DataFun.

# ClickHouse向量化执行与Pipeline设计

冯吕 ClickHouse社区贡献者

DataFun.

# About Me

- ClickHouse Top 50 Contributor, 100+ merged PRs
  - https://github.com/ucasfl
- MS student at Institute of Computing Technology, CAS
  - Expected to graduate in July
- Internship at Tencent WeChat
  - Working on ClickHouse development

# 目录 CONTENT

# 01

# ClickHouse 简介

# ClickHouse是什么

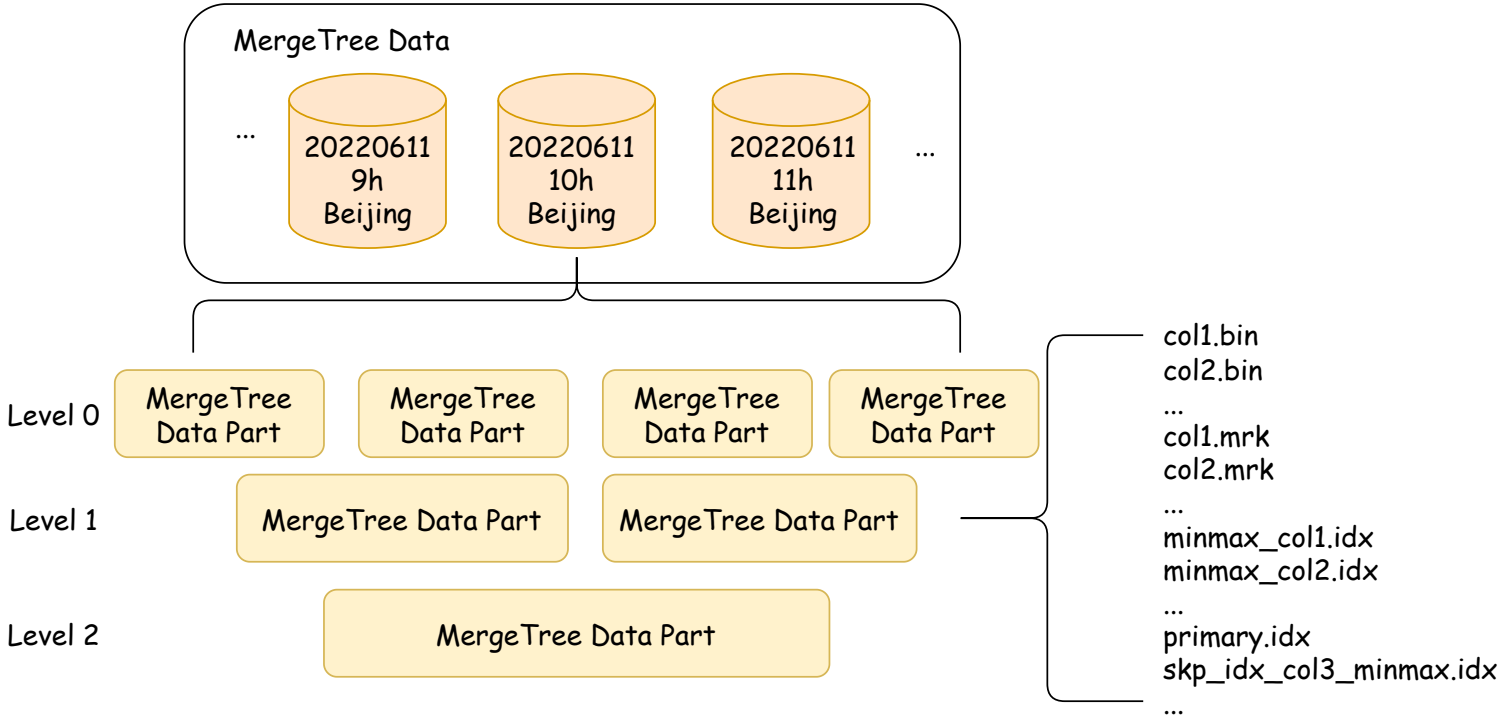
- 一个高效的OLAP数据库
- 最早由Yandex公司开源（2016年）
- 现已从Yandex独立出去 ⟶ ClickHouse, Inc.

# ClickHouse是什么

- 工程艺术品
  - 现有技术的充分运用
- 极致性能优化
  - Substring search：从几十种算法中选择出最合适的一种
  - Aggregation：不同数据类型使用不同的Hash表
- 核心特性：
  - True Column-Oriented Storage
  - Vectorized Query Execution

