ClickHouse performance optimization techniques

About me

Maksim, database management systems developer.

Performance optimization techniques

1. Performance optimization basics.
2. Low-level optimizations.
3. Abstractions and Algorithms.
4. Tools.
5. Examples.
6. Sorting.

Performance optimization basics

Performance optimization basics

CI/CD Pipeline.

Performance Tests.

Introspection.

Libraries.

Latency numbers.

My presentation from CPP Russia 2022 ClickHouse performance optimization practices:

https://www.youtube.com/watch?v=KedCUDZE9N4

CI/CD Pipeline

CI/CD Pipeline is required.

The more checks the better.

Functional, Integration tests. Fuzzers.

Performance tests.

Run all tests with all available sanitizers (ASAN, MSAN, UBSAN, TSAN).

Project specific tests. Example: custom stress tests, Jepsen tests.

Performance Tests

Must be able to report about performance changes (improvement or regression).

Collect different statistics during each performance test run. Can be useful for later debugging:

1. Processor metrics (CPU cycles, cache misses same as perf-stat).

2. Project specific profile events (read bytes from disk, transferred bytes over network, etc).

Performance Tests ClickHouse

Write test in special XML configuration.

```
<test>
    <substitutions>
        <substitution>
            <name>func</name>
                <values>
                    bitCount
                    bitNot
                    abs
                    ...
                </values>
        </substitution>

        <substitution>
            <name>expr</name>
                <values>
                    <value>number</value>
                    <value>toUInt32(number)</value>
                    ...
                </values>
        </substitution>
    </substitutions>

    <query>SELECT {func}({expr}) FROM numbers(100000000) FORMAT Null</query>
</test>
```

Performance Tests ClickHouse

ClickHouse performance comparison

Tested Commits ?

Old	New
commit 10fc871de95eec8be158c43b277783f81ce3238d (origin/master) Merge: 8f1d7e67c 822cc0fec Author: alexey-milovidov Date: Mon Jul 12 08:59:34 2021 +0300 Merge pull request #26232 from ClickHouse/read-pread Support for `pread` in `ReadBufferFromFile`	commit 36bc22df98d91feb69661aab3da26f4298d070b6 Author: Raúl Marín Date: Mon Jul 12 13:38:54 2021 +0200 Speed up addition of nullable native integers Real tested commit is: commit b908da84025c257fbaa3683bed25f16822bb3830 (HEAD -> master, pr) Merge: 10fc871de 36bc22df9 Author: Raúl Marín Date: Mon Jul 12 12:36:15 2021 +0000 Merge 36bc22df98d91feb69661aab3da26f4298d070b6 into 10fc871de95eec8be158c43b277783f81ce3238d

Changes in Performance ?

Old, s	New, s	Ratio of speedup (-) or slowdown (+)	Relative difference (new − old) / old	p < 0.01 threshold	Test	#	Query
0.418	0.266	-1.569x	-0.364	0.363	questdb_sum_int32	1	SELECT sum(x) FROM `zz_Int32 NULL_Memory`
0.136	0.095	-1.435x	-0.304	0.288	or_null_default	1	SELECT sumOrDefault(toNullable(number)) FROM numbers(100000000)
0.131	0.095	-1.385x	-0.278	0.277	sum	8	SELECT sum(toNullable(number)) FROM numbers(100000000)
0.180	0.142	-1.267x	-0.211	0.210	sum	9	SELECT sum(toNullable(toUInt32(number))) FROM numbers(100000000)

Performance Tests ClickHouse

It is not trivial to implement infrastructure for performance testing.

https://clickhouse.com/blog/testing-the-performance-of-click-house/

Performance Tests

Helps find performance regressions.

Tool that can help to find places where performance can be improved:

1. Try different allocators, different libraries.

2. Try different compiler options (loop unrolling, inline threshold)

3. Enable **AVX**/**AVX2**/**AVX512** for build.

Introspection

Ideally collect as much information as possible in user space.

Important to check if something is improved because of your change, or to understand why there is some degradation.

Can be used as additional statistics that are collected during performance tests.

Introspection

RealTimeMicroseconds, UserTimeMicroseconds, SystemTimeMicroseconds, SoftPageFaults, HardPageFaults using **getrusage** system call.

Collect **:taskstats** from procFS (Also support Netlink interface).

OSCPUVirtualTimeMicroseconds, OSCPUWaitMicroseconds (when **/proc/thread-self/schedstat** is available). OSIOWaitMicroseconds (when **/proc/thread-self/stat** is available). OSReadChars, OSWriteChars, OSReadBytes, OSWriteBytes (when **/proc/thread-self/io** is available)

https://man7.org/linux/man-pages/man2/getrusage.2.html

https://man7.org/linux/man-pages/man5/proc.5.html

Introspection

Project specific profile events (read bytes from disk, transferred bytes over network, etc).

Introspection ClickHouse Perf

```
SELECT PE.Names AS ProfileEventName, PE.Values AS ProfileEventValue
            FROM system.query_log ARRAY JOIN ProfileEvents AS PE
            WHERE query_id='344b07d9-9d7a-48f0-a17e-6f5f6f3d61f5'
            AND ProfileEventName LIKE 'Perf%';
```

```
┌─ProfileEventName─────────────────┬─ProfileEventValue─┐
│ PerfCpuCycles                    │       40496995274 │
│ PerfInstructions                 │       57259199973 │
│ PerfCacheReferences              │        2072274618 │
│ PerfCacheMisses                  │         146570206 │
│ PerfBranchInstructions           │        8675194991 │
│ PerfBranchMisses                 │         259531879 │
│ PerfStalledCyclesFrontend        │         813419527 │
│ PerfStalledCyclesBackend         │       15797162832 │
│ PerfCpuClock                     │       10587371854 │
│ PerfTaskClock                    │       10587382785 │
│ PerfContextSwitches              │              3009 │
│ PerfCpuMigrations                │               113 │
│ PerfMinEnabledTime               │       10584952104 │
│ PerfMinEnabledRunningTime        │        4348089512 │
│ PerfDataTLBReferences            │         465992961 │
│ PerfDataTLBMisses                │           5149603 │
│ PerfInstructionTLBReferences     │           1344998 │
│ PerfInstructionTLBMisses         │            181635 │
└──────────────────────────────────┴───────────────────┘
```

Libraries

Reusing existing libraries can significantly improve overall performance. In ClickHouse there are external libraries for:

1. Different algorithms for parsing floats, json (multiple libraries).

2. A lot of integrations.

3. Embedded storages.

4. LLVM for JIT compilation.

5. libcxx (C++ standard library).

Libraries

One of the easiest method to try to improve something is to reuse generic components with components from highly optimized library:

1. Abseil. Example: Replace **std::unordered_map** with **absl::flat_hash_map** if you do not need pointer stability.

2. Folly.

3. Boost.

Do not use outdated libraries.

Latency Comparison Numbers

```
Latency Comparison Numbers
----------------------------------
L1 cache reference                           0.5 ns
Branch mispredict                            5   ns
L2 cache reference                           7   ns 14x L1 cache
Mutex lock/unlock                           25   ns
Main memory reference                      100   ns 20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy             3,000   ns        3 us
Send 1K bytes over 1 Gbps network       10,000   ns       10 us
Read 4K randomly from SSD*             150,000   ns      150 us ~1GB/sec SSD
Read 1 MB sequentially from memory     250,000   ns      250 us
Round trip within same datacenter      500,000   ns      500 us
```

http://norvig.com/21-days.html#answers

Low-level optimizations

Autovectorization

```
void plus(int64_t * __restrict a,
    int64_t * __restrict b,
    int64_t * __restrict c,
    size_t size)
{
   for (size_t i = 0; i < size; ++i) {
       c[i] = b[i] + a[i];
   }
}
```

If there is no **__restrict** modifier specified for pointers, the compiler may not vectorize the loop.

Or it will vectorize the loop but put a couple of runtime checks at the beginning of the function to make sure that the arrays do not overlap.

Autovectorization

Compile program with AVX2 instruction set and **-fno-unroll-loops**.

```
$ /usr/bin/clang++-15 -mavx2 -fno-unroll-loops -O3 -S vectorization_example.cpp
```

Autovectorization

In the final assembly, there are two loops. Vectorized loop that processes 4 elements at a time:

```
.LBB0_4:                                # =>This Inner Loop Header: Depth=1
        vmovdqu (%rdi,%rax,8), %ymm0
        vpaddq  (%rsi,%rax,8), %ymm0, %ymm0
        vmovdqu %ymm0, (%rdx,%rax,8)
        addq    $4, %rax
        cmpq    %rax, %r8
        jne     .LBB0_4
```

Scalar loop:

```
LBB0_6:                                 # =>This Inner Loop Header: Depth=1
        movq    (%rdi,%r8,8), %rax
        addq    (%rsi,%r8,8), %rax
        movq    %rax, (%rdx,%r8,8)
        incq    %r8
        cmpq    %r8, %rcx
        jne     .LBB0_6
```

Autovectorization

Additionally, if we compile this example without **-fno-unroll-loops** and look at the generated loop, we will see that compiler unrolled vectorized loop, which now processes 16 elements at a time.

```
.LBB0_4:                                # =>This Inner Loop Header: Depth=1
        vmovdqu (%rdi,%rax,8), %ymm0
        vmovdqu 32(%rdi,%rax,8), %ymm1
        vmovdqu 64(%rdi,%rax,8), %ymm2
        vmovdqu 96(%rdi,%rax,8), %ymm3
        vpaddq  (%rsi,%rax,8), %ymm0, %ymm0
        vpaddq  32(%rsi,%rax,8), %ymm1, %ymm1
        vpaddq  64(%rsi,%rax,8), %ymm2, %ymm2
        vpaddq  96(%rsi,%rax,8), %ymm3, %ymm3
        vmovdqu %ymm0, (%rdx,%rax,8)
        vmovdqu %ymm1, 32(%rdx,%rax,8)
        vmovdqu %ymm2, 64(%rdx,%rax,8)
        vmovdqu %ymm3, 96(%rdx,%rax,8)
        addq    $16, %rax
        cmpq    %rax, %r8
        jne     .LBB0_4
```

Autovectorization

There is a very useful tool that can help you identify places where the compiler does or does not perform vectorization to avoid assembly checking.

