**Orca: 大数据优化器结构模块**

Mohamed A. Soliman*∗*, Lyublena Antova*∗*, Venkatesh Raghavan*∗*, Amr El-Helw*∗*, Zhongxian Gu*∗*, Entong Shen*∗*, George C. Caragea*∗*, Carlos Garcia-Alvarado*∗*, Foyzur Rahman*∗*, Michalis Petropoulos*∗*, Florian Waas*‡*,

Sivaramakrishnan Narayanan*§*, Konstantinos Krikellas*†*, Rhonda Baldwin*∗*

### *∗* Pivotal Inc. Palo Alto, USA

*‡* Datometry Inc. San Francisco, USA

*†* Google Inc.

### Mountain View, USA

*§* Qubole Inc.

### Mountain View, USA

**摘要**

在数据管理系统中分析查询的处理性能主要依赖于系统内优化器的能力。不断增长的数据量和对于复杂的分析查询的浓厚兴趣促使Pivotal打造一个新的查询优化器。

在这篇论文中我们提出Orca架构，针对所有Pivotal数据管理产品新的优化器，包括PivotalGreenplum数据库和Pivotal HAWQ。Orca是最先进的查询优化器结合我们自己的研究成果，在模块化和轻便化得到了全面发展。

除此之外描述了全部的结构，我们着重几个独特特征与其他的系统进行比较。

# Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query pro- cessing; Distributed databases*

# 关键字

查询优化, 代价估算, MPP, 并行处理

# 导论

大数据已经给查询优化带来重新的吸引力，作为一种新型的数据管理系统在可扩展性、可用性、处理能力已经推到前所未有的一层。这也使得对TB甚至是PB级的数据用SQL已经类SQL接口更为方便

尽管在这个领域研究成果很多，但大部分在商业已经开源项目的查询优化器仍旧主要基于的技术可以追溯到早起的商业数据库开发，经常也能得到最佳的效果。

实现研究与实际之间的重大差距，我们制定了一个体系结构来满足当前的需求，还保证对未来发展有足够的余量。

在这篇论文中，我们描述了Orca在Greenplum/Pivotal的研究成果和开发工作。Orca是专门为要求极高的负载分析的先进的查询优化器。与其他优化器最大的不同有以下几个重要方面：

**模块化。**对元数据和系统描述使用高度的可扩展性抽象。Orca不像传统的优化器只针对特定的系统。而是可以其元数据提供SDK通过插件的方式来移植到其他数据管理系统

**可扩展性.** 所有的元素查询和它的优化都是平等的，Orca可以避免在多阶段优化中某些优化带来的陷阱。

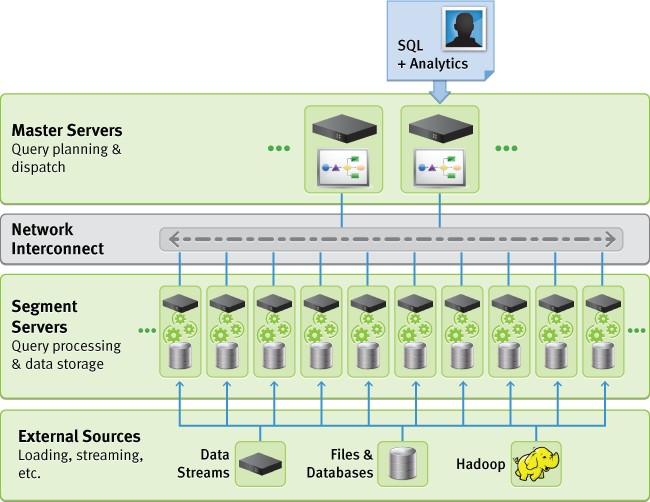
多阶段优化很难扩展，作为新的优化或查询往往造成不匹配以前设置的阶段边界

**可利用多核.** Orca利用一个高效的多核调度器来将独立的细粒优化子任务部署在多个内核来优化处理

**可变性.** Orca有特点的规定为确定的正确性和性能水平提供内建机制。除了提高工程的实践，这些工具能够使得更自信的快速开发，在引进新功能以及修复bug时减少周转时间。

**性能.** Orca很大的提高了我们以前的系统，在许多情况下提高了查询速度在10-1000倍.

我们描述了Orca的体系结构以及他设计的突出的先进特性 我们提供了各种组件的设计方案和详细的工程实践，我们已经率先部署了这个方案。最后，我们基于TPC-DS测试对Orca和其他系统进行了比较，尤其是，我们集中查询处理系统已经发布到开源空间



**Figure 1:** **High level GPDB architecture**

剩余的论文按一下组织的。第二节我们给出技术体系结构的预备知识。第三节，我们提出Orca体系结构并描述它的组件。第四节，提出查询优化流程。第五节，描述Orca如何与后端数据库系统交换元数据。第六节，利用工具来构建一个可变的查询优化器。第七节，提出我们研究经验，第八节，讨论相关工作。第九节最后总结

# PRELIMINARIES

我们提供了预备知识在大规模并行处理数据库 (Section [2.1](#_bookmark2)), 和 Hadoop 查询引擎(Section [2.](#_bookmark3)2).

# 大规模并行处理

Pivota 的Greenplum数据库（GPDB）是一个大规模并行处理分析型数据库。GPDB 采用无共享的计算体系结构，具有两个或更多的协同处理器。每个处理器有它自己的内存，操作系统和磁盘。GPDB利用高性能的系统架构来分布载入PB数据仓库中，并行利用系统资源来处理给定查询。

图 [1](#_bookmark0) 表示了一个高层次的GPDB架构.通过分散在多个主机和服务器创建一组独立的数据库，所有工作在一起提交到单独的服务数据库镜像。当客户端连接和提交一条SQL语句时，Master 是GPDB的入口点。与其他数据库主要的协调工作被称为segments, 来处理数据的加工和存储。当一个查询发送到master, 它被优化并以更小的组件分发到segment来一起工作来提交最终的结果。与segment之间的通信通过网络来进行。相互关系使用标准的Gb网络交换结构。

在查询执行期间，数据以多种方式发送到segment,包括hash分布（一个元组通过一些hash函数来发送到segments）,复制发送（通过一张表的全量副本来存储到每一个segment）和单实例分布（所有的被分布的表通过segment聚合来发送到单独的主机(通常是master)）。

# Hadoop用SQL

在Hadoop上处理分析查询日渐流行。首先，查询被表达成MapReduce的工作，hadoop的吸引力归功于它的伸缩性和容错能力。编码，用MapReduce手动的去优化和维护复杂的查询很困难，像Hive这种类SQL的语言被开发在Hadoop上层。HiveQL查询被编译成MapReduce工作以及通过Hadoop来执行。HiveQL加速复杂查询的编码也使得在Hadoop生态系统优化变得透明，因为编译成MapReduce工作表现糟糕。

Pivotal通过引入HAWQ来应对挑战，在HDFS上一个大规模并行SQL编译引擎。HAWQ在它的核心利用Orca来设计有效的查询机会来以最小的代价访问Hadoop集群中的数据。HAWQ架构体系结合一个前所未有的具有基于代价的优化器使得可伸缩性和容错性的Hadoop更能交互式的处理PB规模数据。

