向量化执行引擎概述

向量化论文参考

15-721

向量化执行引擎概念

向量化执行引擎是一种数据库查询处理技术，旨在通过同时处理一组数据（即向量）来提高查询性能。这种方法不同于传统的行式处理，后者通常逐行处理数据。向量化执行引擎利用现代硬件（如CPU和GPU）的并行计算能力，能够更高效地执行复杂的查询。

为什么需要向量化执行引擎：

1. **性能提升**：通过并行处理多个数据项，向量化执行引擎可以显著减少处理时间，尤其在处理大规模数据时效果尤为明显。
2. **更好的利用硬件资源**：现代处理器通常具有多个核心和SIMD（单指令多数据）指令集，向量化能够更好地利用这些硬件特性。
3. **减少数据移动**：向量化方法可以在内存中更高效地访问和处理数据，减少因频繁访问存储而造成的延迟。
4. **简化查询执行计划**：通过向量化执行，查询的复杂性可以被简化，使得执行计划更高效。
5. **适应大数据处理需求**：随着大数据的兴起，传统的行式处理方法往往无法满足性能需求，向量化执行引擎应运而生，能够处理更大规模的数据集。

MonetDB/X100: Hyper-Pipelining Query Execution

- 感觉需要看看这篇论文，它是第一个提出向量化引擎概念的论文

PolarDB中的向量化执行引擎

应用场景

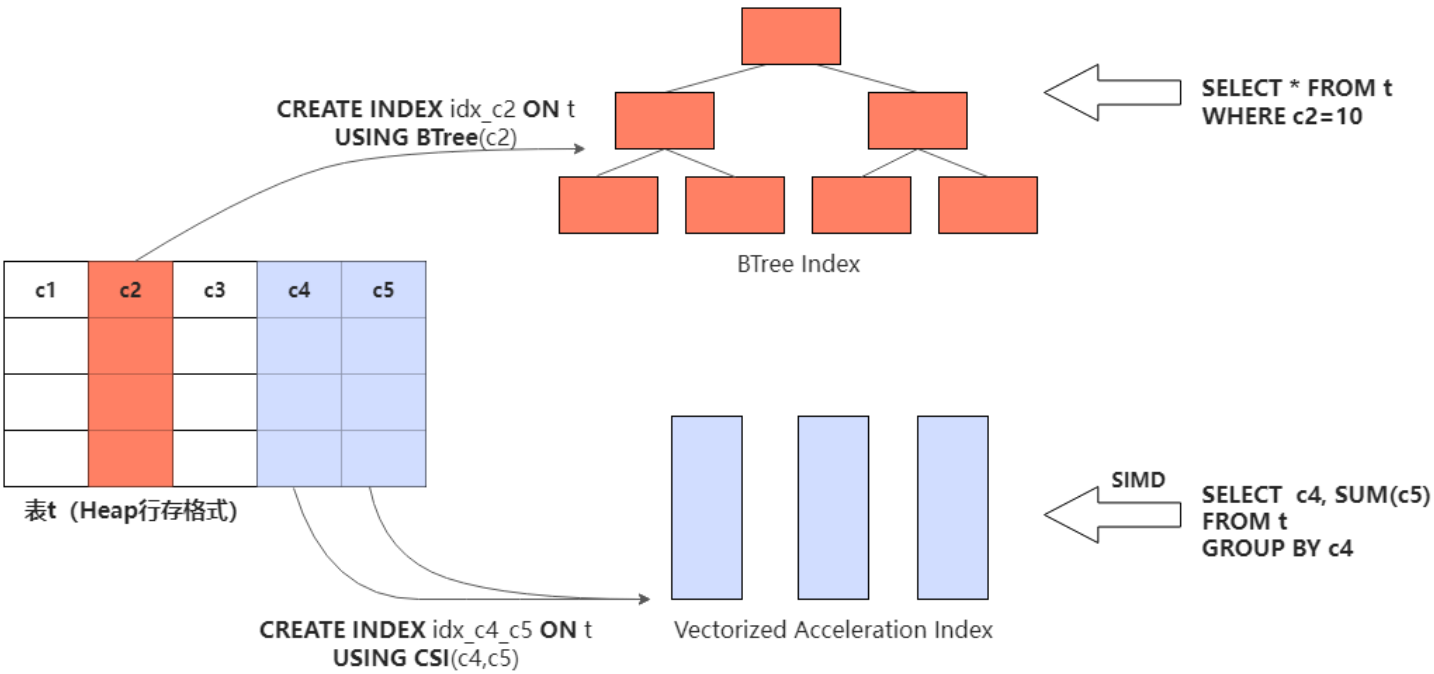
- 全表扫描或无法使用BTree索引的场景。例如对表中的列进行大量的 `Group By` 、 `Order By` 、 `Count` 、 `Sum` 、 `Filter` 、 `Join` 等耗时操作。
- JSON数据类型的查询与分析。例如在嵌套JSON中提取Key和Value，PostgreSQL传统的TOAST机制无法很好地满足性能需求的场景。
- 时空查询与分析场景。例如基于地理网格统计时空热力图等。

