Yandex

Yandex

Data processing into ClickHouse

Nikolai Kochetov, ClickHouse developer

Agenda

- Data layout and compression
- In-memory layout and data processing
- Pipelining and parallelism
- Specialized data structures

Data layout and compression

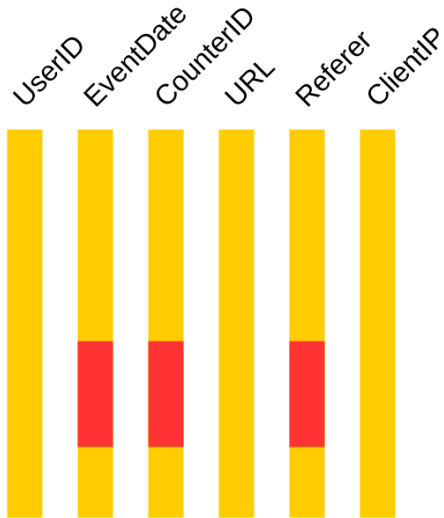
Column-Oriented DBMS

General ideas

- Separate column is stored in separate file (or several files)
- Only affected columns are read
- Columnar data representation in memory

Additional concepts

- Sparse index
- Per-column compression



Compression

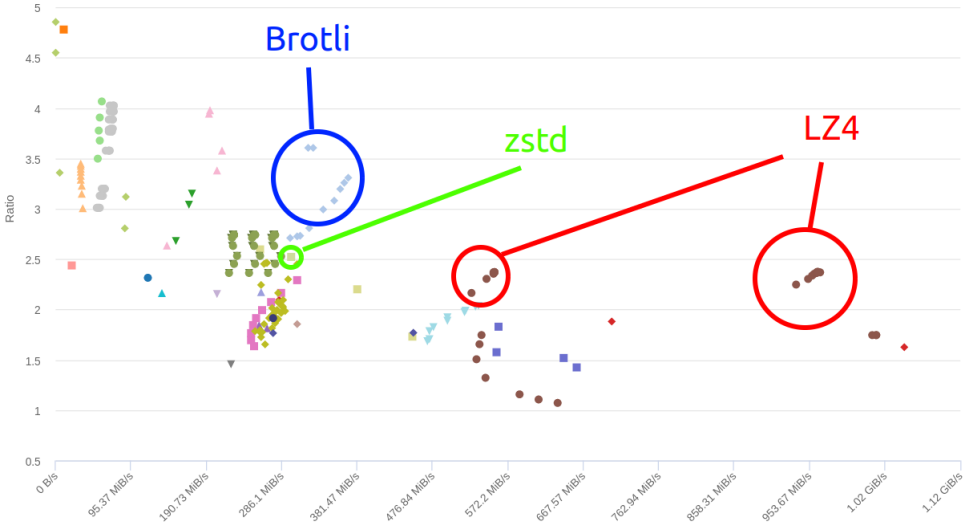
Highly customizable in `CREATE TABLE` statement

```
CREATE TABLE codec_example
(
    `dt`         DateTime,  -- default CODEC is LZ4HC(9)
    `dt_none`    DateTime CODEC(NONE),
    `dt_lz4_4`   DateTime CODEC(LZ4HC(4)),
    `dt_zstd`    DateTime CODEC(ZSTD),
    `dt_dd_lz4` DateTime CODEC(DoubleDelta, LZ4HC)  -- combined
)
ENGINE = MergeTree
ORDER BY dt
```

