
    - Too specific setting, e.g., 32/64-bit key-rid pairs & single equality predicate
  - Generic approaches to SIMD too limited
    - Auto-vectorization by compiler can handle simple cases only
    - Applying single SIMD instruction in a loop is too limited
  - VIP aims to support a more <u>realistic</u> set of operations
    - Multiple columns with multiple data types as input to operators
    - Any combination of predicates on selections and joins
    - Arithmetic expressions on aggregate functions on group-by

WHAT IS VIP

- VIP is an execution engine design for analytical databases
  - Goal: data parallelism via **SIMD vectorization**
    - O(n) scalar instructions => O(n/W) SIMD instructions
  - Operators built bottom-up from **sub-operators**
    - Function “kernels” invoked during query execution
    - **Design** choices based on data parallelism, e.g., bitmaps over rid lists
  - Sub-operators process a **column** at a time from a **block** of tuples at a time
    - **Highly-optimized** code tailored to data type
    - Are **type-specific** and highly-optimized
    - Operator invokes sub-operators via **interpretation**

VIP SUB-OPERATOR EXAMPLE

- Sub-operators are optimized SIMD functions
  - Fully vectorized for data parallelism
    - Also allows easy extension to newer SIMD ISAs
  - Different code per data type or size
    - Still manageable number of versions per sub-operator

Hash prototypes:

```
void hash_T(const T* data,
            uint32_t* hash,
            size_t tuples);
```

Execution logic example:

```
x: integer (int32_t)
y: bigint (int64_t)
h: uint32_t

for (size_t i = 0; i < tuples; i += block) {
  hash_init(hash, block);
  hash_int32(x + i, hash, block);
  hash_int64(y + i, hash, block);
  hash_finalize(hash, block);
}
```

Hash sub-operator code example:

```
void hash_int32(const int32_t* data, uint32_t* hash, size_t tuples) {
  const __m512i m_255 = _mm512_set1_epi32(255);
  const __m512i m_fnv = _mm512_set1_epi32(16777619);
  for (size_t i = 0; i < tuples; i += 16) {
    __m512i h = _mm512_load_epi32(hash + i);
    __m512i d = _mm512_load_epi32(data + i);
    for (size_t j = 0; j < 4; ++j) {  // unrolled
      h = _mm512_ternarylogic_epi32(h, d, m_255, 120);
      h = _mm512_mullo_epi32(h, m_fnv);
      d = _mm512_srli_epi32(d, 8);
    }
    _mm512_store_epi32(hash, h);
  }
}
```

SELECTION SCANS

- Intermediate results kept in **bitmaps**
  - Evaluate predicates for all values in the SIMD register
    - Use input bitmap to determine which values to evaluate
    - Use output bitmap to determine which values qualify
    - Combine the bitmaps across predicate tree levels
  - **Skip** (short-circuit) as many tuples as possible
    - Scan the bitmap and skip 256 consecutive values if all invalid
    - Find which strides of 16 consecutive values to process per 256 values

