# 概述

参考：<https://blog.csdn.net/baichoufei90/article/details/84328666>

<https://baijiahao.baidu.com/s?id=1717147765473335948&wfr=spider&for=pc>

MPP，全称为Massively Parallel Processor，翻译过来就是大规模并行处理。MPP系统是由许多松耦合的处理单元组成的（要注意的是这里指的是处理单元而不是处理器）。每个处理单元内的CPU都有自己私有的资源，如总线，内存，硬盘等，且都有操作系统和管理数据库的实例复本。这种结构最大的特点在于不共享资源(share-nothing)。

MPP处理数据的思路：

面对海量数据和计算时，采用大事化小的思路，对数据进行分割，数据分割后单独存储，数据处理消耗的资源也是相互隔开的，对于MPP数据库来讲，整个数据库由多个完全独立的数据库构成，各个拥有完整的数据存储、数据管理、数据操作能力。基于网络实现节点互联，形成一个整体对外提供服务，节点间互不干扰，即Share Nothing，不共享磁盘和计算能力。

MPP架构有如下特点：

Share Nothing、节点之间数据不共享，只有通过网络连接实现的协同。

每个节点有独立的存储和内存。

数据根据某种规则(如Hash)散布到各个节点。

计算任务也是会发布到各个节点并行执行，最后再将结果聚合到整体返回。

用户使用时会看做整体。

MPP数据库（如GreePlum）往往优先考虑C一致性，然后是A可用性，最后考虑P分区容忍。

MPP架构目前被并行数据库广泛采用，一般通过scan、sort和merge等操作符实时返回查询结果。

## 优点

任务并行执行；

数据分布式存储(本地化)；

分布式计算；

高可用、易维护：数据通过副本提供冗余保护，自动故障探测和管理，自动同步元数据和业务数据。提供图形化工具，以简化管理员对数据库的管理工作；

高并发：读写不互斥，支持数据的边加载边查询，单个节点并发能力大于300 用户；

高扩展、高可靠：支持集群节点的扩容和缩容，支持全量、增量的备份/恢复；

行列混合存储：提供行列混合存储方案，从而提高了列存数据库特殊查询场景的查询响应耗时；

标准化：支持SQL92标准，支持C API、ODBC、JDBC、ADO.NET等接口规范。

## 缺点

1、很难高可用 -> 影响可用性和可靠性

因为数据按某种规则如HASH已经散布到了各个节点上。

2、节点数=任务并行数 -> 影响扩展性

一个作业提交时，每个节点都要执行相同任务。而不像MapReduce那样做了根据实际开销进行任务拆分后散发到有资源的几个节点上。这一点大大影响了MPP架构应用的可扩展性。

3、每个客户端同时连接所有节点通信 -> 影响网络

MPP架构每个节点独立，所以客户端往往需要连接所有节点进行通信，这使得网络也成为瓶颈。

4、分区容错性差

前面提到过MPP主要考虑CA，最次才是P。那么一旦扩展节点太多后，元数据管理十分困难。

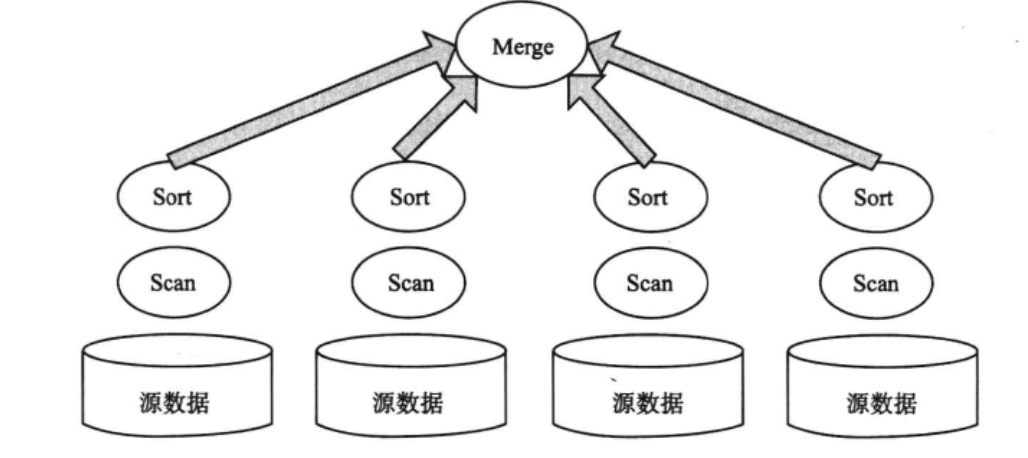
# 架构

## MPP架构

并行数据库往往采用MPP（Massively Parallel Processing，大规模并行处理）架构。MPP架构是一种**不共享**的结构，每个节点可以运行自己的操作系统、数据库等。每个节点内的CPU不能访问另一个节点的内存，节点之间的信息交互是通过节点互联网络实现的。