Compression ratio vs decompression speed

Intel Xeon E3-1225V3, enwik8

https://quixdb.github.io/squash-benchmark



Compression ratio

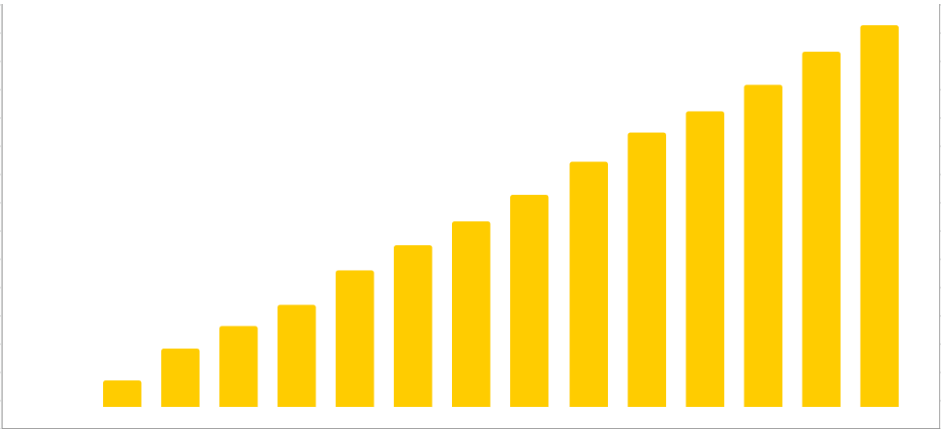
```
SELECT
  column,
  formatReadableSize(column_data_compressed_bytes) AS compressed,
  formatReadableSize(column_data_uncompressed_bytes) AS uncompressed,
  column_data_uncompressed_bytes / column_data_compressed_bytes AS r
FROM system.parts_columns
WHERE (table = 'codec_example') AND active ORDER BY r ASC
```

column	compressed	uncompressed	r
dt_none	67.73 MiB	67.70 MiB	0.999618408127124
dt	3.06 MiB	67.70 MiB	22.156958788868835
dt_lz4_4	3.06 MiB	67.70 MiB	22.156958788868835
dt_zstd	1.08 MiB	67.70 MiB	62.91648262048673
dt_dd_lz4	938.17 KiB	67.70 MiB	73.89642182401099

Data transformation chain

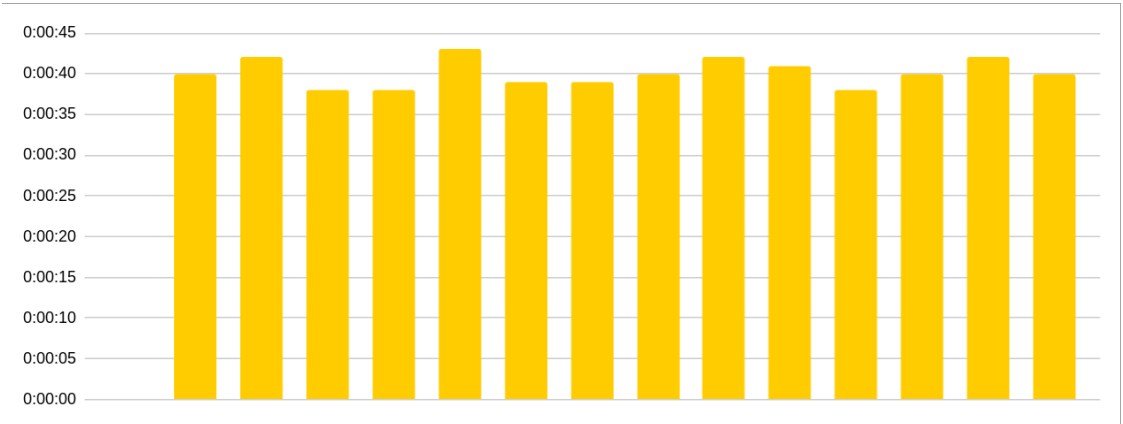
Time series data

2020-02-20 20:20:00
2020-02-20 20:20:40
2020-02-20 20:21:22
2020-02-20 20:22:00
2020-02-20 20:22:38
2020-02-20 20:23:21
2020-02-20 20:24:00
2020-02-20 20:24:39
2020-02-20 20:25:19
2020-02-20 20:26:01
2020-02-20 20:26:42
2020-02-20 20:27:20
2020-02-20 20:28:00
2020-02-20 20:28:42
2020-02-20 20:29:22



