Hologres（中文名交互式分析）是阿里云自研的一站式实时数仓，这个云原生系统融合了实时服务和分析大数据的场景，全面兼容PostgreSQL协议并与大数据生态无缝打通，能用同一套数据架构同时支持实时写入实时查询以及实时离线联邦分析。它的出现简化了业务的架构，与此同时为业务提供实时决策的能力，让大数据发挥出更大的商业价值。从阿里集团诞生到云上商业化，随着业务的发展和技术的演进，Hologres也在持续不断优化核心技术竞争力，为了让大家更加了解Hologres，我们计划持续推出Hologers底层技术原理揭秘系列，从高性能存储引擎到高效率查询引擎，高吞吐写入到高QPS查询等，全方位解读Hologers，请大家持续关注！

本期我们将带来Hologers高效率分布式查询引擎的技术原理解析。

Hologres作为HSAP服务分析一体化的落地最佳实践，其查询引擎是一个完全自研的执行引擎，它的核心设计目标是支持所有类型的分布式分析和服务查询，并做到极致查询性能。为了做到这一点，我们借鉴了各种分布式查询系统，包括分析型数据库，实时数仓等，吸取了各方面的优势从零开始打造出一个全新的执行引擎。

为什么要选择从零开始做一个新的查询引擎？开源的分布式分析查询系统主要有两大类：

• 一类是传统的 Massively Parallel Processing 系统，能够支持通用的 SQL 查询，但是对实时场景支持不够好，性能不够理想。

• 一类是 Apache Druid 和 ClickHouse这些实时数仓，是专门为实时场景设计和优化的，能够比较好地支持一些常见的单表实时查询，但是复杂查询的性能比较差。

• 另外大数据生态圈基于 MapReduce 的引擎比较适合批处理 ETL，一般不太适合在线服务和多维分析的场景，性能也差不少。

Hologres 执行引擎是在一个能支持复杂查询和上述高性能实时服务查询的通用架构，先首先实现了常用的实时数仓场景，深入优化并用内部 Benchmark 验证了性能和稳定性超过包括专用实时数仓的其它竞品之后，再扩展到其它复杂查询的支持。扩展的过程中，在不可避免地系统变得越来越复杂的同时，也用 Benchmark 帮助保持简单实时查询的性能没有回退。如果在已有的查询引擎上做改进，因为很多架构和设计上的选择已经定型，牵一发而动全身，就很难达到这样的效果。

Hologres执行引擎从开发到落地实践面临了非常多的挑战，但也给我们提供了机会把这个领域的各种新进展都结合利用起来，并超越已有系统做到对各种查询类型的高性能处理，其背后主要是基于以下特点：

• 分布式执行模型：一个和存储计算分离架构配合的分布式执行模型。执行计划由异步算子组成的执行图 DAG（有向无环图） 表示，可以表达各种复杂查询，并且完美适配 Hologres 的数据存储模型，方便对接查询优化器，利用业界各种查询优化技术。

• 全异步执行：端到端的全异步处理框架，可以避免高并发系统的瓶颈，充分利用资源，并且最大可能地避免存储计算分离系统带来的读数据延迟的影响。

• 向量化和列处理：算子内部处理数据时最大可能地使用向量化执行，和存储引擎的深度集成，通过灵活的执行模型，充分利用各种索引，并且最大化地延迟向量物化和延迟计算，避免不必要的读数据和计算。

• 自适应增量处理：对常见实时数据应用查询模式的自适应增量处理。  
• 特定查询深度优化：对一些查询模式的独特优化

下面将会对各个模块一一介绍。

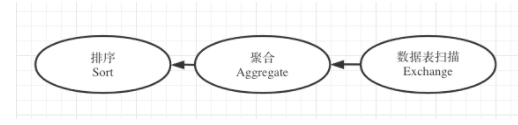
分布式执行模型

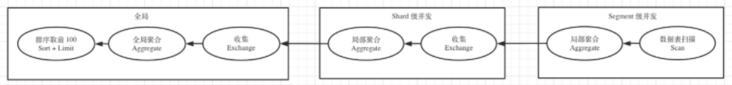
Hologres 是能够弹性无限水平扩展数据量和计算能力的系统，需要能够支持高效的分布式查询。