如图所示，将数据分布到多个节点，每个节点扫描本地数据，并由Merge操作符执行结果汇总。

**MPP Merge操作符：**



常见的数据分布算法有两种：

范围分区（Range Partitioning）：按照范围划分数据。

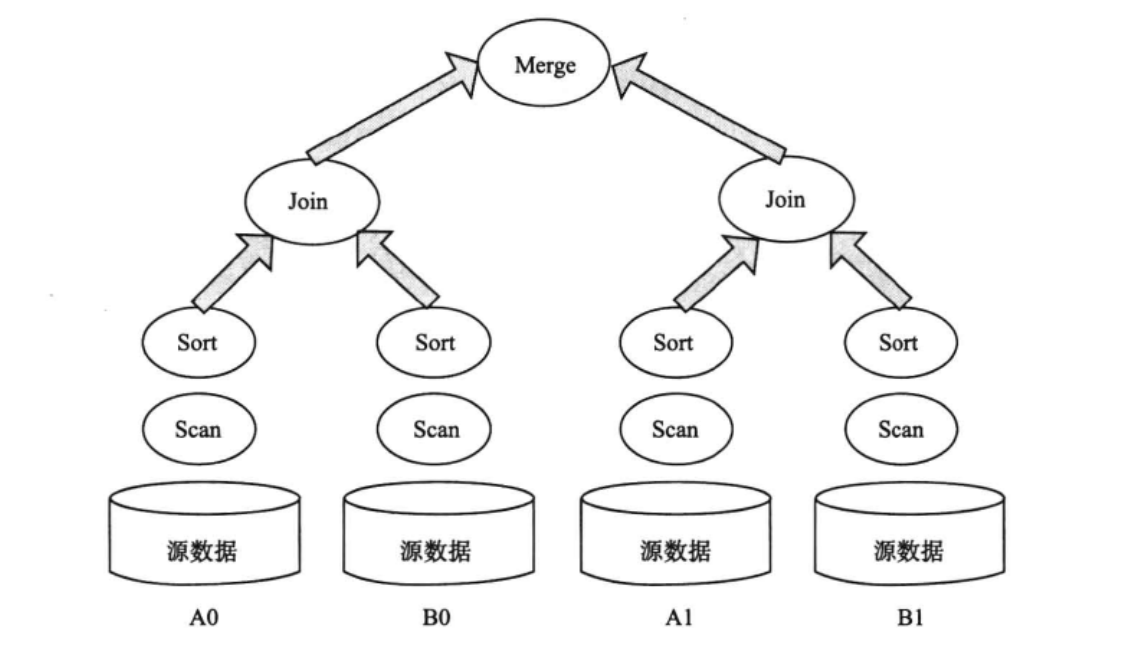
哈希分区（Hashing）：根据哈希函数计算结果将每个元祖分配给相应的节点。

Merge操作符：系统中存在一个或多个合并节点，它会发送命令给每个数据分片请求相应的数据，每个数据分片所在的节点扫描本地数据，排序后回复合并节点，由合并节点通过merge操作符执行数据汇总。**Merge操作符是一个统称，涉及的操作可能是limit、order by、group by、join等**。这个过程相当于执行一个Reduce任务个数为1的MapReduce作业，不同的是，这里不需要考虑执行过程中服务器出现故障的情况。

如果Merge节点处理的数据量特别大，可以通过Split操作符将数据划分到多个节点，每个节点对一部分数据执行group by、join等操作后再合并最终结果。

如图所示，假如需要执行“select \* from A,B where A.x=B.y”，可以分别根据A.x和B.x的哈希值将表A和B划分为A0、A1和B0、B1。由两个节点分别对A0、B0以及A1、B1执行join操作后再合并join结果。

**MPP Split操作符：**



并行数据库的SQL查询和MapReduce计算有些类似，可以认为MapReduce模型是一种更高层次的抽象。由于考虑问题的角度不同，并行数据库处理的SQL查询执行时间通常很短，出现异常时整个操作重做即可，不需要像MapReduce实现那样引入一个主控节点管理计算节点，监控计算节点故障，启动备份任务等。

## 其他架构

### SMP

### NUMA

### Hadoop

### 集群和MPP

# 产品

## PolarDB

### 概述

参考：

<https://help.aliyun.com/document_detail/178155.html?spm=a2c4g.138803.0.0.22ee227fsJ2Jrn>

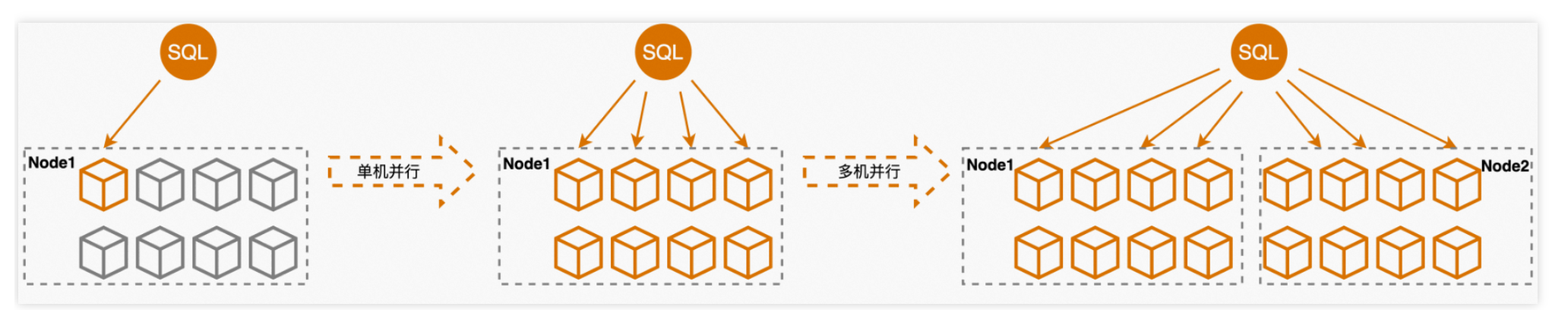
PolarDB MySQL版8.0版本重磅推出弹性并行查询框架，当您的查询数据量到达一定阈值，就会自动启动并行查询框架，从而使查询耗时指数级下降。

弹性并行查询（Elastic Parallel Query，ePQ）目前支持单机并行和多机并行两种并行引擎，单机并行引擎等效于原有的并行查询，多机并行引擎支持集群内跨节点的自适应弹性调度。