You can add **-Rpass=loop-vectorize**, **-Rpass-missed=loop-vectorize** and **-Rpass-analysis=loop-vectorize** options to clang. There are similar options for gcc.

```
$ /usr/bin/clang++-15 -mavx2 -Rpass=loop-vectorize -Rpass-missed=loop-vectorize
        -Rpass-analysis=loop-vectorize -O3

vectorization_example.cpp:7:5: remark: vectorized loop (vectorization width: 4,
    interleaved count: 4) [-Rpass=loop-vectorize]
    for (size_t i = 0; i < size; ++i) {
```

Autovectorization

```
class SumFunction
{
public:
    void sumIf(int64_t * values, int8_t * filter, size_t size);

    int64_t sum = 0;
};

void SumFunction::sumIf(int64_t * values, int8_t * filter, size_t size)
{
    for (size_t i = 0; i < size; ++i) {
        sum += filter[i] ? 0 : values[i];
    }
}
```

Autovectorization

```
/usr/bin/clang++-15 -mavx2 -O3 -Rpass-analysis=loop-vectorize -Rpass=loop-vectorize
    -Rpass-missed=loop-vectorize -c vectorization_example.cpp

vectorization_example.cpp:31:13: remark: loop not vectorized: unsafe dependent
memory operations in loop. Use #pragma loop distribute(enable) to allow loop
distribution to attempt to isolate the offending operations into a separate loop
Unknown data dependence. Memory location is the same as accessed at
vectorization_example.cpp:31:13 [-Rpass-analysis=loop-vectorize]
        sum += filter[i] ? 0 : values[i];

        vectorization_example.cpp:28:9: remark: loop not vectorized
[-Rpass-missed=loop-vectorize]
        for (size_t i = 0; i < size; ++i) {
```

Autovectorization

Make local sum inside **sumIf** function:

```
class SumFunction
{
public:
    void sumIf(int64_t * values, int8_t * filter, size_t size);

    int64_t sum = 0;
};

void SumFunction::sumIf(int64_t * values, int8_t * filter, size_t size)
{
    int64_t local_sum = 0;

    for (size_t i = 0; i < size; ++i) {
        local_sum += filter[i] ? 0 : values[i];
    }

    sum += local_sum;
}
```

Autovectorization

```
/usr/bin/clang++-15 -mavx2 -O3 -Rpass-analysis=loop-vectorize -Rpass=loop-vectorize
-Rpass-missed=loop-vectorize -c vectorization_example.cpp

vectorization_example.cpp:31:5: remark: vectorized loop (vectorization width: 4,
    interleaved count: 4) [-Rpass=loop-vectorize]
    for (size_t i = 0; i < size; ++i) {
```

Autovectorization

```
.LBB0_5:                                # =>This Inner Loop Header: Depth=1
        vmovd   (%rdx,%rax), %xmm5              # xmm5 = mem[0],zero,zero,zero
        vmovd   4(%rdx,%rax), %xmm6             # xmm6 = mem[0],zero,zero,zero
        vmovd   8(%rdx,%rax), %xmm7             # xmm7 = mem[0],zero,zero,zero
        vmovd   12(%rdx,%rax), %xmm1            # xmm1 = mem[0],zero,zero,zero
        vpcmpeqb        %xmm5, %xmm8, %xmm5
        vpmovsxbq       %xmm5, %ymm5
        vpcmpeqb        %xmm6, %xmm8, %xmm6
        vpmovsxbq       %xmm6, %ymm6
        vpcmpeqb        %xmm7, %xmm8, %xmm7
        vpmovsxbq       %xmm7, %ymm7
        vpcmpeqb        %xmm1, %xmm8, %xmm1
        vpmaskmovq      -96(%r8,%rax,8), %ymm5, %ymm5
        vpmovsxbq       %xmm1, %ymm1
        vpmaskmovq      -64(%r8,%rax,8), %ymm6, %ymm6
        vpaddq  %ymm0, %ymm5, %ymm0
        vpmaskmovq      -32(%r8,%rax,8), %ymm7, %ymm5
        vpaddq  %ymm2, %ymm6, %ymm2
        vpmaskmovq      (%r8,%rax,8), %ymm1, %ymm1
        vpaddq  %ymm3, %ymm5, %ymm3
        vpaddq  %ymm4, %ymm1, %ymm4
        addq    $16, %rax
        cmpq    %rax, %r9
        jne     .LBB0_5
```

Low-level optimizations

Each algorithm and data structure can be tuned using different low-level optimizations:

1. Remove unnecessary copying.

2. Decrease amount of virtual function calls.

3. Tune data layout.

4. Specializations for special cases.

5. CPU dispatch.

6. JIT compilation.

CPU Dispatch

For example your binary distributed only with old instruction set **SSE4.2**.

For **AVX**, **AVX2**, **AVX512** instructions need to use runtime instructions specialization using **CPUID**.

It is important that compilers can vectorize even complex loops. We can rely on this.

Blog post about CPU dispatch:

https://maksimkita.com/blog/jit_in_clickhouse.html

JIT Compilation

JIT compilation can transform dynamic configuration into static configuration.

Not all functions can be easily compiled, not all algorithms can be easily compiled.

Has its own costs (compilation time, memory, maintenance).

But can greatly improve performance in special cases.

Blog post about JIT in ClickHouse:

https://maksimkita.com/blog/jit_in_clickhouse.html

Abstractions and Algorithms

Abstractions and Algorithms

For high performance systems interfaces must be determined by data structures and algorithms.

Top-down approach does not work.

High-performance system must be designed concentrating on doing at least a single task efficiently.

Designed from hardware capabilities.

Abstractions and Algorithms

There is no silver bullet, or best algorithm for any task.

Try to choose the fastest possible algorithm/algorithms for **your specific task**.

Performance must be evaluated on real data.

Most of the algorithms are affected by data distribution.

Tools

Tools

Real time system and per-process resource (CPU, Memory) utilization - **top**, **htop**.

CPU per-process-function - **perf top**.

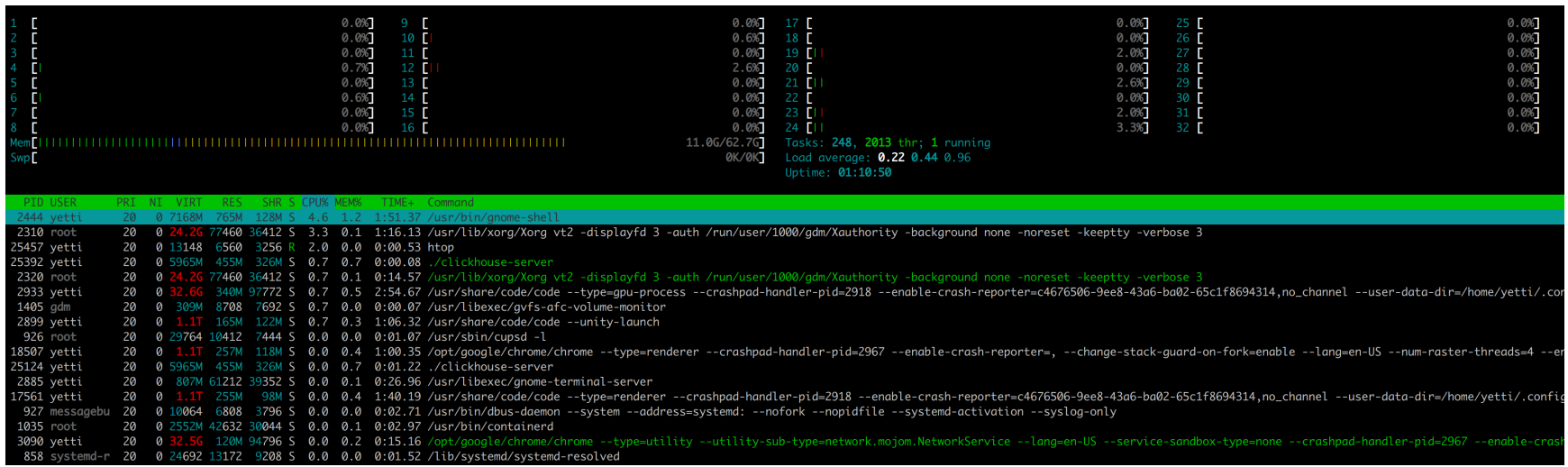
Performance counter statistics - **perf stat**.

CPU Profile - FlameGraphs, **perf record**.

