ClickHouse向量化执行简介

冯吕 – 查询引擎组

2021-07-30

About Me

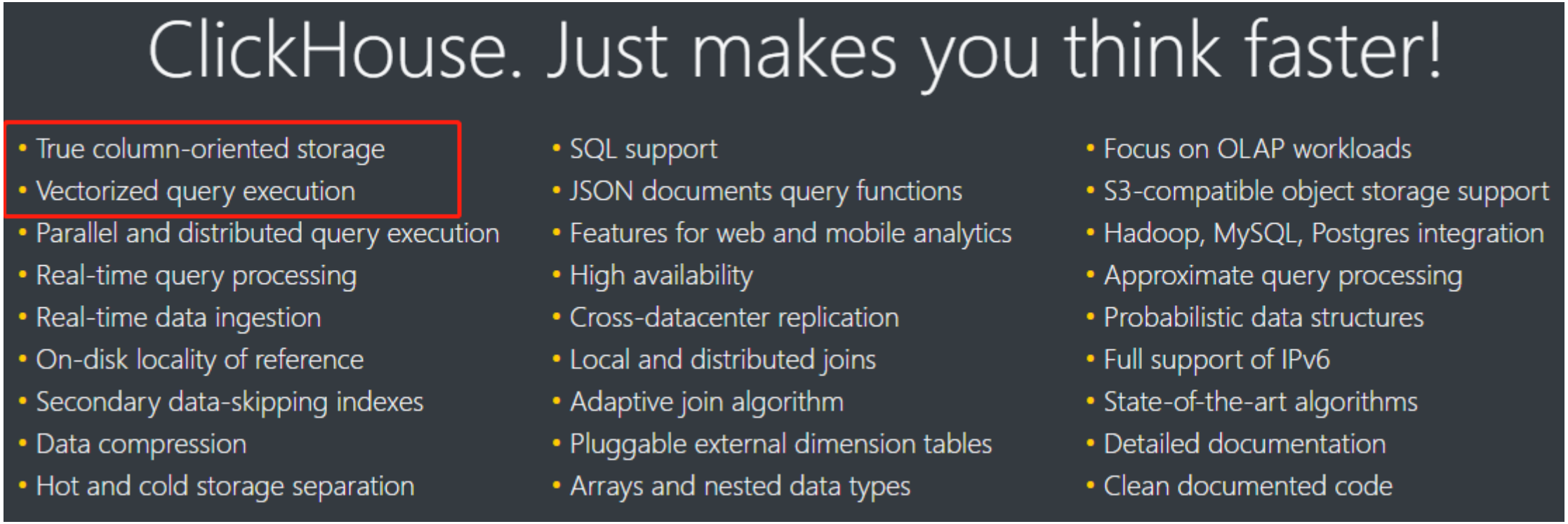
- 中科院计算所硕士二年级学生
- 主要研究方向：OLAP数据库，大数据系统
- ClickHouse Contributor(Top 50), 60+ PRs

