# THE BRANCHLESS PROGRAMMING IN CLICKHOUSE: LOGICAL FUNCTION OPTIMIZATION AS AN EXAMPLE

ZHIGUO ZHOU (zhiguo.zhou@intel.com)

CLOUD SOFTWARE DEVELOPMENT ENGINEER

# AGENDA

- Binary logical function
  - Short-circuit in logic evaluation
  - Branch miss penalty
  - Vectorization optimization
- Ternary logical function
  - Bottleneck analysis
  - Two-phase auto-vectorization
- Implicit short-circuit operators

# BINARY LOGICAL FUNCTION IN WHERE CLAUSE

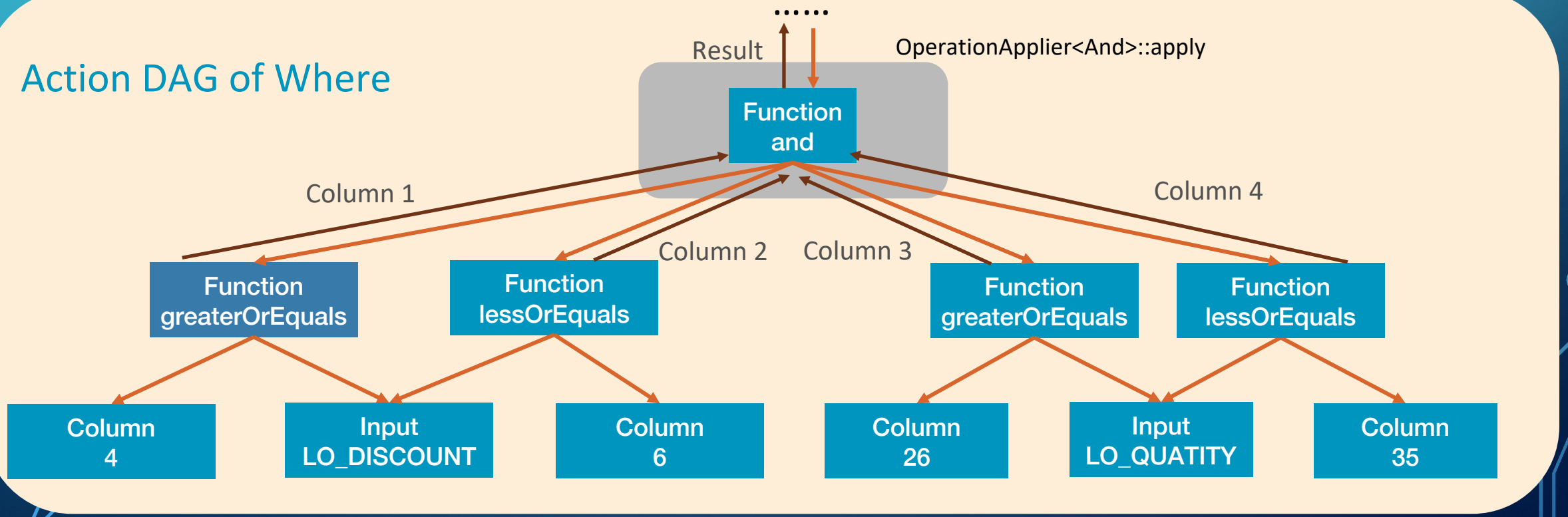
SSB|Q1.2

```
SELECT sum(LO_EXTENDEDPRICE * LO_DISCOUNT) AS revenue
FROM lineorder_flat
WHERE toYYYYMM(LO_ORDERDATE) = 199401
AND LO_DISCOUNT BETWEEN 4 AND 6
AND LO_QUANTITY BETWEEN 26 AND 35;
```

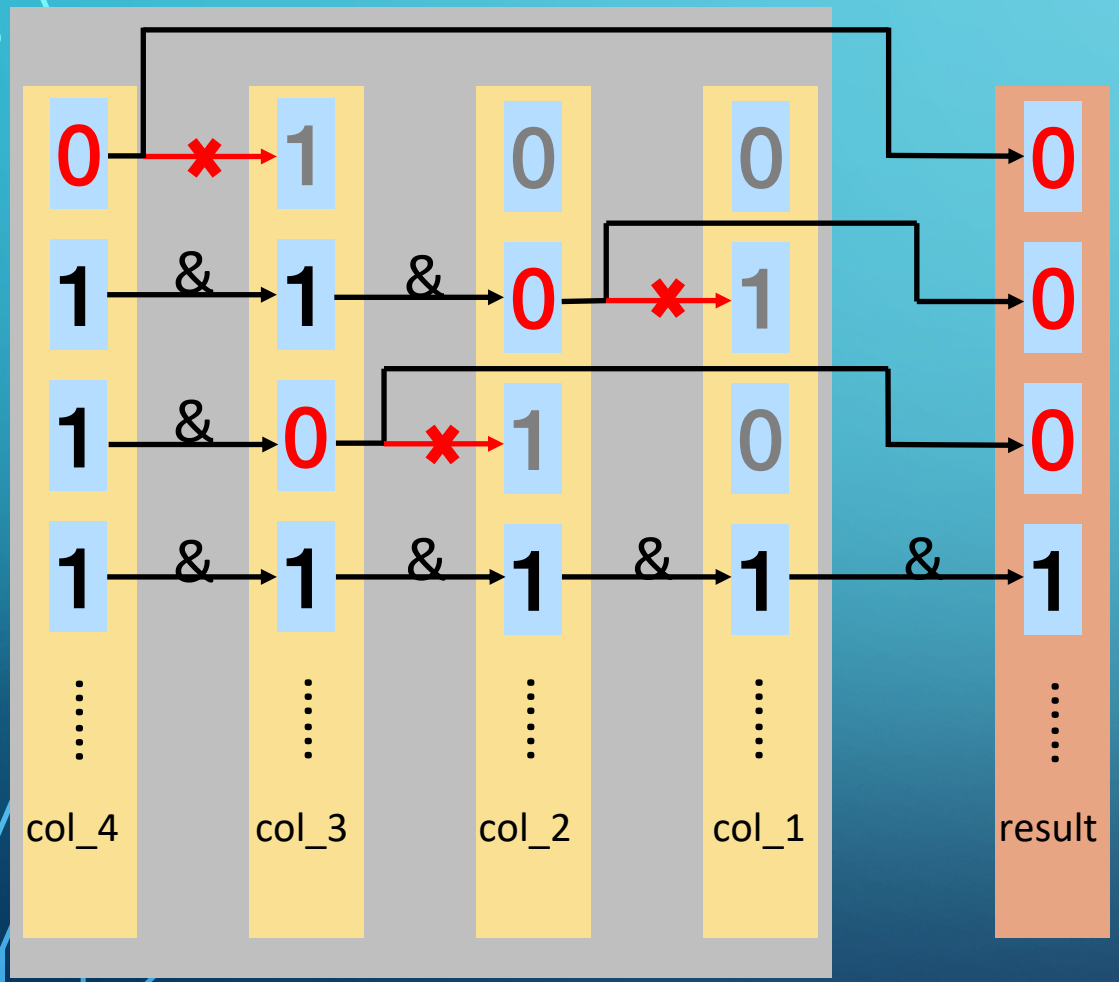
OperationApplier<And>::apply evaluates the columnwise **binary *And*** logical evaluation