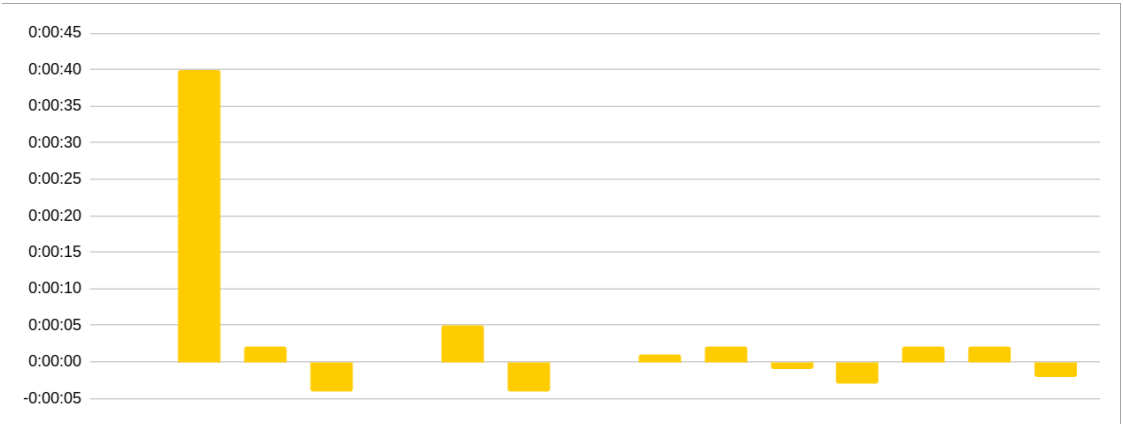
Data transformation chain

Time series data -> Delta



Data transformation chain

Time series data -> Delta -> Delta



Data transformation chain

Time series data -> Delta -> Delta -> variable length encoding

	Prefix	Sign	abs(double_delta - 1)	#bits
double_delta = 0	0			1
-63 < double_delta < 64	10	x	xxxxxx	9
-255 < double_delta < 256	110	x	xxxxxxxx	12
-2047 < double_delta < 2048	1110	x	xxxxxxxxxxx	16
double_delta fits into 32-bit	11110	x	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	37
otherwise	11111	x	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	69

Time series data -> DoubleDelta

Time series data -> DoubleDelta -> LZ4HC

Read time

Test query

```
SELECT dt_dd_lz4 FROM codec_example FORMAT Null
```

Enable `system.query_log`

```
SET log_queries = 1
```

xml config:
clickhouse.tech/docs/en/operations/server_settings/settings/#server_settings-query-log

Drop FS cache

```
$ echo 3 | sudo tee /proc/sys/vm/drop_caches
```

Read time

Profile events are in `system.query_log`

```
SELECT
    pe.Names,
    pe.Values
FROM system.query_log
ARRAY JOIN ProfileEvents AS pe
WHERE event_date = today() AND type = 'QueryFinish'
      AND query_id = '...'

┌─pe.Names──────────────────────────┬─pe.Values─┐
│ DiskReadElapsedMicroseconds       │    123970 │
│ RealTimeMicroseconds              │    596084 │
...
```

Read time

Column	Query Time, sec	RealTime, sec (total for threads)	DiskReadTime, sec (total for threads)	DiskRead ratio
dt_none	0.289	9.385	8.161	0.870
dt_lz4_4	0.030	0.785	0.432	0.550
dt	0.030	0.877	0.368	0.420
dt_zstd	0.021	0.549	0.168	0.306
dt_dd_lz4	0.022	0.596	0.124	0.208

Higher compression rate means

- **less** IO and **more** CPU time
- **less** real time for IO-bounded queries

Read time

```
select sum(halfMD5(halfMD5(dt))) from codec_example
```

Column	Query Time, sec	RealTime, sec (total for threads)	DiskReadTime, sec (total for threads)	DiskRead ratio
dt_none	0.436	14.481	3.460	0.239
dt_lz4_4	0.357	12.342	0.412	0.033
dt	0.346	11.869	0.564	0.048
dt_zstd	0.351	11.703	0.176	0.015
dt_dd_lz4	0.357	13.118	0.200	0.015

For CPU-bounded queries decompression time is usually insignificant

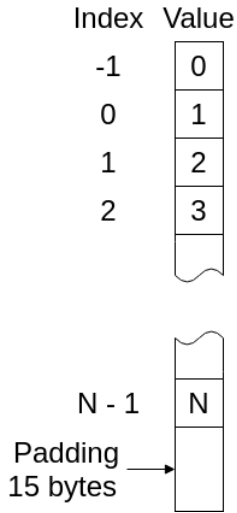
In-memory layout and data processing

Data processing

- Data is processed by blocks
- Block stores slices of columns
- Column is represented in one or several buffers