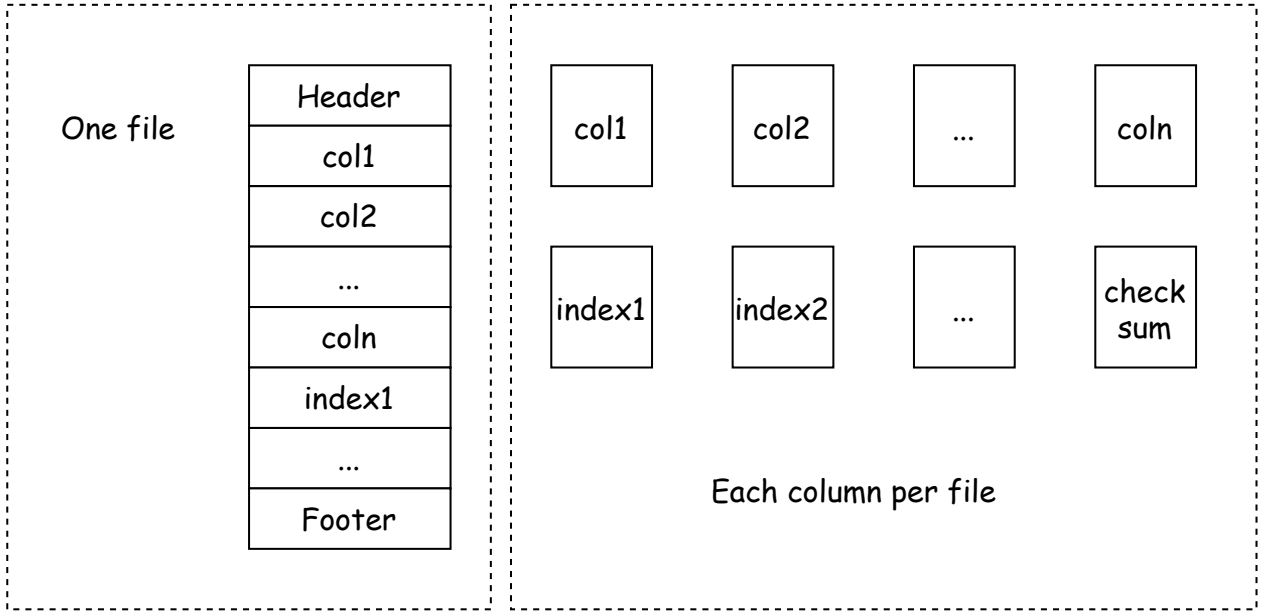
# ClickHouse核心特性

- 基于MergeTree(类似LSM Tree)的列式存储



# 列式存储

- 其他列式存储 VS ClickHouse列式存储



# ClickHouse核心特性

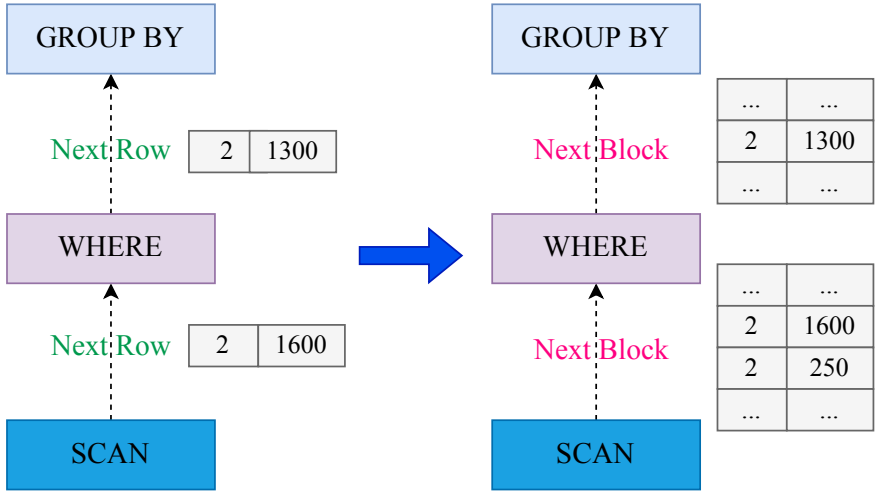
- 向量化执行：
  - ➢ 以行(Row)为单位 -> 以Block为单位，按列对数据进行处理

样例数据及查询

| OrderKey | OrderDate | Quantity | TotalPrice |
|---|---|---|---|
| 1 | 2022-01-01 | 5 | 900 |
| 1 | 2022-01-01 | 3 | 500 |
| 2 | 2022-01-06 | 7 | 1300 |
| 2 | 2022-01-06 | 9 | 1600 |
| 2 | 2022-01-06 | 2 | 250 |
| ... | ... | ... | ... |

```
SELECT SUM(TotalPrice)
FROM line_order
WHERE OrderKey = 2
GROUP BY OrderKey;
```

查询执行：逐行模型 -> 向量化模型

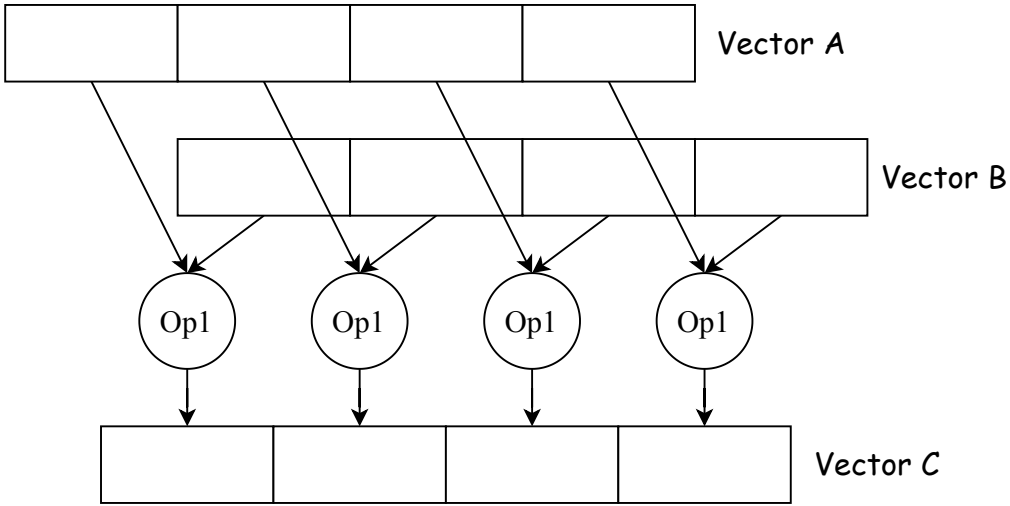


# 02

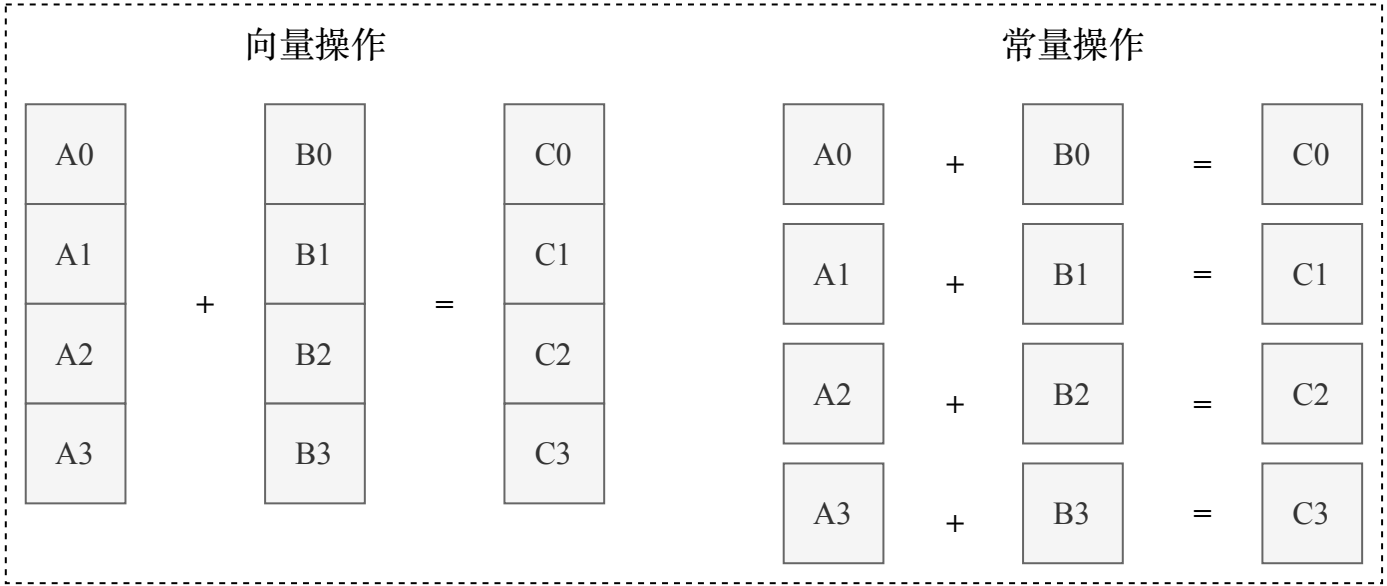
# 向量化执行

# 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，即SIMD
- 通俗地说，对一个数组进行连续操作，即可看做向量化



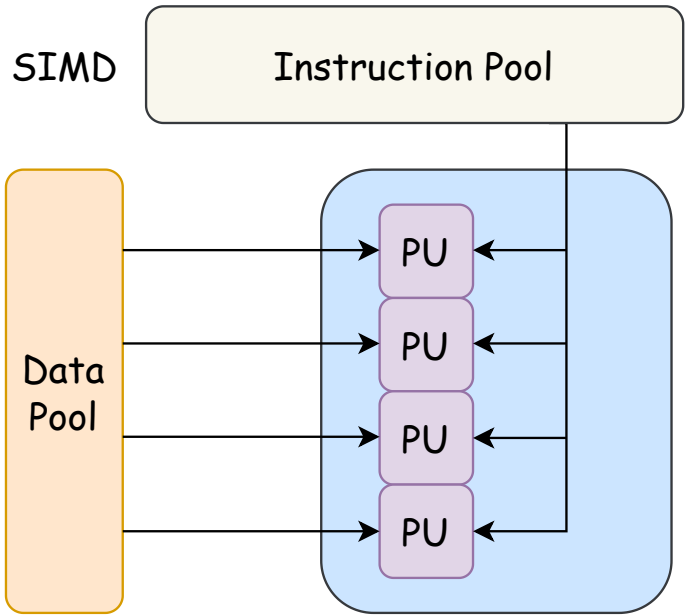
# 向量计算 VS 常量计算




- 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

- 向量化的本质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执行相同操作，实现空间上的并行
- 单指令流：同时只能执行一种操作
- 多数据流：在一组同构（向量）的数据上进行操作