Note: Prewhere is applied to toYYYYMM(LO_ORDERDATE)=199401

## Action DAG of Where



# SHORT-CIRCUIT IN LOGICAL FUNCTION



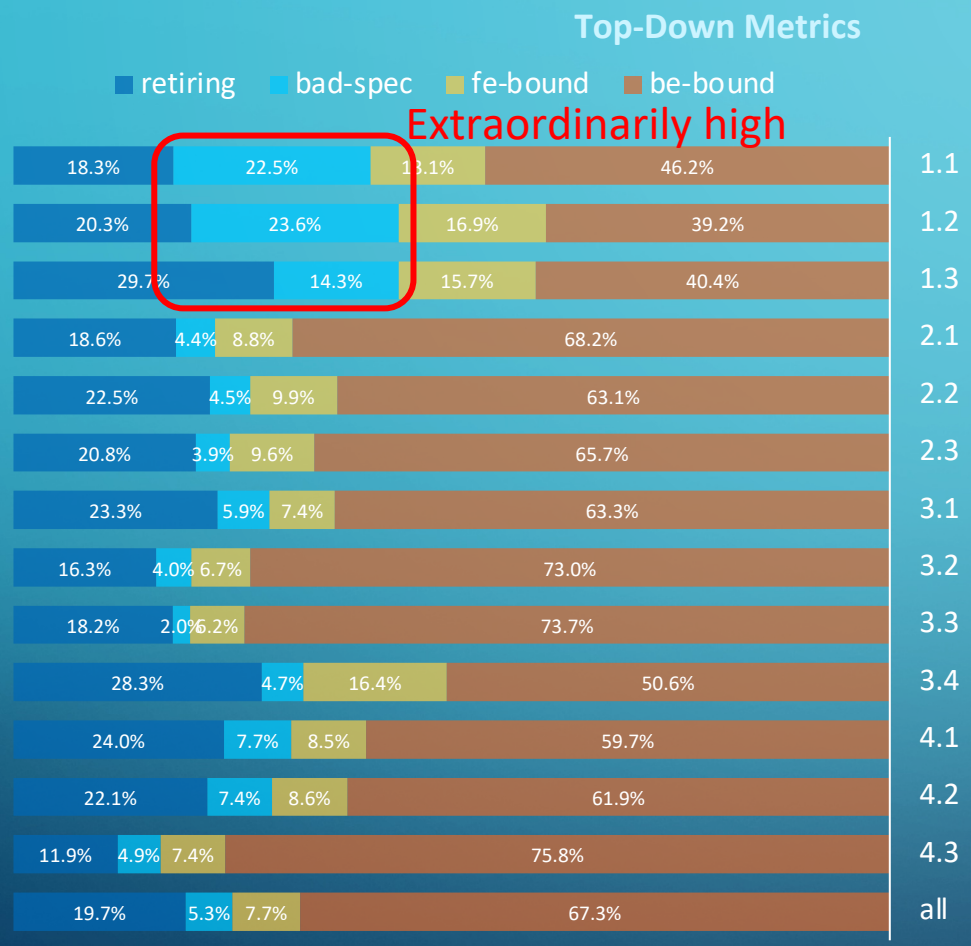

Short-circuit

```
40:   xor   %ebx,%ebx # ebx = 0
./20220928-bad-spec/../src/Functions/FunctionsLogical.cpp:326
      mov   %bl,(%rsi,%rcx,1) # bl: ebx[7:0], rsi: 2nd arg, result_data
./20220928-bad-spec/../src/Functions/FunctionsLogical.cpp:321
      add   $0x1,%rcx # i++
      cmp   %rcx,%rdx # rdx size
    ↓ je    7f
./20220928-bad-spec/../src/Functions/FunctionsLogical.cpp:170 # Op::apply if (CarryResult) branch
4e:   cmpb  $0x0,(%rax,%rcx,1) # col_4[i] cmp 0
./20220928-bad-spec/../src/Functions/FunctionsLogical.cpp:172
    ↑ je    40
./20220928-bad-spec/../src/Common/PODArray.h:327 # PODArray.t_start()
      mov   0x10(%r11),%rbx #
./20220928-bad-spec/../src/Functions/FunctionsLogical.cpp:170
      cmpb  $0x0,(%rbx,%rcx,1) # col_3[i] cmp 0
./20220928-bad-spec/../src/Functions/FunctionsLogical.cpp:172
    ↑ je    40
./20220928-bad-spec/../src/Common/PODArray.h:327
      mov   0x10(%r10),%rbx # col_2
./20220928-bad-spec/../src/Functions/FunctionsLogical.cpp:170
      cmpb  $0x0,(%rbx,%rcx,1) # col_2[i] cmp 0
./20220928-bad-spec/../src/Functions/FunctionsLogical.cpp:172
    ↑ je    40
      mov   0x10(%r9),%rbx # col_1
./20220928-bad-spec/../src/Functions/FunctionsLogical.cpp:191
      cmpb  $0x0,(%rbx,%rcx,1) # col_1 cmp 0
      setne %bl    # bl = ZF # result_data[i] = col_4[i] & col_3[i] & col_2[i] & col_1[i]
./20220928-bad-spec/../src/Functions/FunctionsLogical.cpp:326
      mov   %bl,(%rsi,%rcx,1) # result_data[i] = ZF
./20220928-bad-spec/../src/Functions/FunctionsLogical.cpp:321
      add   $0x1,%rcx
      cmp   %rcx,%rdx
    ↑ jne   4e
```

result[i] = 0

result[i] = ZF

# BAD SPECULATION IN SSB




Bad speculation is closely related to branch misses

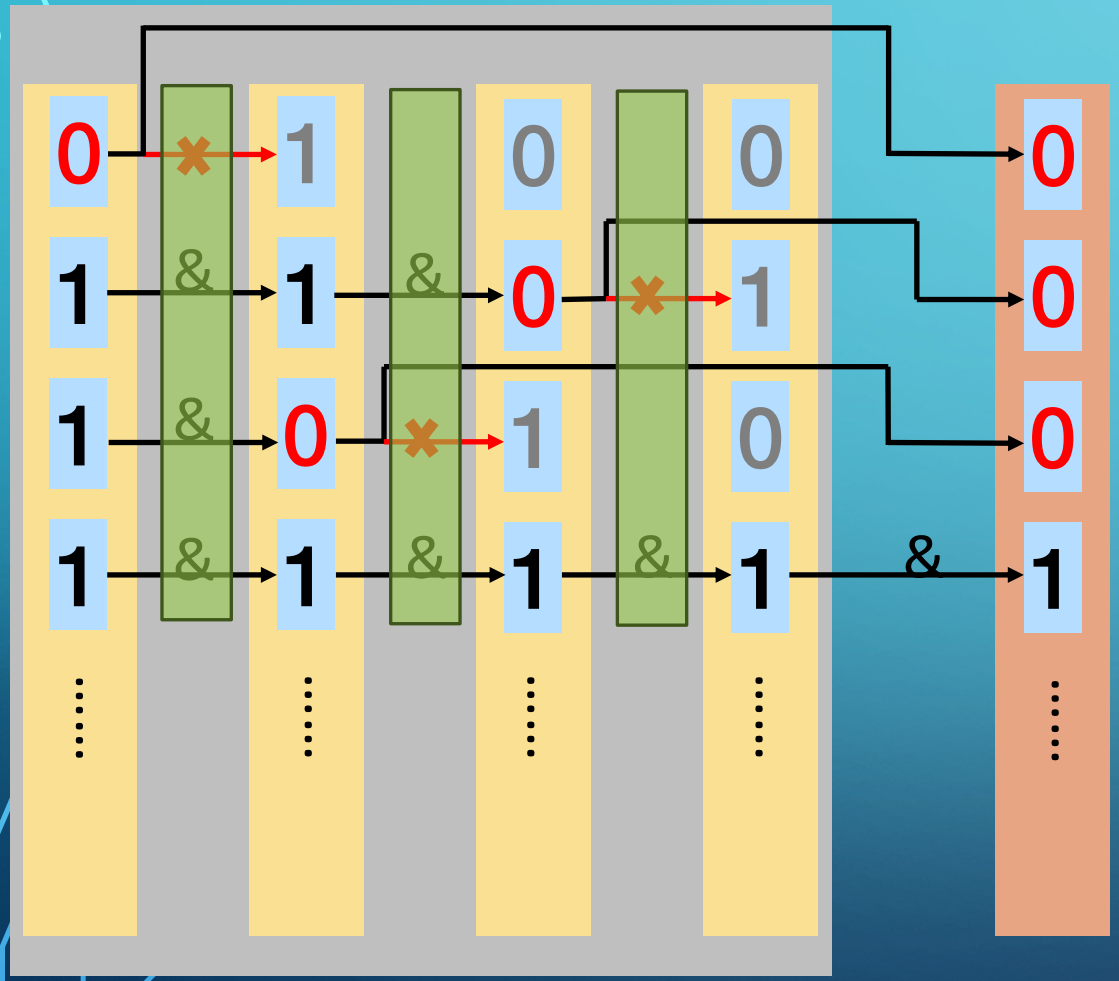
In **Q1.2**, 10.63% of branches are failed to be speculated

| | |
|---|---|
| Branches | 7.26E+10 |
| Branch-misses | 7.72E+09 |
| Branch Miss Rate | 10.63% |