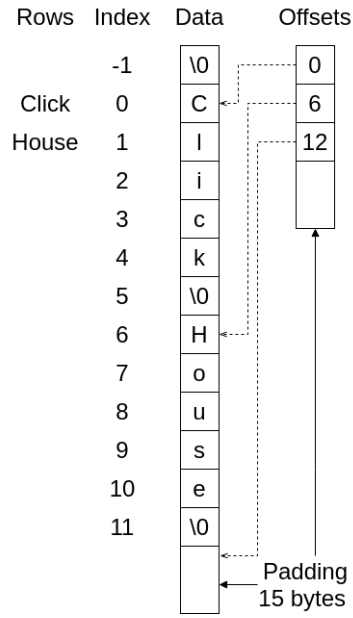
Integers

- Single buffer
- Stores zero at position -1
- Extra 15 bytes are allocated at array's tail



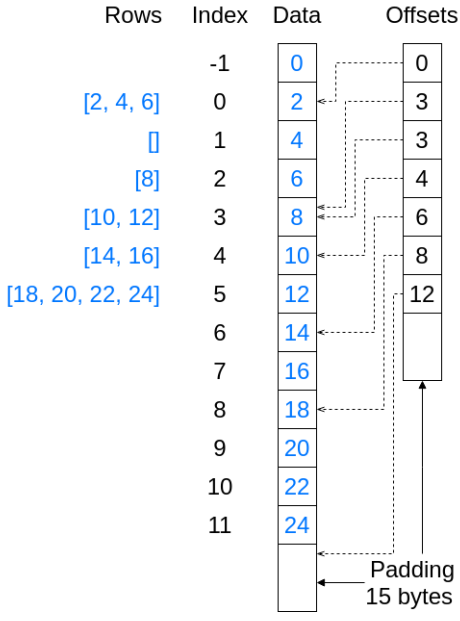
Strings

- Buffers with data and offsets
- Offsets are prefix sums of sizes
- Store \0 at string's end



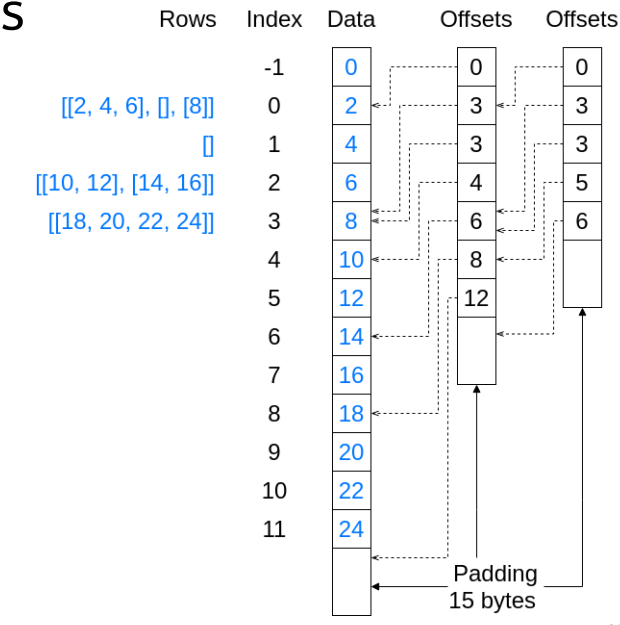
Arrays

- As well as Strings
- Offsets are stored in a separate file on FS



N-dimensional Arrays

- N-dimensional Array is an Array of (N-1)-dimensional Arrays
- N-dimensional Offsets are Offsets for (N-1)-dimensional offsets
- Natural generalization of 1-dimensional Arrays