# 硬件支持

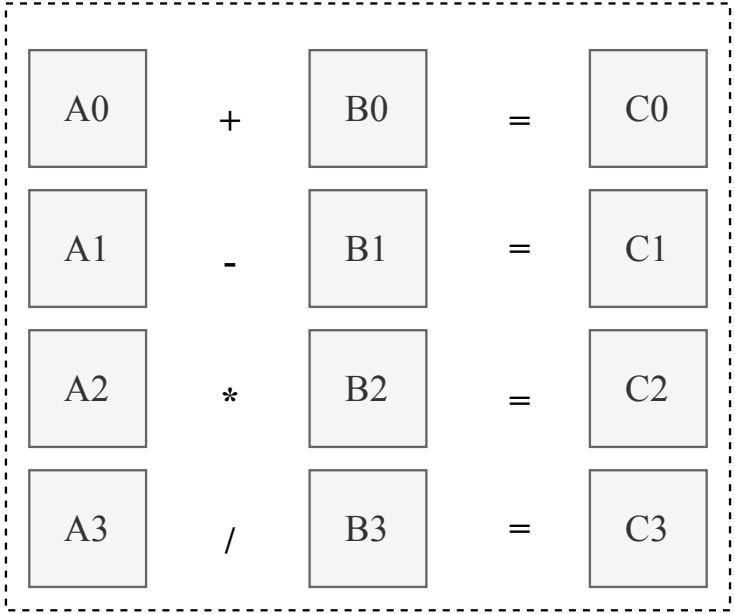
- Intel CPU提供了一系列SSE，AVX扩展向量化指令集

$ cat /proc/cpuinfo

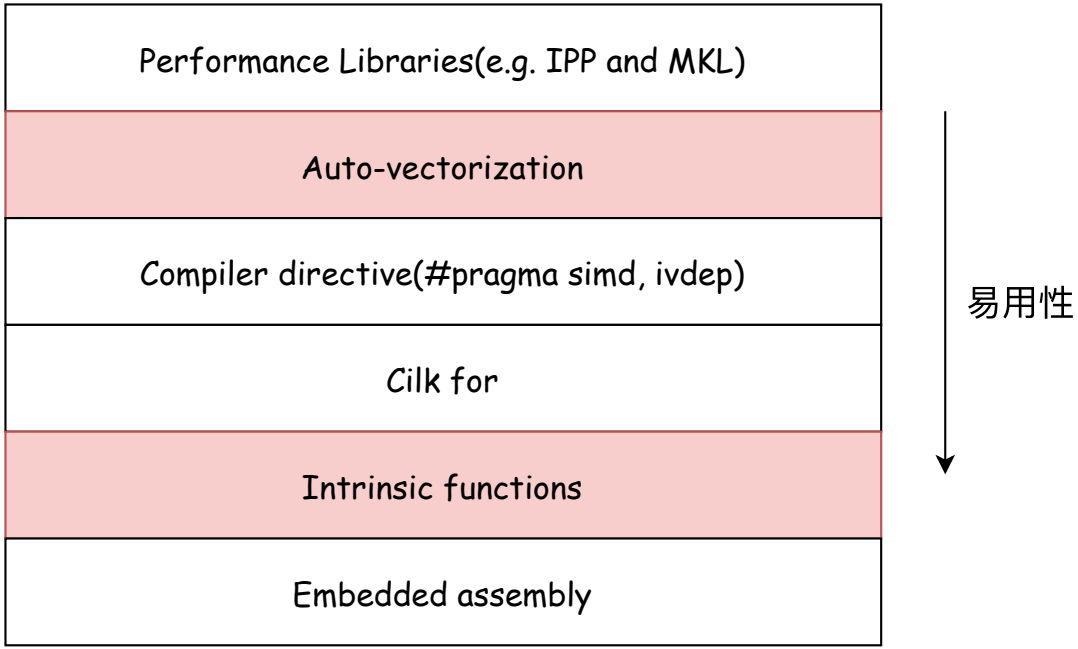
```
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx
pdpe1gb rdtscp lm constant_tsc rep_good nopl eagerfpu pni pclmulqdq
ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer
aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch
fsgsbase bmi1 hle avx2 smep bmi2 erms invpcid rtm mpx avx512f avx512dq
rdseed adx smap clflushopt clwb avx512cd avx512bw avx512vl xsaveopt xsavec
xgetbv1 arat
```

# SIMD缺陷

- 不适用于严重依赖控制流的任务，即有大量分支、跳转和条件判断的语句
- 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



# 如何实现向量化




- 自动向量化和Intrinsic函数在通用性和易用性方面更强

# Intrinsic函数实现向量化

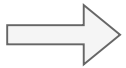
- 使用SSE _mm_add_ps intrinsic函数，一次实现8个单精度浮点数的加法

```
int main()
{
    __m128 v0 = _mm_set_ps(1.0f, 2.0f, 3.0f, 4.0f);
    __m128 v1 = _mm_set_ps(1.0f, 2.0f, 3.0f, 4.0f);

    __m128 v3 = _mm_add_ps(v0, v1);

     float result[4];
     _mm_storeu_ps(result, v3)
}
```

# 编译器自动向量化

- clang++ -O3，自动使用向量化指令

```
int arraySum(int * array, int length)
{
    int sum = 0;
    for(int i = 0; i < length; ++i)
    {
        sum += array[i];
    }
    return sum;
}
```



```
movdqu(%rdi,%rsi,4),%xmm2
paddd  %xmm0,%xmm2
movdqu 0x10(%rdi,%rsi,4),%xmm0
paddd  %xmm1,%xmm0
movdqu 0x20(%rdi,%rsi,4),%xmm1
movdqu 0x30(%rdi,%rsi,4),%xmm3
movdqu 0x40(%rdi,%rsi,4),%xmm4
paddd  %xmm1,%xmm4
paddd  %xmm2,%xmm4
movdqu 0x50(%rdi,%rsi,4),%xmm2
paddd  %xmm3,%xmm2
paddd  %xmm0,%xmm2
movdqu 0x60(%rdi,%rsi,4),%xmm0
paddd  %xmm4,%xmm0
movdqu 0x70(%rdi,%rsi,4),%xmm1
paddd  %xmm2,%xmm1
```

# ClickHouse向量化

- Intrinsic函数
  - 大量使用intrinsic函数对关键路径代码进行优化

- 编译器自动向量化
  - 通过良好的架构设计和代码设计，使得编译器能够生成良好的向量化代码
  - 关键：基于Pipeline的执行引擎设计，能够按列对数据进行处理

# ClickHouse中的intrinsic函数

- https://github.com/ClickHouse/ClickHouse/issues/37005

memcpySmall.h memcmpSmall.h

ColumnsCommon.h/cpp AggragationCommon.h ColumnsHashing.h

IPv6ToBinary.cpp remapExecutable.cpp StringSearcher.h

FunctionsHashing.h randomString.h randomFixedString.h

TargetSpecific.cpp generateUUIDv4.cpp greateCircleDistance.cpp

UTF8Helpers.h/cpp divice.cpp divideImpl.cpp FunctionRando.h/cpp.

