Apache Spark作为编译器：深入介绍新的Tungsten执行引擎

本文原文：Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop Deep dive into the new Tungsten execution engine：https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html

本文已经投稿自：http://geek.csdn.net/news/detail/77005

《Spark 2.0技术预览：更容易、更快速、更智能》文中简单地介绍了Spark 2.0相关技术，本文将深入介绍新的Tungsten执行引擎。

Apache Spark已经非常快了，但是我们能不能让它再快10倍？

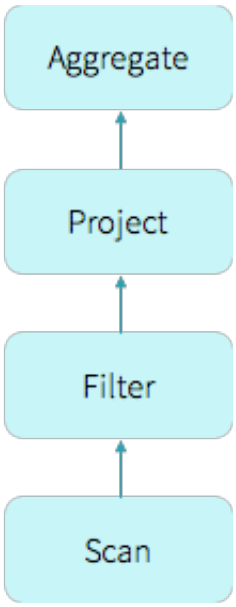
这个问题使我们从根本上重新思考Spark物理执行层的设计。当你随便调查一个现代数据引擎（比如Spark、其他的MPP数据库），你会发现大部分的CPU周期都花费在无用的工作之上，比如虚函数的调用；或者读取/写入中间数据到CPU高速缓存或内存中。通过减少花在这些无用功的CPU周期一直是现代编译器长期性能优化的重点。

Apache Spark 2.0中附带了第二代Tungsten engine。这一代引擎是建立在现代编译器和MPP数据库的想法上，并且把它们应用于数据的处理过程中。主要想法是通过在运行期间优化那些拖慢整个查询的代码到一个单独的函数中，消除虚拟函数的调用以及利用CPU寄存器来存放那些中间数据。作为这种流线型策略的结果，我们显著提高CPU效率并且获得了性能提升，我们把这些技术统称为"整段代码生成"(whole-stage code generation)。

过去：Volcano迭代模型(Volcano Iterator Model)

在我们深入介绍whole-stage code generation之前，让我们先回顾一下现在的Spark（以及大多数数据库系统）是如何运行的。让我们看看这个简单的查询：扫描一个表，然后计算出满足给定条件属性值的总行数。

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



为了计算这个查询，旧版本的Spark(1.x)会利用基于迭代模型的经典查询评估策略(通常被称为Volcano model)。在这个模型中，一个查询由多个算子(operators)组成，每个算子都提供了next()接口，该接口每次只返回一个元组(tuple)给嵌套树中的下一个算子。比如上面查询中的Filter算子大致可以翻译成下面的代码：

```
class Filter(child: Operator, predicate: (Row => Boolean))
  extends Operator {
  def next(): Row = {
    var current = child.next()
    while (current == null || predicate(current)) {
      current = child.next()
    }
    return current
  }
}
```

让每个算子实现迭代器接口允许查询执行引擎来优雅地组合任意的算子，而不必担心每个算子提供的数据类型。结果Volcano模型在过去的二十年间变成数据库系统的标准，这个也是Spark使用的架构。

