# 概述

参考：

<https://www.cnblogs.com/lovezhr/p/15986295.html>

# CPU处理特性

## 超标量流水线与乱序执行

## 分支预测

## 多级存储与数据预取

## SIMD

# 执行模型

## 迭代模型/火山模型

参考：

<https://zhuanlan.zhihu.com/p/472354855>

### 概述

火山模型又称Volcano Model 或者Pipeline Model。

Iterator Model该计算模型将关系代数中每一种操作抽象为一个Operator，将整个SQL构建成一个Operator树，查询树自顶向下的调用next()接口，数据则自底向上的被拉取处理。

火山模型的这种处理方式也称为拉取执行模型(Pull Based)。

### 原理

一条sql执行过程中，首先进行词法分析和语法分析，然后将由优化器进行判断，如何执行更有效率，生成执行计划，后面的任务就交给了执行器。在执行的过程中，执行器就会和存储引擎交互了，交互是以记录为单位的。

MySQL在做完语法解析后，调用函数 mysql\_execute\_command 进入查询优化器。查询优化器对 sql 语句进行了一系列的转换，重写，优化最终生产了 AccessPath(访问路径)，并且根据AccessPath创建Iterator迭代器。

火山模型是数据库查询执行最著名的模型，也是在各种数据库系统中应用最广泛的模型。SQL语句在数据库中经过语法解析生产AST语法树，然后遍历语法树，生成执行树。执行树的每个节点为代数运算符(Operator)。火山模型把Operator看成迭代器，每个迭代器都会提供一个next()接口。一般Operator的next()接口实现分为三步：

（1）调用子节点Operator的next() 接口获取一行数据(tuple)

（2）对tuple进行Operator特定的处理(如filter或project等)

（3）返回处理后的tuple

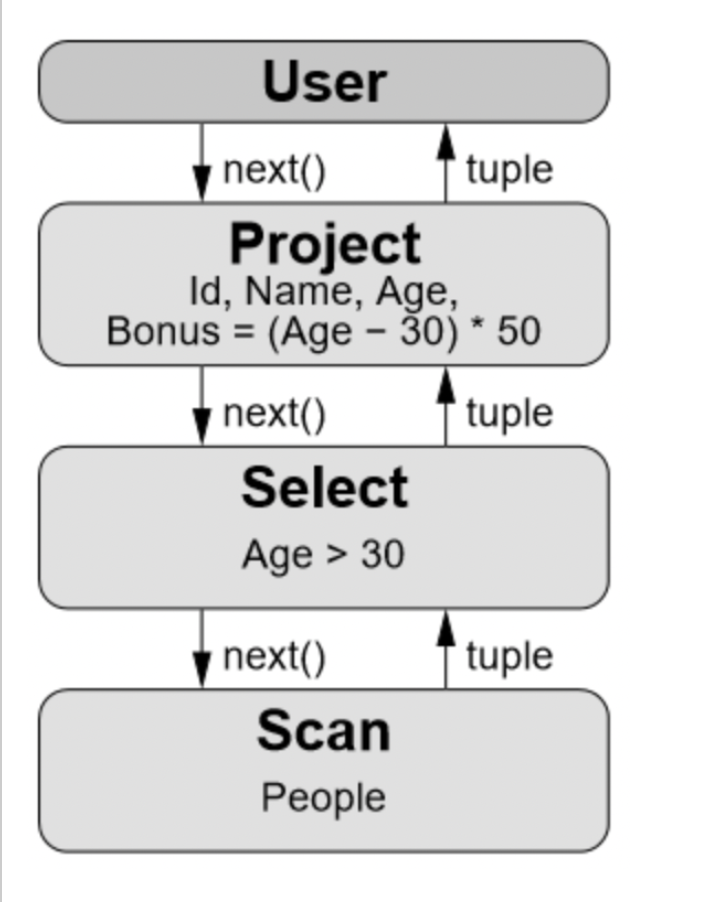
因此，查询执行时会由查询树自顶向下的调用next()接口，数据则自底向上的被拉取处理。这种处理方式也称为拉取执行模型(Pull Based)。例如以下SQL:

SELECT Id, Name, Age, (Age - 30) \* 50 AS Bonus

FROM People

WHERE Age > 30

对应火山模型如下：



User：客户端；

Project：垂直分割（投影），选择字段；

Select（或Filter）：水平分割（选择)，用于过滤行，也称为谓词；

Scan：扫描数据。

这里包含了3个Operator，首先User调用最上方的Operator（Project）希望得到next tuple，Project调用子节点（Select），而Select又调用子节点（Scan），Scan获得表中的tuple返回给Select，Select会检查是否满足过滤条件，如果满足则返回给Project，如果不满足则请求Scan获取next tuple。Project 会对每一个tuple选择需要的字段或者计算新字段并返回新的tuple给User。当Scan发现没有数据可以获取时，则返回一个结束标记告诉上游已结束。

