2022 DATA ROCKS

Database Vectorized

StarRocks

康凯森

StarRocks Database Vectorized

01

Database Performance

2

Vectorized Basic

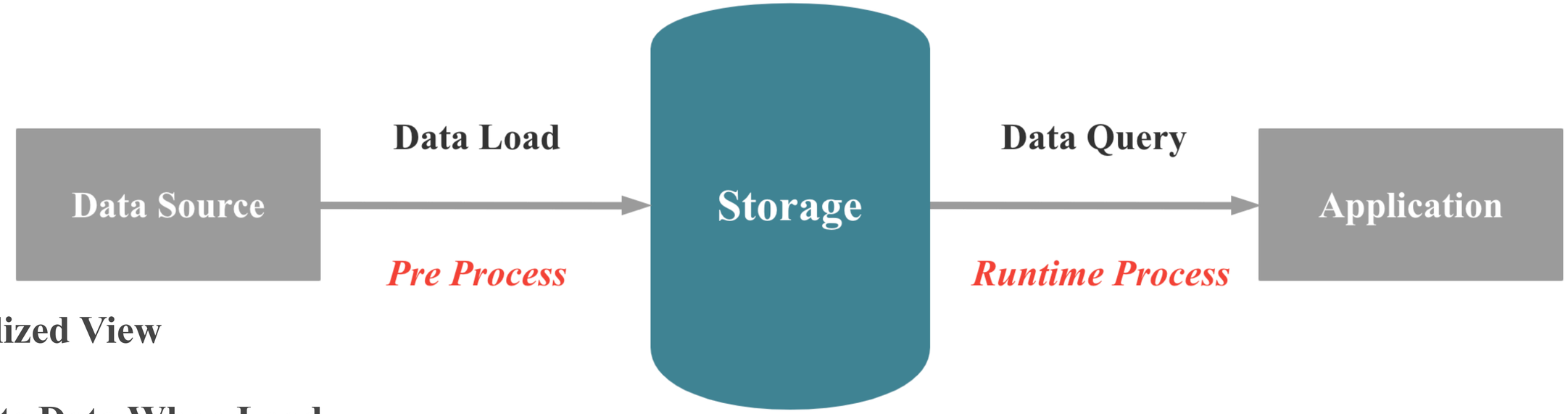
3

Database Vectorized

4

Thinking Vectorized

How To Build A Fast Database： Pre Process VS Runtime Process

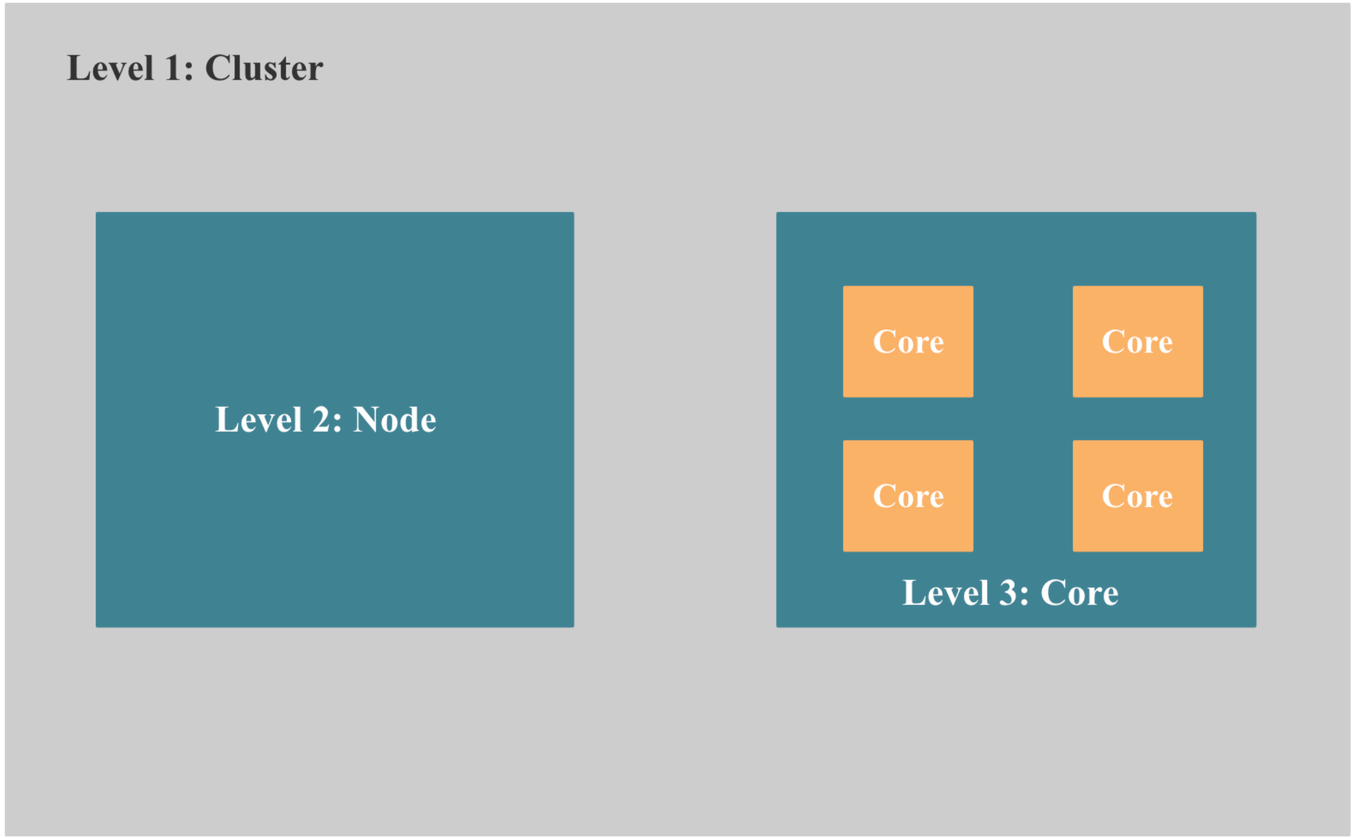


- Materialized View
- Aggregate Data When Load
- Index
- Cache

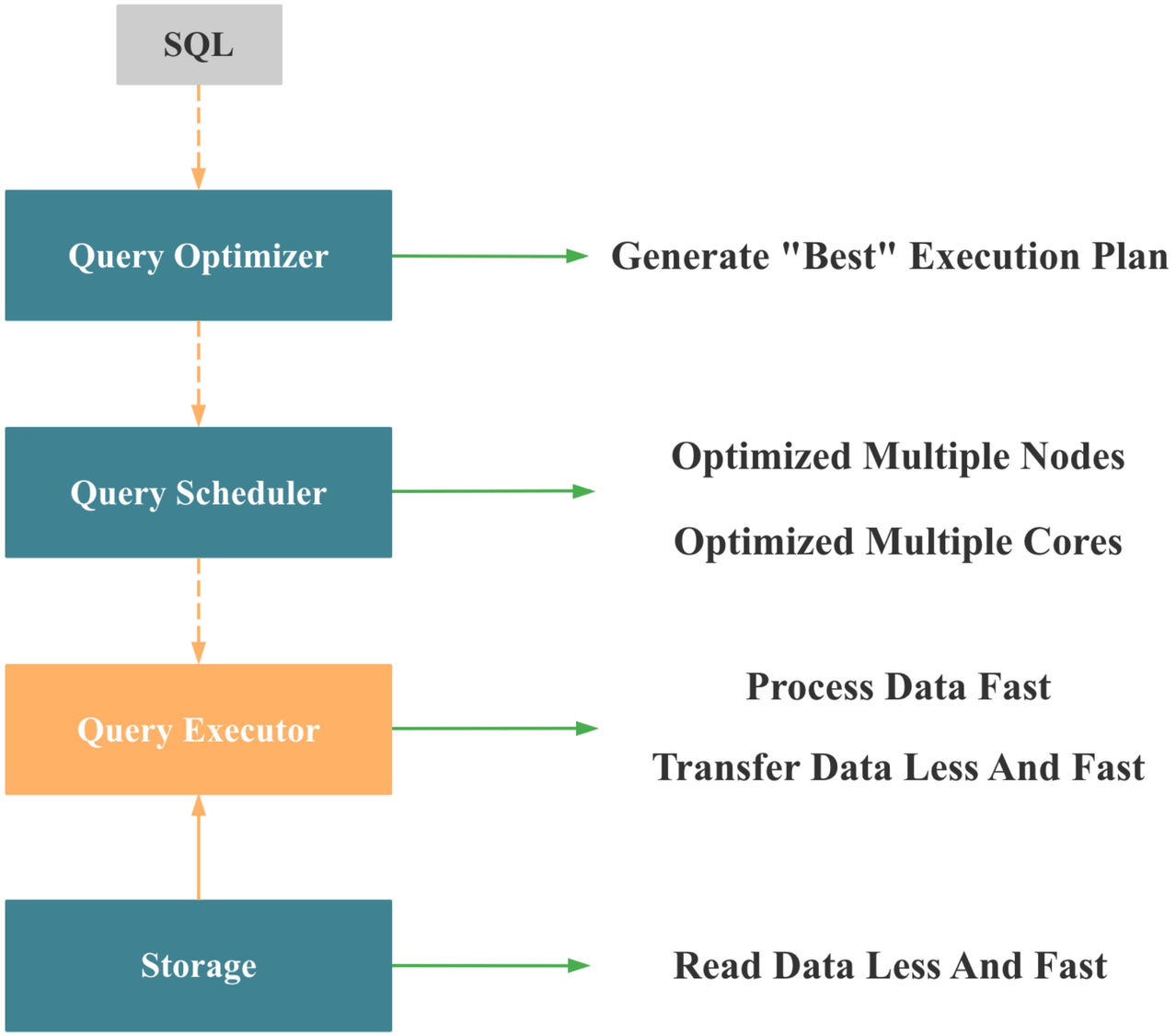
The more pre process, The less runtime process

How To Build A Fast Database: Architecture Perspective

- Level 1: Optimized Multiple Nodes
- Level 2: Optimized Multiple Cores
- Level 3: Optimized Single Core

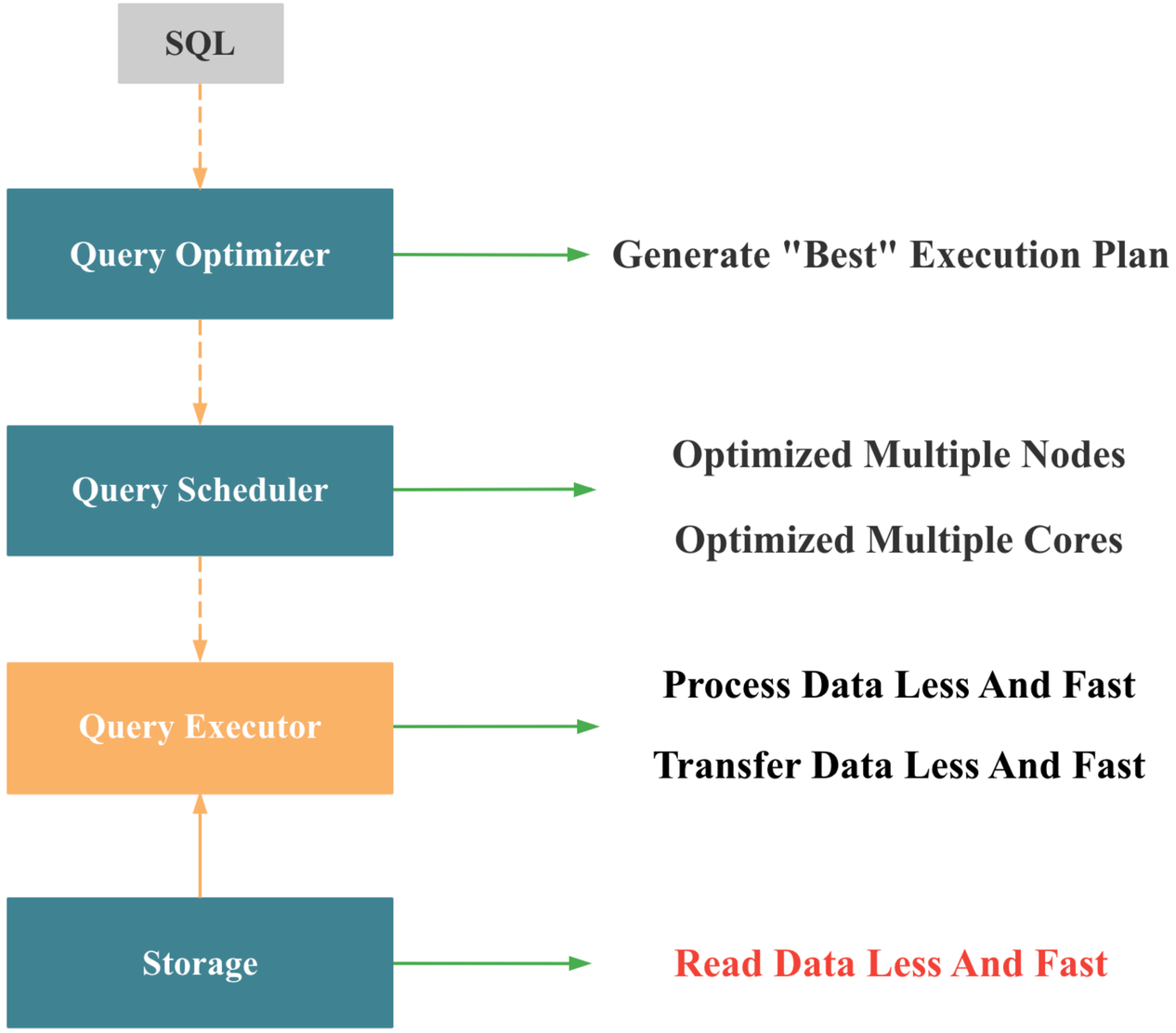


How To Build A Fast Database: Data Flow Perspective 1



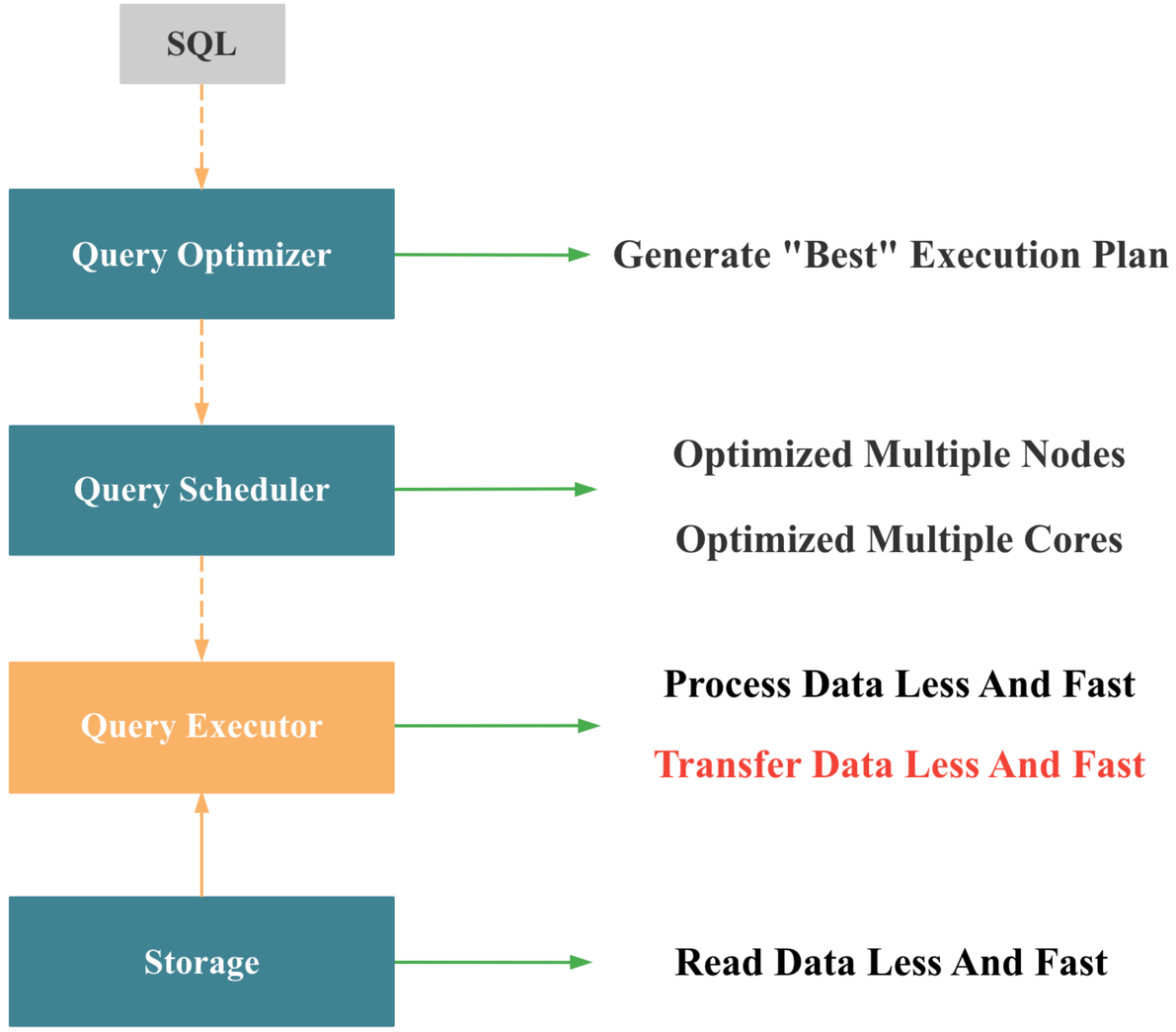
How To Build A Fast Database: Data Flow Perspective 2

- Partition And Bucket Prune
- Read Necessary Column
- Read Compressed Data
- Skip Data By Index
- Skip Data By Metadata
- Late Materialization
- Operations On Encoded Data
- Vectorized Process



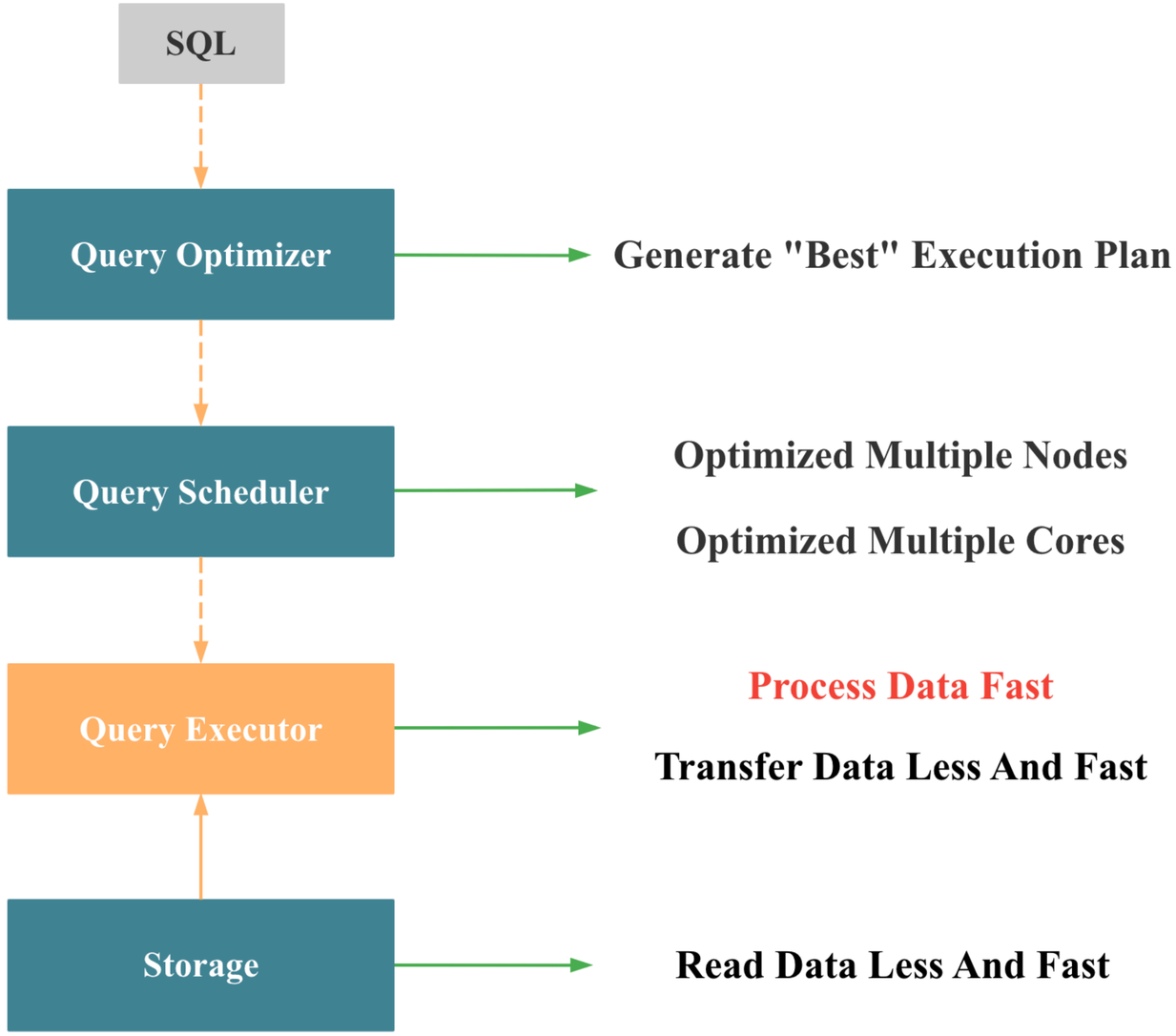
How To Build A Fast Database: Data Flow Perspective 3

- Shuffle By Column
- Compress Data By Column
- Global Runtime Filter
- Operations On Encoded Data
- Colocate Join
- Replication Join
- Bucket Shuffle Join



How To Build A Fast Database: Data Flow Perspective 4

- Data Structure and Algorithms
- Vectorization
- SIMD
- Adaptive Strategy
- Cache Optimization
- C++ Low Level Optimization



How To Build A Fast Database: Resource Perspective

- Read Data Less And Fast (IO)
- Transfer Data Less And Fast (Network)
- Process Data Less And Fast (CPU & Memory)

