3-Way QuickSort in Umbra

Thomas Neumann

Technische Universität München

May 28, 2020

Motivation

```
SELECT x FROM foo ORDER BY x
```

- conceptually a simple sort
- ordering logic depends upon the type of x
- only known at runtime
- can be anything, including compounds

The Current Situation

std::sort(tupleOrder.begin(), tupleOrder.end(), [&compare](void* a, void* b) { return compare(a, b) <0; });

- code for `compare` is generated at query compile time
- usually JITed if the relation is large enough
- the C++ sort calls into generated code for comparisons

Note: We only consider basic sort here, multi-threading is orthogonal

The Problem

```
Samples: 84K of event 'cycles', Event count (approx.): 56467493969
Overhead  Command  Shared Object              Symbol
  27,17%  sql.old  dump1.so                  [.] _1_compare
  26,07%  sql.old  sql.old                   [.] std::__introsort_loop<__gnu_cxx::__normal_iterator<void**, std::vector<void*, umbra::RegionAllocator<void*
   7,08%  sql.old  sql.old                   [.] umbra::StringOperations::writeUnsignedBackwards
   6,84%  sql.old  sql.old                   [.] umbra::SortOperator::sortLocal
   5,90%  sql.old  sql.old                   [.] umbra::GenericSortOperator::sortGlobal<umbra::Backend::FunctorI32PP>
   3,81%  sql.old  sql.old                   [.] umbra::BigIntRuntime::output
   2,99%  sql.old  libstdc++.so.6.0.28       [.] std::ostream::write
   2,85%  sql.old  sql.old                   [.] umbra::GenericSortOperator::storeTuple
   2,29%  sql.old  dump1.so                  [.] _1_map_sort
   2,15%  sql.old  libstdc++.so.6.0.28       [.] std::ostream::sentry::sentry
   1,18%  sql.old  sql.old                   [.] std::vector<void*, umbra::RegionAllocator<void*> >::_M_realloc_insert<void* const&>
   0,75%  sql.old  dump1.so                  [.] _1_sort_tablescan_foo
   0,73%  sql.old  sql.old                   [.] umbra::RuntimeFunctions::printNlResult
   0,64%  sql.old  libstdc++.so.6.0.28       [.] std::ostream::sentry::~sentry
   0,54%  sql.old  dump1.so                  [.] umbra::IntegerRuntime::output@plt
   0,52%  sql.old  libstdc++.so.6.0.28       [.] 0x000000000009da64
   0,48%  sql.old  sql.old                   [.] umbra::(anonymous namespace)::ForwardTarget::write
   0,48%  sql.old  libc-2.31.so              [.] __memset_avx2_erms
   0,41%  sql.old  sql.old                   [.] umbra::(anonymous namespace)::OStreamTarget::prepare
   0,40%  sql.old  sql.old                   [.] umbra::(anonymous namespace)::OStreamTarget::write
   0,35%  sql.old  dump1.so                  [.] umbra::SortOperator::storeTuple@plt
   0,30%  sql.old  libstdc++.so.6.0.28       [.] std::basic_ios<char, std::char_traits<char> >::clear
   0,30%  sql.old  sql.old                   [.] umbra::RuntimeFunctions::bumpResultValue
   0,30%  sql.old  [kernel.kallsyms]         [k] clear_page_erms
   0,30%  sql.old  libstdc++.so.6.0.28       [.] 0x000000000009afb4
   0,26%  sql.old  sql.old                   [.] umbra::(anonymous namespace)::ForwardTarget::prepare
   0,26%  sql.old  sql.old                   [.] 0x0000000001c71e84
   0,26%  sql.old  [kernel.kallsyms]         [k] prepare_exit_to_usermode
   0,23%  sql.old  sql.old                   [.] umbra::IntegerRuntime::output
   0,21%  sql.old  libstdc++.so.6.0.28       [.] 0x000000000009df70
   0,20%  sql.old  libstdc++.so.6.0.28       [.] 0x000000000009df74
   0,19%  sql.old  dump1.so                  [.] umbra::RuntimeFunctions::bumpResultValue@plt
   0,19%  sql.old  dump1.so                  [.] umbra::RuntimeFunctions::printNlResult@plt
   0,16%  sql.old  sql.old                   [.] umbra::SortOperator::storeTuple
   0,13%  sql.old  sql.old                   [.] umbra::OutputTarget::flush
   0,13%  sql.old  [kernel.kallsyms]         [k] native_irq_return_iret
   0,10%  sql.old  [kernel.kallsyms]         [k] swapgs_restore_regs_and_return_to_usermode
   0,10%  sql.old  [kernel.kallsyms]         [k] rmqueue
   0,09%  sql.old  [kernel.kallsyms]         [k] error_entry
   0,07%  sql.old  [kernel.kallsyms]         [k] zap_pte_range.isra.0
```

