# 概述

参考：

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

## codegen/查询编译/编译执行

参考：

<https://zhuanlan.zhihu.com/p/508017564>

<https://blog.csdn.net/u011436427/article/details/122857260>

<https://zhuanlan.zhihu.com/p/508017564>

<https://code84.com/751721.html>

<https://zhuanlan.zhihu.com/p/566818479>

## 向量化

# 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

玩转SIMD指令编程：<https://zhuanlan.zhihu.com/p/591900754>

# 执行模型

## 迭代模型/火山模型

参考：

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

MySQL8.0执行器及改进：<https://www.modb.pro/db/58408>

### 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

#### MySQL5.7迭代器

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，优化最终生成了 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

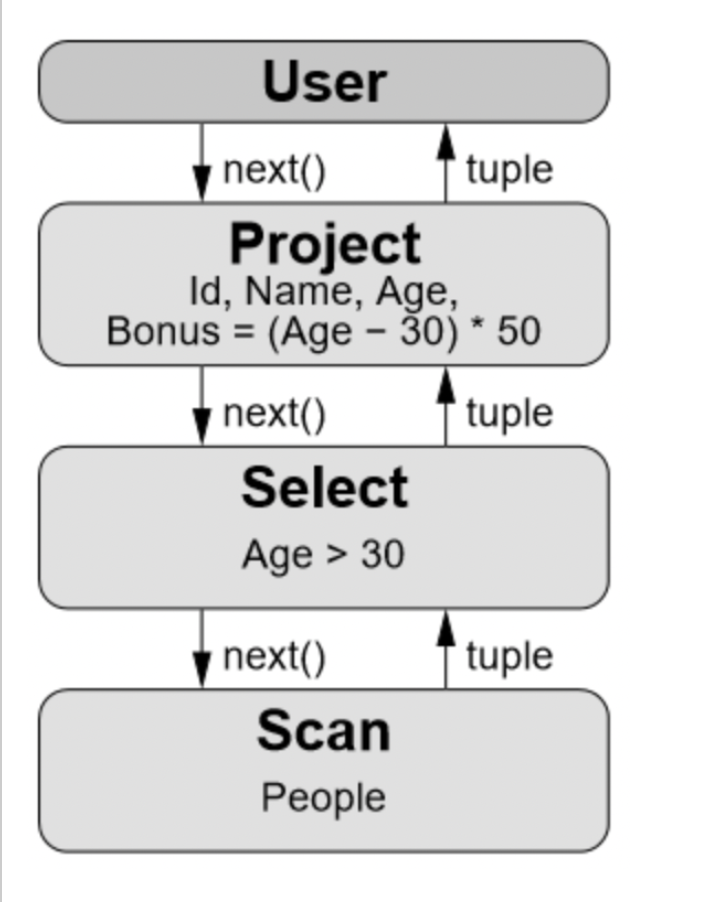
因此，查询执行时会由查询树自顶向下的调用next()接口，数据则自底向上的被拉取处理。这种处理方式也称为拉取执行模型（Pull Based）。例如以下SQL:

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。

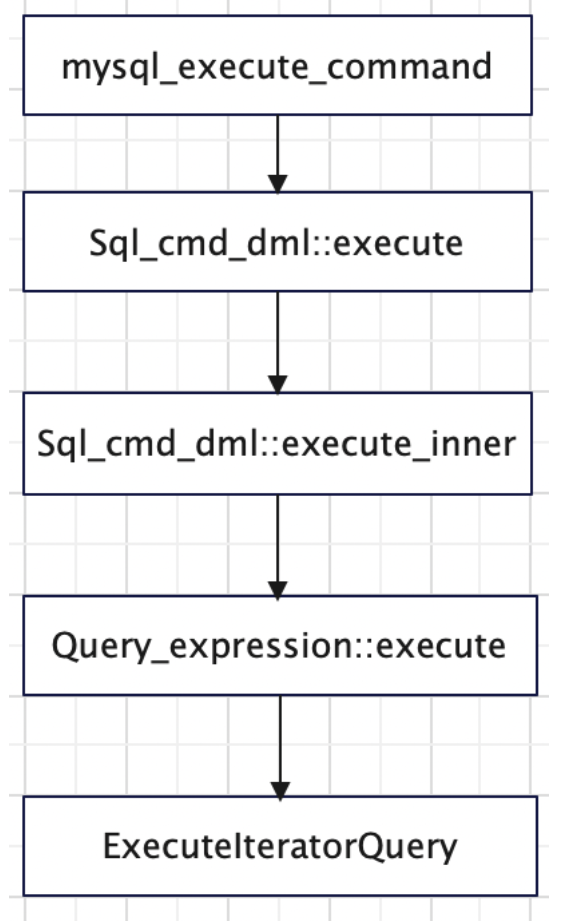
read()：读取一行，将行放入记录缓存中

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/>

为什么需要向量化执行引擎：<https://zhuanlan.zhihu.com/p/587568943>

### 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

参考：

<https://zhuanlan.zhihu.com/p/88436962>

##### Code generation

Code generation（代码生成）是指将高级编程语言或其他形式的抽象表示转换为可执行代码的过程。它是编译器和解释器等软件工具中的重要组成部分。Code generation 的原理可以简要概括为以下几个步骤：

解析（Parsing）：首先，将源代码转换为抽象语法树（Abstract Syntax Tree，AST）。解析器会对源代码进行词法分析和语法分析，识别语言结构并构建AST。

语义分析（Semantic Analysis）：在这个阶段，编译器会检查代码的语义正确性，包括类型检查、符号表构建和作用域解析。语义分析器会根据语言规范和编程语言的语义约束来验证代码。

中间表示（Intermediate Representation）：生成中间表示是为了方便进行代码优化和目标代码生成。常见的中间表示形式有三地址码（Three-Address Code）、控制流图（Control Flow Graph）等。

代码优化（Code Optimization）：在生成最终代码之前，编译器会进行一系列的代码优化操作，以提高程序的性能和效率。代码优化的目标包括减少执行时间、减少内存占用和简化代码结构等。