And 83% of the branch misses took place in OperationApplier<And>::apply

```
Samples: 1M of event 'BR_MISP_RETIRED.ALL_BRANCHES_PS', Event count (approx.): 10051727802
Overhead  Command          Shared Object  Symbol
  83.15%  QueryPipelineEx  clickhouse     [.] _ZN2DB12_GLOBAL__N_116OperationApplierINS_22FunctionsLogic
   3.39%  QueryPipelineEx  clickhouse     [.] _ZNK2DB12ColumnVectorIDuE6filterERKNS_8PODArrayIDuLm4096E
   3.36%  QueryPipelineEx  clickhouse     [.] DB::ColumnVector<unsigned int>::filter
   1.16%  QueryPipelineEx  clickhouse     [.] LZ4::(anonymous namespace)::decompressImpl<8ul, true>
   0.93%  QueryPipelineEx  clickhouse     [.] LZ4::(anonymous namespace)::decompressImpl<16ul, true>
```

# SHORT-CIRCUIT HINDERS VECTORIZATION




Vectorization

- Operands packed in vector registers
- SIMD (Single **Instruction** Multiple Data)
- Data parallelism, higher data throughput

Problem

- Distinct code path for each row, distinct instruction for the data

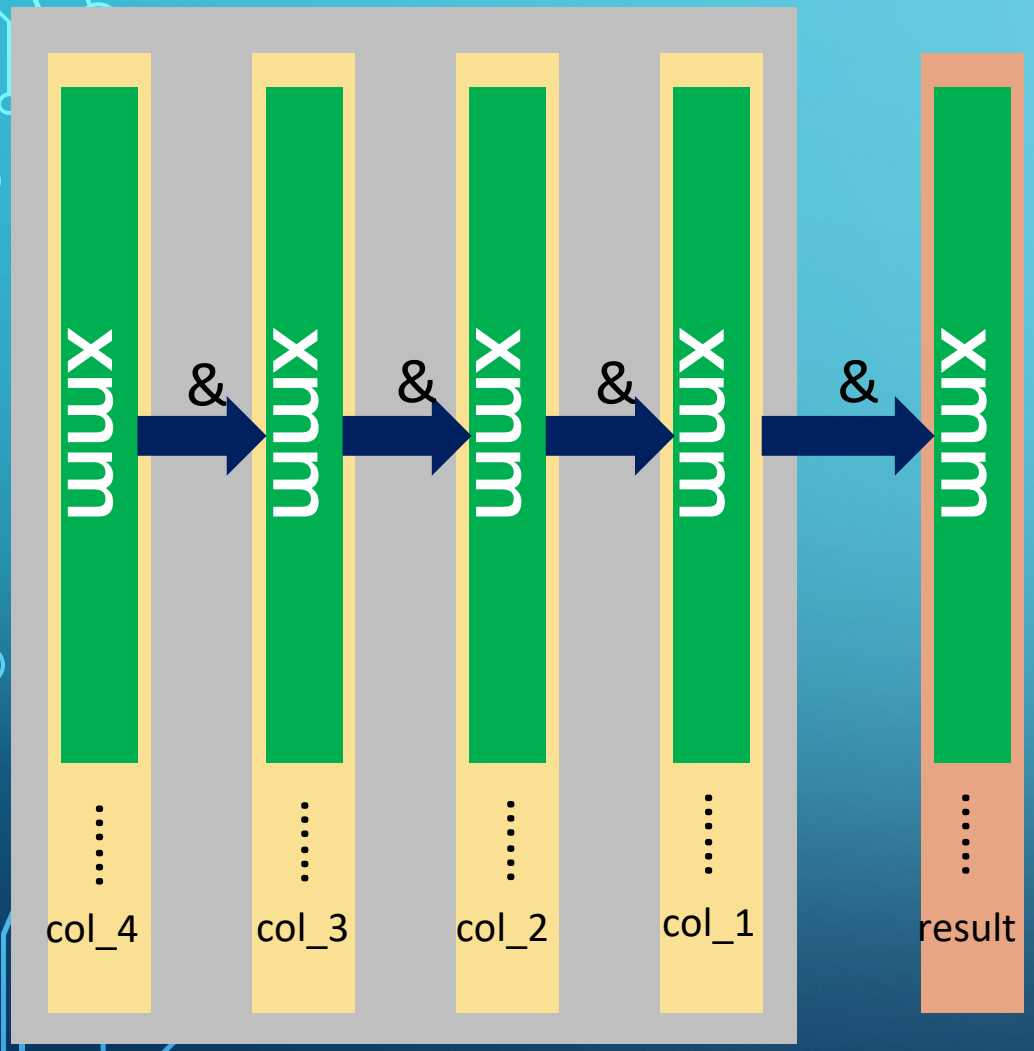
Q: Could short-circuit be safely removed?
A: YES! Short-circuit logical function is equivalent to the standard one semantically.

# OPTIMIZATION: REMOVE SHORT-CIRCUIT

```
Author: Zhiguo Zhou <zhiguo.zhou@intel.com>
Date:   Thu Oct 27 10:35:45 2022 +0800

    Remove short-circuit evaluation in AssociativeApplierImpl::apply

    The short-circuit evaluation was implemented when applying the
    saturable operators (and, or) on a vector of ColumnUInt8. However,
    its control flow would be compiled as a series of conditional
    branch instructions which are hard to predict by the hardware and
    at the same time hinder the vectorization optimization by the
    compiler. This commit removes the short-circuit and evaluates the
    whole expression.

diff --git a/src/Functions/FunctionsLogical.cpp b/src/Functions/FunctionsLogical.cpp
index 2ac7688737..7e52c55e5b 100644
--- a/src/Functions/FunctionsLogical.cpp
+++ b/src/Functions/FunctionsLogical.cpp
@@ -168,10 +168,7 @@ public:
     inline ResultValueType apply(const size_t i) const
     {
         const auto a = !!vec[i];
-        if constexpr (Op::isSaturable())
-            return Op::isSaturatedValue(a) ? a : Op::apply(a, next.apply(i));
-        else
-            return Op::apply(a, next.apply(i));
+        return Op::apply(a, next.apply(i));
     }

 private:
```

#42214: Remove short-circuit in AssociativeApplierImpl::apply

# VECTORIZE BY REMOVING SHORT-CIRCUIT




col_4[0:255] packed-byte compare with all-zero

```
14c8bf60:   f3 41 0f 6f 1c 19       movdqu (%r9,%rbx,1),%xmm3 # col_4[0-15] -> xmm3
14c8bf66:   f3 41 0f 6f 64 19 10    movdqu 0x10(%r9,%rbx,1),%xmm4 # col_4[16-31] -> xmm4
14c8bf6d:   66 41 0f 74 da          pcmpeqb %xmm10,%xmm3 # packed-byte comp with all zero
14c8bf72:   66 41 0f 74 e2          pcmpeqb %xmm10,%xmm4 # packed-byte comp with all zero
14c8bf77:   f3 41 0f 6f 2c 1a       movdqu (%r10,%rbx,1),%xmm5 # col_3
14c8bf7d:   f3 41 0f 6f 74 1a 10    movdqu 0x10(%r10,%rbx,1),%xmm6
14c8bf84:   66 41 0f 74 ea          pcmpeqb %xmm10,%xmm5
14c8bf89:   66 41 0f 74 f2          pcmpeqb %xmm10,%xmm6
14c8bf8e:   f3 41 0f 6f 3c 1b       movdqu (%r11,%rbx,1),%xmm7 # col_2
14c8bf94:   f3 41 0f 6f 4c 1b 10    movdqu 0x10(%r11,%rbx,1),%xmm1
14c8bf9b:   66 41 0f 74 fa          pcmpeqb %xmm10,%xmm7
14c8bfa0:   66 41 0f 74 ca          pcmpeqb %xmm10,%xmm1
14c8bfa5:   f3 41 0f 6f 54 1d 00    movdqu 0x0(%r13,%rbx,1),%xmm2 # col_1
14c8bfac:   f3 41 0f 6f 44 1d 10    movdqu 0x10(%r13,%rbx,1),%xmm0
14c8bfb3:   66 41 0f 74 d2          pcmpeqb %xmm10,%xmm2 #
14c8bfb8:   66 41 0f ef d0          pxor   %xmm8,%xmm2
14c8bfbd:   66 0f df fa             pandn  %xmm2,%xmm7
14c8bfc1:   66 0f df ef             pandn  %xmm7,%xmm5
14c8bfc5:   66 0f df dd             pandn  %xmm5,%xmm3 # byte compare results in col_1-col_4[0-15] in xmm3
14c8bfc9:   66 41 0f 74 c2          pcmpeqb %xmm10,%xmm0
14c8bfce:   66 41 0f ef c0          pxor   %xmm8,%xmm0
14c8bfd3:   66 0f df c8             pandn  %xmm0,%xmm1
14c8bfd7:   66 0f df f1             pandn  %xmm1,%xmm6
14c8bfdb:   66 0f df e6             pandn  %xmm6,%xmm4
14c8bfdf:   66 41 0f db d9          pand   %xmm9,%xmm3
14c8bfe4:   66 41 0f db e1          pand   %xmm9,%xmm4 # byte compare results in col_1-col_4[16-31] in xmm4
14c8bfe9:   f3 0f 7f 1c 1e          movdqu %xmm3,(%rsi,%rbx,1)
14c8bfee:   f3 0f 7f 64 1e 10       movdqu %xmm4,0x10(%rsi,%rbx,1)
```