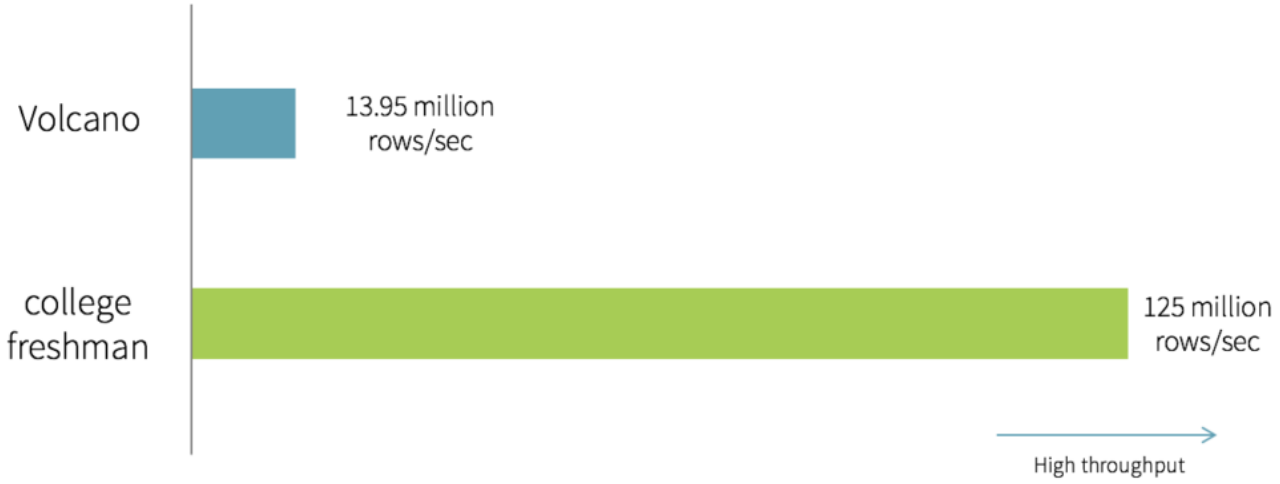
Volcano与手写代码

如果我们给一个大学新生十分钟的时间使用Java来实现上面的查询。他很可能会想出一段迭代代码来循环遍历输入，判断条件并计算行数，如下所示：

```
var count = 0
```

```
for (ss_item_sk in store_sales) {
  if (ss_item_sk == 1000) {
    count += 1
  }
}
```

上面编写的代码仅仅是专门解决一个给定的查询，而且很明显不能和其他算子进行组合。但是这两种实现（Volcano与手写代码）方式在性能上有啥重要区别呢？一方面Spark和大多数关系型数据库选择这种可以对不同算子进行组合的结构；另一方面，我们有一个由新手在十分钟编写的程序。我们运行了一个简单的基准测试，对比了"大学新生"版的程序和Spark版的程序在使用单个线程的情况下运行上面同一份查询，并且这些数据存储在磁盘上，格式为Parquet。下面是它们之间的对比：



正如你所看到的，大学新生手写版本的程序要比Volcano模式的程序要快一个数量级！事实证明，6行的Java代码是被优化过的，其原因如下：

1、没有虚函数调用
：在Volcano模型中，处理一个元组(tuple)最少需要调用一次next()函数。这些函数的调用是由编译器通过虚函数调度实现的（通过vtable）；而手写版本的代码没有一个函数调用。虽然虚函数调度是现代计算机体系结构中重点优化部分，它仍然需要消耗很多CPU指令而且相当的慢，特别是调度数十亿次。

2、内存和CPU寄存器中的临时数据
：在Volcano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；而在手写版本的代码中，编译器(这个例子中是JVM JIT)实际上将临时数据存放在CPU寄存器中。访问内存中的数据所需要的CPU时间比直接访问在寄存器中的数据要大一个数量级！

3、循环展开(Loop unrolling)和SIMD

：当运行简单的循环时，现代编译器和CPU是令人难以置信的高效。编译器会自动展开简单的循环，甚至在每个CPU指令中产生SIMD指令来处理多个元组。CPU的特性，比如管道(pipelining)、预取(prefetching)以及指令重排序(instruction reordering)使得运行简单的循环非常地高效。然而这些编译器和CPU对复杂函数调用图的优化极少，而这些函数正是Volcano模型依赖的。