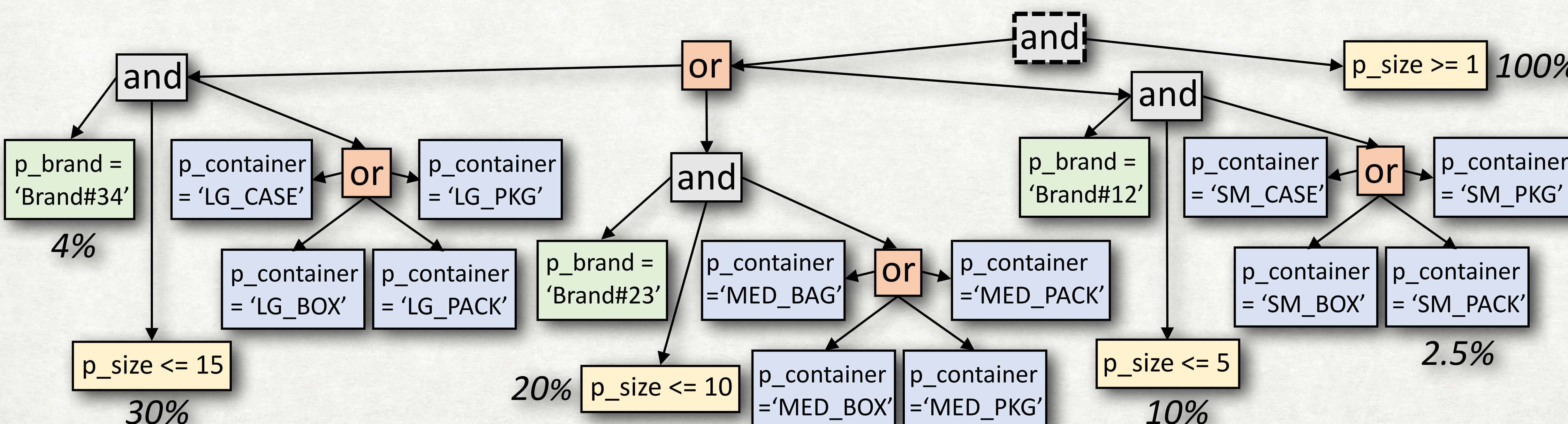
Selection prototype example:

```
void select_int32(const int32_t* data,
                  size_t tuples,
                  const uint16_t* bitmap_in,
                  const uint16_t* bitmap_out,
                  int32_t constant,
                  int operand);
```

SELECTION SCANS

- Traverse predicate trees and invoke sub-operators
  - Predicate trees are provided as **<u>input</u>**
    - Output of query optimization, not execution engine
  - Not limited to CNF or DNF (2-level trees)
    - VIP supports any alternating conjunction/disjunction tree
    - Combine bitmaps across levels using bitwise **<u>and-not</u>**

TPC-H Q19:



COMPRESSION

- Can apply **compression** alongside the scan
  - Using sorted dictionary of distinct values
    - Execute selection directly on **compressed** data
    - Allow us to skip **compressed** codes directly
    - Bit-unpacking is **fast**: 5 instructions per 32/16/8 codes for 16/32/64-bit data
    - We can still use **localized** compression for columns with unique values

Bit-unpacking code:

```
__m512i x1 = _mm512_permutevar_epi32(mask_1, x);
__m512i x2 = _mm512_permutevar_epi32(mask_2, x);
x1 = _mm512_srlv_epi32(x1, mask_3);
x2 = _mm512_sllv_epi32(x2, mask_4);
x = _mm512_ternarylogic_epi32(x1, x2, mask_5, 168);
```

HASH JOINS

- Algorithm split in multiple steps
  - **Hash** both inputs from any type or number of columns
    - Map to fixed-type: uint32_t
  - Hash table build & probe uses the hash values as join key
    - **Single** sub-operator to build and single sub-operator to probe hash table
    - Probe sub-operator implicitly generates rids to access columns

Hash join sub-operator prototypes:

```
typedef struct {
  uint32_t hash;
  int32_t rid;
} join_bucket_t;
```

```
void build(const uint32_t* hash,
           size_t tuples,
           join_bucket_t* hash_table,
           size_t hash_buckets);
```

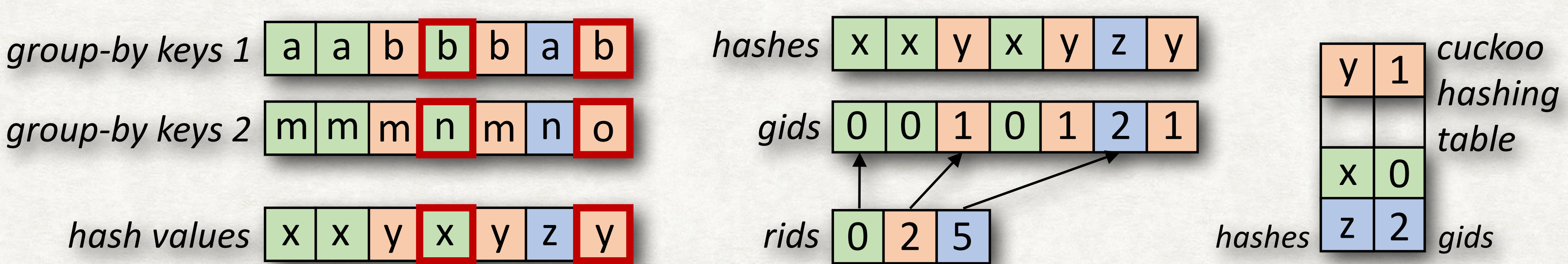
```
size_t probe(const uint32_t* hash,
             size_t tuples,
             const join_bucket_t* hash_table,
             size_t hash_buckets,
             int32_t* inner_rids,
             int32_t* outer_rids);
```