PolarDB MySQL版8.0.1版本支持单机并行查询，查询时在存储层将数据分片到不同的线程上，单个节点内多个线程并行计算，将结果流水线汇总到总线程。最后总线程做简单归并返回给用户，提高查询效率。

PolarDB MySQL版8.0.2版本除了支持原有的单机并行查询，又将线性加速能力提升了一个等级，引入了多节点分布式并行计算能力，即多机并行查询。基于代价将执行计划优化为更灵活的并行执行计划，改进了单机并行查询可能存在的Leader单点瓶颈和Worker负载不均衡的问题，同时突破了单个节点在CPU、Memory、IO上的资源瓶颈。基于多节点的资源视图，自适应的调度并行计算任务，在大幅提升并行计算能力、降低查询延迟的同时，平衡了各节点的资源负载，提升集群整体的资源利用率。

弹性并行查询（Elastic Parallel Query）针对云上用户实例CPU资源利用率较低、使用不均衡的特征，充分挖掘集群中多核CPU的并行处理能力，以8核32 GB（独享规格）的PolarDB MySQL版集群版为例，示意图如下所示：



### 使用

#### 并行查询EXPLAIN

#### 并行度控制策略

#### 通过Hint控制

#### 并行查询使用限制与串行执行结果兼容问题

**并行查询的使用限制**

PolarDB会持续迭代并行查询的能力，目前以下情况在并行计划中会有一定的局限性：

* 查询非Innodb表，查询无法并行。
* 使用全文索引的查询，查询无法并行。
* 包含存储过程Procedures的表达式，该表达式必须在leader上执行。
* Index Merge方式进行表扫描，则该表无法并行。
* 串行化隔离级别事务内的查询语句无法并行。
* 隔离级别是Repeatable-read的情况下，事务内的INSERT ... SELECT/REPLACE ... SELECT查询部分无法并行。

**与串行执行结果可能不兼容的地方**

* 错误提示次数可能会变化

串行执行中出现错误提示的查询，在并行执行的情况下，总体错误提示数可能会与串行有所不同。

* 精度问题

并行查询的执行过程中，可能会出现比串行执行多出中间结果的存储，如果中间结果是浮点型，可能会导致浮点部分精度差别，导致最终结果有细微的差别。

* 网络包或者中间结果长度超出max\_allowed\_packet允许的最大长度

并行查询的执行过程中，相比串行执行可能会多出中间结果。如果中间结果的长度超出了max\_allowed\_packet定义的最大长度，可能出现错误提示，可以通过增加max\_allowed\_packet参数的值来解决。如何修改参数请参见设置集群参数和节点参数。

* 结果集顺序差别

当并行执行未加ORDER BY关键字的SELECT ... LIMIT n语句时，返回的结果集可能与执行顺序不一致。由于有多个Worker同时执行，每次执行时Worker的执行速度是不确定的，当Leader得到足够的数据后，就会返回结果，因此返回的结果集可能与执行顺序不一致。

* 加了行锁的数据记录数增多

当并行执行SELECT ... FROM ... FOR SHARE语句时，InnoDB会将访问到的每一行数据都加锁，因此加了行锁的记录数可能会比非并行执行的情况下要多，这属于正常现象。

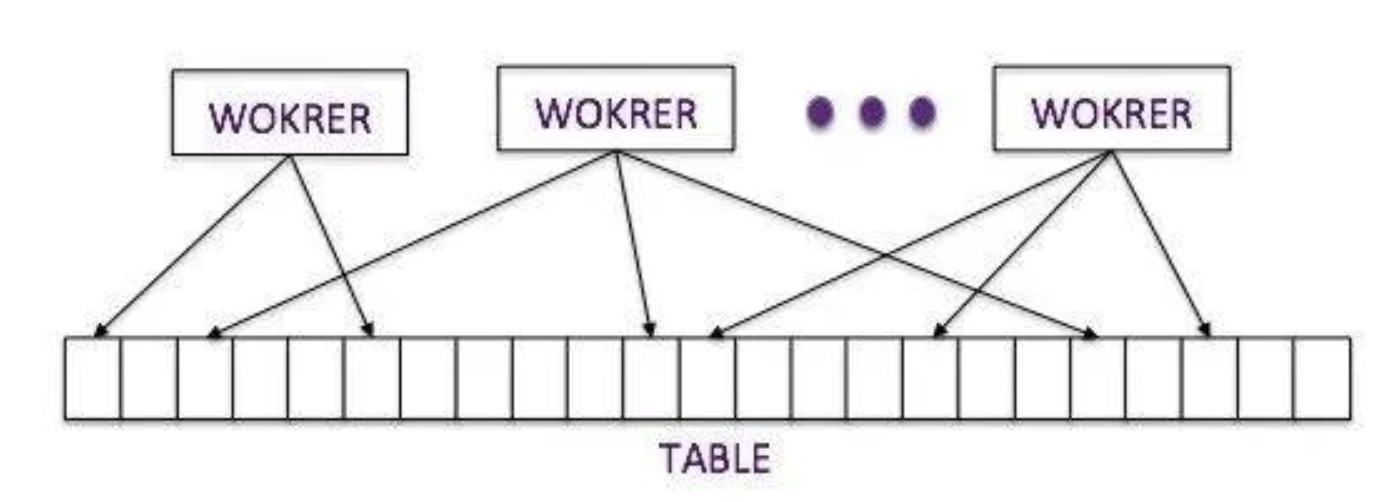
### 原理

参考：

#### 如何将查询并行起来

对于一个类OLAP的查询，显而易见的是它通常是对大批量数据的查询，数据量大意味着数据远大于数据库的内存容量，大部分数据可能无法缓存到数据库的缓冲区中，而必须在查询执行时才动态加载到缓冲区中，这样就会造成大量IO操作，而IO操作又是耗时的，因此首先要考虑的就是如何能加速IO操作。