原理

PolarDB PostgreSQL版向量化引擎的核心工作集中在两个部分：

- 向量化版本的查询算子。如 `Scan`、`Group By`、`Order By`、`Hash Join`、`Filter`、`Aggregation` 等算子的向量化，使其能够接受批量化的数据输入并利用SIMD指令进行处理。
- 与向量化引擎匹配的加速索引（即向量化加速索引，Vectorized Acceleration Index）。PolarDB PostgreSQL版向量化引擎中，加速索引与BTree索引、GiST索引类似，但存储结构不同。一个表中可以同时创建向量化加速索引和其他类型的索引来应对不同的查询，PolarDB PostgreSQL版优化器会根据查询计划的代价来选择不同的索引。

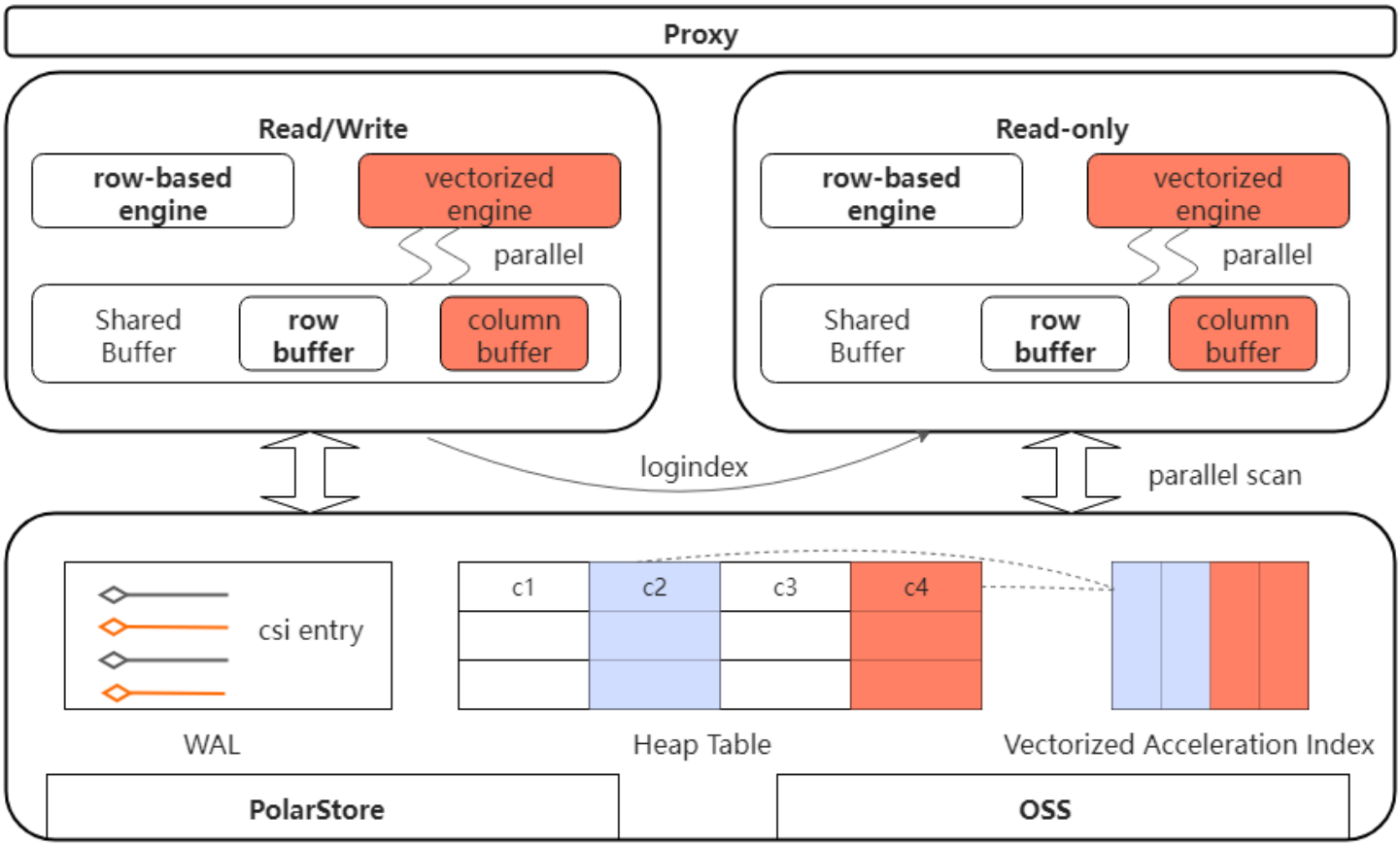
如下图所示，您可以为表t的c2列创建Btree索引来应对点查（`SELECT * FROM t WHERE c2=10`），为c4和c5列创建加速索引来应对统计类查询（`SELECT c4, SUM(c5) FROM t GROUP BY c4`），查询优化器会根据查询SQL的查询代价选择合适的索引。