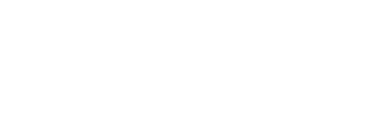
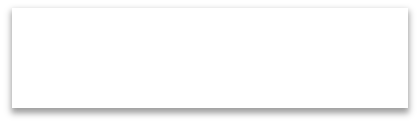
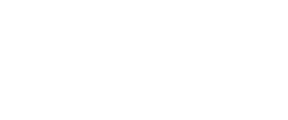
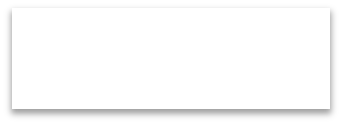
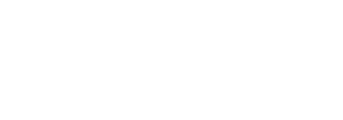
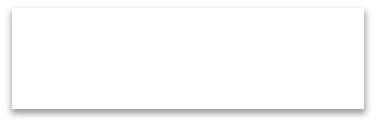
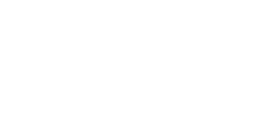
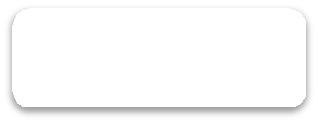
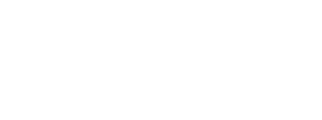
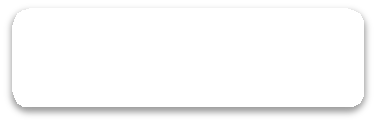
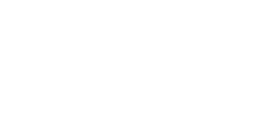
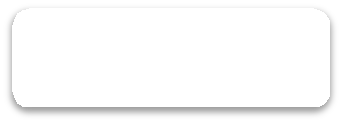
最近一些其它的工作，包括Cloudera的Impala和Facebook的Presto,引入了一个新的优化器通过SQL处理Hadoop.最近，这些工作只是支持了标准SQL的一个子集以及他们的优化基于规则的限制。比较而言，HAWQ具有成熟的符合SQL标准的接口和基于代价的优化器，在Hadoop查询引擎中都是前所未有特性。在第七节我们会阐述我们的试验研究，从功能和性能的两个方面来表明Orca与其它Hadoop SQL引擎的不同。

# ORCA 体系结构

Orca对于Pivotal数据管理产品是一个新型的查询优化器，包括GPDB和HAWQ。Orca是一个现代的基于Cascades优化框架自顶向下的查询优化器。许多Cascades优化器是与他们的系统紧耦合的，Orca一个独一无二的特征是能够做为一个独立的优化器运行在外部的数据库系统。这个能力对于使用一个优化器来支持不同的计算架构(eg.,MPP和Hadoop)是至关重要的。它也可以把遗留下来的关系优化用在诸如Hadoop的新的查询处理范式。此外做为一个独立的产品来运行能够在不通过数据库系统更精准的测试。

**DXL.** 从数据库系统中解耦优化器需要建立一个通信机制来处理查询。Orca在优化器和数据库系统直接包含一层信息交换框架被称为*Data eXchange Language (DXL)*。这层框架使用基于XML的语言来翻译所需要的交流信息， 例如输入查询，输出计划和元数据。

Query Results



Catalog

Query2DXL

MD Provider

DXL2Plan

**Orca**

**Database System**

DXL Query

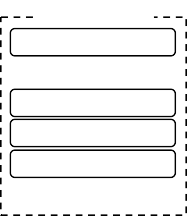
DXL MD DXL Plan

Parser

Executor

**Figure 2:** **Interaction of Orca with database system**

DXL Query DXL Plan



**Orca**

Search

Job Scheduler

**Optimizer Tools**

Operators Property Enforcement

Transformations Card. Estimation Cost Model

MD Cache

**Memo**



File I/O Exception Handling

**GPOS**

Concurrency Control

Memory Manager

**OS**

##### Figure 3: Orca architecture

叠加在DXL之上是一个简单的通信协议来发送初始的查询结构和接收优化计划。DXL主要的好处是把Orca封装成一个独立的产品。

图2表示了Orca和一个外部数据库系统的相互交互。输入到Orca一个DXL查询，输出一个DXL查询计划。优化期间，数据库能够查到元数据（e.g,表定义）。Orca抽象出元数据访问信息，在发送到Orca之前序列化元数据为DXL。元数据也可以以DXL格式的普通文件来使用。

数据库需要包含一个用DXL格式的翻译器。Query2DXL 翻译转换一个查询树到一个DXL查询。DXL2PLAN转换一个DXL计划到一个可执计划。执行这样的翻译是。运行多个系统使用Orca提供适当的翻译是Orca在外部需要做的。

Orca架构是高度扩展的，所有的组件都可以独立替代配置分离。图3表名Orca组件的不同。接下来我们清晰的描述这些组件。

**Memo.** 优化器编译出一个在内存中紧凑的数据结构来生成一个计划被称作Memo。Memo结构有一系列容器组成被称为groups。每个group包含逻辑等价的表达式。Memo的groups捕获不同的查询子目标（e.g 表的过滤器，或者两张表的join）.Group成员被称为group表达式。用不同的逻辑方式来实现group的目标（e.g 不同join orders）。每个group是一个由其他groups做为其孩子的表达式的操作。Memo的递归结构运行编译成更大的可能的计划，我们会在4.1说明。

**查找和作业调度.** Orca使用搜索的机制来在可能和确定的备选方案中找到最少的代价估计。搜索机制能够被用到作业调度器中，它能通过三步建立独立并行的工作单元来执行优化查询。exploration，形成等价的逻辑查询，implementation， 形成物理查询，optimization 评估强制的物理属性和备选方案。我们会在4.2节详细讨论作业调度。

**转换. [**[**13]**](#_bookmark46)通过提供的等价的逻辑表达式（e.g, InnerJoin(A,B) → InnerJoin(B,A))或已经存在的物理表达式（e.g，Join(A,B) → HashJoin(A,B)）来提供转换规则来生成备选计划应用这些转换规则的结果被复制到Memo,可能在已存在的groups添加新的groups或添加新的group表达式。每个转换规则都可以明确在Orca配置激活活停用的自包含组件。

**Property Enforcement.**Orca包括一个可扩展的框架用来描述查询要求和基于已存在的属性说明的角色计划。属性包括逻辑属性（输出字段），物理属性（order排序和数据限制条件）和标量属性（在join条件中的字段）。在优化查询期间，每个属性可从它的子属性的得到。一个优化的子计划可能也满足自己要求的属性（一个IndexScan计划传递已经排序数据），或者强制需要插入到计划来传递要求的属性。该框架也运行每个操作控制安排基于子计划的属性和局部操作行为。在4.1会更详细的描述

**Metadata Cache. 因为元数据改动频繁，每个查询都装载他会带来瓶颈。Orca缓存元数据，如果缓存一些不可利用则只从目录里接收一部分，或者最近时刻已经改变，则将其加载到cache。元数据cache也从优化器中摘取系统详细信息，这在使用测试和调试时非常有用**。

**GPOS. 为了与操作系统不同的API连接，Orca使用一个被称为GPOS的抽象层。GPOS抽象层提供一个扩展的基础包括内存管理，简单并发控制，异常处理，文件I/O和同步数据结构**

# 查询优化

4.1我们描述了Orca的查询优化流程。4.2我们将演示如何并行优化处理

<? xml version =" 1.0 " encoding =" UTF -8 "?>

< dxl:DXLMessage xmlns:dxl=" http: // greenplum . com / dxl/ v1 ">

< dxl:Query >

< dxl:O utputColum ns >

< dxl:Ident ColId =" 0 " Name=" a" Mdid=" 0.23.1.0 "/>

</ dxl:O utputColum ns >

< dxl:Sorting Colum n List >

< dxl:Sorting Colum n ColId =" 0 " OpMdid =" 0.97.1.0 ">

</ dxl:Sorting Colum n List >

< dxl:Distribution Type=" Singleton " />

< dxl:LogicalJoin JoinType=" Inner">

< dxl:LogicalGet >

< dxl:TableD escriptor Mdid=" 0.1639448.1.1 " Name=" T1 ">



*Logical Expression*

Inner Join (T1.a=T2.b)

Get(T1)

Get(T2)

*Initial Memo*

**GROUP 0**

**GROUP 1**

**GROUP 2**

0: Inner Join [1,2]

0: Get(T1) []

0: Get(T2) []

< dxl:Columns >

< dxl:Ident ColId =" 0 " Name=" a" Mdid=" 0.23.1.0 "/>

< dxl:Ident ColId =" 1 " Name=" b" Mdid=" 0.23.1.0 "/>

</ dxl:Columns >

</ dxl:TableD escriptor >

</ dxl:LogicalGet >

< dxl:LogicalGet >

< dxl:TableD escriptor Mdid=" 0.2868145.1.1 " Name=" T2 ">

< dxl:Columns >

< dxl:Ident ColId =" 2 " Name=" a" Mdid=" 0.23.1.0 "/>

< dxl:Ident ColId =" 3 " Name=" b" Mdid=" 0.23.1.0 "/>

</ dxl:Columns >

</ dxl:TableD escriptor >

</ dxl:LogicalGet >

< dxl:Comparison Operator="=" Mdid=" 0.96.1.0 ">

< dxl:Ident ColId =" 0 " Name=" a" Mdid=" 0.23.1.0 "/>

< dxl:Ident ColId =" 3 " Name=" b" Mdid=" 0.23.1.0 "/>

</ dxl:Comparison >

</ dxl:LogicalJoin >

</ dxl:Query >

</ dxl:DXLMessage >

##### Listing 1: DXL query message

SELECT T1.a FROM T1, T2 WHERE T1.a = T2.b ORDER BY T1.a;