目标代码生成（Target Code Generation）：最后一步是将优化后的中间表示转换为目标平台上的机器代码或字节码。这涉及到指令选择、寄存器分配和内存管理等过程，以生成可执行的目标代码。

需要注意的是，不同的编译器和解释器可能采用不同的实现策略和技术。具体的 code generation 过程可能因编程语言、编译器类型和目标平台而有所差异。

数据库 Code Generation（代码生成）是指根据数据库模式或数据定义语言（DDL）的描述，自动生成与数据库操作相关的代码。它可以减少手动编写重复性、繁琐的数据库访问代码的工作量，提高开发效率。

数据库Code Generation的原理可以简述如下：

数据库模式解析：首先，解析数据库的模式，包括表、列、关系和约束等信息。这可以通过读取数据库的元数据（metadata）或解析DDL语句来实现。

代码模板定义：定义用于生成代码的模板。代码模板可以包含占位符（placeholder）或变量，用于将具体的数据库模式信息插入到生成的代码中。

代码生成：根据定义的代码模板，将数据库模式信息填充到相应的位置，生成最终的代码。生成的代码通常包括数据库连接、查询、更新、删除等操作的代码片段。

可选的扩展功能：生成的代码可能还包含与业务逻辑相关的功能，例如数据校验、权限控制或业务规则的处理等。

代码输出：生成的代码可以以文件或字符串的形式输出，供开发人员在应用程序中使用。

数据库Code Generation可以应用于多种编程语言和数据库管理系统，常见的工具和框架有Hibernate、Entity Framework、CodeSmith、MyBatis Generator等。这些工具提供了配置选项和定制化功能，使开发人员能够根据自己的需求生成符合特定环境和标准的数据库操作代码。

##### Code explosion

静态类型的代码爆炸（Static Type Code Explosion）是指在使用静态类型语言时，由于类型系统的限制，导致在编写代码时需要显式声明和处理大量的类型信息，从而导致代码量急剧增加的现象。

在静态类型语言中，变量、函数参数和返回值等都需要显式声明它们的类型。这样的要求可以提供类型安全性和编译时错误检查，但也会增加代码的复杂性和冗余性。

静态类型代码爆炸可能出现的几种情况包括：

类型重复声明：在静态类型语言中，变量和函数通常需要显式声明它们的类型。如果同一个类型需要在多个地方声明，就会导致代码冗余和可读性下降。

类型转换和强制类型声明：当需要将一个类型转换为另一个类型时，可能需要进行显式的类型转换操作，或者使用类型断言等机制来强制指定类型。这些额外的类型转换代码会增加代码量，并且可能引入错误。

泛型参数特化：某些静态类型语言支持泛型编程，但在使用具体类型参数时需要进行特化。这可能导致需要为每个特定类型编写额外的代码，从而增加了代码量。

为了应对静态类型代码爆炸的问题，可以考虑以下方法：

使用更高级别的语言功能：某些静态类型语言提供了高级的类型推导、泛型推断等功能，可以减少显式类型声明的需求，从而减少代码量。

使用元编程技术：通过使用元编程技术，可以在编译时或运行时生成代码，减少手动编写大量重复的类型相关代码的工作。

使用代码生成工具：利用代码生成工具或框架，根据模板或配置文件自动生成大量类型相关的代码，减少手动编写的工作量。

总体而言，静态类型语言在提供类型安全和编译时检查等优势的同时，可能会导致静态类型代码爆炸的问题。选择适当的语言功能、采用元编程技术或使用代码生成工具，可以帮助减轻代码爆炸带来的负担。

### 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。

那么每两个处理节点（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

### 并行执行

### 向量执行

参考：

<https://www.oceanbase.com/news/ly37vv>

#### 概述

与Oracle和SQL Server等数据库系统类似，OceanBase的用户场景除了OLTP类的简单查询，还有报表分析、业务决策等复杂OLAP查询。很多用户希望在完成在OLTP联机事务处理的同时，提供连接查询、聚合分析等OLAP分析能力。而OLAP查询具有数据处理量大、计算查询复杂、耗时高的特点，对数据库的SQL执行引擎的执行效率要求较高。

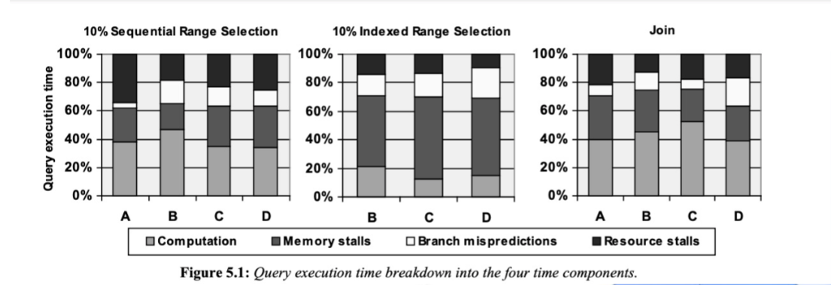
早期我们通过并行执行技术将数据均匀分摊到分布式系统中多个CPU上，通过降低每个CPU处理的数据量实现查询响应时间（RT）降低。随着用户数据量的不断增多，在不增加计算资源的前提下，每个CPU计算数据的量也不断增大。我们在客户现场发现，OLAP场景下部分特殊情况的CPU利用率接近100%。这在聚合分析、连接查询等大数据量分析查询中变得尤为明显。

如何提高数据库单核计算性能，降低查询响应时间（RT）对客户至关重要。为帮助客户解决HTAP混合负载下数据查询效率难的问题，OceanBase引入向量化技术，并完全自主设计了向量化查询引擎，极大地提高了CPU单核处理性能，实现了HTAP场景下复杂分析查询性能的10倍提升，并在TPC-H测试（数据分析型基准测试，业界公认衡量数据库数据分析能力的权威标准）中得到了充分验证。

##### 传统火山模型存在的问题

在详细介绍向量化引擎特点前，我们先了解一下火山模型以及火山模型存在的典型问题。在数据库发展早期，由于IO速度低下、内存和CPU资源非常昂贵，为了避免爆内存的情况出现，每次只计算一行数据的火山模型成为了经典的SQL计算引擎。火山模型又叫迭代器模型，正式提出是在1994年论文《Volcano—An Extensible and Parallel Query Evaluation System》。早期很多关系型数据库都在使用火山模型，如Oracle、Db2、SQLServer、MySQL、PostgreSQL、MongoDB等。