PolarDB PostgreSQL版加速索引形式

PolarDB PostgreSQL版向量化引擎会部署在主节点以及每一个只读节点中，加速索引会存储在底层的共享存储中，每个节点上的向量化引擎都可以访问。

当通过集群地址来访问时，会根据各个节点的负载情况选择某个节点的向量化引擎来执行，也可以通过只读地址指定某个只读节点执行。



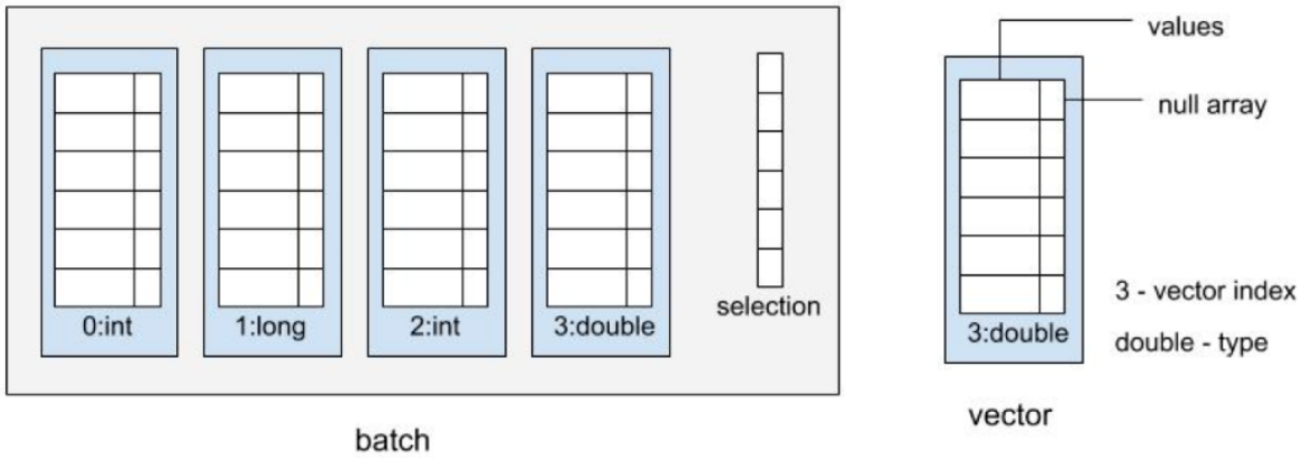
PolarDB PostgreSQL版向量化执行引擎产品形态

运行时数据结构

运行时结构

数据结构

在PolarDB-X向量化执行系统中，采用以下的数据结构来存放数据：



向量化表达式执行时，所有的数据都会存放在batch这一数据结构中。batch由许多向量（vector）和一个selection数组而组成。其中，向量vector包括一个存储特定类型的数值列表（values）和一个标识null值位置的null数组组成，它们在内存中都是连续存储的。null数组中的bit位以0和1来区分数值列表中的某个位置是否为空值。

我们可以用vector(type, index)来标识batch中一个向量。每个向量有其特定的下标位置(index)，来表示向量在batch中的顺序；类型信息（type）来指定向量的类型。在进行向量化表达式求值之前，我们需要遍历整个表达式树，根据每个表达式的操作数和返回值来分配好下标位置，最后根据下标位置统一为向量分配内存。

延迟物化

selection数组的设计体现了延迟物化的思想，参考论文《Materialization Strategies in a Column-Oriented DBMS》。所谓延迟物化，就是尽可能地将物化（matrialization）这一过程后推，减少内存访问带来的开销。在执行表达式计算时，往往会先经过Filter表达式过滤一部分数据，再对过滤后的数据执行求值处理；每次过滤都会影响到batch中所有的向量。以上图中的batch为例，如果我们针对第0个向量设置 vector(int, 0) != 1这一过滤条件，假设vector(int, 0)中有90%的数据满足该过滤条件（选择率selectivity = 0.9），那么我们需要将batch中所有向量90%的数据重新物化到另一块内存中。而如果我们只记录满足该过滤条件的位置，存入selection数组，我们就可以避免这一物化过程。相应的，以后每次向量化求值过程中，都需要参考此selection数组。

