**论文编码：RUC-BK-080901-2013202585**

**中国人民大学本科学生毕业论文**

**（毕业设计）**

**基于Greenplum的时间敏感查询可靠性优化研究**

**作 者：王传雯**

**学 院：信息学院**

**专 业： 计算机科学与技术**

**年 级： 2013级**

**学 号：2013202585**

**指导教师：**

**论文成绩：**

**日 期：**

**摘要**

Greenplum数据库（GPDB）就是一款典型的可以存取大规模关系数据并且支持OLAP功能的大规模并行处理（MPP）数据库，具有线性可拓展性和很不错的处理性能。很多现有的并行数据库系统是通过重启查询来处理结点故障的。虽然这对于在少量服务器上运行的短查询不是问题，但是对于那些在大集群上运行的长查询来说成为大问题。对于那些需要保证在规定时间内完成的查询，若是在查询中间宕机，造成的经济损失可能会非常大。所以，为了提高Greenplum数据库可靠性，本文针对时间敏感的查询，提出了提高查询可靠性的方案，在保证一致性的前提下，增加查询成功执行的概率；另外，提出选取物化节点的策略，保证查询在按时完成的基础上，减少容错增加的开销；并且在最后设计实验验证了方案的效果和性能。

关键词：大规模并行处理数据库，容错，一致性，物化策略

**Abstract**

Greenplum database (GPDB) is a typical large-scale parallel processing (MPP) database that can access large-scale relational data and supports OLAP functionality, with linear scalability and very good processing performance. Many existing parallel database systems handle node failures by restarting the query. Although this is not a problem for short queries running on a small number of servers, it is a big problem for long queries that run on large clusters. For those who need to ensure that the required time to complete the query, if the query in the middle of the downtime, resulting in economic losses may be very large. Therefore, in order to improve the reliability of Greenplum database, this paper proposes a scheme to improve the query reliability and improve the probability of query execution under the premise of ensuring consistency for time-sensitive queries. In addition, the strategy of selecting the materialized node is proposed, On the basis of on time, to reduce the cost of fault tolerance increase; and in the final design experiments to verify the effectiveness and performance of the program.

Key Words: Massively Parallel Processing Database, Fault Tolerance, Consistency, Materialization strategy

目录

[绪论 7](#_Toc482718261)

[1.1背景介绍 7](#_Toc482718262)

[1.1.1规模并行数据库的涌现 7](#_Toc482718263)

[1.1.2Greenplum数据库简要介绍 7](#_Toc482718264)

[1.2相关工作 8](#_Toc482718265)

[1.2.1大规模并行数据库研究方向概述 8](#_Toc482718266)

[1.2.2查询容错方面相关工作详述 9](#_Toc482718267)

[1.3问题描述 10](#_Toc482718268)

[2 Greenplum 数据库查询处理机制 12](#_Toc482718269)

[2.1 基本架构 12](#_Toc482718270)

[2.2查询执行机制 13](#_Toc482718271)

[2.3容错机制 14](#_Toc482718272)

[3 整体方案概述 15](#_Toc482718273)

[3.1目标 15](#_Toc482718274)

[3.2 中间结果保存机制 15](#_Toc482718275)

[3.2.1 算子的中间结果保存 15](#_Toc482718276)

[3.2.2 中间结果的同步 15](#_Toc482718277)

[3.3 查询重启 16](#_Toc482718278)

[3.4 查询续作 16](#_Toc482718279)

[4 查询控制流程的改进与实现 17](#_Toc482718280)

[4.1 GPDB故障检测服务（FTS） 17](#_Toc482718281)

[4.2GPDB查询流程分析 19](#_Toc482718282)

[4.2.1查询派发流程 20](#_Toc482718283)

[4.2.2查询重启流程实现 25](#_Toc482718284)

[5查询一致性保证与实现 28](#_Toc482718285)

[5.1问题描述 28](#_Toc482718286)

[5.2 Greenplum现有保证事务一致性的方案 29](#_Toc482718287)

[5.2.1 多版本并发控制 29](#_Toc482718288)

[5.2.2 slave上本地XID与分布式事务ID的映射关系 31](#_Toc482718289)

[5.3 设计方案 32](#_Toc482718290)

[5.4 方案实现 33](#_Toc482718291)

[5.4.1 保存快照 33](#_Toc482718292)

[5.4.2 保存事务ID 34](#_Toc482718293)

[5.4.3 保存分布式事务上下文 34](#_Toc482718294)

[6保存点选择模型设计与实现 35](#_Toc482718295)

[6.1 代价估计 35](#_Toc482718296)

[6.1.1 路径的代价估算的基本步骤 36](#_Toc482718297)

[6.1.2 哈希（Hash） 37](#_Toc482718298)

[6.1.3 排序（Sort） 38](#_Toc482718299)

[6.1.4聚集（Aggregate） 39](#_Toc482718300)

[6.1.5物化代价的计算 40](#_Toc482718301)

[6.2路径代价的传递和打印 40](#_Toc482718302)

[6.3 物化节点选取算法 42](#_Toc482718303)

[7实验测试与结果分析 47](#_Toc482718304)

[7.1 实验环境 47](#_Toc482718305)

[7.2实验方案及结果 47](#_Toc482718306)

[7.2.1 TPC-H 测试集 47](#_Toc482718307)

[7.2.2 查询重启性能测试 48](#_Toc482718308)

[7.2.3 查询一致性验证 48](#_Toc482718309)

[7.2.4 保存点选取模型测试 51](#_Toc482718310)

[7.3结论 53](#_Toc482718311)

[8 总结 53](#_Toc482718312)

[参考文献 54](#_Toc482718313)

**图片目录**

图1... .. 12

图2 18

图3 18

图4 19

图5 20

图6 23

图7 24

图8 26

图9 28

图10 30

图11 32

图12 33

图13 37

图14 40

图15 41

图16 42

图17 43

图18 44

图19 45

图20 45

# 绪论

## 1.1背景介绍

### 1.1.1规模并行数据库的涌现

20世纪80年代和90年代初期是大规模并行处理技术急速发展的时期。这种技术的原始驱动力是科学和工程应用，像是天机预报，分子模拟，油气勘探和气候研究。大约在同一时间，几家企业开始看到分析日益增长的交易数据量的潜在价值。这样的分析催生出了决策支持系统的出现，支持在大量数据上的复杂查询。这种趋势最先推动了对并行数据库系统的需求。

在过去的十年间，越来越多的数据源的涌现催生出大数据时代。网页点击，社交网络，科学实验和数据中心的监测作为这些源头的一部分，每天生产着大量数据。根据IDC（International Data Corporation）提供的数据显示，全球数据在以每年40%的速率飞速增长，到2020年，人类每年创建和复制的数据将达到44ZB。这些数据背后隐藏的价值不可估量，企业可以通过这些数据可以更好的理解他们的顾客，发现问题，提升服务，减少风险。可以预见的未来，企业的竞争力往往取决于对数据及时有效的分析和更深入的洞察力。也就是说，数据处理分析系统在很大程度上决定了企业能否在竞争中占据优势。然而，传统的数据管理和分析方法显然已经远远不适合在现今如此大规模的数据量上扩展了。传统数据库系统需要大量的资源支出和技术干预，很难适应当今对快速智能化、深度分析的趋势。从性能的角度来看，摩尔定律的发展意味着可以在CPU堆叠更多的核，数据的增长比摩尔定律预期的要快。所以企业需要通过增加新的结点来增加系统的容量和性能。这种需求导致了在线大规模数据分析系统（OLAP）的大量创新。

### 1.1.2Greenplum数据库简要介绍

Greenplum就是一款典型的可以存取大规模关系数据并且支持OLAP功能的大规模并行处理（MPP）数据库。该数据库是Pivotal公司开发了近十年的产品，在80多万行Postgresql数据库内核上添加了近50万行代码，工程量巨大。在其巅峰时期，Greenplum数据库的年收益达到了一亿美元。Greenplum数据库于2015年10月开源，作为一款典型的MPP数据库，具有很好的研究价值。

GPDB采用的是shared-nothing架构。在并行数据库领域，有三种维度的并行：共享内存，共享磁盘，无共享（shared-nothing）。在共享内存架构，所有处理器共享访问中央存储器和所有磁盘。这个架构可扩展性是有限的，因为访问内存很快成为瓶颈。在共享磁盘架构中，每个处理器都有其专用内存，但所有处理器共享对所有磁盘的访问。但是这种架构扩展非常昂贵，因为处理器到磁盘的连接十分复杂。近些年，无共享架构被证明是最可行的，结点之间无共享，可以发挥每个结点的最大作用，具有线性可拓展性和很不错的处理性能。在容错方面，GPDB支持主节点和数据结点镜像备份；并且给每个主机配置多个网口并使用多个交换机来避免网络故障。同时，GPDB还有很多优越的特性：数据并行加载，加载速率高达16TB每小时，并且支持包括Hadoop在内的各种数据源和数据格式；支持行存储和列存储；支持多种压缩方法和多即分区表；此外，GPDB还支持多种编程语言来实现自定义函数，如C++，Perl，Python，Java等，还提供了ODBC，JDBC等编程接口。这些特性使得GPDB成为数据库市场上很受欢迎的产品。