火山模型应用十分广泛，但这种设计并**没有充分利用CPU的执行效率**，**进行Joins、Subqueries、Order By等复杂查询操作时也经常会产生阻塞**。论文《DBMSs On A Modern Processor: Where Does Time Go?》在微观层面分析了数据库系统在现代CPU框架下的主要消耗细节。数据库在顺序扫描、索引扫描和连接查询三个典型查询场景下，可以很明显的看到CPU真正用在计算上的占比不超过50%。相反，等待资源（Memory / Resource Stalling）占比非常高（平均 50%）。加上分支预测失败的代价，很多场景下CPU真正用来计算的比率往往大幅低于50%。例如Index Scan索引扫描下CPU计算最低占比小于20%，无法真正发挥CPU的最大能力。



##### 向量化引擎理论的诞生

2005年，一篇题为《MonetDB/X100: Hyper-Pipelining Query Execution》的论文首次提出“向量化引擎”的概念。不同于传统的火山模型按行迭代的方式，向量化引擎采用批量迭代方式，可以在算子间一次传递一批数据。换句话说，向量化实现了从一次对一个值进行运算，到一次对一组值进行运算的跨越。

向量化引擎的技术价值：

###### 批量返回数据，函数调用少，提升Cache友好性

为了更好地提高CPU利用率，减少SQL执行时的资源等待（Memory/Resource Stall)，向量化引擎被提出并应用到现代数据库引擎设计中。

与数据库传统的火山模型迭代类似，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next调用一次传递一行数据，向量化引擎一次传递一批数据，并**尽量保证该批数据在内存上紧凑排列**。由于数据连续， CPU可以通过**预取指令**快速把数据加载到**level 2 cache**中，**减少memory stall 现象**，从而提升CPU的利用率。其次由于数据在内存上是紧密连续排列的，可以通过SIMD指令一次处理多个数据，充分发挥现代CPU的计算能力。

向量化引擎大幅减少了框架函数的调用次数。假设一张表有1亿行数据，按火山模型的处理方式需要执行1亿次迭代才能完成查询。使用向量化引擎返回一批数据，假设设置向量大小为1024，则执行一次查询的函数调用次数降低为小于10万次（1亿/1024 = 97657），大大降低了函数调用次数。在算子函数内部，函数不再一次处理一行数据，而是通过循环遍历的方式处理一批数据。通过批量处理连续数据的方式**提升CPU DCache和ICache的友好性，减少Cache Miss**。

说明：

DCache（Data Cache）和ICache（Instruction Cache）是计算机体系结构中的两种高速缓存（Cache）。

DCache（数据缓存）是用于存储从主存（RAM）中读取的数据的缓存。当处理器需要读取数据时，首先会检查DCache中是否存在所需的数据。如果存在（命中），则可以直接从DCache中获取数据，这比从主存中读取数据要快得多。如果DCache中不存在所需的数据（未命中），则需要从主存中加载数据到DCache中，以供后续访问使用。

ICache（指令缓存）则是用于存储指令的缓存。当处理器执行程序时，需要按照指令序列从内存中获取指令并执行。ICache的作用是在执行过程中预先缓存一部分指令，以加速指令的获取和执行。类似于DCache，当处理器需要执行指令时，首先会检查ICache中是否存在所需的指令。如果存在（命中），则可以直接从ICache中获取指令。如果ICache中不存在所需的指令（未命中），则需要从内存中加载指令到ICache中。

DCache和ICache的存在主要是为了解决处理器与主存之间的速度差异。处理器的运算速度远快于主存的访问速度，而Cache作为位于二者之间的高速缓存，可以提供更快的数据访问速度，减少处理器等待数据的时间。

由于数据访问和指令执行是计算机程序执行的核心操作，DCache和ICache的设计和优化对于提高计算机系统的性能非常重要。合理的Cache设计可以减少对主存的访问次数，减少访存延迟，提高程序的执行效率。

###### 减少分支判断提升CPU流水处理能力

论文《DBMSs On A Modern Processor: Where Does Time Go?》还介绍了分支预测失败对数据库性能的影响。由于CPU中断了流水执行，重新刷新流水线，因此分支预测失败对数据库处理性能的影响很大。SIGMOD13 的论文《Micro Adaptivity in Vectorwise》也对分支在不同选择率下的执行效率有详细论述（下图）。

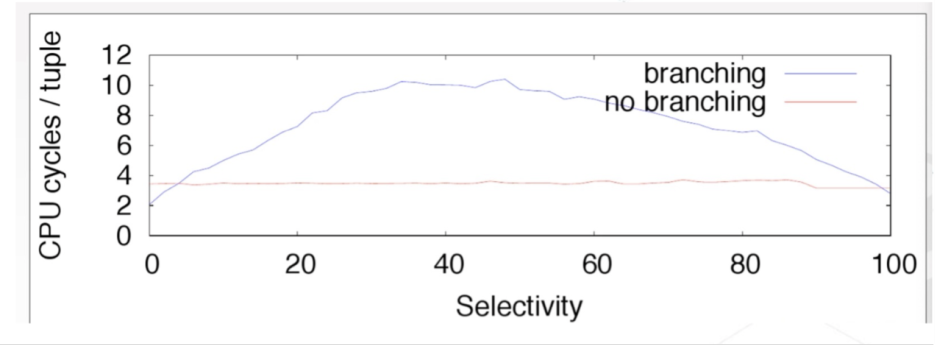


图3 分支对执行的影响

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。

参考：

<https://new.qq.com/rain/a/20220823A0AGRO00>

###### SIMD 指令加速计算

由于向量引擎处理内存连续数据，因此向量引擎可以很方便的把一批数据装载到向量寄存器中。然后通过SIMD指令，替换传统的标量（scalar）算法，进行向量（Vector）计算。需要说明的是SIMD指令CPU架构密切相关，在 X86，ARM，PPC上都有相应的指令集。目前以Intel x86 架构指令最为丰富，下图4给出了x86下各个SIMD指令的推出时间和其支持的数据类型。更详细的信息可以查看Intel的官方手册。