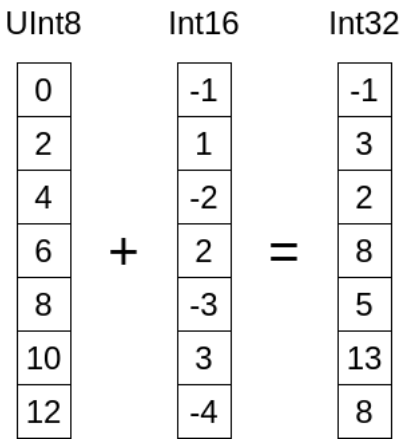
Functions

Concepts

- Pure (with some exceptions)
- Strong typing
- Multiple overloads

Per-columns execution

- Less virtual calls
- SIMD optimizations
- Complication for UDF



SIMD operations

```
int memcmpSmallAllowOverflow15(const Char * a, size_t a_size,
                               const Char * b, size_t b_size)
{
    size_t min_size = std::min(a_size, b_size);

    for (size_t offset = 0; offset < min_size; offset += 16)
    {
        /// Compare 16 bytes at once
        uint16_t mask = _mm_movemask_epi8(_mm_cmpeq_epi8(
            _mm_loadu_si128(reinterpret_cast<const __m128i *>(a + offset)),
            _mm_loadu_si128(reinterpret_cast<const __m128i *>(b + offset))));

        if (~mask) /// if mask has zero bit (some bytes are different)
        {
            /// Find and compare first different bytes
            ...
        }
    }

    return detail::cmp(a_size, b_size);
}
```

SIMD operations

Main loop for `memcmpSmallAllowOverflow15`

```
0xa187fc0 : add    $0x10,%r8           ; offset += 16

0xa187fc4 : cmp    %r9,%r8             ; if (offset >= min_size)
0xa187fc7 : jae    0xa188008           ;     exit loop

0xa187fc9 : movdqu (%rdx,%r8,1),%xmm0  ; xmm0 = a[offset] (16 bytes)
0xa187fcf : movdqu (%rdi,%r8,1),%xmm1  ; xmm1 = b[offset] (16 bytes)
0xa187fd5 : pcmpeqb %xmm1,%xmm0        ; xmm0 = (xmm0 == xmm1)
                                       ;        (16 bytes at once)

0xa187fd9 : pmovmskb %xmm0,%eax        ; mask = `bit mask from xmm0`
0xa187fdd : xor    $0xffff,%ax         ; mask = ~mask
0xa187fe1 : je     0xa187fc0           ; if (mask == 0)
                                       ;     continue loop
```

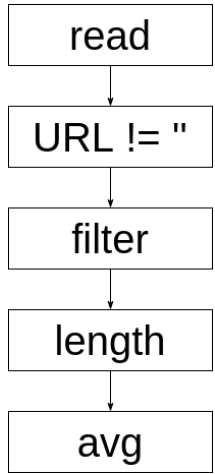
Pipelining and parallelism

Query Pipeline

```
SELECT avg(length(URL)) FROM hits WHERE URL != ''
```

Independent execution steps

- Read column **URL**
- Calculate expression **`URL != ''`**
- Filter column **URL**
- Calculate function **`length(URL)`**
- Calculate aggregate function **`avg`**

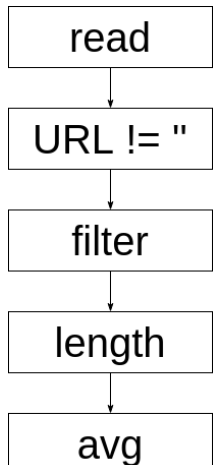


Query Pipeline

```
SELECT avg(length(URL)) FROM hits WHERE URL != ''
```

Properties

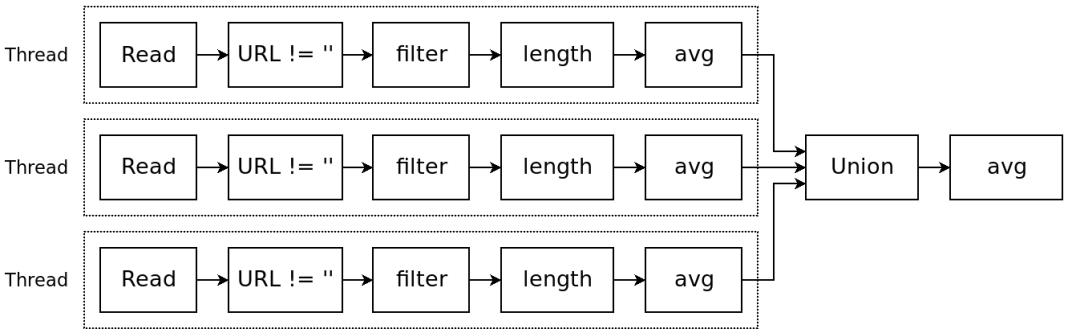
- Arbitrary graph
- Support parallel execution
- Dynamically changeable