## 1.2相关工作

针对Greenplum数据库的研究仅仅限于企业实际应用中，但是对于MPP数据库的研究却数不胜数。本部分先对

### 1.2.1大规模并行数据库研究方向概述

作为数据库的一部分，MPP数据库本身有很多维度可以拿来研究。从功能特征方面可以分为以下几个范畴，其中每个范畴都有大量的研究成果和研究空间。

**数据模型和接口。**数据模型指的是数据在数据库中存储、组织、操纵的逻辑结构。其中最典型的要数关系模型。当然，也有很多其他组织形式的数据，例如，MapReduce类型的数据库可以存储任何类型的数据。

**存储层。**存储层是来持久化数据的，并且提供访问和修改数据的方法。不同的系统对存储层的设计、实现和功能特征差异巨大。

**查询执行引擎**。当数据库收到查询时会将其转化为执行计划来访问和处理查询的输入数据。查询引擎就是负责运行一个给定的查询计划，产生查询结果；也负责并行计算、处理机器错误和设置不同结点间的通讯网路和带宽等。

**查询优化。**查询优化的目的是帮助系统在若干个不同的候选计划中找出一个执行查询最有效的计划。优化考虑的因素有很多，包括系统可用空间、代价、搜索查询计划采用的方式等。

**调度。**鉴于大多数数据分析系统的分布式性质，调度查询执行计划是系统的关键部分。系统现在必须做出几个调度决策，包括调度运行每个计算的位置，调度节点间数据传输，以及调度滚动更新和维护任务。

**资源管理**。为了高效地使用集群资源，资源管理可以根据查询地资源请求和当前系统地资源状况合理有效地安排资源。现在有很多系统支持用户动态地添加移除资源。

**查询容错。**机器故障在大集群中是十分常见的问题。为了增加系统可靠性，容错成为并行数据库系统重大挑战之一。大部分数据库系统都会增加容错功能来解决硬件故障，软件错误和数据损坏的问题。

### 1.2.2查询容错方面相关工作详述

当大规模运行并行查询时，执行过程中不可避免的会发生某种形式的故障。不同的数据库产品会根据系统特点采取不同的系统容错方案。一般来讲，大型数据库会建立在高可用性的平台上，比如用**独立磁盘冗余阵列来减少单个磁盘故障**对整个系统可用性的影响。像TeraData和Exadata就使用RAID来实现系统级别的容错。还有数据库系统采用镜像结点（Mirror）备份的方式，如Greenplum给每个主备结点都设置了一至两个镜像结点，保存有数据备份。当某一个结点出现故障时，镜像结点就会代替故障结点执行功能。

通常，数据库中的容错是通过完全重新启动查询来处理的，但是在更大的集群中，在查询内部容错也十分重要。Hadoop，Impala或Shark使用细粒度容错方案——将中间结果物化使系统可以从中间恢复查询。许多这样的系统将查询编译进基于MapReduce的执行计划，**这样就可以将每个map和reduce函数的输出结果都物化下来了**。尽管细粒度地容错方案可以有效地处理在复杂地长查询中间产生的故障，**但是却不适合对延迟敏感的短查询**。**因为物化中间结果的开销往往会高于本身执行查询的开销**。在学术方面，只有少数论文对MPP数据库查询内部容错方案进行研究。FTOps提出在不阻塞的情况下进行查询容错的方法，。该系统让流水线上的每个算子都记录并备份自己的状态，一旦出现故障，每个算子会恢复到自己备份的状态，并通知上游算子自己的流水进度。多个算子通过协调就可以恢复流水线运作。由于备份算子中间状态会带来一定的额外开销，FTOps系统提出了一个优化器，对开销和收益进行权衡，来制定保存算子的方案，并且找到从中间结果恢复一个算子的最佳方案。于FTOps不同的是，Opsprey和XDB选择对查询吃力的流水线进行阻断，即将查询分解为分区上的子查询，并且最终将所有分区上的查询合并。在每个子查询结束后，系统会对其中间结果进行备份。如果子查询失败，则会在其备份的副本上重启。XDB在Opsprey的基础上提出了一个基于成本的优化模型，在查询执行效率和容错能力上进行折中，来找到划分子查询的最佳方案。

## 1.3问题描述

很多现有的并行数据库系统是通过重启查询来处理结点故障的。如果在执行过程中出现错误，并行数据库管理系统会抛弃原有查询，重新启动查询；这种方法的局限性在于很小的故障也能导致系统重新处理整个查询。虽然这对于在少量服务器上运行的短查询不是问题，但对于使用大量服务器的对完成时间敏感的长时间查询来说成为大问题。为了更好的理解这个问题，我们通过一个实际的例子来看。假设一个shared-nothing的分布式数据库，要求每天要生成一系列六个报告。假设每个报告需要3个小时才能在1具有100个结点、800个磁盘的数据库上生成。假设单个结点上的操作系统每30天崩溃一次，即每8×30个报告崩溃一次（因为每个报告花费1/3天）；一个磁盘每两年失效一次，即每8×365×2份报告失效一次。那么，在查询运行期间没有结点故障的概率：，这意味着，在执行查询期间，会有35%的概率出现故障，平均每个故障会浪费1.5小时的工作时间。同理，在查询运行期间，磁盘出现故障的概率为：。尽管假设与实际系统的CPU和磁盘的数量可能有出入，失败率也不尽相同，但是随着数据库集群的增大，查询中间容错会变得越来越重要。

像上述需要在规定时间内完成的查询，我们称之为对时间敏感的查询。我们的目标是提高系统在规定时间内完成查询的概率，即提高数据库系统的可靠性。为了提高可靠性，一方面要实现系统容错的功能，增加查询成功执行的可能性；另一方面要减少容错增加的开销，保证在规定时间内完成。本文旨在Greenplum数据库上设计并实现一个查询内部的容错方案，增加系统对时间敏感查询的可靠性。系统实现需要解决的问题：

1) 主结点（master）如何检测到结点故障？如何通知从节点（segment）重新开始查询？

2) 在查询执行和重启的过程中，如果有数据更新，如何保证查询重启前后结果的一致性？

3) Segment如何实现中间结果的保存？

4) 在查询开始前，如何预估各个阻塞结点的代价，选取保存点？

# 2 Greenplum 数据库查询处理机制

## 2.1 基本架构

Greenplum 数据库（GPDB）基本上是几个PostgreSQL数据库实例共组成的一个连贯的数据库管理系统。从SQL支持、功能、配置选项和终端用户功能等方面来看，在大多数情况下与PostgreSQL没什么两样。为了支持并行结构，GPDB修改或增强了PostgreSQL的内部组件，如：系统目录，查询计划器，查询优化器，查询执行器等。GPDB通过**网络层**将不同PostgreSQL实例连接起来。GPDB拥有一个主控节点**（Master）**管理整个集群，负责整个集群PostgreSQL数据库（Segment）实例的运行。每个Segment分布在不同的物理主机上，协同工作。如图所示：

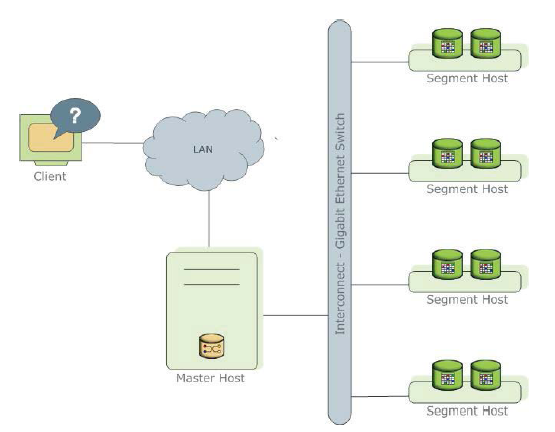


图1 GPDB 分布示意图

主节点（master）上并不存储任何用户数据，只存储全局系统表（Global System Catalog），负责客户段连接、解析SQL语句、生成并派发查询计划、整合Segment处理结果并将最终结果呈现在客户端。子节点（segment）负责存取数据，执行主节点分发的查询计划。数据通过哈希算法或者随机拆分成互不重叠的集合分布在各个segment上。用户表以及相应的索引（Index）也都分布在各个segment上。用户只能通过master来访问segment上的数据。网络层组件是GPDB的重要部分之一。查询计划下发、查询结果回传、数据重分布、数据库备份等都是网络来实现的。网络层支持UDP和TCP两种协议，在默认情况下使用UDP协议。GPDB实现了为UDP协议做数据包检验的功能，其可靠性与TCP协议一致，但其性能和扩展性好很多。

## 2.2查询执行机制

系统启动后，用户可以在master主机的终端上输入psql命令输入并提交查询语句。GPDB会创建多个进程来处理查询。GPDB将进程分为查询分发器QD（Query Dispatcher）和查询执行器（QueryExecutor）。QD进程在master上，负责查询计划的查询和分发，结果的汇总；QE负责执行分发下的查询。

当master以QD身份接收到客户端输入的查询字符串（input message）,会将其进行词法分析、语法分析生成分析树，然后进行语义分析和查询改写得到查询树。最终，QD最终调用查询计划器在（planner）接收改写后的查询树，最终得到可以被执行器执行的最优计划。在查询计划阶段，QD会移居该查询树完成逻辑优化和物理优化，主要是提升子链接和提升子查询，并且还要使用一定的算法生成最优的连接路径。在寻找最优路径时，查询计划器会从全局的角度进行代价估算，综合考虑CPU计算开销、磁盘IO开销以及结点之间移动数据的开销。在生成最优的查询计划后，QD会将其分发到相应的segment执行，每个segment只会负责处理自己主机上的数据。每个segment都会收到一份查询计划，创建工作进程QE执行任务。

GPDB将查询计划分割成了一个个**切片（slice）**，每个切片负责整个查询工作的一部分。在两个切片之间会存在不同segment之间的数据移动操作。我们可以这样理解，切片是segment能不需要其他segment的干预、单独执行任务的最小单位。所以，查询计划是一个一个切片地执行，每做完一个切片，segment之间会通过**网络层组件（Interconnect）**进行数据移动。每个segment上的QE进程会给每个切片分配一个postgres进程，这样可以实现很高程度的并行化，该进程内会执行多个操作。每一个slice代表着特定的数据库操作，如表扫描、连接、聚合、排序等。单个segment上的slice中的算子从上向下调用，数据从下向上传输。不同segment上同一个切片完成同样的任务，被称为一个**簇（gang）**。在当前gang上的工作完成后，数据将向上传递，直到查询计划完成。

之前提到，在slice之间会有数据的移动。在GPDB中，数据移动有其对应的算子：**移动（motion）**。motion分为**广播（broadcast）**和**重分布（redistribute）。**在某一层slice做完后，上一层slice会读取其广播或者重分布的数据。GPDB使用元组（**Tuple**）存储左右的信息，包括系统信息、数据等，并且使用流水的方式处理元组。motion操作也是以元组为单位收发数据，最终结果也会以元组的形式通过motion传回mater。

