



下载APP



21 | 查询执行引擎：如何让聚合计算加速？

2020-09-25 王磊

分布式数据库30讲

[进入课程 >](#)**讲述：王磊**

时长 16:42 大小 15.30M



你好，我是王磊。

在 19、20 两讲中，我已经介绍了计算引擎在海量数据查询下的一些优化策略，包括计算下推和更复杂的并行执行框架。这些策略对应了从查询请求输入到查询计划这个阶段的工作。那么，整体查询任务的下一个阶段就是查询计划的执行，承担这部分工作的组件一般称为查询执行引擎。

单从架构层面看，查询执行引擎与分布式架构无关，但是由于分布式数据库要面对海量数据，所以对提升查询性能相比单体数据库有更强烈的诉求，更关注这部分的优化。



你是不是碰到过这样的情况，对宽口径数据做聚合计算时，系统要等待很长时间才能给出结果。那是因为这种情况涉及大量数据参与，常常会碰到查询执行引擎的短板。你肯定想

知道，有优化办法吗？

当然是有的。查询执行引擎是否高效与其采用的模型有直接关系，模型主要有三种：火山模型、向量化模型和代码生成。你碰到的情况很可能是没用对模型。

火山模型


火山模型 (Volcano Model) 也称为迭代模型 (Iterator Model)，是最著名的查询执行模型，早在 1990 年就在论文 “[Volcano, an Extensible and Parallel Query Evaluation System](#)” 中被提出。主流的 OLTP 数据库 Oracle、MySQL 都采用了这种模型。

在火山模型中，一个查询计划会被分解为多个代数运算符 (Operator)。每个 Operator 就是一个迭代器，都要实现一个 next() 接口，通常包括三个步骤：

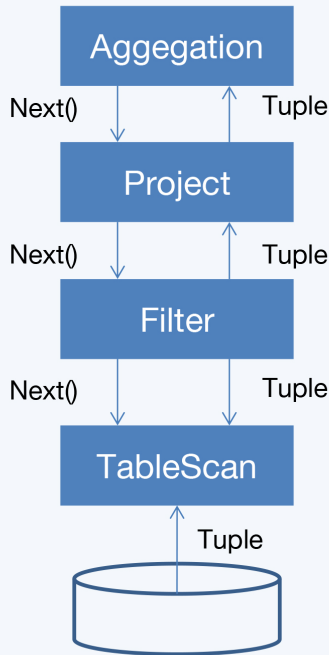
1. 调用子节点 Operator 的 next() 接口，获取一个元组 (Tuple)；
2. 对元组执行 Operator 特定的处理；
3. 返回处理后的元组。

通过火山模型，查询执行引擎可以优雅地将任意 Operator 组装在一起，而不需要考虑每个 Operator 的具体处理逻辑。查询执行时会由查询树自顶向下嵌套调用 next() 接口，数据则自底向上地被拉取处理。所以，这种处理方式也称为拉取执行模型 (Pull Based)。

为了更好地理解火山模型的拉取执行过程，让我们来看一个聚合计算的例子，它来自 Databricks 的 [一篇文章](#) (Sameer Agarwal et al. (2016))。

 复制代码

```
1 select count(*) from store_sales where ss_item_sk = 1000;
```



开始从扫描运算符 TableScan 获取数据，通过过滤运算符 Filter 开始推动元组的处理。然后，过滤运算符传递符合条件的元组到聚合运算符 Aggregate。

你可能对“元组”这个词有点陌生，其实它大致就是指数据记录（Record），因为讨论算法时学术文献中普遍会使用元组这个词，为了让你更好地与其他资料衔接起来，我们这里就沿用“元组”这个词。

火山模型的优点是处理逻辑清晰，每个 Operator 只要关心自己的处理逻辑即可，耦合性低。但是它的缺点也非常明显，主要是两点：

1. 虚函数调用次数过多，造成 CPU 资源的浪费。
2. 数据以行为单位进行处理，不利于发挥现代 CPU 的特性。

问题分析

我猜你可能会问，什么是虚函数呢？

在火山模型中，处理一个元组最少需要调用一次 next() 函数，这个 next() 就是虚函数。这些函数的调用是由编译器通过虚函数调度实现的；虽然虚函数调度是现代计算机体系结构中重点优化部分，但它仍然需要消耗很多 CPU 指令，所以相当慢。