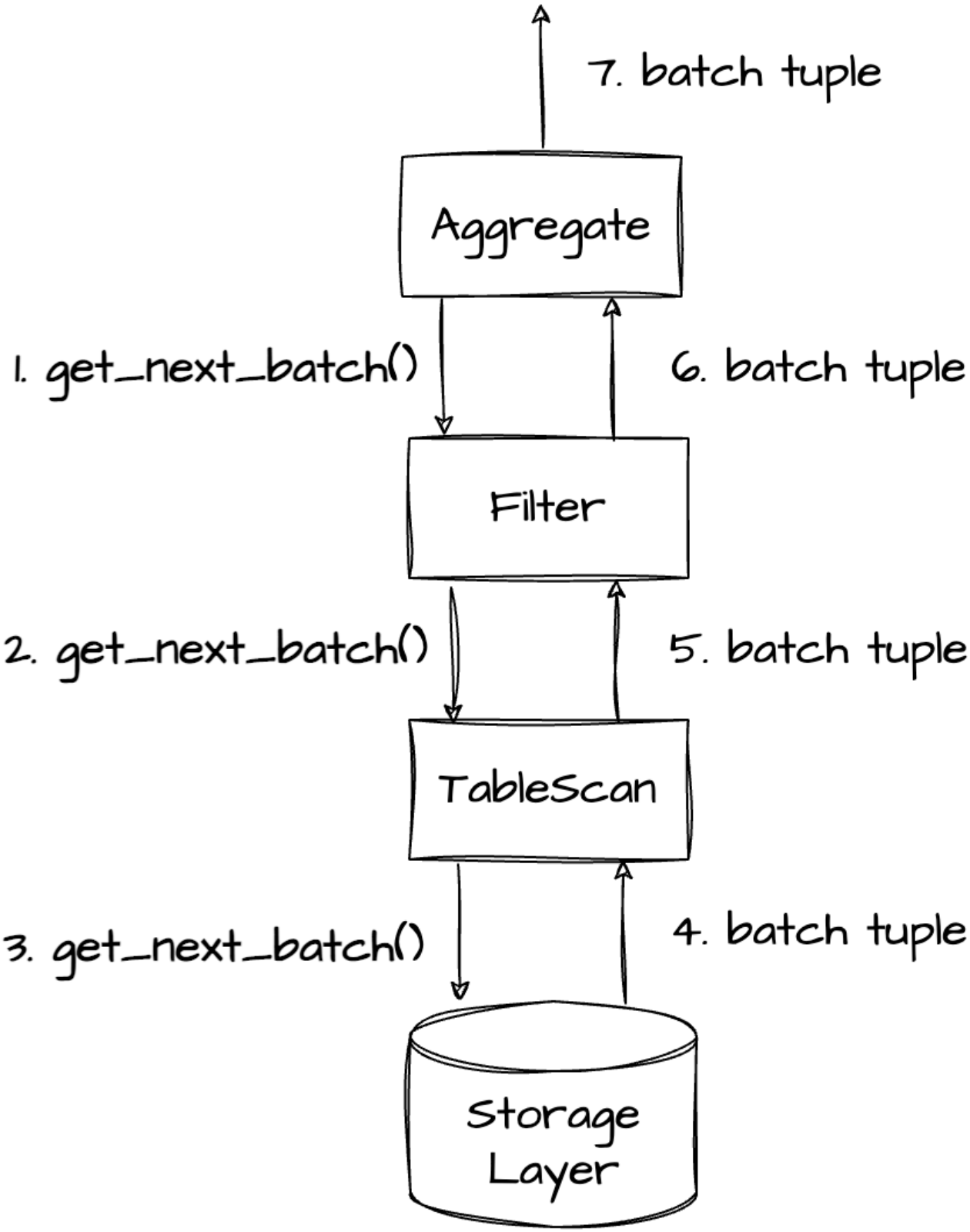
OB向量化执行引擎

博客参考

原理

向量化模型最早是在 MonetDB-X100（Vectorwise）系统的论文《MonetDB/X100: Hyper-Pipelining Query Execution》中提出的，不同于传统的火山模型按行迭代的方式，向量化引擎采用按批迭代方式，可以在算子间一次传递一批数据。向量化引擎被提出后，因为可以有效提高 CPU 利用率并充分发挥现代 CPU 的特性，很快就被广泛应用到了现代数据库引擎设计中。

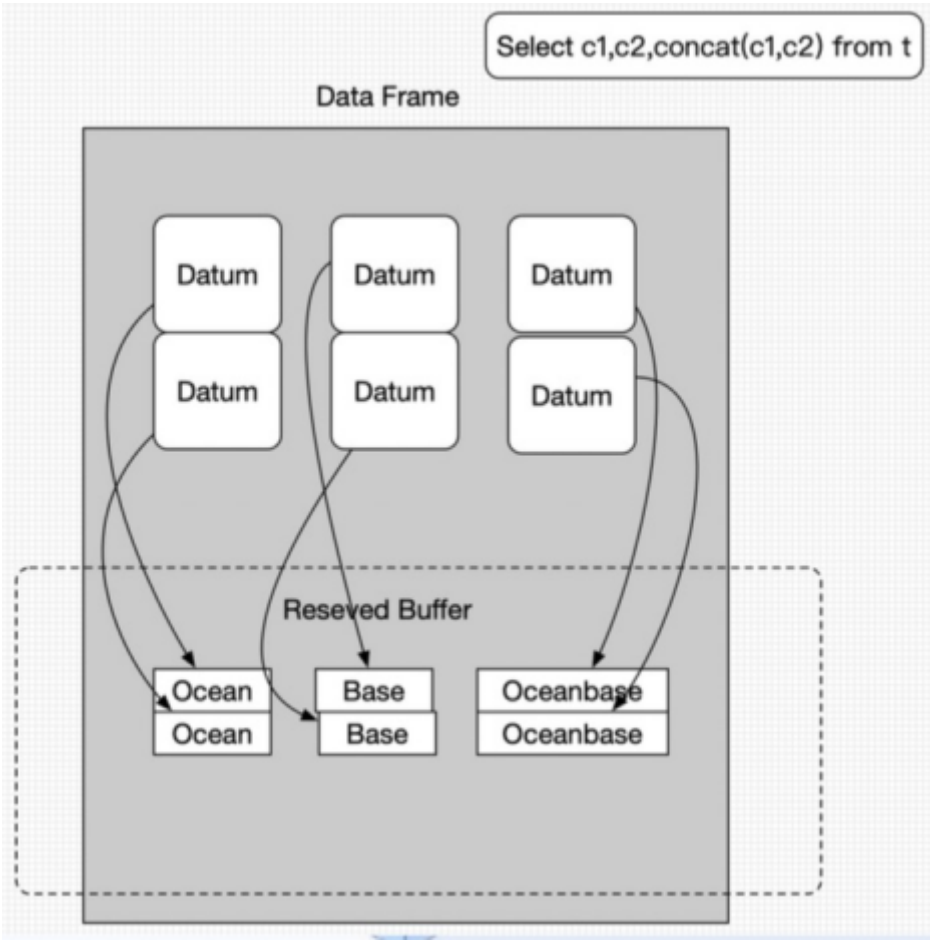
向量化模型也是通过 PULL 模式从算子树的根节点层层拉取数据。区别于 get_next_row() 调用一次传递一行数据，向量化引擎通过 get_next_batch() 一次传递一批数据，并尽量保证该批数据在内存上紧凑排列。



