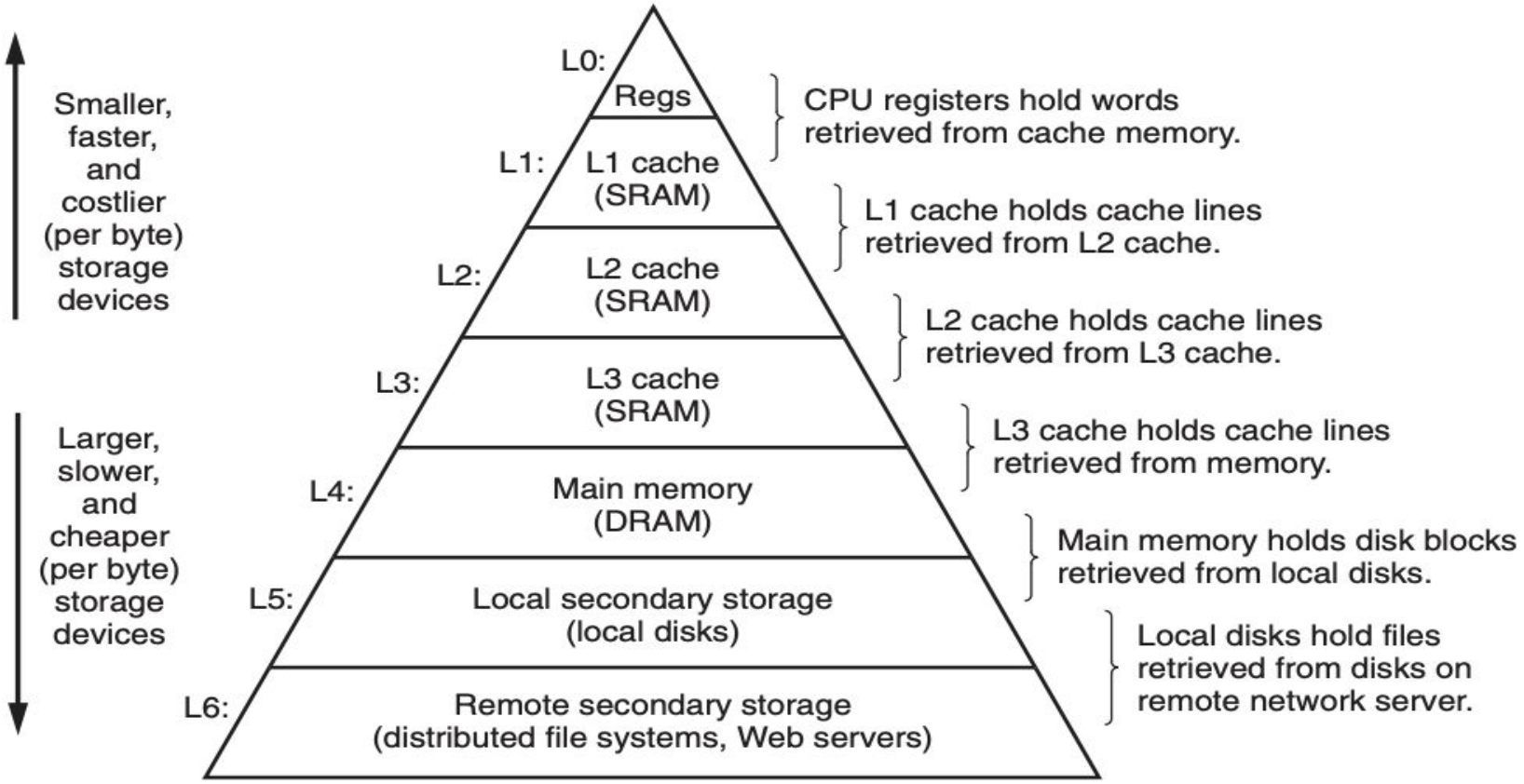
Performance tuning in databend

sundyli@datafuselabs（李本旺）
http://github.com/sundy-li/

目录
CONTENT

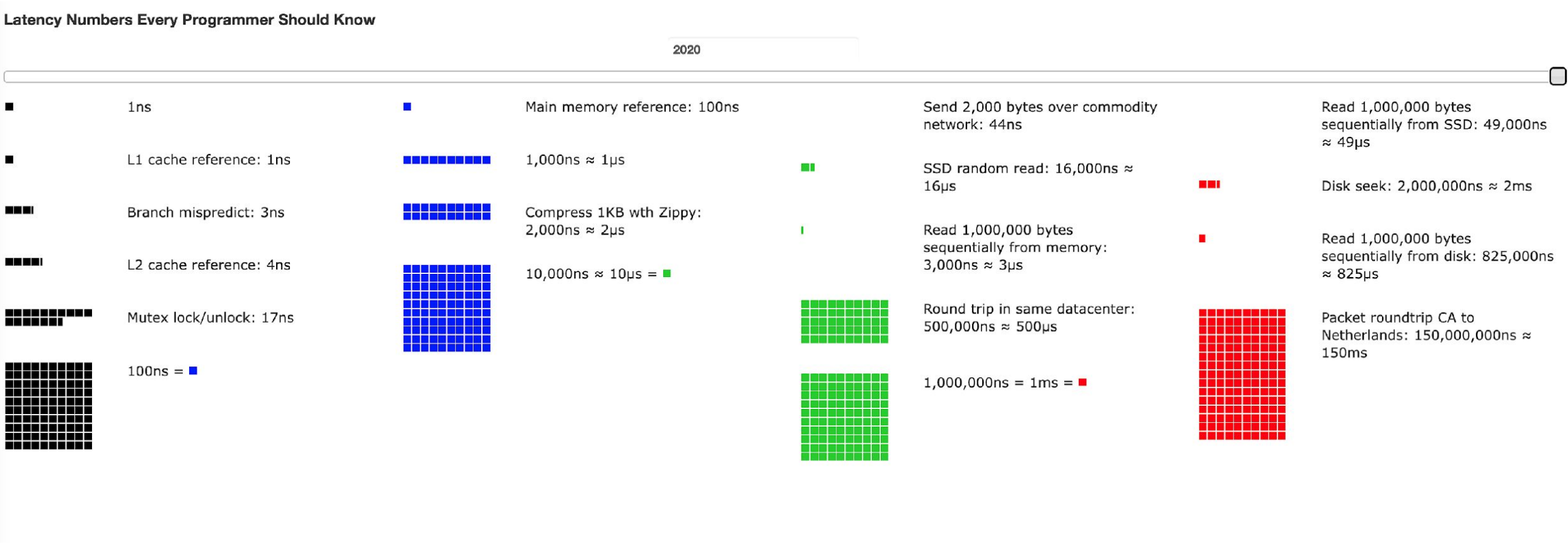
1 Knowledges before tuning the codes

memory hierarchy



From CSAPP

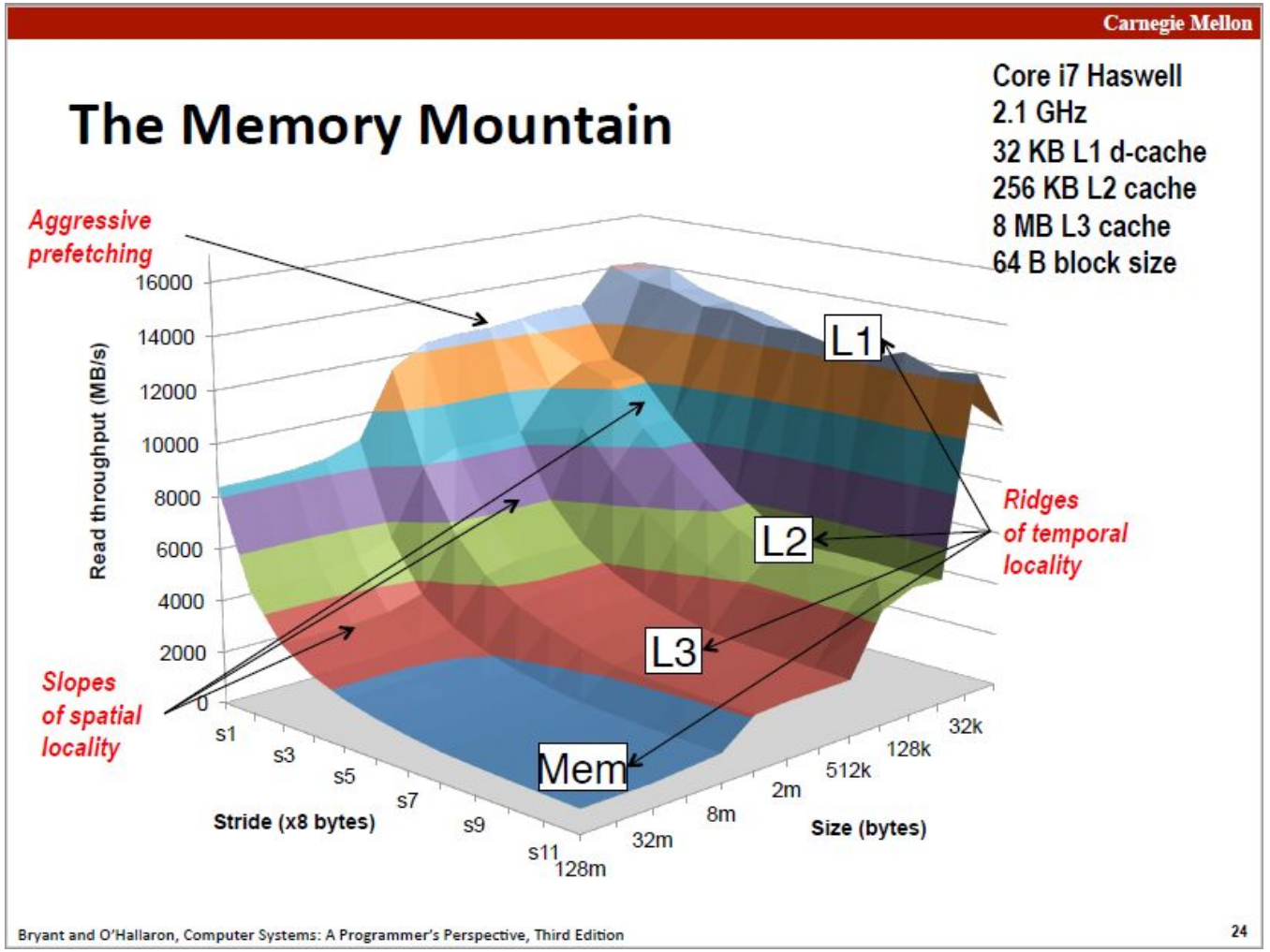
Latency Numbers



[Numbers Every Programmer Should Know By Year (colin-scott.github.io)]