StarRocks Database Vectorized

1 Database Performance

02 Vectorized Basic

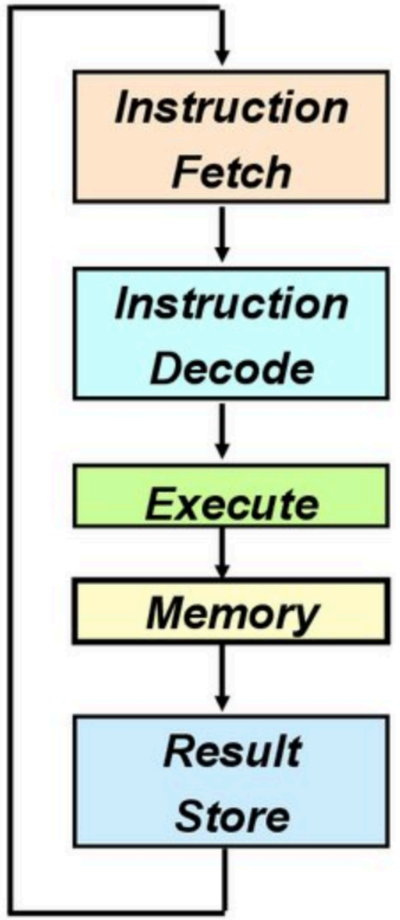
3 Database Vectorized

4 Thinking Vectorized

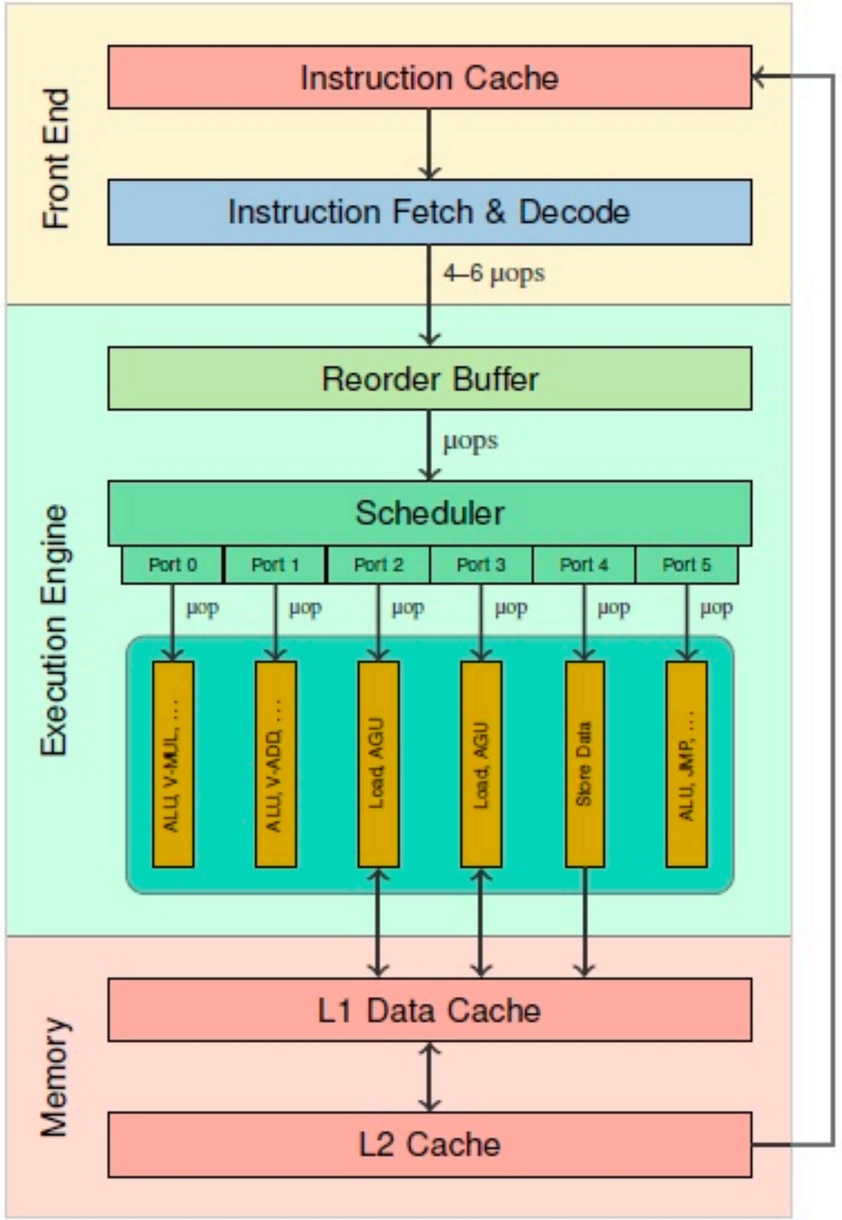
CPU Instruction Execution

Instruction Execution Stages

- **Fetch**
  - Fetch instruction pointed by PC from I-Cache
- **Decode**
  - Decode instruction (generate control signals)
  - Fetch operands from register file
- **Execute**
  - For a memory access: calculate effective address
  - For an ALU operation: execute operation in ALU
  - For a branch: calculate condition and target
- **Memory Access**
  - For load: read data from memory
  - For store: write data into memory
- **Write Back**
  - Write result back to register file
  - update program counter



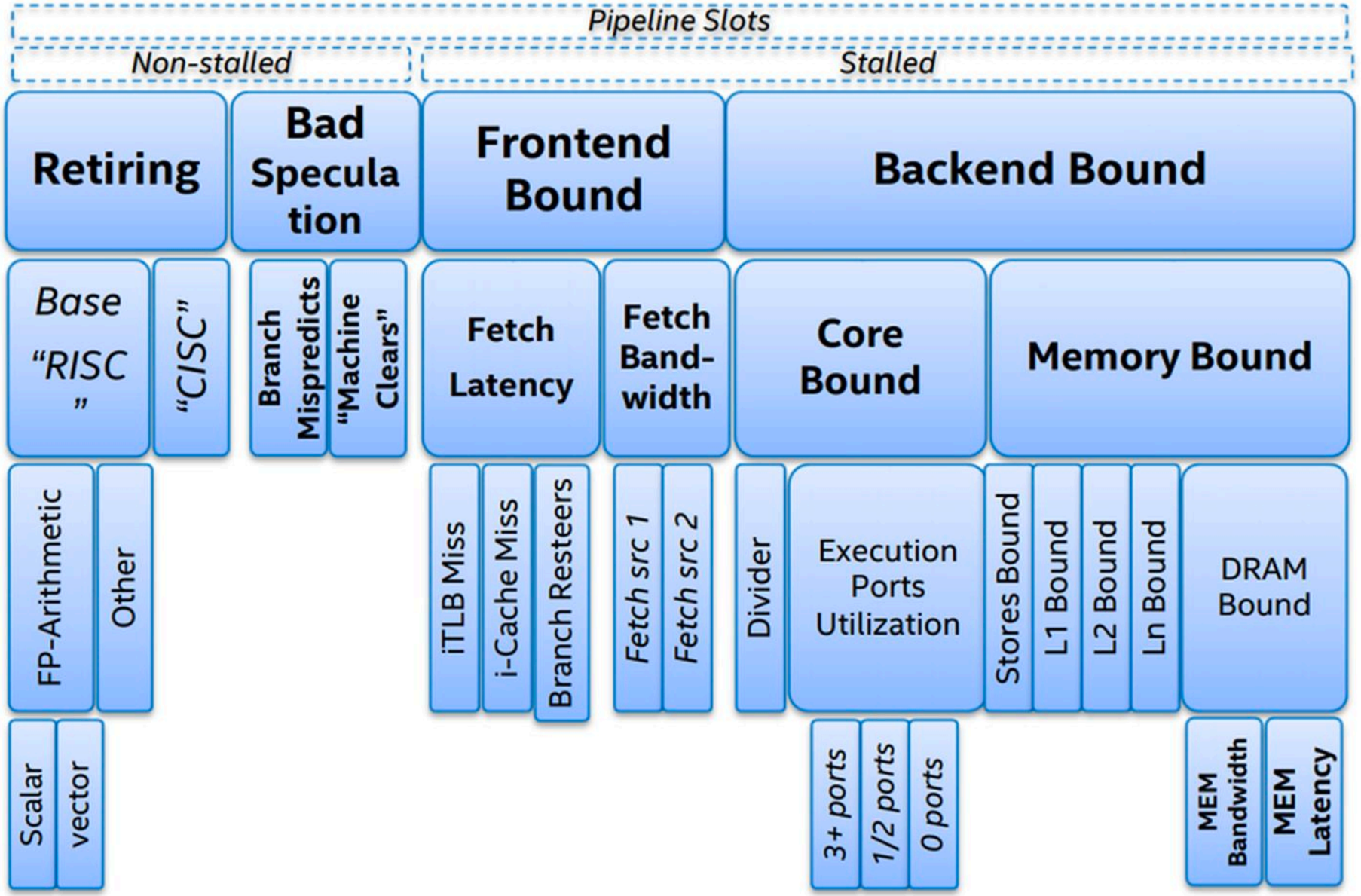
CPU FrontEnd And Backend



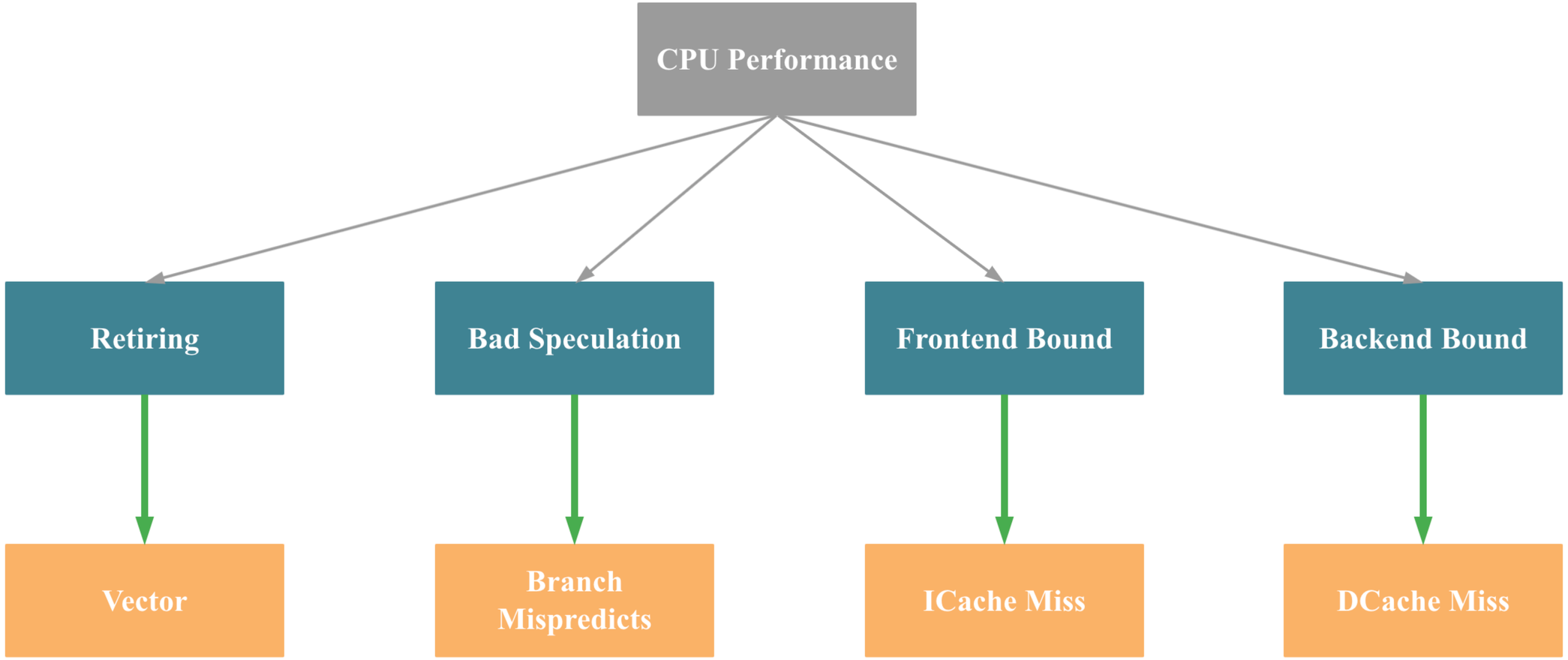
CPU Time

$$\text{CPU Time} = \text{Instruction Number} * \text{CPI} * \text{Clock Cycle Time}$$

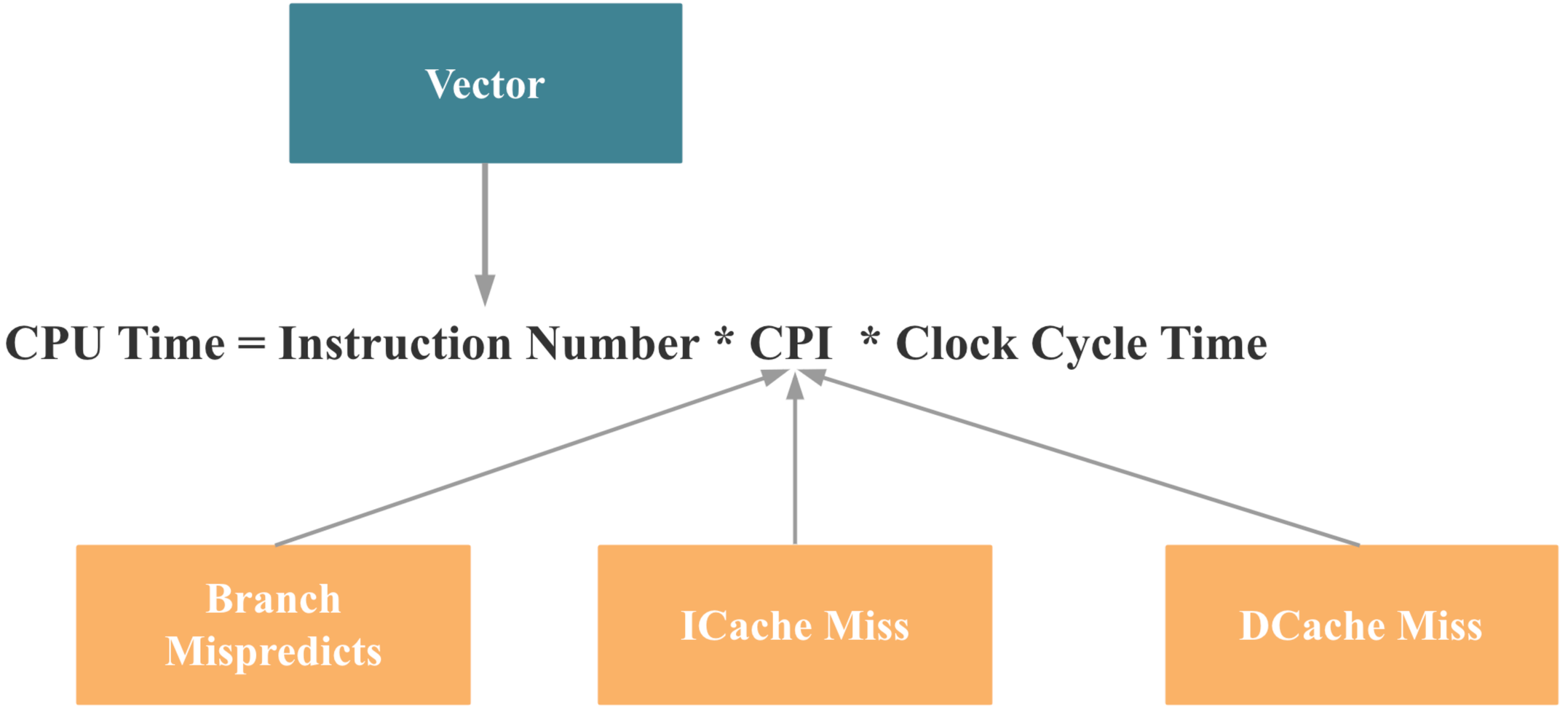
CPU Performance Analysis Top-Down Hierarchy



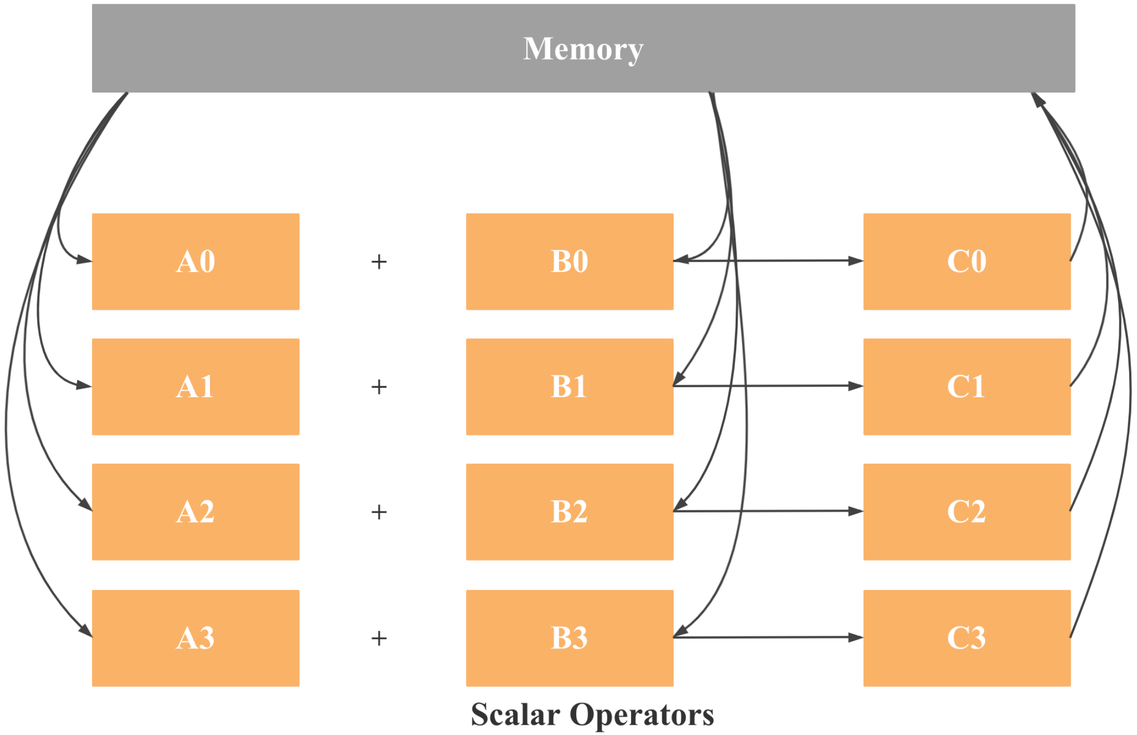
CPU Performance Analysis Top-Down Hierarchy



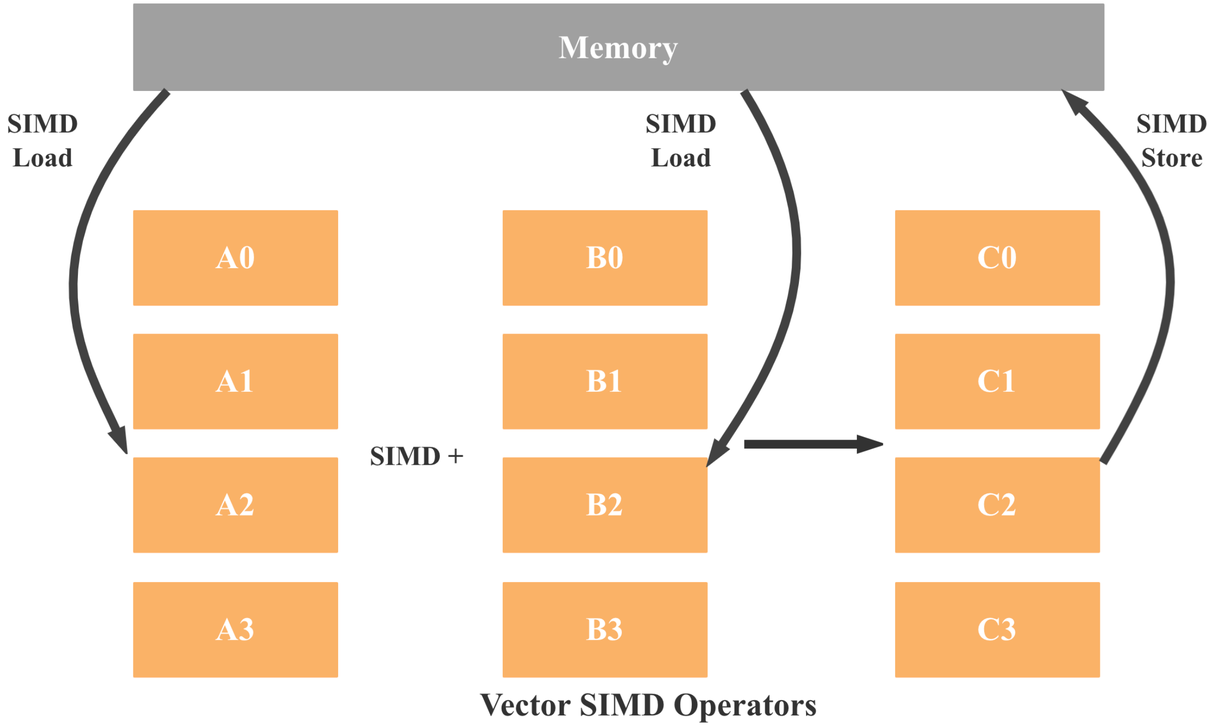
CPU Time


Vector
CPU Time = Instruction Number * CPI * Clock Cycle Time
Branch Mispredicts
ICache Miss
DCache Miss