优势

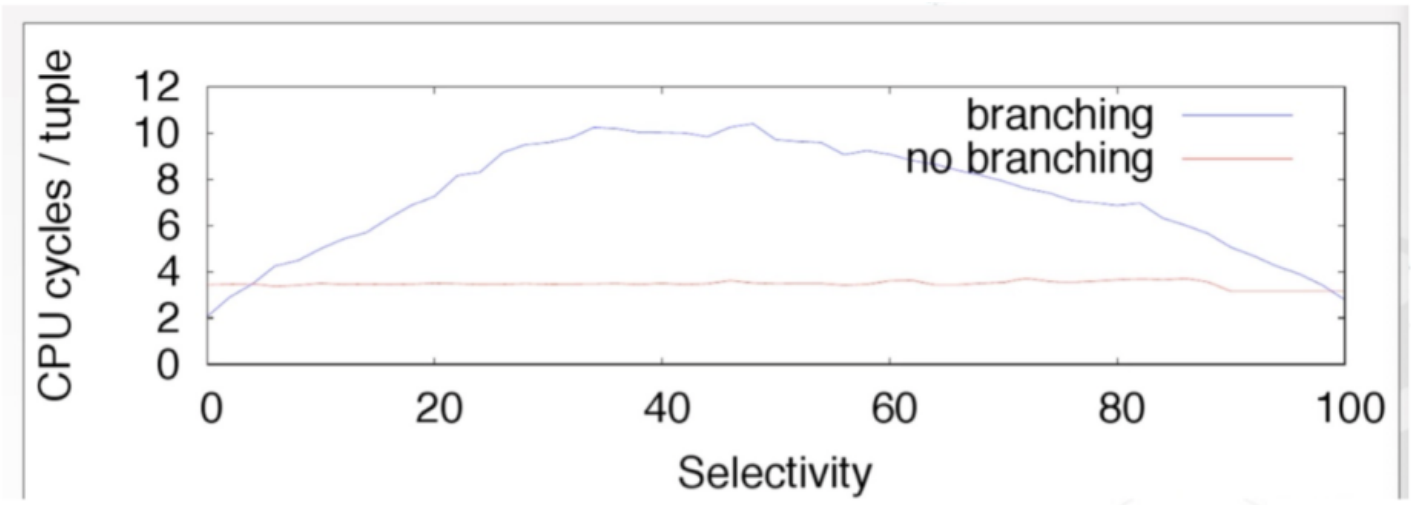
1. 减少虚函数的开销
2. 充分发挥现代CPU的特性，数据紧密排列

a. 由于中间数据连续，CPU 就可以通过预取指令快速把即将进行计算的数据提前加载到 L2 cache 中，以此减少 memory stall 的现象，提升 CPU 的利用率。在算子函数内部，函数也不再是一次处理一行数据，而通过批量处理连续数据的方式进行计算，可以提升 CPU DCache 和 ICache 的友好性，减少 Cache Miss。