Exact operation For col1-3

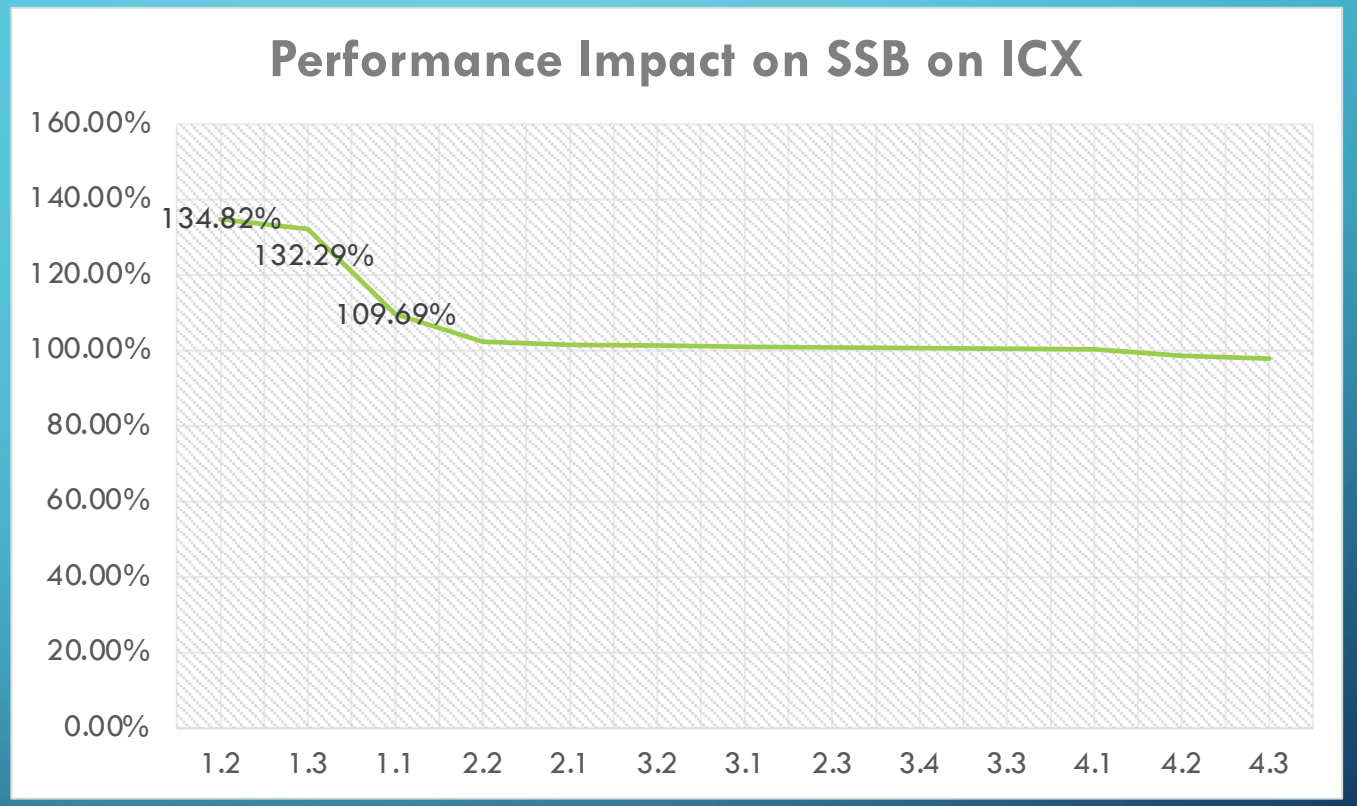
Accumulate [0:127]

Accumulate [128:255]

Write to result column

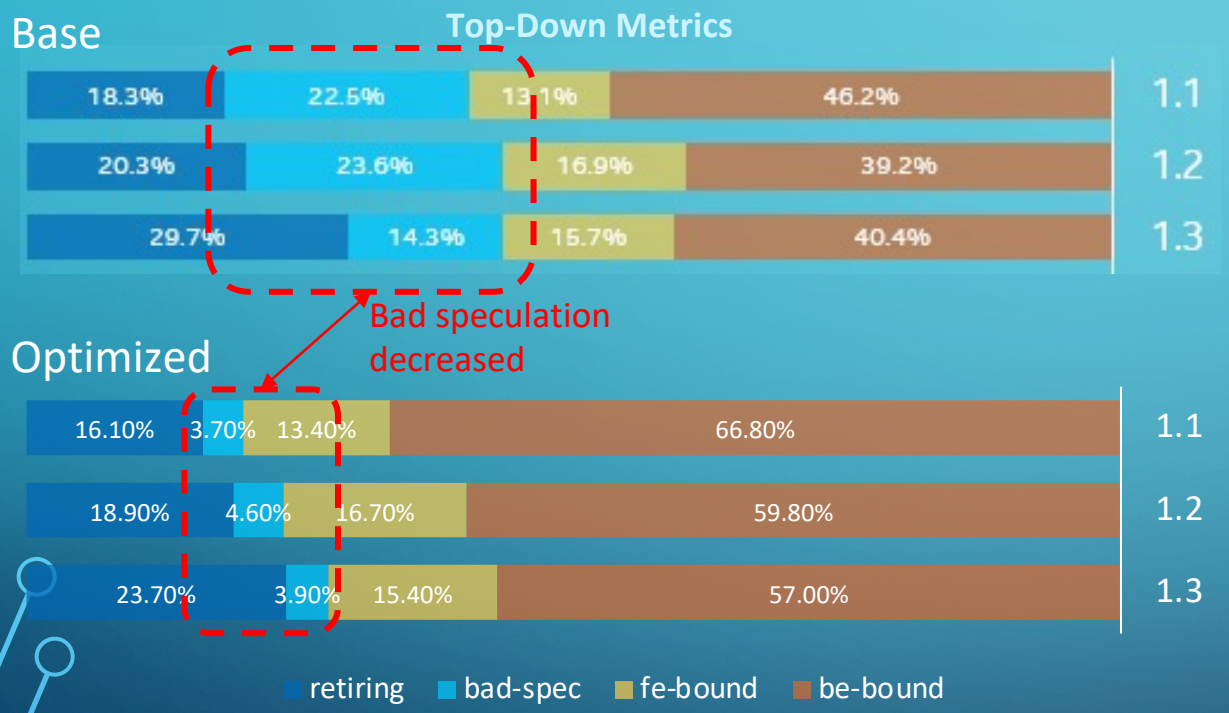
# PERFORMANCE IMPACT

| Query | Gain |
|---|---|
| 1.2 | 134.82% |
| 1.3 | 132.29% |
| 1.1 | 109.69% |
| 2.2 | 102.48% |
| 2.1 | 101.62% |
| 3.2 | 101.36% |
| 3.1 | 100.98% |
| 2.3 | 100.89% |
| 3.4 | 100.78% |
| 3.3 | 100.47% |
| 4.1 | 100.41% |
| 4.2 | 98.65% |
| 4.3 | 97.92% |
| Geomean | **105.74%** |




| | |
|---|---|
| **Microarchitecture** | ICX 8380 |
| **CPUs** | 160 |
| **Sockets** | 2 |