* 1. **优化流程**

我们使用如下的运行示例来建立一个查询优化流程：

SELECT T1.a FROM T1,T2 WHERE T1.a = T2.b ORDER BY T1.a

Hashed(T1.a)，T是Hashed(T2.a) 表1用DXL来表示查前，我们给出要求输入的字段，排序的字段，数据分布，逻辑.元数据被解析成元数据 id,允许在优化期间更进一步信息。一个Mdid是一个数据中唯一的特征，一个对象标识和一个版本号。例如，0.96.1.0指的是GPDB整数相等运算符与版本1.0.元数据版本被使用在判断一个被整个查询修改的无效缓存对象。在第5节我们更详细的讨论元数据交换

DXL查询信息装载到Orca, 被解析和转换成在内存中的逻辑表达的树填到Memo。图4表示初始的Memo内容。逻辑表达式创建了3个groups,俩个表和一个innerjoin操作。我们忽略掉join条件。Group 0称为根group,因为他发生在根逻辑表达式。两个groups被捕获到相互之间有依赖。例如，innerjoin[1,2]需要group1和Group2作为孩子。优化如下步骤发生.

**Figure 4:** **Copying-in initial logical expression**

1. **Exploration.** 生成逻辑等价表达式的转换规则被触发。例如，一个Join交换性规则在生成InnerJoin[2,1]到InnerJoin[1,2]被出发。探索结果会在已经存在的group中加入或创建新的group表达式。Memo结构已经内建了重复检测机制，基于表达式的拓扑结构，检测并消除有不同转换创建的任何重复的表达式
2. **Statistics Derivation. 在探索后，Memo对给定的查询维持一个完整的逻辑空间。Orca的数据推演机制在计算统计Memo的groups被触发。在Orca数据对象主要是列统计直方图来估计派生基数和数据倾斜。在Memo结构数据的推演用来避免搜索空间的扩大**

为了获得最高目标group统计推演，Orca选取最高可能的group表达式提供可靠的统计数据。数据保证计算是表达式特定的。例如，一个InnerJoin表达式有少数join条件比另一个等价的InnerJoin有大量join条件表示式更好。（这种情况当多个join orders时能被提升）。基于原理是大量的join条件有更大的可能估算错误被传播放大。计算一个基于估计的置信度需要聚合所有节点给出的表达式置信度。我们目前探索出几种计算Memo结构的置信度分数。

在目标group中选取最有可能的group表达式。Orca递归的触发数据推演在选定的group表达式的子group中。最终，目标group的数据对象从个子group中合并

图5表明了示例的数据推演机制。首先，自顶向下传递父group表达式要求从它的子group统计。例如，InnerJoin(T1,T2) on (a=b) 要求在T1.a和T2.b的统计直方图。被要求的统计直方图被按需通过MD Provider从目录加载。解析成DXL，存进MDcache来服务以后需求。下一步自定向下传递是执行合并子数据对象到一个父数据对象。这个结果在T1.a和T2.b字段的直方图。因为join条件要字段的直方图相互比较。

{ }



Inner Join(a=b) [1,2]

{T1.a} {T2.b}



GROUP 1 GROUP 2

Gather操作从所有的segment聚合所有元组到master。

GatherMerge操作从所有的segment聚合已经排序的数据到master，在进行排序order期间。Redistribute 基于给定参数的hash值来进行跨segment分布操作

图7表示*req.* #*1通过Inner-HashJoin[1,2]优化。对于这个请求，一个备选方案计划是基于join条件匹配子分布，达到元组共同的被join。这个是可以达到的，通过从group1请求*Hashed(T1.a) 分布和group 2请求Hashed(T2.b)分布。两个group被要求提供Any排序。最优子计划被找到，InnerHashJoin结合子属性来决定提供分发和排序。提示对于group2最优机会需要基于T2.b来分布T2。因为最初T2是通过T2.a来分布的。group 1则是简单的Scan，因为T1已经安装T1.a去分布。

[**2**](#_bookmark12)



Inner Join(a=b) [1,2]

Hist(T1.a) Hist(T2.b)

Hist(T1.a) Hist(T2.b)

GROUP 1 GROUP 2

1. ***Top(down statistics requests***

**GROUP 1**



0: Get(T1) []

**Hist(T1.a)**



**GROUP 2**

0: Get(T1) []

**Hist(T2.b)**

1. ***Computed statistics are a8ached to child groups***
2. ***Bo8om(up statistics derivation***

**GROUP 0**

0: Inner Join [1,2]

1: Inner Join [2,1]

****

**Hist(T1.a) Hist(T2.b)**

1. ***Combined statistics are a8ached to parent group***

When it is determined that delivered properties do not

**Figure 5: Statistics derivation mechanism**

构造的统计对象可以在优化的过程增量更新附加到各个group,这是保持统计推导成本管理的关键。

1. **Implementation.** 创建物理实现的逻辑表达式的转换规则被触发。例如， Get2Scan 规则触发生成物理表扫描。同样的，InnerJoin2HashJoin和InnerJoin2NLJoin规则在生成hash和嵌套循环join 执行。
2. **Optimization.** 在这一步，属性被执行和备选方案被计算。优化开始通过提交初始优化请求到Memo的根组指定查询的要求，如结果分布和排序顺序。提交一个r请求到一个g group相当于求得最小代价计划来满足r与g的根物理操作。

对于每个传入的请求，每个物理表达式传递相应的子group表达式这取决于传入的请求和操作者的本地请求。在优化期间，许多相同的请求可以提交到相同的group。Orca 缓存计算请求到一个group hash表中。一个传入的请求只有当hash表内不存在时才会计算。另外，每个物理group表达式维护了一个本地hash表映射传入的请求道相应的子请求。本地的hash表提供了一个连接结构当从Memo提取一个物理计划，我们会在这节的后面讲的。

图6表示示例在Memo里优化请求。最开始的优化请求是*req.* #*1: {Singleton, <T1.a>}，查询结果需要聚合到master根据给定的T1.a的order 。我们也显示group hash表对于每次请求与只是满足最初估计要求的最佳group表达式的联系。黑框表示执行者的操作，出入到Memo提供数据排序和数据分布*

当提交的属性不满足最开始的要求时，不满足的属性一定要执行。Orca的属性执行可以灵活执行，允许每个操作定义子计划和本地操作行为来执行。例如，一个保证有序的NLjoin操作也许不需要在join上面执行排序，如果order是准备提交到外部child。

执行者正在优化时添加到group表达式内容到group。图7表示两种可能计划，满足*req.* #*1通过属性的执行左计划在segment排序join结果，然后在master合并排序结果。右计划从segment合并join结果到master，然后进行排序。俩个方案的不同编译到Memo，由计算模型来计算他们的代价。*

最终，最优计划基于联动结构得到优化请求从Memo里提取。图表6阐述运行示例的计划提取。我们显示了本地hash表相关的group表达式。每个本地的hash表将传入的优化请求映射到相应的子优化请求。

我们首先在根group查找*req.* #*1的*最优group表达式，进行GatherMerge 操作。在hash表中相应的子请求是*req* #*3 。req* #*3 最优的group表达式是排序。req* #*4最优的group表达式是*InnerHashJoin[1,2]。我们因此将排序连接到InnerHashJoin。经过同样的步骤来完成计划得到图6的最终方案。

提取出来的计划以DXL格式序列化并装载到数据库执行。DXL2Plan翻译器在数据库翻译。DXL计划到一个可执行计划根据下属的查询执行框架。

**Multi-Stage Optimization.我们正在进行用Orca实现多级优化。在Orca优化阶段被定义为使用转换规则和超时和代价阈值的子集来完整的优化流程。一个阶段被终止当以下任意一个条件出现，（1）一个计划代价低于代价阈值被找到，超时发生，或转换规则的子集被耗尽。优化阶段的规范可以有Orca的配置来制定。这个技术允许优化资源限制，例如，最昂贵的转换规则配置在运行以后的阶段避免增加优化时间。这种技术也是尽快降低对于复杂的查询搜索空间来获得查询计划基础。**

1Required properties also include output columns, rewindability, common table expressions and data partitioning. We omit these properties due to space constraints.