目录

目录

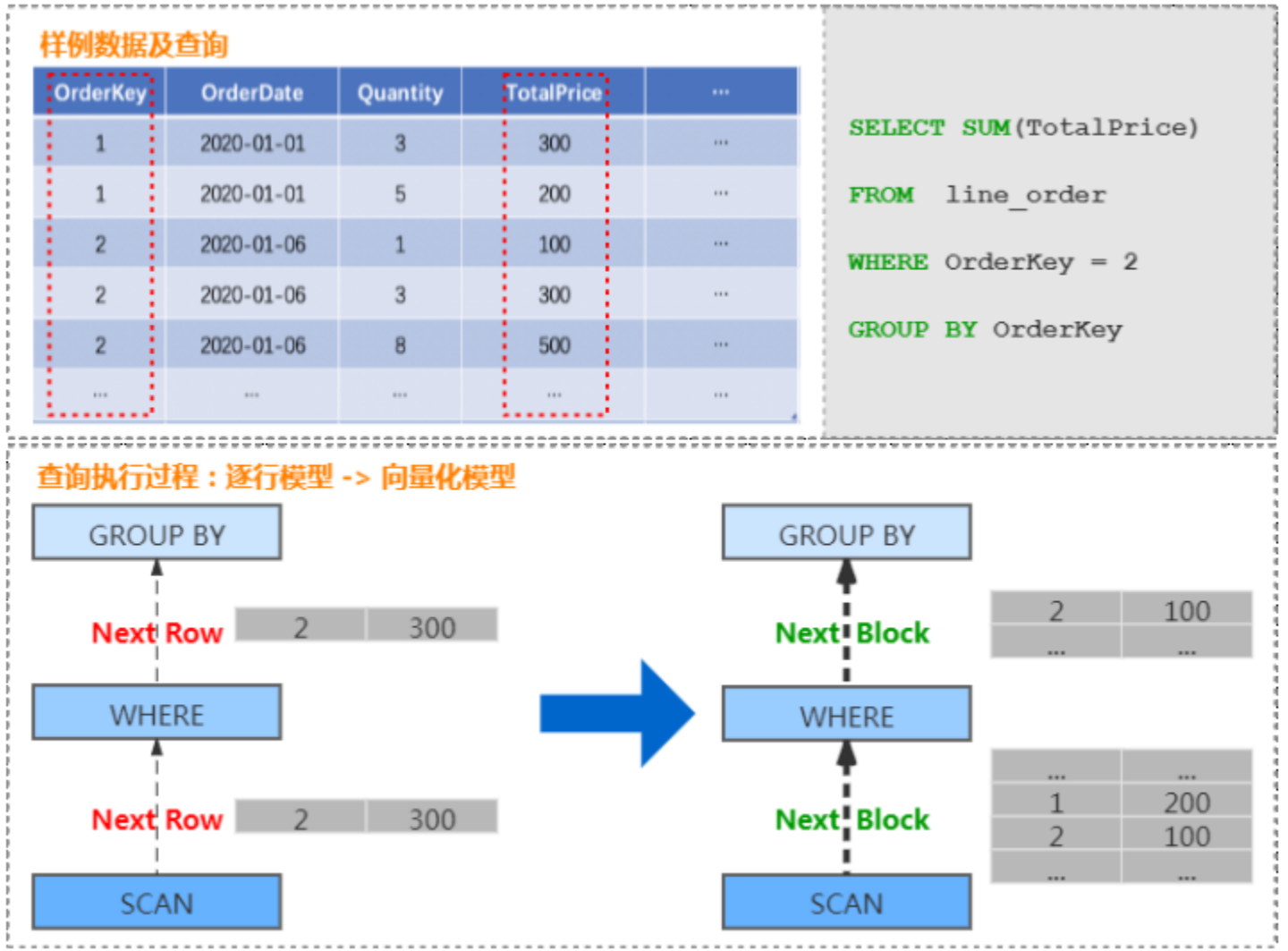
ClickHouse是什么

- 由俄罗斯Yandex开源的高性能OLAP引擎



ClickHouse为什么快

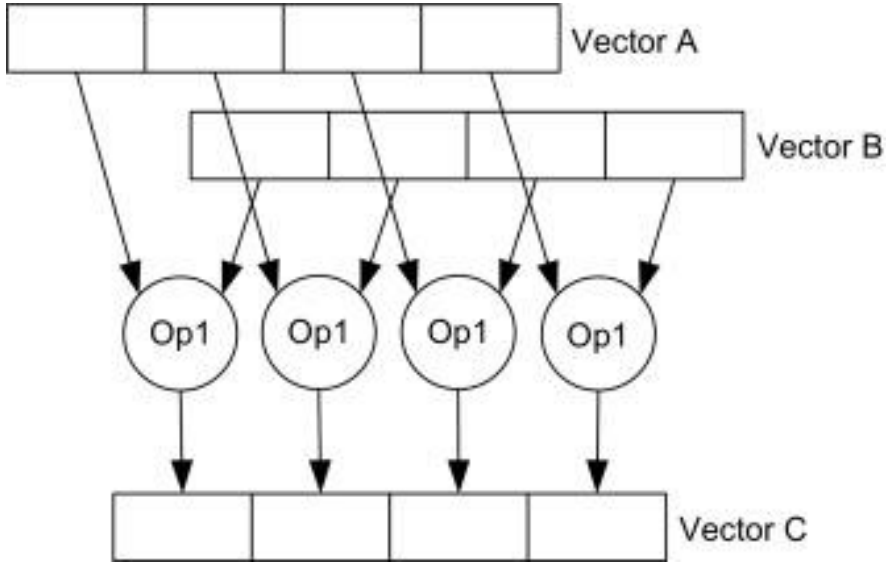
- 列式存储
- 向量化执行

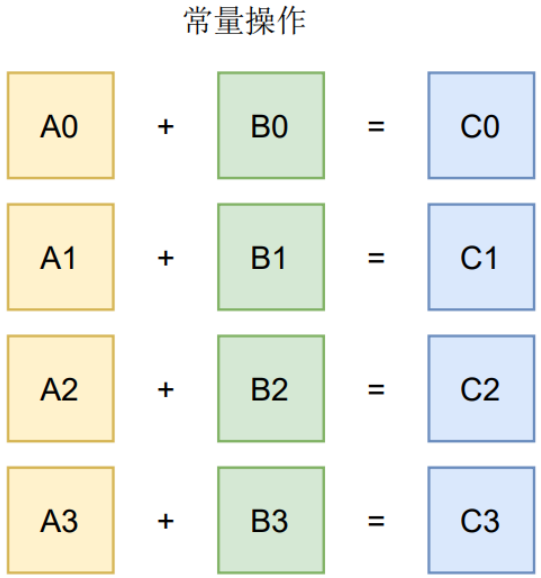


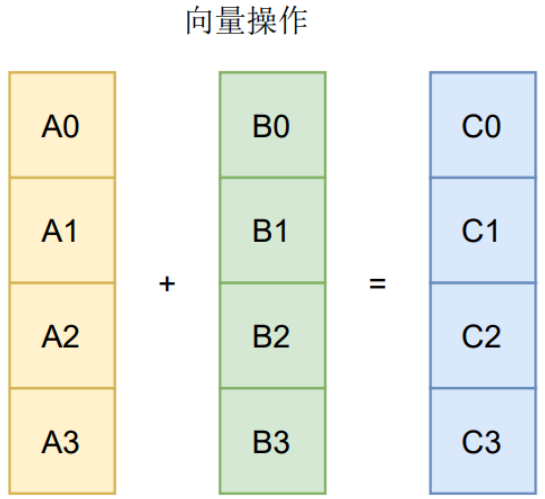
目录

什么是向量化

- 对不同的数据执行同样的一个或一批指令，或者说把指令应用于一个数组/向量，通过CPU数据并行提高性能，即SIMD
- 通俗地说，对一个数组进行连续操作，就可以看作向量化
- 从CPU流水线角度来看，向量化能够充分填满CPU计算单元