## 2.3容错机制

GPDB自身提供备份容错机制。对于主节点，提供mater/standby机制，当主节点宕机后，standby会代替master进行工作。对于每个segment（称为Primary），GPDB都配置了镜像节点（Mirror）。Primary与相对应的Mirror配置在不同的物理机上，每台物理机可以存不同一个Primary和多个其他主机的镜像节点。master会通过心跳监测来获知segment的状态，当发现某一个镜像节点出现问题时，会自动切换到对应的镜像节点上。在数据库启动后，会开启单独的FTS进程对Segment状态进行心跳监测。每当 Master 发现无法连接到某 Primary Segment 时，将在GPDB的系统日志表中标记为故障状态，并用相应的镜像节点以替换原始的节点以继续后续工作。

# 3 整体方案概述

## 3.1目标

首先，数据库因某个结点故障而中断查询后，主节点（master）应该及时捕捉故障信息，并且重新启动查询，并且通知从节点（segment）进行第二次查询。所有子节点（segment）应该收到主节点发出的信号，读取保存的中间结果，从查询中断处重启查询。其次，在首次执行查询时，需要选择查询流水线上的阻塞结点保存中间结果。在Greenplum数据库中，阻塞结点一般是哈希、聚集和排序。在这里，我们需要知道如何衡量保存中间结果的收益与代价，制定合理的保存策略，以较小的花销，较高的概率在规定时间内完成任务。最后，保证查询正确性。在查询重启期间，数据库中的元组有很大几率会被更新。本文提出的方案需要对两次查询结果的一致性进行保证。

## 3.2 中间结果保存机制

### 3.2.1 算子的中间结果保存

下面以 Hash Join 为例，简述算子的中间结果保存。在 Hash Join 过程中，只要 Hash 表建表过程没有完成，连接操作就会一直被阻塞，同时因为 Hash 无法完全保存在内存中，算子本身就不得不对大部分数据分 batch 进行物化落盘。我们修改这部分逻辑，保存下完整的 Hash 表，并调用修改过的同步接口，异步将物化数据备份至 Mirror Segment。因为在连接过程中，一般用小表建表，所以对比保存完整连接结果，我们保存 HASH 表不仅不多阻断查询，I/O 和同步操作上也更节省资源。

### 3.2.2 中间结果的同步

GPDB 有一套完整的机制用于在主备机之间传递消息及文件。原有机制采用了生产消费者模型，每一个 Primary 与 Mirror 节点都有 2-4 块共享内存用于在 Primary 节点与对应的Mirror节点之间发送消息与文件。Primary 所有需要发送消息的进程都会把消息内容写到该节点的共享内存中，同时还要添加消息头、校验码等，Primary 节点有一个发送者进程负责将共享内存中的消息发送给Mirror节点，Mirror节点的接收者进程负责接收消息，将消息写到对应的共享内存中，之后会有不同的消费者进程读取消息，并做相应的处理。我们在同步中间结果时，在上述机制基础上，增加一种新的文件传输操作以区别原有的消息。Primary节点产生中间结果的进程，调用函数接口，将需要同步的文件以新的文件操作类型的消息发送到Mirror节点。

## 3.3 查询重启

在此阶段，GPDB需要做到能保存查询计划，及时检测到故障，并在主备机切换完毕之后，重启查询。详细的内容将在下一部分介绍。

## 3.4 查询续作

Master 重启查询后，以 QD 身份拿到在内存中保存的查询计划等信息，然后向 Segment 发送指令，要求上传物化进度。QD 结合查询树和各节点的物化进度信息，由上至下标注 slice 执行信息，判断哪些 slice 可以跳过不做，哪些 slice可以直接读取本地物化中间结果，哪些 slice 需要完整执行。因此重启后 slice 执行类型分为“跳过”，“读取”和“执行”三种，“执行”为默认类型。例如某需要重做的查询在 slice7 执行 Hash Join，且根据物化进度表得知所有 Segment 都在这里进行了物化。那么 QD 将会在递归分析查询树的过程中，找出 slice7 的子孙中哪些是提供小表数据的 slice，将之标记为“跳过”，因为 slice7 重做时可以本地读出 Hash 表，对应提供建表数据的的子孙 slice 已经没必要再做了。同时将 slice7 本身标记为"读取"，即读取本地数据后执行。而提供大表数据的子 slice 还需要继续提供元组流水完成 Hash Join，因此将保持原执行状态不变。

QE 上各 slice 对应的进程启动后会首先读取 QD 派发下来的 slice 执行信息，依据新的执行类型选择跳过本 slice 的全部操作，读取本地已有物化数据的中间结果继续执行相关操作，和正常执行所有算子操作。

# 4 查询控制流程的改进与实现

## 4.1 GPDB故障检测服务（FTS）

在数据库开启后，master会开启FTS进程来按照一定的时间间隔检测每个segment的状态，以便及时发现系统故障。FTS的原理非常简单，通过建立与各个segment之间的网络连接（socket），发送检测使用的相关信息，接收segment的反馈信息来确定segment是否是活跃的。一旦出现宕机的情况，建立的网络连接（socket）就关闭，此时，master的检测进程就无法收到反馈消息或者等待超时。当出现连接不上的情况是，检测进程会立即进入转入change tracking状态，将这个segment的状态由UP转换为DOWN。

在FTS机制中，probeSegment函数负责完成对某一个segment的检测。该函数的输入是某个segment的CdbComponentDatabaseInfo指针dbInfo。CdbComponentDatabaseInfo结构体如下所示，我们省略了与FTS检测无关的属性。dbid相当于是segment的标识，role表示该sengment是primary还是mirror，preferred\_role表示用户希望这条机器是primary还是mirror，mode表示的是该机器是否处于切换状态，status表示机器是否活跃，hostname和address分别表示主机名和IP地址。

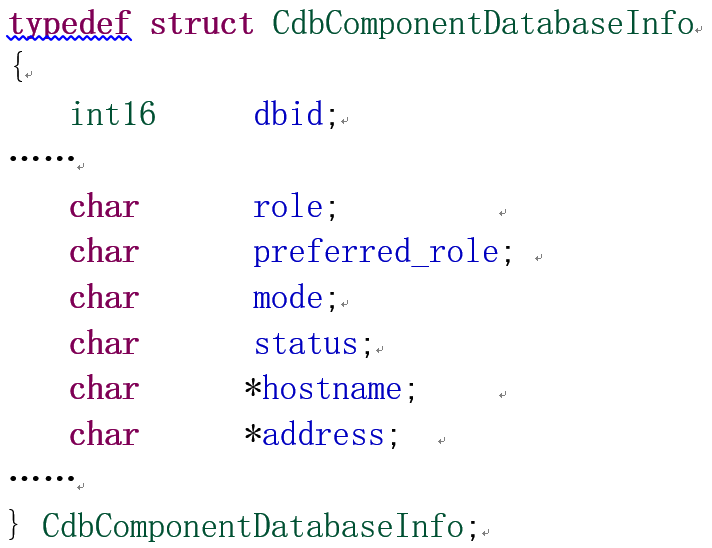


图2 CdbComponentDatabaseInfo结构体

在probeSegment函数内部会声明一个ProbeConnectionInfo指针类型的变量probeInfo用来记录扫描得到的segment的状态。如下面结构体所示，可以看到ProbeConnectionInfo与CdbComponentDatabaseInfo的属性是有一些重复的，如dbid，role，mode，address，segmentStatus等，这是因为在检测中不会直接修改segment的信息，而是返回segment的检测状态。此外该结构体还存有一些网络连接信息，如套接字（socket）信息等。

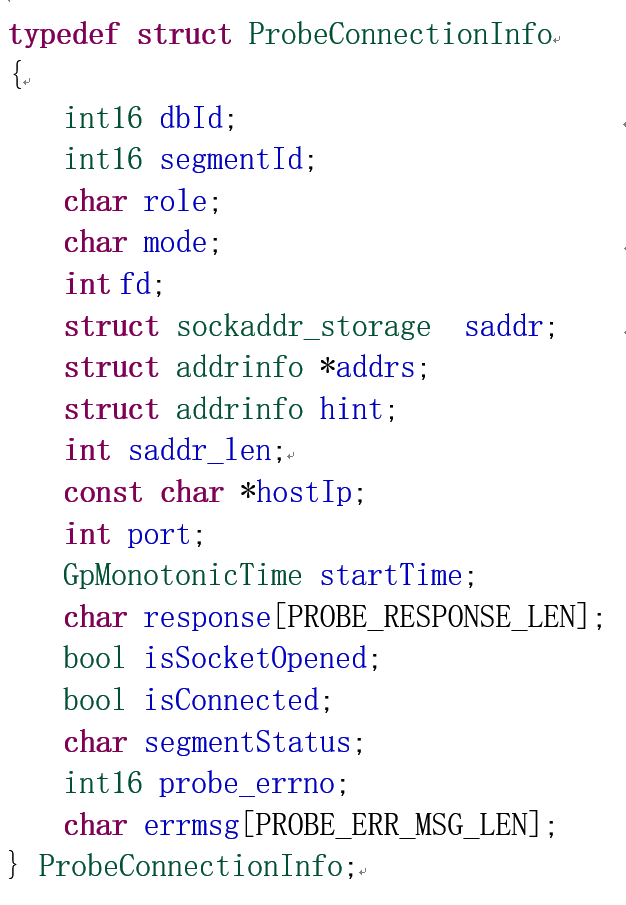


图3 ProbeConnectionInfo结构体

probeSegment函数会根据在dbInfo中储存的segment的IP地址，生成对应的套接字、建立连接，发送信息并且等待回复。这写工作是被循环调用的，在循环中会有睡眠时间来等待该segment回复。如果等到回复，将probeInfo设置成PROBE\_ALIVE，并且跳出循环。如果循环次数超过了用户设置的重试次数retryCount,probeInfo就会被设置成PROBE\_DEAD。probeSegmentFromThread函数通过调用probeSegment函数，实现了对所有primary和mirror的检测，但是只有当primary状态是PROBE\_DEAD时，才会启动对Mirror的检测。为了尽快完成对所有segment的检测，FTS实现了多线程调用probeSegmentFromThread，默认的线程数是16。