FunctionsRound.h isValidUTF8.cpp LowerUpperImpl.h module.cpp

……

# Column Filter

- 将64字节压缩成64位的数，通过pop_count指令进行计算

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

#if defined(__SSE2__) && defined(__POPCNT__)
    const Int8 * end_pos64 = pos + (end - start) / 64 * 64;

    for (; pos < end_pos64; pos += 64)
        count += __builtin_popcountll(toBits64(pos));
#endif

    for (; pos < end_pos; ++pos)
        count += *pos != 0;
    return count;
}
```

# Column Filter

```
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;
}
```

# ClickHouse-编译器自动向量化

- 关键：基于Pipeline的执行引擎
  - 以Block为单位，按列对数据进行处理

- 优化：
  - 代码设计中大量使用模板（template），对类型和长度进行分派
    - ✓ 消除分支跳转语句
  - 大量使用内联函数
    - ✓ 消除函数调用
  - 减少虚函数调用

# Plus函数实现

- c = a + b，执行时，类型长度已知，无分支跳转语句

```
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);
    }
 }
```

# apply函数

- 该函数被inline，process的for循环中不会发生函数调用

```
template <typename Result = ResultType>
static inline NO_SANITIZE_UNDEFINED Result apply(A a, B b)
{
    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;
}
```

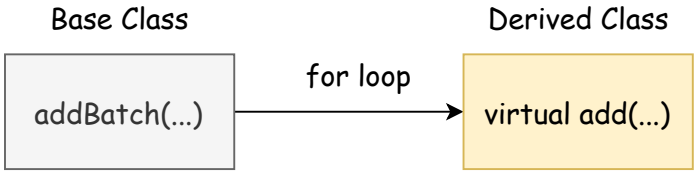
# __restrict关键字