第二个缺点是没有发挥现代 CPU 的特性，那具体又是怎么回事？

CPU 寄存器和内存

在火山模型中，每次一个算子给另外一个算子传递元组的时候，都需要将这个元组存放在内存中，在 18 讲我们已经介绍过，以行为组织单位很容易带来 CPU 缓存失效。

循环展开 (Loop unrolling)

当运行简单的循环时，现代编译器和 CPU 是非常高效的。编译器会自动展开简单的循环，甚至在每个 CPU 指令中产生单指令多数据流 (SIMD) 指令来处理多个元组。

单指令多数据流 (SIMD)

SIMD 指令可以在同一 CPU 时钟周期内，对同列的不同数据执行相同的指令。这些数据会加载到 SIMD 寄存器中。

Intel 编译器配置了 AVX-512 (高级矢量扩展) 指令集，SIMD 寄存器达到 512 比特，就是说可以并行运算 16 个 4 字节的整数。

在过去大概 20 年的时间里火山模型都运行得很好，主要是因为这一时期执行过程的瓶颈是磁盘 I/O。而现代数据库大量使用内存后，读取效率大幅提升，CPU 就成了新的瓶颈。因此，现在对火山模型的所有优化和改进都是围绕着提升 CPU 运行效率展开的。

改良方法 (运算符融合)

要对火山模型进行优化，一个最简单的方法就是减少执行过程中 Operator 的函数调用。比如，通常来说 Project 和 Filter 都是常见的 Operator，在很多查询计划中都会出现。OceanBase1.0 就将两个 Operator 融合到了其它的 Operator 中。这样做有两个好处：

1. 降低了整个查询计划中 Operator 的数量，也就简化了 Operator 间的嵌套调用关系，最终减少了虚函数调用次数。
2. 单个 Operator 的处理逻辑更集中，增强了代码局部性能力，更容易发挥 CPU 的分支预测能力。

分支预测能力

你可能还不了解什么是分支预测能力，我这里简单解释一下。

分支预测是指 CPU 执行跳转指令时的一种优化技术。当出现程序分支时 CPU 需要执行跳转指令，在跳转的目的地址之前无法确定下一条指令，就只能让流水线等待，这就降低了 CPU 效率。为了提高效率，设计者在 CPU 中引入了一组寄存器，用来专门记录最近几次某个地址的跳转指令。

这样，当下次执行到这个跳转指令时，就可以直接取出上次保存的指令，放入流水线。等到真正获取到指令时，如果证明取错了则推翻当前流水线中的指令，执行真正的指令。

这样即使出现分支也能保持较好的处理效率，但是寄存器的大小总是有限的，所以总的来说还是要控制程序分支，分支越少流水线效率就越高。

刚刚说的运算符融合是一种针对性的优化方法，优点是实现简便而且快速见效，但进一步的提升空间很有限。

因此，学术界还有一些更积极的改进思路，主要是两种。一种是优化现有的迭代模型，每次返回一批数据而不是一个元组，这就是向量化模型（Vectorization）；另一种是从根本上消除迭代计算的性能损耗，这就是代码生成（Code Generation）。

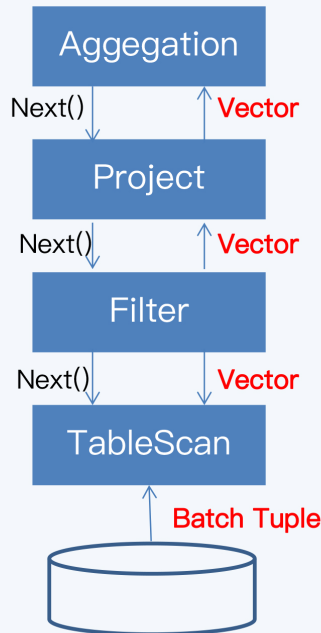
我们先来看看向量化模型。

向量化：TiDB&CockroachDB

向量化模型最早提出是在 [MonerDB-X100 \(Vectorwise\)](#) 系统，已成为现代硬件条件下广泛使用的两种高效查询引擎之一。

向量化模型与火山模型的最大差异就是，其中的 Operator 是向量化运算符，是基于列来重写查询处理算法的。所以简单来说，向量化模型是由一系列支持向量化运算的 Operator 组成的执行模型。

我们来看一下向量化模型怎么处理聚合计算。



通过这个执行过程可以发现，向量化模型依然采用了拉取式模型。它和火山模型的唯一区别就是 Operator 的 next() 函数每次返回的是一个向量块，而不是一个元组。向量块是访问数据的基本单元，由固定的一组向量组成，这些向量和列 / 字段有一一对应的关系。