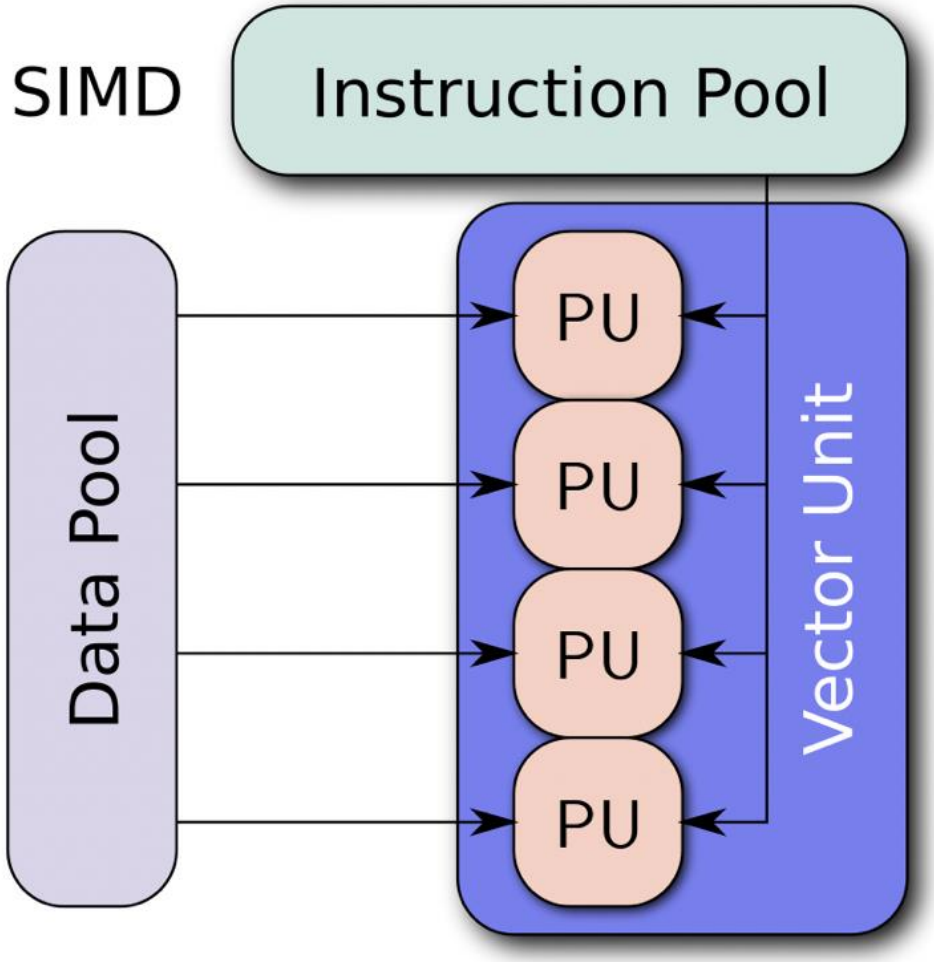






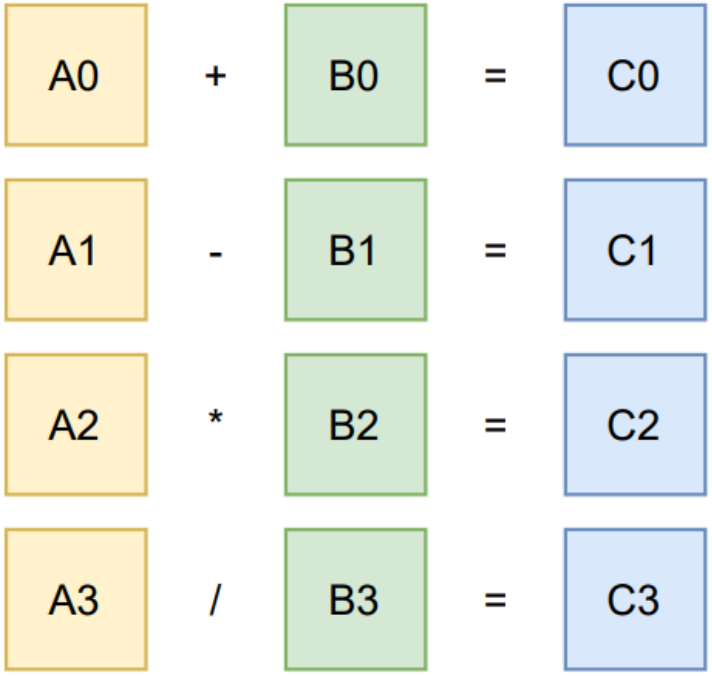
什么是向量化

- 向量化本质上是采用一个控制器来控制多个处理器，同时对一组数据中的每一条分别执行相同的操作，从而实现空间上的并行性的技术；
- “单指令流”指的是同时只能执行一种操作，“多数据流”则指的是在一组同构的数据（即向量）上进行操作；



SIMD缺陷

- 不适用于严重依赖控制流程的任务，即有大量分支、跳转和条件判断的任务
- 主要被用来优化可并行计算的简单场景，以及可能被频繁调用的基础逻辑
- 不能以不同的方式处理不同的数据