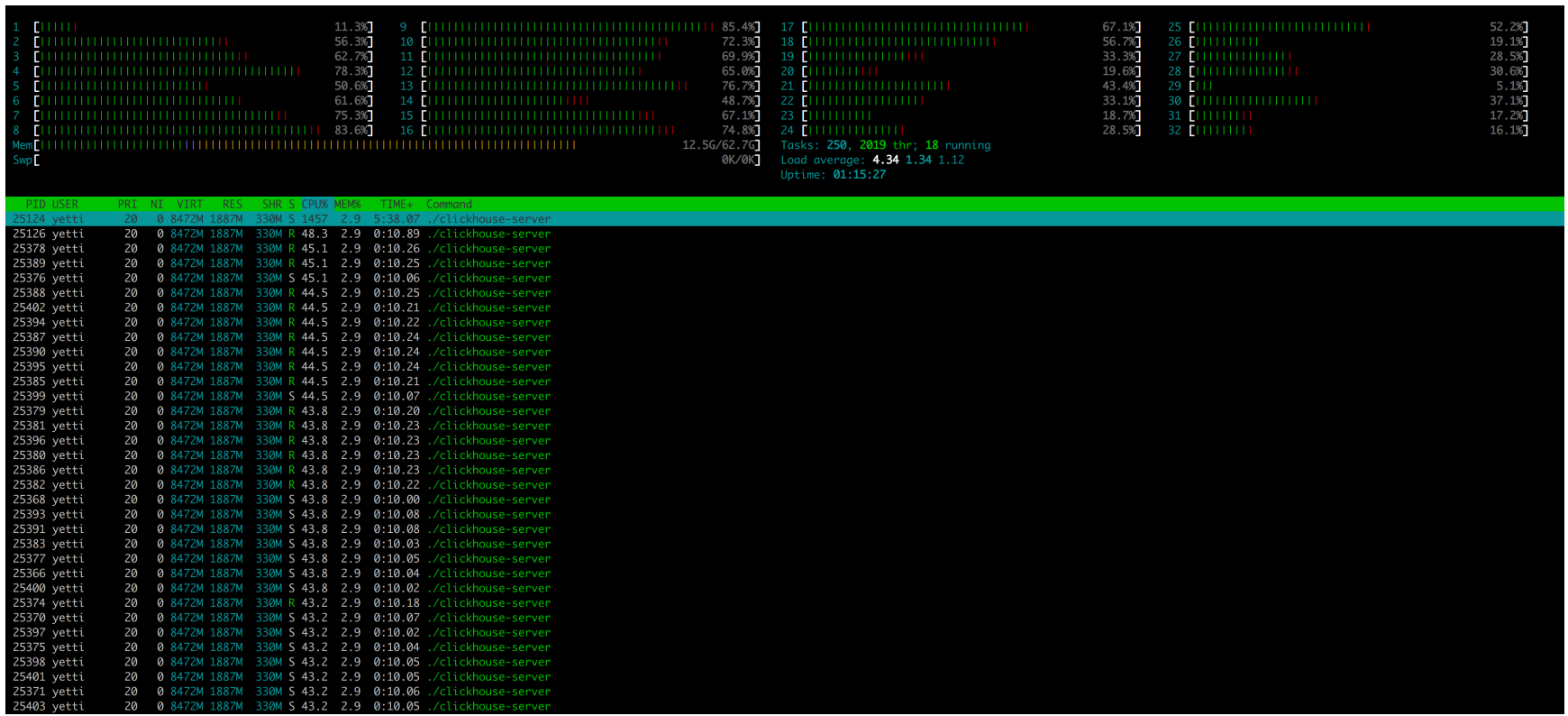
Preparing and analyzing data - **clickhouse-local**.

IO/Network - **iostat**, **dstat**, **sar**.

htop


1 [0.0%] 9 [0.0%] 17 [0.0%] 25 [0.0%]
2 [0.0%] 10 [0.6%] 18 [0.0%] 26 [0.0%]
3 [0.0%] 11 [0.0%] 19 [2.0%] 27 [0.0%]
4 [0.7%] 12 [2.6%] 20 [0.0%] 28 [0.0%]
5 [0.0%] 13 [0.0%] 21 [2.6%] 29 [0.0%]
6 [0.6%] 14 [0.0%] 22 [0.0%] 30 [0.0%]
7 [0.0%] 15 [0.0%] 23 [2.0%] 31 [0.0%]
8 [0.0%] 16 [0.0%] 24 [3.3%] 32 [0.0%]
Mem[11.0G/62.7G]
Swp[0K/0K]
Tasks: 248, 2013 thr; 1 running
Load average: 0.22 0.44 0.96
Uptime: 01:10:50
PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+ Command
2444 yetti 20 0 7168M 765M 128M S 4.6 1.2 1:51.37 /usr/bin/gnome-shell
2310 root 20 0 24.2G 77460 36412 S 3.3 0.1 1:16.13 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3
25457 yetti 20 0 13148 6560 3256 R 2.0 0.0 0:00.53 htop
25392 yetti 20 0 5965M 455M 326M S 0.7 0.7 0:00.08 ./clickhouse-server
2320 root 20 0 24.2G 77460 36412 S 0.7 0.1 0:14.57 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3
2933 yetti 20 0 32.6G 340M 97772 S 0.7 0.5 2:54.67 /usr/share/code/code --type=gpu-process --crashpad-handler-pid=2918 --enable-crash-reporter=c4676506-9ee8-43a6-ba02-65c1f8694314,no_channel --user-data-dir=/home/yetti/.cor
1405 gdm 20 0 309M 8708 7692 S 0.7 0.0 0:00.07 /usr/libexec/gvfs-afc-volume-monitor
2899 yetti 20 0 1.1T 165M 122M S 0.7 0.3 1:06.32 /usr/share/code/code --unity-launch
926 root 20 0 29764 10412 7444 S 0.0 0.0 0:01.07 /usr/sbin/cupsd -l
18507 yetti 20 0 1.1T 257M 118M S 0.0 0.4 1:00.35 /opt/google/chrome/chrome --type=renderer --crashpad-handler-pid=2967 --enable-crash-reporter=, --change-stack-guard-on-fork=enable --lang=en-US --num-raster-threads=4 --er
25124 yetti 20 0 5965M 455M 326M S 0.0 0.7 0:01.22 ./clickhouse-server
2885 yetti 20 0 807M 61212 39352 S 0.0 0.1 0:26.96 /usr/libexec/gnome-terminal-server
17561 yetti 20 0 1.1T 255M 98M S 0.0 0.4 1:40.19 /usr/share/code/code --type=renderer --crashpad-handler-pid=2918 --enable-crash-reporter=c4676506-9ee8-43a6-ba02-65c1f8694314,no_channel --user-data-dir=/home/yetti/.config
927 messagebu 20 0 10064 6808 3796 S 0.0 0.0 0:02.71 /usr/bin/dbus-daemon --system --address=systemd: --nofork --nopidfile --systemd-activation --syslog-only
1035 root 20 0 2552M 42632 30044 S 0.0 0.1 0:02.97 /usr/bin/containerd
3090 yetti 20 0 32.5G 120M 94796 S 0.0 0.2 0:15.16 /opt/google/chrome/chrome --type=utility --utility-sub-type=network.mojom.NetworkService --lang=en-US --service-sandbox-type=none --crashpad-handler-pid=2967 --enable-crash
858 systemd-r 20 0 24692 13172 9208 S 0.0 0.0 0:01.52 /lib/systemd/systemd-resolved

htop during benchmark



htop during benchmark

High CPU utilization does not mean your program is fast. There can be several reasons:

1. Program is Memory/IO bound.

2. Inneficient algorithms, data structures.

3. Excessive copying.

perf top

```
7.94%  perf                          [.] __symbols__insert
4.55%  perf                          [.] d_print_comp_inner
3.79%  perf                          [.] rb_next
2.60%  [nvidia]                      [k] _nv033096rm
2.22%  perf                          [.] rust_demangle_callback
1.87%  perf                          [.] d_count_templates_scopes
1.75%  [unknown]                     [.] 0x00007f60959607e4
1.48%  perf                          [.] d_print_comp
1.15%  [unknown]                     [.] 0x00007f6095960834
0.91%  libc-2.31.so                  [.] _int_free
0.89%  [unknown]                     [.] 0x00007f609596083c
0.86%  libc-2.31.so                  [.] unlink_chunk.isra.0
0.71%  [kernel]                      [k] psi_group_change
0.66%  libc-2.31.so                  [.] __strlen_avx2
0.60%  perf                          [.] d_name
0.60%  perf                          [.] d_make_comp
0.60%  libc-2.31.so                  [.] _int_malloc
0.59%  libc-2.31.so                  [.] cfree@GLIBC_2.2.5
0.57%  [kernel]                      [k] update_sd_lb_stats.constprop.0
0.54%  perf                          [.] cplus_demangle_type
0.54%  [unknown]                     [.] 0x00007f6095960846
0.52%  libgobject-2.0.so.0.6400.6    [.] g_type_check_instance_is_a
0.52%  [unknown]                     [.] 0x00007f60959606f2
0.50%  perf                          [.] rb_insert_color
```