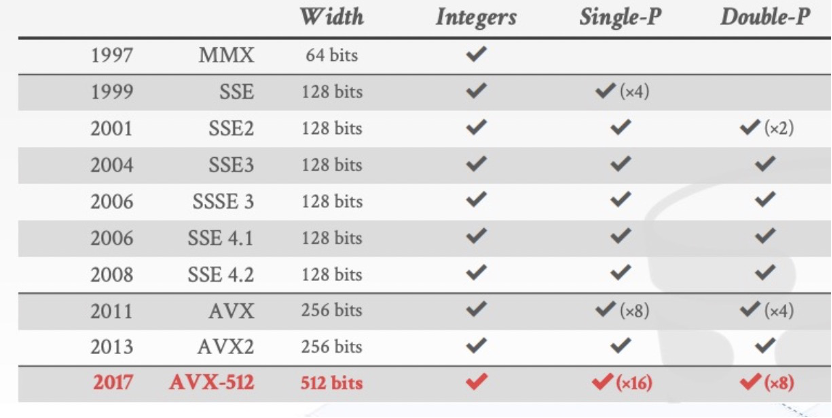


图4 Intel Intrinsic 指令支持的数据类型

说明：

SIMD（Single Instruction, Multiple Data）加速是一种并行计算技术，旨在通过同时对多个数据元素执行相同的操作来加速计算过程。它利用向量寄存器和特定指令集，使得单条指令能够同时处理多个数据元素，从而提高计算效率。

SIMD加速常用于多媒体处理、图像处理、信号处理等需要对大量数据进行相同类型的操作的应用场景。以下是SIMD加速的一些关键概念和原理：

向量寄存器：SIMD技术使用向量寄存器，它可以容纳多个数据元素（如浮点数、整数等）。这样的向量寄存器与普通的标量寄存器不同，可以在一次操作中同时处理多个数据元素。

SIMD指令集：处理器需要支持SIMD指令集，如Intel的SSE（Streaming SIMD Extensions）和AVX（Advanced Vector Extensions），ARM的NEON等。这些指令集提供了特定的指令，可以对向量寄存器中的多个数据元素执行相同的操作，从而实现并行计算。

数据并行性：SIMD加速利用数据并行性，即对相同类型的操作应用于多个数据元素。通过一次性处理多个数据元素，可以显著提高计算效率。

数据对齐：为了使SIMD指令能够高效地操作向量寄存器中的数据，数据需要按照特定的对齐要求进行排列。对于特定的SIMD指令集，通常要求数据在内存中以特定的字节对齐方式进行存储。

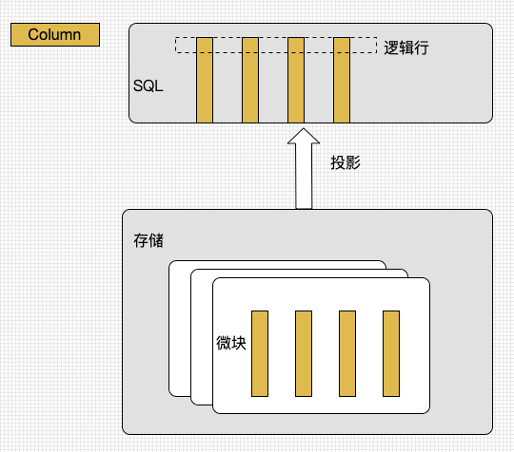
编译器优化：编译器可以识别适合使用SIMD加速的代码模式，并自动将其转换为SIMD指令。编译器优化对于充分发挥SIMD加速的性能至关重要。

通过使用SIMD加速，可以在处理大量数据时显著提高计算效率。然而，SIMD加速的适用性取决于具体的应用场景和数据操作模式。在一些情况下，数据之间的依赖关系或非规则的数据访问可能限制了SIMD的有效性。因此，在应用SIMD加速时，需要评估数据的并行性和指令级并行度，以确保能够获得实际的性能提升。

#### 原理

##### 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。

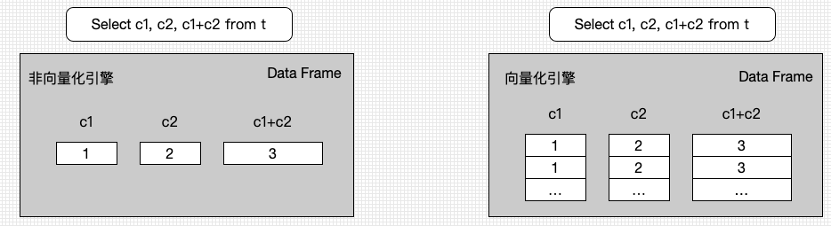


图6 OceanBase SQL 引擎内存编排

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。

###### 过滤标识设计

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：

1、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；

2、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

算子的向量化是OceanBase向量化引擎的重要工作。在向量化引擎中，所有查询算子都按照向量化引擎的特点进行了新的设计实现。按照向量化引擎的设计原则，每个算子都通过向量接口向下层算子拿一批数据，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。由于算子实现众多，这里重点介绍Hash Join和Sort Merge Group By 2个典型实现，其它算子不再一一赘述。