# IMPACT ON ARCHITECTURE




Query 1.2, fixed workload

| | Base | Opt | Gain |
|---|---|---|---|
| Cycles | 7.74E+11 | 5.42E+11 | -29.96% |
| Instructions | 4.50E+11 | 3.78E+11 | -15.91% |
| IPC | 0.58 | 0.70 | 20.06% |
| Branches | 7.26E+10 | 4.97E+10 | -31.58% |
| Branch-misses | 7.72E+09 | 7.44E+08 | -90.36% |
| Branch Miss Rate | 10.63% | 1.50% | -85.90% |

- Less retired instruction/cycle
- Less branches and misses
- Higher IPC
- Much lower branch miss ratio

# IS THIS OPTIMIZATION GENERAL?

```cpp
int main()
{
    size_t size = 10000000;

    for (double zero_ratio = 0.0; zero_ratio < 1.1; zero_ratio += 0.2)
    {
        measureAssociativeApplierPerf<AndImpl, NameAnd>(size, zero_ratio);
        measureAssociativeApplierPerf<OrImpl, NameOr>(size, zero_ratio);
    }
}
```

Column with 10M elements

Pseudo random data with zero-ratio from 0 to 1

```cpp
template <
    typename Op, typename OpName, size_t N = 8>
void measureAssociativeApplierPerf(size_t size, double zero_ratio)
{
    LinearCongruentialGenerator gen;

    for (size_t width = 1; width <= N; ++width)
    {
        UInt8ColumnPtrs uint8_args;
        auto col_res = ColumnUInt8::create(size);

        for (size_t i = 0; i < width; ++i)
        {
            auto col = ColumnUInt8::create();
            auto & col_data = col->getData();
            col_data.resize(size);

            generateRandomUInt8Column(gen, col_data.data(), size, zero_ratio);

            uint8_args.push_back(col.get());
        }

        Stopwatch watch;
        OperationApplier<Op, AssociativeApplierImpl>::apply(uint8_args, col_res->getData(), false);
        std::cerr << OpName::name << " operation on " << width << " columns with the zero ratio of "
    }
}
```

Column count (N)
Range: 1-8

OperationApplier::apply is measured

Microbenchmark shows:
1. All N-zero-ratio combinations enhanced
2. Improvement is obvious for equally distributed cases

| | | Zero Ratio | | | | | |
|---|---|---|---|---|---|---|---|
| Operator | N | 0 | 0.2 | 0.4 | 0.6 | 0.8 | 1 |
| And | 1 | 1.26x | 1.88x | 1.35x | 1.65x | 1.31x | 1.38x |
| | 2 | 6.79x | 37.50x | 68.72x | 64.99x | 37.09x | 8.04x |
| | 3 | 7.57x | 40.93x | 74.17x | 71.94x | 46.52x | 7.69x |
| | 4 | 18.56x | 42.21x | 72.70x | 69.10x | 28.81x | 6.84x |
| | 5 | 19.76x | 39.50x | 59.09x | 59.74x | 33.15x | 7.89x |
| | 6 | 19.08x | 36.83x | 50.71x | 50.27x | 31.16x | 4.85x |
| | 7 | 11.93x | 36.39x | 50.19x | 48.24x | 29.32x | 5.90x |
| | 8 | 20.54x | 28.46x | 45.13x | 43.82x | 26.67x | 5.14x |
| Or | 1 | 1.17x | 1.51x | 1.02x | 1.65x | 1.38x | 1.03x |
| | 2 | 6.25x | 37.06x | 70.42x | 70.77x | 39.78x | 13.43x |
| | 3 | 9.09x | 46.33x | 75.60x | 75.37x | 44.29x | 20.78x |
| | 4 | 4.46x | 45.08x | 73.40x | 70.56x | 46.67x | 25.40x |
| | 5 | 4.38x | 45.16x | 71.02x | 71.75x | 49.68x | 27.52x |
| | 6 | 7.92x | 30.73x | 70.03x | 69.96x | 50.46x | 31.87x |
| | 7 | 7.29x | 39.45x | 63.71x | 68.54x | 50.53x | 32.30x |
| | 8 | 6.48x | 36.11x | 61.48x | 63.59x | 49.84x | 32.81x |

# HOW ABOUT NULLABLE COLUMNS?

New logic state: Null (Unknown)

Ternary Logic Truth Table

| a | b | a And b | a Or b |
|---|---|---|---|
| False | False | False | False |
| False | Null | False | Null |
| False | True | False | True |
| Null | False | False | Null |
| Null | Null | Null | Null |
| Null | True | Null | True |
| True | False | False | True |
| True | Null | Null | True |
| True | True | True | True |

Two problems to solve:

- Numerical Representation (NR) of logic states
- Logical Function (LF) implementation

Naïve implementation
NR: Arbitrary values
LF: if-else chain

ClickHouse implementation
NR: True=0xFF, Null=0x01, False=0x00
LF: AND=&, OR=|

Alternative implementation
NR: True > Null > False
LF: AND=std::min, OR=std::max

# TERNARY LOGICAL FUNCTION

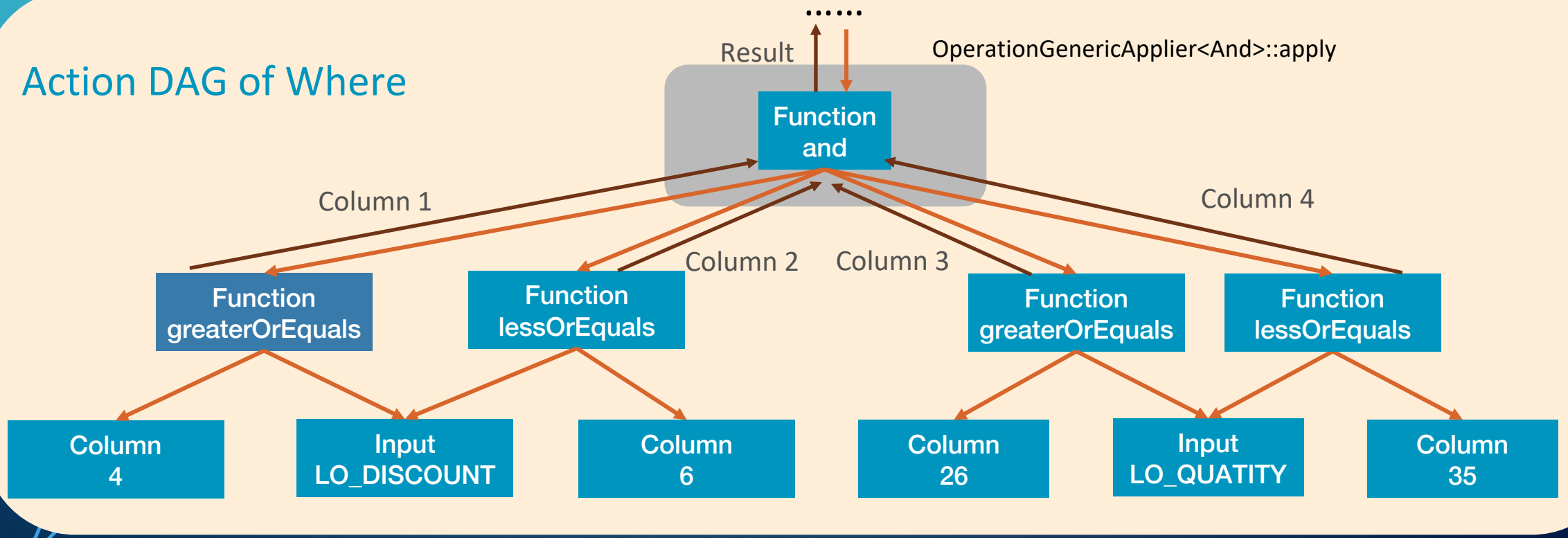
SSB|Q1.2

```
SELECT sum(LO_EXTENDEDPRICE * LO_DISCOUNT) AS revenue
FROM lineorder_flat
WHERE toYYYYMM(LO_ORDERDATE) = 199401
AND LO_DISCOUNT BETWEEN 4 AND 6
AND LO_QUANTITY BETWEEN 26 AND 35;
```