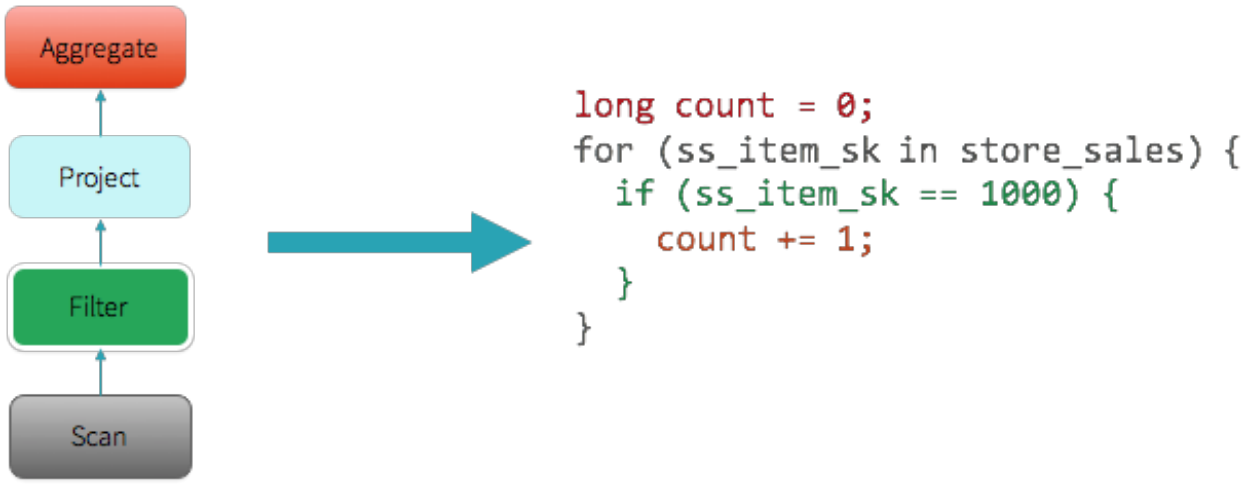
这里面的关键点是手写版本代码的编写是正对上面的查询，所以它充分利用到已知的所有信息，导致消除了虚函数的调用，将临时数据存放在CPU寄存器中，并且可以通过底层硬件进行优化。

未来：整段代码生成

从上面的观察我们下一步的探讨自然是在运行时自动生成手写版的代码的可能性，这些技术统称为整段代码生成(whole-stage code generation)。这个想法是受Thomas Neumann发表在VLDB 2011上的论文Efficiently Compiling Efficient Query Plans for Modern Hardware所启发。

我们的目标是使用整段代码生成使得Spark计算引擎可以实现手写代码的性能，并且提供通用的功能。而不是在运行时依赖算子来处理数据，这些算子在运行时生成代码，如果可能的话将所有的查询片段组成到单个函数中，后面我们仅需要运行生成的代码。

比如对于上面的查询可以作为单个阶段，Spark可以产生以下的JVM字节码(这里展示的是Java代码)。复杂的查询将会产生多个阶段，这种情况下Spark将会产生多个不同的函数。



下面表达式中的explain()函数为整段代码生成进行了扩展。在输出的结果里，当算子前面有一个*，说明整段代码生成被启用。在下面情况下，Range、Filter和两个Aggregates算子都启用了整段代码生成。然而Exchange算子并没有实现整段代码生成，因为它需要通过网络发送数据。

```
spark.range(1000).filter("id > 100").selectExpr("sum(id)").explain()

== Physical Plan ==
*Aggregate(functions=[sum(id#201L)])
+- Exchange SinglePartition, None
   +- *Aggregate(functions=[sum(id#201L)])
      +- *Filter (id#201L > 100)
         +- *Range 0, 1, 3, 1000, [id#201L]
```

对于这些紧密关注Spark开发的人可能会问：我在Apache Spark 1.1中就听说过代码生成(code generation)，它和这里讲的有啥区别？在过去和其他MPP查询引擎一样，Spark仅仅将代码生成应用于表达式求值(expression evaluation)，而且仅限于很小一部分的算子(比如Project, Filter)。也就是说，之前的代码生成技术仅仅是加快了表达式的求值(比如1+a)；然而今天的整段代码生成技术实际上为整个查询计划生成代码。