What SIMD

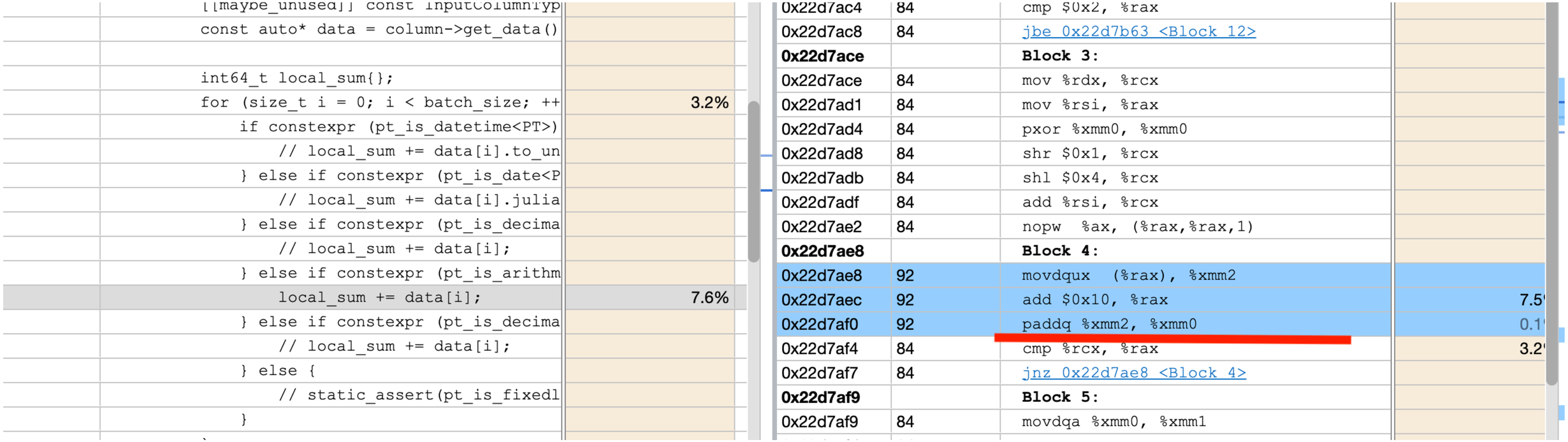


SISD



SIMD

SIMD Registers



```
[[maybe_unused]] const InputColumnTyp
const auto* data = column->get_data()

int64_t local_sum{};
for (size_t i = 0; i < batch_size; ++                3.2%
    if constexpr (pt_is_datetime<PT>)
        // local_sum += data[i].to_un
    } else if constexpr (pt_is_date<P
        // local_sum += data[i].julia
    } else if constexpr (pt_is_decima
        // local_sum += data[i];
    } else if constexpr (pt_is_arithm
        local_sum += data[i];                        7.6%
    } else if constexpr (pt_is_decima
        // local_sum += data[i];
    } else {
        // static_assert(pt_is_fixedl
    }
```

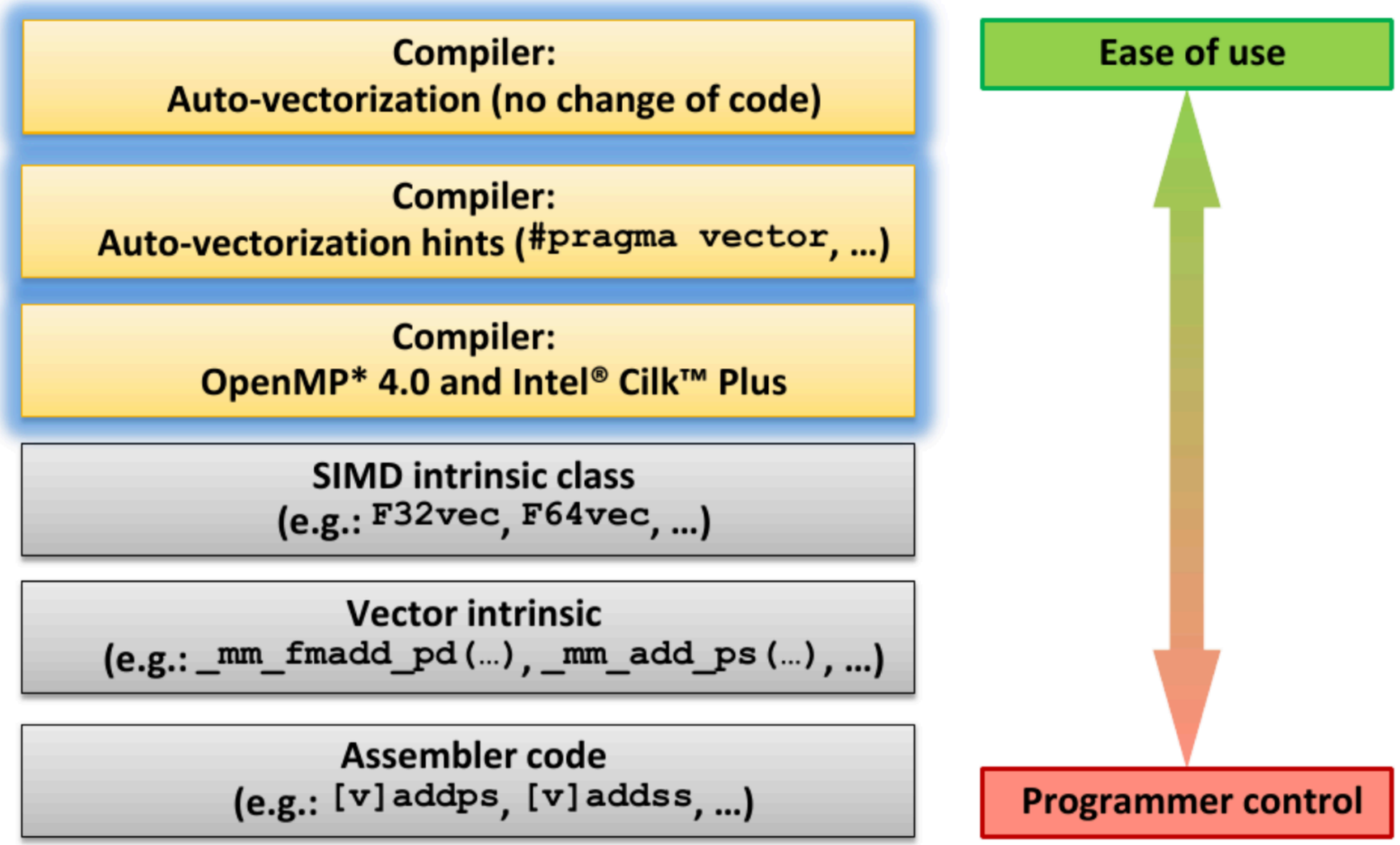
```
0x22d7ac4   84   cmp $0x2, %rax
0x22d7ac8   84   jbe 0x22d7b63 <Block 12>
0x22d7ace        Block 3:
0x22d7ace   84   mov %rdx, %rcx
0x22d7ad1   84   mov %rsi, %rax
0x22d7ad4   84   pxor %xmm0, %xmm0
0x22d7ad8   84   shr $0x1, %rcx
0x22d7adb   84   shl $0x4, %rcx
0x22d7adf   84   add %rsi, %rcx
0x22d7ae2   84   nopw  %ax, (%rax,%rax,1)
0x22d7ae8        Block 4:
0x22d7ae8   92   movdqux  (%rax), %xmm2
0x22d7aec   92   add $0x10, %rax                     7.5
0x22d7af0   92   paddq %xmm2, %xmm0                  0.1
0x22d7af4   84   cmp %rcx, %rax                      3.2
0x22d7af7   84   jnz 0x22d7ae8 <Block 4>
0x22d7af9        Block 5:
0x22d7af9   84   movdqa %xmm0, %xmm1
```

xmm (128 bit wide register)

ymm (256 bit wide register)

zmm (512 bit wide register)

Many Ways to Vectorize



(Image: Intel)

Compile Auto Vectorize

- Countable loop
- Function call should be inline or simple math function
- No data dependencies
- No complex conditions

Compile Hint Vectorize: Restrict

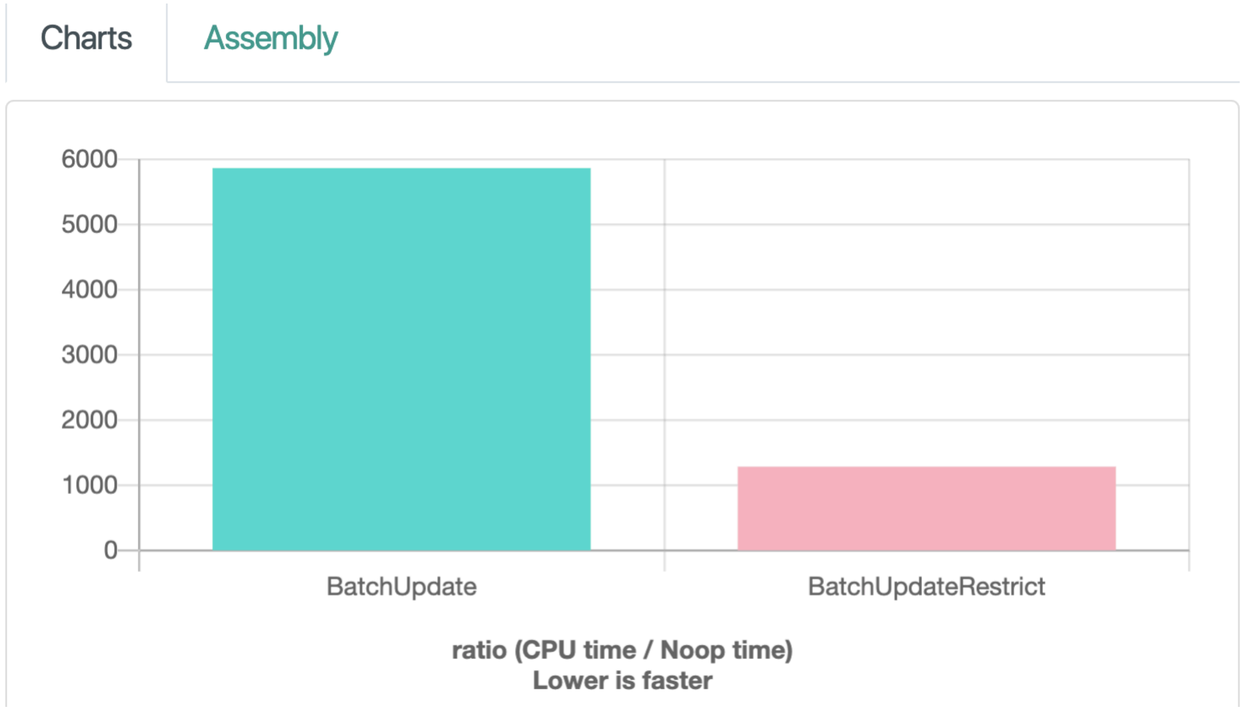
```
void batch_update(int* res, int* col, int size) {
    for (int i = 0; i < size; ++i) {
        *res += col[i];
    }
}
```

Which is Faster ?

```
void batch_update_restrict(int* __restrict res, int* __restrict col,
    for (int i = 0; i < size; ++i) {
        *res += col[i];
    }
}
```

Compile Hint Vectorize: Restrict

```
__attribute__ ((noinline))
void batch_update(int* res, int* col,int size) {
    for(int i = 0;i < size; ++i) {
        *res += col[i];
    }
}

__attribute__ ((noinline))
void batch_update_restrict(int* __restrict res, int* __res
    for(int i = 0;i < size; ++i) {
        *res += col[i];
    }
}
```



Restrict Tells The Compile The Arrays In Distinct Locations In Memory

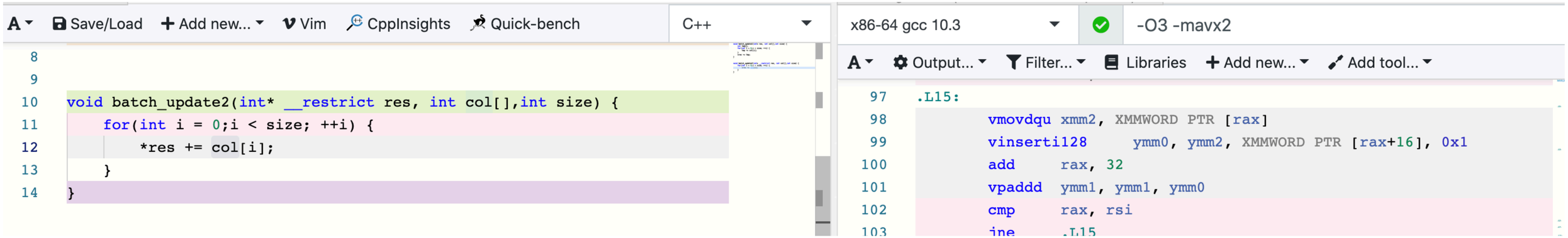
How to Ensure SIMD Instructions Used

```
[#10#kks@sandbox-sql ~ 22:34:07]$g++ vector.cpp -o vector  -std=c++17 -O3 -fopt-info-vec-all -march=native
vector.cpp:14:26: optimized: loop vectorized using 32 byte vectors
vector.cpp:9:26: optimized: loop vectorized using 32 byte vectors
vector.cpp:5:5: note: vectorized 2 loops in function.
```

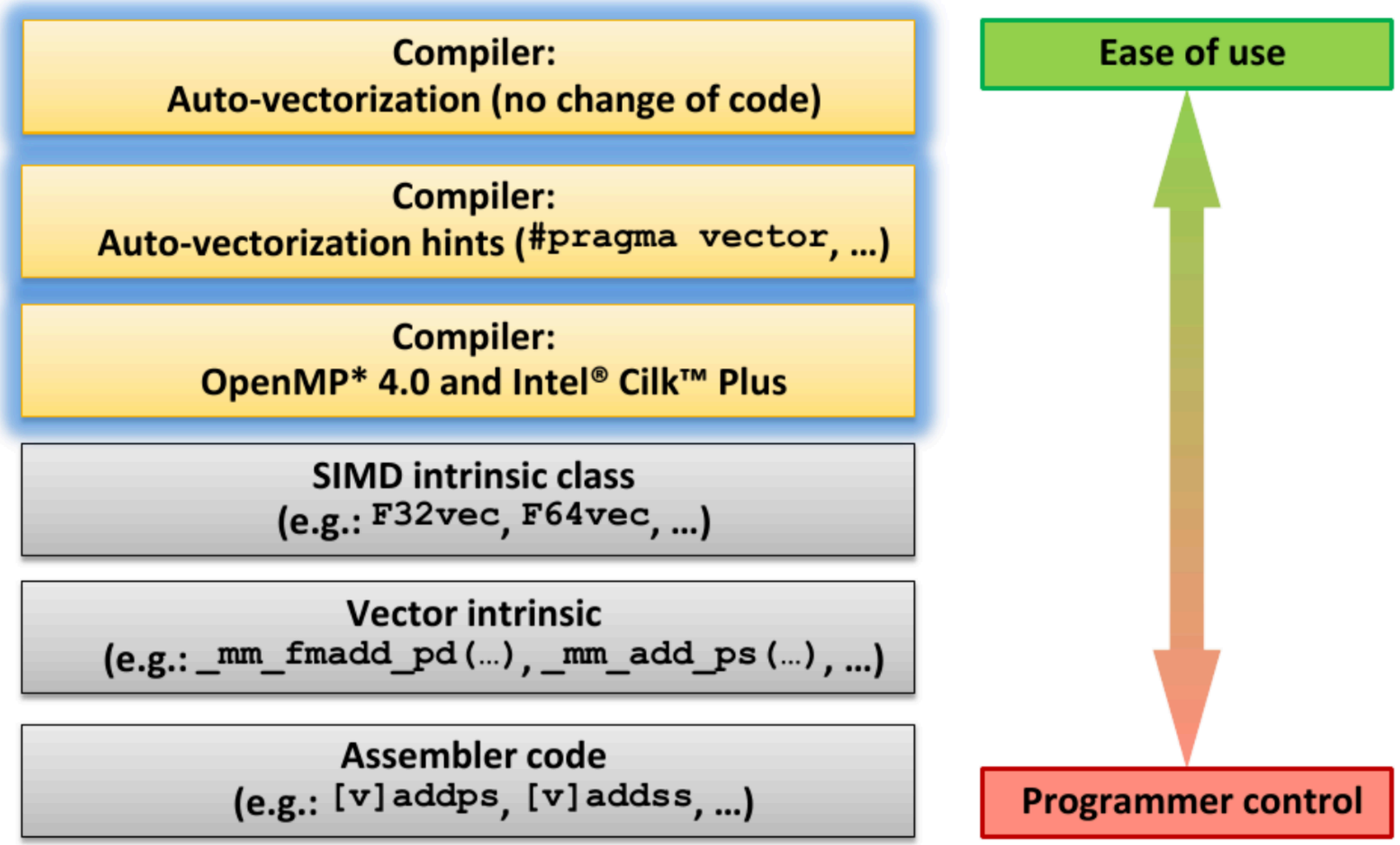
```
[#12#kks@sandbox-sql ~ 22:40:22]$g++ vector.cpp -o vector  -std=c++17 -O3 -fopt-info-vec-all -march=native
vector.cpp:6:26: optimized: loop vectorized using 32 byte vectors
vector.cpp:6:26: optimized:  loop versioned for vectorization because of possible aliasing
vector.cpp:6:26: optimized: loop vectorized using 16 byte vectors
vector.cpp:5:6: note: vectorized 1 loops in function.
```

- -fopt-info-vec-optimized
- -fopt-info-vec-missed
- -fopt-info-vec-note
- -fopt-info-vec-all

How to Ensure SIMD Instructions Used

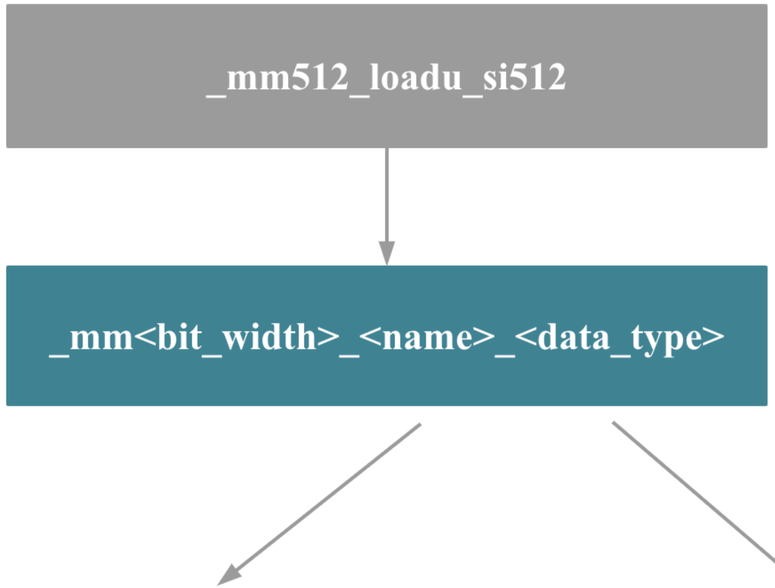


Many Ways to Vectorize



(Image: Intel)

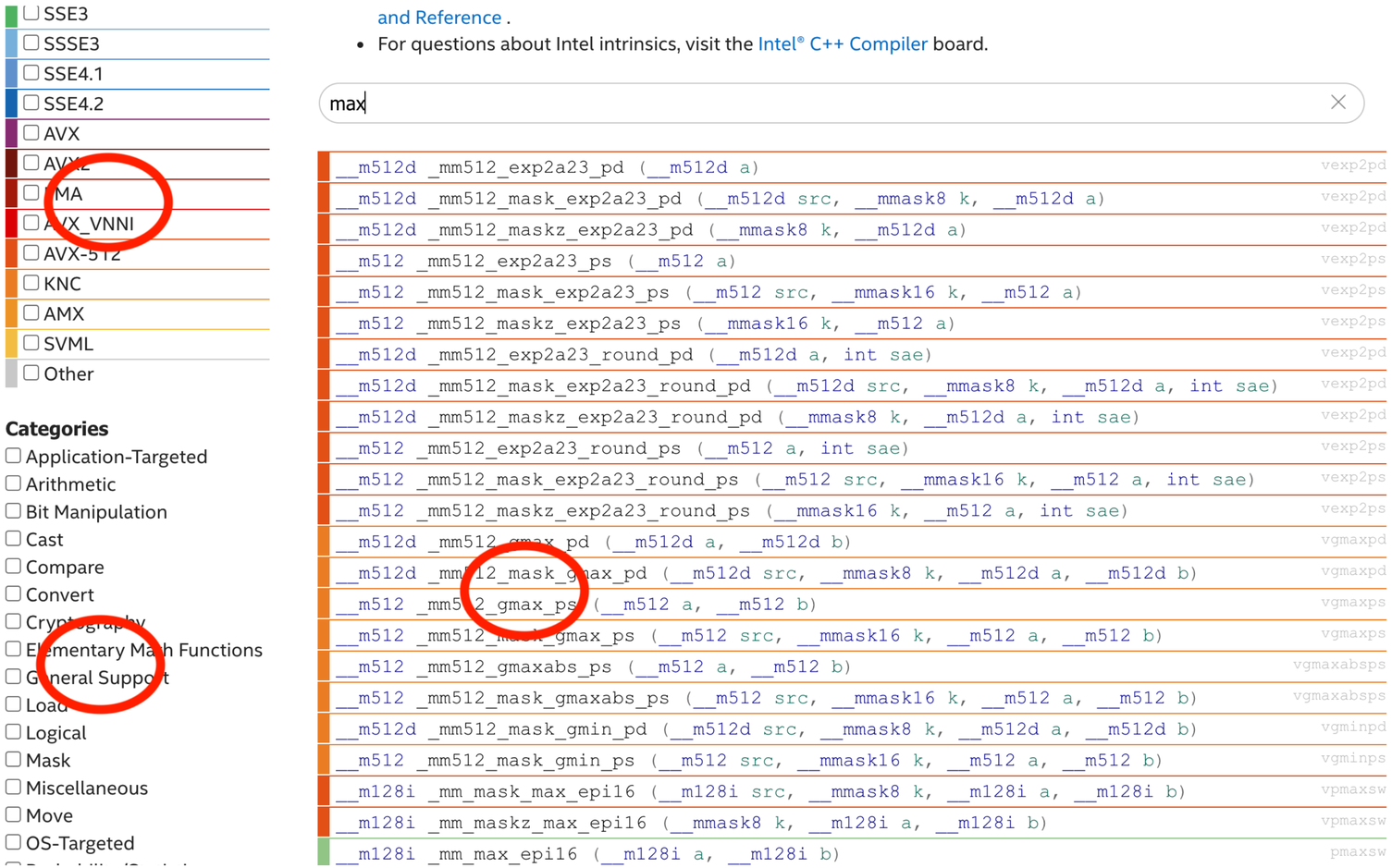
Vector Intrinsics



- Load Store
- ADD, SUB, MUL, DIV, SQRT, MAX, MIN
- AND, OR, XOR, ANDN, ANDPS, ANDNPS
- ==,<,<=,>,>=,!=

- ps
- pd
- epi8/epi16/epi32/epi64
- epu8/epu16/epu32/epu64

Vector Intrinsics



https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

Vector Intrinsics Examples： HLL Merge

```
for (int i = 0; i < HLL_REGISTERS_COUNT; i++) {
    _registers.data[i] = std::max(_registers.data[i], other_registers[i]);
}
```