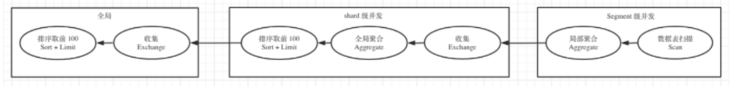
Hologres 查询引擎执行的是由优化器生成的分布式执行计划。执行计划由算子组成。因为 Hologres 的一个表的数据会根据 Distribution Key 分布在多个 Shard 上，每个 Shard 内又可以包含很多 Segment，执行计划也会反映这样的结构，并分布到数据所在的节点去执行。每个Table Shard 会被加载到一个计算节点，数据会被缓存到这个节点的内存和本地存储。因为是存储计算分离的架构，如果一个节点出错，其服务的 Shard 可以被重新加载到任意一个计算节点，只是相当于清空了缓存。

例如一个比较简单的查询。

**select** **key**, count(**value**) **as** total **from** table1 **group** **by** **key** **order** **by** total **desc** **limit** 100。

如果是单机数据库，可以用这样的执行计划。如果数据和计算分布在多个节点上，就需要更复杂的执行计划。  


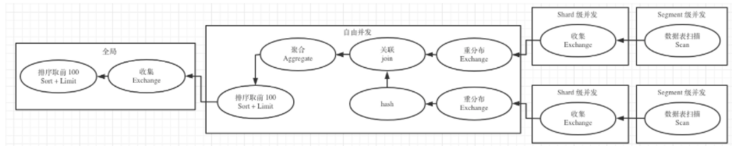
在分布式表上，为了更高效地执行，尽量减少数据传输，可以把执行计划分为不同片段（Fragment）分布到相应节点执行，并且把一些操作下推来减少 Fragment 输出的数据，可能就变成这样的执行计划：  


根据数据的特性，优化器可能会生成不同的计划。例如在某一个局部聚合并没有显著减少数据量的时候，可以省略这个算子。又例如在 Key 就是 Distribution key 的时候，可以优化为：  


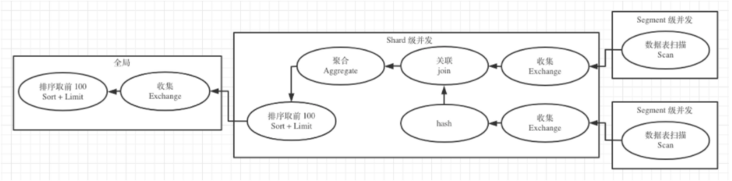
从这些例子可以看出，Hologres 的执行计划根据数据的特性切分为不同的片段之后分布式并发执行。片段之间通过 Exchange 算子进行数据交换。更复杂的比如多表关联（Join）查询，会有更多的片段和更复杂的数据交换模式。

比如以下SQL

**select** user\_name, sum(**value**) **as** total **from** t1 **join** t2 **on** t1.user\_id = t2.user\_id **where** … **group** **by** user\_name **order** **by** total **limit** 100

在Hologres中可以是这样的执行计划  


如果 Join key 和 Distribution Key 一致，可以优化为如下执行计划，减少远程数据传输。根据需要的查询合理地设置 Distribution Key，可能显著提高查询性能。



根据过滤条件和统计信息等等，优化器还可能生成不同的优化执行计划，比如包含动态过滤，局部聚合等等。

这样的分布式执行计划足够通用，可以表达所有的 SQL 查询和一些其它查询。执行计划和大部分 Massively Parallel Processing (MPP) 系统也比较类似，方便借鉴和集成业界的一些适用的优化。稍微独特一些的地方是很多查询计划片段的实例是和 Hologres 的存储结构对齐的，能够进行高效的分区裁剪和文件裁剪。

同时，Hologres 实现了 PostgreSQL 的 Explain 和 Explain Analyze 系列语句，可以展示文本格式的执行计划和相应的执行信息，方便用户自助了解执行计划，并针对性做出SQL优化调整。