perf top during benchmark

```
 21.94%  clickhouse                      [.] DB::Aggregator::executeImplBatch<false, false, true, DB::AggregationMethodOneNumber<unsigned long, Two
 17.98%  libc-2.31.so                    [.] __memset_avx2_unaligned_erms
 16.37%  clickhouse                      [.] DB::IAggregateFunctionHelper<DB::AggregateFunctionCount>::insertResultIntoBatch
 13.42%  clickhouse                      [.] DB::Aggregator::mergeDataImpl<DB::AggregationMethodOneNumber<unsigned long, TwoLevelHashMapTable<unsig
  4.95%  clickhouse                      [.] HashTable<unsigned long, HashMapCell<unsigned long, char*, HashCRC32<unsigned long>, HashTableNoState,
  3.79%  clickhouse                      [.] memcpy
  3.10%  clickhouse                      [.] HashMapTable<unsigned long, HashMapCell<unsigned long, char*, HashCRC32<unsigned long>, HashTableNoSta
  1.67%  [kernel]                        [k] copy_user_generic_string
  1.15%  [kernel]                        [k] clear_page_rep
  0.93%  clickhouse                      [.] DB::IAggregateFunctionHelper<DB::AggregateFunctionCount>::addBatch
  0.77%  clickhouse                      [.] DB::PODArrayBase<8ul, 4096ul, Allocator<false, false>, 63ul, 64ul>::push_back_raw<>
  0.65%  clickhouse                      [.] LZ4_compress_fast_extState
  0.50%  clickhouse                      [.] CityHash_v1_0_2::CityHash128WithSeed
  0.35%  clickhouse                      [.] DB::Aggregator::createAggregateStates<false>
  0.28%  clickhouse                      [.] DB::IAggregateFunctionDataHelper<DB::AggregateFunctionCountData, DB::AggregateFunctionCount>::create
  0.23%  clickhouse                      [.] LZ4::(anonymous namespace)::decompressImpl<16ul, false>
  0.23%  perf                            [.] queue_event
  0.21%  clickhouse                      [.] LZ4::(anonymous namespace)::decompressImpl<32ul, false>
  0.20%  clickhouse                      [.] std::__1::align
  0.19%  clickhouse                      [.] LZ4::(anonymous namespace)::decompressImpl<8ul, true>
  0.17%  clickhouse                      [.] te_event_trigger
  0.15%  clickhouse                      [.] LZ4::(anonymous namespace)::decompressImpl<16ul, true>
  0.14%  clickhouse                      [.] DB::IAggregateFunctionDataHelper<DB::AggregateFunctionCountData, DB::AggregateFunctionCount>::destroy
  0.14%  perf                            [.] dso__find_symbol
```

perf top during benchmark

perf top gives high level view of what program is doing during benchmark, and which functions cost most of the CPU.

Useful to check underlying assembly and try to figure out where the problem is.

But need to be aware that CPU can just wait memory.

perf top during benchmark

In aggregation benchmark if we check the hottest function assembly we will see:

```
                _ZN11HashMapCellImPc9HashCRC32ImE16HashTableNoState10PairNoInitImS0_EE6isZeroERKmRKS3_():
                ./build_release/./src/Common/HashTable/HashMap.h:91
  0.01            mov        (%r12,%rax,1),%rax
                _ZN10ZeroTraits5checkImEEbT_():
                ./build_release/./src/Common/HashTable/HashTable.h:75
 81.43            test       %rax,%rax
```

perf top during benchmark

Underlying code:

```
namespace ZeroTraits
{
template <typename T>
bool check(const T x) { return x == T{}; } /// test %rax, %rax
}

static bool isZero(const Key & key, const State & /*state*/)
{
    return ZeroTraits::check(key);
}
```

perf top during benchmark

Program is memory bound.

perf stat

Useful tool to check something in isolation.

Assume you want to improve performance of some critical data structure like hash table or algorithm like sorting.

Best way to do this, is to isolate this into separate small program and benchmark it.

perf stat

Lets take a look of integer hash table benchmark.

```
perf stat integer_hash_tables_benchmark ch_hash_map UInt64 WatchID.bin 8873898
CH HashMap:
Elapsed: 0.604 (14688324.619 elem/sec.), map size: 8871741

 Performance counter stats for 'integer_hash_tables_benchmark ch_hash_map UInt64 WatchID.bin 8873898':

            683.90 msec task-clock                #    0.968 CPUs utilized
               124      context-switches          #  181.313 /sec
                 0      cpu-migrations            #    0.000 /sec
           252,549      page-faults               #  369.277 K/sec
     2,948,142,592      cycles                    #    4.311 GHz                      (83.10%)
       286,182,948      stalled-cycles-frontend   #    9.71% frontend cycles idle     (83.07%)
     1,276,430,369      stalled-cycles-backend    #   43.30% backend cycles idle      (83.35%)
     2,025,692,416      instructions              #    0.69  insn per cycle
                                                  #    0.63  stalled cycles per insn  (83.64%)
       356,845,237      branches                  #  521.779 M/sec                    (83.65%)
        21,268,813      branch-misses             #    5.96% of all branches          (83.20%)

       0.706636444 seconds time elapsed

       0.479112000 seconds user
       0.203622000 seconds sys
```

perf stat

Lets take a look of integer hash table benchmark.

```
perf stat integer_hash_tables_benchmark std_unordered_map UInt64 WatchID.bin 8873898
std::unordered_map:
Elapsed: 2.473 (3588276.141 elem/sec.), map size: 8871741

 Performance counter stats for 'integer_hash_tables_benchmark std_unordered_map UInt64 WatchID.bin
 8873898':

          3,262.13 msec task-clock                #    0.999 CPUs utilized
               317      context-switches          #   97.176 /sec
                 1      cpu-migrations            #    0.307 /sec
           142,567      page-faults               #   43.704 K/sec
    14,343,682,708      cycles                    #    4.397 GHz                      (83.33%)
       258,005,531      stalled-cycles-frontend   #    1.80% frontend cycles idle     (83.33%)
    12,555,340,086      stalled-cycles-backend    #   87.53% backend cycles idle      (83.34%)
     4,104,335,191      instructions              #    0.29  insn per cycle
                                                  #    3.06  stalled cycles per insn  (83.33%)
       735,241,554      branches                  #  225.387 M/sec                    (83.33%)
        10,749,318      branch-misses             #    1.46% of all branches          (83.33%)

       3.264492212 seconds time elapsed

       3.060895000 seconds user
       0.199797000 seconds sys
```

perf stat

We see that standard hash map is much slower. Lets check cache misses and cache references.

```
perf stat -e cache-misses,cache-references integer_hash_tables_benchmark std_unordered_map
UInt64 WatchID.bin 8873898
std::unordered_map:
Elapsed: 2.484 (3572510.427 elem/sec.), map size: 8871741

 Performance counter stats for 'integer_hash_tables_benchmark std_unordered_map UInt64 WatchID.bin
 8873898':

       124,089,032      cache-misses              #   35.893 % of all cache refs
       345,719,343      cache-references

       3.296088157 seconds time elapsed

       3.117693000 seconds user
       0.175644000 seconds sys
```

perf stat

```
perf stat -e cache-misses,cache-references integer_hash_tables_benchmark ch_hash_map
UInt64 WatchID.bin 8873898
CH HashMap:
Elapsed: 0.580 (15306059.985 elem/sec.), map size: 8871741

 Performance counter stats for 'integer_hash_tables_benchmark ch_hash_map UInt64 WatchID.bin
 8873898':

       32,774,660      cache-misses              #   28.599 % of all cache refs
      114,602,476      cache-references

      0.654290201 seconds time elapsed

      0.452424000 seconds user
      0.200187000 seconds sys
```

perf stat

ClickHouse Hash Table performs 3.8 times less cache misses.

Benchmarking tips

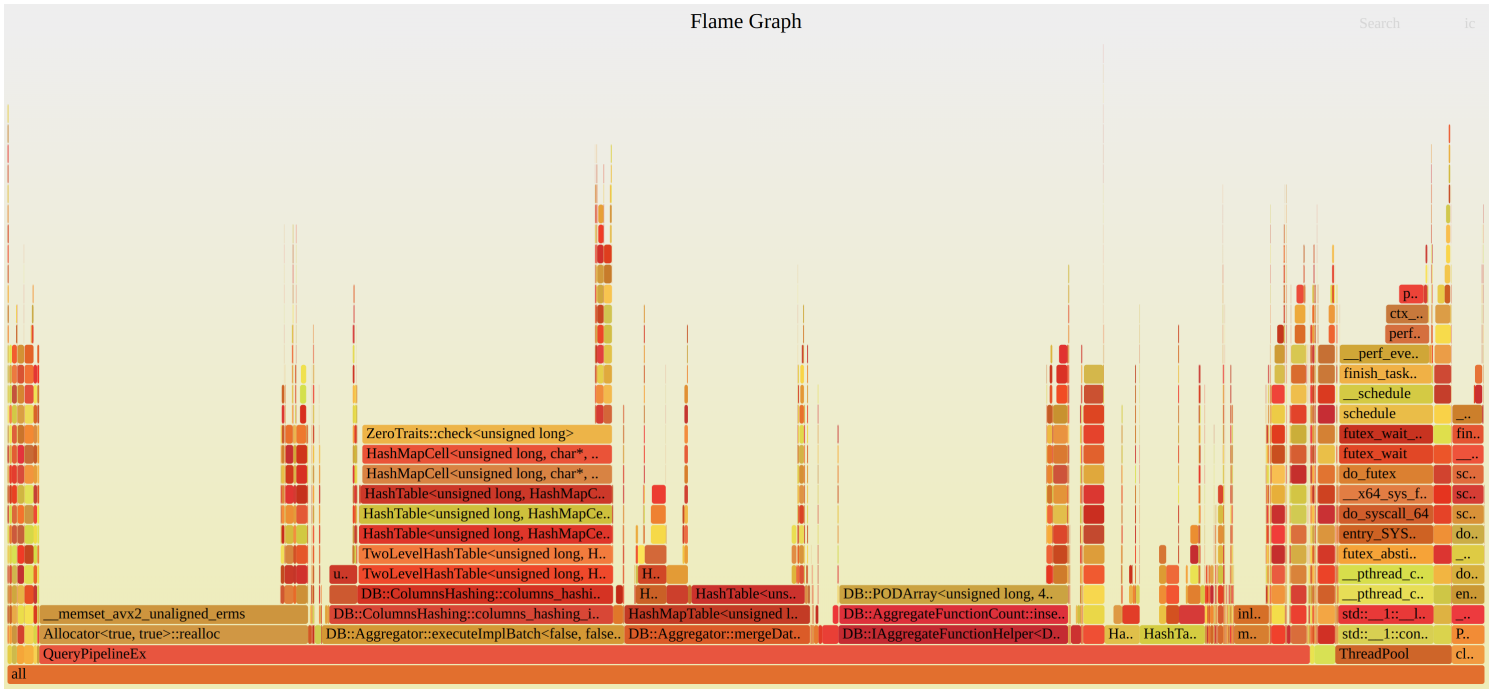
There are a lot of additional things that you can tune to reduce perf variations. Example:

1. Bind program to specific CPUs.

2. Use tmpfs.

https://llvm.org/docs/Benchmarking.html

Flame Graph



Can be build for CPU, Memory.