- Alignment
- Tail Process
- Compatibility
- Only Simple Operation

```
int loop = HLL_REGISTERS_COUNT / 32;
uint8_t* dst = _registers.data;
const uint8_t* src = other_registers;
for (int i = 0; i < loop; i++) {
    __m256i xa = _mm256_loadu_si256((const __m256i*)dst);
    __m256i xb = _mm256_loadu_si256((const __m256i*)src);
    _mm256_storeu_si256((__m256i*)dst, _mm256_max_epu8(xa, xb));
    src += 32;
    dst += 32;
}
```

6X Performance Improvement

StarRocks Database Vectorized

1 Database Performance

2 Vectorized Basic

03

Database Vectorized

4 Thinking Vectorized

The Challenge Of Database Vectorized

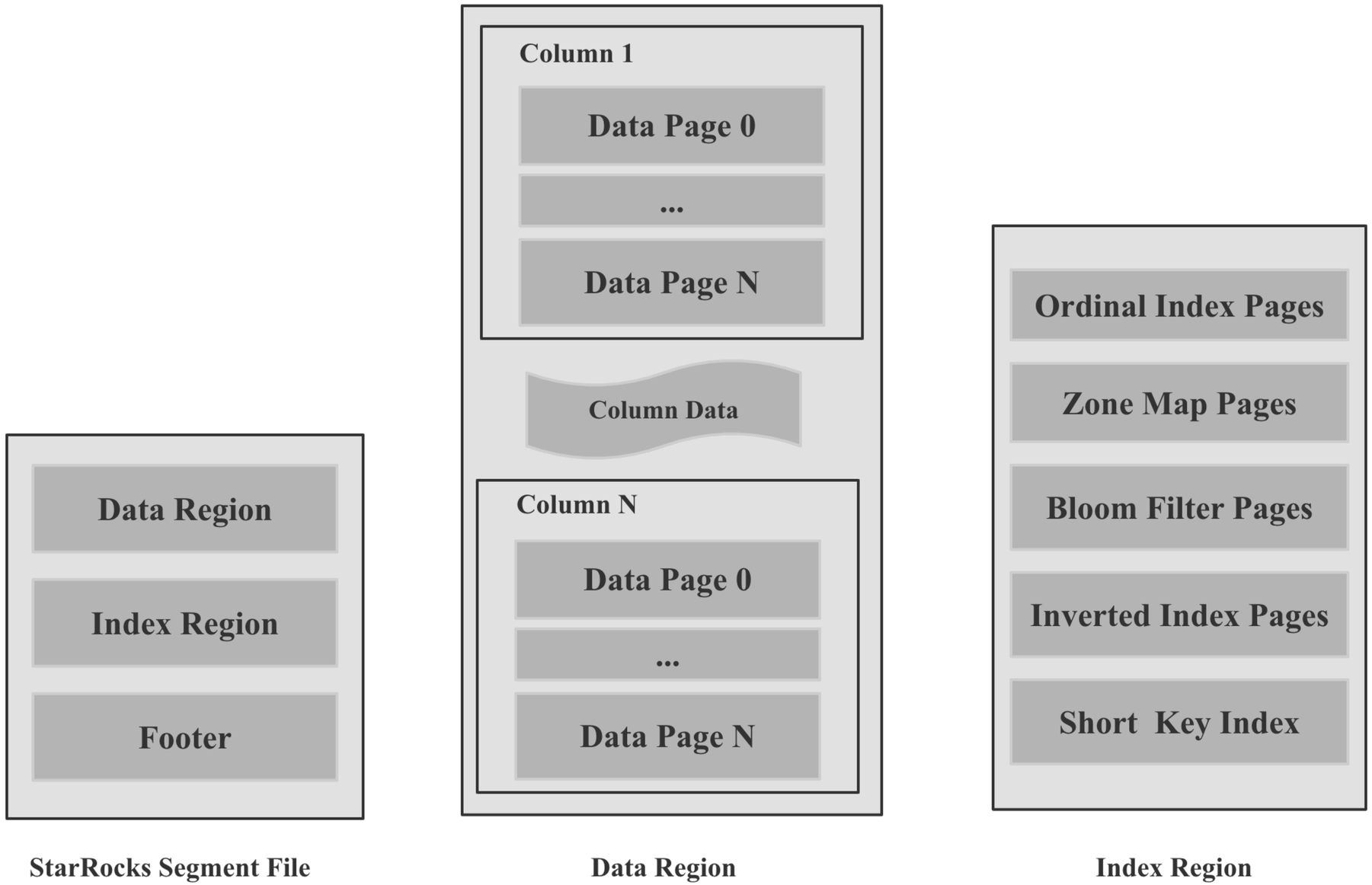
Not Only The CPU SIMD

It's A Huge Performance Improvement Project Based On CPU

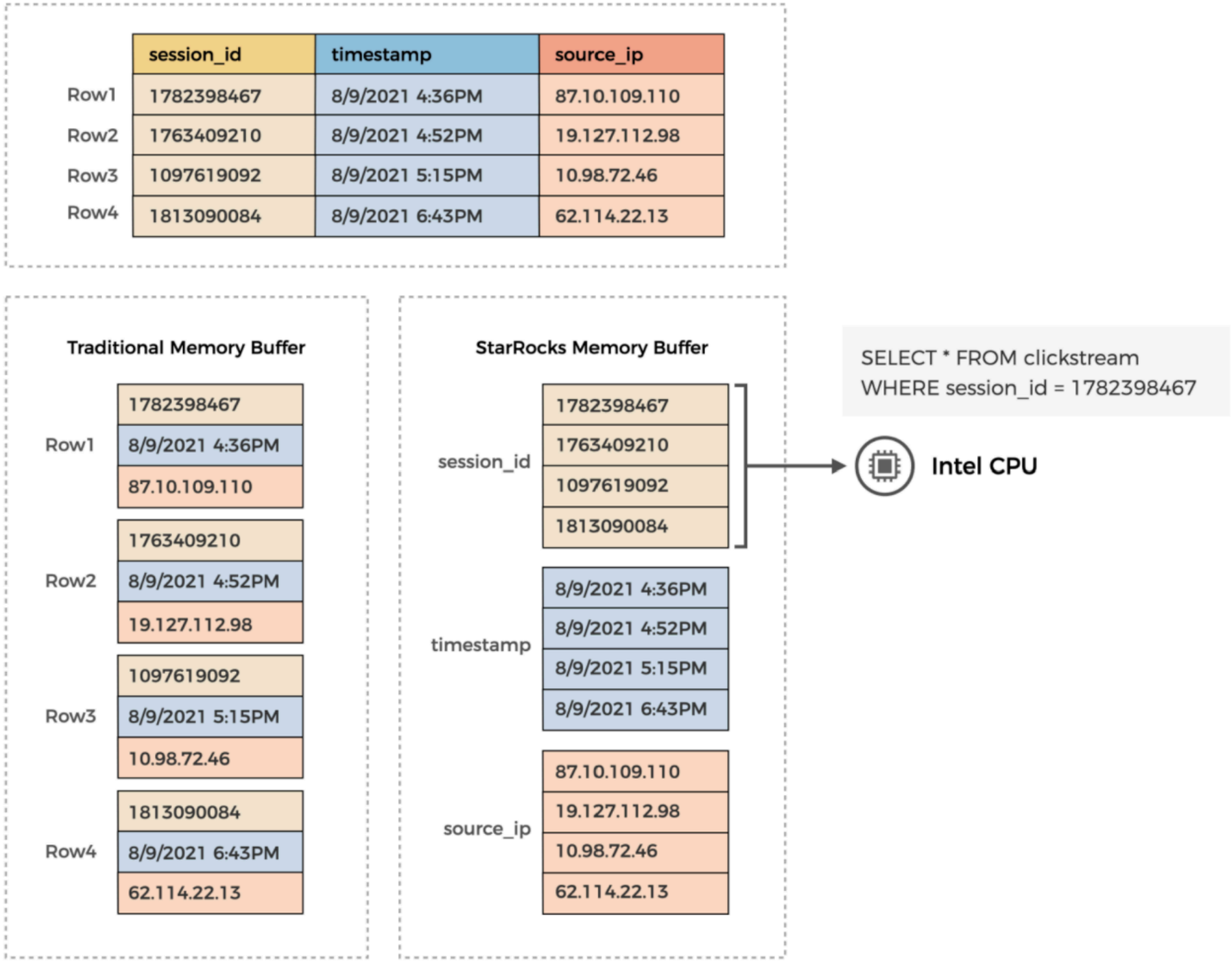
The Challenge Of Database Vectorized

- Data Use **Column Layout** All Time: Disk, Memory, NetWork
- **All Operations** Need To Be Vectorized
- **All Expressions** Need To Be Vectorized
- Use **SIMD** Instructions As Much As Possible
- Redesign **Memory Manage**
- Redesign **Data Structure**
- **Overall Performance** Improve 5x —— All Operations And Expressions Need Improve 5X

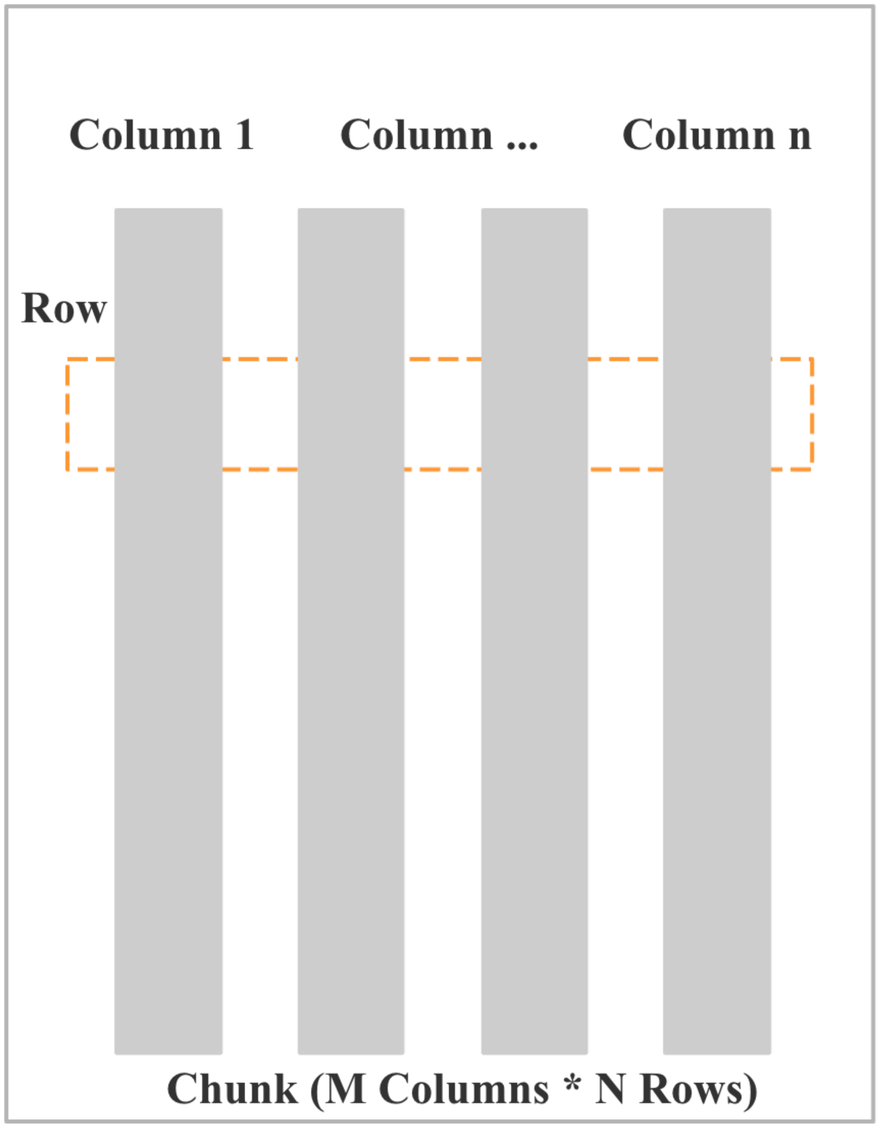
Database Vectorized： Disk Column Store



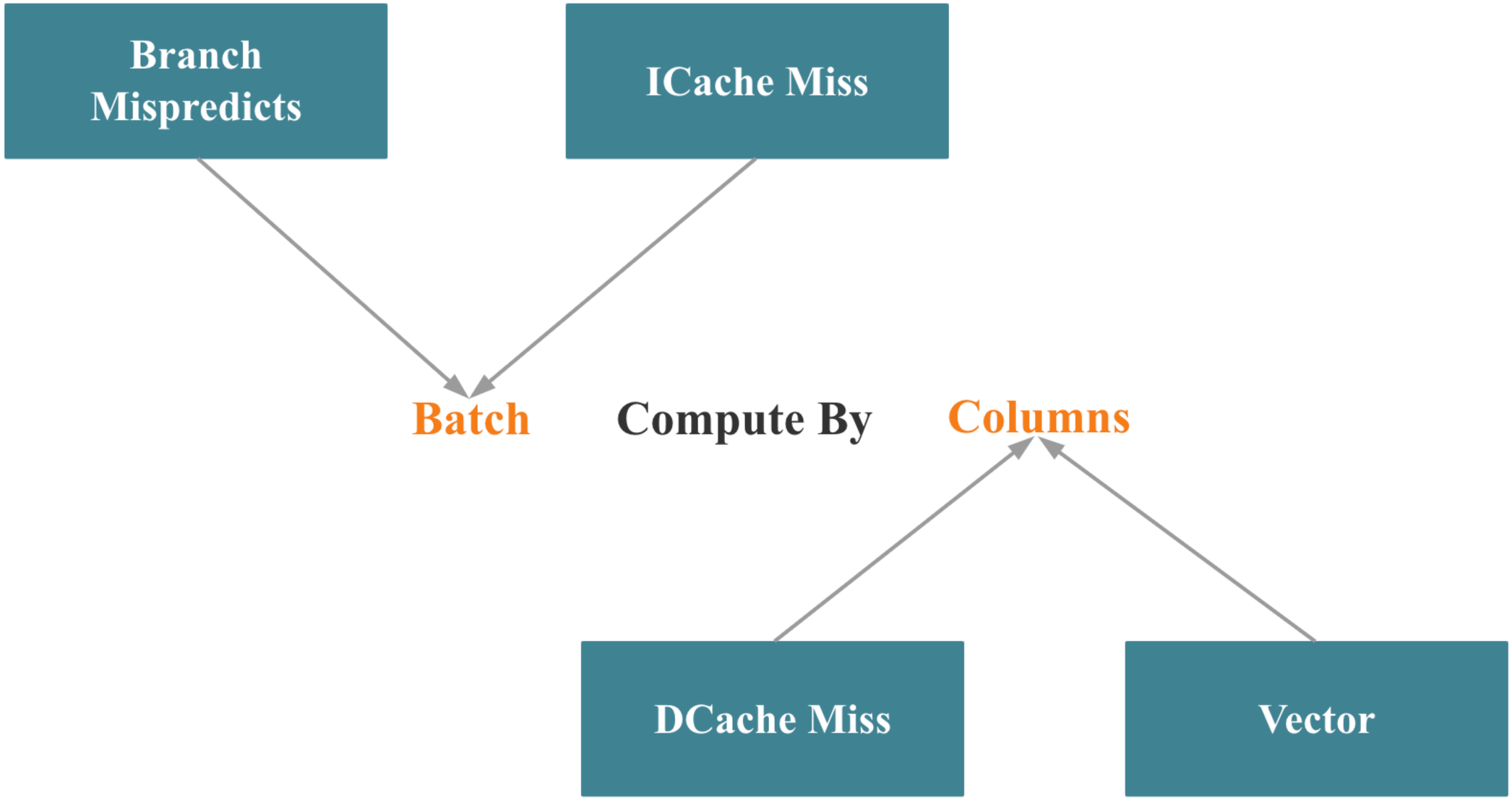
Database Vectorized： Memory Column Layout



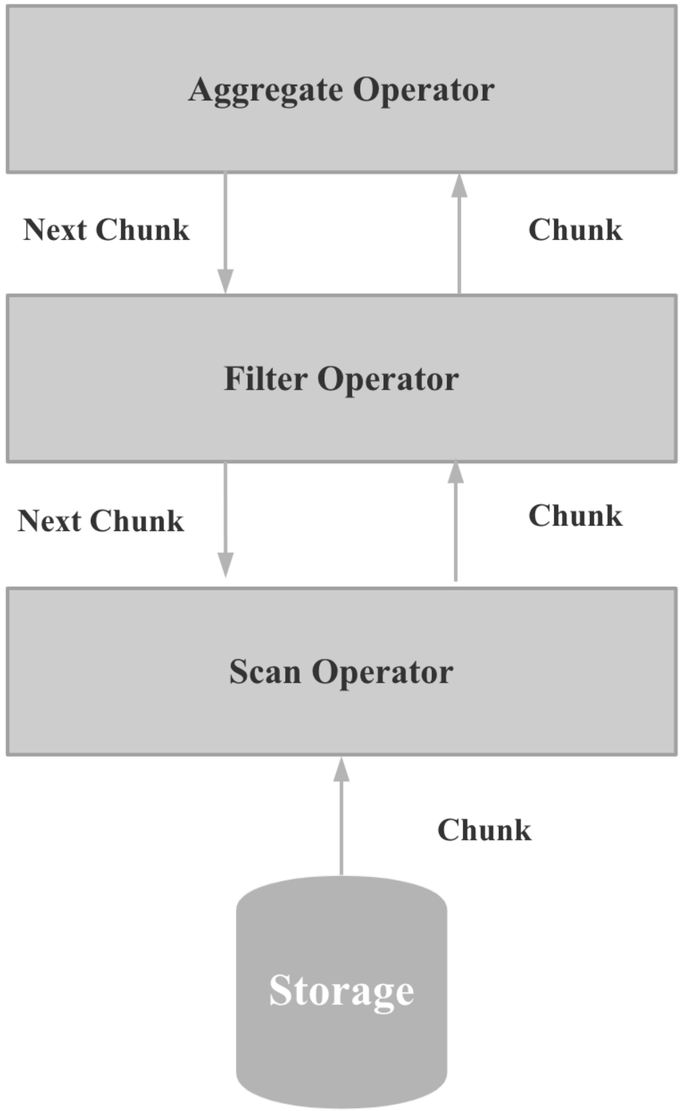
Database Vectorized： Column Layout



Database Vectorized —— Operators & Expresses

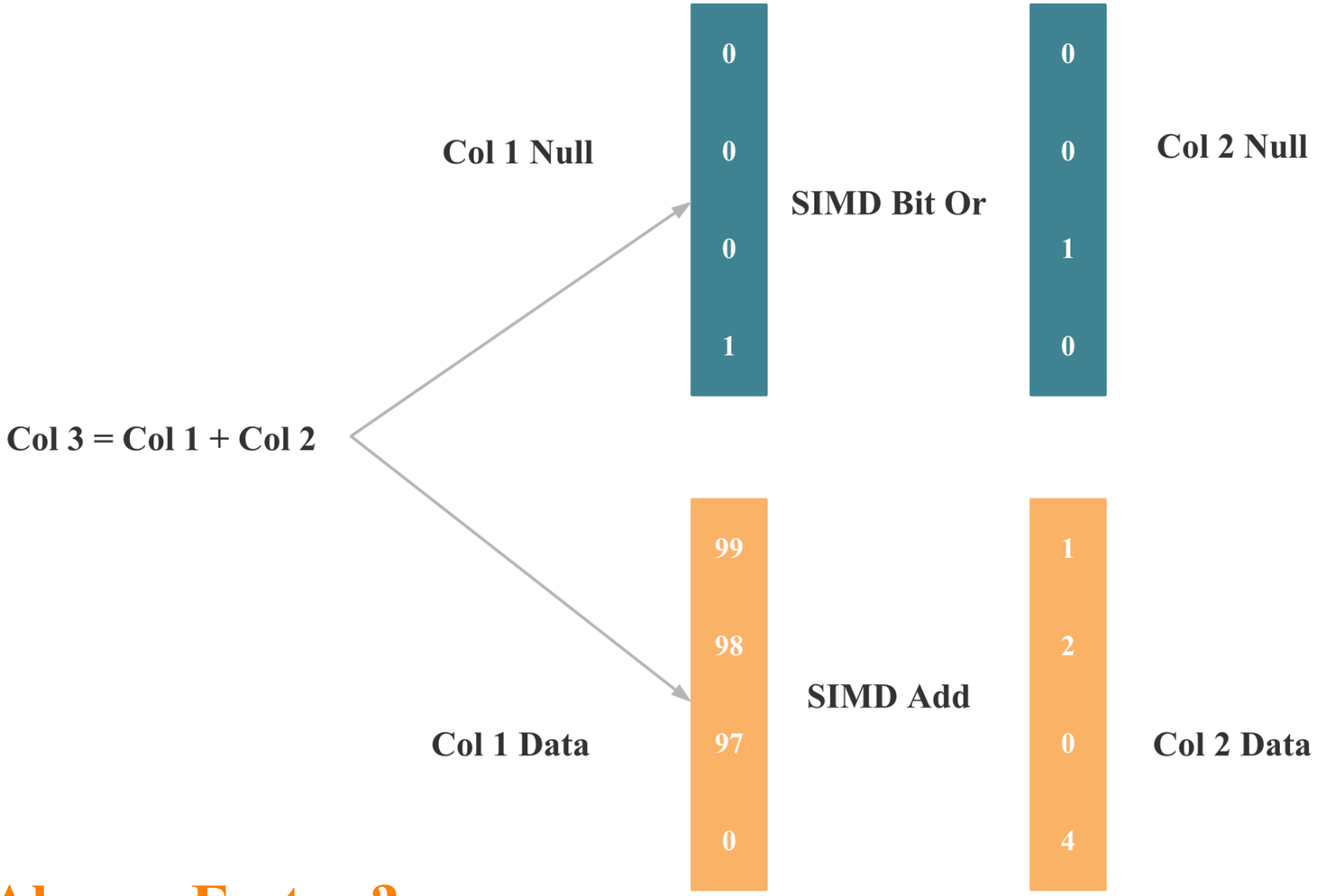


Database Vectorized —— Operator



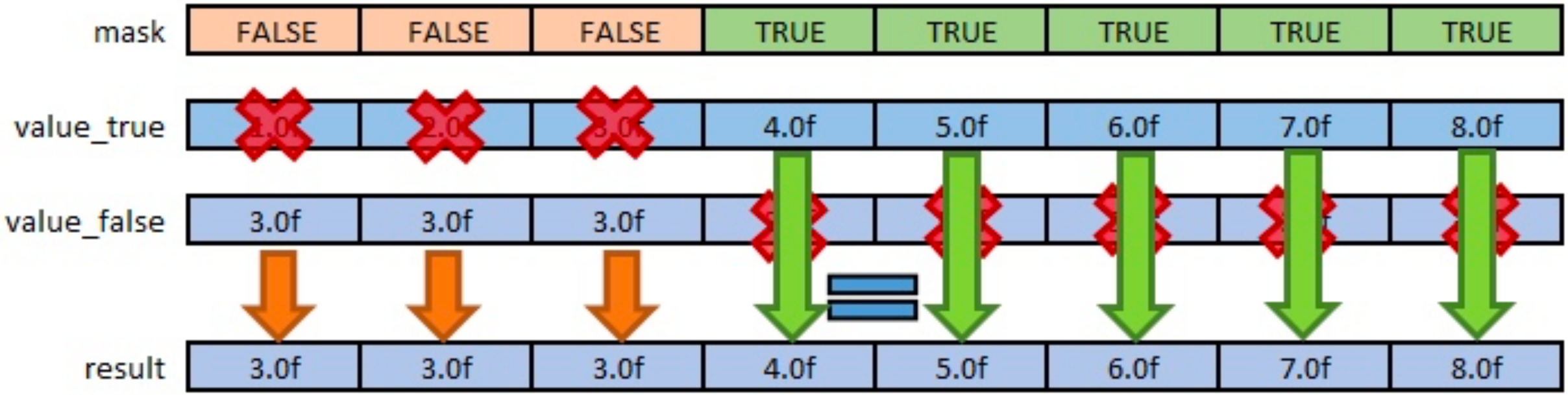
Database Vectorized —— Express Compute

Input Null — Output Null



This Way Always Faster ?