2There can be many other alternatives (e.g., request children to

be gathered to the master and perform the join there). Orca allows extending each operator with any number of possible optimization alternatives and cleanly isolates these alternatives through property enforcement framework.

Groups Hash Tables Memo Extracted ﬁnal plan

**GROUP 0**



…

…

2: Inner NLJoin [2,1]

3: Inner NLJoin [1,2]

4: Inner HashJoin [1,2]

5: Inner HashJoin [2,1]

…

…

…

…

#4

#7, #10

…

…

…

…

#1

#3

…

…

**6: Sort(T1.a) [0]**

**8: GatherMerge(T1.a) [0]**

**7: Gather[0]**

**GatherMerge(T1.a)**

|  |  |  |
| --- | --- | --- |
| **#** | **Opt. Request** | **Best GExpr** |
| 1 | Singleton, <T1.a> | *8* |
| 2 | Singleton, Any | *7* |
| 3 | Any, <T1.a> | *6* |
| 4 | Any, Any | *4* |

**Sort(T1.a)**

|  |  |
| --- | --- |
| #3 | #4 |
| … | … |

**GROUP 1**

…

…

**2: Sort(T1.a) [1]**

Inner Hash Join

Scan(T1)

**Redistribute(T2.b)**

|  |  |  |
| --- | --- | --- |
| **#** | **Opt. Request** | **Best GExpr** |
| 5 | Any, Any | *1* |
| 6 | Replicated, Any | *3* |
| 7 | Hashed(T1.a), Any | *1* |
| 8 | Any, <T1.a> | *2* |

|  |  |  |
| --- | --- | --- |
| 1: Scan(T1)[] | | |
| #7 |  |  |
| … | … | |

|  |  |  |  |
| --- | --- | --- | --- |
| **3: Replicate[1]** | | | |
|  | … | … |  |

**GROUP 2**



**2: Replicate[2] 3: Redistribute(T2.b) [2]**

|  |  |  |
| --- | --- | --- |
| **#** | **Opt. Request** | **Best GExpr** |
| 9 | Any, Any | *1* |
| 10 | Hashed(T2.b), Any | *3* |
| 11 | Replicated, Any | *2* |

|  |  |  |  |
| --- | --- | --- | --- |
| 1: Scan(T2)[] | | | |
|  | #9 |  |  |
|  | … | … |

|  |  |  |
| --- | --- | --- |
|  | … | … |

|  |  |
| --- | --- |
| #10 | #9 |
| … | … |

Scan(T2)

**Figure 6:** **Processing optimization requests in the Memo**

{Singleton, <T1.a>}

Inner Hash Join [1,2]

{Singleton, <T1.a>}

{Hashed(T1.a), Any} {Hashed(T2.b), Any}

Inner Hash Join

Scan(T1)

|  |  |  |
| --- | --- | --- |
| **Redistribute(T2.b)** | | |
|  | Scan(T2) |  |

### 



GROUP 1



GROUP 2

1. *Passing requests to child groups (b) Combining child groups best plans*

并行查询优化最关键的是从越来越多核的高级CPU总获益

Orca是一个多核优化器。优化过程被分解为一些小的工作单元称为优化工作。Orca当前有7个不同类型的优化工作

**GatherMerge(T1.a)**

**Sort(T1.a)**

#### 

Inner Hash Join

**Redistribute(T2.b)**

Scan(T1)

Scan(T2)

**Sort(T1.a)**

**Gather**

#### 

Inner Hash Join

**Redistribute(T2.b)**

Scan(T1)

Scan(T2)

* + Exp(g): 对group g里所有的group表达式生成逻辑等价的表达式。
  + Exp(gexpr): 对一个group表达式 gexpr生成逻辑等价表达式。
  + Imp(g): 对group g里所有的group表达式生成执行Generate implementations of all group expres-

*(c) Enforcing missing properties to satisfy {Singleton, <T1.a>} request*

**Figure 7:** **Generating InnerHashJoin plan alternatives**

**Query Execution.** A一个最终计划的副本被分发到每个segment。在分布式的查询执行期间，每个segment的发布执行者充当数据发送者和接收者。例如，一个Redistribute(T2.b) 实例在Ssegment运行，基于T2.b的hash值来将元组发送到其他segment上。同时也从其它的Redistribute(T2.b) 实例接收元组。

# Parallel Query Optimization

查询优化可能是在数据库中最占用CPU的程序。有效的使用CPU来转化成更好的查询计划以此得到更好的系统性能。

sions in group g.

* + - Imp(gexpr): 对gexpr生成备选执行方案
    - Opt(g, req): 返回一个以group g操作为根的最小代价估计的计划，同时满足req请求。
    - Opt(gexpr, req): 返回一个以gexpr为根的最小代价估计的计划，同时满足req请求。
    - Xform(gexpr, t) 使用规则t来转换gexpr 表达式

对于给定查询，可能给每个类型创建成百上千个作业实例。这给处理相互依赖的工作带来了挑战。例如，一个group表达式直到它的子表达式优化后才能优化。图8显示了部分工作图，group表达式优化在req0优化触发一个依赖工作的深度树。For example, a group expression cannot be optimized until its child groups are also optimized. Figure [8](#_bookmark17) shows a partial job graph, where optimization of group g0 under optimization request req0 triggers a deep tree of dependent jobs. 依赖关系被编为父子连接。一个父job不能在它的子job先完成。当子job被处理时，父job需要挂起。如果不依赖其它job，允许子job选取可利用的线程并行运行。当所有的子job完成，挂起的父job被通知恢复处理。

Orca 包含了一个专用的作用调度器，从头开始设计以最大限度的工作依赖项关系和所提供的基础设施为并行查询优化。

### 

Opt(g0, req0)

… *optimize group expressions in g0*



Imp(g0)

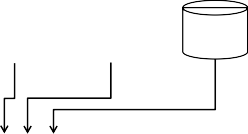
Opt(g0.gexpr, req0)

…

File based Provider

Vanilla Postgres

GPDB HAWQ PG



MD Provider PlugIins

MD Cache

Orca

Oracle, MS SQL DB2, MySQL, etc.

DB

Hadoop, MongoDB

NoSQL

Exp(g0)

…

…

… *optimize children*

… *of g0.gexpr*

Opt(g1, req1)

…

Imp(g1)

Opt(g1.gexpr, req1)

… …

*optimize group expressions in g1*

…

Exp(g1)

Imp(g1.gexpr)

… *explore group*

Exp(g1.gexpr)

… *expressions in g1*

*implement group expressions in g1*

…

… … …

Xform(g1.gexpr, t)

Xform(g1.gexpr, t’)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | MD Accessor | | |
| MD Accessor | | |  | |
|  | MD Accessor | | |  |
| Optimization Engine | | | | |

… … *exploration rules implementation*

*explore children*

*of g1.gexpr*

*of g1.gexpr*

*rules of g1.gexpr*

**Figure 8:** **Optimization jobs dependency graph**

调度器提供API来定义可重入的程序，可以通过可用的处理现场优化工作。它也维护一个工作依赖项关系图，找到机会并行（e.g 在不同的group运行转换）。当依赖的job已经停止通知挂起的job。在并行查询优化期间，若修改一个Memo组的多个并发请求，可能有不同的优化请求触发。为了减少同步开销之间的工作，同样的目标（e.g 探索同样的group），job不知道其他的已经存在。当一个目标优化工作正在处理当中时， 同样的目标的其他传入的job被迫等待，直到收到通知有关正在运行的job完成。在这一点上，已挂起的作业拾起完成作业的结果。此功能能够附加作业到每个组，只要已经存在一个相同目标的活跃的job，传入的job就需要排队