向量处理背后的主要思想是，按列组织数据和计算，充分利用 CPU，把从多列到元组的转化推迟到较晚的时候执行。这种方法在不同的操作符间平摊了函数调用的开销。


向量化模型首先在 OLAP 数据库中采用，与列式存储搭配使用可以获得更好的效果，例如 ClickHouse。

我们课程里定义的分布式数据库都是面向 OLTP 场景的，所以不能直接使用列式存储，但是可以采用折中的方式来实现向量化模型，也就是在底层的 Operator 中完成多行到向量块的转化，上层的 Operator 都是以向量块作为输入。这样改造后，即使是与行式存储结合，仍然能够显著提升性能。在 TiDB 和 CockroachDB 的实践中，性能提升可以达到数倍，甚至数十倍。

向量化运算符示例

我们以 Hash Join 为例，来看下向量化模型的执行情况。

在 [第 20 讲](#) 我们已经介绍过 Hash Join 的执行逻辑，就是两表关联时，以 Inner 表的数据构建 Hash 表，然后以 Outer 表中的每行记录，分别去 Hash 表查找。

 复制代码

```
1 Class HashJoin
2   Primitives probeHash_, compareKeys_, buildGather_;
3   ...
4   int HashJoin::next()
5       //消费构建侧的数据构造Hash表，代码省略
6       ...
7       //获取探测侧的元组
8       int n = probe->next()
9       //计算Hash值
10      vec<int> hashes = probeHash_.eval(n)
11      //找到Hash匹配的候选元组
12      vec<Entry*> candidates = ht.findCandidates(hashes)
13      vec<Entry*, int> matches = {}
14      //检测是否匹配
15      while(candidates.size() > 0)
16          vec<bool> isEqual = compareKeys_.eval(n, candidates)
17          hits, candidates = extractHits(isEqual, candidates)
18          matches += hits
19      //从Hash表收集数据为下个Operator缓存
20      buildGather_.eval(matches)
21      return matches.size()
```

我们可以看到这段处理逻辑中的变量都是 Vector，还有事先定义一些专门处理 Vector 的元语（Primitives）。


总的来说，向量化执行模型对火山模型做了针对性优化，在以下几方面有明显改善：

1. 减少虚函数调用数量，提高了分支预测准确性；
2. 以向量块为单位处理数据，利用 CPU 的数据预取特性，提高了 CPU 缓存命中率；
3. 多行并发处理，发挥了 CPU 的并发执行和 SIMD 特性。

代码生成：OceanBase


与向量化模型并列的另一种高效查询执行引擎就是“代码生成”，这个名字听上去可能有点奇怪，但确实没有更好翻译。代码生成的全称是以数据为中心的代码生成（Data-Centric Code Generation），也被称为编译执行（Compilation）。

在解释“代码生成”前，我们先来分析一下手写代码和通用性代码的执行效率问题。我们还是继续使用讲火山模型时提到的例子，将其中 Filter 算子的实现逻辑表述如下：

 复制代码

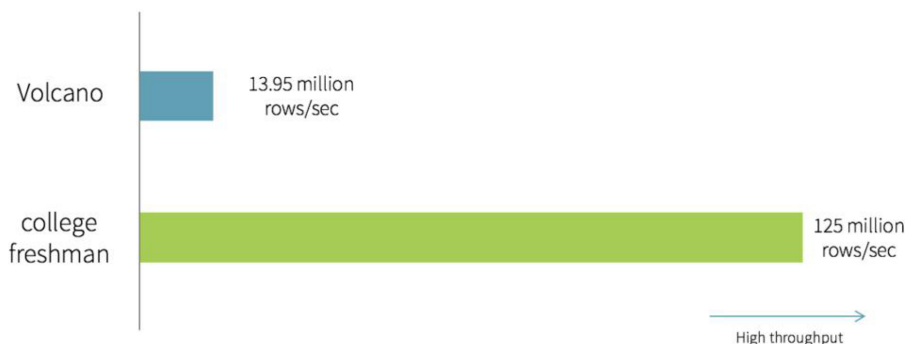
```
1 class Filter(child: Operator, predicate: (Row => Boolean))
2   extends Operator {
3   def next(): Row = {
4     var current = child.next()
5     while (current == null || predicate(current)) {
6       current = child.next()
7     }
8     return current
9   }
10 }
```