Database Vectorized： SIMD Branch



Compute Both Branches + Select

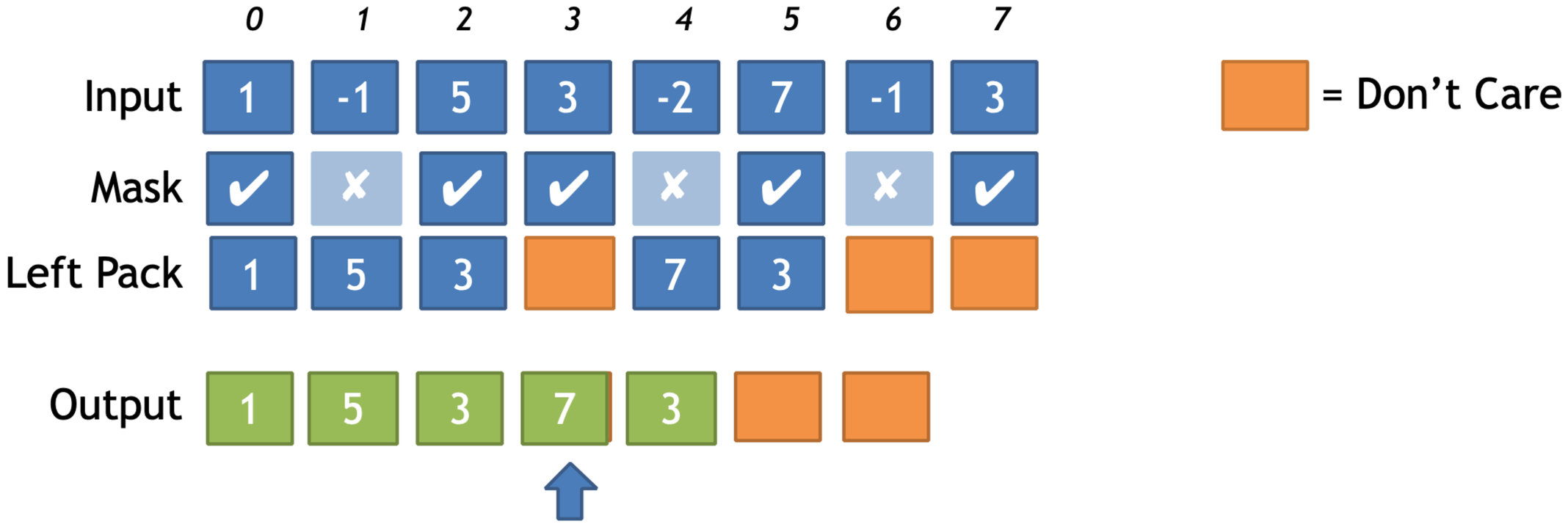
Branchless Programming in C++ - Fedor Pikus - CppCon 2021　Great Talk

Database Vectorized —— SIMD Filter

```
for (int i = 0; i < count; ++i) {
  if (input[i] >= limit)
    *outputp++ = input[i];
}
```

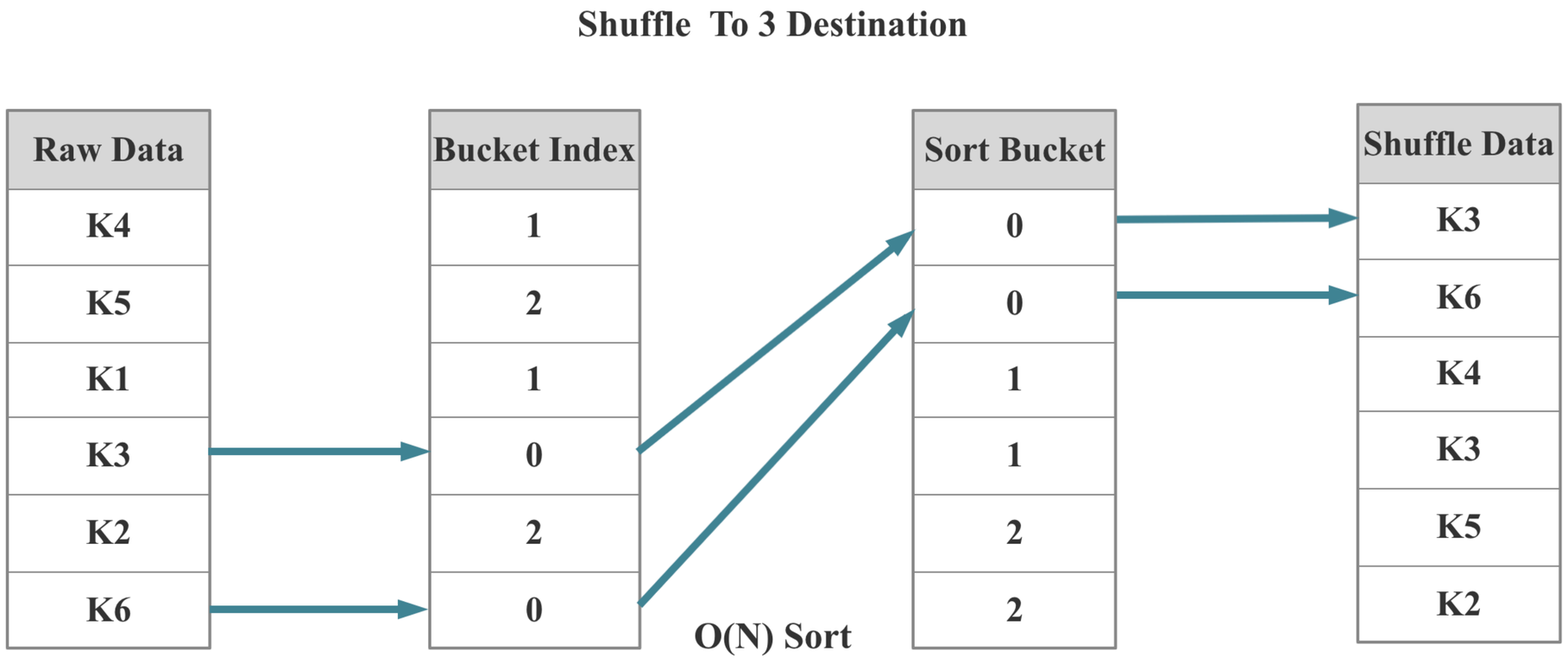
```
for (int i = 0; i < count; i += 4) {
  __m128 val      = _mm_load_ps(input + i);
  __m128 mask     = _mm_cmpge_ps(val, _mm_set1_ps(limit));

  __m128 result   = LeftPack(mask, val);

  _mm_storeu_ps(output, result);

  output += _popcnt(_mm_movemask_ps(mask));
}
```

Database Vectorized —— SIMD Filter



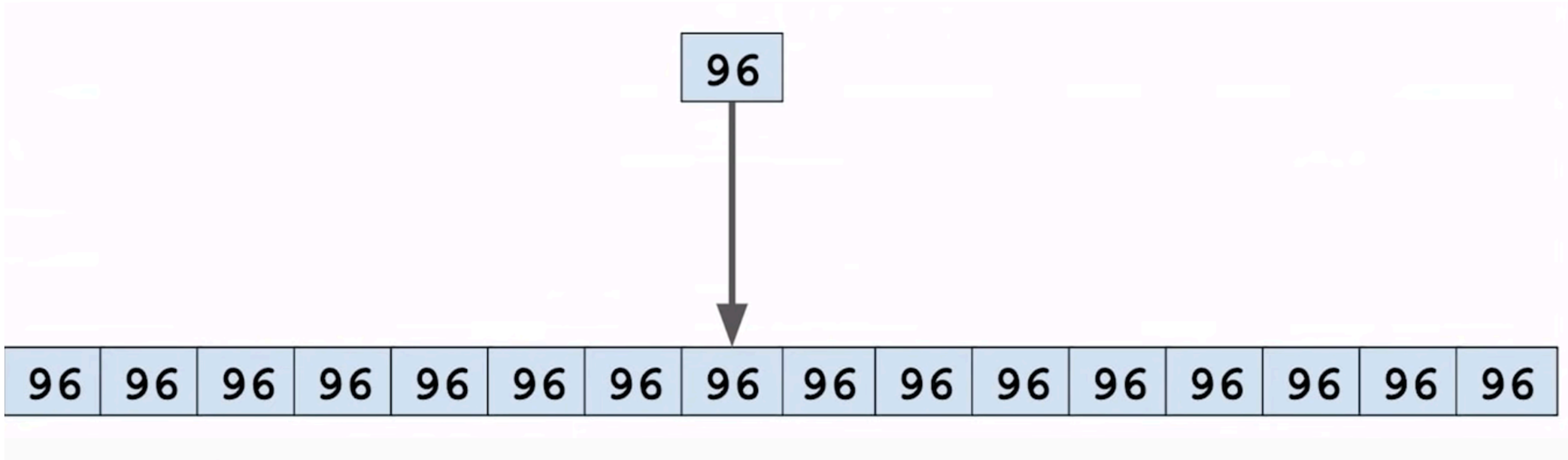
Database Vectorized： Shuffle By Column

- Efficient Compress
- Efficient Serde
- Efficient Compute



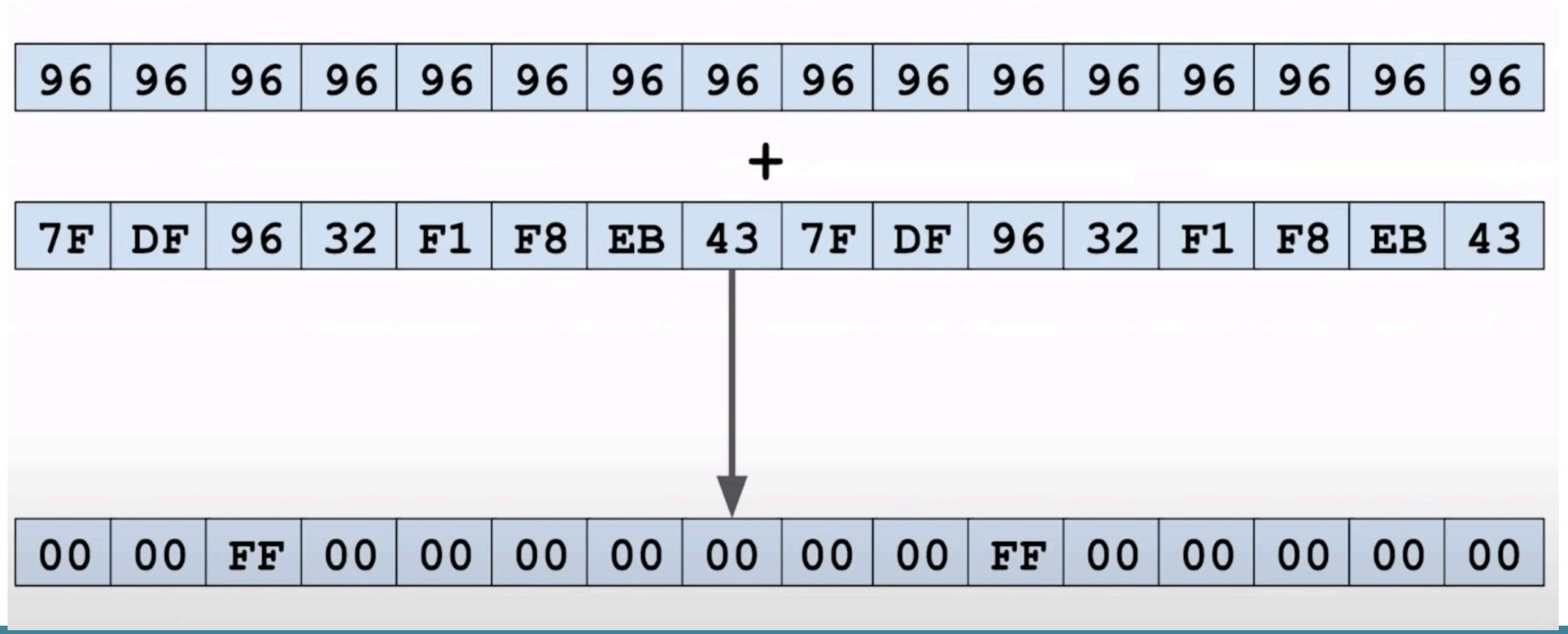
Database Vectorized： Hash Aggregate

```
BitMask<uint32_t, kWidth> Match(h2_t hash) const {
    auto match = _mm_set1_epi8((char)hash);
    return BitMask<uint32_t, kWidth>(
        _mm_movemask_epi8(_mm_cmpeq_epi8(match, ctrl)));
}
```



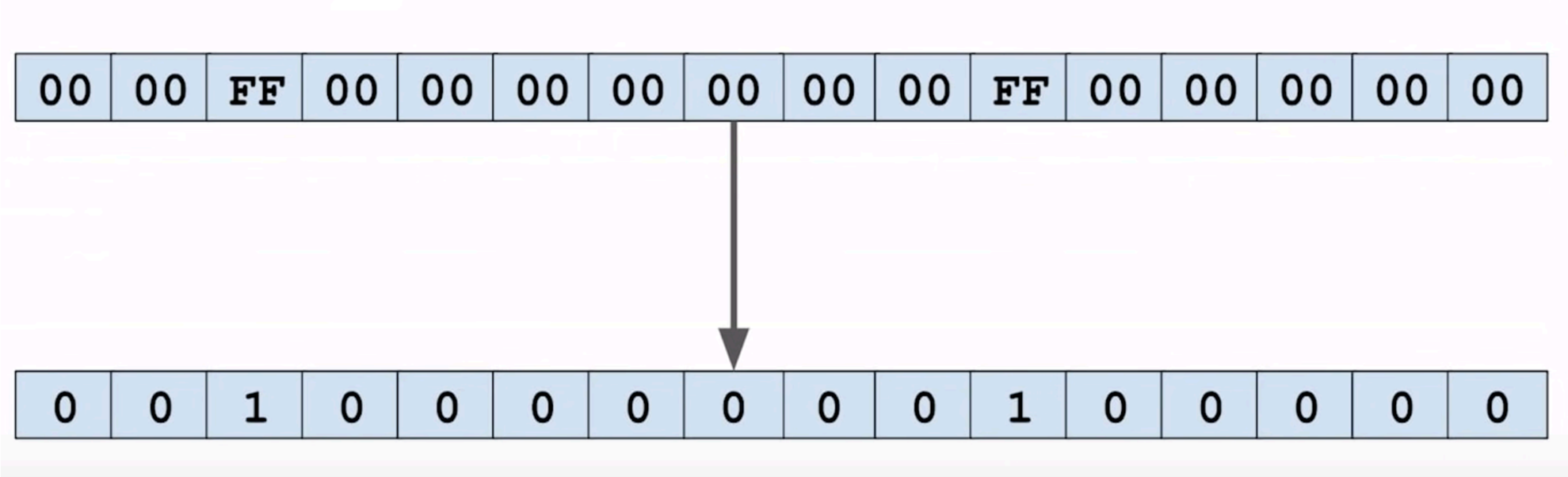
Database Vectorized： Hash Aggregate

```
BitMask<uint32_t, kWidth> Match(h2_t hash) const {
    auto match = _mm_set1_epi8((char)hash);
    return BitMask<uint32_t, kWidth>(
        _mm_movemask_epi8(_mm_cmpeq_epi8(match, ctrl)));
}
```



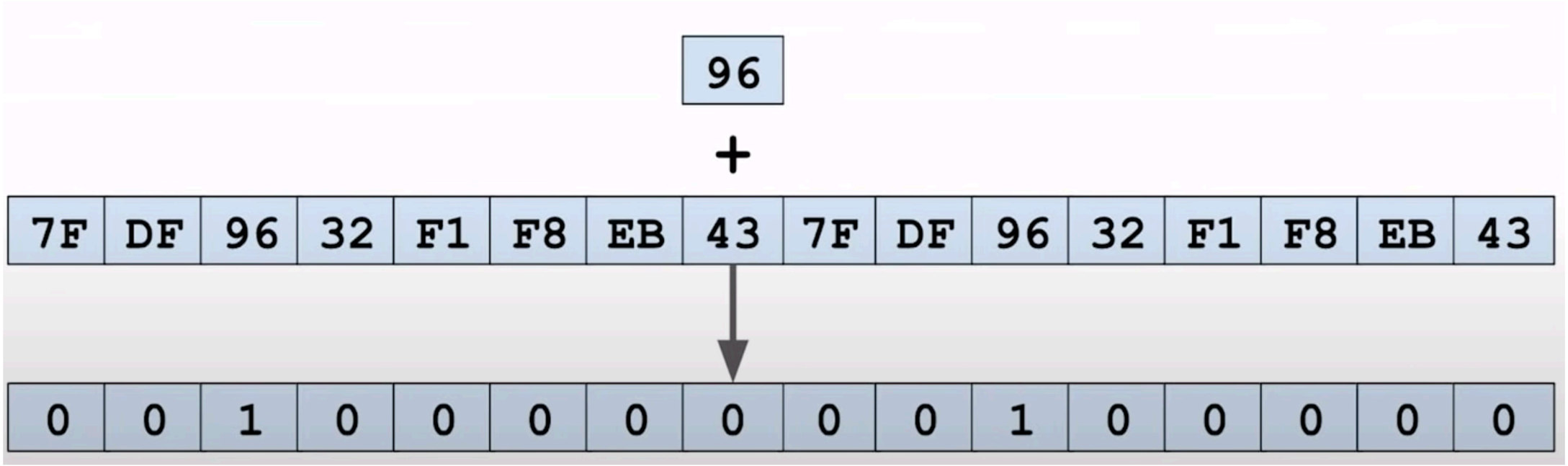
Database Vectorized： Hash Aggregate

```
BitMask<uint32_t, kWidth> Match(h2_t hash) const {
    auto match = _mm_set1_epi8((char)hash);
    return BitMask<uint32_t, kWidth>(
        _mm_movemask_epi8(_mm_cmpeq_epi8(match, ctrl)));
}
```



Database Vectorized： Hash Aggregate

```
BitMask<uint32_t, kWidth> Match(h2_t hash) const {
    auto match = _mm_set1_epi8((char)hash);
    return BitMask<uint32_t, kWidth>(
        _mm_movemask_epi8(_mm_cmpeq_epi8(match, ctrl)));
}
```