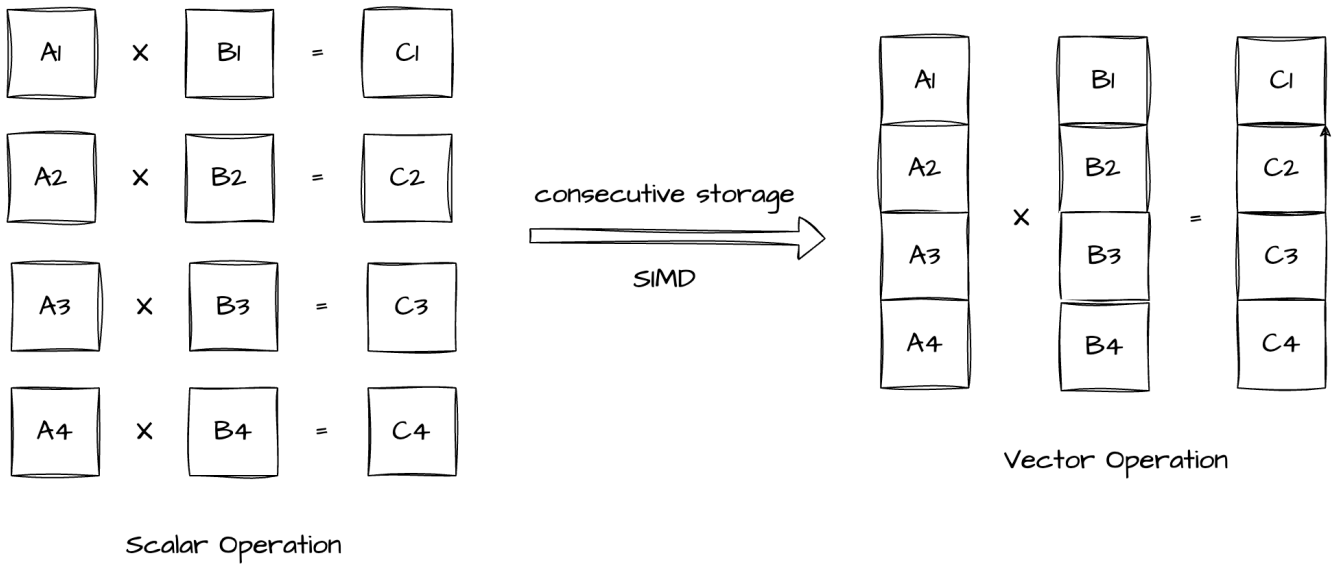


3. 减少分支预测失败对CPU流水线的破坏

a. 由于数据库 SQL 引擎逻辑十分复杂，在火山模型下条件判断逻辑往往不可避免。但向量引擎可以在算子内部最大限度地避免条件判断，**例如向量引擎可以通过默认覆盖写的操作，避免在 for 循环内部出现 if 判断，从而避免分支预测失败对 CPU 流水线的破坏，大幅提升 CPU 的处理能力。**