Principle of locality

1. Data Locality
2. Time Locality



Data locality

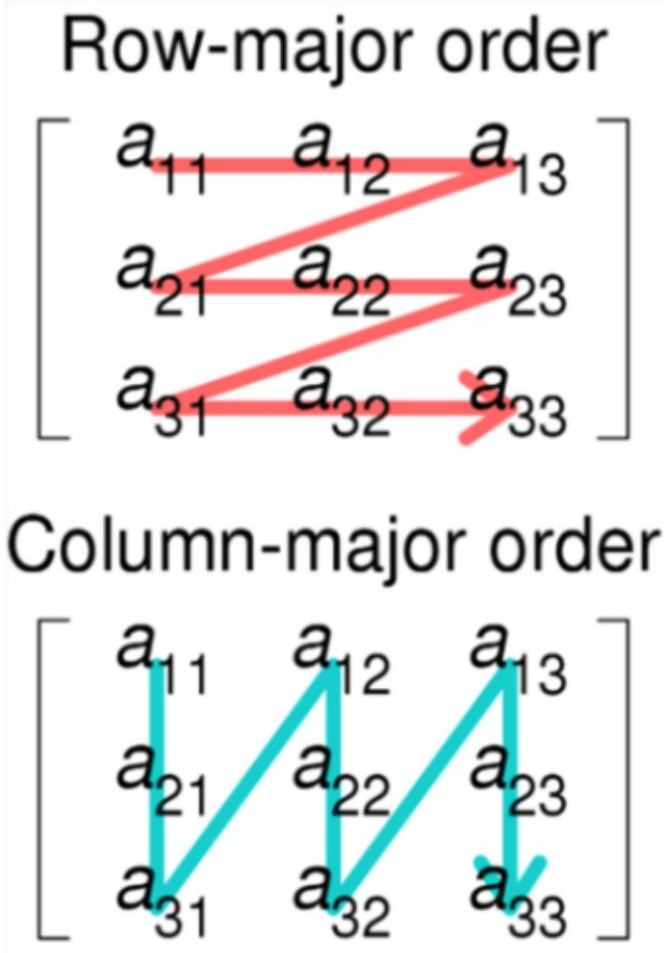
Row-major or Column-major

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



Expressing Performance


```
1   /* Compute prefix sum of vector a */
2   void psum1(float a[], float p[], long n)
3   {
4       long i;
5       p[0] = a[0];
6       for (i = 1; i < n; i++)
7           p[i] = p[i-1] + a[i];
8   }
9
10  void psum2(float a[], float p[], long n)
11  {
12      long i;
13      p[0] = a[0];
14      for (i = 1; i < n-1; i+=2) {
15          float mid_val = p[i-1] + a[i];
16          p[i]    = mid_val;
17          p[i+1]  = mid_val + a[i+1];
18      }
19      /* For even n, finish remaining element */
20      if (i < n)
21          p[i] = p[i-1] + a[i];
22  }
```

CPE

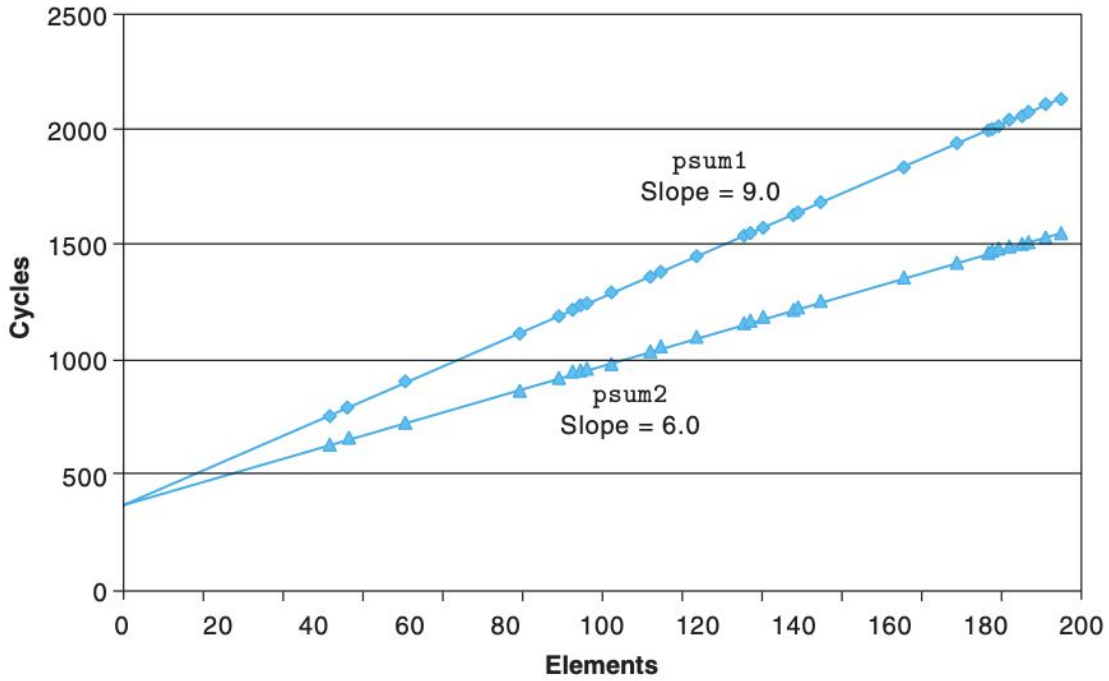


Figure 5.1 **Prefix-sum functions.** These functions provide examples for how we express program performance.

From CSAPP

Tools

Some useful tools & scripts in tuning databend's performance

1. Print & Instant elapsed
2. Perf is a good tool, pprof, flamegraph
3. cargo bench & criteron
4. databend-benchmark
5. numbers table && ignore function && progress

```
select count() from numbers_mt(10000000) where not ignore(toString(number))
```

2 Performance tuning in practice

inline vs never inline

Rust hint: #[inline], #[inline(never)], #[inline(always)]

1. Inlining is an optimizing transformation which replaces a call to a function with its body.
2. Private functions may not need inline due to LLVM's heuristic inline.
3. Generic functions are implicitly inlinable.

Yet:

1. Make compile times worser with larger target size.
2. Generic functions may make cause code bloat with inline

```rust
// Public, generic function.
// Will cause code bloat if not handled carefully!
pub fn parse_str(wat: impl AsRef<str>) -> Result<Vec<u8>> {
  // Immediately delegate to a non-generic function.
  _parse_str(wat.as_ref())
}

// Separate-compilation friendly private implementation.
fn _parse_str(wat: &str) -> Result<Vec<u8>> {
    ...
}
```

inline vs never inline

Case 1: inline vs never inline

```
40
41  fn default_crc32(array: &PrimitiveArray<u32>) -> u32 {
42      array.values().iter().fold(init: 0, f: |acc: u32, x: &u32| {
43          let mut hasher: Hasher = crc32fast::Hasher::new();
44          hasher.update(buf: &[*x as u8]);
45          let checksum: u32 = hasher.finalize();
46          acc + checksum
47      })
48  }
49
50  fn inlined_crc32(array: &PrimitiveArray<u32>) -> u32 {
51      array.values().iter().fold(init: 0, f: |acc: u32, x: &u32| {
52          let mut hasher: Hasher = mycrc32::Hasher::new();
53          hasher.update(buf: &[*x as u8]);
54          let checksum: u32 = hasher.finalize();
55          acc + checksum
56      })
57  }
```

default_crc32 time: [16.184 ms 16.196 ms 16.209 ms]

inlined_crc32 time: [1.6317 ms 1.6353 ms 1.6396 ms]

10X improvement

inline is your friend!!

inline vs never inline

Case 2: inline vs never inline

```
42
43  impl<Method: HashMethod + PolymorphicKeysHelper<Method>> Aggregator<Method> {
44      pub fn create(method: Method, params: AggregatorParamsRef) -> Aggregator<Method> {
45          Aggregator { method, params }
46      }
47
48      // If we set it to inline(performance degradation).
49      // Because it will make other internal functions to no inline
50      #[inline(never)]      zhang2014, 3 months ago • Try to determine inline
51      pub async fn aggregate(…
55      ) -> Result<Method::State> {
56          // This may be confusing
57          // It will help us improve performance ~10% when we declare local references for them.
58          let hash_method: &Method = &self.method;
59          let aggregator_params: &AggregatorParams = self.params.as_ref();
60
61          let mut state: <Method as PolymorphicKeysHelper<…>>::State = hash_method.aggregate_state();
62
63          match aggregator_params.aggregate_functions.is_empty() {…
87
88          Ok(state)
89      }
90
```

never inline is your friend!!!

codes in databend

No extra allocation/copy

Case 3: numbers stream in arrow format

select sum(number) from numbers_mt(10000000000)

```
76 -            let data: Vec<u64> = (current.begin..current.end).collect();
77 -            let block =
78 -                DataBlock::create(self.schema.clone(), vec!
   [Arc::new(UInt64Array::from(data))]);
79 -            Some(Ok(block))
```

```
87 +            unsafe {
88 +                let layout = Layout::from_size_align_unchecked(
89 +                    length * mem::size_of::<u64>(),
90 +                    mem::size_of::<u64>(),
91 +                );
92 +                let p = std::alloc::alloc(layout) as *mut u64;
93 +                for i in current.begin..current.end {
94 +                    *p.offset((i - current.begin) as isize) = i;
95 +                }
96 +                let buffer =
97 +                    Buffer::from_raw_parts(NonNull::new(p as *mut u8).unwrap(), length,
   length);
98 +
99 +                let arr_data = ArrayData::builder(DataType::UInt64)
100 +                    .len(length)
101 +                    .offset(0)
102 +                    .add_buffer(buffer)
103 +                    .build();
104 +
105 +                let block = DataBlock::create(
106 +                    self.schema.clone(),
107 +                    vec![Arc::new(UInt64Array::from(arr_data))],
108 +                );
109 +                Some(Ok(block))
110 +            }
```

Databend PR #19

No extra allocation/copy

Case 3: numbers stream in arrow format

Profile and tunning

- Count, sum(top3) = 86.44%

```
    31.70%  tokio-runtime-w  libc-2.19.so        [.] __memcpy_sse2_unaligned
    27.95%  tokio-runtime-w  libc-2.19.so        [.] memset
    26.79%  tokio-runtime-w  fuse-query          [.] <alloc::vec::Vec<T> as alloc::vec::spec_from_iter::SpecFr
     1.69%  tokio-runtime-w  libc-2.19.so        [.] malloc
```

- Max, sum(top4) = 87.31%

```
    28.34%  tokio-runtime-w  libc-2.19.so       [.] __memcpy_sse2_unaligned
    24.52%  tokio-runtime-w  fuse-query         [.] <alloc::vec::Vec<T> as alloc::vec::spec_from_iter::SpecFrc
    24.20%  tokio-runtime-w  libc-2.19.so       [.] memset
    10.25%  tokio-runtime-w  fuse-query         [.] arrow::compute::kernels::aggregate::max
     1.60
```

As we can see, `max` query has about `10.25%` extra cpu costs in `arrow::compute::kernels::aggregate::max`.
But why it performances the same as the `count` query?