Database Vectorization —— Hash Join

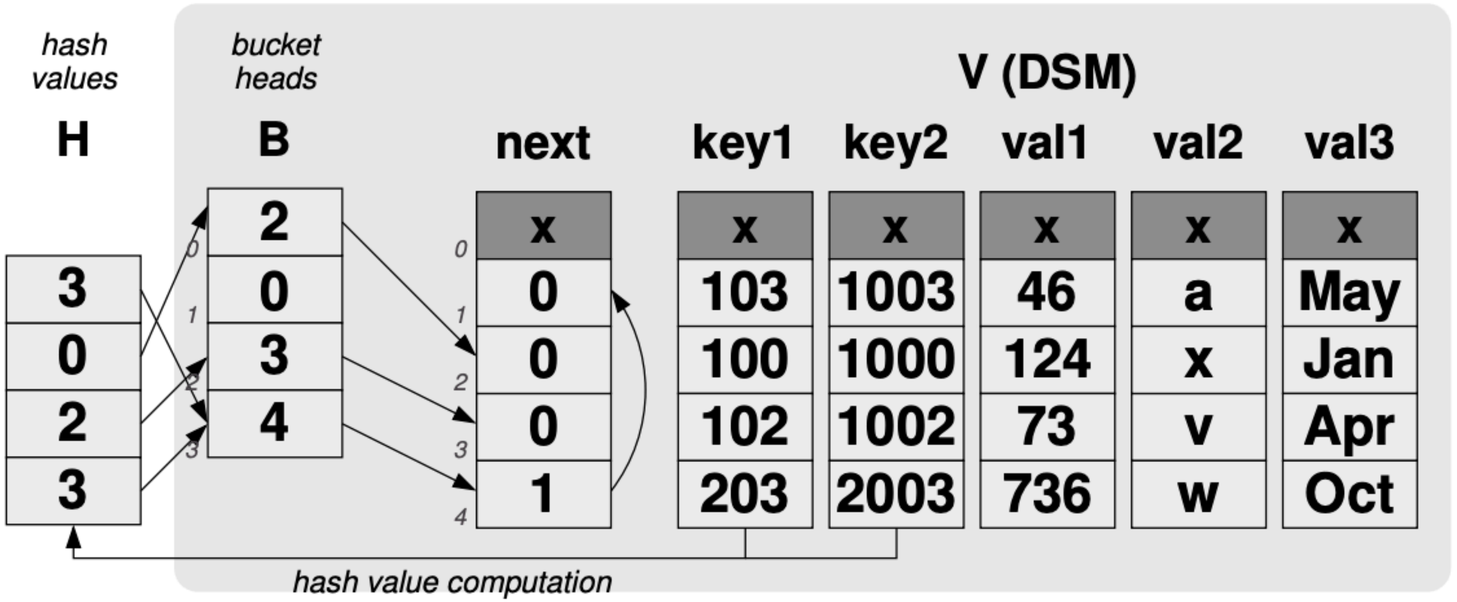
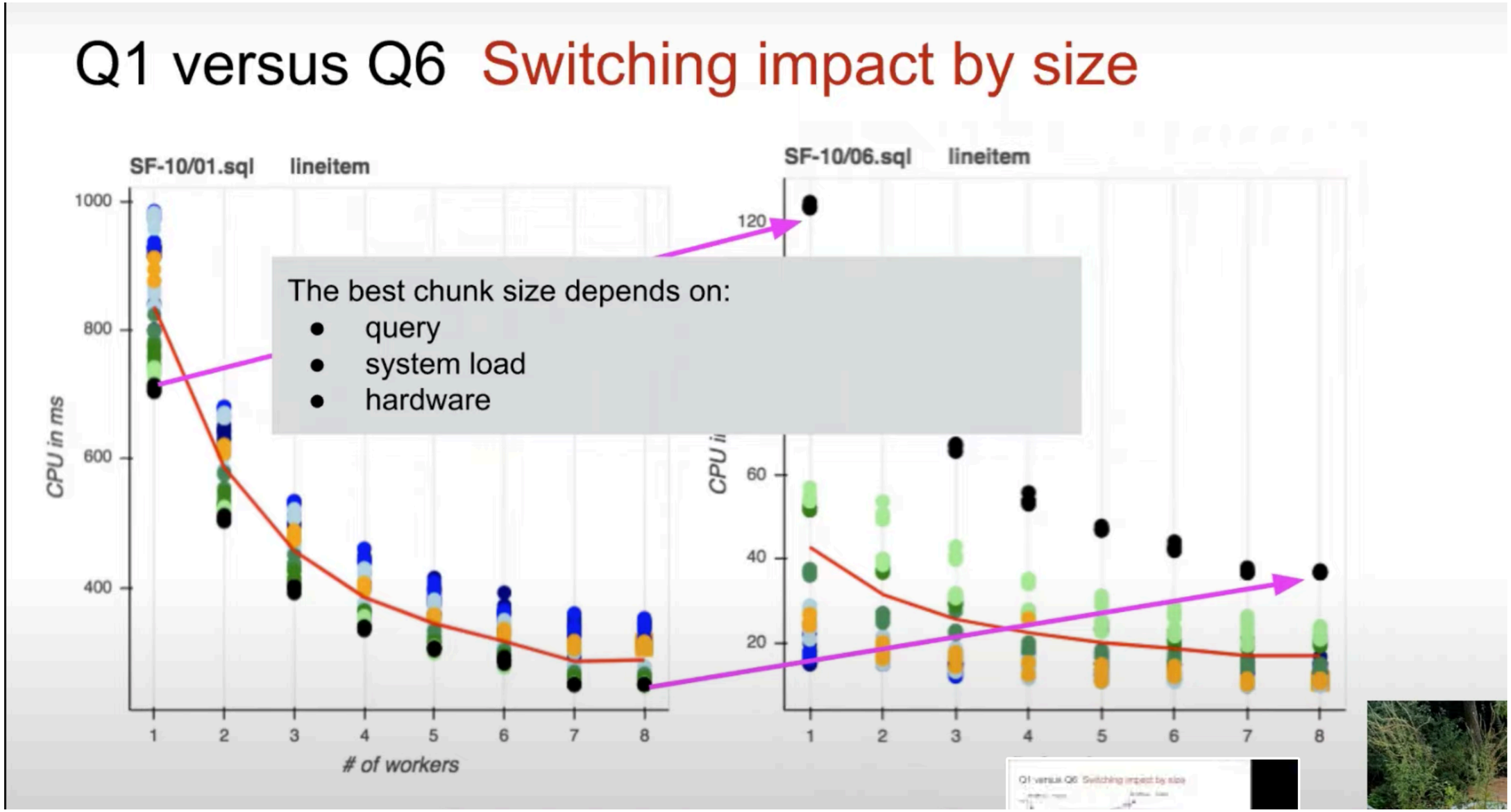


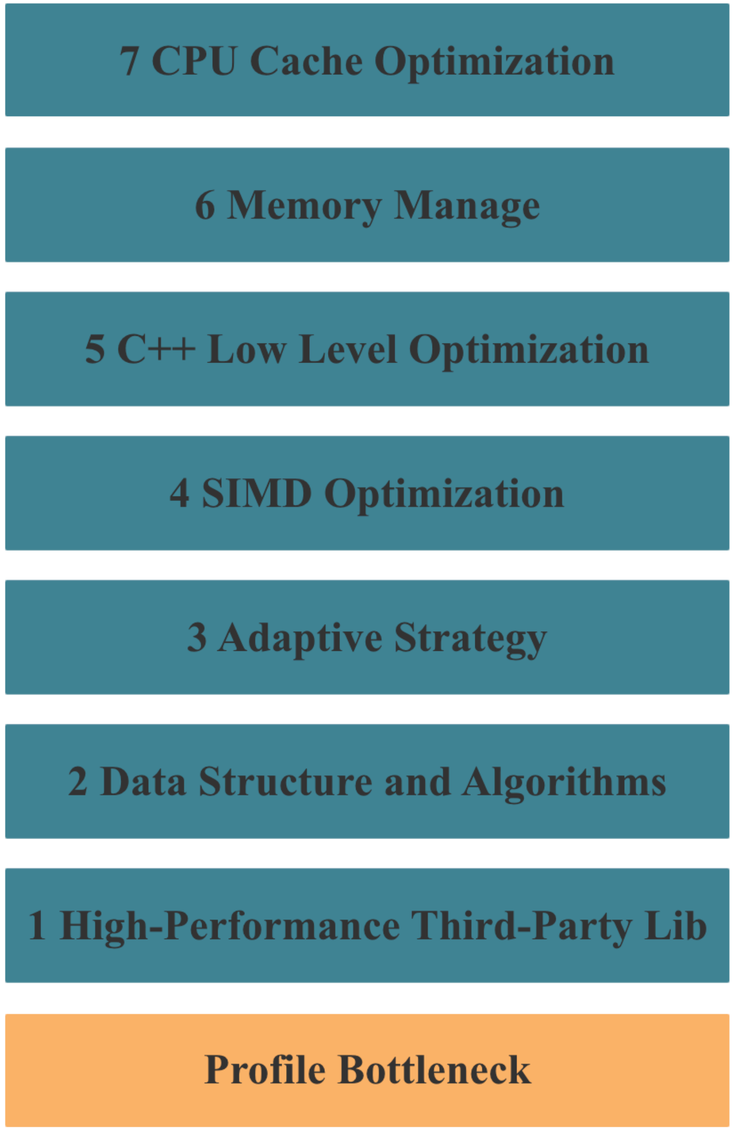
Figure 3: Bucket-chain hash table as used in VectorWise. The value space V presented in the figure is in DSM format, with separate array for each attribute. It can also be implemented in NSM, with data stored tuple-wise.

Balancing vectorized query execution with bandwidth-optimized storage

Database Vectorized —— Chunk Size



Database Vectorized Improvement Methods



1 High-Performance Third-Party Lib： Parallel Hashmap

- Parallel Hashmap
- Fmt
- Simdjson
- Hyperscan

Merged **[DSDB-3570] Improve Hll aggregate column performance** #3260

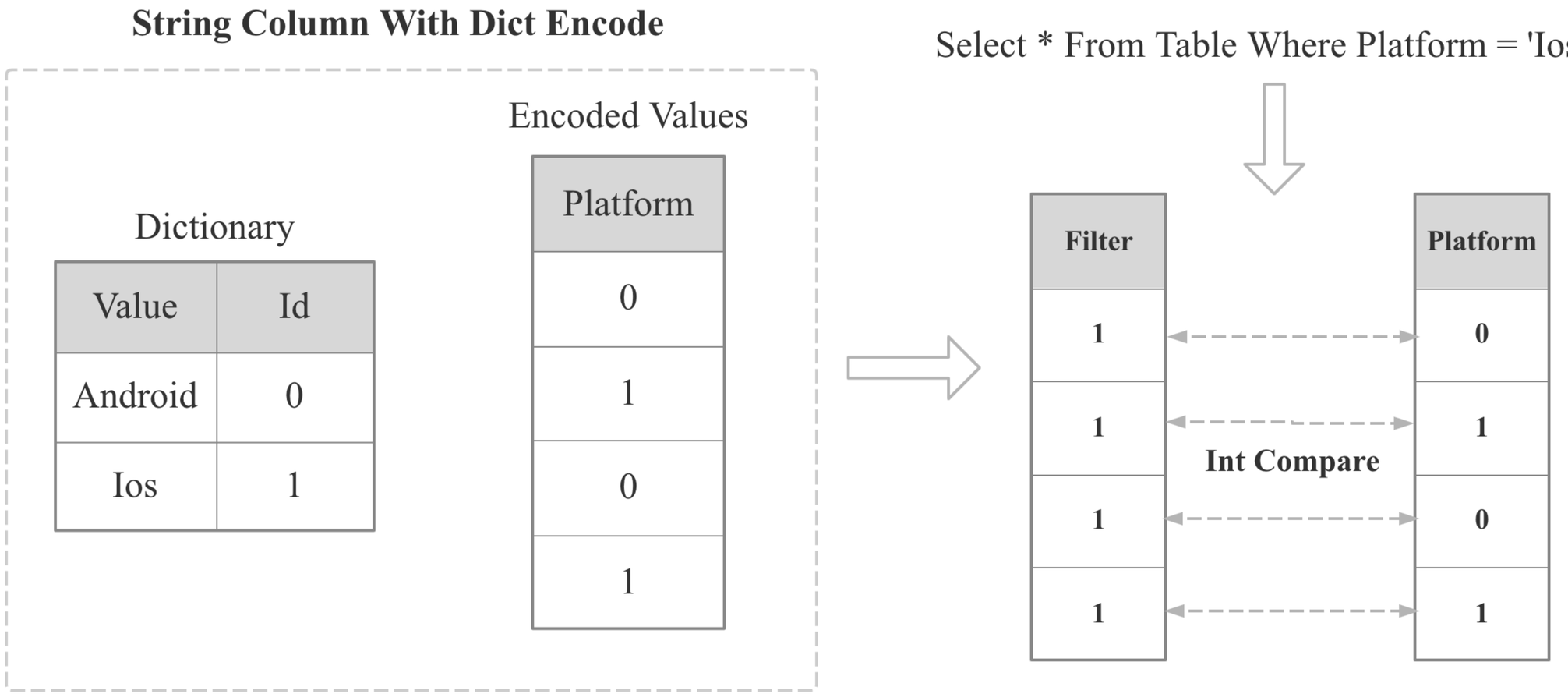
Changes from **all commits** ▾ **File filter** ▾ **Conversations** ▾ **Jump to** ▾

3 be/src/olap/hll.h

```
30  31   namespace doris {
31  32
    @@ -141,7 +142,7 @@ class HyperLogLog {
41  142
42  143  private:
43  144      HllDataType _type = HLL_DATA_EMPTY;
44      -    std::set<uint64_t> _hash_set;
    145  +    phmap::flat_hash_set<uint64_t> _hash_set;
45  146
46  147      // This field is much space consumming(HLL_REGISTERS_COUNT)
47  148      // it only when it is really needed.
48  149      std::vector<uint8_t> _registers;
```

3X Performance Improvement

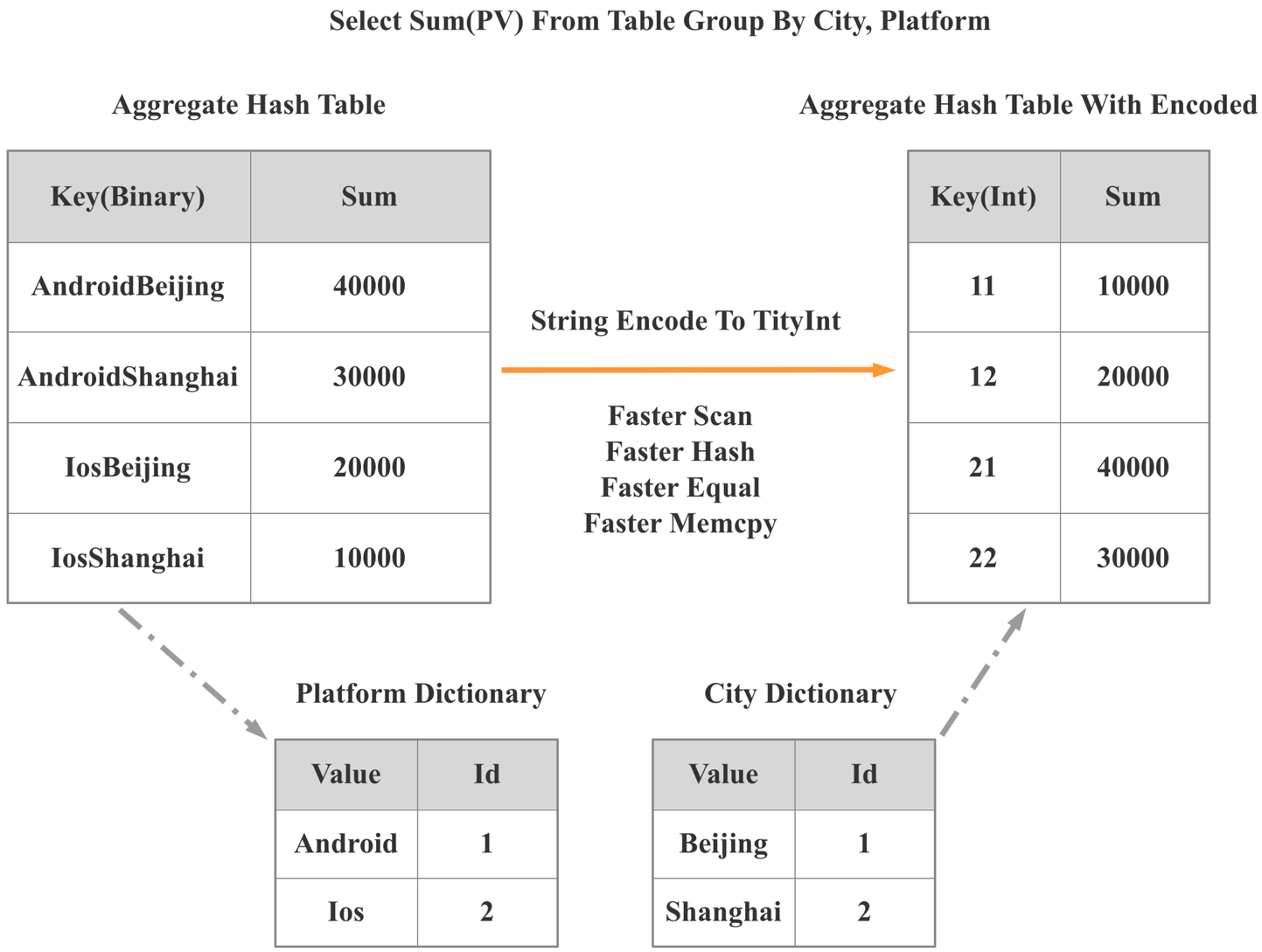
2 Data Structure and Algorithms： Operations On Encoded Data



Int Compare is Very Faster Than String

2 Data Structure and Algorithms： Operations On Encoded Data

- Scan
- Filter
- Agg
- Sort
- Join
- String Functions



3X Performance Improvement For Aggregate

3 Adaptive Strategy： Join Runtime Filter Compute

```
double selectivity = true_count * 1.0 / chunk_size;
if (selectivity <= 0.5) {      // useful filter
    if (selectivity < 0.05) { // very useful filter, could early return
        _selectivity.clear();
        _selectivity.emplace(selectivity, rf_desc);
        chunk->filter(new_selection);
        return;
    }

    // Only choose three most selective runtime filters
    if (_selectivity.size() < 3) {
        _selectivity.emplace(selectivity, rf_desc);
    } else {
```

Prefer The Low Selectivity Filter

4 SIMD Optimization： Improve Ascii Substring

```
char tail_has_error = 0;
for (size_t i = 0; i < len; i++) {
    tail_has_error |= src[i];
}
return !(tail_has_error & 0x80);
```

Validate Ascii String

Ascii Chars From 0x00 To 0x7F

```
__m256i has_error = _mm256_setzero_si256();
if (len >= 32) {
    for (size_t i = 0; i <= len - 32; i += 32) {
        __m256i current_bytes = _mm256_loadu_si256((const __m256i*
        has_error = _mm256_or_si256(has_error, current_bytes);
    }
}
int error_mask = _mm256_movemask_epi8(has_error);
```

5X Performance Improvement

5 C++ Low level Optimization

- Inline
- Loop Optimization
- Unrolling
- Resize No Initialize
- Copy To Move
- Std::vector
- Compile-time Computation

5 C++ Low level Optimization： Remove Unnecessary Copy

```
void serialize_to_column(FunctionContext* ctx, ConstAggDataPt
    BitmapValue bitmap = this->data(state);
    BitmapColumn* col = down_cast<BitmapColumn*>(to);
    col->append(&bitmap);
}
```

Reduce 2 Copy

```
void serialize_to_column(FunctionContext* ctx, ConstAggDataPtr __rest
    BitmapColumn* col = down_cast<BitmapColumn*>(to);
    BitmapValue& bitmap = const_cast<BitmapValue&>(this->data(state))
    col->append(std::move(bitmap));
}
```

1X Performance Improvement

6 Memory Manage： HLL Memory Manage

- Allocate By Chunk
- Reuse Memory

```
+ HyperLogLog::~HyperLogLog() {
+     if (_registers.data != nullptr) {
+         ChunkAllocator::instance()->free(_registers);
+     }
+ }
+
  // Convert explicit values to register format, and clear explicit values.
  // NOTE: this function won't modify _type.
  void HyperLogLog::_convert_explicit_to_register() {
      DCHECK(_type == HLL_DATA_EXPLICIT)
              << "_type(" << _type << ") should be explicit(" << HLL_DATA_EXPLICIT << ")";
-     _registers.clear();
-     _registers.resize(HLL_REGISTERS_COUNT, 0);
+     DCHECK_EQ(_registers.data, nullptr);
+     ChunkAllocator::instance()->allocate(HLL_REGISTERS_COUNT, &_registers);
+     memset(_registers.data, 0, HLL_REGISTERS_COUNT);
+
```

5X Performance Improvement

7 CPU Cache Optimization: Why

Table 2.2 Example Time Scale of System Latencies

Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50–150 µs	2–6 days
Rotational disk I/O	1–10 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1–3 s	105–317 years
OS virtualization system reboot	4 s	423 years
SCSI command time-out	30 s	3 millennia
Hardware (HW) virtualization system reboot	40 s	4 millennia
Physical system reboot	5 m	32 millennia

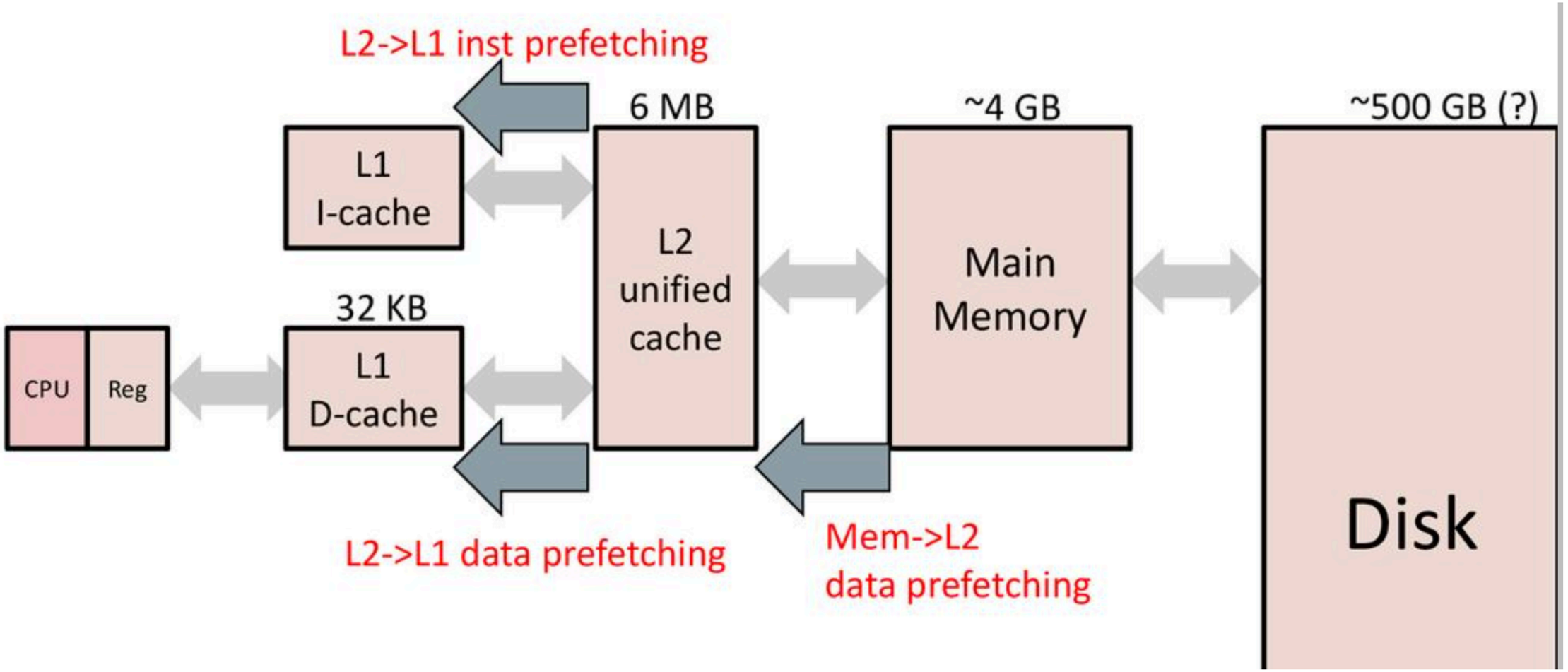
7 CPU Cache Optimization: Why

The Performance Bottleneck Will Vary

7 CPU Cache Optimization

- Improve Locality (Spatial And Temporal)
- Align The Code And Data
- Reduce Memory Footprint
- Block
- Prefetch