HASH JOINS

- Final steps of hash join algorithm
  - Use rids to <u>**gather**</u> from the columns and evaluate the predicates
    - Including equality predicates to resolve <u>**hash collisions**</u>
  - Use rids to materialize the projected columns (optional)
    - We can push rids as payloads to follow-up joins
    - VIP supports both early- and late-materialized joins
- Cache-conscious hash joins
  - Partitioning is a <u>**pre-processing**</u> step
    - VIP supports both partitioned and non-partitioned hash join
    - Single sub-operator to compute hash partitioning output offsets
    - Type-specific sub-operators to shuffle data

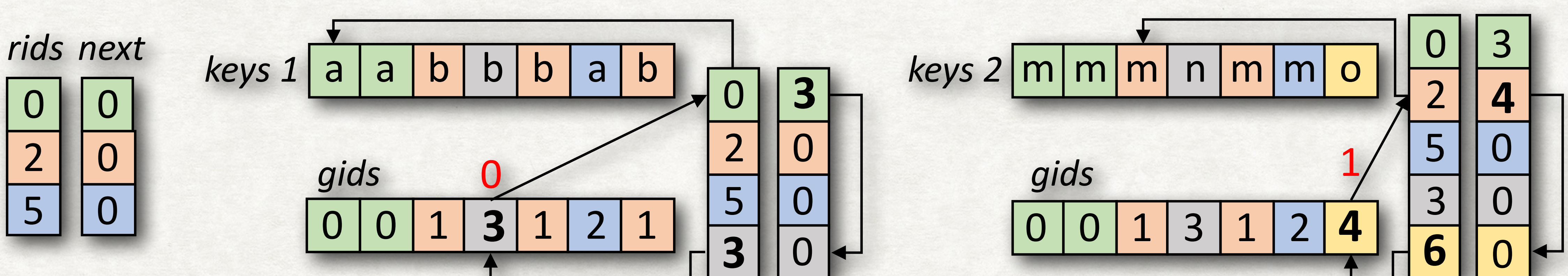
GROUP-BY AGGREGATION

- Split in multiple steps
  - **Estimate** the group cardinality
    - By estimating the number of distinct hash values
    - Determine whether to use **partitioning**
  - **Map** hashes to unique group ids (gids)
    - **Single** (very complex) sub-operator using cuckoo hashing
    - Include an "rid" to the **first** tuple of each group



GROUP-BY AGGREGATION

- Split in multiple steps
  - Fix **hash collisions** of group-by keys
    - Compare with the value of the first tuple per group and add a new group id
    - Done column at-a-time with type-specific sub-operator
  - Use group ids to compute aggregates
    - Use **direct mapping** from column values to aggregate functions
    - Store expression results in cache-resident buffers