4. 使用SIMD指令加速计算

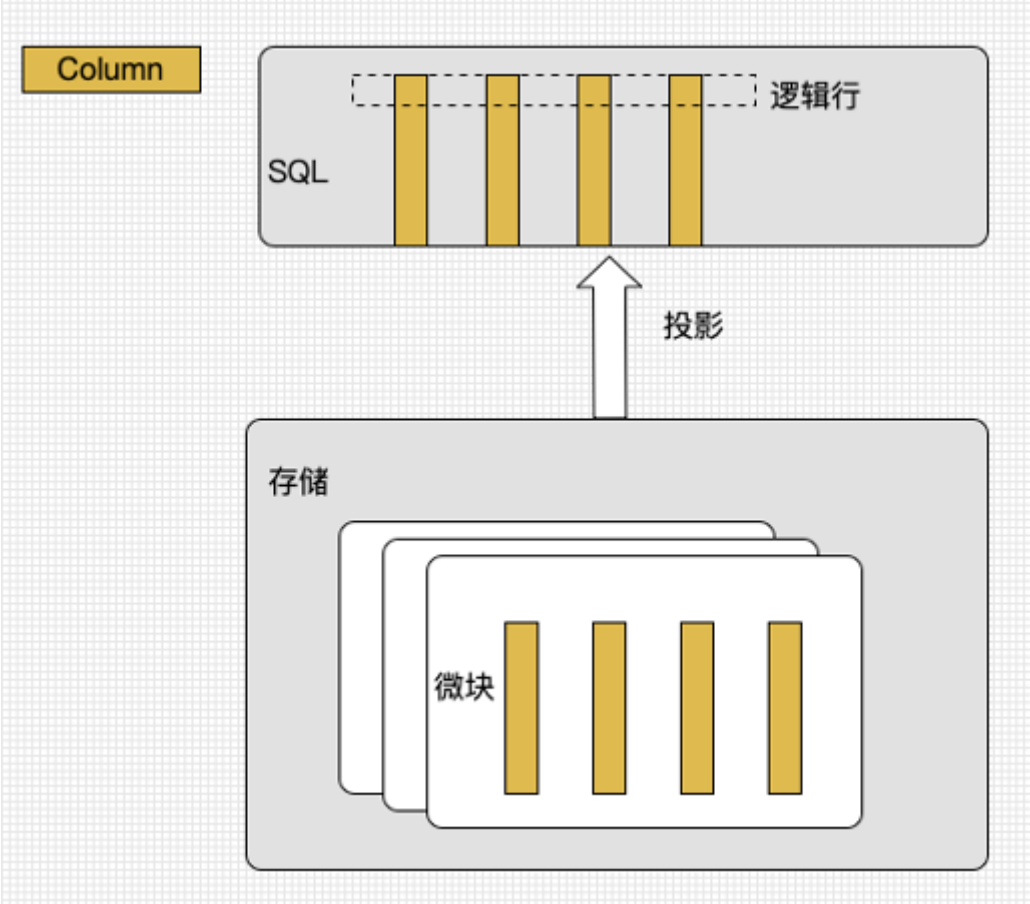


实现

博客参考

存储的向量化实现

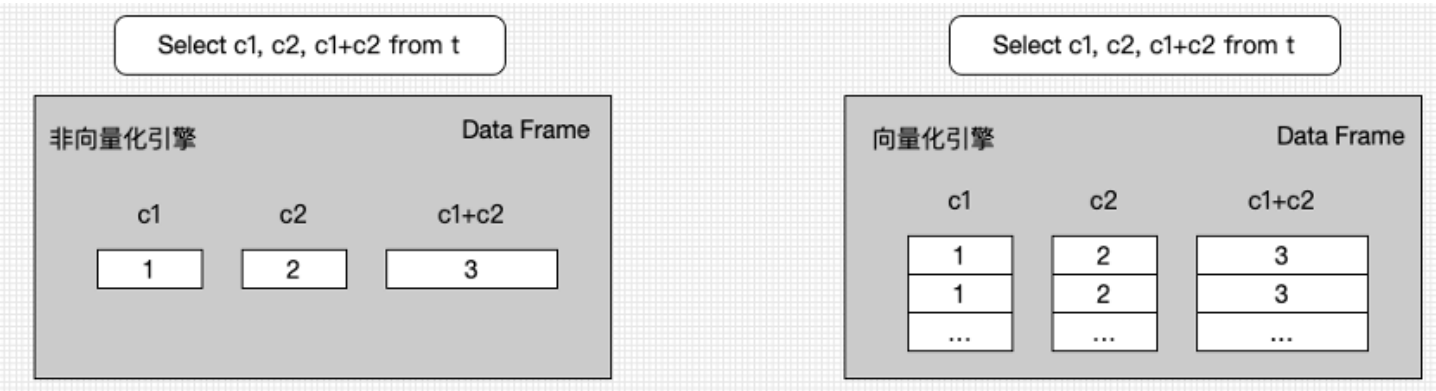
OceanBase 的存储系统的最小单元是微块，每个微块是一个默认 64KB（大小可调）的 IO 块。**在每个微块内部，数据按照列存放**。查询时，存储直接把微块上的数据按列批量投影到 SQL 引擎的内存上。由于数据紧密排列，有着较好的 cache 友好性，同时投影过程都可以使用 SIMD 指令进行加速。由于向量化引擎内部不再维护物理行的概念，和存储格式十分契合，数据处理也更加简单高效。整个存储的投影逻辑如下图：



SQL向量引擎的数据组织

内存编排

SQL 引擎的向量化先从的数据组织和内存编排说起。在 SQL 引擎内部，所有数据都被存放在表达式上，表达式的内存通过 Data Frame 管理。**Data Frame 是一块连续内存（大小不超过 2MB），负责存放参与 SQL 查询的所有表达式的数据。SQL 引擎从 Data Frame 上分配所需内存，内存编排如图 6。**



在非向量化引擎下，一个表达式一次只能处理一个数据（Cell）（图 6 左）。向量化引擎下，每个表达式不再存储一个 Cell 数据，而是存放一组 Cell 数据，Cell 数据紧密排列（图 6 右）。这样表达式的计算都从单行计算变成了批量计算，对 CPU 的 cache 更友好，数据紧密排列也非常方便的使用 SIMD 指令进行计算加速。另外**每个表达式分配 Cell 的个数即向量大小**，根据 CPU Level2 Cache 大小和 SQL 中表达式的多少动态调整。**调整的原则是尽量保证参与计算的 Cell 都能存在 CPU 的level2 cache 上，减少 memory stalling 对性能的影响。**

过滤标识

向量引擎的过滤标识也需要重新设计。向量引擎一次返回一批数据，该批数据内有的数据被删除掉，有的数据需要输出。如何高效的标识需要输出的数据，是一个重要的工作。论文**《Filter Representation in Vectorized Query Execution》**中介绍了目前业界的两种常用方案：

- 通过 BitMap 标记删除行：创建 bitmap， bitmap 中 bit 个数和返回数据向量大小相同。当对应 bit 为 1 时，该列需要输出，bit 为 0 时，该列被标记删除；
- 通过额外数组 Select Vector 记录输出行。需要输出的行的下标存在 Select Vector 中。

OceanBase 采用 bitmap 方案描述数据过滤，**即每个算子都有一个 Bitmap ， filter 过滤掉的数据，通过 bitmap 标识删除。**使用 Bitmap 的一大优势是内存占用小，可以在查询算子过多或者查询向量 size 过大时，避免出现内存使用过多的情况。

另外当数据的选择率很低时，可能会出现 bitmap 标识的数据过于稀疏，性能不佳的情况。一些数据库通过增加整理方法，使数据稠密排列来避免上述情况。但我们在实践中发现， HTAP 场景下 SQL 执行往往会出现阻塞算子（Sort, Hash Join, Hash Group by）或 Transmit 跨机执行算子，**而这些算子本身具备数据整理让稠密输出的特点， 额外的数据整理反而会出现不必要的开销。因此 OceanBase 向量化引擎没有提供单独的方法改变 bitmap 数据排列。**

SQL引擎的算子实现

每个算子内部最大限度地按照 branchless 编码、内存预取、 SIMD 指令等指导原则进行工程化编码，并取得大幅性能收益。

Hash join

并行hash算法参考

Hash Join 通过 Hash 表的构建和探测，实现两张表 (R 表和 S 表)的 hash 查找。当 hash 表的大小超过 CPU 的 level2 cache 时， hash 表随机访问会引起 memory stall，大大影响执行效率。Cache 的优化是 Hash Join 实现的一个重要方向，Hash Join 的向量化实现重点考虑了 cache miss 对性能的影响。