Flame Graph

Commands:

```
git clone https://github.com/brendangregg/FlameGraph
cd FlameGraph
perf record -F 99 -a -g or perf record -F 99 -a -g -p process_pid
perf script | ./stackcollapse-perf.pl > out.perf-folded
./flamegraph.pl out.perf-folded > perf.svg
```

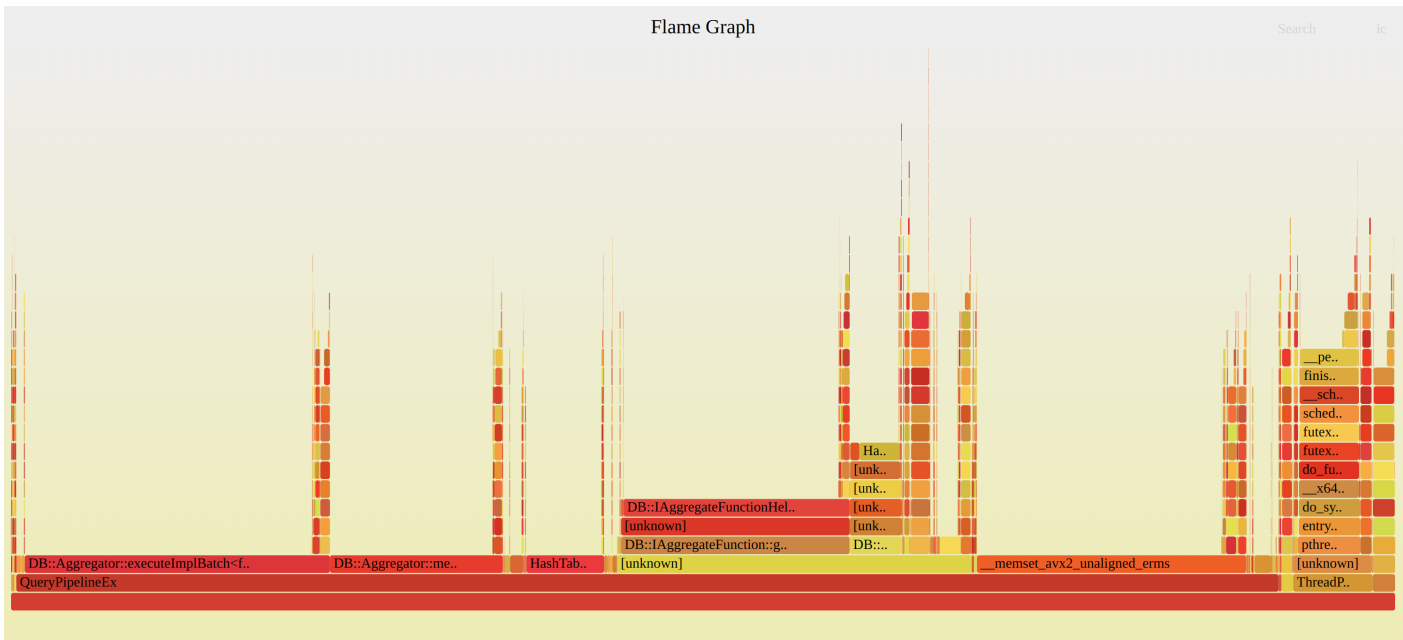
https://www.brendangregg.com/flamegraphs.html

Flame Graph

By default perf uses frame-pointer stack unwinding, so to see all traces your program must be compiled with **-fno-omit-frame-pointer**.

Flame Graph

Example of broken Flame Graph:

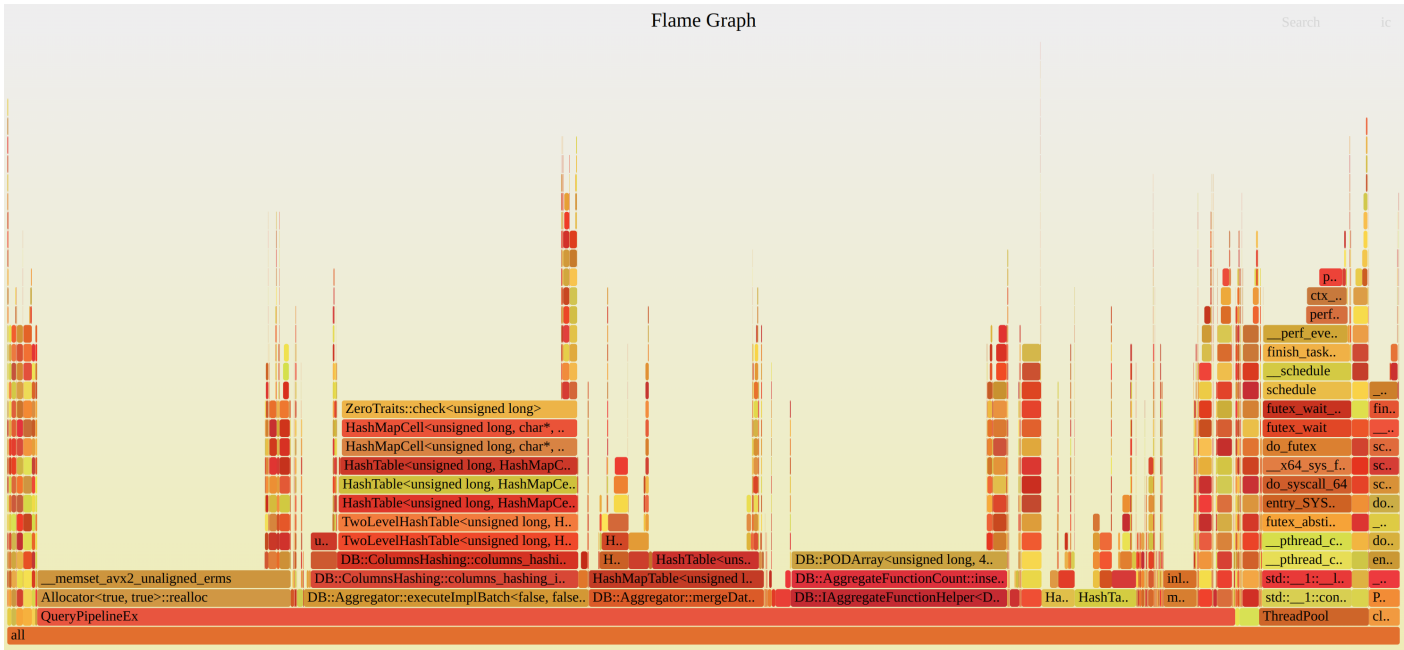


Flame Graph

Use DWARF stack unwinding:

```
perf record -F 99 -a -g -p 32238 -g --call-graph=dwarf
```

Flame Graph


Flame Graph
Search
ic
p..
ctx_..
perf..
__perf_eve..
finish_task..
__schedule
schedule
_..
ZeroTraits::check<unsigned long>
futex_wait_..
fin..
HashMapCell<unsigned long, char*, ..
futex_wait
__..
HashMapCell<unsigned long, char*, ..
do_futex
sc..
HashTable<unsigned long, HashMapC..
__x64_sys_f..
sc..
HashTable<unsigned long, HashMapCe..
do_syscall_64
sc..
HashTable<unsigned long, HashMapCe..
entry_SYS..
do..
TwoLevelHashTable<unsigned long, H..
futex_absti..
_..
u..
TwoLevelHashTable<unsigned long, H..
H..
__pthread_c..
do..
DB::ColumnsHashing::columns_hashi..
H..
HashTable<uns..
DB::PODArray<unsigned long, 4..
__pthread_c..
en..
__memset_avx2_unaligned_erms
DB::ColumnsHashing::columns_hashing_i..
HashMapTable<unsigned l..
DB::AggregateFunctionCount::inse..
inl..
std::__1::__l..
_..
Allocator<true, true>::realloc
DB::Aggregator::executeImplBatch<false, false..
DB::Aggregator::mergeDat..
DB::IAggregateFunctionHelper<D..
Ha..
HashTa..
m..
std::__1::con..
P..
QueryPipelineEx
ThreadPool
cl..
all

clickhouse-local

I use **clickhouse-local** almost every day:

1. Preprocess data for benchmarks.

2. Analyze data for benchmarks.

3. Analyze benchmark results.

4. Analyze output of different tools.

Analyze output of different tools

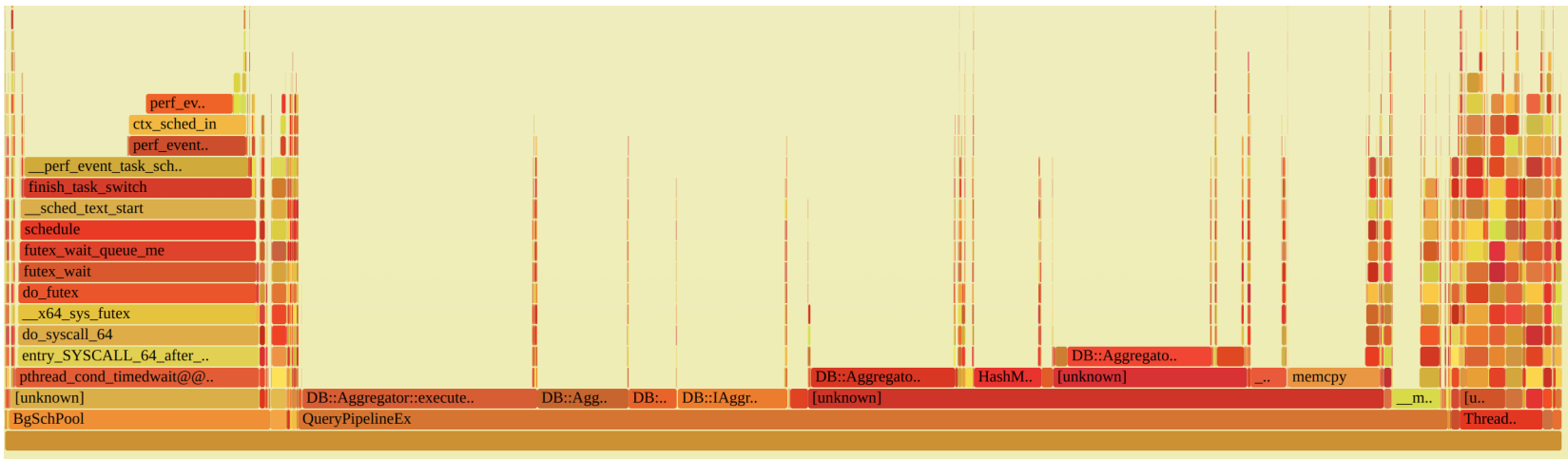
Amount of non vectorized loops in ClickHouse AggregateFunctions, after compile with **-Rpass=loop-vectorize**, **-Rpass-missed=loop-vectorize** and **-Rpass-analysis=loop-vectorize**.

```
SELECT count() FROM (SELECT splitByChar(' ', line)[1] AS file_with_line
    FROM file("out.txt", LineAsString)
    WHERE line LIKE '%loop not vectorized%'
    AND line LIKE '%ClickHouseClang/src/AggregateFunctions/%'
    GROUP BY file_with_line
);

┌─count()─┐
│     403 │
└─────────┘
```

Examples

High CPU usage issue



High CPU usage issue

Background pool produce a lot of calls to wait with timeout for **Poco::NotificationQueue**.

Such calls are also visible on flame graphs and for some benchmarks can take 15-20% of CPU time.

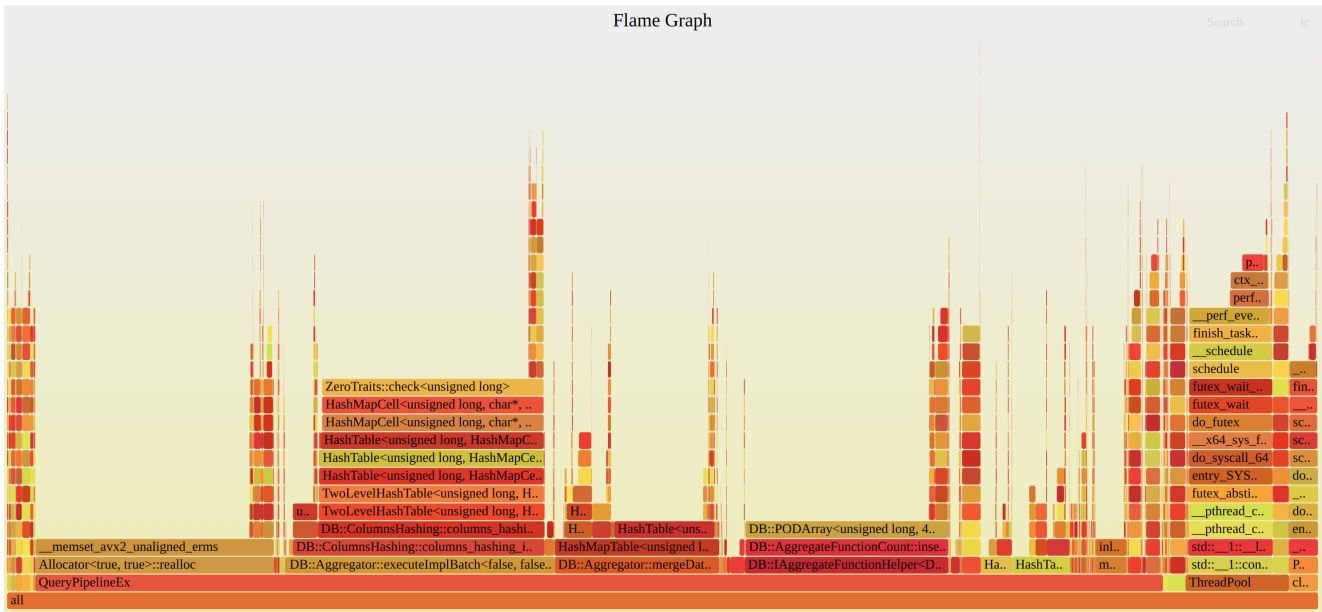
High CPU usage issue

This timeout was added long time ago to avoid rare deadlock with **Poco::NotificationQueue**.

Solution was to replace **Poco::NotificationQueue** to conditional variable and queue.

https://github.com/ClickHouse/ClickHouse/pull/38028

High CPU usage issue



Standard containers

```
INSERT INTO FUNCTION file('test_file_{_partition_id}', 'TSV',
    'partition_id UInt64, value1 UInt64, value2 UInt64, value3 UInt64,
        value4 UInt64, value5 UInt64') PARTITION BY partition_id
SELECT
    number % 5 AS partition_id,
    number,
    number,
    number,
    number,
    number
FROM numbers(100000000);
```

Standard containers

Split chunk into multiple chunks.

```
void PartitionedSink::consume(Chunk chunk)
{
    ...

    std::unordered_map<String, size_t> partition_id_to_chunk_index;
    IColumn::Selector chunk_row_index_to_partition_index;

    for (size_t row = 0; row < chunk.getNumRows(); ++row)
    {
        auto value = column->getDataAt(row);
        auto [it, inserted] = partition_id_to_chunk_index.emplace(value,
            partition_id_to_chunk_index.size());
        chunk_row_index_to_partition_index.push_back(it->second);
    }

    ...
}
```

Standard containers

Use **HashMapWithSavedHash<StringRef, size_t>** instead of **std::unordered_map<String, size_t>**.

Store hash table and arena as class member, to reuse them between **consume** calls.

Standard containers

```
void PartitionedSink::consume(Chunk chunk)
{
    ...

    size_t chunk_rows = chunk.getNumRows();
    chunk_row_index_to_partition_index.resize(chunk_rows);
    partition_id_to_chunk_index.clear();

    for (size_t row = 0; row < chunk_rows; ++row)
    {
        auto partition_key = partition_by_result_column->getDataAt(row);
        auto [it, inserted] = partition_id_to_chunk_index.insert(
            makePairNoInit(partition_key, partition_id_to_chunk_index.size()));
        if (inserted)
            it->value.first = copyStringInArena(partition_keys_arena, partition_key);

        chunk_row_index_to_partition_index[row] = it->getMapped();
    }

    ...
}
```

Standard containers

```
INSERT INTO FUNCTION file('test_file_{_partition_id}', 'TSV',
            'partition_id UInt64, value1 UInt64, value2 UInt64, value3 UInt64,
                value4 UInt64, value5 UInt64') PARTITION BY partition_id
        SELECT
            number % 5 AS partition_id,
            number,
            number,
            number,
            number,
            number
        FROM numbers(100000000);
```

Was: **9.714** sec (10.36 million rows/s., 82.90 MB/s.)

Now: **2.868** sec (35.10 million rows/s., 280.76 MB/s.)

Avoid virtual function calls

Assume that we have interface that provide some generic methods for working with object. For example in ClickHouse there is **IColumn** interface that provide a lot of methods for working with columns (filter, getPermutation, compareAt).

Sometimes we need to write complex logic that is implemented on top of such interface.

The problem is that if this logic will be implemented in terms of small virtual methods like **compareAt** there will be a lot of virtual function calls.

Avoid virtual function calls

In most of the cases we use CRTP (The Curiously Reccuring Template Pattern).

Also we can use just standalone template function.

Aggregate function CRTP

```
class IAggregateFunction
{
    ...

    virtual ~IAggregateFunction() = default;

    virtual void add(
        AggregateDataPtr place,
        const IColumn ** columns,
        size_t row_num,
        Arena * arena) const = 0;

    virtual void addBatch(
        size_t row_begin,
        size_t row_end,
        AggregateDataPtr * places,
        size_t place_offset,
        const IColumn ** columns,
        Arena * arena,
        ssize_t if_argument_pos = -1) const = 0;
    ...
}
```

Aggregate function CRTP

```
template <typename Derived>
class IAggregateFunctionHelper : public IAggregateFunction
{
    void addBatch(size_t row_begin, size_t row_end, AggregateDataPtr * places,
        size_t place_offset, const IColumn ** columns, Arena * arena,
        ssize_t if_argument_pos = -1) const override
    {
        if (if_argument_pos >= 0)
        {
            auto * column = columns[if_argument_pos];
            const auto & flags = assert_cast<const ColumnUInt8 &>(*column).getData();
            for (size_t i = row_begin; i < row_end; ++i)
            {
                if (flags[i] && places[i])
                    static_cast<const Derived *>(this)->add(places[i] + place_offset,
                        columns, i, arena);
            }
        }
        {
            for (size_t i = row_begin; i < row_end; ++i)
                if (places[i])
                    static_cast<const Derived *>(this)->add(places[i] + place_offset,
                        columns, i, arena);
        }
    }

}
```