由于硬件的限制，每次IO的耗时基本是固定的，虽然还有顺序IO和随机IO的区别，但在SSD已经盛行的今天，两者的差异也在逐渐接近。那么还有没有其它方式可以加速IO呢？显然并行IO是一个简单易行的方法，如果多个线程可以同时发起IO，每个线程只读取部分数据，这样就可以快速将数据读到数据库的缓冲区中。



并行读取数据的示意如上图所示，每个worker代表一个线程，如果数据已经有partition分区，可以每个线程读取一个partition；也可以将全部数据按固定大小进行分片，比如按一个数据页面大小，然后每个线程以Round-robin模式轮询读取一个分片。

这里需要注意的是，按已有partition分配给不同worker可能会导致每个worker处理的数据不均匀，而按Round-robin模式进行轮询，如果分片设置的比较小，相对来说就比较容易做到每个worker处理的数据比较均匀。

如果只是将数据读取到缓冲区中，而不是立即进行后续处理，那么这些数据就会因缓冲区爆满导致数据被换出，从而失去加速IO的意义。因此，**在并行读取数据的同时，必须同时并行处理这些数据，这是并行查询加速的基础**（pipeline的必要性）。

传统的优化器只能生成串行的执行计划，为了实现并行读取数据，同时并行处理数据，首先必须对现有的优化器进行改造，让优化器可以生成我们需要的并行计划。比如选择哪个表或哪些表可以并行读取，并且通过并行读取会带来足够的收益；或者哪些操作可以并行执行，并且可以带来足够的收益。

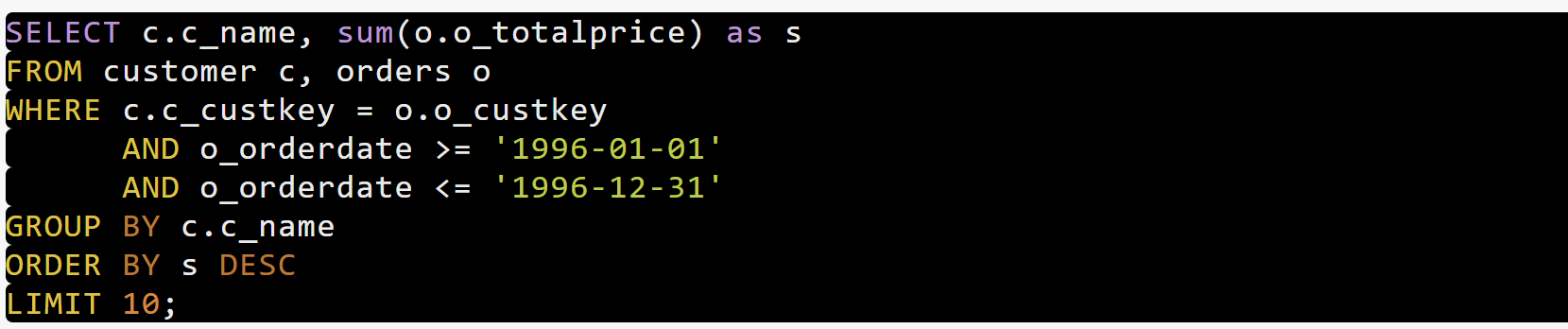
并不是说并行化改造一定会有收益，比如对一个数据量很小的表，可能只是几行，如果也对它进行并行读取的话，并行执行所需要的多线程构建再加上线程间的数据同步等所需要的代价可能远大于所得到的收益，总体来说，并行执行会需要更多的资源和时间，这就得不偿失了。因此查询计划的并行化必须是基于代价的，否则可能会导致更严重的性能退化问题。

#### 如何选择并行扫描的表

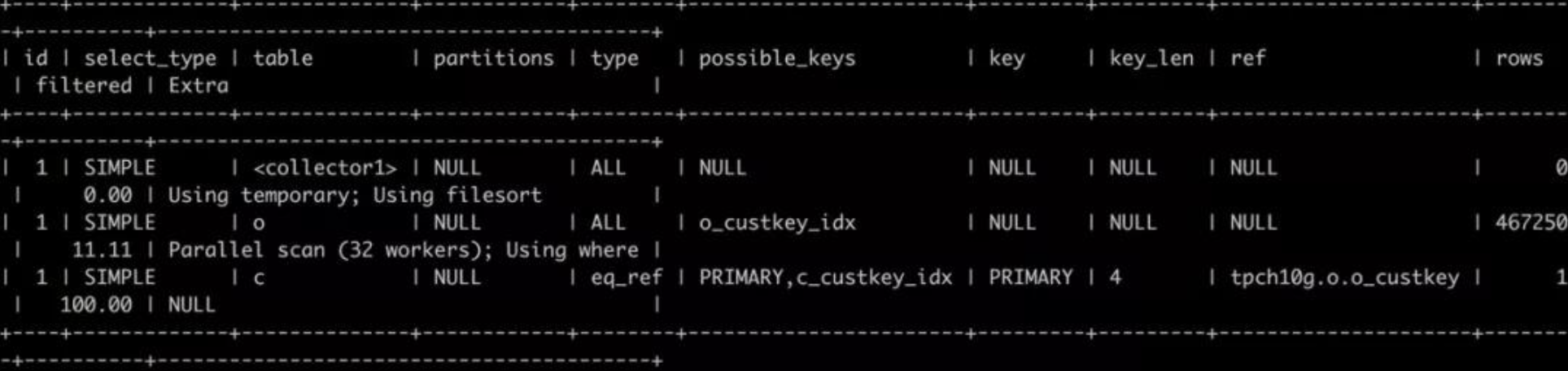
选择并行扫描的表是生成并行计划的重要基础，通过基于并行扫描代价的计算和比较，选择可以并行扫描的表作为候选，是并行执行计划迭代的步。基于新的并行代价，也许会有更优的JOIN顺序选择，尤其是当参与JOIN的表的数量比较多时，这需要更多额外的迭代空间，为防止优化过程消耗太多的时间，保持原有计划的JOIN顺序是一个不错的选择。另外，对于参与JOIN的每张表，因为表的访问方法不同，比如全表扫描、Ref索引扫描、Range索引扫描等，这些都会影响到终并行扫描的代价。