现在，我们通过检测函数获取了检测整个segment的结果，那么如何得出哪个segment出现故障了呢？FTS会通过调用probePublishUpdate函数来对比检测结果和之前结果，然后根据primary和mirror的状态来执行不同策略。如果检测到主节点宕机情况，probePublishUpdate会返回真值，并且调用updateConfiguration函数进行状态切换，并且通知用户数据库正在进行状态切换（change tracking）。FTS用下面的枚举类型来标记primary和Mirror的状态组合。当出现TRANS\_D\_D状态时，FTS什么也做不了，只能通知用户出现故障的segment号码。

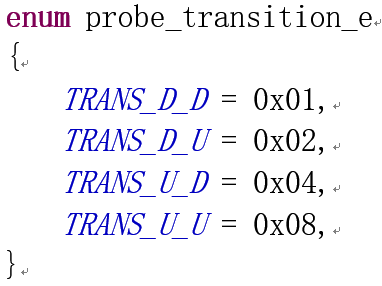


图4 probe\_transition\_e状态

*TRANS\_D\_D*代表主备机全部宕机；*TRANS\_D\_U*代表primary宕机、备用机正常；*TRANS\_U\_D*代表primary正常，备用机宕机；*TRANS\_U\_U*代表主备机均正常

下面的流程改进部分会充分利用FTS的原理，来实现对故障的快速检测和恢复。

## 4.2GPDB查询流程分析

当用户开启一个会话（session），准备输入查询语句的同时，GPDB会启动PostgresMain()函数。这个函数是所有后端（backend）、交互或是其他的进程的开始。其输入参数是命令行，用户名。在此函数的开始会做一些准备工作，包括：初始化变量，分配查询语句（statement）、配置文件的句柄，记录时间戳，启动查询执行器（Query Executor）、分配内存上下文等。在GPDB中有专门记录内存上下文的结构体MemoryContextData，并且可以使用AllocSetContextCreate()函数来分配内存上下文。接下来，PostgresMain()函数会有处理最外层错误的if语句，来关闭中断，重置变量或者放弃当前事务等。紧接着，就开始进入查询的循环。在这个循环中，会根据调用进程的不同角色来走不同的分支路线。我们关心的主要是两个角色，一是查询派发器（Query Dispatcher），另一个是查询执行器（QueryExecutor）。

### 4.2.1查询派发流程

查询派发器（Query Dispatcher）是运行在master上的进程，负责与客户端交互，查询计划的生成和派发。在PostgresMain()中，会根据传入的第一个字符来判断应该执行那一部分代码。当第一个字符是Q的情况下，会执行查询派发器的功能。在进入上面提到的循环前，需要做一些只有派发器才会做的准备工作。

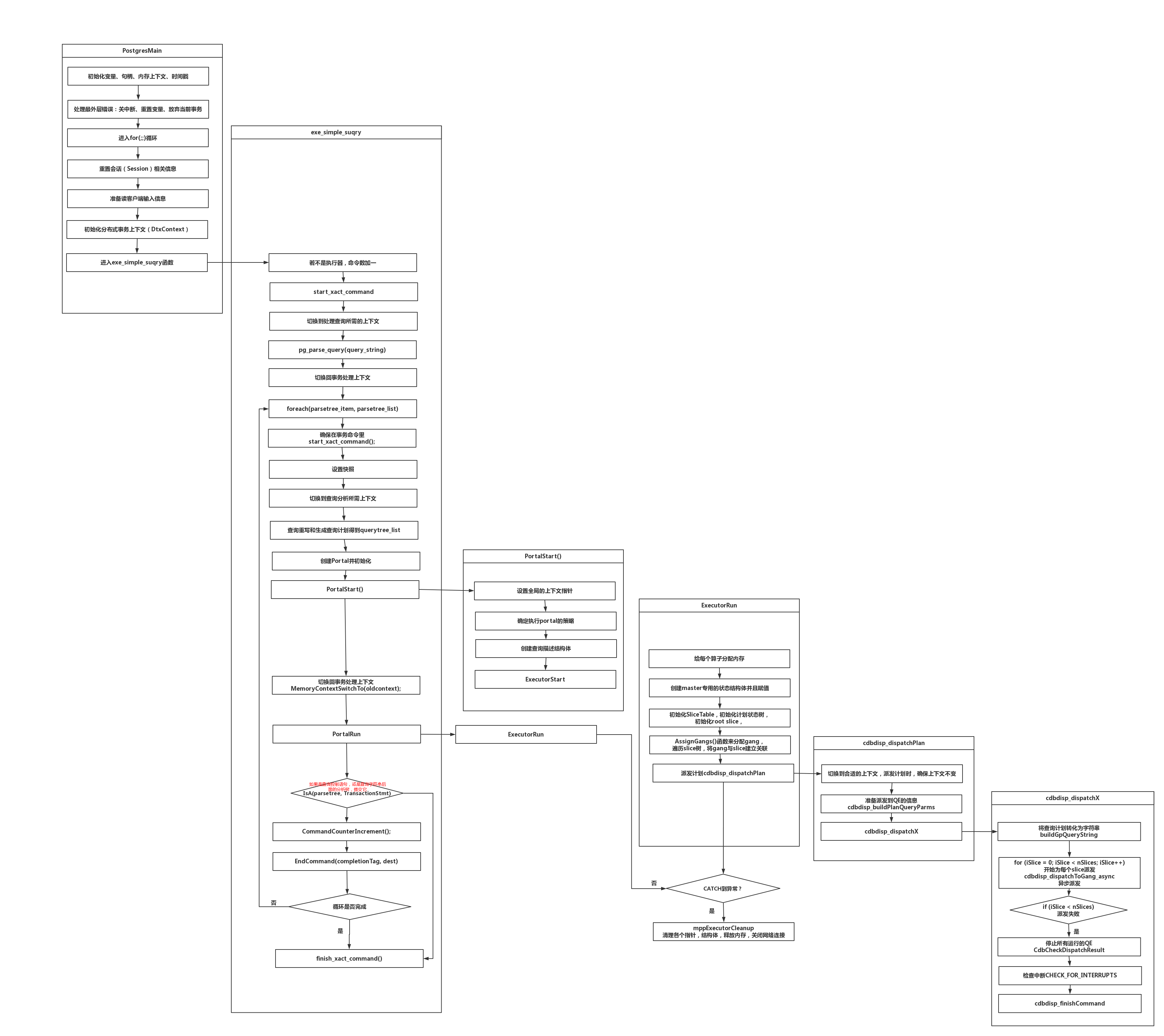


图5 PostgresMain函数执行图

第一步，初始化相关变量。首先，分配一个StringInfoData结构体类型的输入缓冲区来存储变长的字符串。在此结构体中有一个游标来标识读取的位置。接着，程序需要生成全局ID标记这个会话。调用CheckForResetSession()函数来重置session的相关信息。在确保所有的Gang已经执行完毕的情况下，生成新的uint32类型的会话ID，赋给数据库全局变量gp\_session\_id；并且，将记录命令数目的全局变量gp\_command\_count置零；接着改变快照对应的会话ID（一个会话会有一个对应的快照）。

第二步，等待用户输入。程序调用ReadCommand()函数，等待读入用户在客户端输入的信息。并且在此处判断查询是否完成，通过判断后台是否真正的空闲（idle）来判定。如果是空闲的，说明查询已经执行完毕，可以释放相关内存。

第三步，进入for循环，调用exec\_simple\_query()函数开始对查询的处理。派发器会在进入循环后会设置 dtx(distributed transaction context)，这是只与分布式事务相关的上下文。