如果专门对这个操作编写代码（手写代码），那么大致是下面这样：

 复制代码

```
1 var count = 0
2 for (ss_item_sk in store_sales) {
3   if (ss_item_sk == 1000) {
4     count += 1
5   }
6 }
```

在两种执行方式中，手写代码显然没有通用性，但 Databricks 的工程师对比了两者的执行效率，测试显示手工代码的吞吐能力要明显优于火山模型。



引自Sameer Agarwal et al. (2016)

手工编写代码的执行效率之所以高，就是因为它的循环次数要远远小于火山模型。而代码生成就是按照一定的策略，通过即时编译（JIT）生成代码可以达到类似手写代码的效果。

此外，代码生成是一个推送执行模型（Push Based），这也有助于解决火山模型嵌套调用虚函数过多的问题。与拉取模型相反，推送模型自底向上地执行，执行逻辑的起点直接就在最底层 Operator，其处理完一个元组之后，再传给上层 Operator 继续处理。

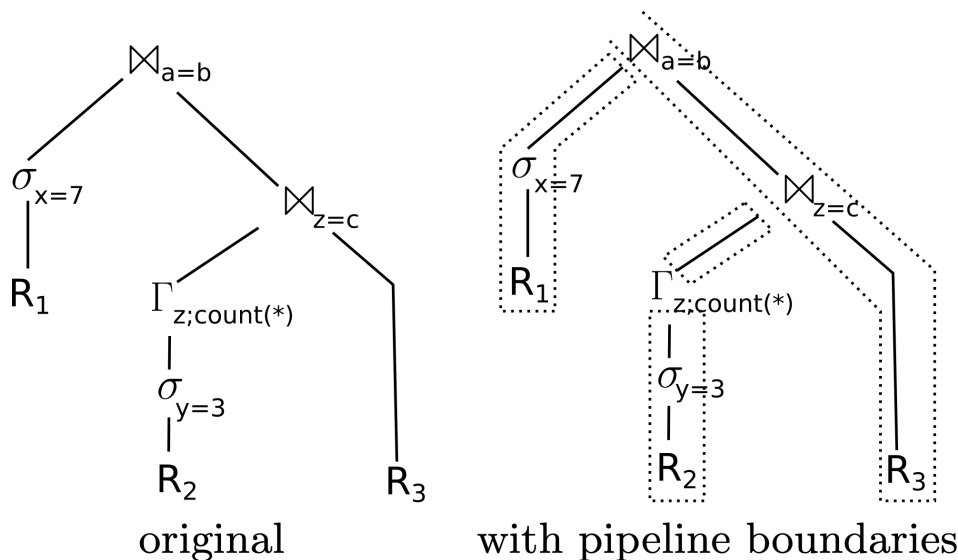
Hyper 是一个深入使用代码生成技术的数据库，[Hyper 实现的论文](#)（Thomas Neumann (2011)）中有一个例子，我这里引用过来帮助你理解它的执行过程。

要执行的查询语句是这样的：

```
1 select * from R1,R3,  
2 (select R2.z,count(*)  
3   from R2  
4   where R2.y=3  
5   group by R2.z) R2  
6 where R1.x=7 and R1.a=R3.b and R2.z=R3.c
```

[复制代码](#)

SQL 解析后会得到一棵查询树，就是下图的左侧的样子，我们可以找到 R1、R2 和 R3 对应的是三个分支。



引自Thomas Neumann (2011)

要获得最优的 CPU 执行效率，就要使数据尽量不离开 CPU 的寄存器，这样就可以在一个 CPU 流水线（Pipeline）上完成数据的处理。但是，查询计划中的 Join 操作要生成 Hash 表加载到内存中，这个动作使数据必须离开寄存器，称为物化（Materialize）。所以整个

执行过程会被物化操作分隔为 4 个 Pipeline。而像 Join 这种会导致物化操作的 Operator，在论文称为 Pipeline-breaker。

通过即时编译生成代码得到对应 Pipeline 的四个代码段，可以表示为下面的伪码：

```

[ for each tuple  $t$  in  $R_1$ 
  if  $t.x = 7$ 
    materialize  $t$  in hash table of  $\bowtie_{a=b}$ 
[ for each tuple  $t$  in  $R_2$ 
  if  $t.y = 3$ 
    aggregate  $t$  in hash table of  $\Gamma_z$ 
[ for each tuple  $t$  in  $\Gamma_z$ 
  materialize  $t$  in hash table of  $\bowtie_{z=c}$ 
[ for each tuple  $t_3$  in  $R_3$ 
  for each match  $t_2$  in  $\bowtie_{z=c}[t_3.c]$ 
    for each match  $t_1$  in  $\bowtie_{a=b}[t_3.b]$ 
      output  $t_1 \circ t_2 \circ t_3$ 