After some thinking, I realized that must be the problem of source data generation, the speed of upstream generation is slow, so the downstream are hungry. Just like two men are waiting for food, one of them can finish the dinner in 5 minutes, the other may take 1 minute, but food preparation time may take an hour, so we see they just finish the dinner at the same time.

memcpy is the hot path via perf

No extra allocation/copy

Case 4.1: cast primitive to string

select count() from numbers_mt(10000000000) where not ignore(toString(number))

```
@@ -19,9 +17,21 @@
19  pub fn primitive_to_binary<T: NativeType + lexical_core::ToLexical, O: Offset>(
20      from: &PrimitiveArray<T>,
21  ) -> BinaryArray<O> {
22 -    let iter = from.iter().map(|x| x.map(|x| lexical_to_bytes(*x)));
23 -
24 -    BinaryArray::from_trusted_len_iter(iter)
25  }
26
```

```
@@ -19,9 +17,21 @@
17  pub fn primitive_to_binary<T: NativeType + lexical_core::ToLexical, O: Offset>(
18      from: &PrimitiveArray<T>,
19  ) -> BinaryArray<O> {
20 +    let mut buffer = vec![];
21 +    let builder = from.iter().fold(
22 +        MutableBinaryArray::<O>::with_capacity(from.len()),
23 +        |mut builder, x| {
24 +            match x {
25 +                Some(x) => {
26 +                    lexical_to_bytes_mut(*x, &mut buffer);
27 +                    builder.push(Some(buffer.as_slice()));
28 +                }
29 +                None => builder.push_null(),
30 +            }
31 +            builder
32 +        },
33 +    );
34 +    builder.into()
35  }
36
```

[Arrow2 PR #646]

Improved performance in cast Primitive to Binary/String (2x) #646

Merged jorgecarleitao merged 2 commits into jorgecarleitao:main from datafuse-extras:primitive-cast 2 days ago

Conversation 2 Commits 2 Checks 24 Files changed 2

sundy-li commented 2 days ago • edited Contributor

Reuse the temp bytes buffer.

```
cast int32 to binary 512
                time:   [10.650 us 10.761 us 10.885 us]
                change: [-47.175% -46.939% -46.649%] (p = 0.00 < 0.05)
                Performance has improved.
```

3

No extra allocation/copy

Case 4.2: cast primitive to string no extra

```
    /// Write values directly into the underlying data.
    /// # Safety
    /// Caller must ensure that `len <= self.capacity()`
    #[inline]
    pub unsafe fn write_values<F>(&mut self, mut f: F)
    where
        F: FnMut(&mut [u8]) -> usize,
    {
        // ensure values has enough capacity and size to write
        self.values.set_len(self.values.capacity());
        let buffer = &mut self.values.as_mut_slice()
[self.offsets.last().unwrap().to_usize()..];
        let len = f(buffer);
        let new_len = self.offsets.last().unwrap().to_usize() + len;
        self.values.set_len(new_len);

        let size = O::from_usize(new_len).ok_or(ArrowError::Overflow).unwrap();
        self.offsets.push(size);
        match &mut self.validity {
            Some(validity) => validity.push(true),
            None => {}
        }
    }
```

select count() from numbers_mt(10000000000) where not ignore(toString(number))

```
pub fn primitive_to_binary<T: NativeType + lexical_core::ToLexical, O: Offset>(
    from: &PrimitiveArray<T>,
) -> BinaryArray<O> {

    let builder = from.iter().fold(
        MutableBinaryArray::<O>::with_capacity(from.len()),
        |mut builder, x| {
            match x {
+               Some(x) => unsafe {
+                   builder.reserve(1, T::FORMATTED_SIZE_DECIMAL);
+                   builder.write_values(|bytes| lexical_core::write(*x, bytes).len());
+               },
                None => builder.push_null(),
            }
            builder
```

Improved performance in cast Primitive to Binary/String again (2x) #6

Open sundy-li wants to merge 2 commits into jorgecarleitao:main from datafuse-extras:primitive-cast3

Conversation 3 Commits 2 Checks 0 Files changed 4

sundy-li commented 20 hours ago • edited Contributor

Memcpy style write, no extra copy.

```
cast int32 to binary 512
                        time:   [7.2389 us 7.2844 us 7.3306 us]
                        change: [-32.305% -31.940% -31.566%] (p = 0.00 < 0.05)
                        Performance has improved.
Found 8 outliers among 100 measurements (8.00%)
```

[Arrow2 PR #651]

No extra allocation/copy

Case 5: runtime_tracker

```
116 +
117 -     pub fn get_memory_tracker(&self) -> Arc<MemoryTracker> {
118 -         self.memory_tracker.clone()
119       }
```

```
160
161 +     #[inline]
162 +     pub fn get_memory_tracker(&self) -> &MemoryTracker {
163 +         &self.memory_tracker
164       }
```

```
56  +
57  +     #[inline]
58  +     pub fn alloc_memory(size: i64) {
59  +         unsafe {
60  +             if !TRACKER.is_null() {
61  +                 (*TRACKER).untracked_memory += size;
62  +
63  +                 if (*TRACKER).untracked_memory > UNTRACKED_MEMORY_LIMIT {
64  +                     (*TRACKER)
65  +                         .rt_tracker
66  +                         .memory_tracker
67  +                         .alloc_memory((*TRACKER).untracked_memory);
68  +                     (*TRACKER).untracked_memory = 0;
69  +                 }
70  +             }
71  +         }
72  +     }
```

Databend PR #3140

Less extra function call

Case 6: extra function call in closure

3 src/compute/sort/primitive/indices.rs

View full | Viewed

```
@@ -18,10 +18,11 @@ where
     T: NativeType,
     F: Fn(&T, &T) -> std::cmp::Ordering,
 {
+    let values = array.values().as_slice();
     unsafe {
         common::indices_sorted_unstable_by(
             array.validity(),
-            |x: usize| *array.values().as_slice().get_unchecked(x),
+            |x: usize| *values.get_unchecked(x),
             cmp,
             array.len(),
             options,
```

sundy-li commented on 2 Aug — Contributor — Author

`as_slice` is the bottleneck.

I think we can use `values: &[T]` to replace `get: G`.

Improved in commit: datafuse-extras#1

Now:

```
arrow2-sort-limit 2^13 f32
  time:   [46.601 us 46.665 us 46.728 us]
  change: [-24.847% -24.656% -24.483%] (p = 0.00 < 0.05)
  Performance has improved.
```

Another bottleneck is `from_usize().unwrap()` in the loop.

2

[Arrow2 ISSUE #245]

Less branch prediction miss

Case 7: unwrap inside loop

```
168         let mut indices: MutableBuffer<I> = unsafe {
169             MutableBuffer::from_trusted_len_iter_unchecked(
170                 iterator: (0..length).map(|x: usize| I::from_usize(index: x).unwrap()),
171             )
172         };
173
174         // Soundness:
175         // indices are by construction `< values.len()`
176         // limit is by construction `< values.len()`
177         sort_unstable_by(&mut indices, get, cmp, descending, limit);
178
```

```
155     } else {
156         let mut indices: MutableBuffer<I> = MutableBuffer::from_trusted_len_iter(iterator: I::range(start: 0, end: length).unwrap());
157
158         // Soundness:
159         // indices are by construction `< values.len()`
160         // limit is by construction `< values.len()`
161         sort_unstable_by(&mut indices, get, cmp, descending, limit);
162         indices.truncate(len: limit);
163         indices.shrink_to_fit();
```

Arrow2 Codes

Less branch prediction miss

Case 8: transform primitive array

```rust
let result: DFUInt8Array = array.i8()?.iter().map(|x| abs(*x) ).collect();
```

Arrow's format

You, 6 days ago | 1 author (You)

```
#### Column Iteration and validity bitmap combination
To iterate the column, we can use `column.iter()` to generate an iterator, the item is `Option<T>`, `None` means it's null.
But this's inefficient because we need to check the null value every time we iterate the column inside the loop which will pollute the CPU cache.

According to the memory layout of Arrow, we can directly use the validity bitmap of the original column to indicate the null value.
So we have `ArrayApply` trait to help you iterate the column. If there are two zip iterator, we can use `binary` function to combine the validity bitmap of two columns.

ArrayApply
```rust
let array: DFUInt8Array = self.apply_cast_numeric(|a| {
                                AsPrimitive::<u8>::as_(a - (a / rhs) * rhs)
                            });
```     You, 6 days ago • Update docs

binary
```rust
binary(x_series.f64()?, y_series.f64()?, |x, y| x.pow(y))
```
```