Hash Join

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。

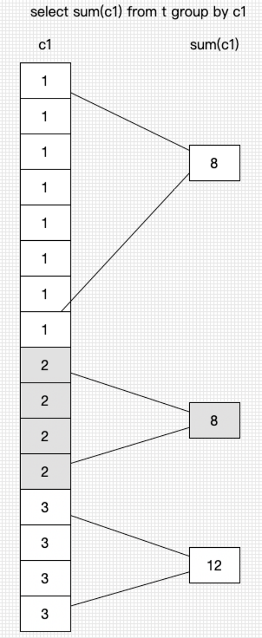


图7 Sort Merge Group By向量化实现

## TiDB

## TDSQL

### 火山模型

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。

### 并行执行

### 向量化执行

TDSQL3.0采用的是向量化模型。

## PolarDB-X

### 向量化

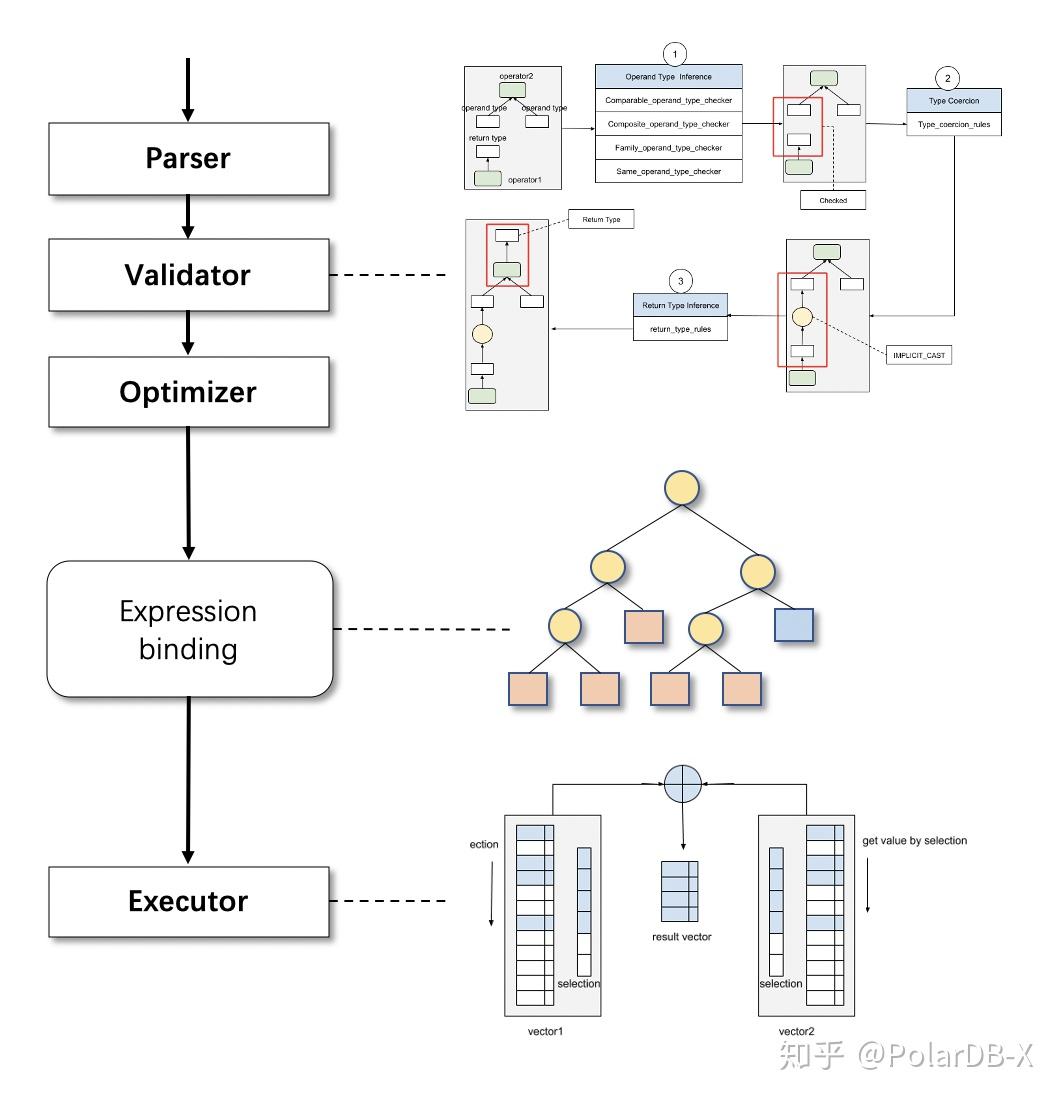
#### 主要流程

PolarDB-X中，向量化表达式的执行分为以下几个阶段：

1、用户SQL经解析后，在validator中进行校验，推导和修正表达式的类型信息；这一阶段为向量化运算提供正确的、静态的类型信息；

2、在优化器形成执行计划之后，需要对表达式树进行表达式绑定，实例化对应的向量化原语，同时分配好向量下标，供运行时内存分配；

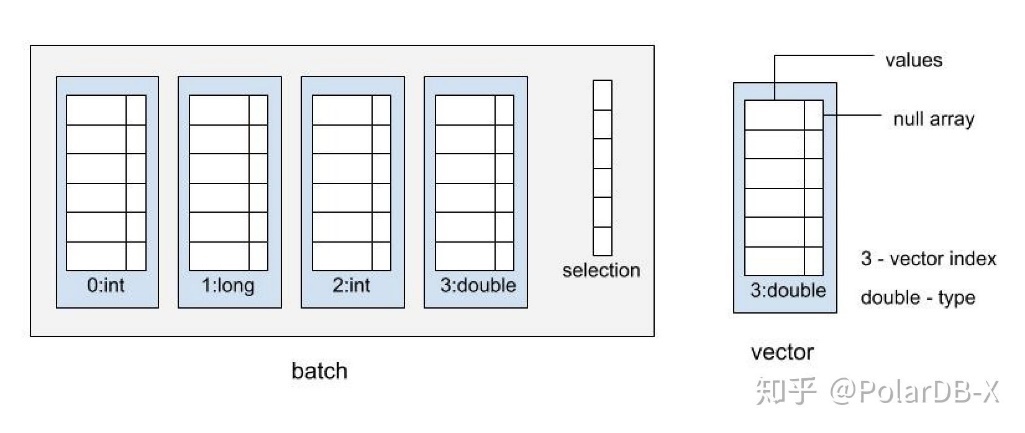
3、执行阶段，依据Volcano式的结构，自顶向下的触发执行向量化原语，并将向量作为运行时数据结构。