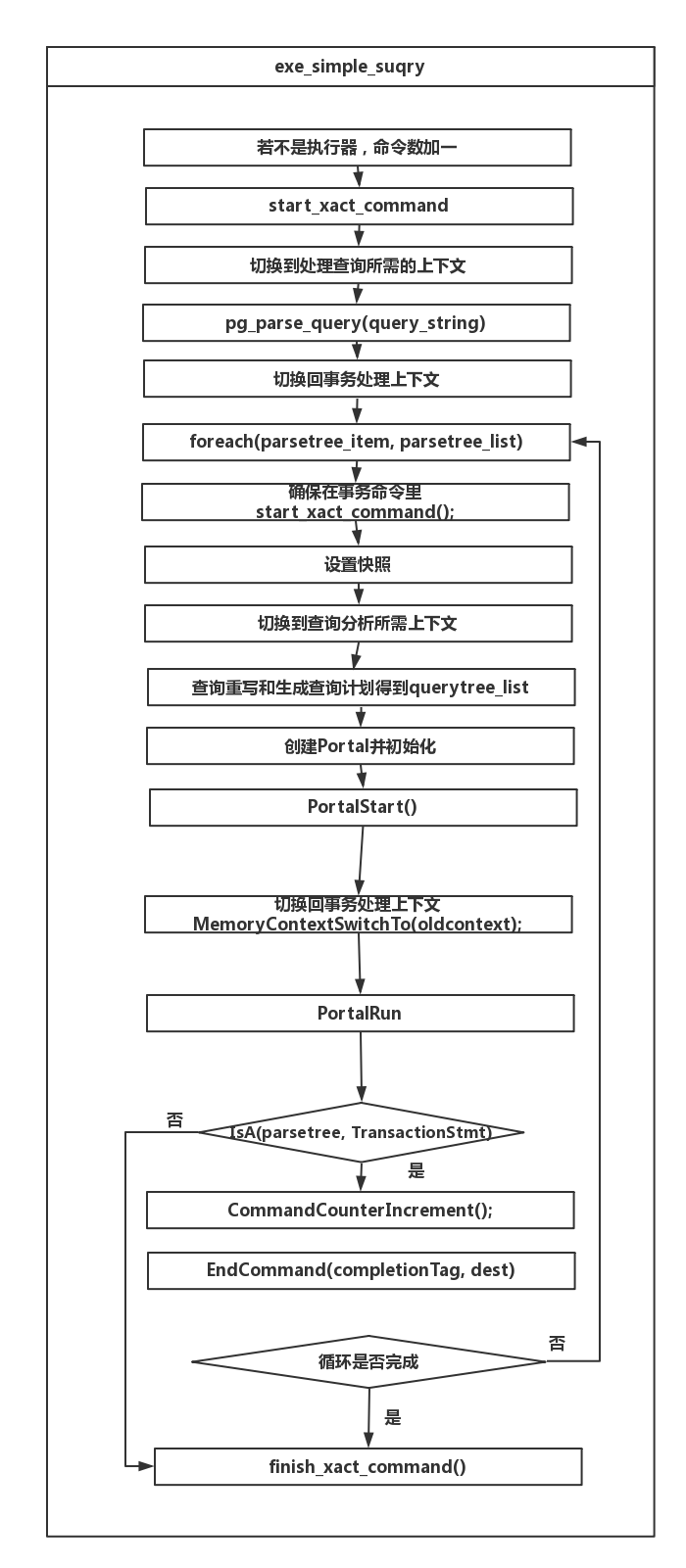


图6 exe\_simple\_query函数执行图

exec\_simple\_query()函数十分重要，这个函数通过调用其他相关函数依次完成语法、词法分析，查询重写和查询计划。在函数的开始，会初始化一些变量，包括查询结果输出的目的地dest,解析树列表parsetree\_list，计划树列表plantree\_list等。然后调用start\_xact\_command函数，根据不同命令的类型设置一些变量。接着切换到与解析相关的上下文，进行语法词法解析，得到parsetree\_list。在解析结束后，要切回事务处理的上下文。切换上下文的函数是MemoryContextSwitchTo()，这个函数会将当前上下文切换到输入的上下文中，并且返回之前处于的上下文。在上下文（Context）中存有函数执行相关的信息，例如事务处理上下文中保存了事务处理所需要的全局变量。MemoryContextSwitchTo()是个非常重要的上下文切换函数，在下文会反复用到。

解析树列表parsetree\_list是由表示解析树的结构体组成的列表，可以存储多个不同查询的解析树。在现实情况下，用户可能一次输入不止一个查询语句，所以，需要将查询解析树设计成列表的形式，其中的每一项对应一个查询语句的解析树。

接下来，遍历解析树列表，对解析树列表中的每一项逐一处理。如果只有一个查询，只需要执行一次循环即可。在循环中，程序会执行查询重写和查询计划。重写和计划需要单独的上下文来存储相关信息，因此，在这里需要分配新的上下文，并且进行内存上下文切换。先后调用pg\_analyze\_and\_rewrite和pg\_plan\_queries函数对查询进行重写和计划。这里我们不关注查询计划如何得到的，只需要梳理查询派发的流程。至此，查询计划树planTreeList已经生成。但是，现在还不能切换回事务上下文。程序还需要执行这个查询计划。

数据库会创建一个门户（portal）来保存运行或可运行查询的执行状态。首先程序会调用PortalDefineQuery和PortalStart函数初始化一个用户查询执行的门户（Portal），这两个函数会将查询计划plantree\_list传入Portal结构体存储。PortalStart()函数会初始化描述查询的结构体QueryDesc，设置portal上下文，根据查询类型确定执行策略，设置并且拷贝快照。在此过程中，通过调用函数**voidExecutorStart**(QueryDesc \*queryDesc, **int** eflags)来实现查询计划的下发。在次函数中，dispatcher会初始化SliceTable，初始化计划状态树，初始化root slice，并且调用**voidAssignGangs**(QueryDesc \*queryDesc)函数来分配gang，遍历slice树，将gang与slice建立关联。最后，调用**voidcdbdisp\_dispatchPlan**(**struct**QueryDesc \*queryDesc,bool planRequiresTxn,bool cancelOnError, **struct**CdbDispatcherState \*ds)来派发查询计划。

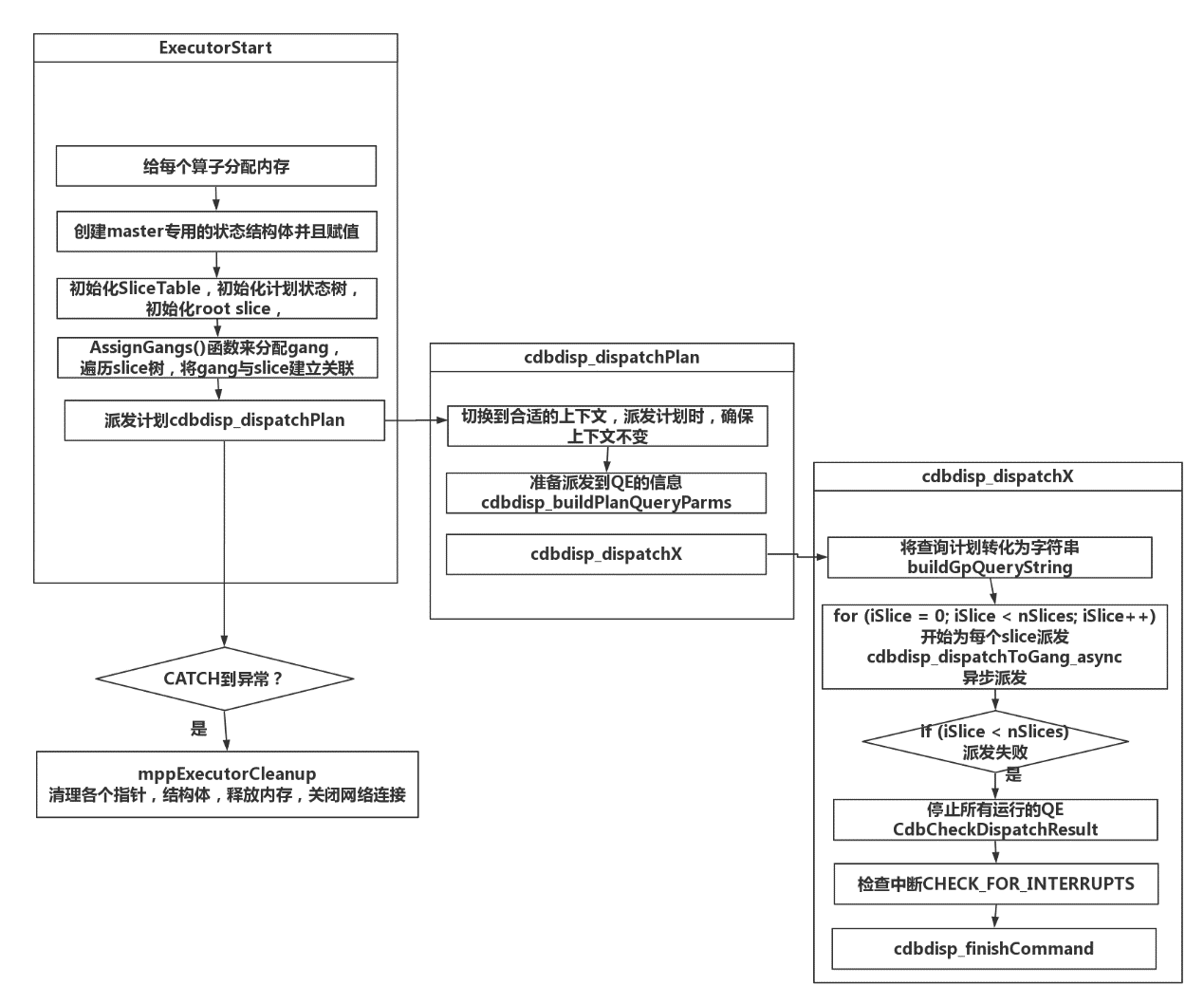


图7 查询计划下发相关函数图

调用PortalRun()函数执行这个计划树

PortalRun()根据查询的类型采取不同的策略来执行。现阶段，GPDB将查询分成四种：单个SELECT查询;INSERT/UPDATE/DELETE这样会返回一个从句的操作（或者是重写生成的辅助语句）；类似于SELECT的utility语句，例如EXPLAIN,SHOW；其他类型的语句。以单个SELECT查询为例：PortalRunSelect()会根据一定的扫描方向（Scan direction）顺序读取n个元组，并且调用excutorRun()函数，按照查询计划对元组进行处理，并且返回处理的元组个数。excutorRun()的输入是由PortalRunSelect()函数确定扫描方向和传入的元组数。此函数获取当前执行的slice,并且根据slice类型来确定这个executor是服务于谁。例如，当调用者是dispatcher时，但是slice是属于executor执行，dispatcher应该忽略情况。又如，executor服务于root slice，由于接近于根节点的一般是motion操作，所以executor进行发送结果，并不会调用下面的节点。在excutorRun()中，使用了try/catch方法来检测程序运行时产生的异常。一旦发现错误，会调用mppExecutorCleanup()函数来关闭网络连接，清空内存等操作。

在查询计划执行完成后，会调用PortalDrop()释放门户。最后执行finish\_xact\_command()函数，处理命令执行完毕的一些事宜。执行的命令数量gp\_command\_count会随循环次数增加一。若是用户只输入了一个查询，那么for循环结束，从exec\_simple\_query()返回，PostgresMain()函数会继续等待查询输入。

### 4.2.2查询重启流程实现

#### 4.2.2.1方案设计

收到客户端的查询请求，生成查询计划，并将查询计划保存在上下文中，开始查询执行。如果查询过程中发生错误，标记本次查询出错，并处理错误，处理完错误之后，回到查询执行的流程中，停止当前正在做的工作、关闭网络连接等，返回开始处重做查询。重做查询是通过发生错误的时候做的标记来判断的。在重做开始的时候，为了防止主备机没有切换好再次出现错误，需要循环检测 FTS 是否已经做好了主备机切换，直到 FTS 做好了主备机切换才允许开始重新查询。由于第一次查询的时候已经保存了查询计划，所以重做查询的时候不需要重新生成查询计划，而且使用与上次相同的查询计划能够保证第一次查询的时候保存的中间结果能够在查询重做的时候使用。查询重启的过程如下图所示：