The Problem (2)

```
Samples: 84K of event 'cycles', 4000 Hz, Event count (approx.): 56467493969
_1_compare  /home/thomas/work/umbra/dump1.so [Percent: local period]
Percent

         Disassembly of section .text:

         0000000000022f0 <_1_compare>:
         _1_compare():
         }

         define int32 @_1_compare(int8* %trampoline, int8* %left, int8* %right) [
         ] {
         body:
         %134 = load int32 %left
18,72      mov  (%rsi),%ecx
         %198 = load int32 %right
11,90      mov  (%rdx),%edx
         %238 = cmpslt i32 %134, %198
 4,39      xor  %esi,%esi
 7,00      cmp  %edx,%ecx
10,77      setl %sil
         %252 = zext i32 %238
         %262 = cmpslt i32 %198, %134
 5,16      xor  %eax,%eax
 5,15      cmp  %ecx,%edx
10,89      setl %al
         %276 = zext i32 %262
         %286 = sub i32 %276, %252
17,09      sub  %esi,%eax
         return %286
 8,93      ← retq
```

The Obvious Solution

```
template <class C> void weakHeapSort(uintptr_t* data, uintptr_t* end, C cmp) {
   constexpr uintptr_t maskShift = sizeof(void*)*8-1, mask = static_cast<uintptr_t>(1) << maskShift;
#define doCmp(a, b) cmp(reinterpret_cast<void*>((a)&(~mask)),reinterpret_cast<void*>((b)&(~mask)))

   if (uint64_t n = end-data;n>1) {
      // Build the heap
      for (uint64_t j=n-1;j>0;--j) {
         auto i = j>>(IntegerOperations<uint64_t>::ctz(j)+1);
         if (doCmp(data[j], data[i])>0) {
            auto tmp=data[i];
            data[i]=data[j];
            data[j]=tmp|mask;
         }
      }

      // Construct the result
      for (uint64_t m=n-1;m>1;--m) {
         // Extract the root element
         auto tmp=data[0];
         data[0]=data[m]&(~mask);
         data[m]=tmp;

         // Restore the heap condition
         uint64_t j = 1, i = 0;
         while (true) {
            uint64_t n=2*j+(data[j]>>maskShift);
            if (n>=m) break;
            j=n;
         }
         while (j>0) {
            if (doCmp(data[j], data[i])>0) {
               auto jMask=data[j]&mask;
               auto tmp=data[i];
               data[i]=data[j]&(~mask);
               data[j]=tmp|(jMask~mask);
            }
            j=j>>1;
         }
      }

      // Process the last two elements explicitly to simplify the regular loop
      auto tmp=data[0];
      data[0]=data[1]&(~mask);
      data[1]=tmp&(~mask);
   }
#undef doCmp
}
```

The Obvious Solution - Weak Heap Sort

- it is compact, beautiful, $O(n \log n)$ runtime, $O(1)$ space, low number of comparisons...
- but slower
- ca. 15% slower when sorting 100K integers
- still a bit slower for 100K strings
- wins only for long strings with common prefix
- our comparison call is still too cheap for weak heap sort

New Idea - Generate Quick Sort

- runtime dominated by partitioning phase
- relatively simple code
- can be generated easily
- then we have one call per partitioning, not per comparison
- hides call overhead