Less branch prediction miss

Case 8.1: transform aggregator

loop inside the match vs match inside the loop

```
match aggregator_params.aggregate_functions.is_empty() {
    true => {
        while let Some(block: Result<DataBlock, ErrorCode>) = stream.next().await {
            let block: DataBlock = block?;

            // 1.1 and 1.2.
            let group_columns: Vec<&DataColumn> = Self::group_columns(names: &group_cols, &block)?;
            let group_keys: Vec<<Method as HashMethod>::HashKey> = hash_method.build_keys(&group_columns, block.num_rows())?;
            self.lookup_key(group_keys, &mut state);
        }
    }
    false => {
        while let Some(block: Result<DataBlock, ErrorCode>) = stream.next().await {
            let block: DataBlock = block?;

            // 1.1 and 1.2.
            let group_columns: Vec<&DataColumn> = Self::group_columns(names: &group_cols, &block)?;
            let group_keys: Vec<<Method as HashMethod>::HashKey> = hash_method.build_keys(&group_columns, block.num_rows())?;

            let places: Vec<StateAddr> = self.lookup_state(group_keys, &mut state);
            Self::execute(aggregator_params, &block, &places)?;
        }
    }
}
```

Less branch prediction miss

Case 9: null & if combinator in sum

```
select sumIf(number, number > 3) from numbers_mt(N)
```

```
/// Merge the 2 sets of flags (null and if) into a single one. This allows us to use parallelizable sums when
available
const auto * if_flags = assert_cast<const ColumnUInt8&>(*columns[if_argument_pos]).getData().data();
```

```
62  size_t countBytesInFilterWithNull(const IColumn::Filter & filt, const UInt8 * null_map)
63  {
64      size_t count = 0;
65
66      /** NOTE: In theory, `filt` should only contain zeros and ones.
67        * But, just in case, here the condition > 0 (to signed bytes) is used.
68        * It would be better to use != 0, then this does not allow SSE2.
69        */
70
71      const Int8 * pos = reinterpret_cast<const Int8 *>(filt.data());
72      const Int8 * pos2 = reinterpret_cast<const Int8 *>(null_map);
73      const Int8 * end = pos + filt.size();
74
75  #if defined(__SSE2__) && defined(__POPCNT__)
76      const Int8 * end64 = pos + filt.size() / 64 * 64;
77
78      for (; pos < end64; pos += 64, pos2 += 64)
79          count += __builtin_popcountll(toBits64(pos) & ~toBits64(pos2));
80
81          /// TODO Add duff device for tail?
82  #endif
83
84      for (; pos < end; ++pos, ++pos2)
85          count += (*pos & ~*pos2) != 0;
86
87      return count;      Alexey Milovidov, 5 years ago • Changed tabs to spaces in code [#CLICKHOUSE-3].
88  }
```

ClickHouse codes

java - Why is processing a sorted array faster than processing an unsorted array? - Stack Overflow

Case 10: simd sum

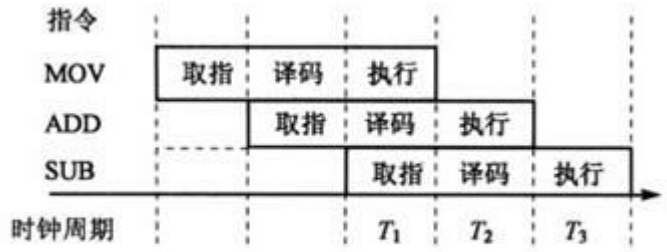


图 1　ARM7 单周期指令最佳流水线

```
double CommonAdd(double *data, int count)
{
    double result = 0;

    for (int i = 0 ; i < count; i++) {
        result += data[i];
    }

    return result;
}
```

normal version

```
double AVXAdd(double *data, int count)
{
    int offset = 0;

    __m256d v1;
    __m256d sum = _mm256_setzero_pd();

    double ret = 0;

    for (int i = 0; i < count/4; i++) {
        v1 = _mm256_load_pd(data + offset);
        sum = _mm256_add_pd(sum, v1);
        offset += 4;
    }

    sum = _mm256_hadd_pd(sum, sum);

    ret += sum[0];
    ret += sum[2];

    return ret;
}
```

simd version

SIMD in rust

Case 11: null sum in arrow2

```rust
#[multiversion]
#[clone(target = "x86_64+avx")]
fn null_sum_impl<T, I>(values: &[T], mut validity_masks: I) -> T
where
    T: NativeType + Simd,
    T::Simd: Add<Output = T::Simd> + Sum<T>,
    I: BitChunkIterExact<<<T as Simd>::Simd as NativeSimd>::Chunk>,
{
    let mut chunks = values.chunks_exact(T::Simd::LANES);

    let sum = chunks.by_ref().zip(validity_masks.by_ref()).fold(
        T::Simd::default(),
        |acc, (chunk, validity_chunk)| {
            let chunk = T::Simd::from_chunk(chunk);
            let mask = <T::Simd as NativeSimd>::Mask::from_chunk(validity_chunk);
            let selected = chunk.select(mask, T::Simd::default());
            acc + selected
        },
    );

    let remainder = T::Simd::from_incomplete_chunk(chunks.remainder(), T::default());
    let mask = <T::Simd as NativeSimd>::Mask::from_chunk(validity_masks.remainder());
    let remainder = remainder.select(mask, T::Simd::default());
    let reduced = sum + remainder;

    reduced.simd_sum()
}
```

Jorge Leitao, 7 months ago • Refactored aggregate. (#54)

rust-lang/portable-simd: The testing ground for the future of portable SIMD in Rust (github.com)

packed_simd_2 - Rust (rust-lang.github.io)

Auto vectorized


```
arrow2-sum 2^13 u32     time:   [545.01 ns 546.26 ns 547.72 ns]
                        change: [-0.2234% +0.2358% +0.8664%] (p = 0.41 > 0.05)
                        No change in performance detected.

arrow2-sum 2^13 u32 nosimd
                        time:   [316.59 ns 317.36 ns 318.20 ns]
                        change: [+0.4290% +0.8618% +1.2727%] (p = 0.00 < 0.05)
                        Change within noise threshold.
arrow2-sum null 2^13 u32
                        time:   [4.3197 us 4.3290 us 4.3394 us]
                        change: [-0.1470% +0.3333% +0.8057%] (p = 0.19 > 0.05)

arrow2-sum null 2^13 u32 nosimd
                        time:   [10.149 us 10.168 us 10.190 us]
                        change: [-0.8429% -0.4038% +0.1192%] (p = 0.11 > 0.05)
```

```rust
fn nosimd_sum_u32(arr_a: &PrimitiveArray<u32>) -> Option<u32> {
    let mut sum = 0;
    if arr_a.null_count() == 0 {
        arr_a.values().as_slice().iter().for_each(|f| {
            sum += *f;
        });
    } else {
        if let Some(c) = arr_a.validity() {
            arr_a
                .values()
                .as_slice()
                .iter()
                .zip(c.iter())
                .for_each(|(f, is_null)| sum += *f * (is_null as u32));
        }
    }
```

Auto vectorized vs packed_simd in arrow · Issue #255 · jorgecarleitao/arrow2 (github.com)

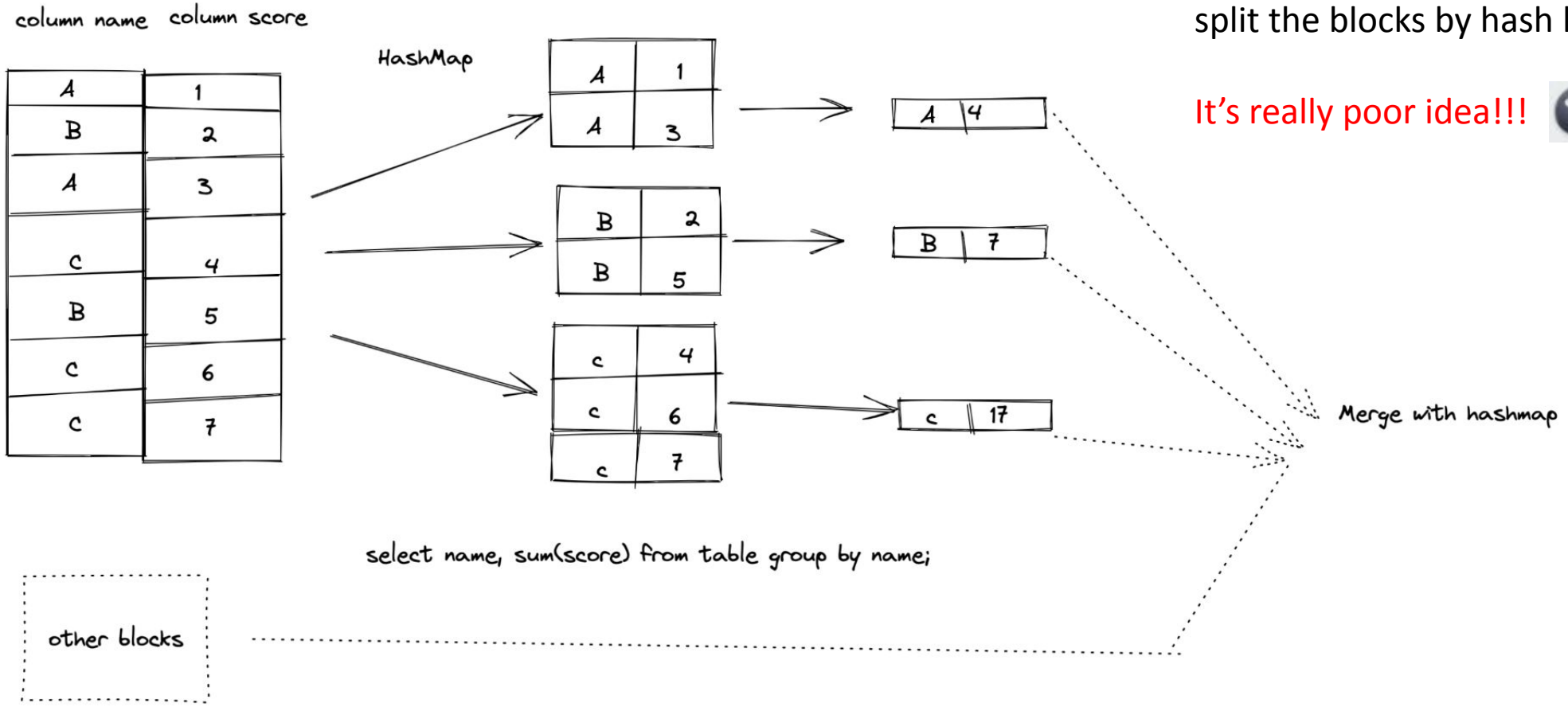
Aggregation optimization

A Long and sad story to tell

```
SELECT max(number), sum(number) FROM numbers_mt(1000000000)
GROUP BY number % 3, number % 4, number % 5 LIMIT 10;
```

This SQL works extremely slow in databend’s early versions.