通常我们选择大的那张表作为并行表，这样并行扫描的收益大，当然也可以选择多个表同时做并行扫描，后面会继续讨论更复杂的情况。

下面以查询年度消费TOP 10的用户为例：



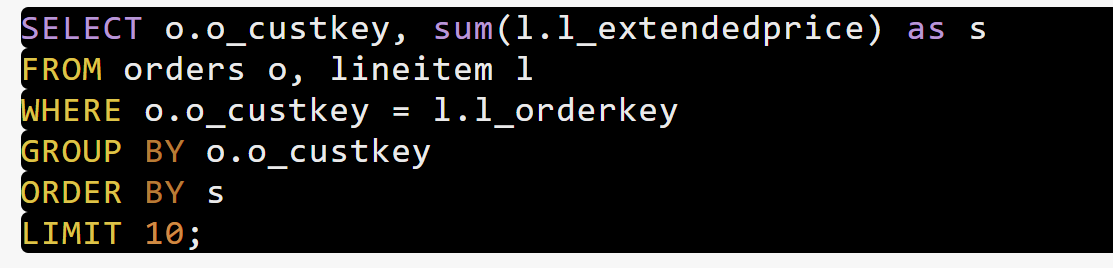
其中orders表为订单表，数据很多，这类表也被称之为事实表，customer表为客户表，数据相对较少，这类表也被称之为维度表。那么此SQL的并行执行计划如下图所示：



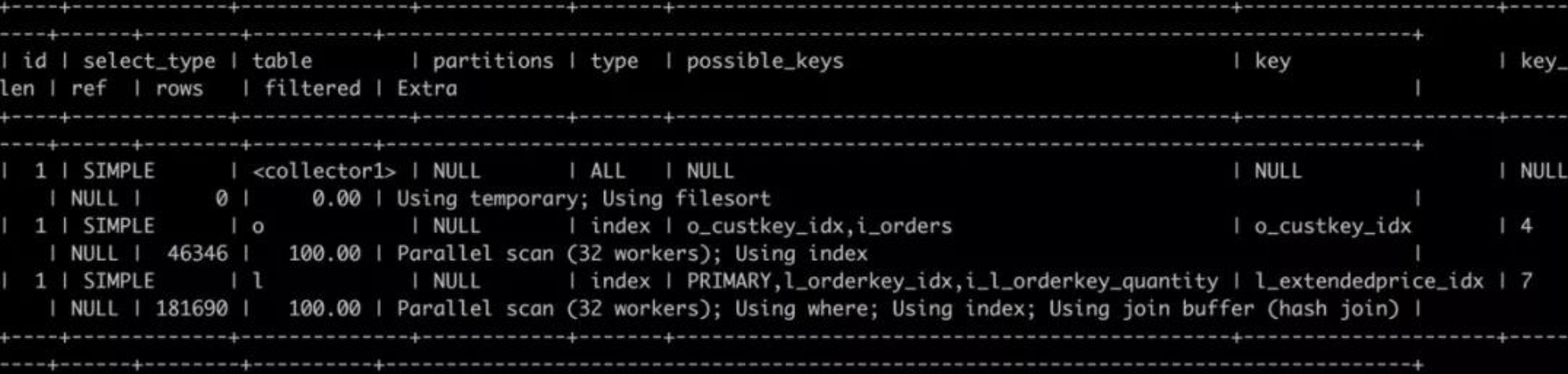
从计划中可以看出orders表会做并行扫描，由32个workers线程来执行，每个worker只扫描orders表的一部分数据分片，然后与customer表按o\_custkey做index lookup进行JOIN，JOIN的结果发送到一个collector组件，然后由collector组件继续做后续的GROUP BY、ORDER BY及LIMIT操作。

#### 选择多表并行的JOIN

将一张表做并行扫描之后，就会想为什么只能选择一张表？如果SQL中有2张或更多的FACT表，能不能可以将FACT表都做并行扫描呢？答案是当然可以。以下面SQL为例：