实现向量化的三种方式

- 编译器优化
- SIMD内置函数
- 通过内联汇编在代码中嵌入SIMD指令

编译器优化（clang –O3）

```
int sumInt(int *arr)
{
    int sum = 0;
    for (int i = 0; i < 2048; ++i)
    {
        sum += arr[i];
    }
    return sum
}
```

```
00000000004011b0 <_Z6sumintPi>:
  4011b0:       66 0f ef c0             pxor   %xmm0,%xmm0
  4011b4:       31 c0                   xor    %eax,%eax
  4011b6:       66 0f ef c9             pxor   %xmm1,%xmm1
  4011ba:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
  4011c0:       f3 0f 6f 14 87          movdqu (%rdi,%rax,4),%xmm2
  4011c5:       66 0f fe d0             paddd  %xmm0,%xmm2
  4011c9:       f3 0f 6f 44 87 10       movdqu 0x10(%rdi,%rax,4),%xmm0
  4011cf:       66 0f fe c1             paddd  %xmm1,%xmm0
  4011d3:       f3 0f 6f 4c 87 20       movdqu 0x20(%rdi,%rax,4),%xmm1
  4011d9:       f3 0f 6f 5c 87 30       movdqu 0x30(%rdi,%rax,4),%xmm3
  4011df:       f3 0f 6f 64 87 40       movdqu 0x40(%rdi,%rax,4),%xmm4
  4011e5:       66 0f fe e1             paddd  %xmm1,%xmm4
  4011e9:       66 0f fe e2             paddd  %xmm2,%xmm4
  4011ed:       f3 0f 6f 54 87 50       movdqu 0x50(%rdi,%rax,4),%xmm2
  4011f3:       66 0f fe d3             paddd  %xmm3,%xmm2
  4011f7:       66 0f fe d0             paddd  %xmm0,%xmm2
  4011fb:       f3 0f 6f 44 87 60       movdqu 0x60(%rdi,%rax,4),%xmm0
  401201:       66 0f fe c4             paddd  %xmm4,%xmm0
  401205:       f3 0f 6f 4c 87 70       movdqu 0x70(%rdi,%rax,4),%xmm1
  40120b:       66 0f fe ca             paddd  %xmm2,%xmm1
  40120f:       48 83 c0 20             add    $0x20,%rax
  401213:       48 3d 00 08 00 00       cmp    $0x800,%rax
  401219:       75 a5                   jne    4011c0 <_Z6sumintPi+0x10>
  40121b:       66 0f fe c8             paddd  %xmm0,%xmm1
  40121f:       66 0f 70 c1 4e          pshufd $0x4e,%xmm1,%xmm0
  401224:       66 0f fe c1             paddd  %xmm1,%xmm0
```

使用SIMD内置函数

- __m128 : 4个浮点数构成的向量

- __mm_set_ps(d, c, b, a) : 返回一个__m128的向量[a, b, c, d]
- __mm_add_ps(a, b) : 将两个__128的向量按位(pos)相加并返回
- __mm_storeu_ps(d, c) : 将一个_m128向量存到一个float数组中

```
#include <iostream>

#ifdef __SSE2__
  #include <emmintrin.h>
#else
  #warning SSE2 support is not available. Code will not compile
#endif

int main(int argc, char **argv)
{
    __m128 a = _mm_set_ps(4.0, 3.0, 2.0, 1.0);
    __m128 b = _mm_set_ps(8.0, 7.0, 6.0, 5.0);
    __m128 c = _mm_add_ps(a, b);

    float d[4];
    _mm_storeu_ps(d, c);

    std::cout << "result equals " << d[0] << "," << d[1]
              << "," << d[2] << "," << d[3] << std::endl;
    return 0;
}
```

- ClickHouse中主要用到的是就是前两种向量化方式
  - 编译器优化
  - SIMD内置函数

- 一个系统向量化的好坏，和系统的架构设计、代码设计有非常大的关系

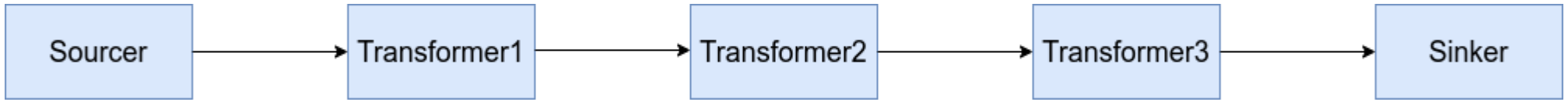
目录

ClickHouse向量化执行

- 高效的Pipeline执行引擎
- 基于ActionsDAG的表达式执行
- 其他：
  - 关键代码大量使用intrinsic指令
  - 消除虚函数调用
  - ……
- 关键：数据按列存储，计算也按列进行

ClickHouse Pipeline

- ClickHouse中，一个SQL的执行分为三个阶段：
  - Parse阶段：将SQL解析为AST
  - Plan阶段：AST -> QueryPlan -> QueryPipeline



  - Execute阶段：完成QueryPipeline的执行。

Transformer编排

- ClickHouse中实现了一系列的Transformer模块，比如：
  - FilterTransform：WHERE/HAVING条件过滤
  - SortingTransform：ORDER BY排序
  - LimitByTransform：LIMIT裁剪

```
SELECT * FROM table1 WHERE id=1 ORDER BY time LIMIT 10
```

```
QueryPipeline::addSimpleTransform(Sourcer)
QueryPipeline::addSimpleTransform(FilterTransform)
QueryPipeline::addSimpleTransform(SortingTransform)
QueryPipeline::addSimpleTransform(LimitByTransform)
QueryPipeline::addSimpleTransform(Sinker)
```