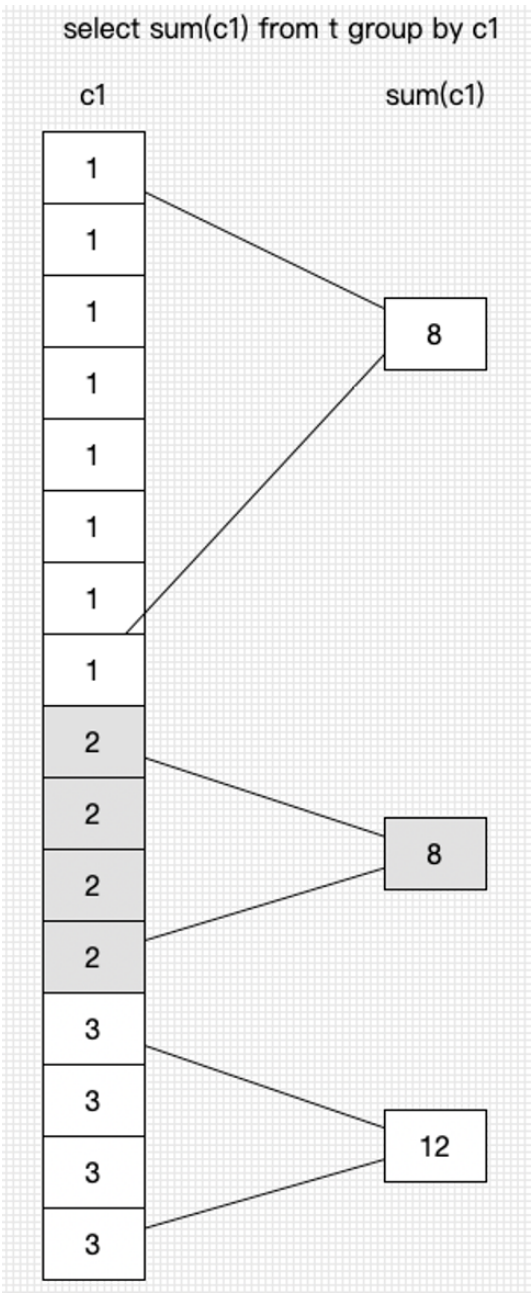
值得一提的是，OceanBase 的向量化 Hash Join 算子没有实现 Radix Hash Join 等 HashWare concious 的 Join 算法，而是通过向量计算 hash value 和内存 prefech 预取的方式避免 cache miss 和 memory stalling。

Radix Hash Join 可以有效降低 cache 和 TLB 的 miss rate，但是它需要两次扫描 R 表数据，并引入了创建直方图信息和额外的物化代价。OceanBase 的向量化 Hash Join 实现更为简洁，先通过 partition 分区，构建 hash 表。在探测 hash 表阶段，首先通过批量计算的方式，得到向量数据的 hash 值。然后通过 prefetch 预取，把该批数据对应的 hash bucket 的数据装载到 CPU 的 cache 中。最后按照 join 连接条件比较结果。通过控制向量的大小，保证预取的一批数据可以装载到 CPU 的 level 2 cache 中，从而最大程度的避免数据比较时的 cache miss 和 memory stalling，进而提升 CPU 的利用率。

Sort merge Group by

Sort Merge Group By 是一个常见的聚合操作。Sort Merge Group By 要求数据有序排列，group by 算子通过比较数据是否相同找到分组边界，然后计算相同分组内的数据。例如下图 c1 列数据有序排列，在火山模型下，由于一次只能迭代一行数据对于分组 1 需要比较 8 次，sum(c1) 也需要累加 8 次才能得到计算结果。在向量化引擎中，我们可以把比较和聚合分开计算，即先比较 8 次，找到分组1的所有数据个数(8)。由于分组内数据相同，针对 sum/count 等聚合计算还可以做进一步优化，例如 sum(c1) 可以直接通过 1 * 8，把 8 次累加变成 1 次乘法。count 则可以直接加 8 即可。

另外向量化实现还可以通过引入二分等方法，实现算法加速。例如下图向量的大小是16，通过二分的方法，第一次推进行的 step 大小为 8，即比较 c1 列的第 0 行和第 7 行数据。数据相等，则直接对 c1 列的前 8 个数据求和。第二次推进的 step 大小为 8，比较第 7 行和第 15 行数据，数据不相等，回退 4 行再比较数据是否相同，直到找到分组边界。然后再通过二分进行进行下一个分组的查找。通过二分的比较方式，可以在重复数据较多的场景下跳过重复数据的比较，实现计算的加速。当然该方案在数据重复数据较少的场景下，存在 bad case。我们可以通过数据 NDV 等统计信息，在执行期决定是否开启二分比较。



数据库查询计划的pipeline

NUMA-Aware 执行引擎论文参考博客

DBMS在现代处理器上可能的延时问题

DBMSs On A Modern Processor: Where Does Time Go?

得出结论：

1. 一半的执行时间实际是在停顿（stall）。
2. 90%的停顿是因为L1指令cache miss与L2数据cache miss导致的，而L2指令cache miss与L1数据cache miss则不重要。
3. 20%的停顿是因为实现细节（如分支预测失败等）。
4. 内存访问延时对性能的影响大于内存带宽的影响。
5. 更快的CPU与缓存、内存的延时差距越来越大，以上因素会更加明显。

> 这篇文章是1999年的，机器是P II，在当前（2021年）的系统中本文的结论仍然有效，且可能有更大的影响。

资源不可用是因为：

1. 指令相互依赖，指令并行度低；
2. 大量指令同时争抢功能单元。

X86下机器指令会被翻译为微指令再执行，因此编译器没有简单的手段来发现跨机器指令的依赖关系并在微指令层面上优化。

编译查询

编译查询综述