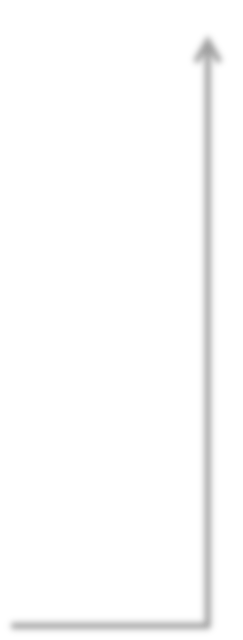
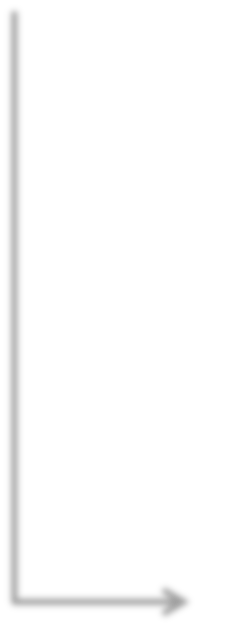
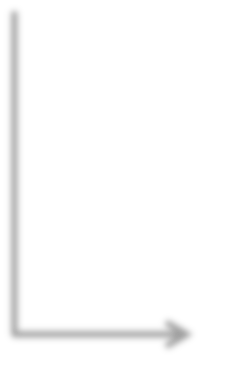
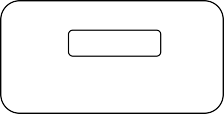
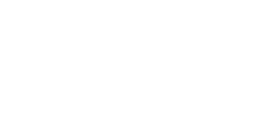
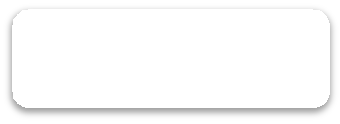
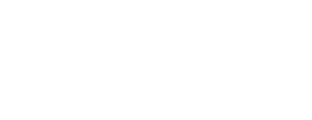
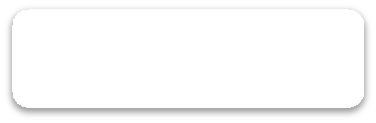
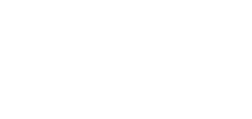
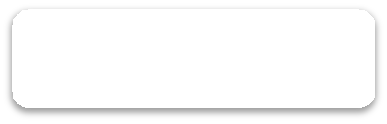
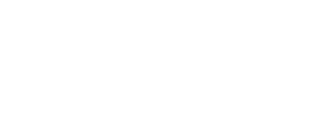
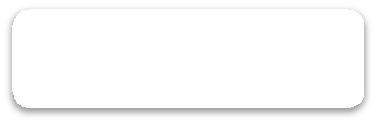
# METADATA EXCHANGE

Orca被设计成可以在外部数据库系统工作。优化程序和数据库系统之间的其中一个重点是相互作用的元数据交换。例如，优化器可能需要知道是否索引的定义对给定的表来制定高效的查询计划。对元数据的访问被通过元数据提供程序的集合是特定于系统的插件来从数据库系统中检索元数据。

图9 显示Orca如何与不同后端系统交换元数据。在查询优化期间，所有元数据对象对象通过内存的缓存来访问，是固定时优化完成或引发错误。对元数据对象的所有访问都是通过 MD 访问器，它在优化会话中访问的对象并可确保他们都在释放时不再需要它们工作。MD 访问器也是负责透明地获取元数据数据从外部 MD 提供商如果请求的元数据对象已经不在缓存中。不同MD访问器服务不同的优化会话，可能有不同的外部MD提供者来获取元数据

除了系统特定的提供程序，Orca实现了一个基于文件的 MD 提供从 DXL 文件加载元数据, 消除了需要访问一个活的后端系统。Orca包括自动化的工具获取元数据，优化器需要到得到一个最小的 DXL 文件。

##### Figure 9: Metadata exchange framework



MD Cache

Optimization Engine

Dumpﬁle

DXLStacks

DXL Query DXL MD DXL Plan

**Orca**

**Figure 10:** **Replay of AMPERe dump**

# VERIFIABILITY

Testing a query optimizer is a task as challenging as build- ing one. Orca is built with testing in mind since the early development phase. There is a built-in testing scheme that makes it difficult for developers to introduce regressions as part of adding new features, and makes it simple for test en- gineers to add test cases to be verified with every build. In addition, we leverage several tools and testing frameworks we built to assure the quality and verifiability of Orca, in- cluding a cardinality estimation testing framework, a num- ber of benchmark tests at various scales, a data generator that can generate data by reversing database statistics [[24](#_bookmark57)], and two unique testing tools we discuss next.

The first tool, discussed in Section [6.1](#_bookmark22), is automatic cap- turing and replaying of optimizer’s anomalies. The second tool, discussed in Section [6.2](#_bookmark25), implements an automated method to measure the accuracy of optimizer’s cost model.

# Minimal Repros

AMPERe [[3](#_bookmark36)] is a tool for Automatic capture of Minimal Portable and Executable Repros. The motivation for build- ing AMPERe was to be able to reproduce and debug cus-

<? xml version =" 1.0 " encoding =" UTF -8 "?>

< dxl:DXLMessage xmlns:dxl=" http: // greenplum . com / dxl/ v1 ">

< dxl:Thread Id=" 0 ">

< dxl:Stacktrace >

1. 0 x000e8106df gpos::CException::Raise
2. 0 x000137d853 CO ptTasks::Pv O ptim izeTask
3. 0 x000e81cb1c gpos::CTask::Execute
4. 0 x000e8180f4 gpos::CW orker::Execute
5. 0 x000e81e811 gpos:: CA uto Task Proxy::Execute

</ dxl:Stacktrace >

< dxl:TraceFlags Value=" gp\_ optim izer\_ hashjoin "/>

< dxl:Metadata System Ids=" 0. GPDB ">

< dxl:Type Mdid=" 0.9.1.0 " Name=" int4 " IsRedistributable =" true" Length =" 4 " />

< dxl:RelStats Mdid=" 2.688.1.1 " Name=" r" Rows=" 10 "/>

< dxl:Relation Mdid=" 0.688.1.1 " Name=" r"

D istribution Policy =" Hash" D istribution Colum ns =" 0 ">

< dxl:Columns >

< dxl:Column Name=" a" Attno =" 1 " Mdid=" 0.9.1.0 "/>

</ dxl:Columns >

</ dxl:Relation >

</ dxl:Metadata >

< dxl:Query >

< dxl:O utputColum ns >

< dxl:Ident ColId =" 1 " Name=" a" Mdid=" 0.9.1.0 "/>

</ dxl:O utputColum ns >

< dxl:LogicalGet >

< dxl:TableD escriptor Mdid=" 0.688.1.1 " Name=" r">

< dxl:Columns >

< dxl:Column ColId =" 1 " Name=" a" Mdid=" 0.9.1.0 "/>

</ dxl:Columns >

</ dxl:TableD escriptor >

</ dxl:LogicalGet >

</ dxl:Query >

</ dxl:Thread >

</ dxl:DXLMessage >

##### Listing 2: Simplifted AMPERe dump

tomer issues in the optimizer without having access to the customer production system.

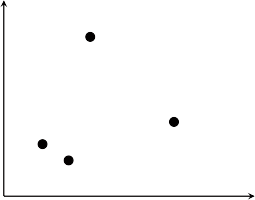
An AMPERe dump is automatically triggered when an un- expected error is encountered, but can also be produced on demand to investigate suboptimal query plans. The dump captures the minimal amount of data needed to reproduce a problem, including the input query, optimizer configura- tions and metadata, serialized in DXL (cf. Section [3](#_bookmark4)). If the dump is generated due to an exception, it also includes the exception’s stack trace.

Listing [2](#_bookmark23) shows an example of a simplified AMPERe dump. The dump includes only the necessary data to reproduce the problem. For example, the dump captures the state of MD Cache which includes only the metadata acquired during the course of query optimization. AMPERe is also built to be extensible. Any component in Orca can register itself with the AMPERe serializer to generate additional information in the output dump.

AMPERe allows replaying a dump outside the system where it was generated. Any Orca instance can load the dump to retrieve the input query, metadata and configura- tion parameters in order to invoke an optimization session identical to the one that triggered the problematic situation at hand. This process is depicted in Figure [10](#_bookmark20), where the optimizer loads the input query from the dump, creates a file-based MD Provider for the metadata, sets optimizer’s configurations and then spawns the optimization threads to reproduce the problem instantly.