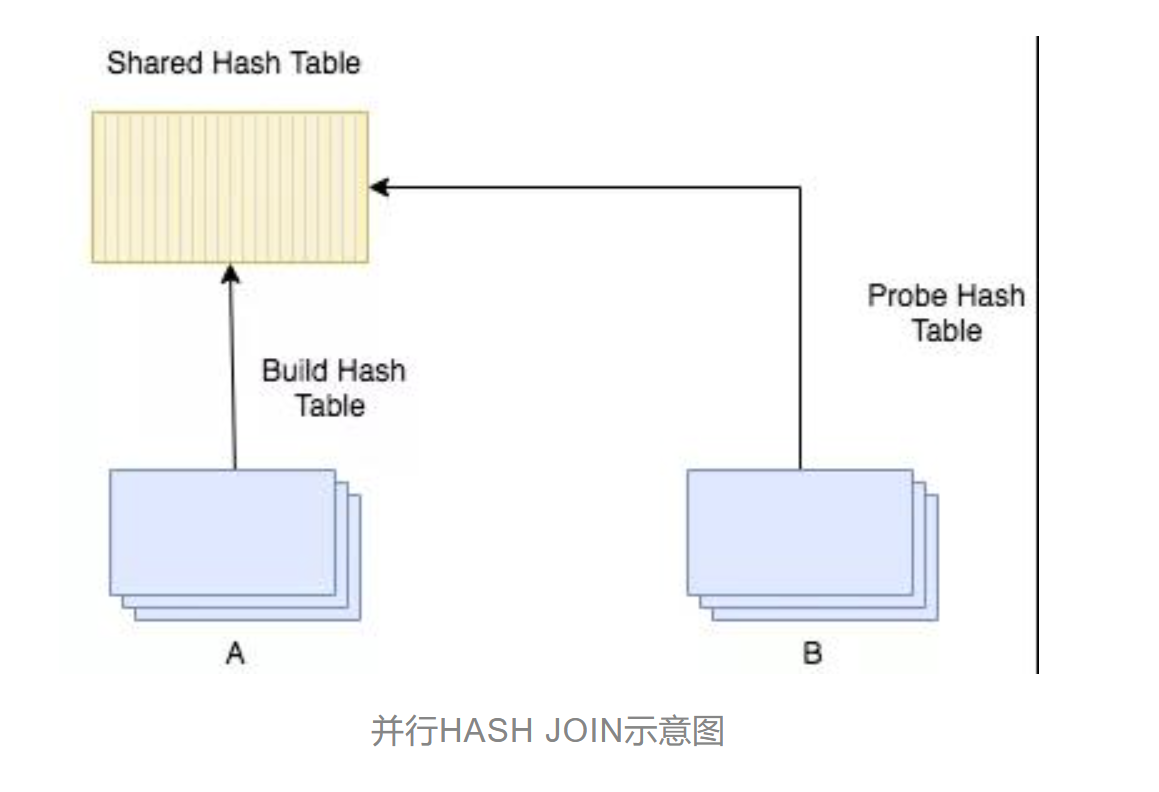
其中orders表和lineitem表都是数据量很大的事实表，此SQL的并行执行计划如下图所示：



从计划中可以看到orders表和lineitem表都会做并行扫描，都由32个workers线程来执行。那么多个表的并行是如何实现的呢？我们以2个表为例，当2个表执行JOIN时，通常的JOIN方式有Nested Loop JOIN、HASH JOIN等，对于不同的JOIN方式，为保证结果的正确性，必须选择合理的表扫描方式。

以HASH JOIN为例，对于串行执行的HASH JOIN来说，首先选择一个表创建HASH表称之为Build表，然后读取另一个Probe表，计算HASH，并在Build表中进行HASH匹配，若匹配成功，输出结果，否则继续读取。如果改为并行HASH JOIN，并行优化器会对串行执行的HASH JOIN进行并行化改造，使之成为并行HASH JOIN，并行化改造的方案可以有以下两种解决方案。

方案一是将2个表都按HASH key进行分区，相同HASH值的数据处于同一个分区内，由同一个线程执行HASH JOIN。方案二是创建一个共享的Build表，由所有执行HASH JOIN的线程共享，然后每个线程并行读取属于自己线程的另外一个表的分片，再执行HASH JOIN。终选择哪种方案，通过代价估算来决定。



对于方案一，需要读取表中的所有数据，根据选中的HASH key，对数据进行分区，并将数据发送到不同的处理线程中，这需要额外增加一个Repartition算子，负责根据分区规则将数据发送到不同的处理线程。

对于方案二，需要并行创建共享的HASH build表，当build表创建成功后，每个线程读取Probe表的一个分片，分别执行HASH JOIN，这里的分片并不需要按照HASH key进行分片，每个线程分别读取互不相交的分片即可。

#### 分析统计的复杂算子的并行

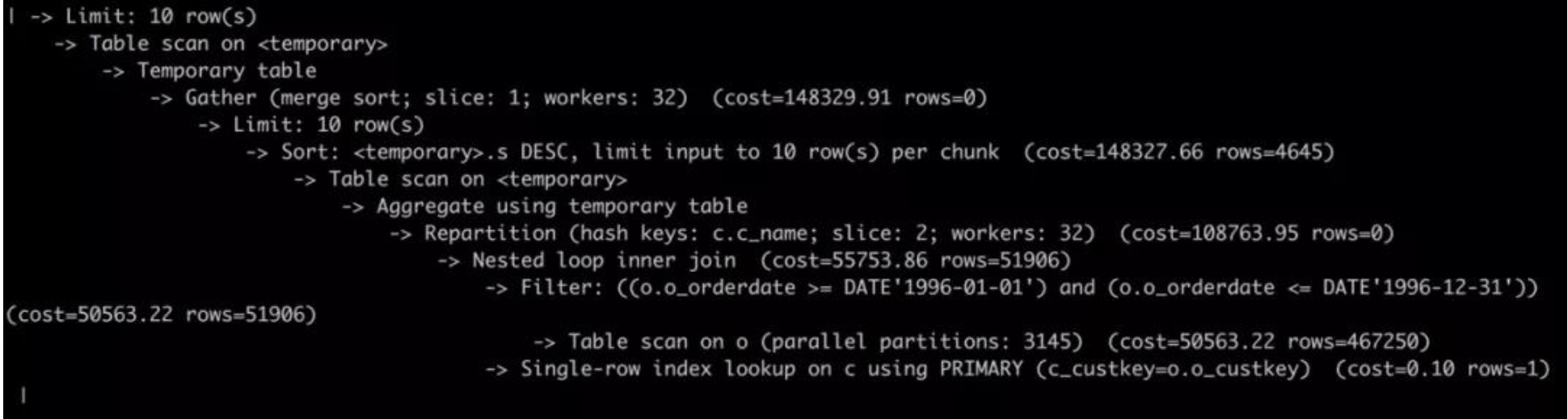
对于一个分析统计的需求，GROUP BY操作是绕不开的操作，尤其对大量的JOIN结果再做GROUP BY操作，是整个SQL中费时的一个过程，因此GROUP BY的并行也是并行查询引擎必须优先解决的问题。