Transformer编排

- Sourcer只有输出端口，没有输入端口，Sinker只有输入端口，没有输出端口，其他Transformer既有输入端口，也有输出端口
- 需要在下层构建DAG实现数据的流通关系：

```
connect(Sourcer.OutPort, FilterTransform.InPort)
connect(FilterTransform.OutPort, SortingTransform.InPort)
connect(SortingTransform.OutPort, LimitByTransform.InPort)
connect(LimitByTransform.OutPort, Sinker.InPort)
```

Pipeline调度

- ClickHouse定义了一系列的Transformer状态：

```
enum class Status
{
    NeedData  // 等待数据流进入
    PortFull, // 管道流出端阻塞
    Finished, // 完成状态，退出
    Ready,    // 切换到 work 函数，进行逻辑处理
    Async,    // 切换到 schedule 函数，进行异步处理
    Wait,     // 等待异步处理
    ExpandPipeline,      // Pipeline 需要裂变
};
```

- Executor完成Transformer的执行和状态迁移

- Pipeline执行时，在各个Transformer之间的数据是以Block为单位流动的，因此能够按列对数据进行计算，充分发挥向量化执行的效率

Expression计算

- 在ClickHouse中，表达式计算是向量化执行发挥作用的一个重要地方，同时也是ClickHouse Pipeline的一个重要组成部分

```
SELECT a, a + b FROM t
```

```
┌─explain─────────────────────────────────────────────────────────────────┐
| Expression ((Projection + Before ORDER BY))                             |
|   SettingQuotaAndLimits (Set limits and quota after reading from storage) |
|     ReadFromPreparedSource (Read from NullSource)                       |
└─────────────────────────────────────────────────────────────────────────┘
```

ExpressionActions

- ClickHouse中表达式计算是通过ExpressionActions来完成的
- 表达式分析 -> ActionsDAG(由表达式构成的一个DAG)
- 节点类型:

```
enum class ActionType
{
        /// Column which must be in input.
        INPUT,
        /// Constant column with known value.
        COLUMN,
        /// Another one name for column.
        ALIAS,
        /// Function arrayJoin. Specially separated because it changes the number of rows.
        ARRAY_JOIN,
        FUNCTION,
};
```

ActionsDAG

- Node实际上描述的是一个列的计算
- DAG方便表达表达式之间的依赖关系
- 基于DAG，更方便对Action进行优化
  - 删除不需要的表达式
  - 子表达式编译（JIT）
  - 节点拆分或合并

示例：

```
SELECT a, a + b FROM t WHERE b > 10
```

```
┌─explain─────────────────────────────────────────────────────────────────────┐
| Expression ((Projection + Before ORDER BY))                                 |
| Actions: INPUT : 0 -> a Int32 : 0                                           |
|          INPUT : 1 -> b Int32 : 1                                           |
|          FUNCTION plus(a : 0, b :: 1) -> plus(a, b) Int64 : 2               |
| Positions: 0 2                                                              |
|   Filter (WHERE)                                                            |
|   Filter column: greater(b, 10) (removed)                                   |
|   Actions: INPUT :: 0 -> a Int32 : 0                                        |
|            INPUT : 1 -> b Int32 : 1                                         |
|            COLUMN Const(UInt8) -> 10 UInt8 : 2                              |
|            FUNCTION greater(b : 1, 10 :: 2) -> greater(b, 10) UInt8 : 3     |
|   Positions: 0 1 3                                                          |
|     SettingQuotaAndLimits (Set limits and quota after reading from storage) |
|       ReadFromPreparedSource (Read from NullSource)                         |
└─────────────────────────────────────────────────────────────────────────────┘
```

- ExpressionActions会对ActionsDAG进行拓扑排序，得到表达式执行的序列，之后，该表达式序列作用到Block上面，按列进行计算，从而充分发挥了向量化执行的效率。