```

引自 Thomas Neumann (2011)

代码生成消除了火山模型中的大量虚函数调用，让大部分指令可以直接从寄存器取数，极大地提高了 CPU 的执行效率。

代码生成的基本逻辑清楚了，但它的工程实现还是挺复杂的，所以会有不同粒度的划分。比如，如果是整个查询计划的粒度，就会称为整体代码生成（Whole-Stage Code Generation），这个难度最大；相对容易些的是代码生成应用于表达式求值（Expression Evaluation），也称为表达式代码生成。在 OceanBase 2.0 版本中就实现了表达式代码生成。

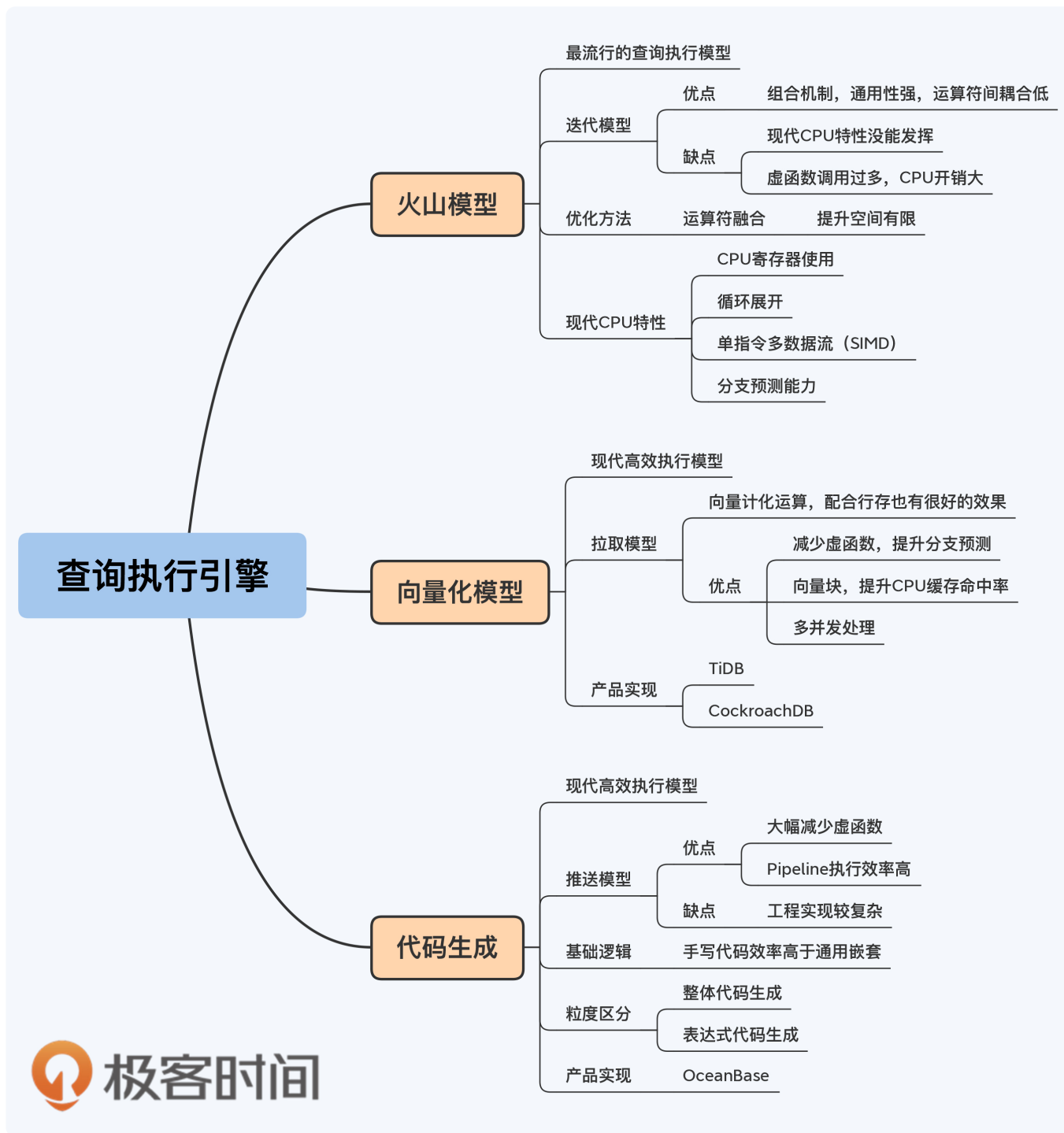
如果你想再深入了解代码生成的相关技术，就需要有更全面的编译器方面的知识做基础，比如你可以学习宫文学老师的编译原理课程。