Full (C++) Code

```cpp
constexpr unsigned maxStackDepth = 64;
array<Range, maxStackDepth> stack;
unsigned stackDepth = 0;
void **data = op.tupleOrder.data(), **end = data + op.tupleOrder.size();
while (true) {
   // A trivial range?
   if ((end - data) <= 1) {
      // Take the next chunk from the stack
      if (!stackDepth) break;
      data = stack[--stackDepth].from;
      end = stack[stackDepth].to;
      continue;
   }

   // A large range, choose a pivot
   if ((end - data) >= 4) {
      uint64_t p1 = 0, p2 = (end - data) / 2, p3 = (end - data) - 1;
      uint64_t mp = median3(p1,p2,p3);
      swap(data[mp], data[p3]);
   }

   // Partition the range
   Range pr;
   partition(data, end, &pr);
   Range range1{data, pr.from}, range2{pr.to, end};

   // Put the larger range on the stack
   if ((range1.to - range1.from) < (range2.to - range2.from)) swap(range1, range2);
   if (stackDepth < maxStackDepth) {
      stack[stackDepth++] = range1;
   } else {
      // Fall back to heap sort to avoid excessive recursion. This should never happen
      std::sort(range1.from, range1.to, [this](void* a, void* b) { return compare(a, b) < 0; }); // unreachable in practice
   }

   // And process the second range as tail call
   data = range2.from;
   end = range2.to;
}
```

Partitioning - Traditional

```
void quicksort(Item a[], int l, int r)
{ int i = l-1, j = r; Item v = a[r];
  if (r <= l) return;
  for (;;)
    {
      while (a[++i] < v) ;
      while (v < a[--j]) if (j == l) break;
      if (i >= j) break;
      exch(a[i], a[j]);
    }
  exch(a[i], a[r]);
  quicksort(a, l, i-1);
  quicksort(a, i+1, r);
}
```

Detail (?): How to handle keys equal to the partitioning element

Source: Sedgewick - QuickSort is Optimal

Equal Keys

How to handle keys equal to the partitioning element?

METHOD A: Put equal keys all on one side?

4	4	4	4	4	4	4	4	4	4	4	4	4
4	4	4	4	4	4	4	4	4	4	4	4	(4)

NO: quadratic for n=1 (all keys equal)

METHOD B: Scan over equal keys? (linear for n=1)

1	4	1	1	4	4	4	1	4	1	1	4	4
1	(1)	1	1	4	4	4	1	4	1	(4)	4	4

NO: quadratic for n=2

METHOD C: Stop both pointers on equal keys?

4	9	4	4	1	4	4	4	9	4	4	1	4
1	4	4	4	1	(4)	4	4	9	4	9	4	4

YES: NlgN guarantee for small n, no overhead if no equal keys

Equal Keys (2)

How to handle keys equal to the partitioning element?

METHOD C: Stop both pointers on equal keys?

4	9	4	4	1	4	4	4	9	4	4	1	4
1	4	4	4	1	4	4	4	9	4	9	4	4

YES: NlgN guarantee for small n, no overhead if no equal keys

METHOD D (3-way partitioning): Put all equal keys into position?

4	9	4	4	1	4	4	4	9	4	4	1	4
1	1	4	4	4	4	4	4	4	4	4	9	9

yes, BUT: early implementations cumbersome and/or expensive

Bentley-McIlroy 3-way Partitioning

Partitioning invariant

- move from left to find an element that is not less
- move from right to find an element that is not greater
- stop if pointers have crossed
- exchange
- if left element equal, exchange to left end
- if right element equal, exchange to right end

Swap equals to center after partition

KEY FEATURES

- always uses N-1 (three-way) compares
- no extra overhead if no equal keys
- only one "extra" exchange per equal key

3-Way QuickSort by Sedgewick

```
void quicksort(Item a[], int l, int r)
{ int i = l-1, j = r, p = l-1, q = r; Item v = a[r];
  if (r <= l) return;
  for (;;)
    {
      while (a[++i] < v) ;
      while (v < a[--j]) if (j == l) break;
      if (i >= j) break;
      exch(a[i], a[j]);
      if (a[i] == v) { p++; exch(a[p], a[i]); }
      if (v == a[j]) { q--; exch(a[j], a[q]); }
    }
  exch(a[i], a[r]); j = i-1; i = i+1;
  for (k = l; k < p; k++, j--) exch(a[k], a[j]);
  for (k = r-1; k > q; k--, i++) exch(a[i], a[k]);
  quicksort(a, l, j);
  quicksort(a, i, r);
}
```