Parallel Execution

```
SELECT avg(length(URL)) FROM hits WHERE URL != ''
```

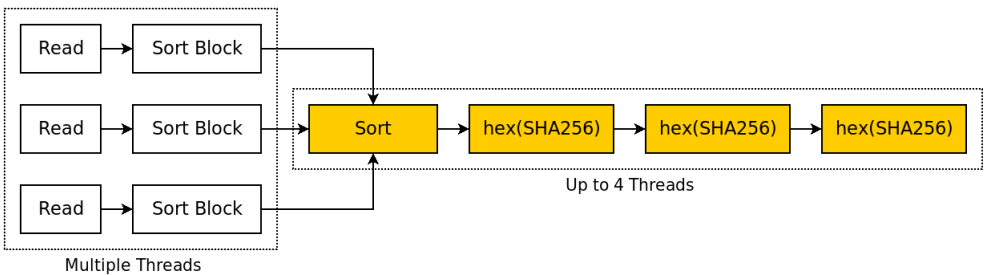
Parallelism by data



Parallel Execution

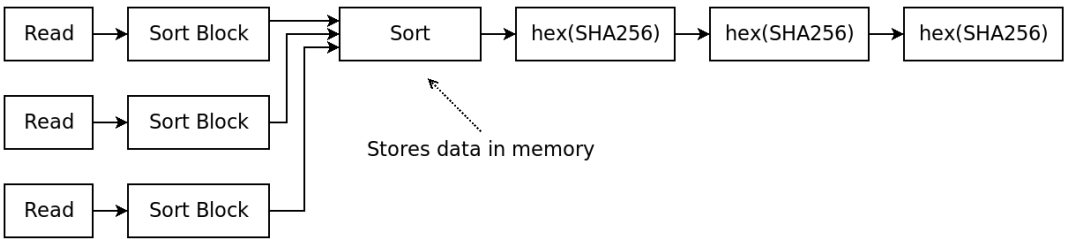
```
SELECT hex(SHA256(*)) FROM (
    SELECT hex(SHA256(*)) FROM (
        SELECT hex(SHA256(*)) FROM (
            SELECT URL FROM hits ORDER BY URL ASC)))
```