为了更好地理解一个Operator中发生了什么，下面通过伪代码来理解 Select Operator：

Tuple Select::next() {

while (true) {

Tuple candidate = child->next(); // 从子节点中获取 next tuple

if (candidate == EndOfStream) // 是否得到结束标记

return EndOfStream;

if (condition->check(candidate)) // 是否满足过滤条件

return candidate; // 返回 tuple

}

}

#### MySQL8.0迭代器

MySQL8.0对执行器进行了改进，创建一个新的用于迭代访问记录的API，它足够通用。主要实现了一个通用的C++类接口，叫做RowIterator，它具有以下成员和函数：

构造和析构函数

init()：打开所有必须的资源，也有可能执行部分功能性操作。比如SortingIterator中会进行排序操作，这个函数可以多次调用，每次调用都会重置迭代器的指示位置。

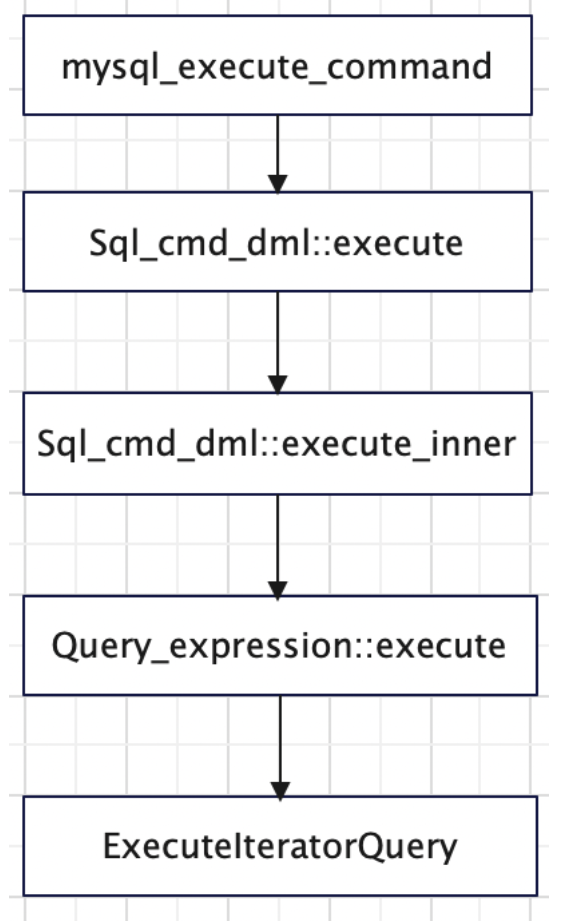
read()：读取一行，将行放入记录缓存中

UlockRow()：将一行过滤出结果集后，允许低事务隔离级别释放该行的所有锁。



**函数调用栈**

如下图所示：调用Query\_expression::execute函数进入执行阶段：



MySQL8.0调用逻辑：

调用关系：

mysql\_executor\_command

-> Sql\_cmd\_dml::execute

->Sql\_cmd\_dml::execute\_inner

->SELECT\_LEX\_UNIT::optimize

->SELECT\_LEX::optimize

->JOIN::optimize

->SELECT\_LEX\_UNIT::create\_access\_paths

->CreateIteratorFromAccessPath

->SELECT\_LEX\_UNIT::execute(5.7 Query\_expression::execute)

->ExecuteIteratorQuery：火山模型

**函数ExecuteIteratorQuery浅析**

1、is\_simple()函数用来判断一个查询表达式是否有union或者多级order，如果没有说明这个查询语句简单。就执行​​add\_select\_number。

2、运行​​ClearForExecution​​函数。在初始化root迭代器之前，把之前的执行迭代器的数据清除。

3、运行​​get\_field\_list()​​​，获取查询表达式的字段列表，并将所有字段都放到一个deque中，即​​mem\_root\_deque<Item\*>​​;对于查询块的并集，返回在准备期间生成的字段列表，对于单个查询块，尽可能返回字段列表