#### 概述

##### 运行时数据结构

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。

##### 向量化原语

向量化原语是向量化执行系统中的执行单位，它最大程度限制了执行期间的自由度。原语不用关注上下文信息，也不用在运行时进行类型解析和函数调用，只需要关注传入的向量即可。它是类型特定（Type-Specific）的，即一类原语只能处理特定类型。向量化原语的主体是Tight-Loop的代码结构。在一个循环体内部，只需要进行取值和运算即可，没有任何的分支运算和函数调用。一个简单的向量化原语结构如下所示：

map\_plus\_double\_col\_double\_col(int n,

double\*\_\_restrict\_\_ res,

double\*\_\_restrict\_\_ vector1, double\*\_\_restrict\_\_ vector2,

int\*\_\_restrict\_\_ selection)

{

if (selection) {

for(int j=0;j<n; j++) {

int i = selection[j];

res[i] = vector1[i] + vector2[i];

}

} else {

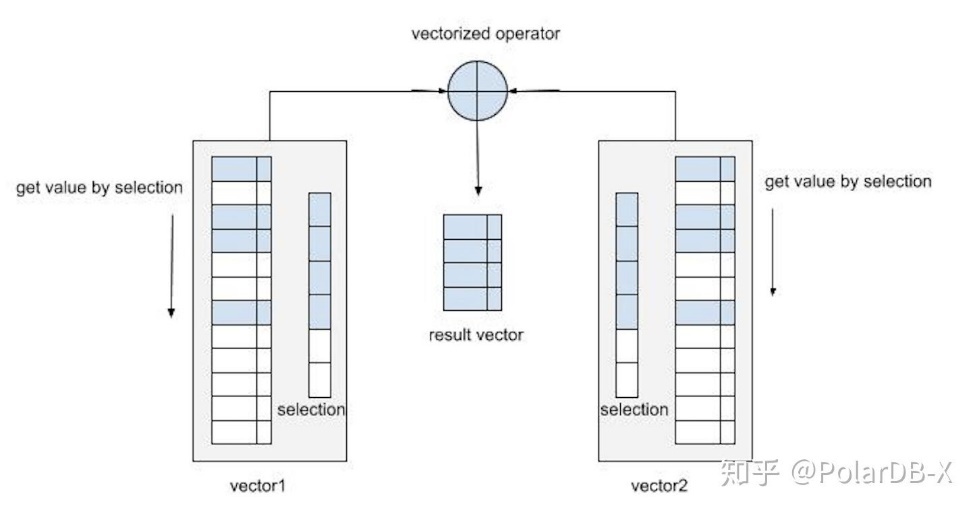
for(int i=0;i<n; i++)

res[i] = vector1[i] + vector2[i];

}

}

其运算过程利用了selection数组，逐步对向量进行取值、运算和存值，如下图所示：



向量化原语带来了以下优点：

Type-Specific以及Tight-Loop的结构，大大减少了指令解释的开销；避免分支预测失败和虚函数调用对CPU流水线的干扰，同时也能有利于loop pipeline优化从向量中存取数据，有利于触发cache prefetch，减少cache miss带来的开销。

我们为各种标量化表达式提供相应的原语实现，从而完成从标量到向量化的转变。例如将加法运算plus(Object, Object) 针对不同操作数类型生成原语，包括plus(double，double)，plus(long, long)等。

##### 短路求值

在向量化原语的基础上，我们可以进一步对分支运算（也称为控制流运算 Control-Flow）进行短路求值（short-circuit calculation）优化，提升表达式计算的性能。 例如，case 表达式由n个when表达式、n-1个then表达式、1个else表达式构成。对于表达式