Aggregate function CRTP

```
template <typename T, typename Derived>
class IAggregateFunctionDataHelper : public IAggregateFunctionHelper<Derived>
{
    ...
}

class AggregateFunctionCount final : public IAggregateFunctionDataHelper<AggregateFunctionCountData,
    AggregateFunctionCount>
{
    void add(AggregateDataPtr __restrict place, const IColumn **, size_t, Arena *) const override
    {
        ++data(place).count;
    }
}
```

IColumn virtual function calls

```
class IColumn
{
    ...

    virtual int compareAt(size_t n, size_t m, const IColumn & rhs,
        int nan_direction_hint) const = 0;

    virtual void compareColumn(const IColumn & rhs, size_t rhs_row_num,
                               PaddedPODArray<UInt64> * row_indexes,
                               PaddedPODArray<Int8> & compare_results,
                               int direction, int nan_direction_hint) const = 0;

    ...

}
```

IColumn virtual function calls

```
class IColumn
{
    ...

protected:
    template <typename Derived, bool reversed, bool use_indexes>
    void compareImpl(const Derived & rhs, size_t rhs_row_num,
                     PaddedPODArray<UInt64> * row_indexes,
                     PaddedPODArray<Int8> & compare_results,
                     int nan_direction_hint) const;

    template <typename Derived>
    void doCompareColumn(const Derived & rhs, size_t rhs_row_num,
                         PaddedPODArray<UInt64> * row_indexes,
                         PaddedPODArray<Int8> & compare_results,
                         int direction, int nan_direction_hint) const;

    ...
}
```

IColumn virtual function calls

```
template <is_decimal T>
void ColumnDecimal<T>::compareColumn(const IColumn & rhs, size_t rhs_row_num,
                                     PaddedPODArray<UInt64> * row_indexes,
                                     PaddedPODArray<Int8> & compare_results,
                                     int direction, int nan_direction_hint) const
{
    return this->template doCompareColumn<ColumnDecimal<T>>(
        static_cast<const Self &>(rhs),
        rhs_row_num,
        row_indexes,
        compare_results,
        direction,
        nan_direction_hint);
}
```

IColumn virtual function calls

Compile time dispatch:

```
template <typename Derived>
void IColumn::doCompareColumn(const Derived & rhs, size_t rhs_row_num,
                              PaddedPODArray<UInt64> * row_indexes,
                              PaddedPODArray<Int8> & compare_results,
                              int direction, int nan_direction_hint) const
{
    if (direction < 0)
    {
        if (row_indexes)
            compareImpl<Derived, true, true>(rhs, rhs_row_num, row_indexes,
                    compare_results, nan_direction_hint);
        else
            compareImpl<Derived, true, false>(rhs, rhs_row_num, row_indexes,
            compare_results, nan_direction_hint);
        }
    else
    {
        if (row_indexes)
            compareImpl<Derived, false, true>(rhs, rhs_row_num, row_indexes,
                    compare_results, nan_direction_hint);
        else
            compareImpl<Derived, false, false>(rhs, rhs_row_num, row_indexes,
                compare_results, nan_direction_hint);
    }
}
```

IColumn virtual function calls

```
template <typename Derived, bool reversed, bool use_indexes>
void IColumn::compareImpl(const Derived & rhs, size_t rhs_row_num,
                          PaddedPODArray<UInt64> * row_indexes [[maybe_unused]],
                          PaddedPODArray<Int8> & compare_results,
                          int nan_direction_hint) const
{
    for (size_t i = 0; i < num_indexes; ++i)
    {
        UInt64 row = i;

        if constexpr (use_indexes)
            row = indexes[i];

        int res = compareAt(row, rhs_row_num, rhs, nan_direction_hint);
        assert(res == 1 || res == -1 || res == 0);
        compare_results[row] = static_cast<Int8<(res);

        if constexpr (reversed)
            compare_results[row] = -compare_results[row];

        if constexpr (use_indexes)
        {
            if (compare_results[row] == 0)
            {
                *next_index = row;
                ++next_index;
            }
        }
    }
}
```

IColumn virtual function calls

```
template <typename Derived, bool reversed, bool use_indexes>
void IColumn::compareImpl(const Derived & rhs, size_t rhs_row_num,
                          PaddedPODArray<UInt64> * row_indexes [[maybe_unused]],
                          PaddedPODArray<Int8> & compare_results,
                          int nan_direction_hint) const
{
    for (size_t i = 0; i < num_indexes; ++i)
    {
        UInt64 row = i;

        if constexpr (use_indexes)
            row = indexes[i];

        int res = static_cast<const Derived *>(this)->compareAt(row, rhs_row_num, rhs,
            nan_direction_hint);
        assert(res == 1 || res == -1 || res == 0);
        compare_results[row] = static_cast<Int8<(res);

        if constexpr (reversed)
            compare_results[row] = -compare_results[row];

        if constexpr (use_indexes)
        {
            if (compare_results[row] == 0)
            {
                *next_index = row;
                ++next_index;
            }
        }
    }
}
```

IColumn virtual function calls

Result performance improvements:

Changes in Performance ?

Old, s	New, s	Ratio of speedup (-) or slowdown (+)	Relative difference (new − old) / old	p < 0.01 threshold	Test	#	Query
0.089	0.067	-1.335x	-0.252	0.249	order_with_limit	9	SELECT intHash64(number) AS n FROM numbers_mt(200000000) ORDER BY n LIMIT 1500 FORMAT Null
0.084	0.069	-1.208x	-0.173	0.170	order_with_limit	3	SELECT number AS n FROM numbers_mt(200000000) ORDER BY n DESC LIMIT 3000 FORMAT Null
0.130	0.109	-1.186x	-0.159	0.153	order_with_limit	18	SELECT intHash64(number) AS n FROM numbers_mt(100000000) ORDER BY n, n + 1, n + 2 LIMIT 5000 FORMAT Null
0.197	0.166	-1.187x	-0.158	0.157	order_with_limit	17	SELECT intHash64(number) AS n FROM numbers_mt(200000000) ORDER BY n, n + 1, n + 2 LIMIT 3000 FORMAT Null
0.086	0.076	-1.126x	-0.113	0.099	order_with_limit	4	SELECT number AS n FROM numbers_mt(200000000) ORDER BY n DESC LIMIT 5000 FORMAT Null

Sorting

How Sorting in ClickHouse works. Let first take a look at pipeline:

```
EXPLAIN PIPELINE SELECT WatchID FROM hits_100m_single
ORDER BY WatchID, CounterID;

┌─explain─────────────────────────────────────┐
│ (Expression)                                │
│ ExpressionTransform                         │
│   (Sorting)                                 │
│   MergingSortedTransform 16 → 1             │
│     MergeSortingTransform × 16              │
│       LimitsCheckingTransform × 16          │
│         PartialSortingTransform × 16        │
│           (Expression)                      │
│           ExpressionTransform × 16          │
│             (SettingQuotaAndLimits)         │
│               (ReadFromMergeTree)           │
│               MergeTreeThread × 16 0 → 1    │
└─────────────────────────────────────────────┘
```

Sorting

In physical query plan we have multiple transform that work together to perform sorting:

PartialSortingTransform — sort single block, apply special optimization if LIMIT is specified.

MergeSortingTransform — sort multiple blocks using k-way-merge algorithm, output of this transform is a stream of sorted blocks.

MergingSortedTransform — sort multiple streams of sorted blocks using k-way-merge algorithm.

K-way merge algorithm

Sorting

Sort of single block in PartialSortingTransform, can be performed in batch, without indirections. There is **getPermutation**, **updatePermutaion** methods in **IColumn** that returns or update permutation.

This permutation can later be applied efficiently to any column using **permute** method.

Sorting

The problem is with MergeSortingTransform and MergingSortedTransform. They must perform k-way-merge algorithm, and this algorithm operates on single rows instead of columns.

In the worst case, we will apply ORDER BY WatchID, CounterID comparator to each row N * log(N) * 2 times during our MergeSortingTransform, MergingSortedTransform.

Sorting

```
SELECT WatchID FROM hits_100m_obfuscated
ORDER BY WatchID FORMAT Null;
```

```
 24.77%  clickhouse                [.] DB::MergingSortedAlgorithm::mergeImpl<DB::SortingQueueImpl<DB::SortCursor, (DB::SortingQueueStrategy)0> >
 16.45%  clickhouse                [.] DB::ColumnVector<unsigned long>::compareAt
 15.96%  clickhouse                [.] DB::MergeSorter::mergeBatchImpl<DB::SortingQueueImpl<DB::SortCursor, (DB::SortingQueueStrategy)1> >
 15.35%  clickhouse                [.] RadixSort<DB::(anonymous namespace)::RadixSortTraits<unsigned long> >::radixSortLSDInternal<true>
  9.54%  clickhouse                [.] DB::SortingQueueImpl<DB::SortCursor, (DB::SortingQueueStrategy)1>::updateBatchSize
  3.15%  clickhouse                [.] DB::ColumnVector<unsigned long>::insertFrom
  1.29%  clickhouse                [.] DB::ColumnVector<unsigned long>::permute
  0.84%  clickhouse                [.] DB::ColumnVector<unsigned long>::getPermutation
  0.75%  [kernel]                  [k] copy_user_generic_string
  0.32%  clickhouse                [.] CityHash_v1_0_2::CityHash128WithSeed
  0.29%  clickhouse                [.] memcpy
  0.28%  clickhouse                [.] DB::ColumnVector<unsigned long>::insertRangeFrom
  0.22%  [nvidia]                  [k] _nv033096rm
  0.16%  perf                      [.] __symbols__insert
  0.16%  [unknown]                 [.] 0x00007f60959607e4
  0.15%  [kernel]                  [k] clear_page_rep
  0.12%  [unknown]                 [.] 0x00007f6095960834
  0.10%  [kernel]                  [k] psi_group_change
  0.10%  [kernel]                  [k] update_sd_lb_stats.constprop.0
  0.09%  perf                      [.] rb_next
```

Sorting

```
struct SortCursor : SortCursorHelper<SortCursor>
{
   using SortCursorHelper<SortCursor>::SortCursorHelper;

   /// The specified row of this cursor is greater than the specified
   /// row of another cursor.
   bool greaterAt(const SortCursor & rhs, size_t lhs_pos, size_t rhs_pos) const
   {
       for (size_t i = 0; i < impl->sort_columns_size; ++i)
       {
           const auto & sort_description = impl->desc[i];
           int direction = sort_description.direction;
           int nulls_direction = sort_description.nulls_direction;
           int res = direction * impl->sort_columns[i]->compareAt(lhs_pos, rhs_pos,
               *(rhs.impl->sort_columns[i]), nulls_direction);

           if (res > 0)
               return true;
           if (res < 0)
               return false;
       }

       return impl->order > rhs.impl->order;
   }
};
```

Sorting

Worst thing for column DBMS is to process elements in rows.