4、运行​​start\_execution​​,准备执行查询表达式或DML查询

5、接下来的一些操作与第二引擎有关，关于该引擎见https://www.h5w3.com/123061.html。Secondary Engine实际上是MySQL sever上同时支持两个存储引擎，把一部分主引擎上的数据，在Secondary Engine上也保存一份，然后查询的时候会根据优化器的的选择决定在哪个引擎上处理数据。

6、如果该查询用于子查询，那么重新reset，指向子查询。

7、接下来是对于复杂句以及简单句的不同处理，从而给​​send\_records\_ptr​​赋值。

函数对于这个情况的解释如下：

We need to accumulate in the first join's send\_records as long as we support SQL\_CALC\_FOUND\_ROWS, since LimitOffsetIterator will use it for reporting rows skipped by OFFSET or LIMIT. When we get rid of SQL\_CALC\_FOUND\_ROWS, we can use a local variable here instead.

情况一：如果该查询块具有UNION或者多级的ORDER BY/LIMIT的话 UNION with LIMIT的话，​​found\_rows()​​用于最外层。

LimitOffsetIterator​​​跳过偏移量行写入​​send\_records​​

情况二：如果是个简单句的话found\_rows()​​直接用到join上。

LimitOffsetIterator​​​跳过偏移量行写入​​send\_records​​

情况三：如果是UNION，但是没有LIMIT。found\_rows()​​用于最外层。

8、重置计数器

9、接下来是一个对查询块遍历，逐个释放内存的操作，用以增加并发性并减少内存消耗。

10、初始化根迭代器

11、然后for循环，从根迭代器一直到引擎的handler，调用读取数据。如果出错就直接返回。如果收到kill信号，也返回。在循环中对​​send\_records\_ptr​​进行累加。行计数器++，指向下一行。

12、将​​send\_records\_ptr​​​赋值给该线程的​​current\_found\_rows​​。

说明：在GoldenDB分布式数据库中，计算引擎采用的就是火山模型。

### 特点

#### 优点

早期数据库受限于硬件水平，IO、内存和CPU资源都非常昂贵，比如计算层的数据一多，内存容易爆掉，所以火山模型采用每次只计算一行数据的方式，极大缩减了内存使用量。

火山模型的优点在于：简单，每个Operator可以单独实现逻辑，不需要关心其他Operator的逻辑。

Volcano模型简单灵活，火山模型将更多的内存资源用于IO的缓存设计而没有优化CPU的执行效率，为什么之前的数据库设计者没有去优化这方面呢？

当时的IO速度是远远小于CPU的计算速度的，那么SQL查询引擎的优化则会被IO开销所遮蔽（毕竟花费很多精力只带来1%场景下的速度提升意义并不大）。

这在当时的硬件基础上是很自然的权衡。

但现在今时不同往日，硬件性能大力发展，在大数据等现代环境场景下，火山模型的弊端逐渐显露。性能表现差强人意。当需要处理的数据量增大时，具有显著的缺陷。

#### 缺点

执行器为了适应复杂的表达式结构，计算一条表达式往往需要引入大量的指令；对于行式执行来说，处理单条数据需要算子树重新进行指令解释（instruction interpretation），从而带来了大量的指令解释开销。

据论文 MonetDB/X100: Hyper-Pipelining Query Execution 统计，在MySQL执行TPC-H测试集的 Query1 时，指令解释就耗费了90%的执行时间。

究其原因。主要有如下几点：

每次 next 都是一次虚函数调用过程是被动拉数据，编译器无法对虚函数进行inline优化，同时也带来分支预测的开销，且很容易预测失败，导致CPU流水线执行混乱。

Volcano Style的代码对数据的局部性并不友好，往往造成cache miss。我们知道CPU cache是存储着连续数据空间，每次可以对连续数据进行集中处理，将受益最大。而Volcano模型每次调用只处理一行。

综上，火山模型的缺点：查询树调用next()接口次数太多，并且一次只取一条数据，CPU执行效率低；而Joins, Subqueries, Order By等操作经常会阻塞。

当然也有优化方式，请参考：[SQL优化之火山模型、向量化、编译执行](https://blog.csdn.net/u011436427/article/details/121805600)。

### 优化

#### 编译执行

考虑到火山模型大量虚函数调用导致的性能损失，推送执行模型(Push Based)很好的解决了这个问题。与拉取模型相反，推送模型自低向上的执行，执行逻辑由底层Operator开始，其处理完一个tuple之后，将tuple传给上层Operator处理。