```
template <OpCase op_case>
static void NO_INLINE process(const A * __restrict a, const B * __restrict b,
                              ResultType * __restrict c, size_t size)
```

- 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
- https://github.com/ClickHouse/ClickHouse/pull/9304
  - ➢ 通过该关键字，使整体查询性能提升5%~200%

# 消除虚函数调用

- 消除虚函数调用也是提高向量化的一个重要手段
- ClickHouse聚合函数计算：
  - 接口类addBatch通过for循环调用派生类的add方法将一批数据聚合到中间状态，for循环无法向量化




  - https://github.com/ClickHouse/ClickHouse/pull/17043
  - 该PR通过在addBatch消除add方法的虚函数调用，从而实现countIf函数的向量化执行，性能提升7倍

# 03

# Pipeline设计与实现

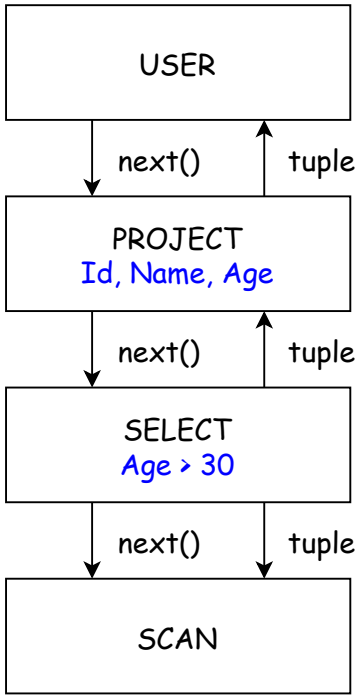
# 传统火山模型

- Tuple-at-a-time processing，无法发挥向量化能力

优点：处理逻辑清晰，简单

缺点：以行为单位处理数据，CPU Cache不友好，虚函数调用开销大，CPU利用率不高

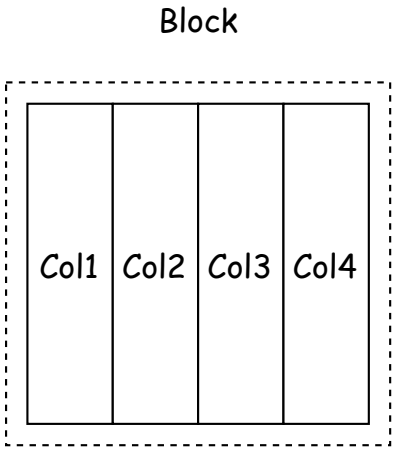
```
SELECT Id, Name, Age
FROM Perple
WHERE Age > 30
```

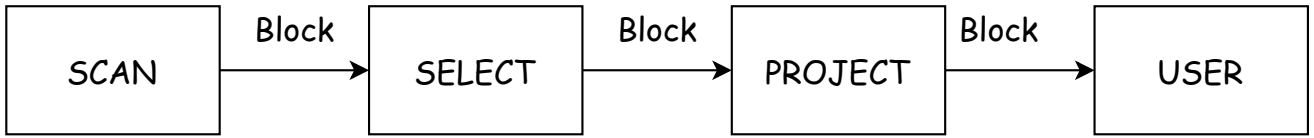


# Pipeline执行模型

- 以Block为单位，按列对数据进行处理，易于实现向量化

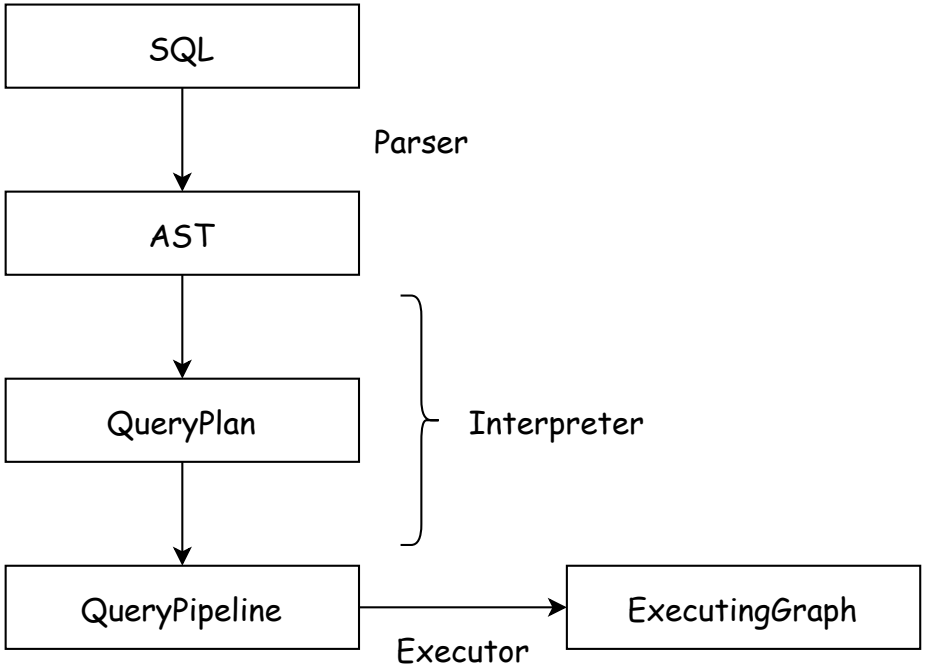
```
SELECT Id, Name, Age
FROM Perple
WHERE Age > 30
```



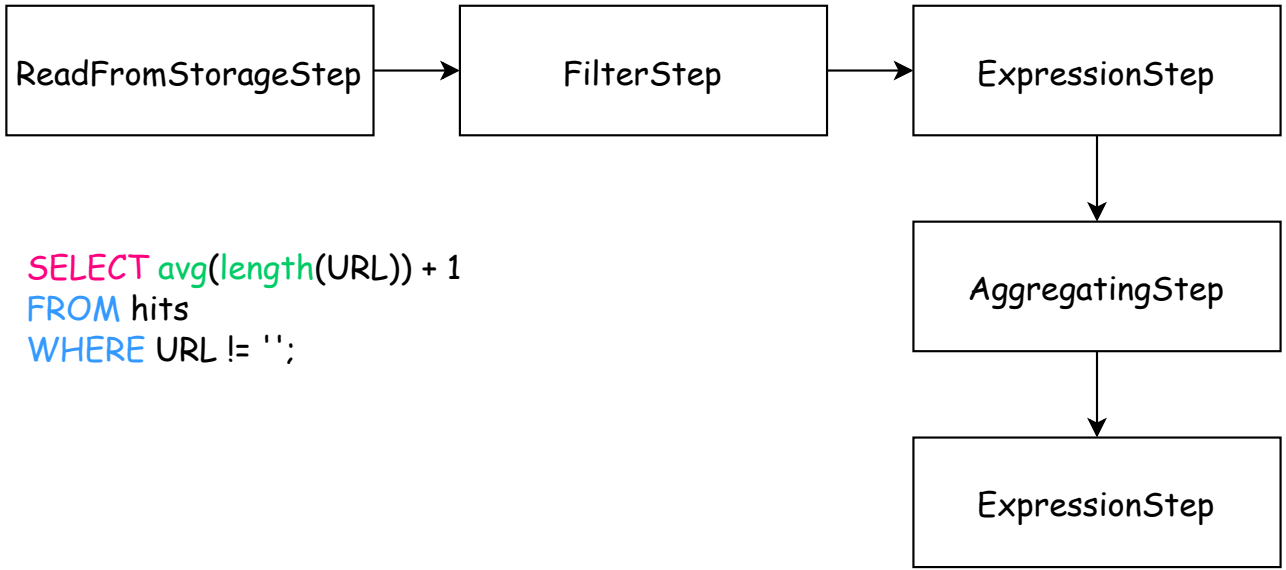
# ClickHouse SQL执行流程

- Executor：在ExecutingGraph上完成Pipeline的调度执行



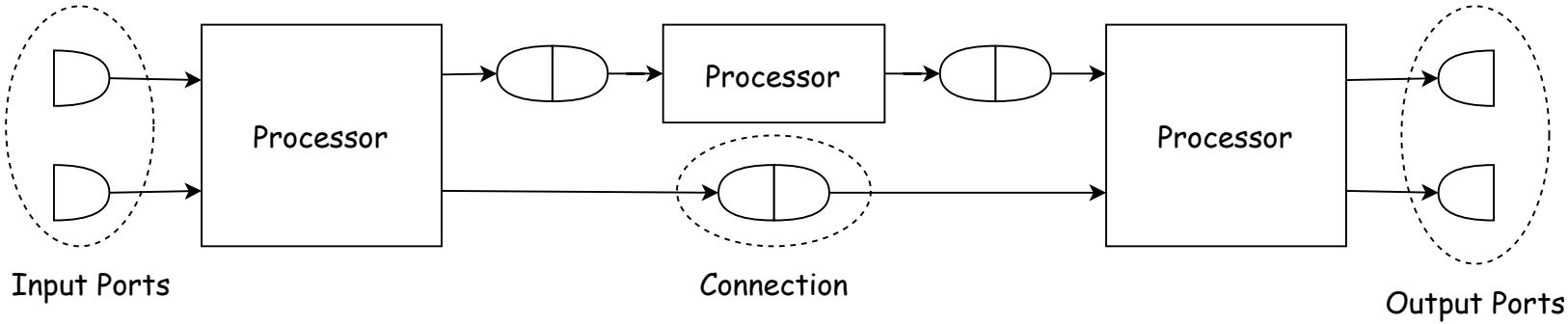
# QueryPlan

- 由QueryStep构成的Tree，除Join外，实际退化为链表
- 延迟Pipeline创建，在其上能够进行Pipeline级别的优化



# Pipeline基本结构

- 有向无环图