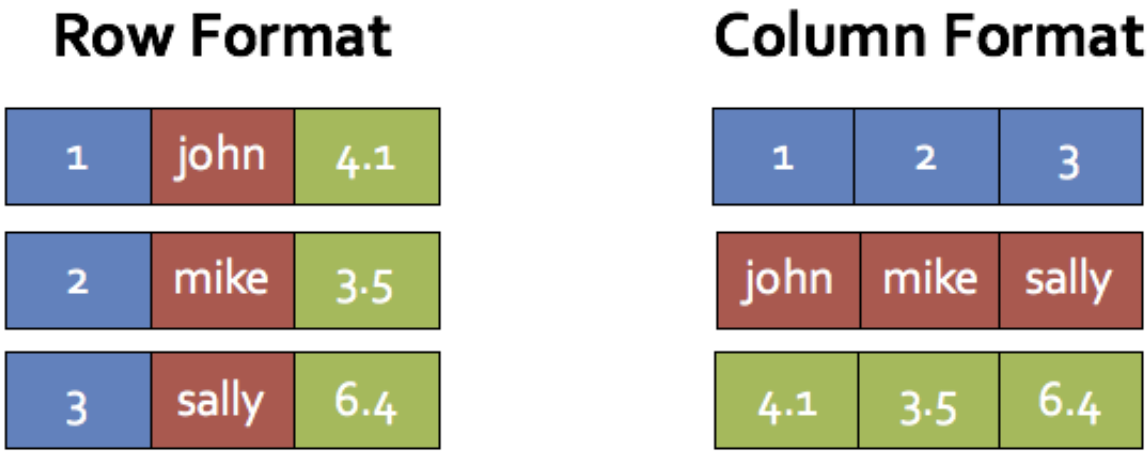
Vectorization

Whole-stage code-generation技术对那些在大型数据集根据条件过滤的大规模简单查询非常有效，但还是存在那些无法生成代码将整个查询融合到一个函数的情况。有些算子可能非常地复杂(比如CSV解析或者Parquet解码)，或者有些情况下我们会整合第三方组件，而这些组件的代码是无法集成到我们生成的代码之中。

为了提高这些情况下的性能，我们提出一个新的方法叫做向量化(vectorization)。核心思想是：我们不是一次只处理一行数据，而是将许多行的数据分别组成batches，而且采用列式格式存储

；然后每个算子对每个batch进行简单的循环来遍历其中的数据。所以每次调用next()函数都会返回一批的元组，这样可以分摊虚函数调用的开销。采用了这些措施之后，这些简单的循环也会使得编译器和CPU运行的更加高效。

假设有一个表有三列 (id, name, score)，下面展示了面向行格式和面向列格式的内存布局：



这种风格的处理可以实现上面提到三点中的两点(也两点分别是：几乎没有虚函数调用以及自动展开循环/SIMD)，这种风格仍然需要将临时数据存放在内存中而不是存放在CPU的寄存器中。所以我们只有在无法使用Whole-stage code-generation技术的情况下才会使用vectorization技术。

比如我们已经实现了一个新的矢量Parquet解码器(vectorized Parquet reader)可以批量解码和解压。当解码存在磁盘上的integer类型的列，新的解码器大概是旧的解码器的9倍!

性能基准测试

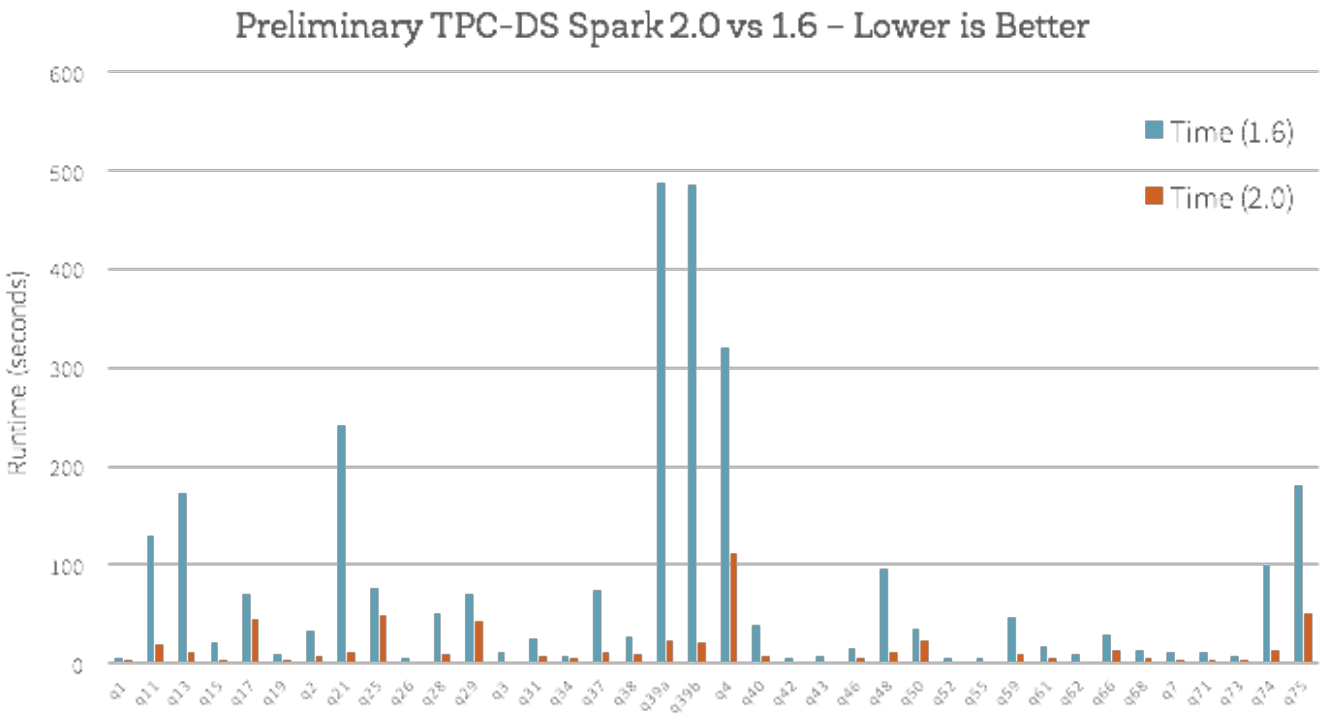
为了有个直观的感受，我们记录下在Spark 1.6和Spark 2.0中在一个核上处理一行的操作时间(单位是纳秒)，下面的表格能够体现出新的Tungsten engine的威力。Spark 1.6使用的表达式代码生成技术同样在今天的其他商业数据库中采用。