Plus函数的计算

- Plus函数以两个输入列为参数，产生一个新的输出列

```
    template <OpCase op_case>
    static void NO_INLINE process(const A * __restrict a, const B * __restrict b, ResultType * __restrict c,
size_t size)
    {
        for (size_t i = 0; i < size; ++i)
            if constexpr (op_case == OpCase::Vector)
                c[i] = Op::template apply<ResultType>(a[i], b[i]);
            else if constexpr (op_case == OpCase::LeftConstant)
                c[i] = Op::template apply<ResultType>(*a, b[i]);
            else
                c[i] = Op::template apply<ResultType>(a[i], *b);
    }
```

- 在执行时，数组的长度已知，类型已知，并且没有任何分支跳转语句

apply函数

```
template <typename Result = ResultType>
static inline NO_SANITIZE_UNDEFINED Result apply(A a, B b)
{
    /// Next everywhere, static_cast - so that there is no wrong result in expressions of the form Int64 c =
UInt32(a) * Int32(-1).
    if constexpr (is_big_int_v<A> || is_big_int_v<B>)
    {
        using CastA = std::conditional_t<std::is_floating_point_v<B>, B, A>;
        using CastB = std::conditional_t<std::is_floating_point_v<A>, A, B>;

        return static_cast<Result>(static_cast<CastA>(a)) + static_cast<Result>(static_cast<CastB>(b));
    }
    else
        return static_cast<Result>(a) + b;
}
```

- apply函数已经被内联，因此不会发生函数调用

__restrict修饰符

- 向编译器表明，在该指针的生命周期内，只有该指针本身或直接由它产生的指针能够用来访问该指针指向的对象
- 作用：限制指针别名，从而帮助编译器进行优化
- #PR9304: 通过该关键字使整体查询性能提升5%~200%

内置intrinsic库函数

- 在ClickHouse中，有大量地方直接使用SSE intrinsic指令来实现向量化，比如，计算Filter中1的个数

```
size_t countBytesInFilter(const UInt8 * filt, size_t sz)
{
    size_t count = 0;
    const Int8 * pos = reinterpret_cast<const Int8 *>(filt);
    const Int8 * end = pos + sz;

#if defined(__SSE2__) && defined(__POPCNT__)
    const Int8 * end64 = pos + sz / 64 * 64;

    for (; pos < end64; pos += 64)
        count += __builtin_popcountll(toBits64(pos));
#endif
    for (; pos < end; ++pos)
        count += *pos != 0;
    return count;
}
```

toBits64函数

```
#if defined(__SSE2__) && defined(__POPCNT__)
/// Transform 64-byte mask to 64-bit mask.
static UInt64 toBits64(const Int8 * bytes64)
{
    static const __m128i zero16 = _mm_setzero_si128();
    UInt64 res =
        static_cast<UInt64>(_mm_movemask_epi8(_mm_cmpeq_epi8(
            _mm_loadu_si128(reinterpret_cast<const __m128i *>(bytes64)), zero16)))
        | (static_cast<UInt64>(_mm_movemask_epi8(_mm_cmpeq_epi8(
            _mm_loadu_si128(reinterpret_cast<const __m128i *>(bytes64 + 16)), zero16))) << 16)
        | (static_cast<UInt64>(_mm_movemask_epi8(_mm_cmpeq_epi8(
            _mm_loadu_si128(reinterpret_cast<const __m128i *>(bytes64 + 32)), zero16))) << 32)
        | (static_cast<UInt64>(_mm_movemask_epi8(_mm_cmpeq_epi8(
            _mm_loadu_si128(reinterpret_cast<const __m128i *>(bytes64 + 48)), zero16))) << 48);

    return ~res;
}
#endif
```

消除虚函数调用

- 在C++中，消除虚函数调用也是提高向量化的一个重要手段

- 以#PR17403为例，在ClickHouse中，聚合函数能够把数据聚合到中间状态，聚合函数的接口类实现了addBatch方法，其会通过一个for循环调用add方法将一批数据聚合到中间状态，派生类需要实现add方法，这导致了虚函数的调用，无法实现for循环的向量化执行，在该PR中，消除了countIf聚合函数的addBatch对add方法的虚函数调用，从而实现向量化执行，性能提升达到7倍。

总结

- 一个系统的向量化能力，和系统的架构设计有非常大的关系

- 大多数情况下，我们无法面向指令编程，但可以在后期对一些性能热点进行特殊优化，比如直接使用intrinsicn内置指令来实现某些需要向量化的地方，尽可能消除虚函数调用等

End