The biggest problem here is that for each column specified in the ORDER BY comparator, we call **compareAt** method N * log(N) * 2 times.

Sorting Single Column Specialization

The most common case of sorting in databases is sorting by single column.

```
SELECT WatchID FROM hits_100m_obfuscated
ORDER BY WatchID FORMAT Null;
```

Lets write specialization.

Sorting Single Column Specialization

```
template <typename ColumnType>
struct SpecializedSingleColumnSortCursor : SortCursorHelper<
    SpecializedSingleColumnSortCursor<ColumnType>>
{
    bool ALWAYS_INLINE greaterAt(
        const SortCursorHelper<SpecializedSingleColumnSortCursor> & rhs,
        size_t lhs_pos, size_t rhs_pos) const
    {
        auto & lhs_columns = this->impl->sort_columns;
        auto & rhs_columns = rhs.impl->sort_columns;
        const auto & lhs_column = assert_cast<const ColumnType &>(*lhs_columns[0]);
        const auto & rhs_column = assert_cast<const ColumnType &>(*rhs_columns[0]);

        const auto & desc = this->impl->desc[0];

        int res = desc.direction * lhs_column.compareAt(lhs_pos, rhs_pos,
            rhs_column, desc.nulls_direction);
        if (res > 0)
            return true;
        if (res < 0)
            return false;

        return this->impl->order > rhs.impl->order;
    }
};
```

Sorting Single Column Specialization

```
SELECT WatchID FROM hits_100m_obfuscated
ORDER BY WatchID FORMAT Null;
```

Was: **5.401** sec (18.51 million rows/s., 148.11 MB/s)

Now: **4.363** sec (22.92 million rows/s., 183.35 MB/s.)

Sorting Multiple Columns JIT

For multiple columns in comparator we applied JIT compilation.

We fused multiple **compareAt** methods in a single function to avoid unnecessary indirections and decrease the number of virtual function calls if multiple columns are specified in the comparator.

Sorting Multiple Columns JIT

```
SELECT WatchID FROM hits_100m_single
ORDER BY WatchID, CounterID
SETTINGS compile_sort_description=0;
```

— 0 rows in set. Elapsed: 6.408 sec. Processed 100.00 million rows, 1.20 GB (15.60 million rows/s., 187.26 MB/s.)

```
SELECT WatchID FROM hits_100m_single
ORDER BY WatchID, CounterID
SETTINGS compile_sort_description=1;
```

— 0 rows in set. Elapsed: 5.300 sec. Processed 100.00 million rows, 1.20 GB (18.87 million rows/s., 226.40 MB/s.)
— **+20% performance improvement!**

Sorting K-Way Merge

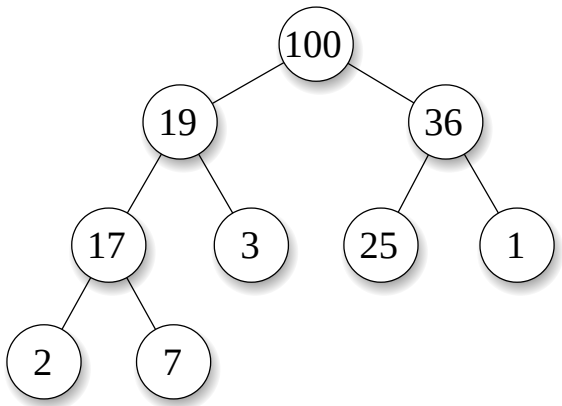
Another problem with MergeSortingTransform and MergingSortedTransform and k-way-merge algorithm, is that during merge we insert only single row into our result.

During row insertion we need to perform **number_of_columns** virtual calls.

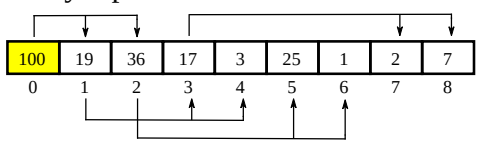
Maybe we can choose data structure that performs minimum amount of comparisons and can help to minimize amount of virtual calls during row insertion ?

Sorting K-Way Merge Heap

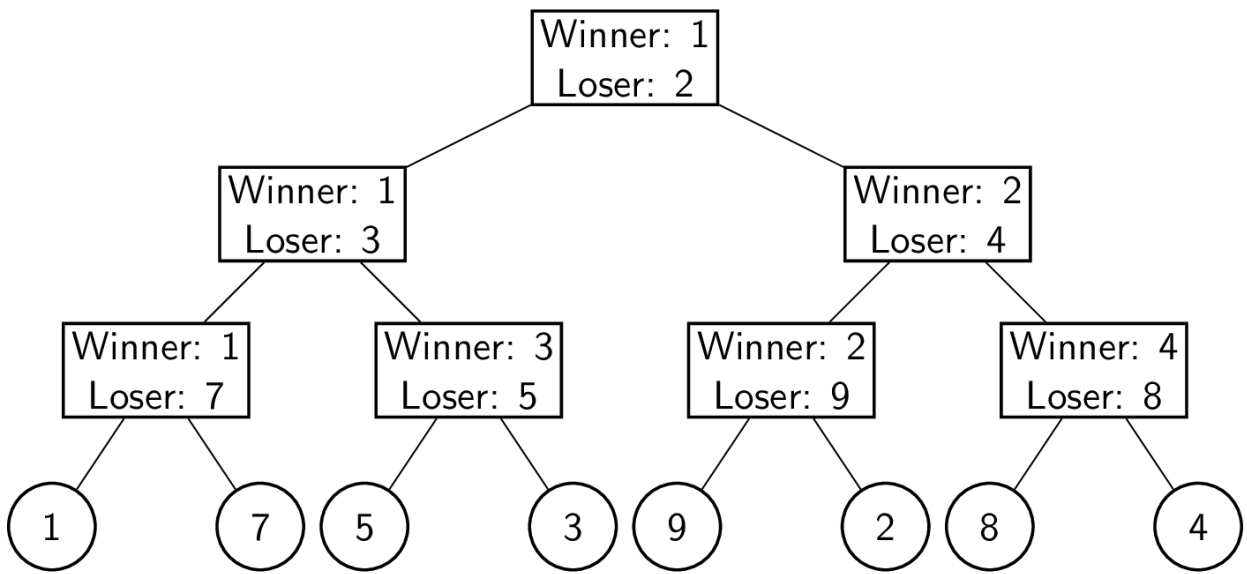
Tree representation



Array representation



Sorting K-Way Merge Winner Tree



Sorting K-Way Merge

K - number of cursors that we want to merge.

N - total number of elements that are in all cursors.

There are 3 data structures that can be used:

1. Heap. Performs around N * log(K) * 1.5 comparisons in worst case.

2. Self balansing BST (AVL, Red-Black). Performs around N * log(K) * constant_factor comparisons.

3. Tournament Tree. Performs N * log(K) comparisons in worst case.

Sorting Batch Heap

The biggest issue with tournament tree is that this data structure always performs N * log(N) comparisons, even for low cardinality data, because all tournament must be replayed.

Theoretically tournament tree should make less comparisons for data with high cardinality. But on real data heap can avoid a lot of comparisons because next minimum cursor is still greater than current minimum cursor.

Sorting Batch Heap

We can choose between heap and tournament tree in runtime using for example bandits approach similar to lz4 optimization, or using statistics to understand if column has high cardinality or low cardinality.

Also there are tournament tree modifications that can allow to store minimum cursor and next minimum cursor.

But for now we decided to use heap property, but improve it with batch interface.

Sorting Batch Heap

The idea is simple, if current minimum cursor in heap is less than next minimum cursor, we compare them, until next mininum cursor is greater, each time we increase batch size.

That way we can insert multiple rows into result, if we sort data that does not have high cardinality (contains duplicates).

Sorting Batch Heap

On real data we can see 2x-10x performance improvement, after replacing heap with batch heap.

```
SELECT DISTINCT Age FROM hits_100m_obfuscated;
┌─Age─┐
│   0 │
│  50 │
│  31 │
│  55 │
│  22 │
│  28 │
└─────┘
SELECT WatchID FROM hits_100m_obfuscated ORDER BY Age FORMAT Null;
```

Was: **4.154** sec (24.07 million rows/s., 216.64 MB/s.)

Now: **0.482** sec (207.47 million rows/s., 1.87 GB/s.)

Conclusion

1. CI/CD infrastructure, especially performance tests, must be the core component of a high performance system.

2. Without deep introspection it is hard to investigate issues with performance.

3. For high performance systems interfaces must be determined by algorithms and data structures.

4. Use tools to profile, benchmark and check how your program works under load.

5. Avoid virtual function calls.

6. Always evaluate algorithms and data structures on real data.

Questions?