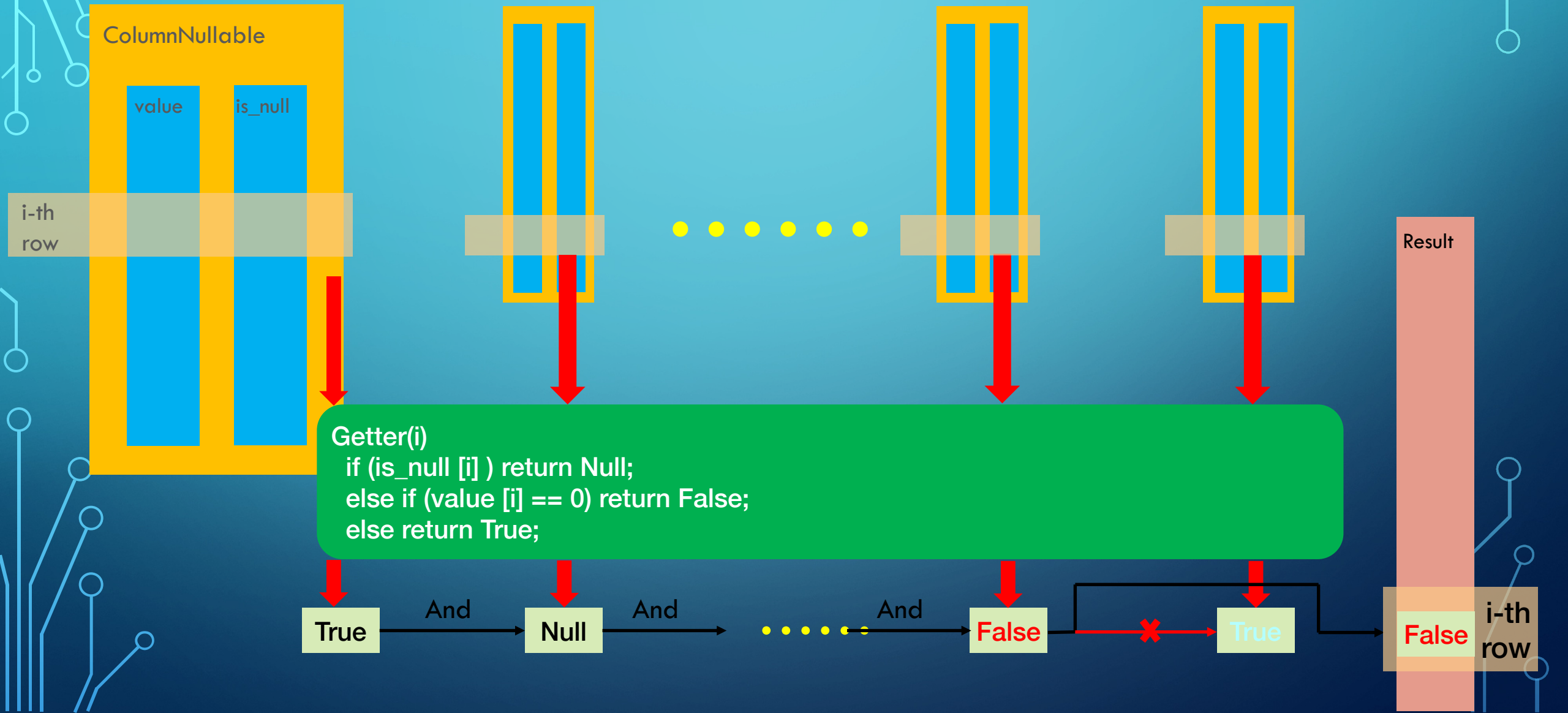
Table schema revision: LO_DISCOUNT/LO_QUANTITY are revised as nullable columns

OperationGenericApplier<And>::apply evaluates the columnwise **ternary *And*** logical evaluation

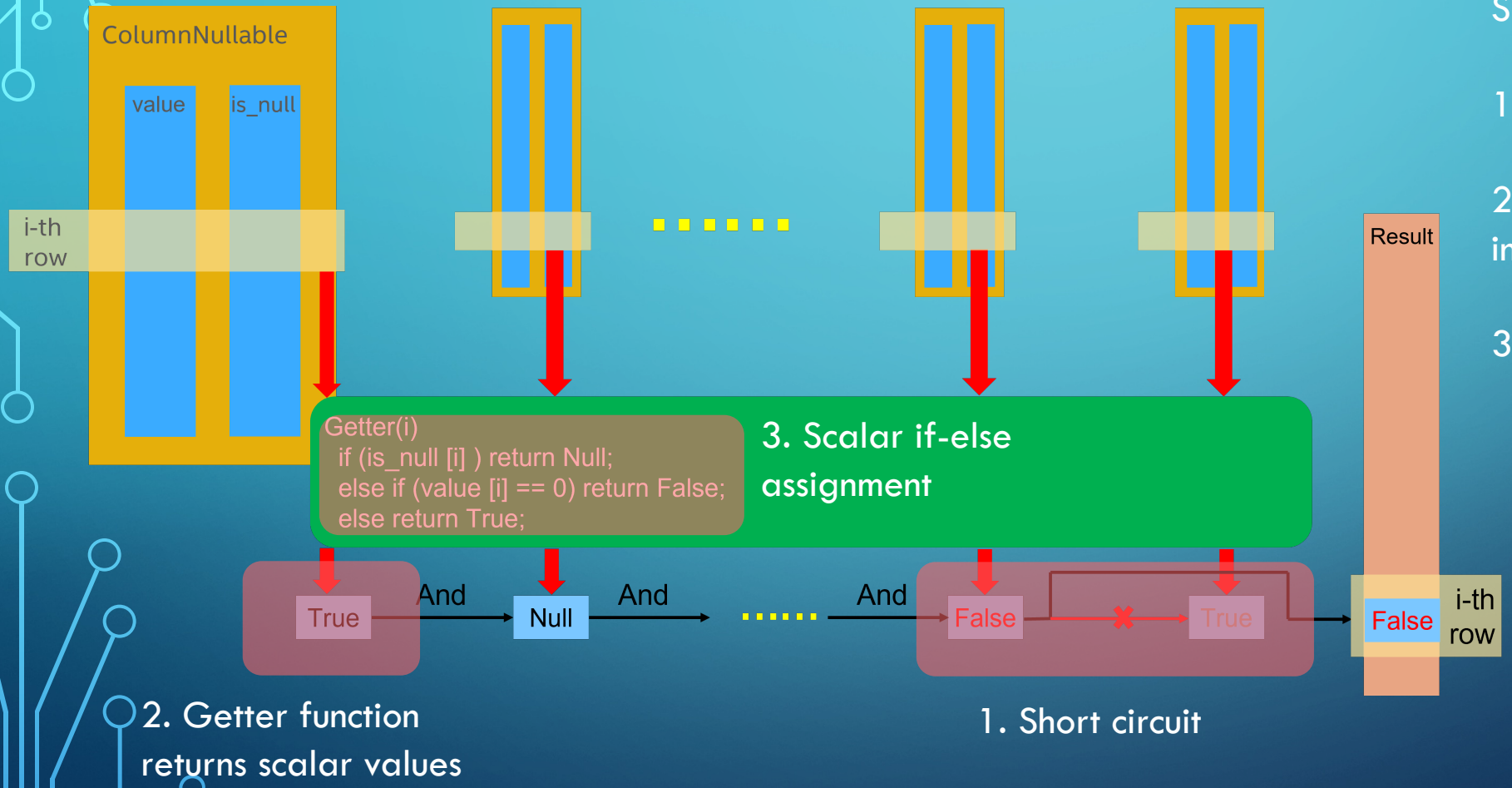
## Action DAG of Where



# COLUMN-WISE TERNARY LOGIC IN CLICKHOUSE



# WHY IT CANNOT BE AUTO-VECTORIZED?




Solution:

1.Remove short circuit

2.Store ternary values in memory

3.How?

# RETHINK GETTER

```
if (is_null [i] ) return Null;
else if (value [i] == 0) return False;
else return True;
```

Implements

| is_null \ value | 0 | 1 |
|---|---|---|
| 0 | False (0x00) | True (0xFF) ⇒ 0x02 |
| 1 | Null (0x01) | Null (0x01) |

Implements

F(is_null, value)
= (( value << 1 ) | is_null ) & ( 1 << !is_null)

F performs vectorizable operations on is_null and value.