select case when a > 1 then a \* 2

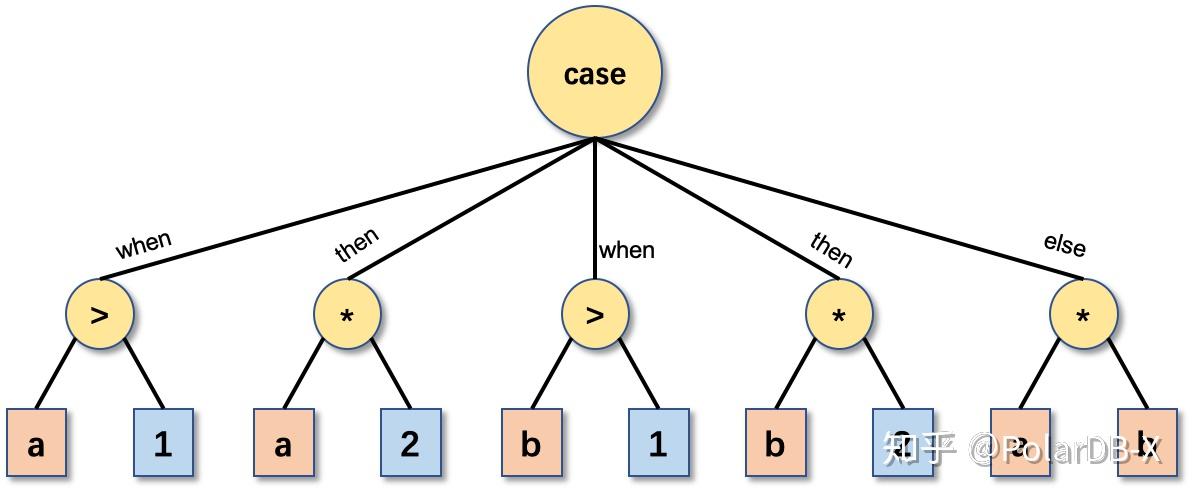
when b > 1 then b \* 2

else a \* b

其逻辑语义是：

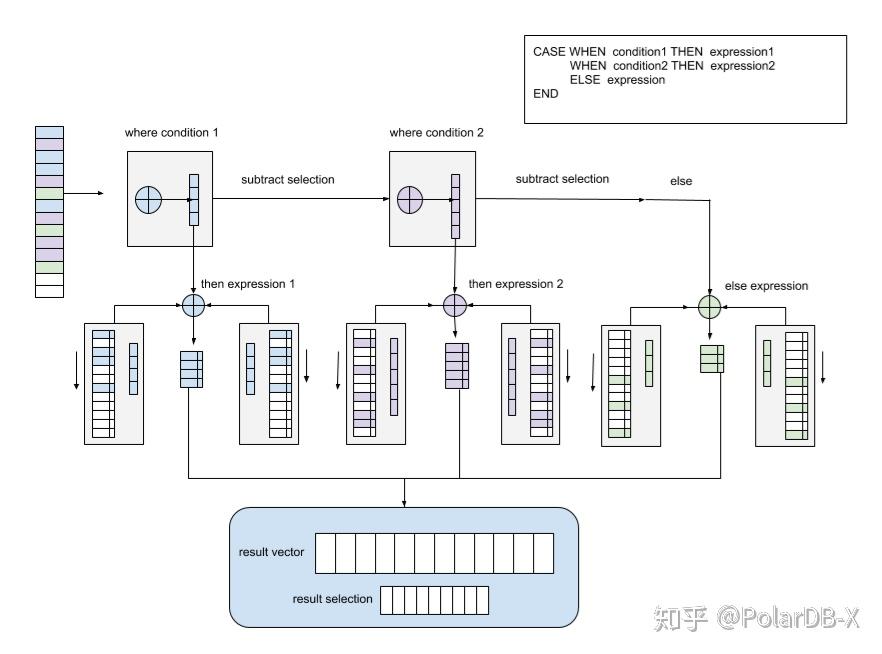
对于满足 a > 1 的向量位置，计算 a \* 2；对于满足 a <= 1 and b > 1 的向量位置，计算 b \* 2；对于满足 a <= 1 and b <= 1 的向量位置，计算 a \* b；把所有位置的数值组合在一起形成新的向量，输出。

具有以下树形结构：



由于标量化表达式按照volcano结构编排，并提供了统一的next()的接口，case表达式必须执行完所有的子表达式a>1，a2，b>1，b2和a\*b之后，将全部结果汇总到一起，最后做case语义处理。这种执行方式不能根据when表达式的处理结果及时终止计算过程，而是对全部子表达式无差别执行。

引入向量化执行器以后，我们可以设计短路求值来优化此问题，每一个子表达式需要被提供合适的selection数组，从而正确选择列中合适的位置来进行向量运算。设第i个when条件表达式接受的selection元素集合为Si，其输出的selection元素集合为Ri，也就是第i个then条件表达式接受的selection元素集合。那么满足Si=S1\R1\...\Ri-1，其中S1是原始的selection数组中的下标集合。我们把求取selection元素集合的步骤称为substract selection，case 运算的整个过程如下图所示：



##### 总结

PolarDB-X向量化引擎利用原语（primitive）来构建表达式，以向量作为运行时数据结构。每种原语仅为特定类型进行服务，从而减少了指令总数；原语中的tight-loop结构不仅对CPU流水线十分友好，也允许CPU进行数据预取，并且避免分支预测。此外，一些优化如延迟物化、短路求值，进一步提升了表达式求值性能。 然而，从用户SQL到向量化执行之间，存在着一道巨大的鸿沟。我们需要解决以下几个重要问题：

1、如何确定表达式的输入输出类型，并为SQL中的表达式分配合适的原语？

2、每个原语需要使用不同的向量来进行输入和输出，如何为正确地为原语分配向量？

3、每种原语仅为特定类型进行服务，那么我们必然需要为一个表达式配备大量不同的原语，来适应不同的数据类型。如何应对原语数量爆炸这一问题？

#### 原理

PolarDB-X引入了表达式绑定、静态类型系统、代码模板等多种技术来解决上述问题。

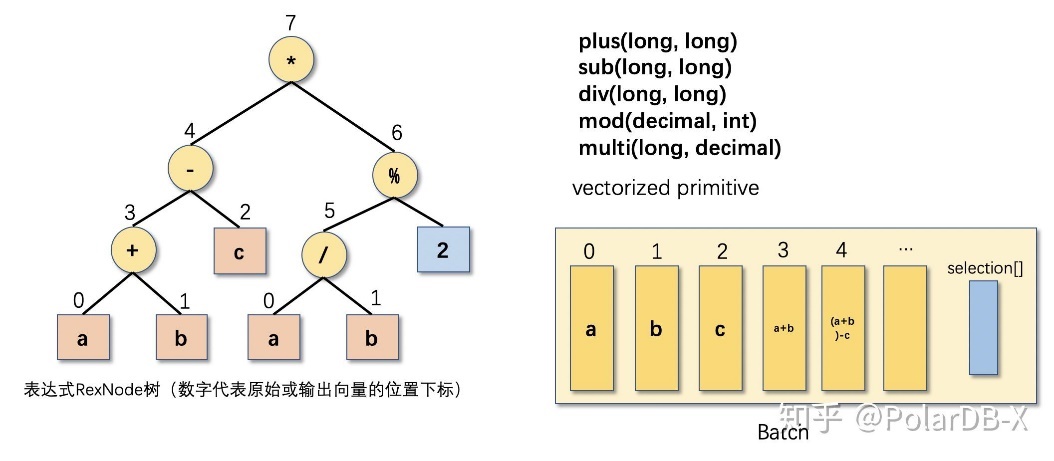
##### 表达式绑定

在优化器完成优化阶段之后，我们从中得到代表着表达式树形结构的RexNode树，对其进行表达式绑定操作。表达式绑定需要对表达式树进行前序遍历，并完成以下工作：

1、对于遍历到的函数调用节点，提取函数签名信息，通过函数签名进行反射，实例化向量化原语；

2、在遍历过程中进行将遍历到的RexNode节点（包括列、表达式调用、常量等）按遍历顺序分配向量位置下标。下标分配完成后，在运行时依据下标信息统一进行内存分配，供表达式输入、输出使用。

下图是表达式 ((a+b)-c)\*((a/b)%2) 的表达式树结构，以及通过表达式绑定得到的向量化原语实例，和分配得到的Batch结构：



此外，通过在表达式绑定时分配输入、输出向量的位置下标，我们可以灵活的处理各类数据依赖问题，包括以下的数据依赖场景：

1、多个表达式输出到一个向量中，例如case表达式中的各个then、else子表达式。可以给这些子表达式分配相同的输出向量Index；

2、多个表达式将同一个向量作为输入。例如select fun\_1(expr), fun\_2(expr)，expr表达式输出的结果可以作为fun\_1和fun\_2表达式的输入。可以通过分配相同的输入向量Index来实现。