全异步执行

高并发系统，特别是有大量 I/O 的系统，频繁地等待或者任务切换是常见的系统瓶颈。异步处理是一种已经被证明行之有效的避免这些瓶颈，并把高并发系统性能推到极致的方法。

Hologres 的整个后端，包括执行引擎、存储引擎和其它组件，统一使用 HOS（Hologres Operation System） 组件提供的异步无锁编程框架，能够最大化异步执行的效果。每个 Fragment 的实例使用 HOS 的一个 EC （逻辑调度单位），使得一个 Fragment 里的所有算子和存储引擎可以异步执行并且无锁安全访问绝大多数资源。

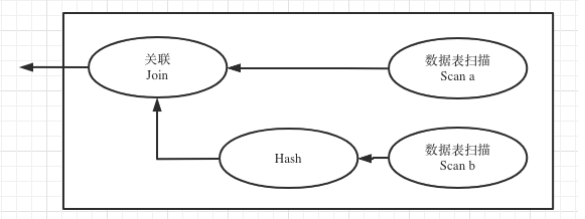
算子和 Fragment 都是类似这样的接口：

future<> Open(const SeekParameters& parameters, ...)

future<RecordBatchPtr, bool> GetNext(...)

future<> Close(...)

除了一般异步处理的好处外，异步算子接口较好地规避了存储计算分离架构下相对较高的读数据延迟对查询性能的影响，并且对分布式查询的执行模型本身也有独特的好处。

DAG 执行引擎一般可以分为拉数据的模性（比如火山模型）和推的模型（比如很多大数据的分阶段执行模型），各有其优缺点。而 Hologres采用的异步的拉模型能够取得两种模型的好处并且避免其缺点（已经申请了专利）。举一个常见的 Hash Join 来说明：  


火山模型可以简单做到先拉完 b 的数据构建 hash table，然后流式处理 a 的数据不用全放在内存里。但是当 a 或者 b 需要读数据的时候，简单的实现需要等待不能把 CPU 打满，需要通过提高 Fragment 的并发数或者引入复杂的 pre-fetch 机制来充分利用资源，而这些又会引入别的性能问题。

推数据的模型，比较容易做到并发读数据请求并在完成的时候触发下游处理，但是上述 Join算子的实现会比较复杂。比如 a 处理完一批数据推到 Join 算子而 b 的 hash table 还没有构建完成，这批数据就需要暂存到内存里或者盘上，或者引入反压机制。在 Fragment 的边界也会有类似问题，造成一些在拉数据模型下不需要的数据缓存。

Hologres 的算子和 Fragment 的异步拉数据模型，可以像火山模型一样简单做到按需从上游获取数据，而同时又可以像推数据模型一样简单做到读数据并发，只要向上游发出多个异步 GetNext，上游处理完成时会自然触发后续处理。异步 GetNext 的数目和时机，可以看做是天然的流控机制，可以有效做到提高 CPU 利用率并且避免不必要的数据暂存。  
Hologres 已经用这个异步模型实现了一个完整的查询引擎，可以支持所有 PostgreSQL 的查询。

列处理和向量化

按列处理和向量化执行都是分析查询引擎常用的优化机制，可以大幅度提高数据处理的效率。Hologres 也不例外，在能使用向量处理的时候尽量使用。

Hologres 在内存里也采用列式存储。在内存里按列存储数据能够使用更多的向量处理。列式组织数据还有一个好处，就是对延迟计算比较友好。比如 select … where a = 1 and b = 2 …，对一批数据（一般对应存储的一个 row group），Hologres的 scan 算子输出的 a 和 b 可以是延迟读取的 a 和 b 的信息，在处理 a = 1 的时候会读取这一批的 a。如果 a=1 对这一批的所有行都不满足，这一批的 b 这一列就根本不会被读取。