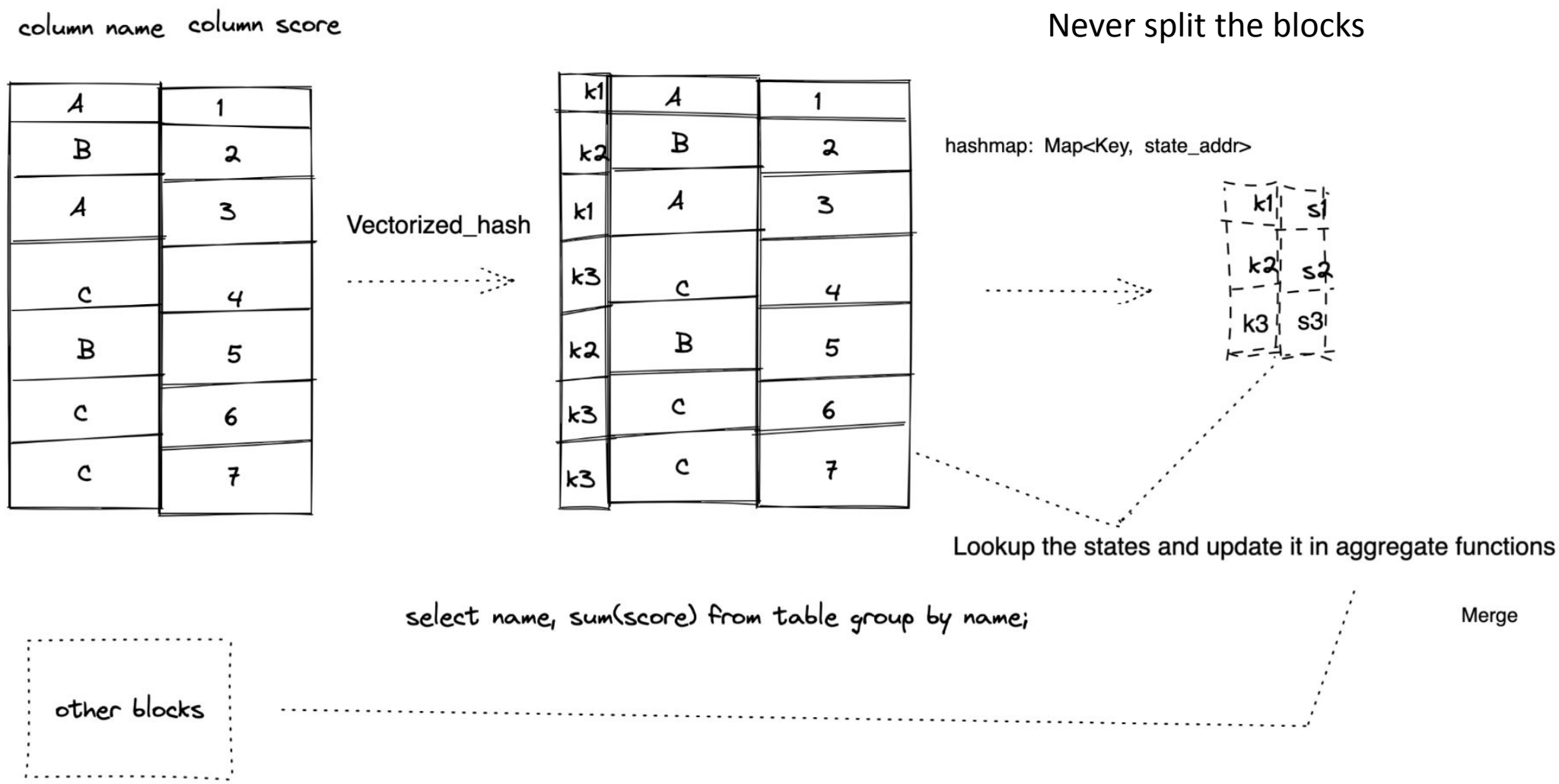
Group by transform



Early version:
split the blocks by hash key

It's really poor idea!!!

Group by transform



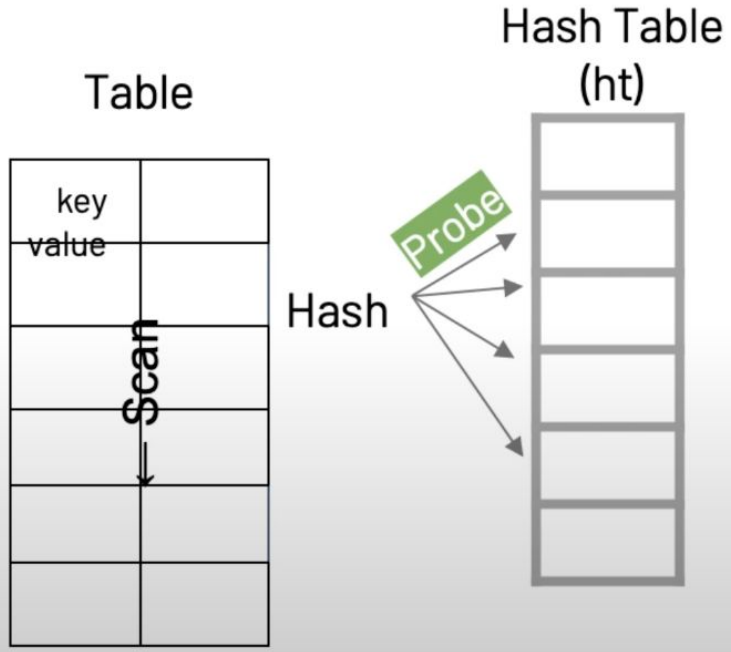
Hash Table: Most Important Data Structure

Example: select sum(value) from table group by key

```
for (int32_t i = 0; i < batchSize; ++i) {
  int32_t bucket = hash(keyCol[i]) % ht->size;

  if (ht[bucket].key == keyCol[i]) {
    ht[bucket].value += valueCol[i];
  }
}
```

Note: "Over-simplified". No hash table collision.

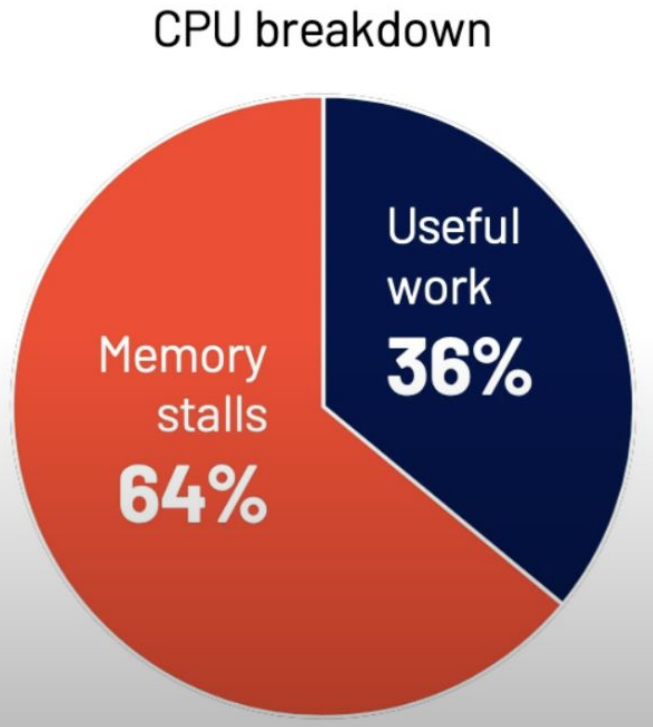


Hash Table: Most Important Data Structure

Example: select sum(value) from table group by key

```
for (int32_t i = 0; i < batchSize; ++i) {
  int32_t bucket = hash(keyCol[i]) % ht->size;

  if (ht[bucket].key == keyCol[i]) {
    ht[bucket].value += valueCol[i];
  }
}
```



Make loop smaller

How do we optimize the following?

```
for (int32_t i = 0; i < batchSize; ++i) {
  int32_t bucket = hash(keyCol[i]) % ht->size;

  if (ht[bucket].key == keyCol[i]) {
    ht[bucket].value += valueCol[i];
  }
}
```

Loop body is large (key comparison can be very expensive, e.g. comparing strings)

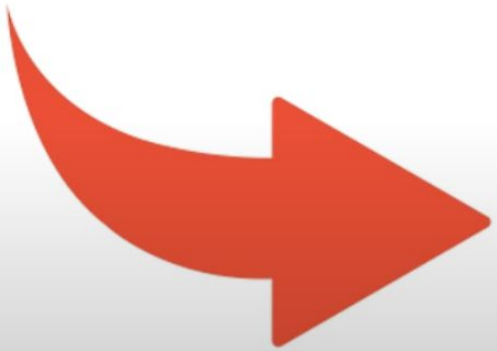
- Instructions to access memory are intermixed with instructions to compute hash code, compare keys and addition.
- Large loop body -> CPU sees fewer memory access instruction -> CPU executes fewer memory access at once.
- Solution: make loops smaller.