primitive	Spark 1.6	Spark 2.0
filter	15 ns	1.1 ns
sum w/o group	14 ns	0.9 ns
sum w/ group	79 ns	10.7 ns
hash join	115 ns	4.0 ns
sort (8-bit entropy)	620 ns	5.3 ns
sort (64-bit entropy)	620 ns	40 ns
sort-merge join	750 ns	700 ns
Parquet decoding (single int column)	120 ns	713 ns

我们已经调查了客户的workloads，并且对那些使用非常频繁的算子实现了whole-stage code generation 技术，这些算子包括：filter, aggregate以及hash join。正如你在上表看到的，许多核心的算子在采用whole-stage code generation 技术之后的性能已经提高了一个数量级！然而有些算子(比如sort-merge join)天生就很慢，所以非常难对其进行优化。

你可以在这里看到whole-stage code generation技术的威力，在那里我们在一台机器上对10亿条记录进行aggregations和joins操作。不管是在Databricks platform(Intel Haswell处理器,3核)还是在一台2013年出售的Macbook Pro(Intel Haswell i7)电脑上，对10亿条元组进行hash join操作采用的时间不到1秒！

在端到端查询这个新引擎是如何工作的？除了whole-stage code generation和vectorization技术，对一般的查询我们在Catalyst optimizer上也进行了很多其他的工作，比如nullability propagation。我们比较了Spark 1.6和Spark 2.0在使用TPC-DS查询的基本分析，如下图：



那是不是意味着你把Spark升级Spark 2.0，所以的workload将会变的比之前快10倍呢？其实不是，虽然我们相信新的Tungsten引擎为我们的性能优化实现了最好的数据处理架构；但是我们需要理解的是，不是所有的workload都能享受到同样的程度。比如可变长度的数据类型(如字符串)对其操作本身就非常昂贵；还有些workloads受其他因素影响，比如I/O吞吐量以及元数据操作等。对于之前那些受CPU效率影响的Workloads将会获得最大的效率。

结论

本文提到的绝大部分工作已经提交到Apache Spark的代码中，并且将会在Spark 2.0版本发布。关于whole-stage code generation技术的JIRA可以到SPARK-12795里查看；而关于vectorization技术的JIRA可以到SPARK-12992查看。

总结一下，本文主要描述了第二代Tungsten执行引擎。通过whole-stage code generation技术，这个引擎可以(1)、消除虚函数调用；(2)、将临时数据从内存中移到CPU寄存器中；（3）、利用现代CPU特性来展开循环并使用SIMD功能。通过vectorization技术，可以加快那些代码生成比较复杂的算子运行速度。对于数据处理中很多核心算子，新的引擎会使它们的运行速度提升一个数量级。在未来，考虑到执行引擎的效率，我们大部分的优化工作将会转移到优化I/O效率以及更好的查询计划。