以年度消费0客户的SQL为例，对GROUP BY并行化后的并行执行计划如下图所示：



与之前的执行计划相比，新的执行计划中多了一个collector组件，总共有2个collector组件。首先我们看第二行的collector组件，它的extra信息中有2条"Using temporary; Using filesort"，这表示它是对从workers接收到的数据执行GROUP BY，然后再按ORDER排序，因为只有个collector组件在用户的session中，所以这个collector也是在worker中并行执行，也就是说并行的做Group by和Order by以及Limit；然后看行的collector组件，它的extra信息中只有一条"Merge sort"，表示session线程对从workers接收到的数据执行一次merge sort，然后将结果返回给用户。这里可能就有人会提出疑问，为什么session线程只做merge sort就可以完成GROUP BY操作呢？另外LIMIT在哪里呢？

首先回答第2个问题，因为explain计划显示的问题，在常规模式下不显示LIMIT操作，但在Tree模式下会显示LIMIT操作。如下所示：



从Tree型计划树上可以清楚的看到LIMIT操作有2处，一处在计划的顶端，也就是在session上，做完limit后将数据返回给用户；另外一处在计划树的中间位置，它其实是在worker线程的执行计划上，在每个worker线程中在排序完成后也会做一次limit，这样就可以极大减少worker返回给session线程的数据量，从而提升整体性能。

下面来回答个问题，为什么GROUP BY只需要在worker线程上执行一次就可以保证结果的正确性。通常来说，每个worker只有所有数据的一个分片，只在一个数据分片上做GROUP BY是有极大的风险得到错误的GROUP BY结果的，因为同一GROUP分组的数据可能不只是在本WORKER的数据分片上，也可能在其它WORKER的数据分片中，被其它WORKER所持有。但是如果我们可以保证同一GROUP分组的数据一定位于同一个数据分片，并且这个数据分片只被一个WORKER线程所持有，那么就可以保证GROUP BY结果的正确性。通过Tree型执行计划可以看到，在并行JOIN之后，将JOIN的结果按GROUP分组的KEY值: c.c\_name进行Repartition操作，将相同分组的数据分发到相同的WORKER，从而保证每个WORKER拥有的数据分片互不交叉，保证GROUP BY结果的正确性。

因为每个WORKER的GROUP BY操作已经是终结果，所以还可以将ORDER BY和LIMIT也下推到WORKER来执行，进一步提升了并行执行的效率。

### 优化

#### Hash Join的并行优化

#### Semi-Join的并行优化

#### Window Function的并行优化

#### ROLLUP性能增强

### TPCH性能加速

### 应用场景

## OceanBase

## OpenGauss

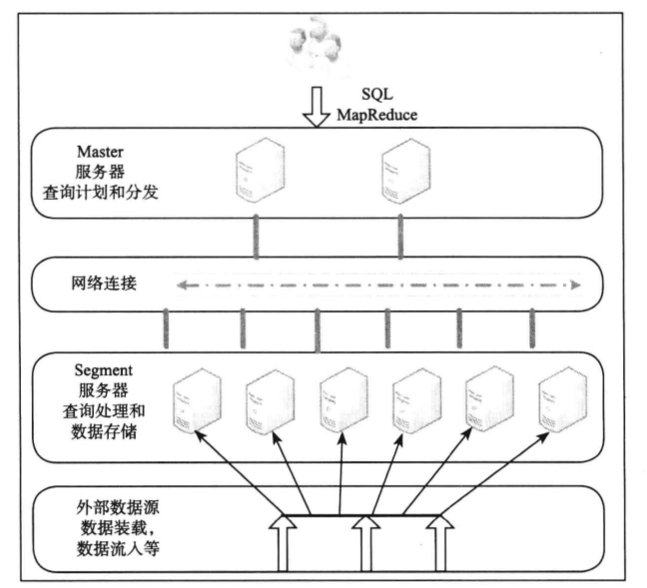
## TDSQL

## EMC Greenplum

Greenplum是EMC公司研发的一款采用MPP架构的OLAP产品，底层基于开源的PostgreSQL数据库。

### 整体架构

Greenplum系统主要包含两种角色：Master服务器（Master Server）和Segment服务器（Segment Server）。在Greenplum中每个表都是分布在所有节点上的。Master服务器首先对表的某个或多个列进行哈希运算，然后根据哈希结果将表的数据分布到Segment服务器中。整个过程中Master服务器不存放任何用户数据，只是对客户端进行访问控制和存储表分布逻辑的元数据。



Greenplum支持两种访问方式：SQL和MapReduce。用户将SQL操作语句发送给Master服务器，由Master服务器执行词法解析、语法分析，生成执行计划树，并将查询请求分发给多台Segment服务器。每个Segment服务器返回部分结果后，Master服务器会进行聚合并将最终结果返回给用户。除了高效查询，Greenplum还支持通过数据的秉性装载，将外部数据秉性装载到所有的Segment服务器。

### 并行查询优化器

Greenplum的并行查询优化器负责将用户的SQL或者MapReduce请求转换为物理执行计划。Greenplum采用基于代价的查询优化算法（cost-based optimization），从各种可能的查询计划中选择一个代价最小的。Greenplum优化器会考虑集群全局统计信息，例如数据分布，另外，除了考虑单机执行的CPU、内存资源消耗，还需要考虑数据的网络传输开销。