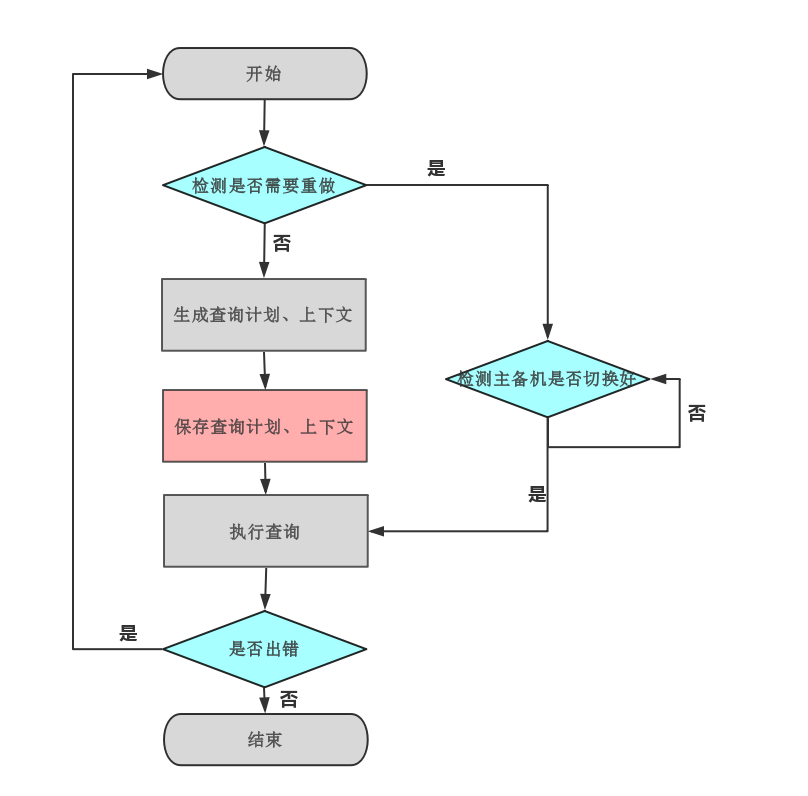


图8 查询重启方案流程图

#### 4.2.2.2方案实现

设置全局变量executorFailed来标记此次查询是否是重启的查询；并且添加全局的内存上下文变量MemoryContext SavedPortalInfoContext来记录用于查询解析、生成查询计划和查询门户的上下文。在检测到主节点宕机情况，probePublishUpdate会返回真值后，我们将布尔型变量executorFailed置成true。在postgresMain()使用AllocSetContextCreate()函数来分配内存上下文时，也需要初始化添加的内存上下文SavedPortalInfoContext。并且在CheckForResetSession()函数来重置session的相关信息时，应该区分是否是重启的情况。若是重启，实际上还是原先的会话，会话ID不能改变。并且在等待输入命令的函数ReadCommand()之前，应该等待主备机切换成功。在exe\_simple\_query()函数中，本方案需要在每次修改上下文的地方保存当前分析、重写和计划所改变的上下文，存入SavedPortalInfoContext中。当重启之后，使用保存的上下文进行查询。从exe\_simple\_query()函数返回后，应该对executorFailed和SavedPortalInfoContex变量进行重置，以便下次重启使用。

在这里需要注意的是，因为重启只是重复执行之前的命令，执行的总命令数量没有改变，所以在遇到命令数量增加的函数应该区分对待。

# 5查询一致性保证与实现

## 5.1问题描述

在加入重启机制后，为什么查询一致性会出现问题呢？通常情况下，在数据库执行查询期间，其他用户往数据库插入数据的情况并不少见。倘若查询能正常完成，此次查询的结果对应的是查询开始时的数据库状态。例如，用户输入查询，系统开始处理这个查询后，其他用户输入插入操作修改了数据库表。此时，数据库的某个节发生故障宕机。在重启机制的作用下，系统重启查询。由于第二次查询完全独立于第一次查询，所以第二次查询可能查询到第一次查询之后插入的那条数据。由于用户并不能感知查询过程中发生了错误而重做，所以用户期待的查询结果应该是第一次查询的结果，即查询结果应该对应于第一次查询启动时的数据库状态。这种情况下，就会出现查询结果不一致的错误。

上述说明了可能会出现查询结果不一致的情况，但并非所有修改操作都能引起查询结果不一致的情况，下面我们举例说明。

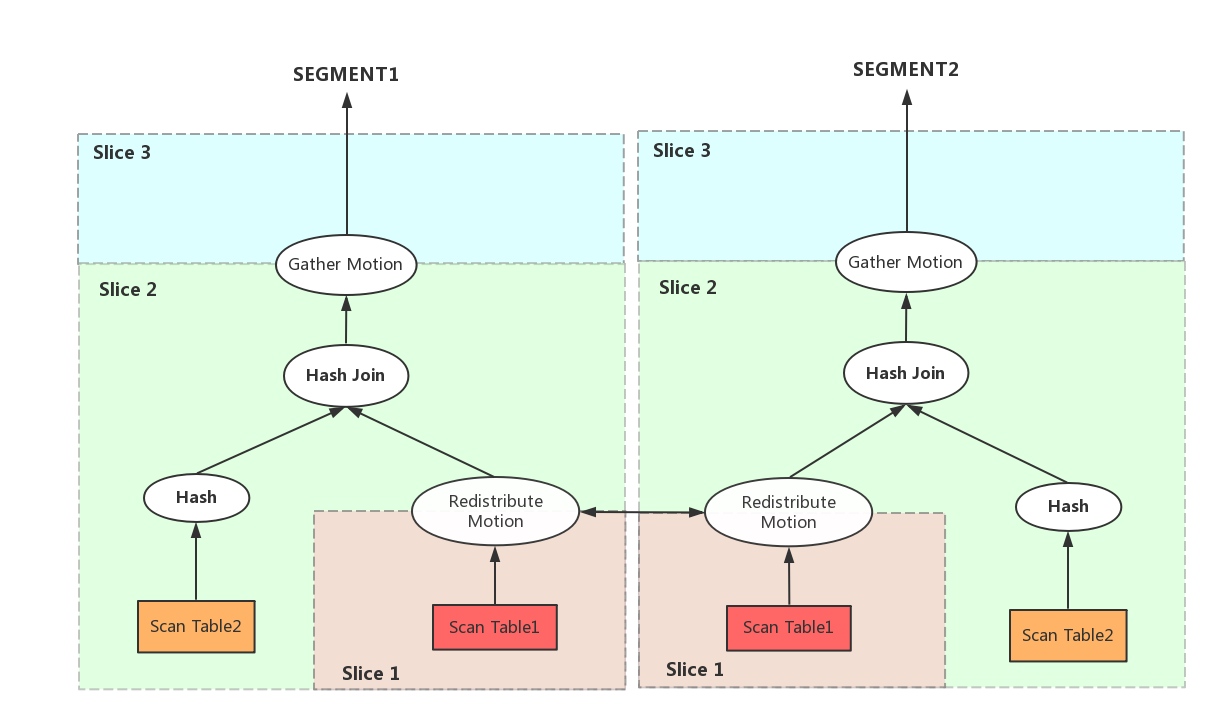


图9 简单查询示例

如图，Table2扫描之后生成的Hash表与Table1扫描、重分布后进行哈希连接，此时，在第一次查询之后插入一条数据，会发生以下三种情况：

(1) 此时已保存Slice2中的hash表，再插入数据到Table2。由于第二次查询时直接读取了保存在本地的hash表，从而不需要读取Table2，第二次不能查询到插入的那条数据。

(2) 此时未保存Slice2中的hash表，就插入数据到Table2。由于第二次查询时不能从本地读取hash表，所以需要读取Table2，第二次可以查询到插入的那条数据。

(3) 无论Slice2中的hash表是否已经保存，插入数据到Table1。由于第二次查询需要读取Table1 ，所以第二次可以查询到插入的那条数据。

综合三种情况来看，**如果保存的中间结果没有被修改，则不会出现结果不一致的情况；否则，就会产生查询结果不一致的情况**。

## 5.2 Greenplum现有保证事务一致性的方案

### 5.2.1 多版本并发控制

Greenplum使用PostgreSQL多版本并发控制（MVCC, Multi-Version Concurrency Control）模型来维护数据的一致性。查询进行时，看到的是数据库开始时的状态（快照），而不是数据库最新的状态。快照是在语句或事务开头可见的一组行。执行时，查询无法查看其他并发事务所做的更改，这确保查询看到数据库的一致视图。读取元组的查询无法阻塞等待写元组的事务；写入行的查询也不能阻塞读取行的事务。这允许比使用锁来协调读取和写入数据更具灵活性，极大地提高了并发处理能力。

每个事务会被分配唯一的事务ID（XID），一个递增的32位值。当新事务开始时，将为其分配下一个XID。当事务插入一条元组时，XID被保存为这个元组的xmin(为了避免混淆，我们记作t\_min)。当事务删除一个元组时，XID被保存为这个元组的xmax(为了避免混淆，我们记作t\_max)。更新一条元组被视为删除和插入，因此将XID保存为当前元组的xmax(t\_max)和新插入元组的xmin(t\_xmin)。

事务启动创建快照的过程简单说就是在事务启动的时刻，遍历当前所有活动的（还未提交）事务，记录在一个活动事务ID数组中；选择所有活跃事务中最小的XID，记录在xmin中，选择所有已提交事务中最大的XID，加1后记录在xmax中。那么：

(1) 所有事务ID小于xmin的事务可以被认为已经完成，即事务已提交，其所做的修改对当前快照可见；

(2) 所有事务ID大于或等于xmax的事务可以被认为是正在执行，其所做的修改对当前快照不可见；

(3) 对于事务ID处在 [xmin, xmax]区间的事务,需要结合活跃事务列表与事务提交日志CLOG，判断其所作的修改对当前快照是否可见。

事务判断某个元组是否可见的方法是：根据事务开始时创建的快照确定哪些事务对于当前事务时可见的，在从外存中获取一个页面的数据之后，由一个检验函数（HeapTupleSatisfiesMVCC等一系列函数）检验所有该页面数据的数据对当前事务的可见性，如果不可见，则将数据除去，最后返回所有可见数据。