Vertical parallelism



Dynamic pipeline modification

Sometimes we need to change pipeline during execution

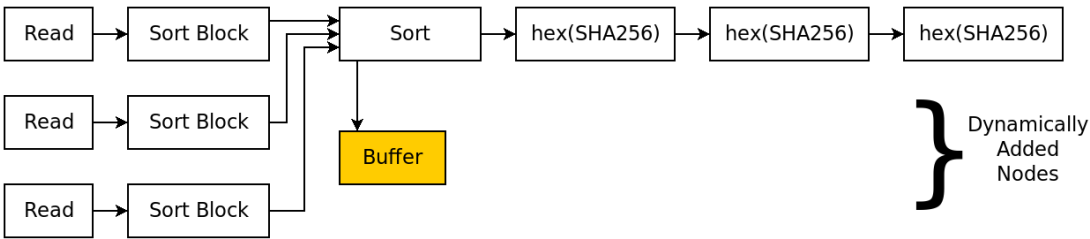


Sort stores all query data in memory

Set `max_bytes_before_external_sort = <some limit>`

Dynamic pipeline modification

Sometimes we need to change pipeline during execution

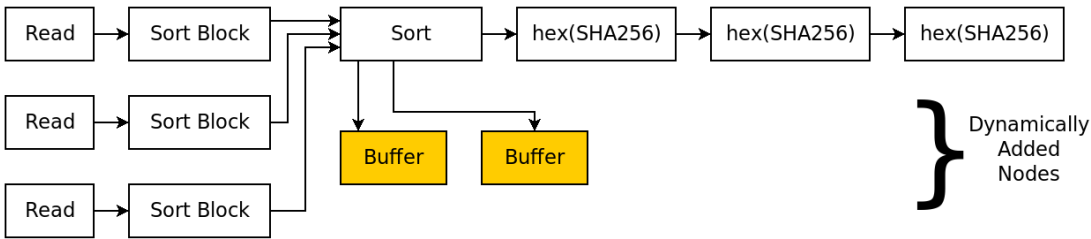


Sort stores all query data in memory

Set `max_bytes_before_external_sort = <some limit>`

Dynamic pipeline modification

Sometimes we need to change pipeline during execution

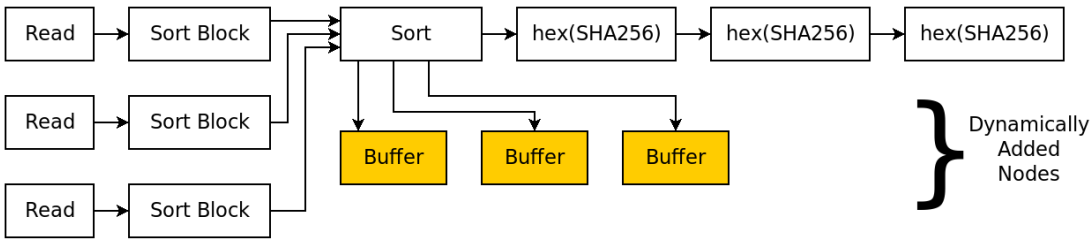


Sort stores all query data in memory

Set `max_bytes_before_external_sort = <some limit>`

Dynamic pipeline modification

Sometimes we need to change pipeline during execution

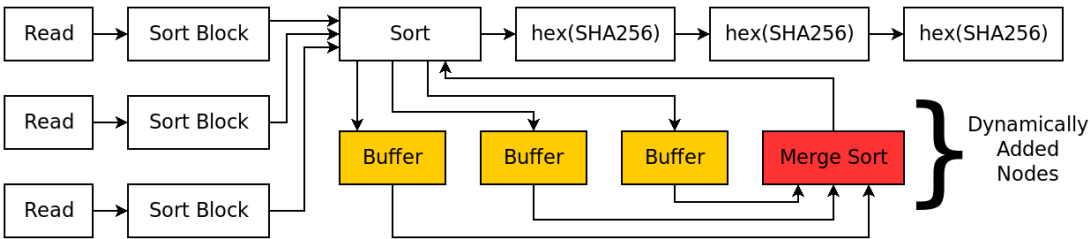


Sort stores all query data in memory

Set `max_bytes_before_external_sort = <some limit>`

Dynamic pipeline modification

Sometimes we need to change pipeline during execution



Sort stores all query data in memory

Set `max_bytes_before_external_sort = <some limit>`

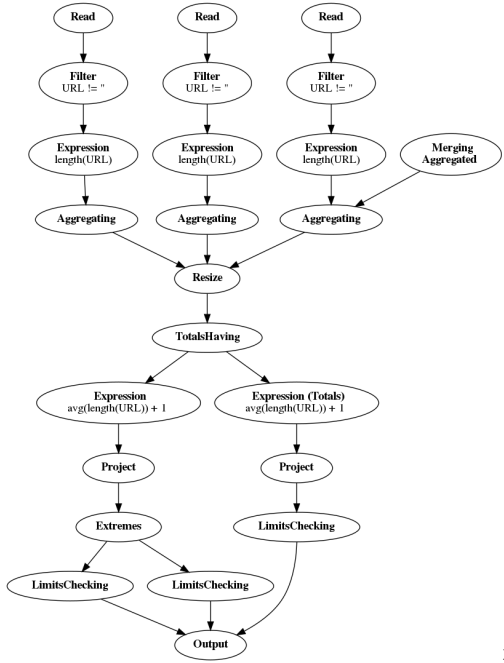
Query Pipeline

```
SELECT avg(length(URL)) + 1
FROM hits WHERE URL != ''
WITH TOTALS SETTINGS extremes = 1

┌─plus(avg(length(URL)), 1)─┐
│          85.3475007793562 │
└───────────────────────────┘

Totals:
┌─plus(avg(length(URL)), 1)─┐
│          85.3475007793562 │
└───────────────────────────┘

Extremes:
┌─plus(avg(length(URL)), 1)─┐
│          85.3475007793562 │
│          85.3475007793562 │
└───────────────────────────┘
```



Specialized data structures

Task analysis

Task example: string search.