AMPERe is also used as a testing framework, where a dump acts as a test case that contains an input query and its expected plan. When replaying the dump file, Orca

Actual Cost



*p4*

*p2*

*p*

*3*

*p1*

Estimated Cost

**Figure 11:** **Plan space**

might generate a plan different from the expected one (e.g., because of changes in the cost model). Such discrepancy causes the test case to fail, and triggers investigating the root cause of plan difference. Using this framework, any bug with an accompanying AMPERe dump, whether filed by internal testing or through customer reports, can be au- tomatically turned into a self-contained test case.

# Testing Optimizer Accuracy

The accuracy of Orca’s cost model can be impacted by a number of error sources including inaccurate cardinality estimates and not properly adjusted cost model parameters. As a result, cost model provides imperfect prediction of the wall clock time for the execution of a plan. Quantifying opti- mizer’s accuracy is crucial to avoid performance regressions introduced by bug fixes and newly added features.

Orca includes a built-in tool called TAQO [[15](#_bookmark47)] for Testing the Accuracy of Query Optimizer. TAQO measures the abil- ity of optimizer’s cost model to order any two given plans correctly, i.e., the plan with the higher estimated cost will in- deed run longer. For example, in Figure [11](#_bookmark24), the optimizer or-

ders (*p*1 *, p*3) correctly, since their actual cost is directly pro- portional to computed cost estimates. On the other hand, the optimizer orders (*p*1 *, p*2) incorrectly, since their actual cost is inversely proportional to computed cost estimates.

TAQO measures the optimizer’s accuracy by costing and executing plans that the optimizer considers when optimiz- ing a given query. Evaluating each single plan in the search space is infeasible, in general. This limitation can be over- come by sampling plans uniformly from the search space. Optimization requests’ linkage structure (cf. Section [4.1](#_bookmark9)) provides the infrastructure used by TAQO to build a uni- form plan sampler based on the method introduced in [[29]](#_bookmark62). Given a sample of plans from the search space of a given query, TAQO computes a correlation score between the rank- ing of sampled plans based on estimated costs and their ranking based on actual costs. The correlation score com- bines a number of measures including importance of plans (the score penalizes optimizer more for cost miss-estimation of very good plans), and distance between plans (the score does not penalize optimizer for small differences in the es- timated costs of plans that are actually close in execution time). The correlation score also allows benchmarking the optimizers of different database systems to evaluate their relative quality. We discuss the testing methodology imple-

mented in TAQO in more detail in [[15](#_bookmark47)].

# EXPERIMENTS

In our experimental study, we chose to conduct an end- to-end evaluation of a database system equipped with Orca, rather than evaluating Orca’s individual components, to highlight the added value of our new query optimizer. We

first compare Orca to the legacy query optimizer of Pivotal GPDB. We then compare Pivotal HAWQ (which employs Orca in its core) to other popular SQL on Hadoop solutions.

# TPC-DS Benchmark

Our experiments are based on the TPC-DS benchmark [[1](#_bookmark34)]. TPC-DS is a widely adopted decision support benchmark that consists of a set of complex business analytic queries. It has superseded the well-known TPC-H by providing a much richer schema and a larger variety of business prob- lems ranging from business reporting, ad-hoc exploration, iterative queries to data mining. In our development process we have observed that TPC-H often lacks the sophistication of the workload from our enterprise customers. On the other hand, TPC-DS with its 25 tables, 429 columns and 99 query templates can well represent a modern decision-supporting system and is an excellent benchmark for testing query op- timizers. The rich SQL syntax (WITH clause, window func- tions, subqueries, outer joins, CASE statement, Intersect, Except, etc.) in the TPC-DS queries is a serious SQL com- pliance test for any query engine.

# MPP Databases

In this part, we compare the performance of Orca with the GPDB legacy query optimizer (a.k.a. *Planner*) that inherits part of its design from the PostgreSQL optimizer. The *Plan- ner* is a robust optimizer that has been serving hundreds of production systems well, and has been improving over the past decade.

## Experiment Setup

For the comparison between Orca and *Planner*, we use a cluster of 16 nodes connected with 10Gbps Ethernet. Each node has dual Intel Xeon processors at 3.33GHz, 48GB of RAM and twelve 600GB SAS drives in two RAID-5 groups. The operating system is Red Hat Enterprise Linux 5.5.

We installed two isolated instances of the same version of GPDB (one uses Orca, and the other uses *Planner*). We use 10 TB TPC-DS benchmark with partitioned tables for performance evaluation.

## Performance

We generated 111 queries out of the 99 templates of TPC- DS. Both Orca and *Planner* support all the queries in their original form without any re-writing. The full SQL compli- ance provides maximum compatibility of BI tools and ease- of-use for data analysts from different backgrounds. As we show in the SQL on Hadoop experiments in Section [7.3](#_bookmark27), many Hadoop SQL engines currently support only a small subset of TPC-DS queries out of the box.

The performance speed up of Orca compared to *Planner* for all queries is shown in Figure [12](#_bookmark28), where bars above the speed-up ratio of 1 indicate performance improvement of Orca. We observe that Orca is able to produce similar or better query plan for 80% of the queries. For the entire TPC-DS suite, Orca shows a 5x speed-up over *Planner*.

In particular, for 14 queries Orca achieves a speed-up ra- tio of at least 1000x - this is due to a timeout we enforced at 10000 seconds. These queries took more than 10000 sec- onds with *Planner*’s plan while they were able to finish with Orca’s plan in minutes.

The performance improvement provided by Orca is due to a combination of salient features including the following:

* + - * *Join Ordering.* Orca includes a number of join or- dering optimizations based on dynamic programming,

left-deep join trees and cardinality-based join ordering.

* + - * *Correlated Subqueries.* Orca adopts and extends a uni- fied representation of subqueries to detect deeply cor- related predicates and pull them up into joins to avoid repeated execution of subquery expressions.
      * *Partition Elimination.* Orca introduces a novel frame- work for on-the-fly pruning of partitioned tables [[2](#_bookmark35)]. This feature is implemented by extending Orca’s en- forcers framework to accommodate new properties.
      * *Common Expressions.* Orca introduces a new producer-consumer model for WITH clause. The model allows evaluating a complex expression once, and con- suming its output by multiple operators.

The interplay of the previous features is enabled by Orca’s architecture and components abstraction. Each feature is designed, implemented and tested with minimal changes in the behavior of other features. The combined benefit and clean interaction of features are manifested by Figure [12](#_bookmark28).

For a smaller number of queries, Orca produced sub- optimal plans with up to 2x slow down compared to *Planner*. These sub-optimal plans are partly due to cardinality esti- mation errors or sub-optimal cost model parameters that need further tuning. We are actively investigating these is- sues and constantly improving Orca.

We have also measured optimization time and Orca’s memory footprint when using the full set of transformation rules. The average optimization time is around 4 seconds, while the average memory footprint is around 200 MB. As we mentioned in Section [4.1](#_bookmark9), our ongoing work involves im- plementing techniques to shortcut optimization and improve resource consumption for complex queries.

# SQL on Hadoop

Hadoop has quickly become a popular analytics ecosys- tem due to its scalability. In recent years, many Hadoop systems have been developed with SQL or SQL-like query in- terfaces. In this section, we compare the performance of Piv- otal HAWQ (powered by Orca) against three Hadoop SQL engines: Impala [[17](#_bookmark50)], Presto [[7](#_bookmark40)], and Stinger [[16](#_bookmark49)]. Please refer to Section [8](#_bookmark32) for a discussion on these systems.

## Experiment Setup

The experiments are conducted on a cluster with 10 nodes; two for HDFS name node and coordinator services of SQL engines, and eight for HDFS data nodes and worker nodes. Each node has dual Intel Xeon eight-core processors at 2.7GHz, 64GB of RAM and 22 900GB disks in JBOD. The operating system is Red Hat Enterprise Linux 6.2.

We used CDH 4.4 and Impala 1.1.1 for Impala, Presto 0.52, and Hive 0.12 for Stinger. We made our best efforts to tune the optimal configurations of each system, including enabling short circuit read, allocating as much memory as possible to workers and having one standalone node for co- ordinator services. For HAWQ, we used Pivotal HD version

1.1 in our experiment.