Constraints:
1. Distinguish 3 logical states as output
2. Limited vectorizable operations
3. NRs are feasible for the vectorization of later AND/OR implementations

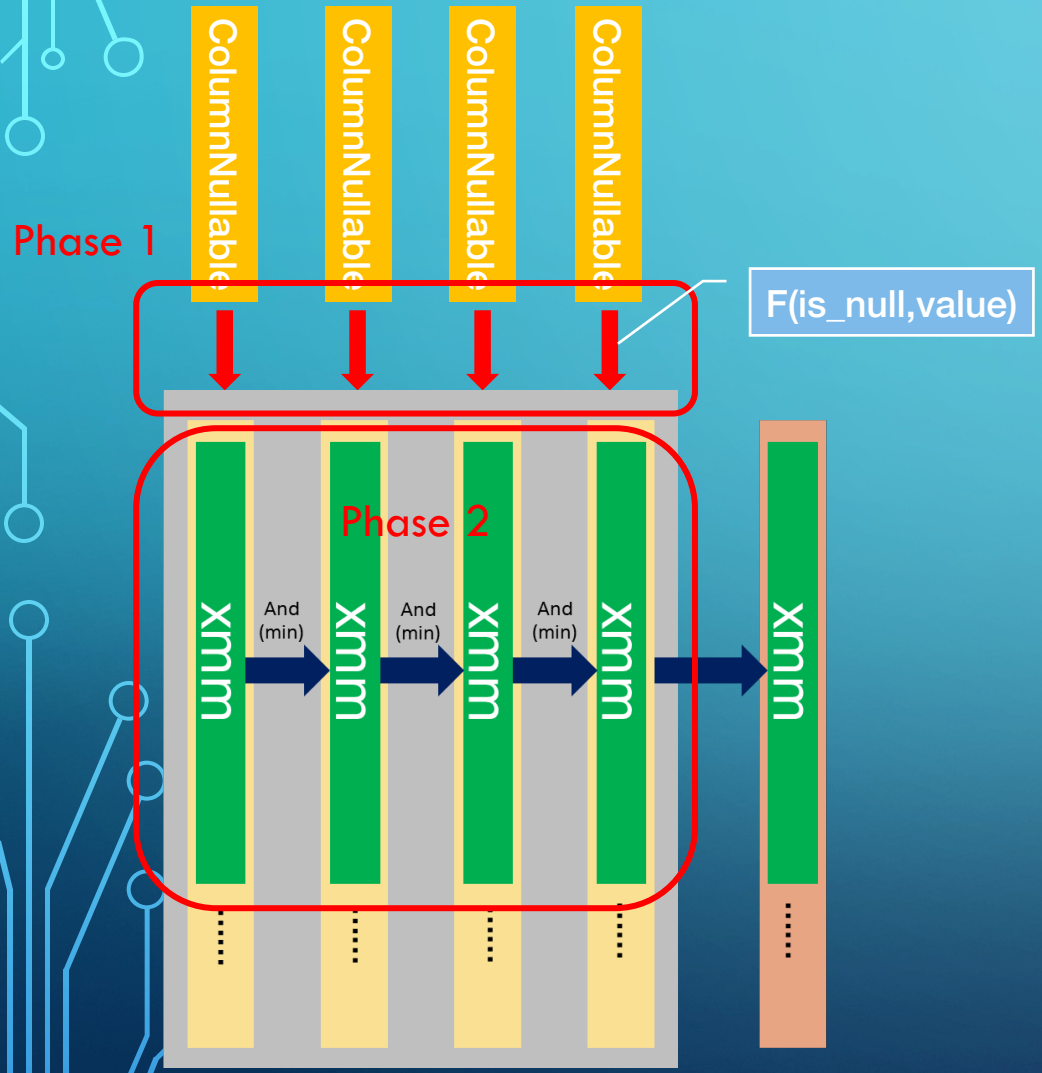
Q: How to implement ternary **And/Or**?

False(0x00) < Null(0x01) < True(0x02)

And is implemented with min
Or is implemented with max

Note: min/max have SIMD instructions

# TWO-PHASE AUTO-VECTORIZATION




Two-phase vectorization:

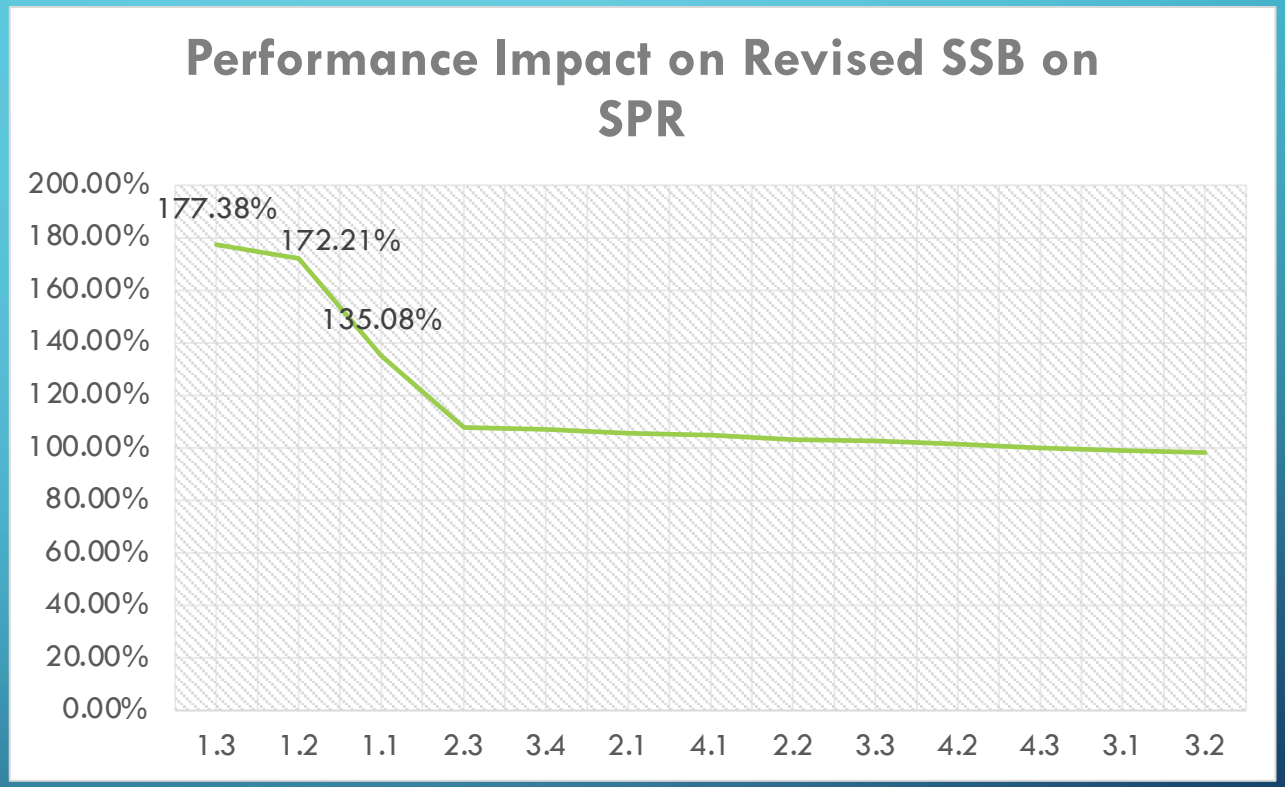
1. Evaluate ternary values from Nullable columns
2. Ternary logic expression calculation

Memory cost for intermediate columns: O(m*n),
for n columns with m elements.