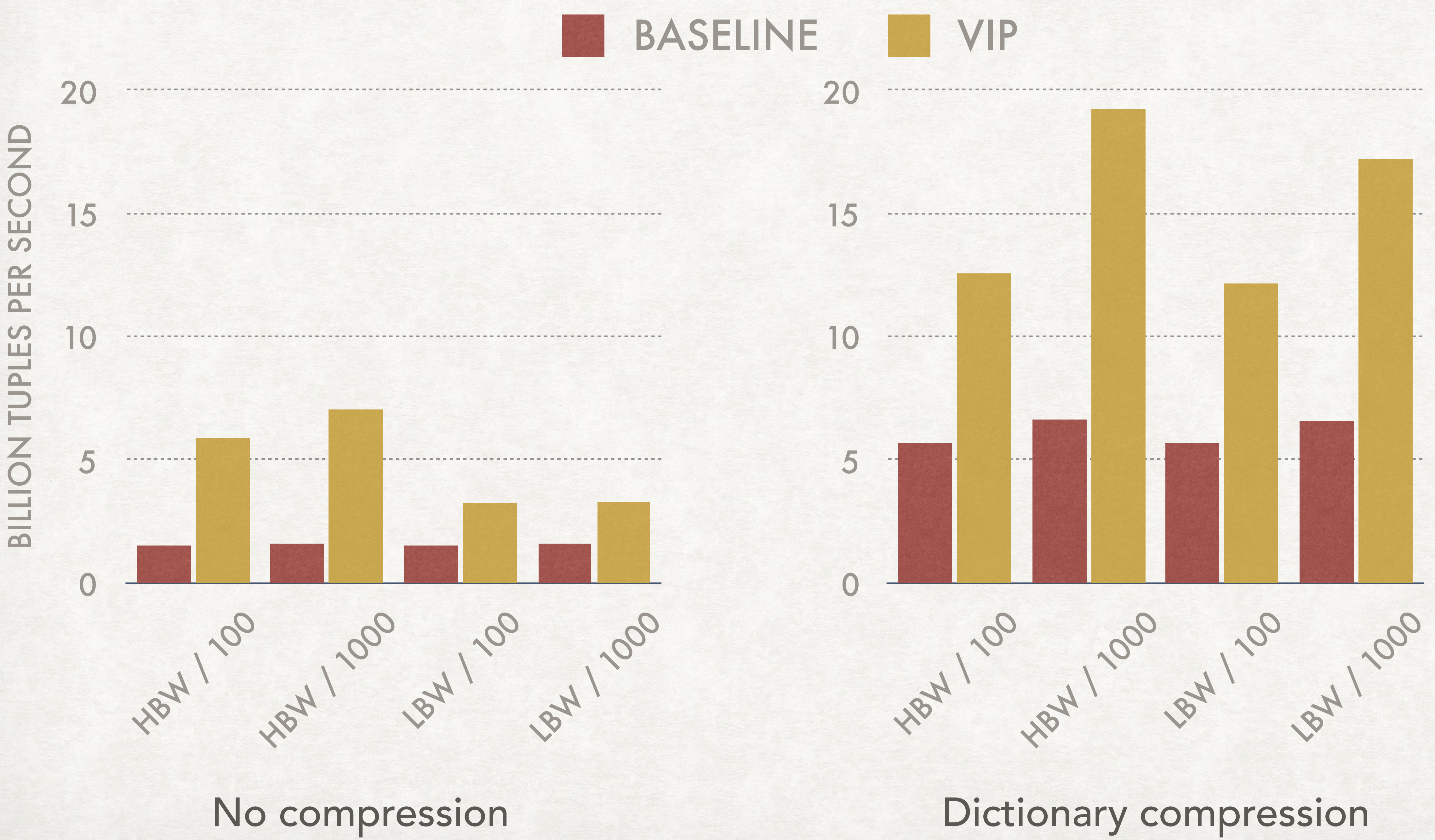


RESULTS

- Setting
  - Hardware
    - Xeon Phi 7210 CPU (Knight's Landing) with 64 physical cores @ 1.3 GHz
    - 16GB of on-chip **high-bandwidth** memory (HBW) with 295GB/s load bandwidth
    - 192GB of off-chip **low-bandwidth** memory (LBW) with 70GB/s load bandwidth
  - Baseline
    - Hand-optimized scalar code emulating **code-generation** designs
    - Ignoring any runtime compilation cost (VIP has no runtime compilation)
  - Workload
    - Derived from the TPC-H benchmark