3-Way QuickSort - Fixed Version

```
void quicksort(Item a[], int l, int r)
{ int i = l-1, j = r, p = l-1, q = r; Item v = a[r];
  if (r <= l) return;
  for (;;)
    {
      while (a[++i] < v) ;
      while (v < a[--j]) if (j == l) break;
      if (i >= j) break;
      exch(a[i], a[j]);
      if (a[i] == v) { p++; exch(a[p], a[i]); }
      if (v == a[j]) { q--; exch(a[j], a[q]); }
    }
  exch(a[i], a[r]); j = i-1; i = i+1;
  for (k = l; k < p; k++, j--) exch(a[k], a[j]);
  for (k = r-1; k > q; k--, i++) exch(a[i], a[k]);
  quicksort(a, l, j);
  quicksort(a, i, r);
}
```

if ((i<=r)&&(a[r]==v)) exch(a[i++], a[r]);

New Version

```
Samples: 76K of event 'cycles', Event count (approx.): 49888750825
Overhead  Command  Shared Object             Symbol
  41,70%  sql      dump1.so                  [.] _1_partition
   8,19%  sql      sql                       [.] umbra::StringOperations::writeUnsignedBackwards
   8,13%  sql      sql                       [.] umbra::SortOperator::sortLocal
   6,78%  sql      sql                       [.] umbra::GenericSortOperator::sortGlobal<umbra::Backend::FunctorI32PP>
   4,25%  sql      sql                       [.] umbra::BigIntRuntime::output
   3,93%  sql      dump1.so                  [.] _1_compare
   3,55%  sql      sql                       [.] umbra::GenericSortOperator::storeTuple
   3,30%  sql      libstdc++.so.6.0.28       [.] std::ostream::write
   2,62%  sql      dump1.so                  [.] _1_map_sort
   2,31%  sql      libstdc++.so.6.0.28       [.] std::ostream::sentry::sentry
   1,23%  sql      sql                       [.] std::vector<void*, umbra::RegionAllocator<void*> >::_M_realloc_insert<void* const&>
   0,97%  sql      dump1.so                  [.] _1_sort_tablescan_foo
   0,86%  sql      sql                       [.] umbra::RuntimeFunctions::printNlResult
   0,72%  sql      libstdc++.so.6.0.28       [.] std::ostream::sentry::~sentry
   0,61%  sql      libstdc++.so.6.0.28       [.] 0x000000000009da64
   0,58%  sql      dump1.so                  [.] umbra::IntegerRuntime::output@plt
   0,54%  sql      libc-2.31.so              [.] __memset_avx2_erms
   0,50%  sql      sql                       [.] umbra::(anonymous namespace)::ForwardTarget::write
   0,50%  sql      dump1.so                  [.] umbra::SortOperator::storeTuple@plt
   0,48%  sql      sql                       [.] umbra::(anonymous namespace)::OStreamTarget::write
   0,44%  sql      sql                       [.] umbra::(anonymous namespace)::OStreamTarget::prepare
   0,35%  sql      libstdc++.so.6.0.28       [.] std::basic_ios<char, std::char_traits<char> >::clear
   0,33%  sql      sql                       [.] 0x0000000001cd6114
   0,33%  sql      sql                       [.] umbra::RuntimeFunctions::bumpResultValue
   0,32%  sql      libstdc++.so.6.0.28       [.] 0x000000000009afb4
   0,32%  sql      sql                       [.] umbra::(anonymous namespace)::ForwardTarget::prepare
   0,32%  sql      [kernel.kallsyms]         [k] clear_page_erms
   0,26%  sql      dump1.so                  [.] umbra::RuntimeFunctions::bumpResultValue@plt
   0,26%  sql      [kernel.kallsyms]         [k] prepare_exit_to_usermode
   0,26%  sql      libstdc++.so.6.0.28       [.] 0x000000000009df70
   0,25%  sql      libstdc++.so.6.0.28       [.] 0x000000000009df74
   0,23%  sql      dump1.so                  [.] umbra::RuntimeFunctions::printNlResult@plt
   0,22%  sql      sql                       [.] umbra::IntegerRuntime::output
   0,21%  sql      sql                       [.] umbra::SortOperator::storeTuple
   0,17%  sql      sql                       [.] umbra::OutputTarget::flush
   0,15%  sql      [kernel.kallsyms]         [k] native_irq_return_iret
   0,13%  sql      [kernel.kallsyms]         [k] swapgs_restore_regs_and_return_to_usermode
   0,10%  sql      [kernel.kallsyms]         [k] error_entry
   0,07%  sql      [kernel.kallsyms]         [k] rmqueue
   0,06%  sql      [kernel.kallsyms]         [k] free_pcppages_bulk
```