Make loop smaller

How do we optimize the following?

```
for (int32_t i = 0; i < batchSize; ++i) {
  int32_t bucket = hash(keyCol[i]) % ht->size;

  if (ht[bucket].key == keyCol[i]) {
    ht[bucket].value += valueCol[i];
  }
}
```



Break the big loop into smaller loops

```
for (int32_t i = 0; i < batchSize; ++i) {
  buckets[i] = hash(keyCol[i]) % ht->size;
}

for (int32_t i = 0; i < batchSize; ++i) {
  keys[i] = ht[buckets[i]].key;
}

for (int32_t i = 0; i < batchSize; ++i) {
  if (keys[i] == keyCol[i]) {
    ht[buckets[i]].value += valueCol[i];
  }
}
```

Varint hash method


```rust
pub enum HashMethodKind {
    Serializer(HashMethodSerializer),
    KeysU8(HashMethodKeysU8),
    KeysU16(HashMethodKeysU16),
    KeysU32(HashMethodKeysU32),
    KeysU64(HashMethodKeysU64),
}
```

```rust
impl DataBlock {
    pub fn choose_hash_method(
        block: &DataBlock,
        column_names: &[String],
    ) -> Result<HashMethodKind> {
        let mut group_key_len: usize = 0;
        for col: &String in column_names {
            let column: &DataColumn = block.try_column_by_name(col)?;
            let typ: DataType = column.data_type();
            if typ.is_integer() {
                group_key_len += typ.numeric_byte_size()?;
            } else {
                return Ok(HashMethodKind::Serializer(HashMethodSerializer::default()));
            }
        }      sundy-li, 5 months ago • [groupby] add KeysU8/KeysU16/KeysU32/KeysU64 hash…
        match group_key_len {
            1 => Ok(HashMethodKind::KeysU8(HashMethodKeysU8::default())),
            2 => Ok(HashMethodKind::KeysU16(HashMethodKeysU16::default())),
            3..=4 => Ok(HashMethodKind::KeysU32(HashMethodKeysU32::default())),
            5..=8 => Ok(HashMethodKind::KeysU64(HashMethodKeysU64::default())),
            _ => Ok(HashMethodKind::Serializer(HashMethodSerializer::default())),
        }
    }
```

This makes hashmap much smaller and efficient

Memory ARENA

Memory ARENA by using `Bumpalo crate`

States are much more cache friendly.

```
96
97      let arena: Bump = Bump::new();
98
99      let (layout: Layout, offsets_aggregate_states: Vec<usize>) = unsafe { get_layout_offsets(&funcs) };
100
101     let places: Vec<usize> = {
102         let place: StateAddr = arena.alloc_layout(layout).into();
103         funcs: Vec<Arc<dyn AggregateFunction>>
104             .iter(): Iter<Arc<dyn AggregateFunction>>      sundy-li, 4 months ago • aggregator usize
105             .enumerate(): impl Iterator<Item = (usize, …)>
106             .map(|(idx: usize, func: &Arc<dyn AggregateFunction>)| {
107                 let arg_place: StateAddr = place.next(offset: offsets_aggregate_states[idx]);
108                 func.init_state(arg_place);
109                 arg_place.addr()
110             }): impl Iterator<Item = usize>
111             .collect()
112     };
113
```

This makes hashmap much smaller and efficient.

Memory ARENA

Memory ARENA by using `Bumpalo crate`

States are much more cache friendly.

```
96
97      let arena: Bump = Bump::new();
98
99      let (layout: Layout, offsets_aggregate_states: Vec<usize>) = unsafe { get_layout_offsets(&funcs) };
100
101     let places: Vec<usize> = {
102         let place: StateAddr = arena.alloc_layout(layout).into();
103         funcs: Vec<Arc<dyn AggregateFunction>>
104             .iter(): Iter<Arc<dyn AggregateFunction>>      sundy-li, 4 months ago • aggregator usize
105             .enumerate(): impl Iterator<Item = (usize, …)>
106             .map(|(idx: usize, func: &Arc<dyn AggregateFunction>)| {
107                 let arg_place: StateAddr = place.next(offset: offsets_aggregate_states[idx]);
108                 func.init_state(arg_place);
109                 arg_place.addr()
110             }): impl Iterator<Item = usize>
111             .collect()
112     };
113
```

This makes hashmap much smaller and efficient.

States combination


```rust
/// layout must ensure to be aligned
pub unsafe fn get_layout_offsets(funcs: &[AggregateFunctionRef]) -> (Layout, Vec<usize>) {
    let mut max_align: usize = 0;
    let mut total_size: usize = 0;

    let mut offsets: Vec<usize> = Vec::with_capacity(funcs.len());
    for func: &Arc<dyn AggregateFunction> in funcs {
        let layout: Layout = func.state_layout();
        let align: usize = layout.align();

        total_size = (total_size + align - 1) / align * align;

        offsets.push(total_size);

        max_align = max_align.max(align);
        total_size += layout.size();
    }
    let layout: Layout = Layout::from_size_align_unchecked(total_size, max_align);
    (layout, offsets)
}
```

States could share a single address with specific layout offset

Div & Rem operators are very slow in databend!

number % 3, number / 3

Why?

	Integer			Floating point		
Operation	Latency	Issue	Capacity	Latency	Issue	Capacity
Addition	1	1	4	3	1	1
Multiplication	3	1	1	5	1	2
Division	3–30	3–30	1	3–15	3–15	1

Figure 5.12 Latency, issue time, and capacity characteristics of reference machine operations. Latency indicates the total number of clock cycles required to perform the actual operations, while issue time indicates the minimum number of cycles between two independent operations. The capacity indicates how many of these operations can be issued simultaneously. The times for division depend on the data values.

Improve constant div

Transform mod to multiply operator

```
uint32_t d = ...;// your divisor > 0

uint64_t c = UINT64_C(0xFFFFFFFFFFFFFFFF) / d + 1;

// fastmod computes (n mod d) given precomputed c
uint32_t fastmod(uint32_t n ) {
  uint64_t lowbits = c * n;
  return ((__uint128_t)lowbits * d) >> 64;
 }
```

4X - 10X improvement

Refer to:

Blog:
https://lemire.me/blog/2019/02/08/faster-remainders-when-the-divisor-is-a-constant-beating-compilers-and-libdivide/
Paper: https://arxiv.org/abs/1902.01961
Go: https://github.com/bmkessler/fastdiv
Rust: https://docs.rs/strength_reduce/0.2.3/strength_reduce/