Greenplum除了生成传统关系数据库的物理运算符，包括表格扫描（Scan）、过滤（Filter）、聚集（Aggregation）、排序（Sort）、联表（Join），还会生成一些并行运算符，用来描述查询执行过程中如何在节点之间传输数据。

广播（Brodcast，N:N）：每个极端节点将目标数据发送给所有其他节点。

重新分布（Redistribute，N:N）：类似MapReduce中的shuffle过程，每个计算节点将目标数据重新哈希后分散到所有其他节点。

汇总（Gather，N:1）：所有的计算节点将目标数据发送给某个节点（一般为Master服务器）。

## ClickHouse

参考：<https://developer.volcengine.com/articles/7176266450804408377>

### 背景

ClickHouse的执行模式与Druid、ES等大数据引擎类似，其基本的查询模式可分为两个阶段。第一阶段，Coordinator在收到查询后，将请求发送给对应的Worker节点。第二阶段，Worker节点完成计算，Coordinator在收到各Worker节点的数据后进行汇聚和处理，并将处理后的结果返回。

两阶段的执行模式能够较为高效地支持目前许多常见的业务场景，例如各类大宽表单的查询，这也是ClickHouse最擅长的场景。ClickHouse的优点是简单、高效，通常来说，简单就意味着高效。但随着企业业务的持续发展，愈加复杂的业务场景对ClickHouse提出了以下三类挑战。

第一类，当一阶段返回的数据较多，且二阶段计算较为复杂时，Coordinator会承受较大压力，容易成为Query的瓶颈。 例如一些重计算的Agg算子，如Count Distinct，若采用哈希表的方式进行去重，第二阶段需在Coordinator单机上去合并各个Worker的哈希表。这个计算量会很重且无法并行。

第二类，由于目前ClickHouse模式并不支持Shuffle，因此对于Join而言，右表必须为全量数据。 无论是普通Join还是Global Join，当右表的数据量较大时，若将数据都放到内存中，会比较容易OOM。若将数据spill到磁盘，虽然可以解决内存问题，但由于有磁盘 IO 和数据序列化、反序列化的代价，因此查询的性能会受到影响。特别是当Join采用Hash Join时，如果右表是一张大表，构建也会比较慢。针对构建问题，近期社区也进行了一些右表并行构建的优化，数据按照Join key进行Split来并行地构建多个Hash Table，但额外的代价是左右表都需要增加一次Split操作。

第三类，则是关于复杂查询（如多表 Join、嵌套多个子查询、window function 等），ClickHouse对这类需求场景的支持并不是特别友好， 由于ClickHouse并不能通过Shuffle来分散数据增加执行并行度，并且其生成的Pipeline在一些case下并不能充分并行。因此在某些场景下，难以发挥集群的全部资源。

随着企业业务复杂度的不断提升，复杂查询，特别是有多轮的分布式Join，且有很多agg的计算的需求会越来越强烈。在这种情况下，业务并不希望所有的Query都按照ClickHouse擅长的模式进行，即通过上游数据 ETL 来产生大宽表。这样做对ETL的成本较大，并且可能会有一些数据冗余。企业的集群资源是有限的，但整体的数据量会持续增长，因此在这种情况下，我们希望能够充分地去利用机器的资源，来应对这种越来越复杂的业务场景和SQL。所以我们的目标是基于ClickHouse能够高效支持复杂查询。

### 方案

对于ClickHouse复杂查询的实现，我们采用了分Stage的执行方式，来替换掉目前ClickHouse的两阶段执行方式。类似于其他的分布式数据库引擎，例如Presto等，会将一个复杂的Query按数据交换情况切分成多个 Stage，各Stage之间则通过Exchange完成数据交换。

Stage之间的数据交换主要有以下三种形式。

* 按照单个或者多个key进行Shuffle
* 将单个或者多个节点的数据汇聚到一个节点上，称为Gather
* 将同一份数据复制到多个节点上，称为Broadcast或广播

对于单个Stage执行，继续复用ClickHouse目前底层的执行方式。开发上按照不同功能切分不同模块。各个模块预定接口，减少彼此的依赖与耦合。即使模块发生变动或内部逻辑调整，也不会影响其他模块。其次，对模块采用插件架构，允许模块按照灵活配置支持不同的策略。这样便能够根据不同业务场景实现不同的策略。

### 优化与诊断

### 效果与展望

## Presto

## HP Vertica

Vertica是Michael Stonebraker的学术研究项目C-Store的商业版本，并最终被惠普公司收购。Vertica在架构上与OceanBase有相似之处。

### 混合存储模型

Vertica的数据包含两个部分：ROS（Read-Optimized Storage）以及WOS（Write-Optimized Storage），WOS的数据在内存中且不排序和加索引，ROS的数据在磁盘中有序且压缩存储。后台的“TUPLE MOVER”会不断地将数据从WOS读出并网ROS更新（同时完成排序和索引）。Vertica的折中设计和OceanBase很相似，ROS对应OceanBase中的ChunkServer，WOS对应OceanBase中的UpdateServer。由于后台采用“BULK”的方式批量更新，性能非常好。

### 多映射（Projections）存储

### 列式存储

Vertica中的每一列数据独立存储在磁盘的连续块上。查询数据时，Vertica只需要读取那些需要的列，而不是被选择的行的所有的列数据。

### 压缩技术

## Google Dremel

### 系统架构

### Dremel vs MapReduce

# 适用场景

集群规模100以内、并发小（50以下）。

MPP架构目前被并行数据库广泛采用，一般通过scan、sort和merge等操作符实时返回查询结果。