前面CPU的多级存储介绍提到，数据访问速度最快的是寄存器。

所以在执行查询树时最理想的情况就是数据一直留在寄存器中(假设寄存器的容量足以放下一个tuple)，每个Operator直接处理寄存器中的数据。

Operator之间从拉取模型的虚函数调用，变成了以数据为中心(data-centric)的顺序执行。

当然，并不是所有的Operator的运算逻辑都可以处理完寄存器中的tuple之后，把tuple留在寄存器中，由下一个Operator 接着处理。

例如Join的时候，需要构建hash表，tuple就必须写入内存了(整个hash表当然不可能放到寄存器)。

论文的思想：

每个Operator会根据规则拆分为两个代码块，一块对应Produce() ，一块对应consume()。代码生成的时候就可以根据这个规则生成代码。

Produce()函数负责产生结果tuple；

Consume()函数负责具体的tuple处理逻辑；

好处：编译执行以数据为中心，消灭了火山模型中的大量虚函数调用开销。甚至使大部分指令执行，可以直接从寄存器取数，极大的提高了执行效率。

## 物化模型

### 概述

物化模型（Materialization Model）的处理方式是：每个operator一次处理所有的输入，处理完之后将所有结果一次性输出。

物化模型更适合OLTP负载，这些查询每次只访问小规模的数据，只需要少量的函数调用。

### 原理

### 特点

火山模型每次处理一行数据，物化模型每次处理全部的数据，虽然确实减少了大量函数调用开销，但是不可避免的会引起其他问题。一个是需要存储全部数据到内存中，很容易引起oom；另外一个是执行书的节点会强制转变成串行执行，多核下无法充分利用cpu。

#### 优点

#### 缺点

## 向量化/批处理模型

参考：

<https://blog.csdn.net/qq_35423190/article/details/123129172>

向量化执行引擎简介：<http://mysql.taobao.org/monthly/2017/01/06/>

### 概述

批处理模型是火山模型和物化模型的折衷。

鉴于火山模型每次处理一行一行数据，而next调用代价又比较高。物化模型又过于极端，有oom风险，所以批量处理模型在业界被提出。

向量化模型和火山模型类似，每个operator需要实现一个next()函数，但是每次调用next()函数会返回一批的元组（tuples），而不是一个元组，所以向量化模型也可称为批处理模型。

在算子间传递数据不再是一条一条记录，而是一批数据，算子每次执行的时候都会在内部攒一批数据，数据大小尽可能和CPU cache对齐，不仅大大提高了cache命中率，而且有效了减少了函数调用次数。

Presto、snowflake、SQLServer、Amazon Redshift等数据库支持这种处理模式。

Spark 2.x 的 SQL 引擎开始也支持向量化执行模型。

向量化执行依然采用类似火山模型的拉取式模型，唯一的区别是其Operator的next()函数每次返回的是一批数据(如1000行)。

一般向量化特指列式存储系统中，按列聚合的一组数据。

### 原理

#### 向量数据结构

#### 向量化原语与向量化计算

#### 代码生成技术

### 优化

#### 编译执行

## push模型/ pull模型

考虑火山模型的每个节点。

一般来说，每个处理节点都有两个通道，一个入口，负责接收子节点的数据；一个出口，负责输出给上层节点处理后的值。

那么每量个处理节点（父子节点），都可以看做是一个生产者消费者模型。

对于消费者而言，有两种方式获取信息：

推模型push：由消息中间件主动将消息推送给消费者；可以尽可能快地将消息发送给消费者，但是若消费者的处理消息的能力较弱（一条消息长时间处理），中间件会不断地向消费者push消息，消费者的缓冲区可能会溢出；

拉模型pull：由消费者主动向消息中间件拉取消息；会增加消息的延迟，即消息到达消费者的时间变长。

push模型比pull模型复杂，但cpu利用率要高于pull模型。由于子算子产生的结果会直接 Push 给父算子进行操作，Push 模型的 Context switch 相对较少，对 CPU Cache 的友好性也更强。

但是使用push模型会不可避免的产生其他问题。如果使用pull模型，那么使用一个线程就可以完成整个sql的执行流程；但是换成push模型，每个节点都会自发运行往父节点推数据，那么一个sql就需要使用多个线程来完成，cpu的利用率肯定是上去了，但是如果存在高并发场景，并发执行sql量很多，那么线程数也会暴增。

所以使用需要考量适度性。

# 分布式数据库执行器

## Spanner

## OceanBase

## TiDB

## TDSQL

在TDSQL数据库中，TDSQL2.0采用的是火山模型，内部是采用的一种邮箱（Mailbox）的方式通知。

TDSQL3.0采用的是向量化模型。

## GoldenDB