下面举例说明：

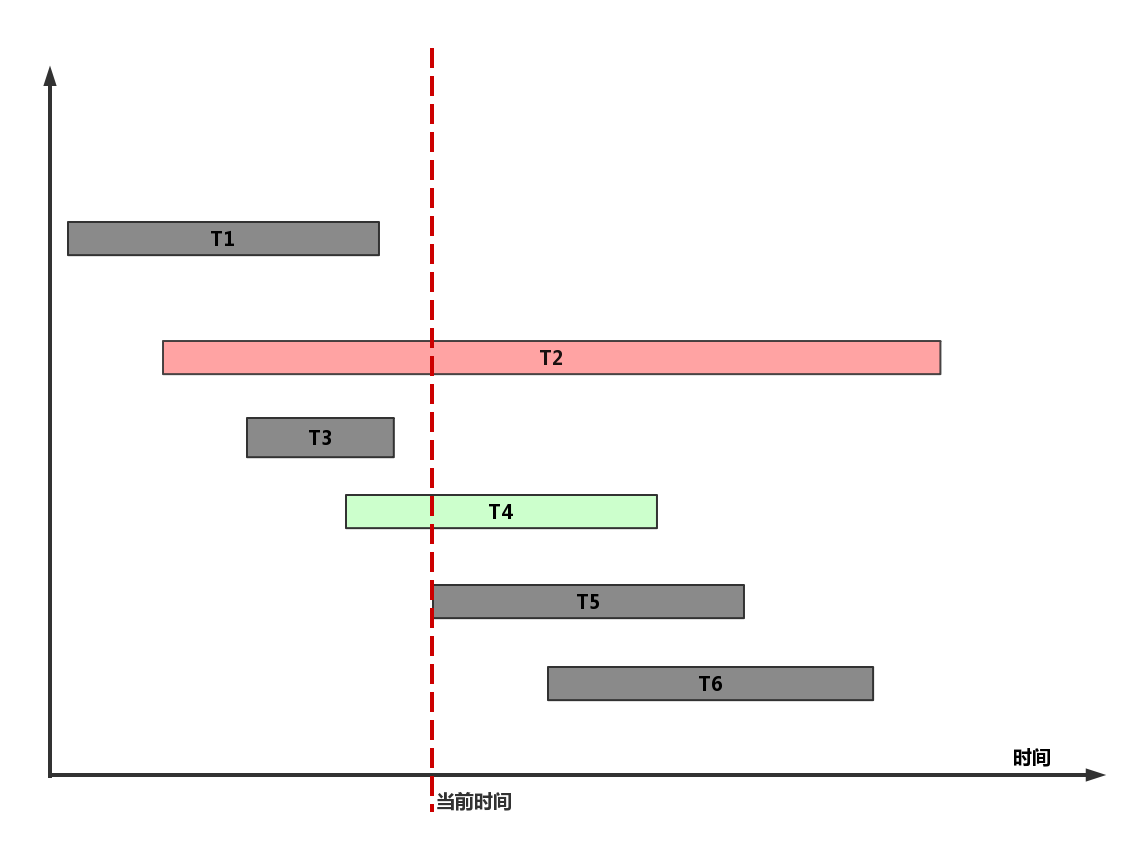


图10 MVCC简单示例

当前事务为T5，在T5开始时创建快照，获取活跃事务列表T1、T2、T3、T4。

① xmin = T2，所以T1已提交，其所做的修改对当前快照可见；

② xmax = T（3+1）= T4，T4和T4之后开始的事务是正在执行，其所做的修改对当前快照不可见；

③ T2-T4之间的事务需要结合活跃事务列表与事务提交日志CLOG，判断其所作的修改对当前快照是否可见，计算结果知T2正在执行，T3已完成。

综上，T1、T3已提交，所做修改可以被T5看到；T2、T4正在执行，所做修改不能被T5看到。

如果一条元组的t\_xmin = T1，t\_xmax = T2，则T5可以看到这条元组；

如果一条元组的t\_xmin = T1，t\_xmax = T3，则T5不可以看到这条元组；

### 5.2.2 slave上本地XID与分布式事务ID的映射关系

XID是数据库的属性。每个slave都有自己的XID序列，无法与其他slave的XID进行比较。master使用会话ID来协调与slave的分布式事务。slave维护分布式事务ID与其本地XID的映射。master使用两阶段提交协议来协调跨所有slave的分布式事务。如果事务在任何一个slave上失败，它将在所slave上回滚。

slave会给每个slice开启一个进程，在接收到master下发的查询计划之后开始这个slice的查询工作。每个slice有自己独立的事务，每个事务都有自己的事务信息，包含本地事务信息和分布式事务信息。本地事务不能不与其他slave上的本地事务相比较，所以需要master用分布式事务来统一管理，分布式事务信息是在master和所有slave上保持一致的，所以本地事务信息和分布式事务信息的映射关系就很重要了（由于master上不需要做本地快照和分布式事务信息的映射，所图中没有展示出master的本地快照）。

slave上有两类slice，一类是需要将数据通过Gather Motion交给master的，如图中的slice2；另一类是将数据通过广播或重分布交换到本机的其他slave上的，如图中的slice1。master生成分布式事务信息，在下发查询计划的时候也将分布式事务信息下发给每个slave的每个slice。slice2生成本地事务ID和本地快照之后，结合分布式事务信息就组成了完整的事务信息，并将分布式事务信息保存在共享内存中，slice1等待从共享内存中获取slice2的本地事务ID和本地快照，一旦读取成功也组成了完整的快照。这样就保证了同一个数据库实例中的快照信息是一致的。简而言之，slice2生成了本地事务与分布式事务的对应关系，slice1保持自己的事务信息与slice2相同。

## 5.3 设计方案

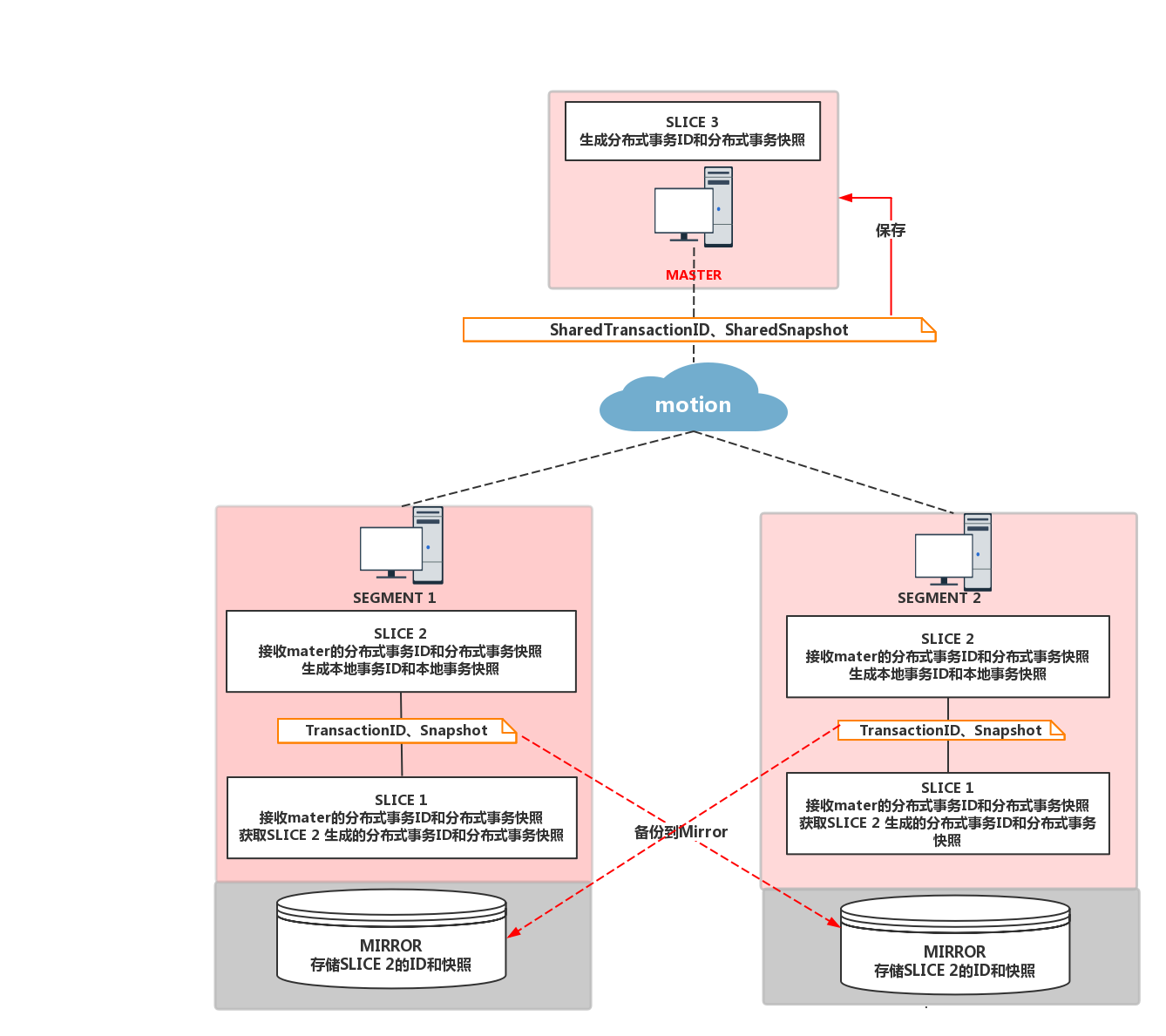
为了保证两次查询结果的一致性，我们了解到Greenplum使用MVCC来控制事务一致性，并且知道**本地事务与分布式事务的对应关系主要由向master提供数据的slice来维护**，所以我们的方案是保存该slice的本地快照信息。借用上例说明，slave获取到slice2本地快照时，我们将快照相关信息保存，并备份。这样，Primary和Mirror都会存有快照信息。

图11 一致性保证解决方案

假设slave1发生宕机，就会如下操作：

(1) slave1的mirror启动开始重做。slice2从mirror上读取到保存的slice2的本地快照信息，并生成本地事务ID，与master派发下来的分布式事务信息结合生成完整的事务信息；slice1从slice2处获取到slice2的完整的事务信息。