New Version (2)

```
Samples: 76K of event 'cycles', 4000 Hz, Event count (approx.): 49888750825
_1_partition  /home/thomas/work/umbra/dump1.so [Percent: local period]
Percent       %1432 = load int32 %1414
              %1472 = cmpeq i32 %1432, %523
  0,09  21:     cmp  %r11d,(%rbx)
              condbr %1472 %then1 %cont2
              ↓ je   77
  0,01          nop
              %732 = load int8* %710
  6,45  30:     mov  0x8(%r14),%rbx
              %710 = getelementptr int8* %i, i32 1
  1,54          add  $0x8,%r14
              %790 = cmpslt i32 %750, %523
  2,68          cmp  %r11d,(%rbx)
              condbr %790 %loopLeft %loopDoneLeft
  7,82        ↑ jl   30
  4,89          nop
              %916 = load int8* %894
 11,37  40:   mov  -0x8(%rax),%rdi
              %974 = cmpne ptr %894, %tuples
  4,62          cmp  %rax,%r10
              %894 = getelementptr int8* %j, i32 -1
  9,24          lea  -0x8(%rax),%rax
              condbr %1002 %loopRight %loopDoneRight
  3,14        ↓ je   52
  4,55        ├─cmp  (%rdi),%r11d
 18,34        └─jl   40
              %1060 = cmpult ptr %822, %1034
  5,10  52:     cmp  %rax,%r14
              condbr %1060 %then %cont3
  1,33        ↓ jae  88
              store int8* %1110, %822
  3,61          mov  %rdi,(%r14)
              store int8* %1092, %1034
  0,33          mov  %rbx,(%rax)
              %1172 = load int8* %822
  1,01          mov  (%r14),%rdi
              %1230 = cmpeq i32 %1190, %523
                cmp  %r11d,(%rdi)
              condbr %1230 %then0 %cont
  5,12        ↑ jne  21
              %1280 = load int8* %1262
```

Performance

	old			new		
	instructions	cycles	time	instructions	cycles	time
1M ints	1492	931	0.07	1134	736	0.06
1M strings	3085	2069	0.11	2863	1813	0.10

- here: duplicate free. much better with duplicates
- instructions and cycles better
- time difference minor, though. call is surprisingly cheap
- a bit more branch misses

requires more work

- tried standard tricks like insertion sort at the end
- branch free partitioning?

Hoare Partitioning

```
T* partition(T* l, T* r) {
   auto v = l[(r-l-1)>>1];
   auto i = l-1, j = r;
   while (true) {
      while (*(++i)<v);
      while (*(--j)>v);
      if (i>=j) return j+1;
      swap(*i, *j);
   }
}
```

- code is surprisingly subtle
- low number of instructions
- but what about the third partition?
- Edelkampf, Weiß: Build it lazily at the end
- move pivots to the middle, stop after run of 4 non-pivots

Performance (2)

	old			new			lazy		
	instr.	cycles	time	instr.	cycles	time	instr.	cycles	time
1M ints	1492	931	0.07	1134	736	0.06	909	605	0.05
1M strings	3085	2069	0.11	2863	1813	0.10	2154	1455	0.08

- wall clock does not improve that much (too little work)
- instructions and cycles improved significantly, but more branch misses
- reducing branch misses might be an option, too (Andre works on that)

Branch-Free Lomuto Partitioning

```
T* partition(T* l, T* r) {
   auto v = r[-1];
   auto j = l;
   for (auto i = l; i<r; ++i) {
      bool less = (*i) < v;
      swap(*i, *j);
      j += less;
   }
   swap(*j,r[-1]);
   return j;
}
```

- performs a high number of pointless swaps
- but has no branch misses within the loop
- with careful coding 2 loads + 2 stores per loop
- one Hoare round in front to handle pathological cases
- can be combined with lazy third partition

Performance (3)

	old			new			lazy			branch-free		
	instr.	cycles	time	instr.	cycles	time	instr.	cycles	time	instr.	cycles	time
1M ints	1492	931	0.07	1134	736	0.06	909	605	0.05	960	537	0.05
1M strings	3085	2069	0.11	2863	1813	0.10	2154	1455	0.08	2171	1380	0.08