Optimization of TPC-DS queries in different systems turned out to be quite challenging because systems currently have limited SQL support. For example, Impala does not yet support window functions, ORDER BY statement without

5



Orca Speed-up ratio

1.0

0.5

1

3

5

6

8

10

12

14a 15

17

18a 20

22

23

25

27

28

30

32

34

36

37

39

41

43

45

47

49

51

52

54

56

58

60

62

64

66

67a 69

70a 72

74

76

77a 79

80a 82

84

86

87

89

91

93

95

97

99

Query ID (every other ID is shown due to space limit)

**Figure 12:** **Speed-up ratio of Orca vs Planner (TPC-DS 10TB)**

100 100

10 10

HAWQ speed-up ratio

HAWQ speed-up ratio

1

3

4

7

11

15

19

21

22a 25

26

27a 29

37

42

43

46

50

52

54

55

59

68

74

75

76

79

82

85

93

96

97

Query ID

(\*) Query runs out of memory in Impala

**Figure 13:** **HAWQ vs Impala (TPC-DS 256GB)**

1

3 12 17 18 20 22 25 29 37 42 52 55 67 76 82 84 86 90 98

Query ID

**Figure 14:** **HAWQ vs Stinger (TPC-DS 256GB)**

LIMIT and some analytic functions like ROLLUP and CUBE. Presto does not yet support non-equi joins. Stinger cur- rently does not support WITH clause and CASE statement. In addition, none of the systems supports INTERSECT, EXCEPT, disjunctive join conditions and correlated subqueries. These unsupported features have forced us to rule out a large num- ber of queries from consideration.

After excluding unsupported queries, we needed to re- write the remaining queries to work around parsing limita- tions in different systems. For example, Stinger and Presto do not support implicit cross-joins and some data types. Af- ter extensive filtering and rewriting, we finally managed to

111

0

# of queries

111 111

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | 31 | 20 | 12 | optimization execution  19 19 | | |
| 0 | | | |  |  |

HAWQ Impala Presto Stinger

**Figure 15:** **TPC-DS query support**

get query plans for 31 queries in Impala, 19 queries in Stinger and 12 queries in Presto, out of the total 111 queries.

## 7.3.2 Performance

Our first attempt was to evaluate the different systems us- ing 10TB TPC-DS benchmark. However, most of the queries from Stinger did not return within a reasonable time limit, and almost all the queries from Impala and Presto failed due to an out of memory error. This mainly happens due to the inability of these systems to spill partial results to disk when an operator’s internal state overflows the memory limits.

To obtain a better coverage across the different systems, we used 256GB TPC-DS benchmark, considering that the total working memory of our cluster is about 400GB (50GB

*×* 8 nodes). Unfortunately even with this setting, we were

unable to successfully run any TPC-DS query in Presto (al- though we managed to run much simpler join queries in Presto). For Impala and Stinger, we managed to run a num- ber of TPC-DS queries, as we discuss next.

Figure [15](#_bookmark31) summarizes the number of supported queries in all the systems. We show the number of queries that each system can optimize (i.e., return a query plan), and the number of queries that can finish execution and return query results for the 256GB dataset.

Figure [13](#_bookmark29) and Figure [14](#_bookmark30) show the speedup ratio of HAWQ over Impala and Stinger. Since not all the queries are

supported by the two systems, we only list the successful

queries. The bars marked with ‘*∗*’ in Figure [13](#_bookmark29) indicate the queries that run out of memory. For query 46, 59 and 68, Impala and HAWQ have similar performance.

For queries where HAWQ has the most speedups, we find that Impala and Stinger handle join orders as literally spec- ified in the query, while Orca explores different join orders to suggest the best one using a cost-based approach. For ex- ample in query 25, Impala joins two fact tables store\_sales and store\_returns first and then joins this huge interme- diate results with another fact table catalog\_sales, which is quite inefficient. In comparison, Orca joins the fact tables with dimension tables first to reduce intermediate results. In general, join ordering is a non-trivial optimization that requires extensive infrastructure on the optimizer side.

Impala recommends users to write joins in the descend- ing order of the sizes of joined tables. However, this sug- gestion ignores the filters (which may be selective) on the tables, adds a non-trivial overhead to a database user for complex queries and may not be supported by BI tools that automatically generates queries. The lack of join ordering optimizations in a query optimizer has negative impact on the quality of produced plans. Other possible reasons for HAWQ speedups such as resource management and query execution is outside the scope of this paper.

The average speedup ratio of HAWQ in this set of exper- iments is 6x against Impala and 21x against Stinger. Note that the queries listed in Figure [13](#_bookmark29) and Figure [14](#_bookmark30) are rel- atively simple queries in TPC-DS benchmark. More com- plex queries (e.g., queries with correlated sub-queries) are not supported by other systems yet, while being completely supported by Orca. We plan to revisit TPC-DS benchmark performance evaluation in the future when all the queries are supported by other systems.

# RELATED WORK

Query optimization has been a fertile field for several ground breaking innovations over the past decades. In this section, we discuss a number of foundational query optimiza- tion techniques, and recent proposals in the space of MPP databases and Hadoop-based systems.

# Query Optimization Foundations

*Volcano Parallel Database* [[12](#_bookmark45)] introduced basic princi- ples for achieving parallelism in databases. The proposed framework introduced *exchange* operators, which enable two means of parallelism, namely inter-operator parallelism, via pipelining, and intra-operator parallelism, via partitioning of tuples across operators running on different processes. The proposed design allows each operator to execute indepen- dently on local data, as well as work in parallel with other copies of the operator running in other processes. Several MPP databases [[6](#_bookmark39), [8](#_bookmark41), [18](#_bookmark51), [20](#_bookmark53), [23](#_bookmark56)] make use of these principles to build commercially successful products.

*Cascades* [[13](#_bookmark46)] is an extensible optimizer framework whose principles have been used to build MS-SQL Server, SCOPE [[6](#_bookmark39)], PDW [[23](#_bookmark56)], and Orca, the optimizer we present in this paper. The popularity of this framework is due to its clean separation of the logical and physical plan spaces. This is primarily achieved by encapsulating operators and transformation rules into self-contained components. This modular design enables Cascades to group logically equiv- alent expressions to eliminate redundant work, allows rules to be triggered on demand in contrast to Volcano’s [[14](#_bookmark48)] ex- haustive approach, and permits ordering the application of rules based on their usefulness to a given operator.

Building on the principles of Cascades, a parallel opti- mization framework is proposed in [[30](#_bookmark63)] that enables building Cascades-like optimizers for multi-core architectures. The parallel query optimization framework in Orca (cf. Sec- tion [4.](#_bookmark16)2) is based on the principles introduced in [[30](#_bookmark63)].

# SQL Optimization On MPP Databases

The exponential growth in the amount of data being stored and queried has translated into a wider usage of *Massively Parallel Processing* (MPP) systems such as Tera- data [[27](#_bookmark60)], Oracle’s Exadata [[31](#_bookmark64)], Netezza [[25](#_bookmark58)], Pivotal Green- plum Database [[20](#_bookmark53)], and Vertica [[18](#_bookmark51)]. Due to space limita- tion, we summarize a few recent efforts in re-designing the query optimizer to meet the challenges of big data.

*SQL Server Parallel Data Warehouse (PDW)* [[23](#_bookmark56)] makes extensive re-use of the established Microsoft’s SQL Server optimizer. For each query, PDW triggers an optimization request to the SQL Server optimizer that works on a *shell database* that maintains only the metadata and statistics of the database and not its user data. The plan alternatives ex- plored by SQL Server optimizer are then shipped to PDW’s *Data Movement Service* (DMS) where these logical plans are retrofitted with distribution information. While this ap- proach avoids building an optimizer from scratch, it makes debugging and maintenance harder since the optimization logic is spread across two different processes and codebases. *Structured Computations Optimized for Parallel Execution (SCOPE)* [[6](#_bookmark39)] is Microsoft’s data analysis platform that lever- ages characteristics of both parallel databases and MapRe- duce systems. SCOPE’s scripting language, like Hive [[28](#_bookmark61)], is based on SQL. SCOPE is developed for the Cosmos dis- tributed data platform that employs an append-only file sys- tem, while Orca is designed with a vision to work with mul-

tiple underlying data management systems.