7 CPU Cache Optimization： Hardware Prefetch



Hardware Prefetch

7 CPU Cache： Hash GroupBy

```
template <typename Func>
void compute_agg_states(size_t chunk_size, const Columns& key_colum
                        Buffer<AggDataPtr>* agg_states) {
    for (size_t i = 0; i < column->size(); i++) {
        FieldType key = column->get_data()[i];
        auto iter = hash_map.lazy_emplace(key, [&](const auto& ctor
        (*agg_states)[i] = iter->second;
    }
}
```

7 CPU Cache： Batch Hash And Prefetch Value (More Better)

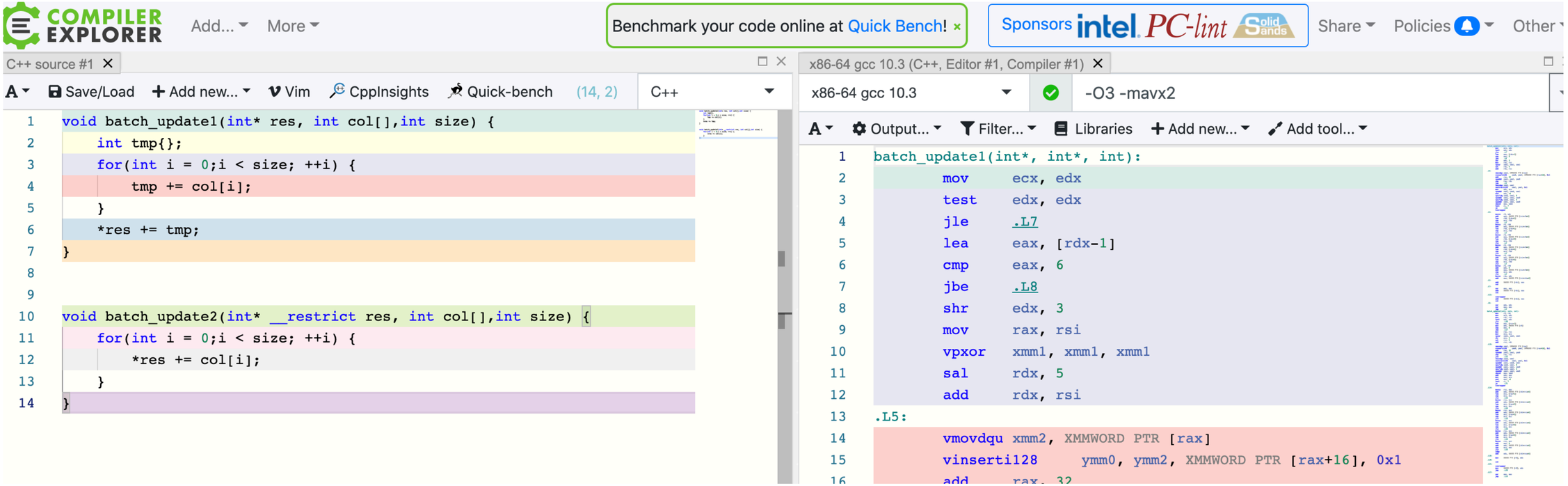
```
#define PRECOMPUTE_HASH_VALUES(column, prefetch_dist)                          \
    size_t const column_size = column->size();                                 \
    size_t* hash_values = reinterpret_cast<size_t*>(agg_states->data());       \
    {                                                                          \
        const auto& container_data = column->get_data();                       \
        for (size_t i = 0; i < column_size; i++) {                             \
            size_t hashval = hash_map.hash_function()(container_data[i]);      \
            hash_values[i] = hashval;                                          \
        }                                                                      \
    }                                                                          \
    size_t __prefetch_index = prefetch_dist;

#define PREFETCH_HASH_VALUE()                                                  \
    if (__prefetch_index < column_size) {                                      \
        hash_map.prefetch_hash(hash_values[__prefetch_index++]);               \
    }
```

```
for (size_t i = 0; i < column->size(); i++) {
PRECOMPUTE_HASH_VALUES(column, AGG_HASH_MAP_DEFAULT_PREFETCH_DIST);
for (size_t i = 0; i < column_size; i++) {
    PREFETCH_HASH_VALUE();

    FieldType key = column->get_data()[i];
    auto iter = hash_map.lazy_emplace(key, [&](const auto& ctor) { ctor(key, allocate_func()); });
    auto iter = hash_map.lazy_emplace_with_hash(key, hash_values[i],
                                                [&](const auto& ctor) { ctor(key, allocate_func());
    (*agg_states)[i] = iter->second;
}
```

- Must Be Useful
- Need Right Time
- Need Right Distance

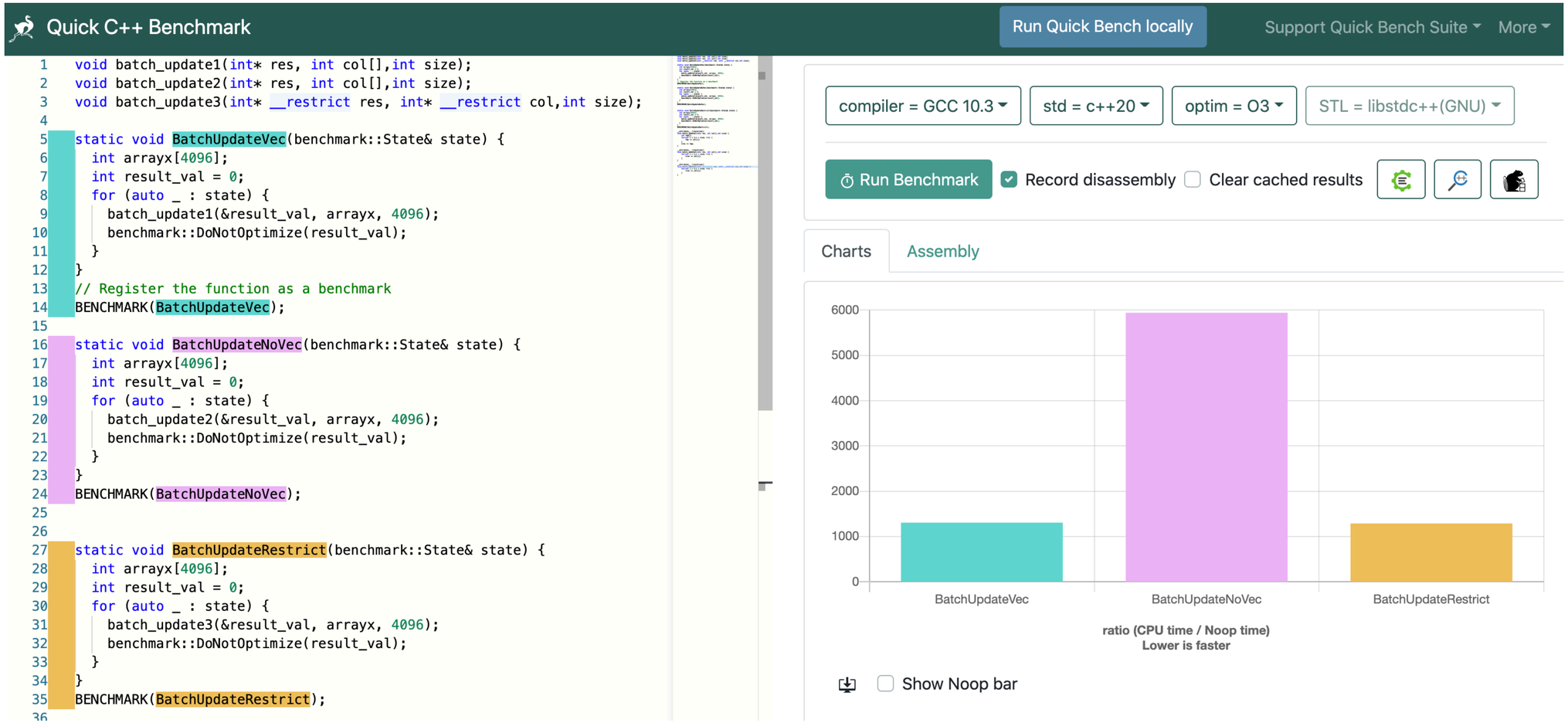
40% ~ 50% Performance Improvement

Profile Tools: godbolt.org



https://godbolt.org/

Profile Tools: quick-bench



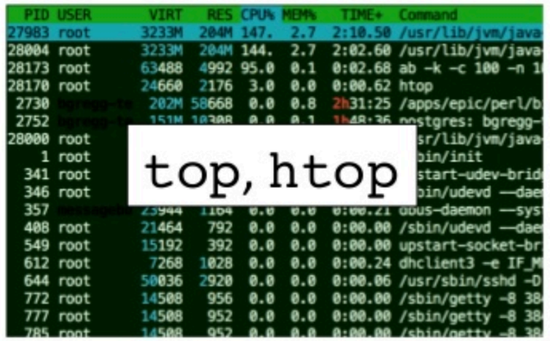

```
1  void batch_update1(int* res, int col[],int size);
2  void batch_update2(int* res, int col[],int size);
3  void batch_update3(int* __restrict res, int* __restrict col,int size);
4
5  static void BatchUpdateVec(benchmark::State& state) {
6    int arrayx[4096];
7    int result_val = 0;
8    for (auto _ : state) {
9      batch_update1(&result_val, arrayx, 4096);
10     benchmark::DoNotOptimize(result_val);
11   }
12 }
13 // Register the function as a benchmark
14 BENCHMARK(BatchUpdateVec);
15
16 static void BatchUpdateNoVec(benchmark::State& state) {
17   int arrayx[4096];
18   int result_val = 0;
19   for (auto _ : state) {
20     batch_update2(&result_val, arrayx, 4096);
21     benchmark::DoNotOptimize(result_val);
22   }
23 }
24 BENCHMARK(BatchUpdateNoVec);
25
26
27 static void BatchUpdateRestrict(benchmark::State& state) {
28   int arrayx[4096];
29   int result_val = 0;
30   for (auto _ : state) {
31     batch_update3(&result_val, arrayx, 4096);
32     benchmark::DoNotOptimize(result_val);
33   }
34 }
35 BENCHMARK(BatchUpdateRestrict);
36
```

https://quick-bench.com

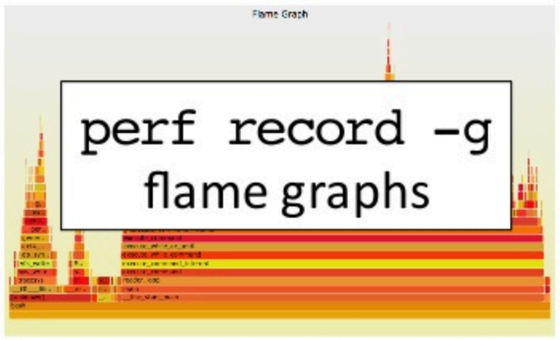
Profile Tools: Perf

CPU Tools

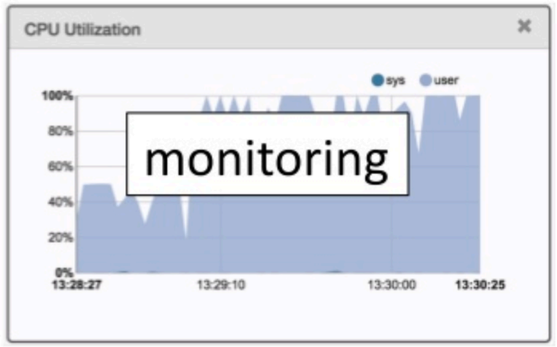
Who



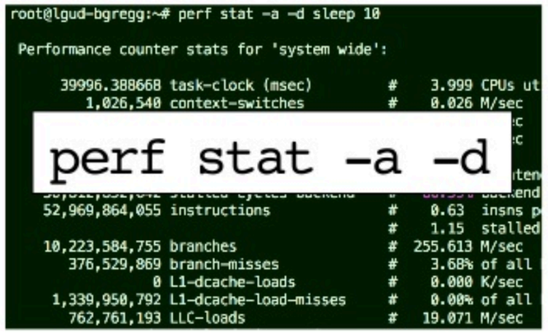
Why



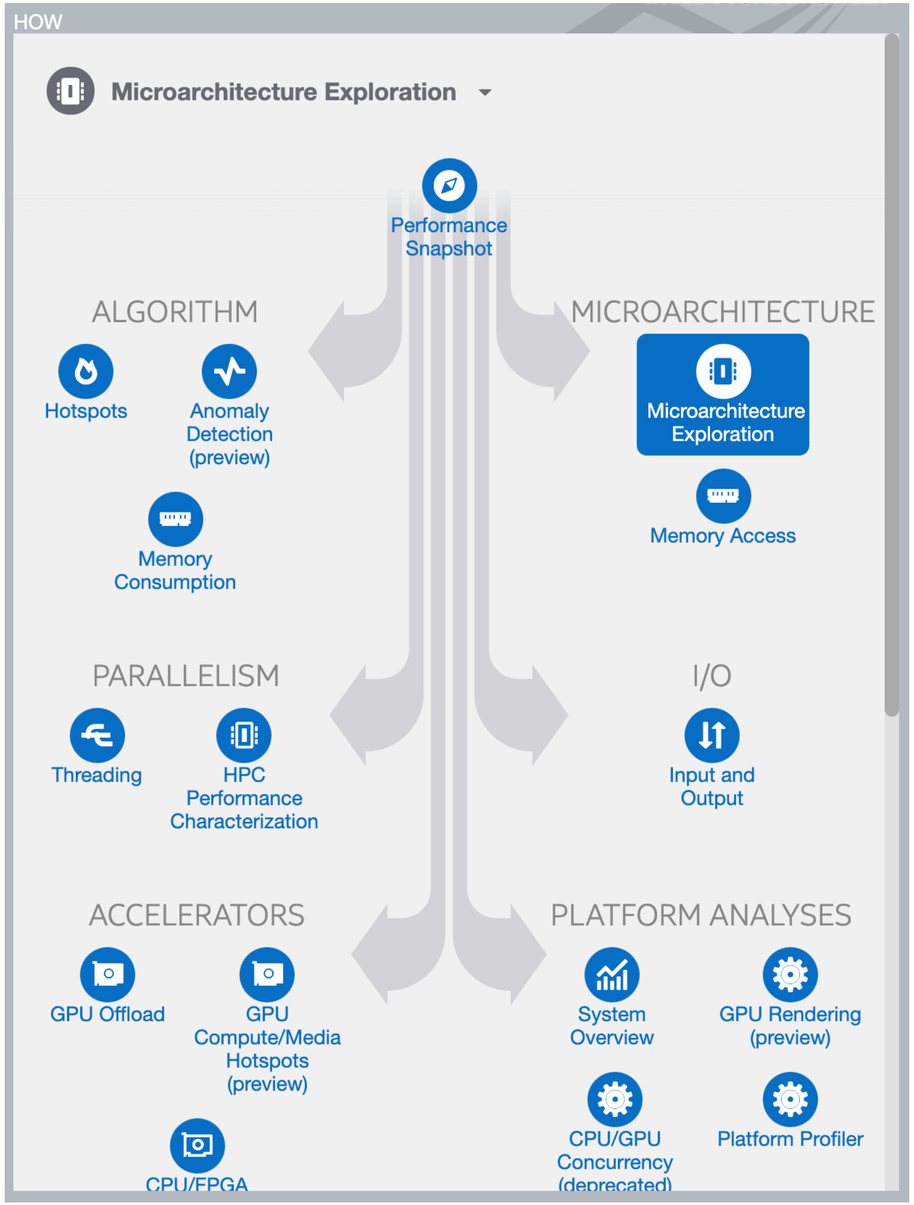
How



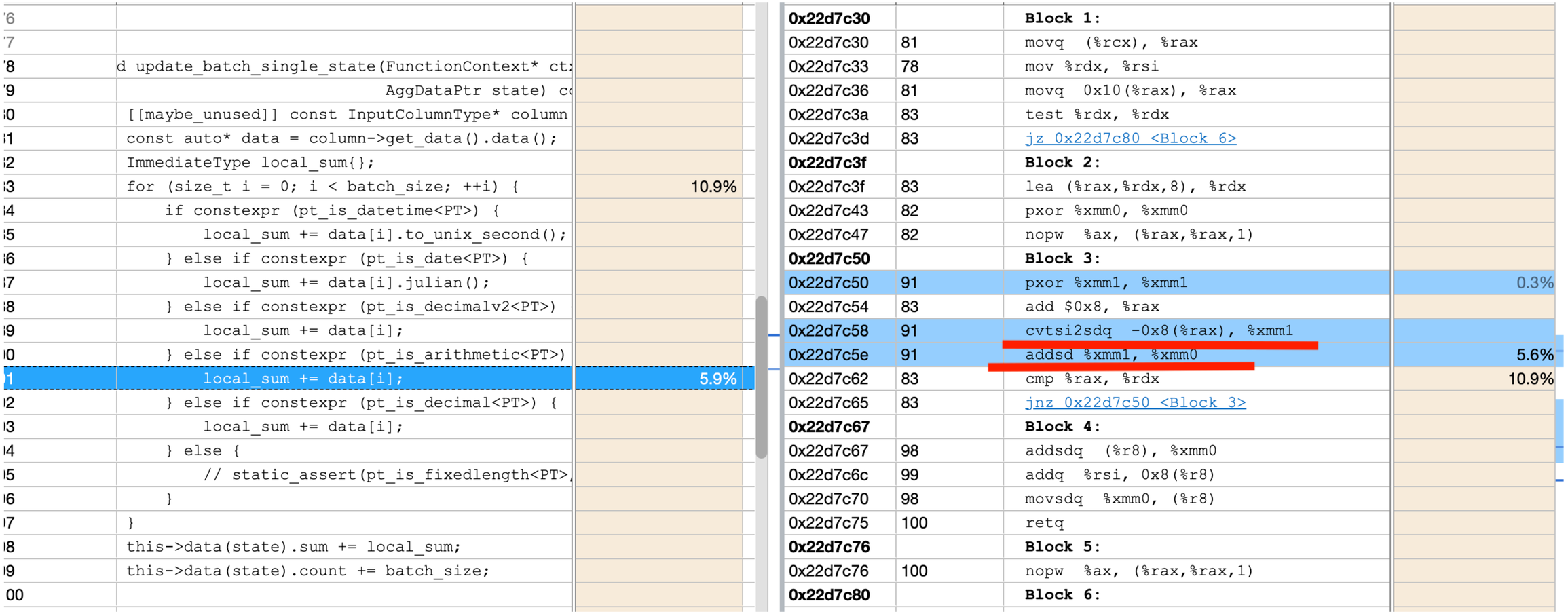
What



Profile Tools: VTune



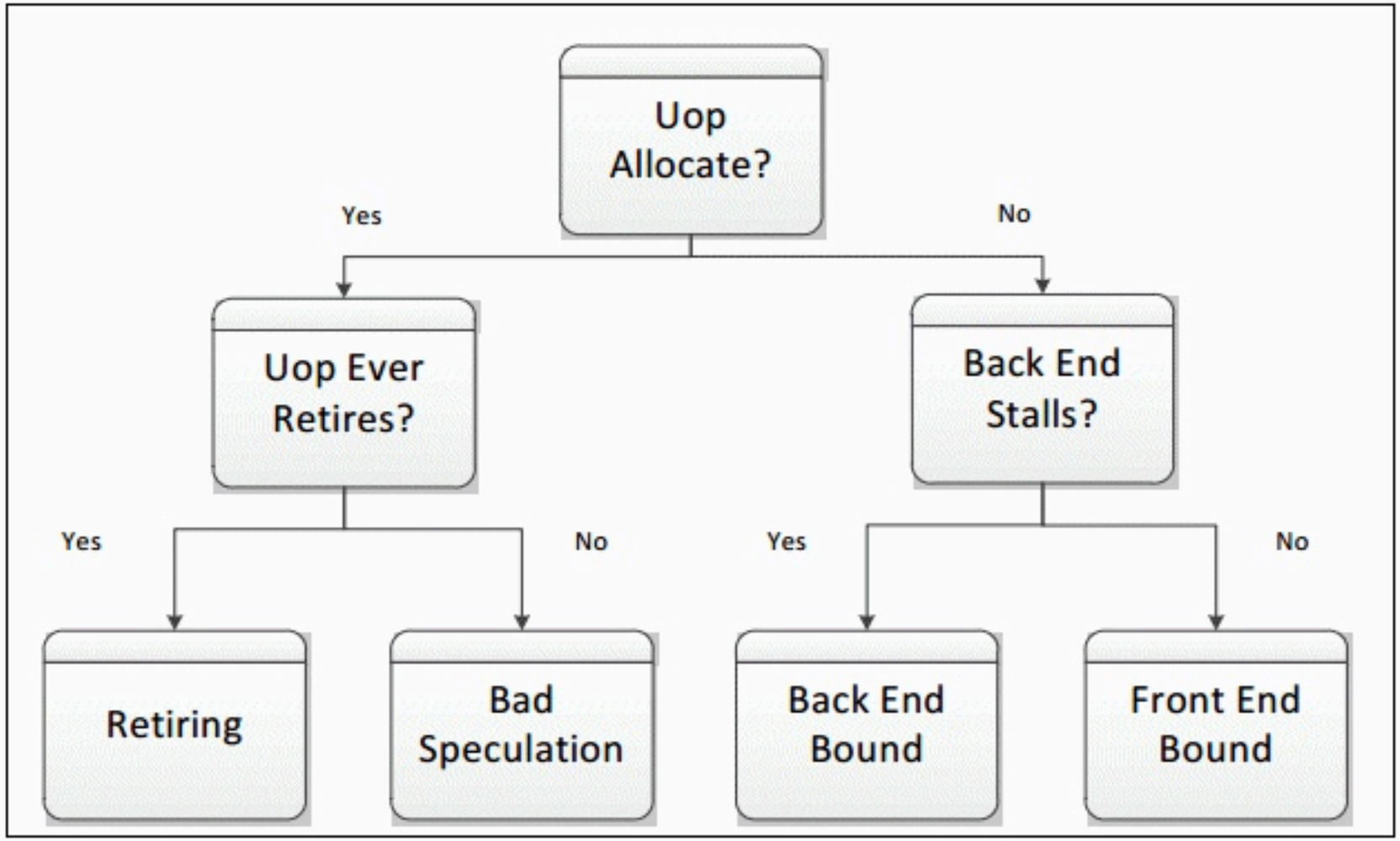
Profile Tools: VTune



6		
7		
8	d update_batch_single_state(FunctionContext* ct	
9	AggDataPtr state) c	
0	[[maybe_unused]] const InputColumnType* column	
1	const auto* data = column->get_data().data();	
2	ImmediateType local_sum{};	
3	for (size_t i = 0; i < batch_size; ++i) {	10.9%
4	if constexpr (pt_is_datetime<PT>) {	
5	local_sum += data[i].to_unix_second();	
6	} else if constexpr (pt_is_date<PT>) {	
7	local_sum += data[i].julian();	
8	} else if constexpr (pt_is_decimalv2<PT>)	
9	local_sum += data[i];	
0	} else if constexpr (pt_is_arithmetic<PT>)	
1	local_sum += data[i];	5.9%
2	} else if constexpr (pt_is_decimal<PT>) {	
3	local_sum += data[i];	
4	} else {	
5	// static_assert(pt_is_fixedlength<PT>,	
6	}	
7	}	
8	this->data(state).sum += local_sum;	
9	this->data(state).count += batch_size;	
00		