  - 节点：Processor
  - Port：输入或输出，能存储一个chunk的数据
  - 边：一对连通的Port

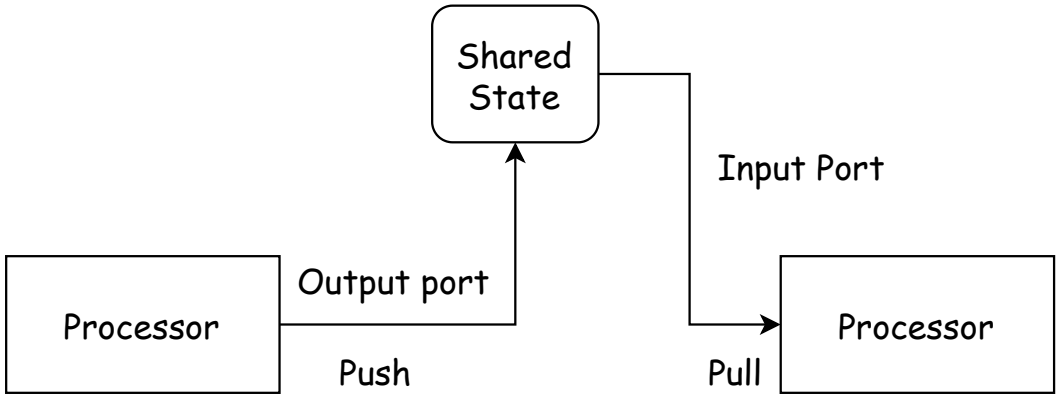


# Processors

- Pipeline的基本构建块，能够对数据进行处理
- 有0个或多个输入端口，0个或多个输出端口
  - Source：只有输出端口，没有输入端口
  - Sink：只有输入端口，没有输出端口
- 在Pipeline执行时，Processor会从输入端口拉取数据，对数据进行处理，然后推到输出端口

# Port

- 连接Processor，实现数据流通
- Output port：push数据到shared state
- Input port：从shared state pull数据



# Processor状态

- ClickHouse中定义了一系列Processor状态，通过状态迁移实现数据流动

States of Processors

NeedData

PortFull

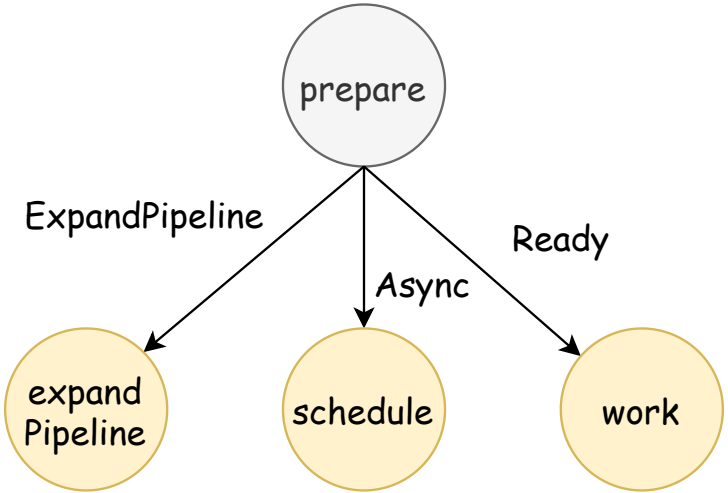
Finished

Ready

Async

ExpandPipeline

Methods of Processors



# Processors

- ClickHouse中实现了一系列Processors(Transform):

FilterTransform LimitTranfrom ExpressionTransform

AggregatingTransform CountingTransform DistinctSortedTransform

ExtremesTransform FinishSortingTransform

JoiningTransform LimitByTransform

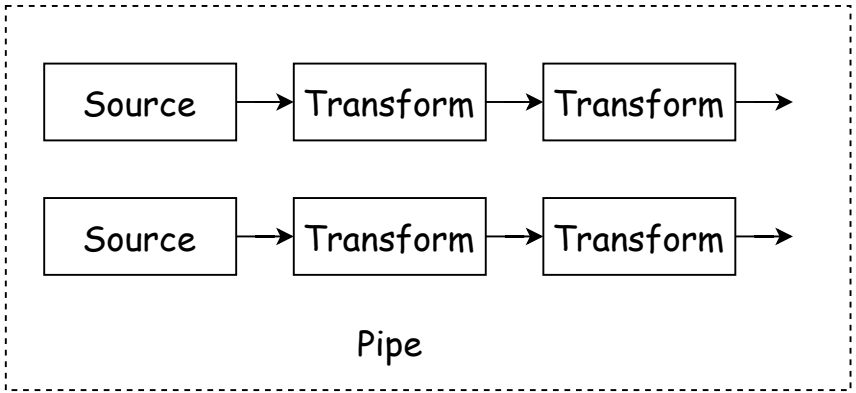
MergeSoringTransform SortingTransform RollupTransform

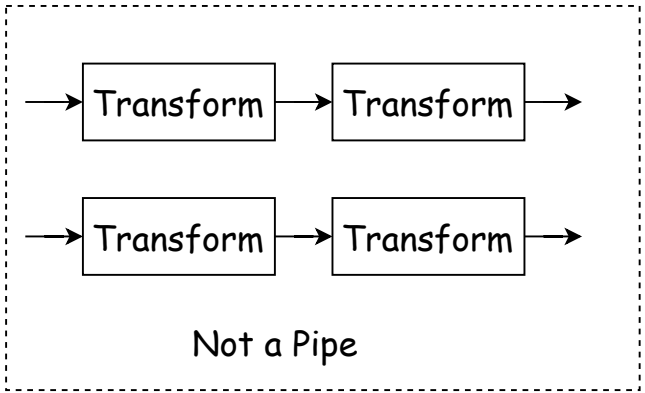
SquashingChunkTransform

……

# Pipe与Processor的区别

- Pipe：
  - Processors的集合，构成pipeline的一部分
  - 如果非空，必须从Source开始
  - Processor（Input Port）必须是连通的，可以有Sink
- 看ClickHouse Pipeline代码的人问的最多的问题

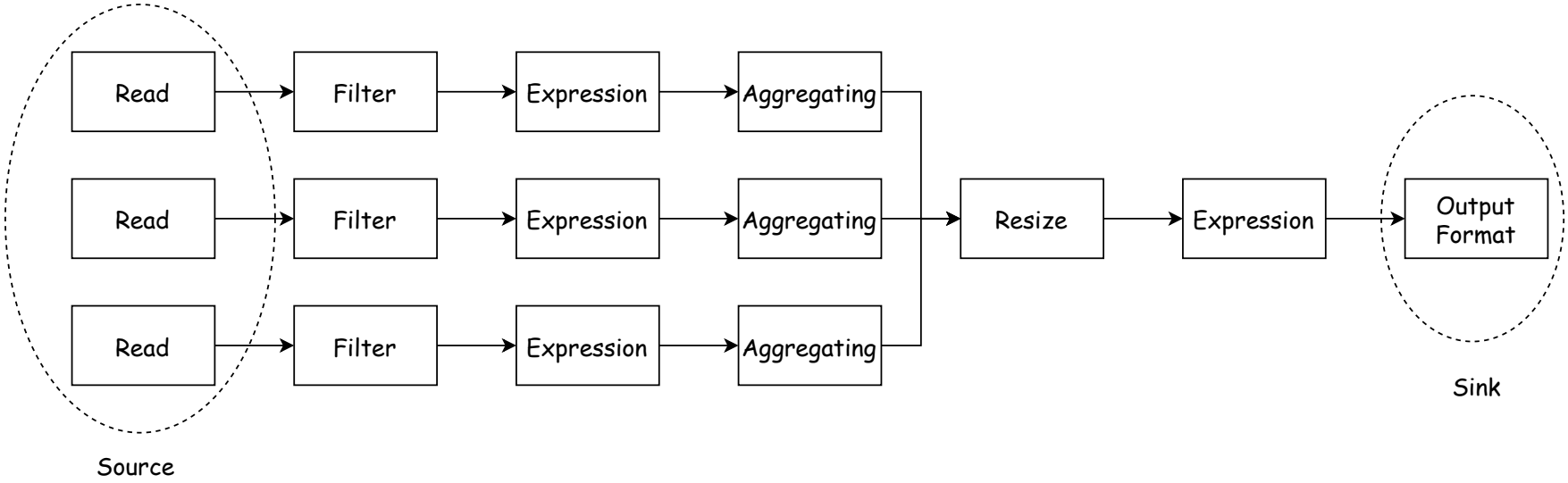


# Pipeline

- 从Source开始，到Sink结束
  - GROUP BY/ORDER BY之前，并行执行，之后只有一个流（缺点？）