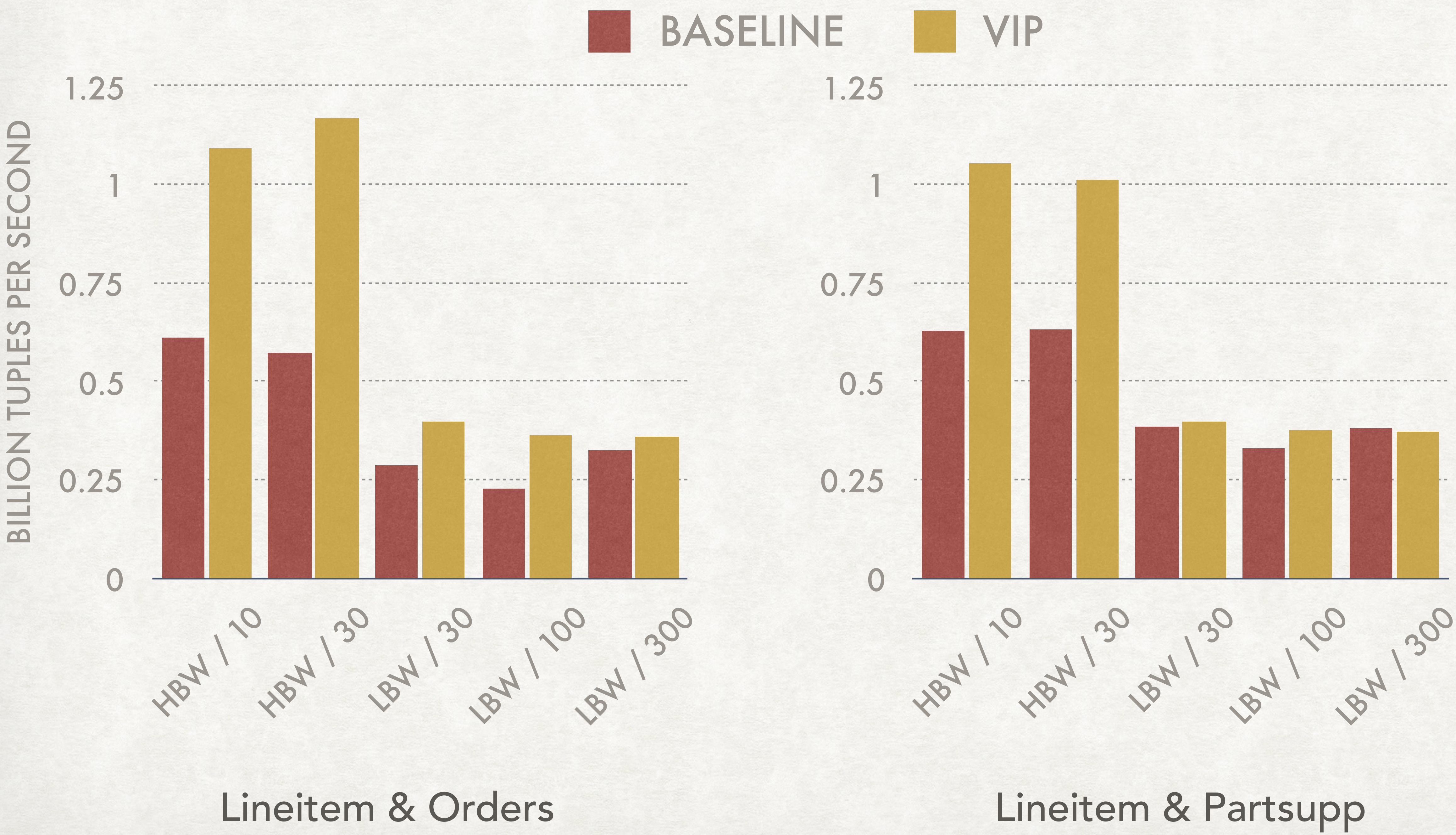
RESULTS

- TPC-H Q19 selection on PART table



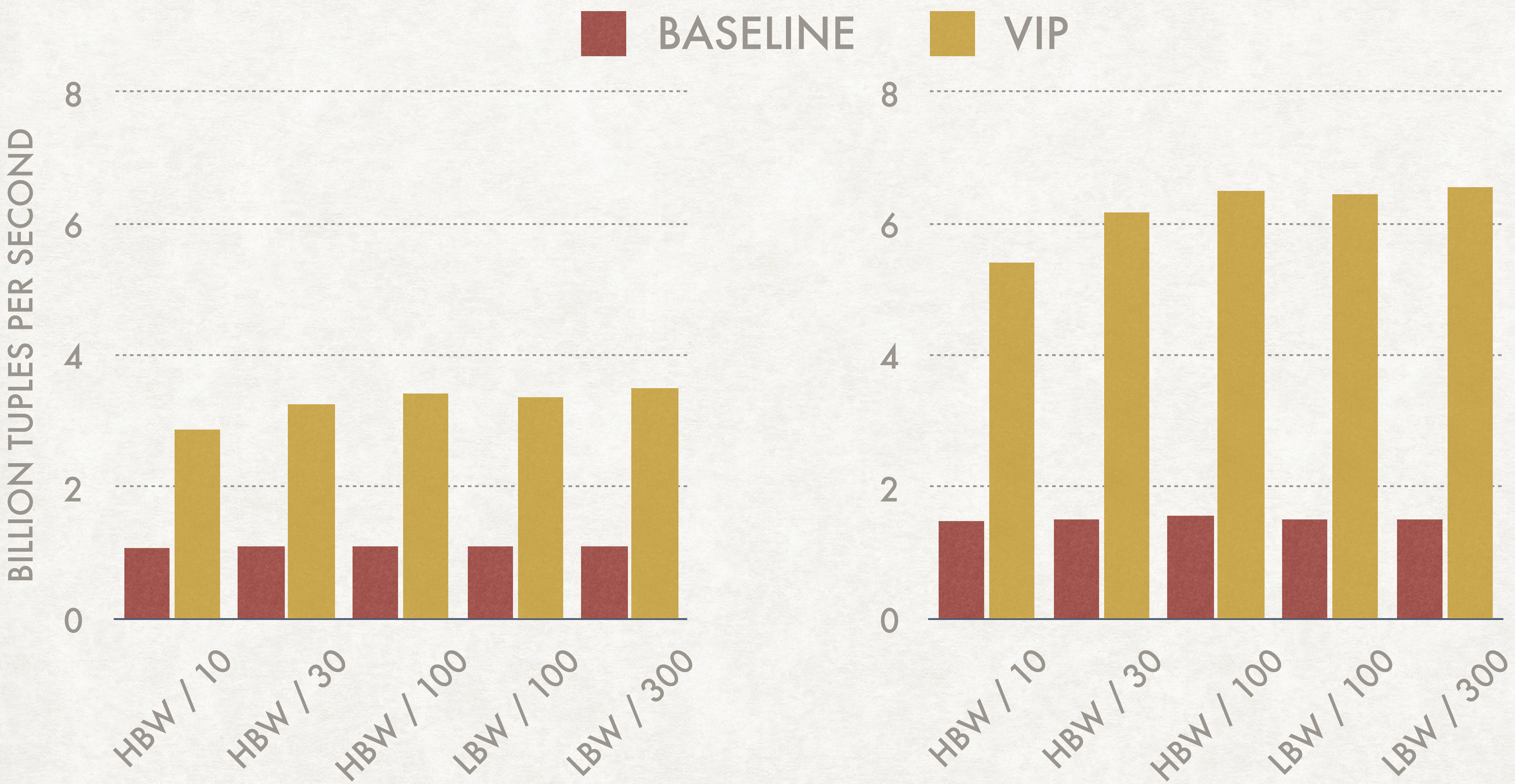
RESULTS

- TPC-H fact table joins



RESULTS

- TPC-H Q1 aggregation on LINEITEM table



With group-by

Without group-by

CONCLUSION

- Towards full SIMD vectorization in databases
  - VIP design achieves full vectorization in a more realistic setting
    - Using pre-compiled type-specific sub-operators to build operators
  - VIP beats the state-of-the-art design
    - Future work: evaluate VIP design on mainstream CPUs with AVX-512