0x22d7c30		**Block 1:**	
0x22d7c30	81	movq (%rcx), %rax	
0x22d7c33	78	mov %rdx, %rsi	
0x22d7c36	81	movq 0x10(%rax), %rax	
0x22d7c3a	83	test %rdx, %rdx	
0x22d7c3d	83	jz 0x22d7c80 <Block 6>	
0x22d7c3f		**Block 2:**	
0x22d7c3f	83	lea (%rax,%rdx,8), %rdx	
0x22d7c43	82	pxor %xmm0, %xmm0	
0x22d7c47	82	nopw %ax, (%rax,%rax,1)	
0x22d7c50		**Block 3:**	
0x22d7c50	91	pxor %xmm1, %xmm1	0.3%
0x22d7c54	83	add $0x8, %rax	
0x22d7c58	91	cvtsi2sdq -0x8(%rax), %xmm1	
0x22d7c5e	91	addsd %xmm1, %xmm0	5.6%
0x22d7c62	83	cmp %rax, %rdx	10.9%
0x22d7c65	83	jnz 0x22d7c50 <Block 3>	
0x22d7c67		**Block 4:**	
0x22d7c67	98	addsdq (%r8), %xmm0	
0x22d7c6c	99	addq %rsi, 0x8(%r8)	
0x22d7c70	98	movsdq %xmm0, (%r8)	
0x22d7c75	100	retq	
0x22d7c76		**Block 5:**	
0x22d7c76	100	nopw %ax, (%rax,%rax,1)	
0x22d7c80		**Block 6:**	

Profile Tools: Toplev



https://github.com/andikleen/pmu-tools/wiki/toplev-manual

Profile Tools: BOLT

- Code Layout
- Link-Time Optimization (LTO)
- Profile-Guided Optimization (PGO)

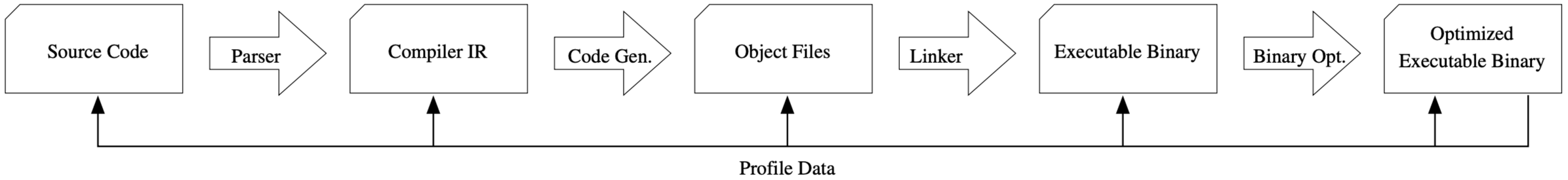
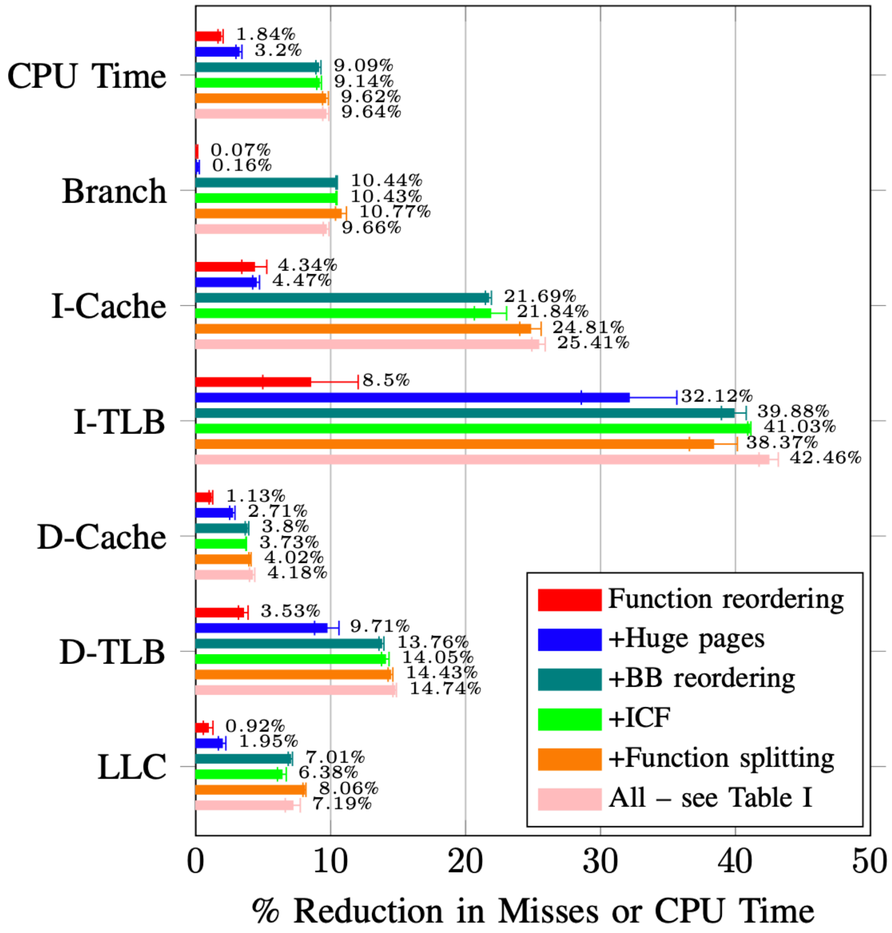


Fig. 1: Example of a compilation pipeline and the various alternatives to retrofit sample-based profile data.

https://github.com/facebookincubator/BOLT

Profile Tools: BOLT



https://github.com/facebookincubator/BOLT

StarRocks Database Vectorized

1 Database Performance

2 Vectorized Basic

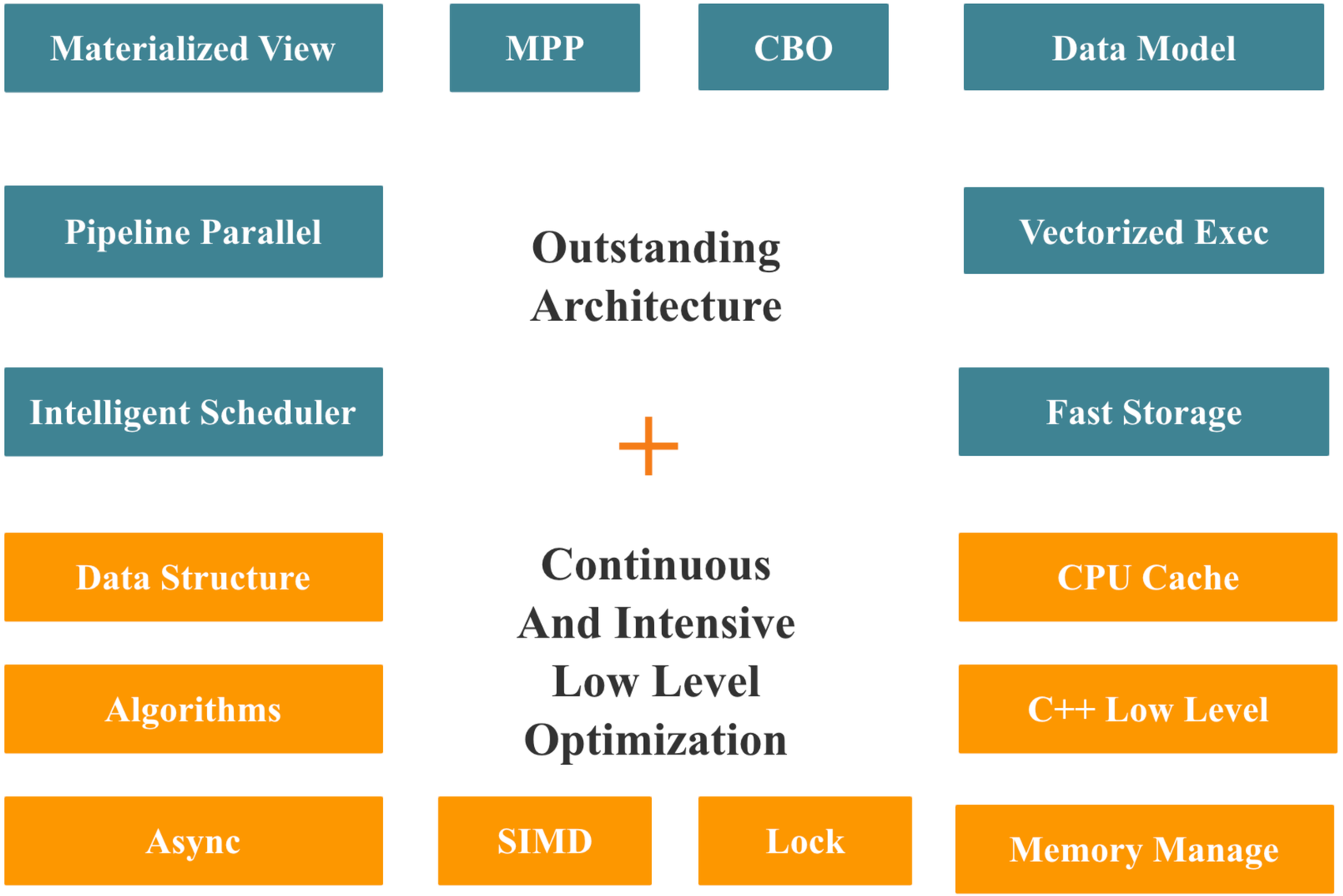
3 Database Vectorized

04

Thinking Vectorized

Thinking —— The Underlying Principle Is Similar

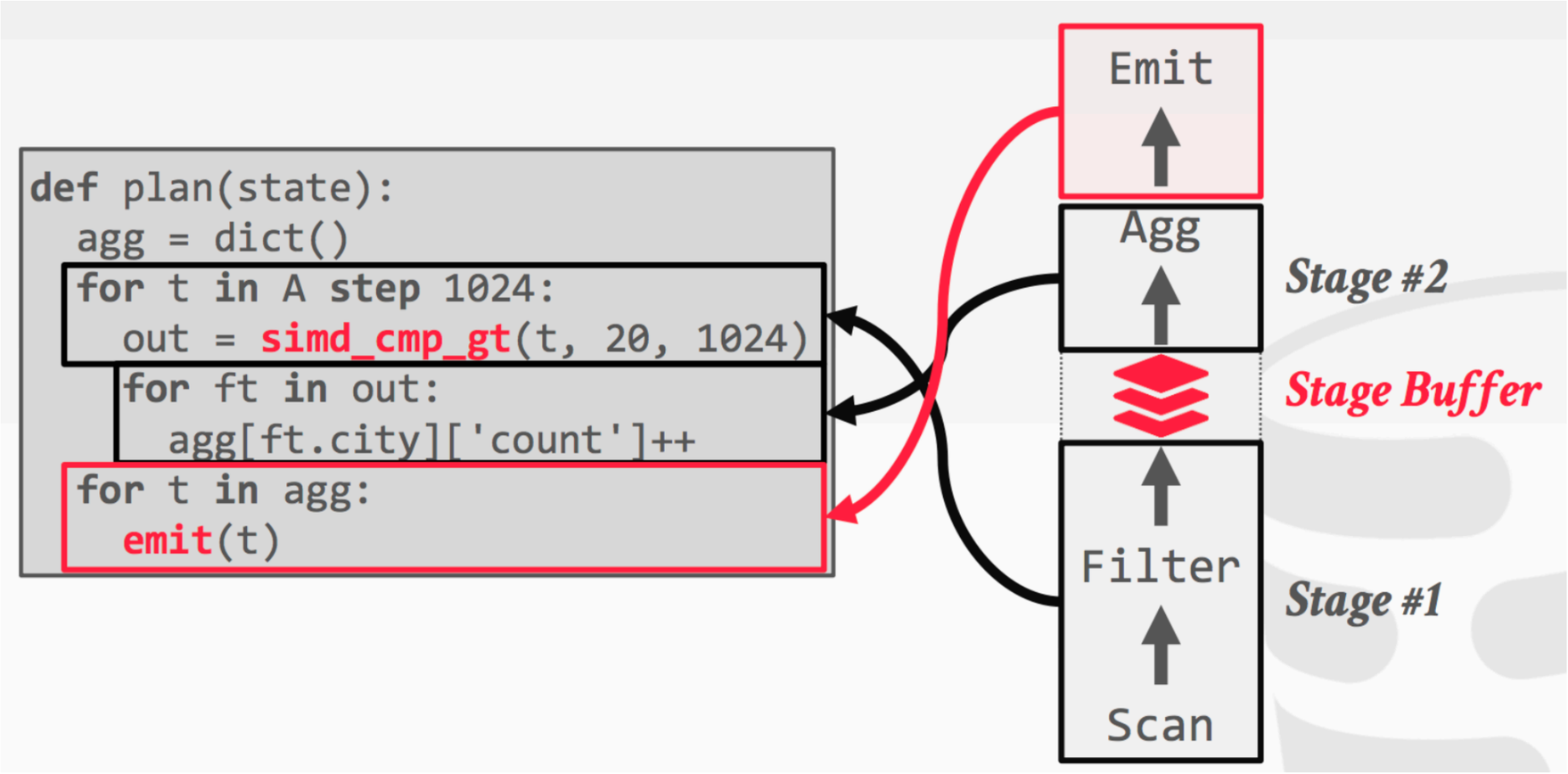
Thinking —— High Performance Database Need



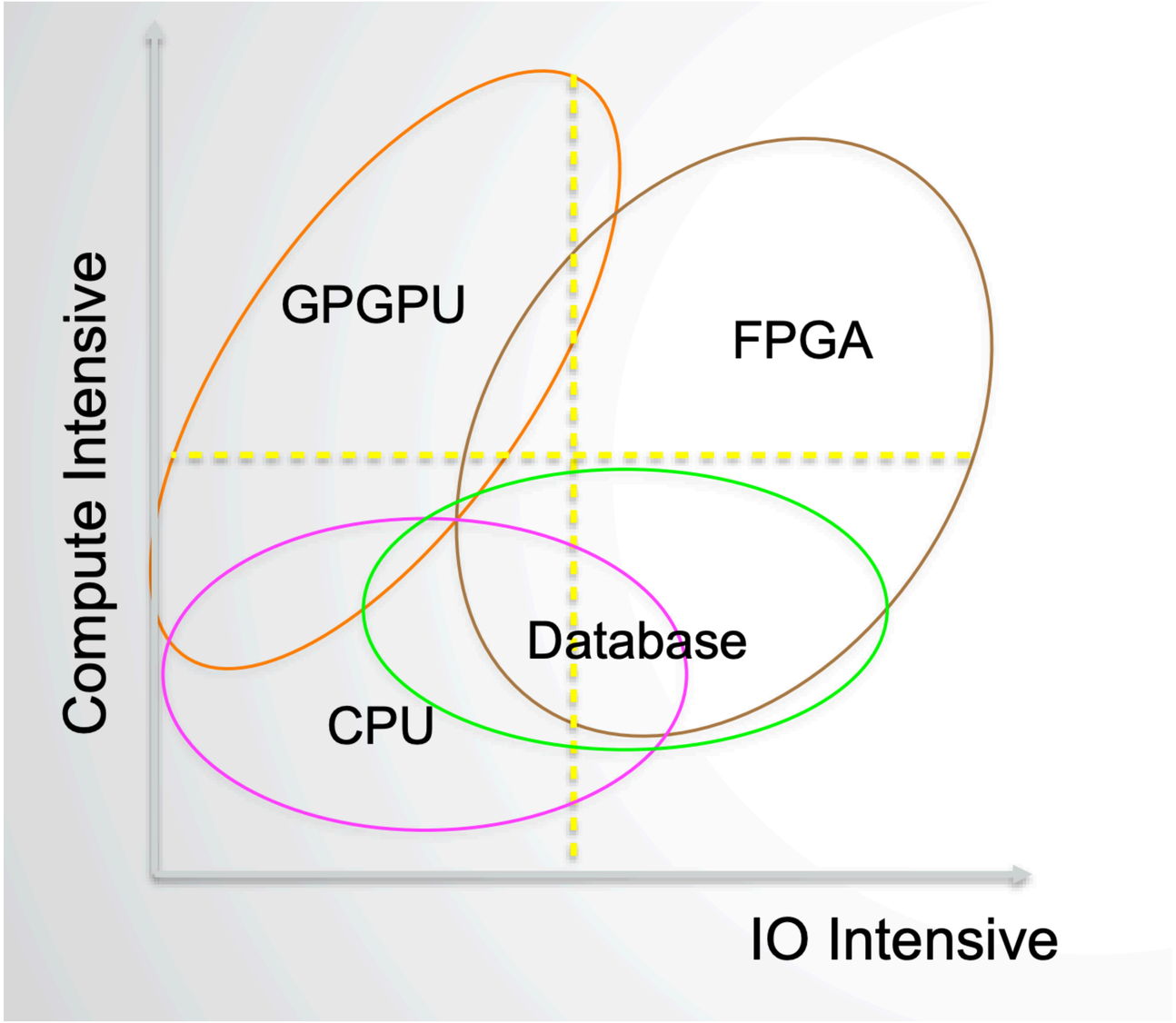
Thinking —— Fuse Vectorized And Compilation

- Complex Expression
- UDF, UDAF, UDTF
- Sort, Aggregate
- GPU

```
def plan(state):
  agg = dict()
  for t in A step 1024:
    out = simd_cmp_gt(t, 20, 1024)
    for ft in out:
      agg[ft.city]['count']++
  for t in agg:
    emit(t)
```

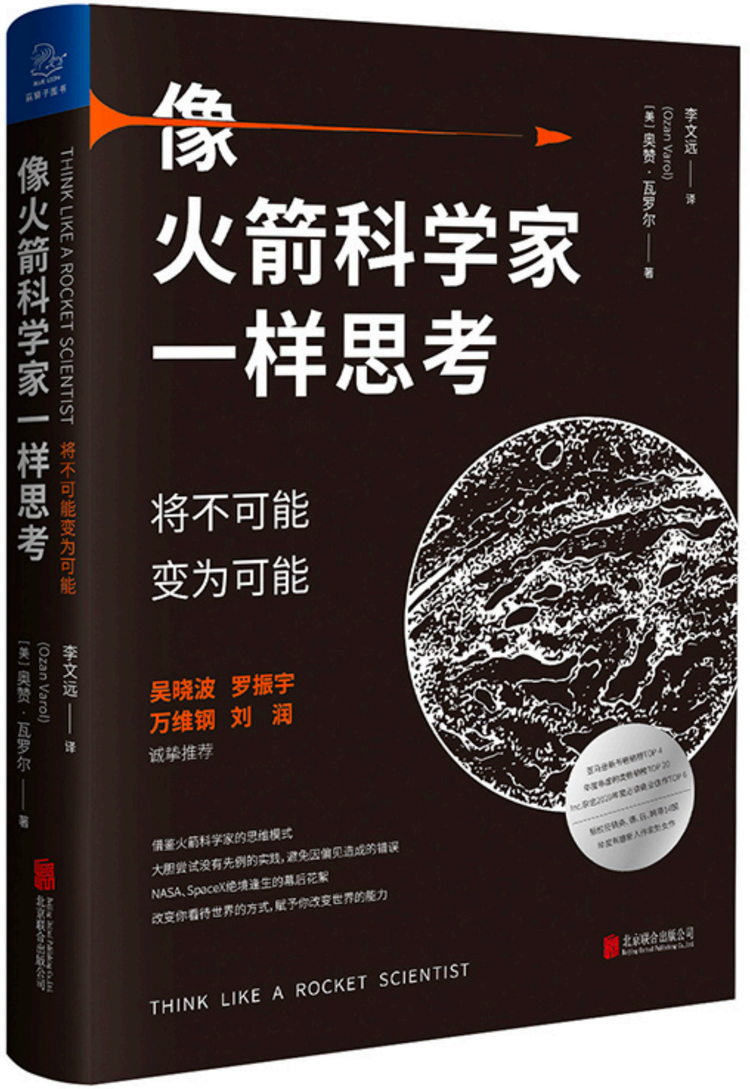


Thinking —— Accelerate Database With GPU Or FPGA



Thinking —— To Be Impossible

- 探险思维
- 第一性原理
- 奇迹思维
- 解决者思维
- 证伪思维
- 压力测试
- 迭代思维



Some Resources: Performance Optimization & Vectorized

- 数据库学习资料
  - 性能优化
    - Profile 工具
    - CPU 微架构
    - CPU Cache
  - 向量化
- How To Build A Fast DataBase

2022

Thanks