```
SELECT avg(length(URL)) + 1 FROM hits WHERE URL != '';
```

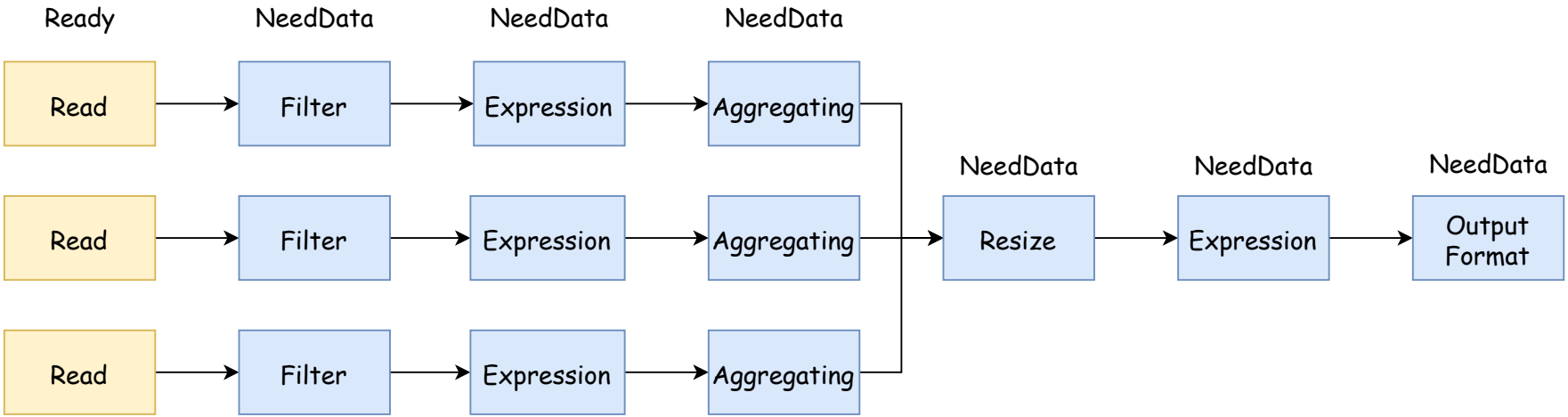


# Pipeline执行

- ExecutingGraph
  - 节点为Processor（引用）
  - 双向边
- 初始时，遍历Graph，找到没有children的节点，更新节点状态
- 维护两个全局队列，线程从队列中取task来执行，执行完更新相邻节点状态
  - Ready队列
  - Async队列

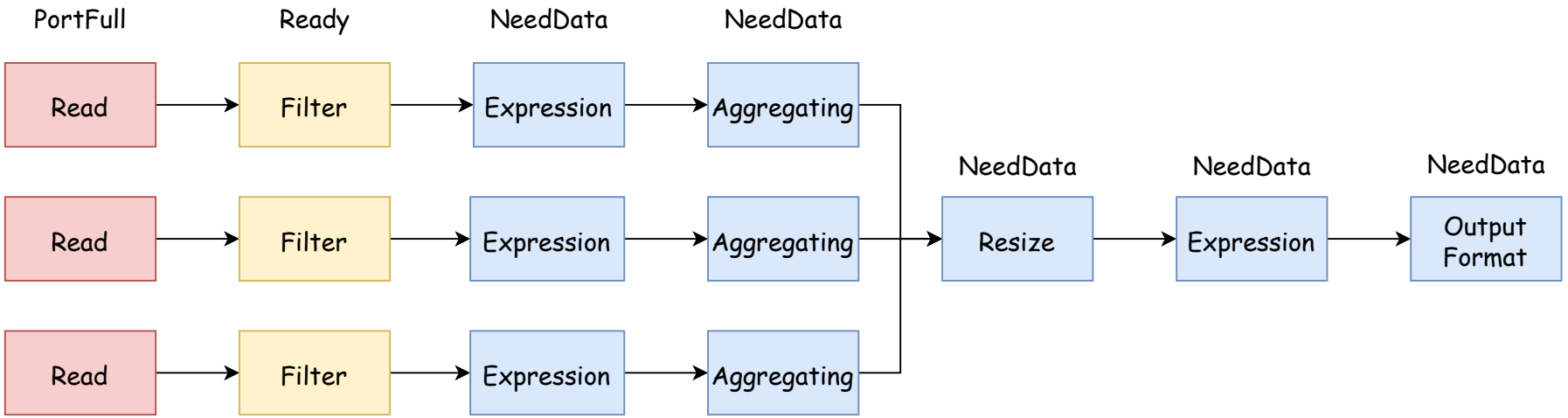
# Pipeline执行

- 从OutputFormat开始执行prepare更新状态



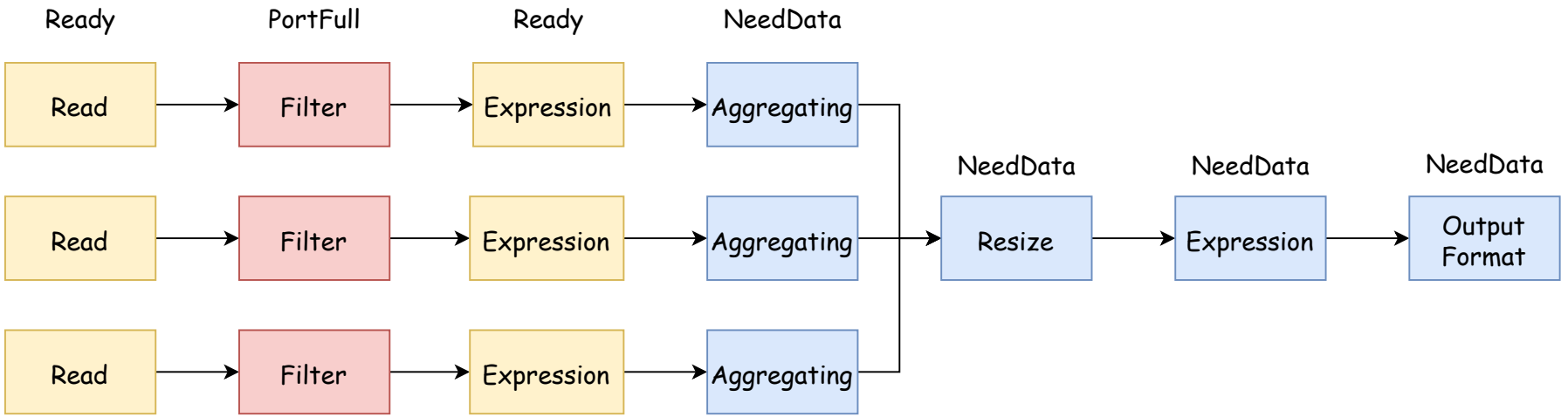
# Pipeline执行

- Read执行完之后，prepare把数据push到Output Port，Filter prepare拉取数据，进入Ready状态



# Pipeline执行

- Filter执行之后，prepare进入PortFull状态，Read和Expression执行prepare进行Ready状态



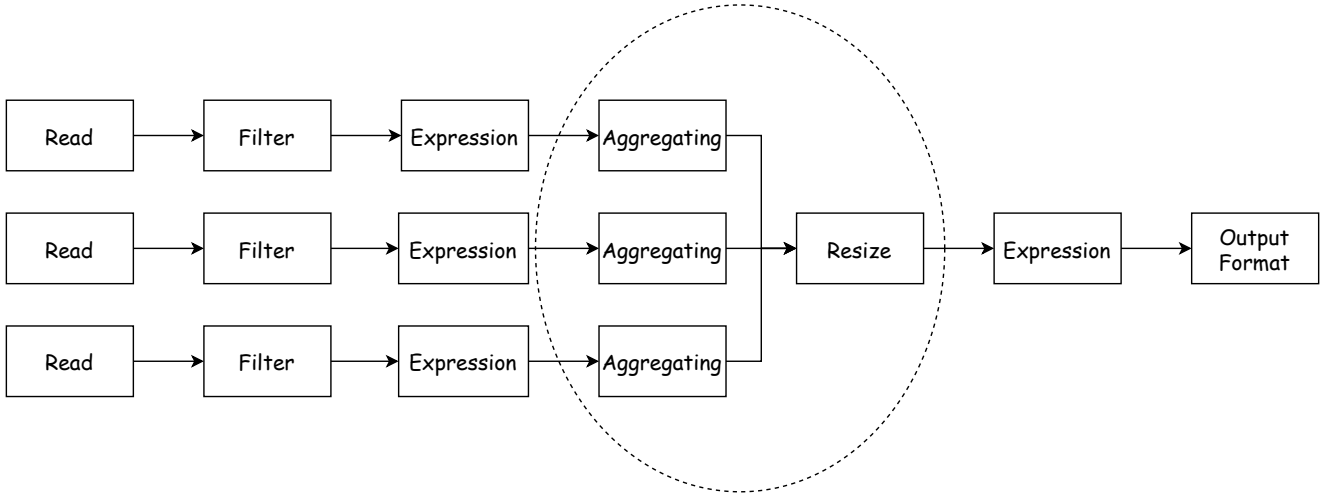
# Pipeline执行

- Pipeline执行过程中，通过Processor状态迁移使得数据在Processor之间流水起来，直到所有Processor进入Finished状态，执行完成

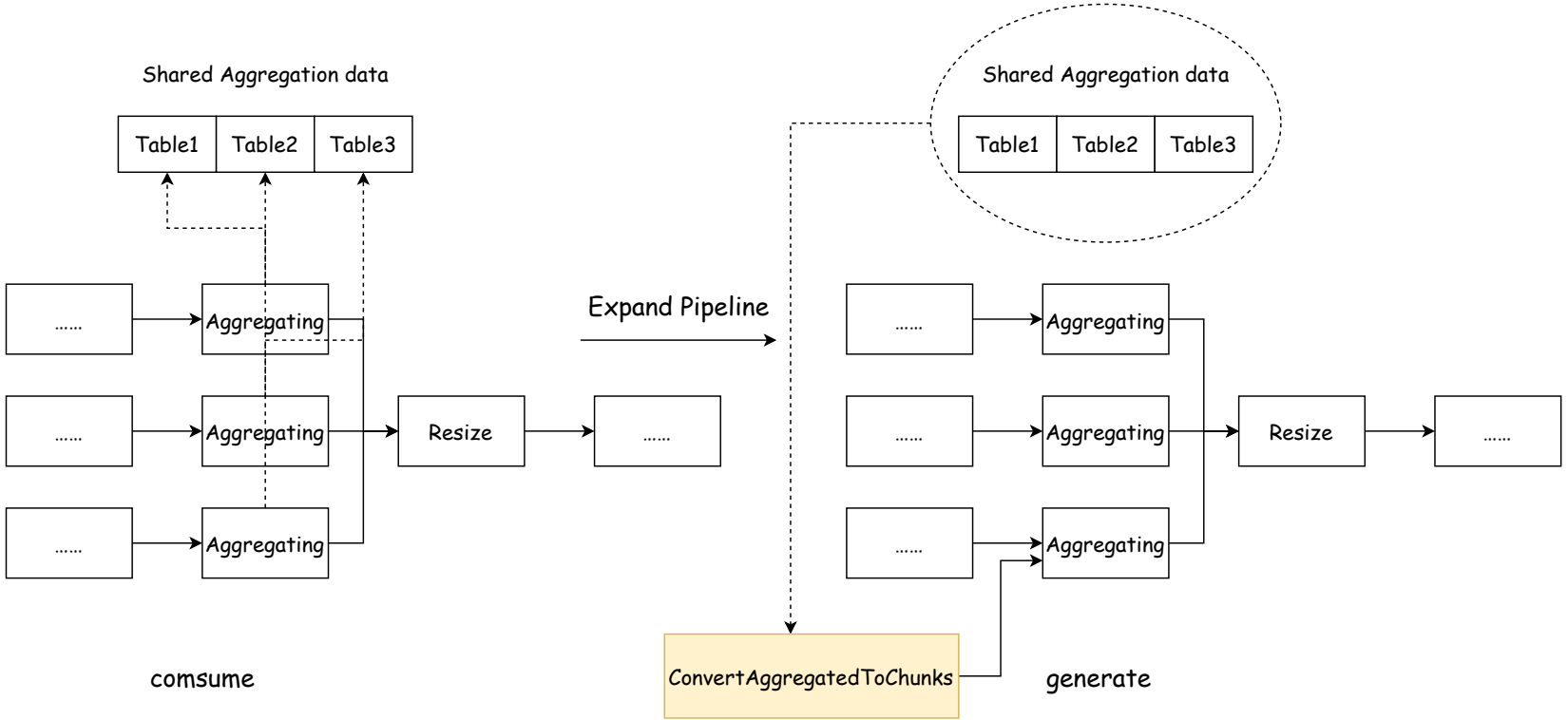
- prepare, work, schedule的执行是无锁的
- 一些有趣的问题：
  - 同一Processor是否可以并行执行prepare，work，schedule？
  - 不同Processor是否可以并行执行prepare？
  - 不同Processor是否可以并行执行work？

# Pipeline动态更改

- Pipeline执行过程中有时需动态更改，创建新的Processor
- Aggregating：两阶段执行
  - comsume阶段：预聚合（并行）
  - generate阶段：Merge预聚合数据生成最终结果



# Aggregating执行



# Expression计算

- ClickHouse通过ExpressionActions来实现表达式计算
- 表达式分析 -> ActionsDAG(由表达式构成的DAG)
- 节点类型：