Aggregation optimization

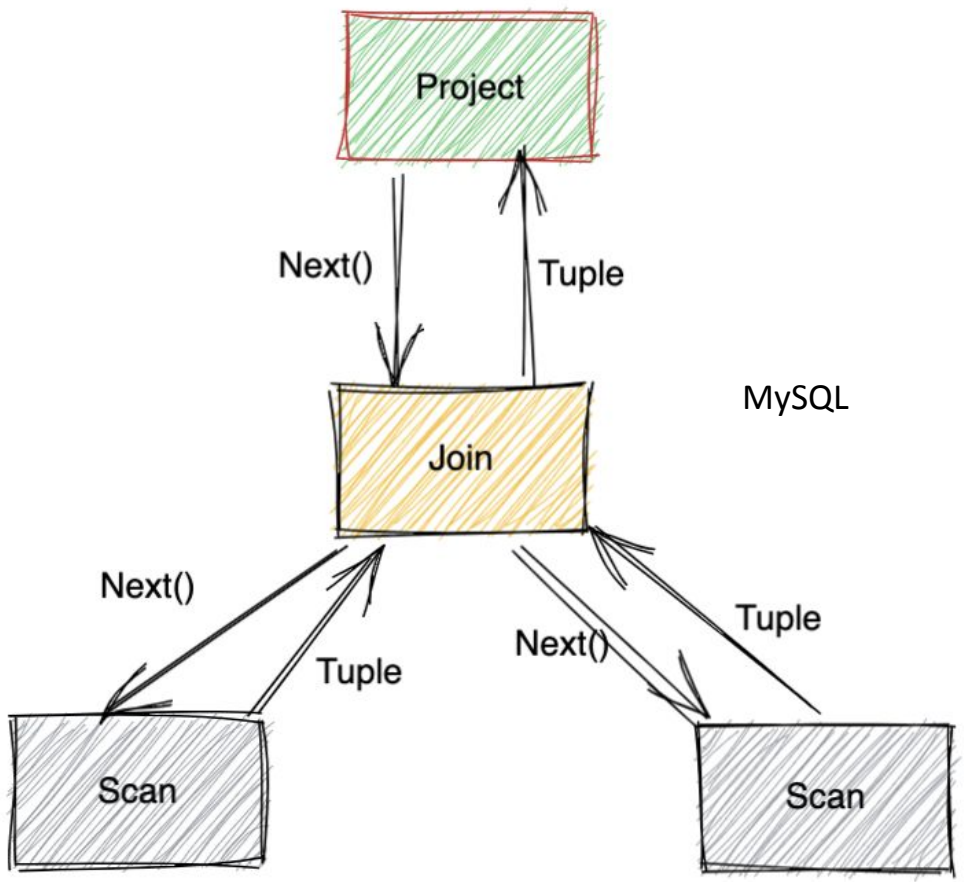
Sad story reaches happy ending

After some more optimizations,
GroupBy performance is as good as ClickHouse now

3 Pipeline/Dataflow executor

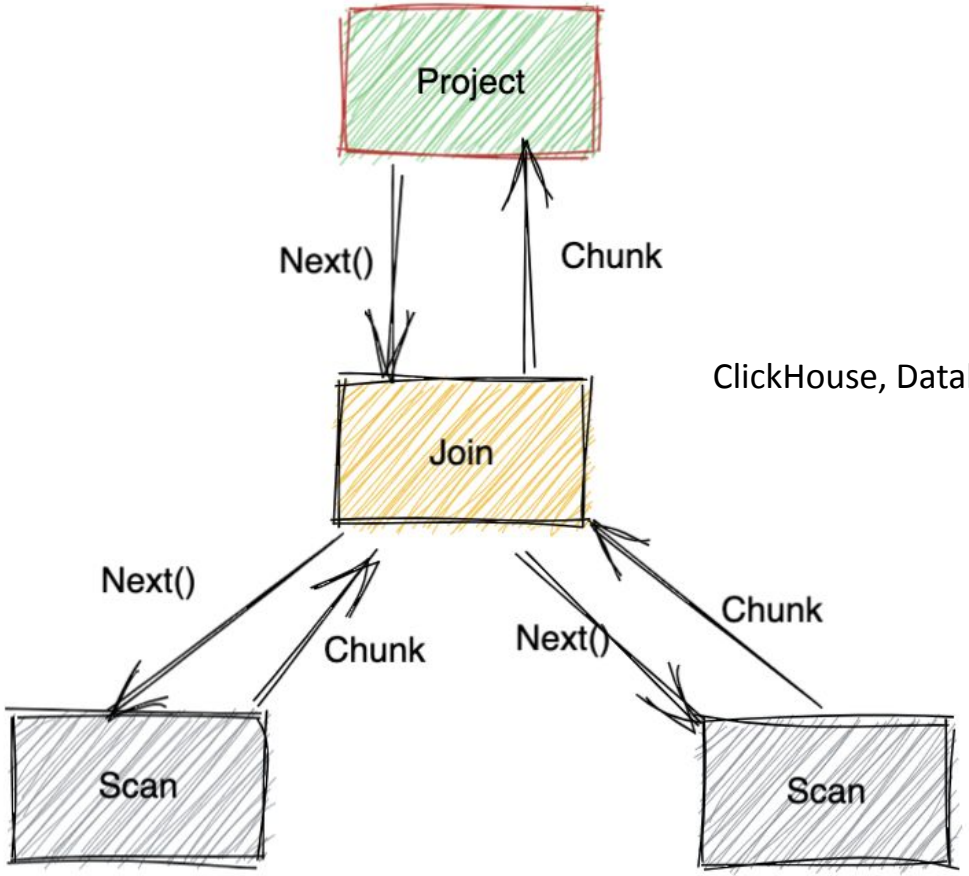
Volcano model

Volcano row model



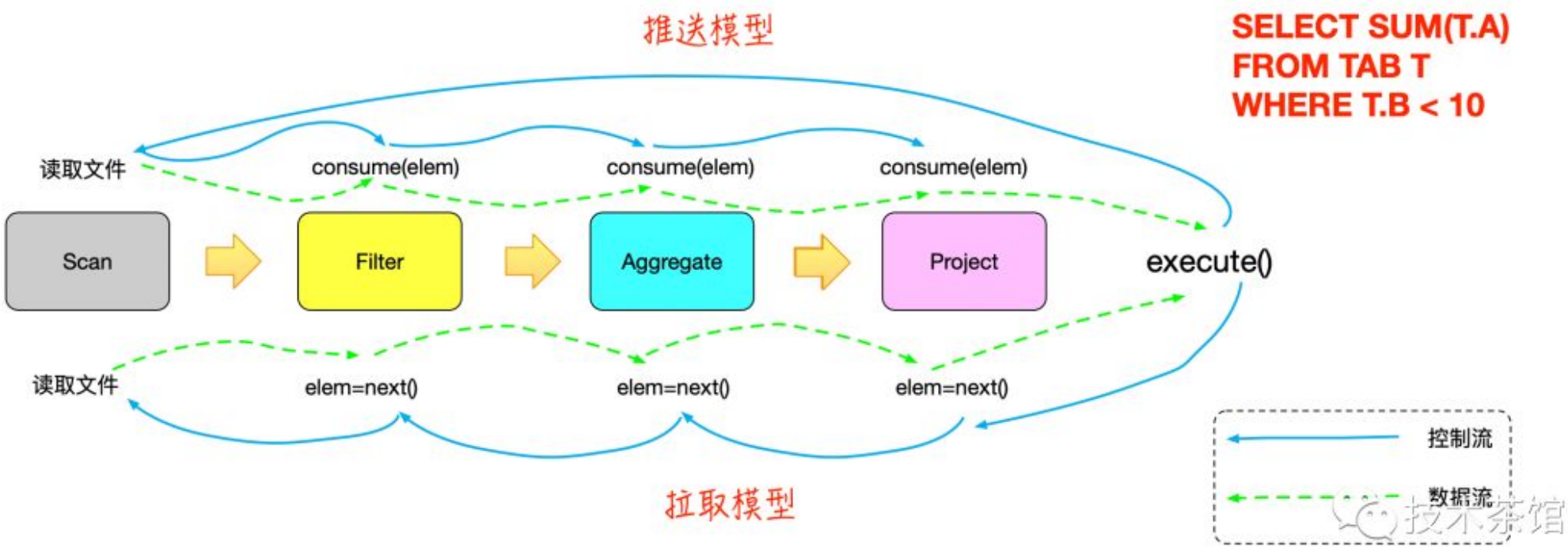
MySQL

Volcano batch model



ClickHouse, Databend

Pull vs pushed



【数据库内核】物理计算引擎Push模型之编译执行_Night_ZW的博客-CSDN博客_数据库编译执行

Pipeline example

1. ClickHouse Pipeline
2. DuckDB Pipeline engine

Compile execution

Consider Query:

```
SELECT SUM(a * (100 - b)  * (100 - c)) FROM table;
```

```
UInt64 (idx n,int *col1 ,chr* col2 ,chr* col3) {
        UInt64 sum = 0;
        for (idx i=0; i<n; i++) {
                sum += col1[i]*((100−col2[i])*(100+col3[i]));
        }
}
```

PURE C

```
map_OP_T_col_T_col(idx n,T* res ,T* col1 ,T* col2){
        for(int i=0; i<n; i++) {
                res[i] = op(col1[i]  col2[i];)
        }
}

// Vectorized code :
map_add_chr_val_chr_col (LEN,tmp1,&one , l
discount );
map_sub_chr_val_chr_col(LEN,tmp2,&one,l tax);
map_mul_chr_col_chr_col(LEN,tmp3,tmp1,tmp2);

// sum up the results using simd
simd_sum(tmp3)
```

vectorized + simd sum

Compile execution

Consider Query:

```
SELECT SUM(a * (100 - b)  * (100 - c)) FROM table;
```

```
UInt64 (idx n,int *col1 ,chr* col2 ,chr* col3) {
        UInt64 sum = 0;
        for (idx i=0; i<n; i++) {
                sum += col1[i]*((100−col2[i])*(100+col3[i]));
        }
}
```

PURE C

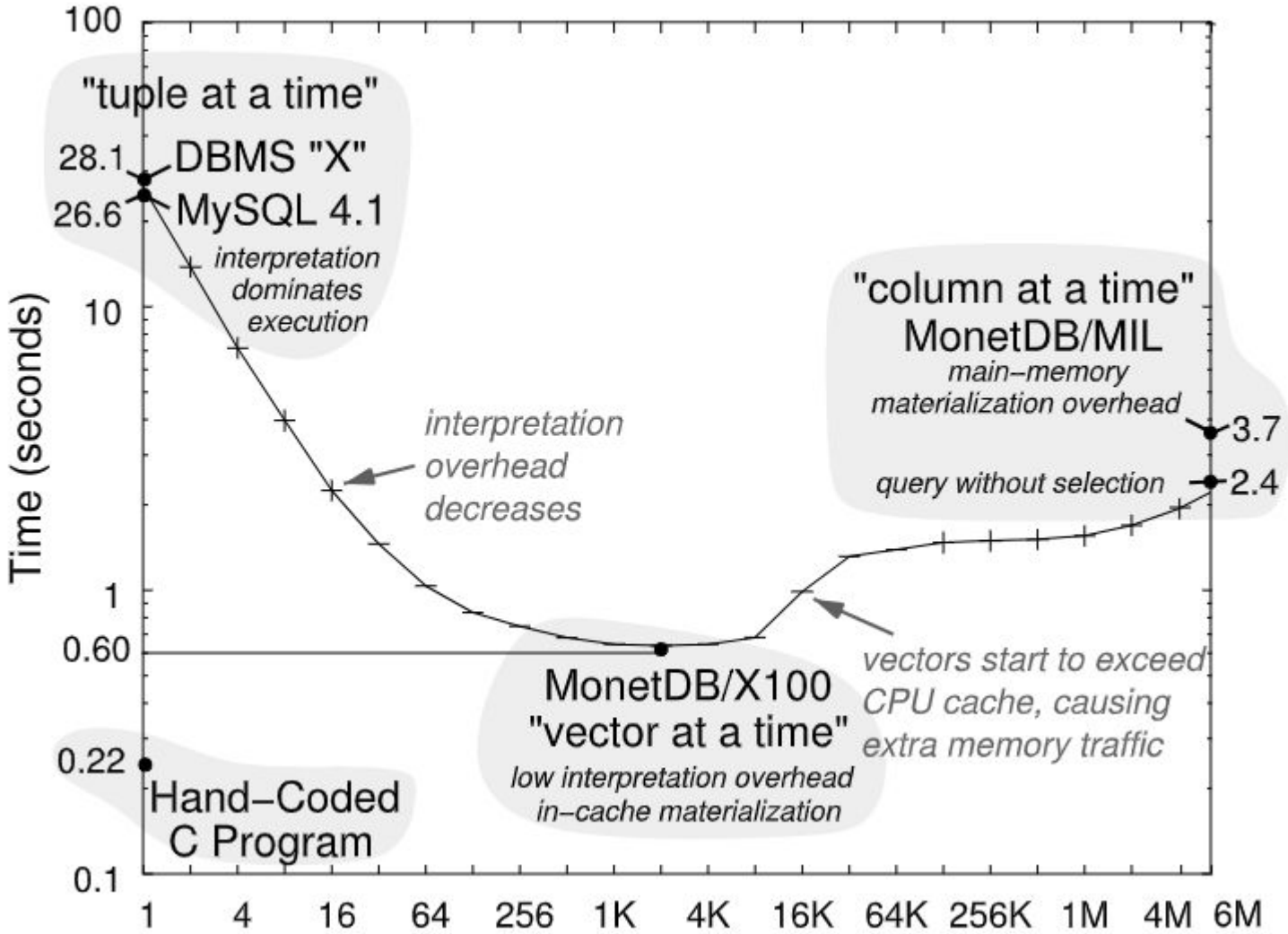
```
map_OP_T_col_T_col(idx n,T* res ,T* col1 ,T* col2){
        for(int i=0; i<n; i++) {
                res[i] = op(col1[i]  col2[i];)
        }
}

// Vectorized code :
map_add_chr_val_chr_col (LEN,tmp1,&one , l
discount );
map_sub_chr_val_chr_col(LEN,tmp2,&one,l tax);
map_mul_chr_col_chr_col(LEN,tmp3,tmp1,tmp2);

// sum up the results using simd
simd_sum(tmp3)
```

vectorized + simd sum

Performances



```
select   *
from     R1,R3,
         (select   R2.z,count(*)
         from      R2
         where     R2.y=3
         group by  R2.z) R2
where    R1.x=7 and R1.a=R3.b and R2.z=R3.c
```