但是对某些按行处理的算子，比如 Join，按列存储的数据可能会造成更多的 CPU cache miss ，带来较大的性能问题。很多查询引擎会在不同的点引入按列存储和按行存储的转换，但是频繁的转换本身会带来不小的开销，而且列转行会造成上述延迟读取列被不必要地读取，还有一些其它的性能问题。

自适应增量处理

很多实时数据应用经常会对一个查询用不同的时间段反复执行。比如一个监控指标页面打开后，会定期执行 select avg(v1) from metrics where d1 = x and d2 = y and ts >= '2020-11-11 00:00:00' and ts < '2020-11-11 03:01:05' and … group by d3 … 这样的查询，下一次会改成 ts < '2020-11-11 00:03:10'，再下一次 ts < '2020-11-11 00:03:15'。

流计算或者增量计算可以对这种查询进行非常高效的处理。但是对这种用户可以随意生成的交互式查询，通常不可能对所有组合都配置流计算或者增量计算任务。如果每次都简单执行查询，又可能有大量的重复计算造成资源浪费和性能不理想。

Hologres充分利用存储引擎和计算引擎的深度集成和列式存储大部分数据在只读文件中的特性，在能提供包含最新写入数据的查询结果的同时尽量避免重复计算，对这种类型的查询能够显著提升性能和减少资源使用。

针对特定查询模式的深度优化

Hologres 对一些特定查询模式有独特的优化。这里以Filter Aggregate 优化为例子。

很多数据应用都有开放列的需求，相当于可以动态添加逻辑列而不用改 Table Schema。比如有一列是多值列 tags（Postgres 可以用 Array 类型）里面存了'{c1:v1, c2:u1}' 这样的多个逻辑列的值。查询的时候，如果使用普通列，一类常见的查询是

*-- Q1:*

**select** c1, sum(x) **from** t1 **where** c1 **in** (v1, v2, v3) **and** **name** = 'abc' **group** **by** c1

使用开放列后，这样的查询会转变为

*-- Q2:*

**select** unnest(tags), sum(x) **from** t1 **where** name = 'abc' **and** tags && **ARRAY**['c1:v1', 'c1:v2', c1:v3']

group by unnest(tags)

having unnest(tags) in ('c1:v1', 'c1:v2', c1:v3')

这种查询，Hologres 可以利用位图索引快速计算过滤条件得到相关的行，但是之后从多值列里面取出相关数据操作不能使用向量处理，性能不能达到最优。经过调研，可以把查询的执行转换为

Q3:

**select** 'c1:v1', sum(x) **from** t1 **where** tags && **ARRAY**['c1:v1']

**UNION** **ALL**

**select** 'c1:v2', sum(x) **from** t1 **where** tags && **ARRAY**['c1:v2']

**UNION** **ALL**

…

这样每个 UNION ALL 分支可以只读取 name 和 tags 的位图索引计算过滤条件，然后用 x 列的数据和过滤条件进行向量计算 SUM\_IF 即可得出想要的结果。这样的问题是，每个分支都要过一遍 t1，读取 x 列以及 name 列的位图索引，带来重复计算。最后引入了一个 filter aggregate 的特殊算子来把这类常用查询优化到极致性能，可以只过一遍 t1 并且去掉重复操作，只用向量计算即可得到结果，不需要读取 tags 列的数据。在一个几十 TB的表上实测性能提升 3 倍以上。

类似的优化，Hologres 的执行引擎都会尽量抽象为比较通用的算子，可以适用于更多场景。Filter Aggregate 算子也是 Hologres 申请的专利之一。

总结

Hologres 执行引擎在一个架构里集中了相关分布式查询系统的几乎所有最高效的优化方式（包括各种类型的索引）并作出了特有的改进。通过和存储引擎深度整合，能充分发挥异步模型的优势，并高效利用各种类型的索引来加速查询。所有这些加起来，带来了超越已有系统的性能，并在阿里巴巴双 11 的数据规模下通过了实战的考验，（2020年双11顶住了5.96亿/秒的实时数据洪峰，基于万亿级数据对外提供多维分析和服务，99.99%的查询可以在80ms以内返回结果），对外高并发高性能地提供分布式 HSAP 查询服务。