*SAP HANA* [[11](#_bookmark44)] is a distributed in-memory database sys- tem that handles business analytical and OLTP queries. An- alytical queries in MPP databases can potentially generate a large amount of intermediate results. Concurrent analyti- cal queries can exhaust the available memory, most of which is already consumed to store and index raw data, and will trigger data to be spilled to disk that results in a negative impact on query performance.

*Vertica* [[18](#_bookmark51)] is the commercialized MPP version of the C- Store project [[26](#_bookmark59)] where the data is organized into *projec- tions* and each projection is a subset of table attributes. The initial *StarOpt* and its modified *StratifiedOpt* optimizer were custom designed for queries over star/snowflake schemas, where the join keys of the same range are co-located. When data co-location cannot be achieved, the pertinent projec- tions are replicated on all nodes to improve performance, as addressed by Vertica’s *V2Opt* optimizer.

# SQL On Hadoop

The classic approach of executing SQL on Hadoop is converting queries into MapReduce jobs using Hive [[28](#_bookmark61)]. MapReduce performance can be unsatisfactory for interac- tive analysis. Stinger [[16](#_bookmark49)] is an initiative to optimize query evaluation on Hadoop by leveraging and extending Hive. This approach, however, could entail significant redesign of MapReduce computing framework to optimize passes on data, and materialization of intermediate results on disk.

Several efforts have addressed interactive processing on Hadoop by creating specialized query engines that allow

SQL-based processing of data in HDFS without the need to use MapReduce. Impala [[17](#_bookmark50)], HAWQ [[21](#_bookmark54)] and Presto [[7]](#_bookmark40) are key efforts in this direction. These approaches are dif- ferent in the design and capabilities of their query optimiz- ers and execution engines, both of which are differentiating factors for query performance. Co-location of DBMS and Hadoop technologies allows data to be processed natively on each platform, using SQL in the DBMS and MapReduce in HDFS. Hadapt [[4](#_bookmark37)] pioneered this approach. Microsoft has also introduced PolyBase [[10](#_bookmark43)] to offer the ability to join ta- bles from PDW [[23](#_bookmark56)] with data on HDFS in order to optimize data exchange from one platform to another.

AsterixDB [[5](#_bookmark38)] is an open-source effort to efficiently store, index and query semi-structured information based on a NoSQL style data model. Currently, AsterixDB’s query planner is driven by user hints rather than a cost driven ap- proach like Orca. Dremel [[19]](#_bookmark52) is a scalable columnar solution from Google used to analyze outputs of MapReduce pipeline. Dremel provides a high level scripting languages similar to AsterixDB’s scripting language (AQL) [[5](#_bookmark38)] and SCOPE [[6](#_bookmark39)] to process read-only nested data.

# SUMMARY

With the development of Orca, we aimed at developing a query optimization platform that not only represents the state-of-the-art but is also powerful and extensible enough to support rapid development of new optimization techniques and advanced query features.

In this paper, we described the engineering effort needed to build such a system entirely from scratch. Integrating into Orca a number of technical safeguards posed a significant investment, yet, has paid already significant dividends in the form of the rapid pace of its development and the resulting high quality of the software. Orca’s modularity allows it to be adapted easily to different data management systems by encoding a system’s capabilities and metadata using a clean and uniform abstraction.

# REFERENCES

[1]TPC-DS. <http://www.tpc.org/tpcds>, 2005.

[2]L. Antova, A. ElHelw, M. Soliman, Z. Gu, M. Petropoulos, and F. Waas. Optimizing Queries over Partitioned Tables in MPP Systems. In *SIGMOD*, 2014.

[3]L. Antova, K. Krikellas, and F. M. Waas. Automatic Capture of Minimal, Portable, and Executable Bug Repros using AMPERe. In *DBTest*, 2012.

[4]K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and

E. Paulson. Efficient Processing of Data Warehousing Queries in a Split Execution Environment. In *SIGMOD*, 2011.

[5]A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li,

N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. ASTERIX: Towards a Scalable, Semistructured Data Platform for Evolving-world Models. *Dist. Parallel Databases*, 29(3), 2011.

[6]R. Chaiken, B. Jenkins, P.- ˚A. Larson, B. Ramsey,

D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *PVLDB*, 1(2), 2008.

[7]L. Chan. Presto: Interacting with petabytes of data at Facebook. [http://prestodb.io](http://prestodb.io/), 2013.

[8]Y. Chen, R. L. Cole, W. J. McKenna, S. Perfilov, A. Sinha, and E. Szedenits, Jr. Partial Join Order Optimization in the Paraccel Analytic Database. In *SIGMOD*, 2009.

[9]J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser,

P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li,

A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan,

R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor,

R. Wang, and D. Woodford. Spanner: Google’s Globally-distributed Database. In *OSDI*, 2012.

[10]D. J. DeWitt, A. Halverson, R. Nehme, S. Shankar,

J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split Query Processing in Polybase. In *SIGMOD*, 2013.

[11]F. F ¨arber, S. K. Cha, J. Primsch, C. Bornh¨ovd, S. Sigg, and

W. Lehner. SAP HANA Database: Data Management for Modern Business Applications. *SIGMOD Rec.*, 40(4), 2012.

[12]G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *SIGMOD*, 1990.

[13]G. Graefe. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.*, 18(3), 1995.

[14]G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*, 1993.

[15]Z. Gu, M. A. Soliman, and F. M. Waas. Testing the Accuracy of Query Optimizers. In *DBTest*, 2012.

[16]Hortonworks. Stinger, Interactive query for Apache Hive.

<http://hortonworks.com/labs/stinger/>, 2013. [17]M. Kornacker and J. Erickson. Cloudera Impala: Real-Time Queries in Apache Hadoop, for Real.

[http://www.cloudera.com/content/cloudera/en/](http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html) [products-and-services/cdh/impala.html](http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html), 2012.

[18]A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver,

L. Doshi, and C. Bear. The Vertica Analytic Database: C-store 7 Years Later. *VLDB Endow.*, 5(12), 2012.

[19]S. Melnik, A. Gubarev, J. J. Long, G. Romer,

S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. *PVLDB*, 3(1):330–339, 2010.

[20]Pivotal. Greenplum Database. [http://www.gopivotal.com/products/pivotal-greenplum-](http://www.gopivotal.com/products/pivotal-greenplum-database) [database](http://www.gopivotal.com/products/pivotal-greenplum-database), 2013.

[21]Pivotal. HAWQ. [http://www.gopivotal.com/sites/](http://www.gopivotal.com/sites/default/files/Hawq_WP_042313_FINAL.pdf) [default/files/Hawq\_WP\_042313\_FINAL.pdf](http://www.gopivotal.com/sites/default/files/Hawq_WP_042313_FINAL.pdf), 2013.

[22]P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, 1979.

[23]S. Shankar, R. Nehme, J. Aguilar-Saborit, A. Chung,

M. Elhemali, A. Halverson, E. Robinson, M. S. Subramanian, D. DeWitt, and C. Galindo-Legaria. Query Optimization in Microsoft SQL Server PDW. In *SIGMOD*, 2012.

[24]E. Shen and L. Antova. Reversing Statistics for Scalable Test Databases Generation. In *Proceedings of the Sixth International Workshop on Testing Database Systems*, pages 7:1–7:6, 2013.

[25]M. Singh and B. Leonhardi. Introduction to the IBM Netezza Warehouse Appliance. In *CASCON*, 2011.

[26]M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen,

1. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O’Neil, P. E. O’Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-oriented DBMS. In *VLDB*, 2005.

[27]Teradata. <http://www.teradata.com/>, 2013.

[28]A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka,

1. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - A Petabyte Scale Data Warehouse using Hadoop. In *ICDE*, 2010.

[29]F. Waas and C. Galindo-Legaria. Counting, Enumerating, and Sampling of Execution Plans in a Cost-based Query Optimizer. In *SIGMOD*, 2000.

[30]F. M. Waas and J. M. Hellerstein. Parallelizing Extensible Query Optimizers. In *SIGMOD Conference*, pages 871–878, 2009.

[31]R. Weiss. A Technical Overview of the Oracle Exadata Database Machine and Exadata Storage Server, 2012.