Possible aspects of a task

- Approximate or exact search
- Substring or regexp
- Single or multiple `needles`
- Single or multiple `haystacks`
- Short or long strings
- Bytes, unicode code points, real words

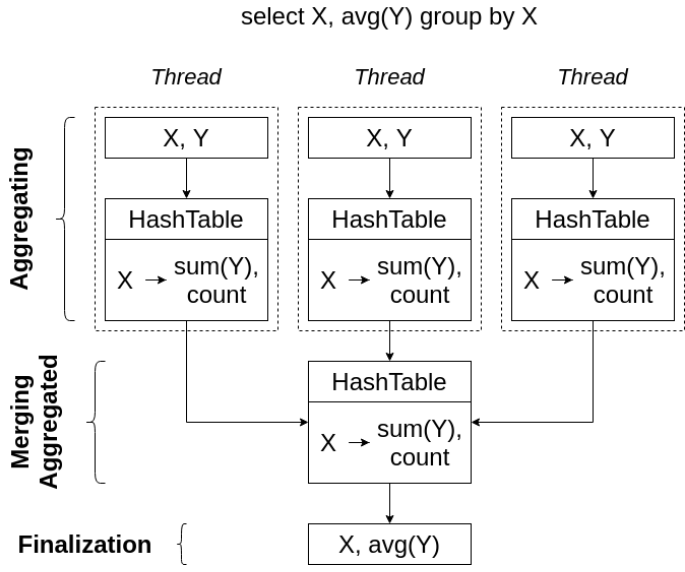
For every option can be created specialized algorithm

Concepts

- Take the best implementations
  Example: `simdjson`, `pdqsort`
- Improve existent algorithms
  `Volnitsky` -> `MultiVolnitsky`
  `memcpy` -> `memcpySmallAllowReadWriteOverflow15`
- Use more optimal specializations
  40 hash table implementations for `GROUP BY`
- Test performance on real data
  Per-commit tests on real (obfuscated) dataset with page hits
- Profiling

GROUP BY

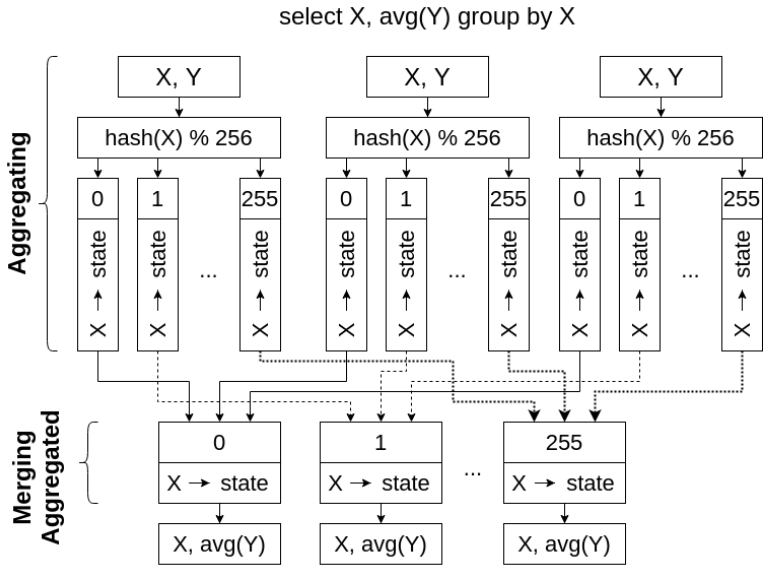
- Hash table
- Parallel
- Merging in single thread



GROUP BY

Two level

- Split data to 256 buckets
- Merging in multiple threads
- More efficient for remote queries



Hash table specializations

- 8-bit or 16 bit key
  lookup table
- 32, 64, 128, 256 bit key
  32-bit hash for aggregating, 64-bit hash for merging
- several fixed size keys
  represented as single integer if possible
- string key
  store pre-calculated hash in hash table
  small string optimization
- LowCardinality key
  pre-calculated hash for dictionaries
  pre-calculated bucket for consecutively repeated dictionaries

Conclusion

- Specialized algorithms and data structures are necessary for the best performance
- Use the same ideas in your projects
- Contribute: https://github.com/ClickHouse/ClickHouse

Thank you!

QA