#43669: Vectorize AssociativeGenericApplierImpl::apply

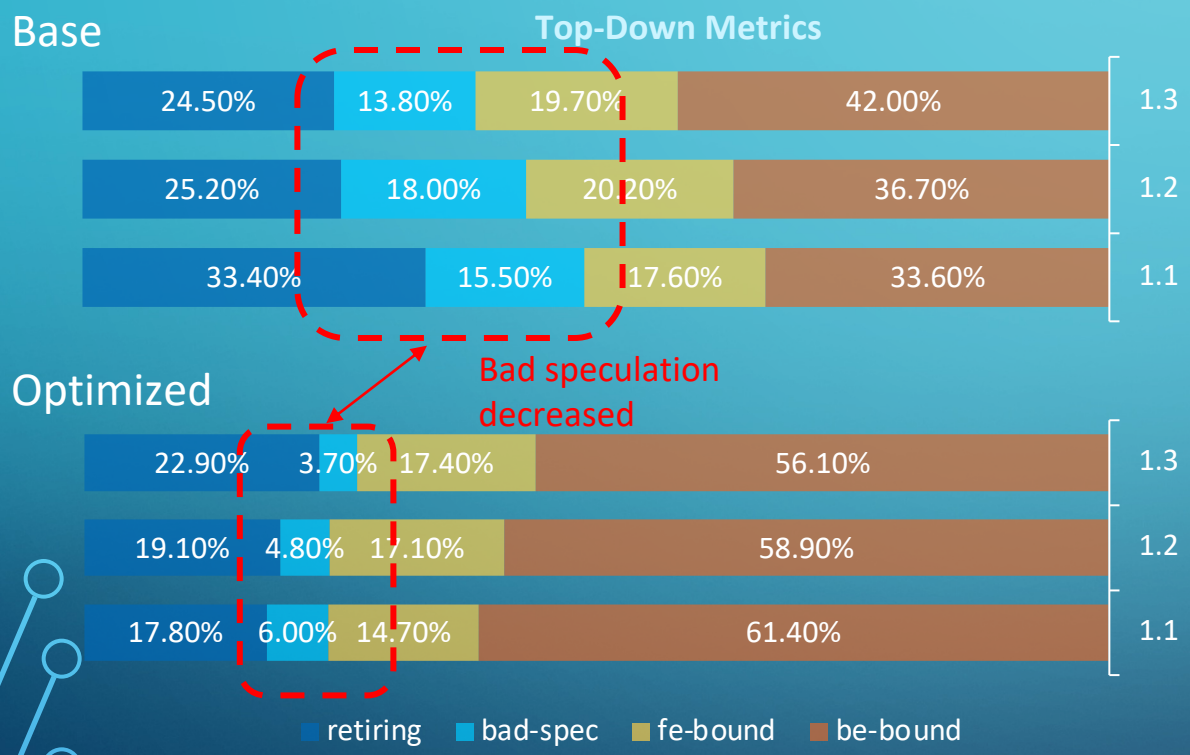
# PERFORMANCE IMPACT

| Query | Gain |
|---|---|
| 1.3 | 177.38% |
| 1.2 | 172.21% |
| 1.1 | 135.08% |
| 2.3 | 107.81% |
| 3.4 | 106.95% |
| 2.1 | 105.52% |
| 4.1 | 104.92% |
| 2.2 | 103.27% |
| 3.3 | 102.57% |
| 4.2 | 101.52% |
| 4.3 | 99.88% |
| 3.1 | 99.04% |
| 3.2 | 98.23% |
| Geomean | **114.02%** |




| | |
|---|---|
| **Microarchitecture** | SPR |
| **CPUs** | 224 |
| **Sockets** | 2 |

# IMPACT ON ARCHITECTURE




Query 1.2, fixed workload

| | Base | Opt | Gain |
|---|---|---|---|
| Cycles | 3.77E+11 | 2.32E+11 | -38.62% |
| Instructions | 4.03E+11 | 2.10E+11 | -47.94% |
| IPC | 1.07 | 0.91 | -15.17% |
| Branches | 6.74E+10 | 2.76E+10 | -59.08% |
| Branch-misses | 3.13E+09 | 5.38E+08 | -82.84% |
| Branch Miss Rate | 4.64% | 1.95% | -58.07% |

- Lower IPC but less retired instruction/cycle
- Less branches and misses
- Much lower branch miss ratio

# MICROBENCH

- Benefit all cases
- Peak gain > 20x
- Overall gain:

  vectorization gain / short-circuit gain

| | | Zero Ratio | | | | | |
|---|---|---|---|---|---|---|---|
| Operator | N | 0 | 0.2 | 0.4 | 0.6 | 0.8 | 1 |
| And | 1 | 6.20 | 6.81 | 7.11 | 6.12 | 7.15 | 7.01 |
| | 2 | 7.87 | 14.74 | 21.34 | 20.18 | 13.13 | 10.05 |
| | 3 | 8.54 | 12.03 | 15.86 | 15.08 | 9.23 | 9.43 |
| | 4 | 8.43 | 10.61 | 12.95 | 11.92 | 7.17 | 9.13 |
| | 5 | 8.74 | 10.02 | 7.14 | 10.35 | 6.41 | 9.09 |
| | 6 | 4.26 | 9.31 | 8.18 | 9.01 | 5.30 | 8.87 |
| | 7 | 7.25 | 9.06 | 7.83 | 4.12 | 4.89 | 8.35 |
| | 8 | 4.91 | 8.80 | 7.43 | 6.42 | 4.37 | 8.55 |
| Or | 1 | 6.86 | 7.10 | 7.04 | 7.00 | 7.16 | 6.72 |
| | 2 | 6.67 | 15.30 | 22.48 | 19.56 | 13.22 | 8.75 |
| | 3 | 4.39 | 10.41 | 16.38 | 13.26 | 11.33 | 5.97 |
| | 4 | 3.41 | 8.43 | 13.41 | 8.76 | 5.96 | 4.72 |
| | 5 | 2.92 | 7.34 | 11.54 | 8.34 | 7.53 | 4.17 |
| | 6 | 2.34 | 6.29 | 9.88 | 7.93 | 7.38 | 3.39 |
| | 7 | 2.21 | 5.73 | 4.94 | 7.63 | 7.46 | 3.14 |
| | 8 | 1.97 | 5.17 | 7.01 | 7.49 | 7.44 | 2.68 |

# IMPLICIT SHORT CIRCUIT: && OPERATOR

src/Columns/FilterDescription.cpp

```
@@ -80,7 +80,12 @@ FilterDescription::FilterDescription(const IColumn & column_)
80  80
81  81          size_t size = res.size();
82  82          for (size_t i = 0; i < size; ++i)
83      -           res[i] = res[i] && !null_map[i];
        83  +       {
        84  +           auto has_val = static_cast<UInt8>(!!res[i]);
        85  +           auto not_null = static_cast<UInt8>(!null_map[i]);
        86  +           /// Instead of the logical AND operator(&&), the bitwise one(&) is utilized
        87  +           res[i] = has_val & not_null;
        88  +       }
84  89
85  90          data = &res;
86  91          data_holder = std::move(mutable_holder);
```

```
457     -   const size_t step = 16;
458     -   size_t i = 0;
459     -   /// NOTE: '&&' must be used instead of '&' for 'AND' operation because UInt8 columns might
460     -   /// value for true and we cannot bitwise AND them to get the correct result.
461     -   for (; i + step < size; i += step)
462     -       for (size_t j = 0; j < step; ++j)
463     -           res_data[i+j] = (c1_data[i+j] && c2_data[i+j]);
464     -   for (; i < size; ++i)
465     -       res_data[i] = (c1_data[i] && c2_data[i]);
        456 +   /// The double NOT operators (!!) convert the non-zeros to the bool value of true (0x01)
        457 +   /// After casting them to UInt8, '&' could replace '&&' for the 'AND' operation implement
        458 +   /// time enable the auto vectorization.
        459 +   for (size_t i = 0; i < size; ++i)
        460 +       res_data[i] = (static_cast<UInt8>(!!c1_data[i]) & static_cast<UInt8>(!!c2_data[i]));
466 461   return res;
467 462   }
468 463
```

result = cond1 && cond2

Is equivalent to

if (cond1== false) result = false //short-circuit
else result = cond2

To remove short-circuit:

result = !!cond1 & !!cond2

#45962: Vectorize filter generation of ColumnNullable in FilterDescription
#60498: Auto-vectorize DB::andFilters

# SUMMARY

- Branchless programming is effective in reduce branch miss penalty and may potentially enable vectorization optimization
- Short-circuit is not always an optimization, especially in the column-wise computations
- The implicit short-circuit operators needs our attention in the performance-sensitive applications

# THANKS FOR YOUR TIME!