```
enum class ActionType
{
    // Column which must be in input
    INPUT;
    // Constant column with known value
    COLUMN;
    // Alias of a column
    ALIAS
    // arrayJoin function
    ARRAY_JOIN
    FUNCTION
}
```

# ActionsDAG

- Node实际描述的是一个列的计算
- 有向无环图方便表达Expression之间的依赖关系
- 基于DAG，方便对Action进行优化
  - 删除不需要的表达式
  - 子表达式编译
  - 节点拆分或合并 (QueryPlan)

# ActionsDAG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)
```

# Expression执行

- 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，实现按列进行计算

# Pipeline与向量化

- Pipeline执行时，数据以Block的形式在Processor之间流动，Processor能够按列对数据进行处理
- Expression基于ActionsDAG实现表达式执行的按列计算

Pipeline实现了数据以Block为单位按列执行，
是ClickHouse向量化执行的核心

# 04

# 总结

# 如何提高向量化

- Writing better code
  - 系统的架构设计、代码设计是影响向量化能力的重要因素
    - 传统火山模型 VS Pipeline执行模型
  - 无法面向指令编程

- Finding out critical path
  - 针对影响性能的关键代码路径，可以在后期通过Intrinsic函数手动向量化

# 非常感谢您的观看

DataFun.

DataFun.