小结

那么，今天的课程就到这里了，让我们梳理一下这一讲的要点。

1. 火山模型自 1990 年提出后，是长期流行的查询执行模型，至今仍在 Oracle、MySQL 中使用。但面对海量数据时，火山模型有 CPU 使用率低的问题，性能有待提升。
2. 火山模型仍有一些优化空间，比如运算符融合，可以适度减少虚函数调用，但提升空间有限。学术界提出的两种优化方案是向量化和代码生成。
3. 简单来说，向量化模型就是一系列向量化运算符组成的执行模型。向量化模型首先在 OLAP 数据库和大数据领域广泛使用，配合列式存储取得很好的效果。虽然 OLTP 数据库的场景不适于列式存储，但将其与行式存储结合也取得了明显的性能提升。
4. 代码生成是现代编译器与 CPU 结合的产物，也可以大幅提升查询执行效率。代码生成的基础逻辑是，针对性的代码在执行效率上必然优于通用运算符嵌套。代码生成根据算法会被划分成多个在 Pipeline 执行的单元，提升 CPU 效率。代码生成有不同的粒度，包括整体代码生成和表达式代码生成，粒度越大实现难度越大。

向量化和代码生成是两种高效查询模型，并没有最先出现在分布式数据库领域，反而是在 OLAP 数据库和大数据计算领域得到了更广泛的实践。ClickHouse 和 Spark 都同时混用了代码生成和向量化模型这两项技术。目前 TiDB 和 CockroachDB 都应用向量化模型，查询性能得到了一个数量级的提升。OceanBase 中则应用了代码生成技术优化了表达式运算。



思考题

课程的最后，我们来看看今天的思考题。这一讲，我们主要讨论了查询执行引擎的优化，核心是如何最大程度发挥现代 CPU 的特性。其实，这也是基础软件演进中一个普遍规律，每当硬件技术取得突破后就会引发软件的革新。那么，我的问题就是你了解的基础软件中，哪些产品分享了硬件技术变革的红利呢？

欢迎你在评论区留言和我一起讨论，我会在答疑篇和你继续讨论这个问题。如果你身边的朋友也对查询执行引擎这个话题感兴趣，你也可以把今天这一讲分享给他，我们一起讨论。

学习资料

Goetz Graefe: [🔗 Volcano, an Extensible and Parallel Query Evaluation System](#)

Peter Boncz et al.: [🔗 MonetDB/X100: Hyper-Pipelining Query Execution](#)

Sameer Agarwal et al.: [🔗 Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop](#)

Thomas Neumann: [🔗 Efficiently Compiling Efficient Query Plans for Modern Hardware](#)

提建议

更多课程推荐

数据结构与算法之美

为工程师量身打造的数据结构与算法私教课

王争

前 Google 工程师



立省 ¥40 🖱️

破 90000 订阅特惠，到手价 ¥89

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 20 | 关联查询：如何提升多表Join能力？

下一篇 22 | RUM猜想：想要读写快还是存储省？ 又是三选二

精选留言 (3)

写留言



崔伟协

2020-09-29

火山模型怎么处理聚合如max, min,sum这些

展开 ▾



扩散性百万咸面包

2020-09-25

老师，这里有个问题，既然MySQL等用了火山模型，那它们为什么不改成向量化接口呢？
既然提升如此明显？

感觉向量化就是把多个tuple一次返回，向量化和行式存储结合的难点在哪呢？

展开 ▾



Jxin

2020-09-25

感觉代码生成==编译器运行期优化。

- 1.针对循环做优化
- 2.减少过程调用开销
- 3.对控制流做优化
- 4.向量计算

展开 ▾

作者回复：总结的很好。