3、某些条件下，作为输出的中间结果向量可以覆盖掉作为输入的中间结果向量。

##### 静态类型系统

在基于行的执行器中，类型系统的静态绑定并不是必须的。例如在 MySQL 中，表达式构成一个树状结构，上层的表达式结构通过调用下层提供的不同返回值类型的接口（例如：val\_int()、val\_decimal()、val\_str()等），递归地计算出最终结果。这种方式实现简单，但是直到运行时才能确定表达式的输入返回类型，进而决定需要调用的计算函数，效率比较低。

向量化系统则要求静态类型系统。在解析器和优化器完成执行计划的构建之后，必须保证每个表达式的类型是正确的、不需要运行时确定的。只有这样，才能为它实例化正确的向量化原语、分配正确类型的向量。

PolarDB-X将用户传来的SQL解析为AST之后，对于树形的AST需要自顶向下地进行类型推导，确定整个表达式树的类型信息。具体过程包括：

1. 操作数类型检查（Operand Type Checker）。

子表达式的返回类型，会作为父表达式的操作数类型。每个表达式配备有相应的操作数类型检查规则，通过此规则来检查操作数类型是否合法；

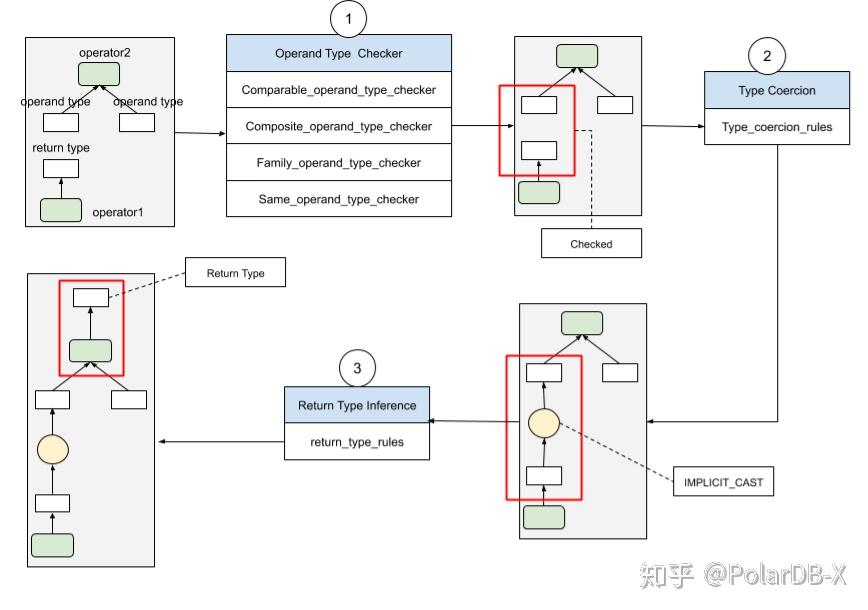
1. 隐式类型转换（Type Coercion）。

当子表达式的返回类型不能成为合法的父表达式操作数类型时，我们需要调用相应的类型转换规则，尝试进行返回值类型return type到操作数类型operand type的转换。办法是，生成一个合法的IMPLICIT\_CAST表达式，将return type强制转换为合法的operand type类型。由于此转换对于用户来说是透明的，所以称为隐式类型转换。

1. 返回值类型推导（Return Type Inference）。

当表达式具备了合法的操作数之后，可以调用相应的返回值推导规则，通过操作数推出正确的返回值类型。

以上三个步骤如下图所示：



##### 代码生成技术

PolarDB-X采用Apache Freemarker框架，根据代码模板来批量生成向量化原语的源码，避免了Type-Specific引入的代码量激增的问题。原语=代码模板X类型配置，原语就是由代码模板，配合以不同的类型，在项目编译时批量生成的。

一个简化后的原语代码模板如下所示（省略了selection数组、VectorSlot等处理），其中${} 符号代表在编译时将要被替换成特定类型和表达式。

public class ${className} {

public ${className}(int outputIndex, VectorizedExpression[] children) {

super(DataType.${type.outputDataType}Type, outputIndex, children);

}

@Override

public void eval(EvaluationContext ctx) {

super.evalChildren(ctx);

VectorBatch batch = ctx.getVectorBatch();

${type.inputType1}[] array1 = ((${type.inputVectorType1}) leftInputVectorSlot).${type.inputType1}Array();

${type.inputType2}[] array2 = ((${type.inputVectorType2}) rightInputVectorSlot).${type.inputType2}Array();

${type.outputType}[] res = ((${type.outputVectorType}) outputVectorSlot).${type.outputType}Array();

for (int i = 0; i < batchSize; i++) {

int j = sel[i];

res[j] = (${type.outputType})array1[j] ${operator.op} (${type.outputType})array2[j];

}

}

}

Freemarker模版框架会帮助我们生成出可以编译的Java代码，例如对于 AddLongLong 这个函数（BIGINT列与BIGINT列相加），会生成以下代码：

public class AddLongColLongColVectorizedExpression extends AbstractVectorizedExpression {

public AddLongColLongColVectorizedExpression(int outputIndex, VectorizedExpression[] children) {

super(DataType.LongType, outputIndex, children);

}

@Override

public void eval(EvaluationContext ctx) {

super.evalChildren(ctx);

VectorBatch batch = ctx.getVectorBatch();

long[] array1 = ((LongVectorSlot) leftInputVectorSlot).longArray();

long[] array2 = ((LongVectorSlot) rightInputVectorSlot).longArray();

long[] res = ((LongVectorSlot) outputVectorSlot).longArray();

for (int i = 0; i < batchSize; i++) {

long right = (long)array2[i];

res[i] = ((long) array1[i]) + (long) right;

}

}

}

这些生成的函数代码会在编译期间由编译器连同其他代码共同编译、打包，运行时无需编译，只要将对应类型的函数绑定到执行计划中，即可调用执行。

## GoldenDB