Figure 2: Example Query

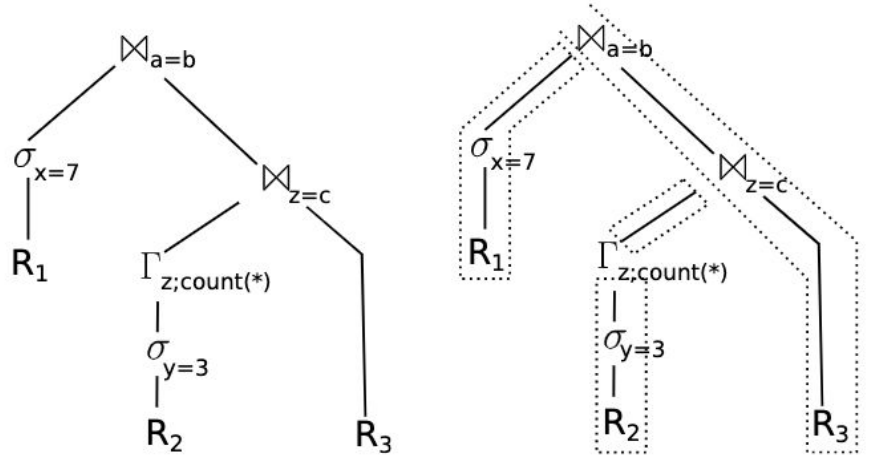


Figure 3: Example Execution Plan for Figure 2

initialize memory of $\bowtie_{a=b}$, $\bowtie_{c=z}$, and Γ_z
for each tuple t in R_1
  if $t.x = 7$
    materialize t in hash table of $\bowtie_{a=b}$
for each tuple t in R_2
  if $t.y = 3$
    aggregate t in hash table of Γ_z
for each tuple t in Γ_z
  materialize t in hash table of $\bowtie_{z=c}$
for each tuple t_3 in R_3
  for each match t_2 in $\bowtie_{z=c}[t_3.c]$
    for each match t_1 in $\bowtie_{a=b}[t_3.b]$
      output $t_1 \circ t_2 \circ t_3$

Figure 4: Compiled query for Figure 3

https://www.vldb.org/pvldb/vol4/p539-neumann.pdf

Compile vs vectorized execution


```
query(...)
  // build hash table
  for(i = 0; i < S.size(); i++)
    ht.insert(<S.att1[i], S.att2[i]>, S.att3[i])
  // probe hash table
  for(i = 0; i < R.size(); i++)
    int k1 = R.att1[i]
    string* k2 = R.att2[i]
    int hash = hash(k1, k2)
    for(Entry* e = ht.find(hash); e; e = e->next)
      if(e->key1 == k1 && e->key2 == *k2)
        ... // code of parent operator
```

(a) Code generated for hash join

```
class HashJoin
  Primitives probeHash_, compareKeys_, buildGather_;
  ...
int HashJoin::next()
  ... // consume build side and create hash table
  int n = probe->next()// get tuples from probe side
  // *Interpretation*: compute hashes
  vec<int> hashes = probeHash_.eval(n)
  // find hash candidate matches for hashes
  vec<Entry*> candidates = ht.findCandidates(hashes)
  // matches: int references a position in hashes
  vec<Entry*, int> matches = {}
  // check candidates to find matches
  while(candidates.size() > 0)
    // *Interpretation*
    vec<bool> isEqual = compareKeys_.eval(n, candidates)
    hits, candidates = extractHits(isEqual, candidates)
    matches += hits
  // *Interpretation*: gather from hash table into
  // buffers for next operator
  buildGather_.eval(matches)
  return matches.size()
```

(b) Vectorized code that performs a hash join

[Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask]

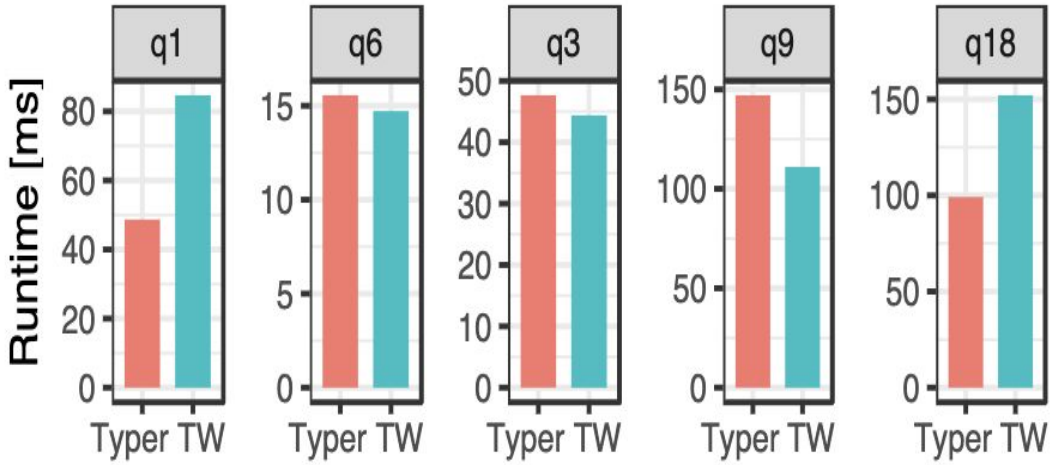


Figure 3: **Performance** – *TPC-H SF=1, 1 thread*

Table 1: **CPU Counters** – *TPC-H SF=1, 1 thread, normalized by number of tuples processed in that query*

		cycles	IPC	instr.	L1 miss	LLC miss	branch miss
Q1	Typer	34	2.0	68	0.6	0.57	0.01
Q1	TW	59	2.8	162	2.0	0.57	0.03
Q6	Typer	11	1.8	20	0.3	0.35	0.06
Q6	TW	11	1.4	15	0.2	0.29	0.01
Q3	Typer	25	0.8	21	0.5	0.16	0.27
Q3	TW	24	1.8	42	0.9	0.16	0.08
Q9	Typer	74	0.6	42	1.7	0.46	0.34
Q9	TW	56	1.3	76	2.1	0.47	0.39
Q18	Typer	30	1.6	46	0.8	0.19	0.16
Q18	TW	48	2.1	102	1.9	0.18	0.37

data size by a factor of 10, causes 0.5 additional cache misses per tuple“).

[Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask]

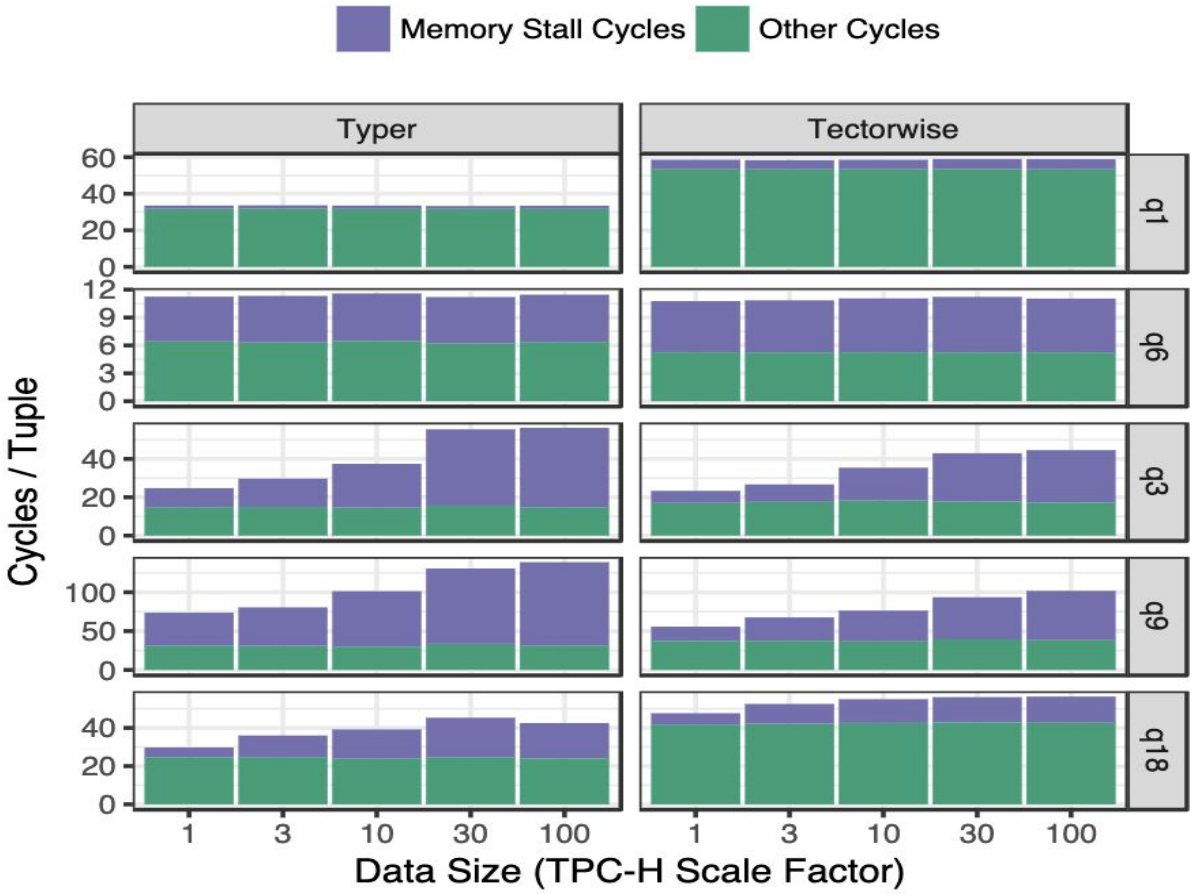


Figure 4: **Memory Stalls** – *TPC-H, 1 thread*

[Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask]

Compile vs vectorized execution

Compiled execution

Pros:

- Less instructions
- Cache friendly, less materialization

Cons:

- May have more branch miss in complex query
- May have more complex loops which break CPU's pipeline and cause more branch prediction miss
- Hard to implement

Pros:

- Easy to implement
- Simd & CPU's pipeline
- Work better in join & aggregation
- Can optimize branch prediction by vector selector

Cons:

- Intermediate state needs more materialization
- More instructions

THANK YOU FOR WATCHING