(2) slave2在primary上重做。slice2从primary上读取到保存的slice2的本地快照信息，并生成本地事务ID，与master派发下来的分布式事务信息结合生成完整的快照信息；slice1从slice2处获取到slice2的完整的事务信息。

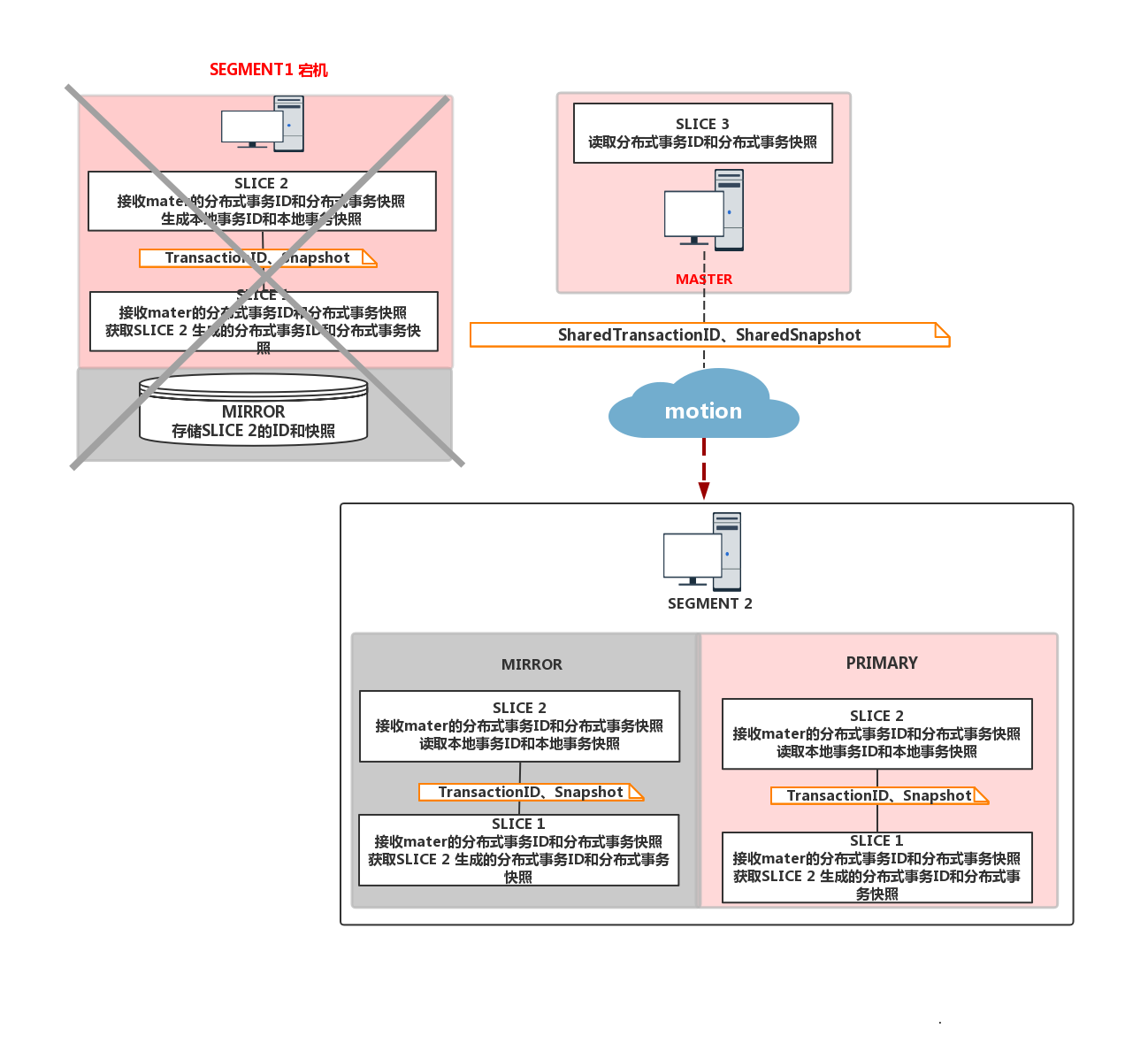


图12 一致性作用示例

## 5.4 方案实现

### 5.4.1 保存快照

在进入exec\_simple\_query()，生成dispatcher查询计划树或者executor准备执行计划之前，系统会调用函数Get\_SnapshotData()初始化快照或者获得快照。Get\_SnapshotData()负责初始化快照，更新快照以及segment之间共享快照（sharedLocalSnapshot）和本地事务快照（QEDtxContexInfo.distributedXid）的一致性。我们修改这个函数，在最后更新快照前，将快照中的数据写到磁盘的文件里，并且调用FileRepPrimary\_MirrorTmpFileWrite()将文件同步到Mirror。一旦executorFailed是true，就从Mirror上读取快照信息并且返回这个快照。

### 5.4.2 保存事务ID

事务开始之后，会调用函数GetNewTransactionId()来生成新的事务ID。所以设置一个全局变量oldTransactionId来记录当前事务ID。我们要修改GetNewTransactionId()函数，当系统正常执行时，保存oldTransactionId并且将其写在磁盘并同步到Mirror上。在executorFailed为true的时候，跳过分配ID的步骤，直接使用oldTransactionId。并且，在CommitTransaction中，我们需要重置快照中的xmax值。如果出现宕机的情况，快照的xmax应该重置为oldTransactionId，因为可见性已经变了。

### 5.4.3 保存分布式事务上下文

在dispatcher派发查询计划时，会调用cdbdisp\_buildPlanQueryParm()函数来创建DispatchCommandQueryParms结构体，并且初始化和序列化一些参数信息，包括命令，查询计划，查询上下文，slice数量等。此处，我们需要关注分布式上下文有关的参数，serializedDtxContextInfo，serializedDtxContextInfolen。在查询正常执行时，我们需要将这两个信息存储下来，并且在executorFailed为true时将其取出重用。

# 6保存点选择模型设计与实现

接下来，要解决的问题就是在生成计划阶段，如何合理选取阻塞点保存。我们的策略建立在衡量保存代价和查询成功率的基础上。因此，本部分会首先介绍GPDB原有的代价估算公式。

## 6.1 代价估计

在生成查询计划阶段，查询规划器会建立建立多条不同的路径（Path）来枚举表的不同访问方式、不同连接方式、不同的连接顺序。每条路径都会有自己的的代价，规划器会根据代价打大小来判断路径的优劣。

衡量某条路径的代价主要考虑CPU代价和磁盘存取两个方面。磁盘代价以从磁盘顺序存取一个页面的代价为单位，所有其他形式的代价计算都是相对磁盘存取代价来计算的。

表格1 GPDB不同代价含义及其默认值

|  |  |  |
| --- | --- | --- |
| 类型 | 含义 | 默认值 |
| seg\_page\_cost | 顺序存取页的代价 | 1.0 |
| random\_page\_cost | 随机存取页的代价 | 4.0 |
| cpu\_tuple\_cost | CPU处理一个元组的代价 | 0.01 |
| cpu\_index\_tuple\_cost | CPU处理一个索引元组的代价 | 0.005 |
| cpu\_operator\_cost | CPU执行一个操作符或函数的代价 | 0.0025 |
| effective\_cache\_size | 量度数据库和操作系统缓存的磁盘页的数量 | |

内核通常会做一些预读（read-ahead）优化，这这意味着seg\_page\_cost会小于random\_page\_cost。然而如果数据完全都在内存中，把两个值设成一样。effective\_cache\_size是粗略估计的，内核的磁盘缓存是不可忽略的。尽管，为这些参数取常数是过于简单的，但是即使比这个程度更详尽也很难做到有效的估计。上述所有的参数都是可以根据不同的平台调试。

对于每个路径，GPDB会计算两个独立的成本。

表格2 GPDB采用代价类型

|  |  |
| --- | --- |
| 总代价 | 获取所有元组的总代价 |
| 启动代价 | 在获取第一个元组之前花费的代价 |

在一些情况下，像是有Limit语句，或是执行exisits()子选择的时候，没有必要获取所有的元组。也可以通过在startup\_cost和total\_cost之间插值获取部分结果的代价，

表格3计算启动代价和总代价所用的参数

|  |  |
| --- | --- |
| 基本参数 | 表元组数、表磁盘块数、符合选择条件的元组数 |
| 统计信息 | 每个表和索引中的元组数，以及每个表和索引占据的磁盘数。 |
| 直方图信息 | 各个属性值出现次数的统计信息 |

### 6.1.1 路径的代价估算的基本步骤

首先根据统计信息和查询条件，估算出这次查询要进行的I/O次数以及要取出的元组个数，计算出CPU代价，最后综合考虑CPU代价和I/O次数即可得到最后的代价。

P表示在执行是所要访问的页面数，反应了磁盘I/O次数；T表示在执行时，所要访问的元组数，反映了CPU开销；W是在磁盘I/O代价和CPU开销间的权重因子。

每个算子会有专门的函数来计算代价，本文感兴趣的是如何计算阻塞算子的代价，下文会分别介绍GPDB排序（Sort）、聚集（Aggregate）、哈希连接（Hash）计算代价的方式。

### 6.1.2 哈希（Hash）

因为哈希算子是哈希连接的一部分，所以在GPDB中，将哈希算子的代价计入哈希连接中。所以接下来会介绍在GPDB中哈希连接的代价时如何计算的。在函数**voidcost\_hashjoin**(HashPath \*path, PlannerInfo \*root)中实现计算哈希算子的代价。

哈希连接的原理是，将小表作为Hash表，扫描另一个表M的每一行数据，根据连接条件去映射建立的哈希表。所以代价可以分为内表建立哈希表的代价和用外表映射的代价。

在计算哈希代价时，会将Joinpath结构体(如下图)传入外表路径outerjoinpath和内表路径innerjoinpath，这两个结构体指针内包含着该路径的代价、元组数以及限制条件的个数，在计算启动代价和执行代价时会使用到。

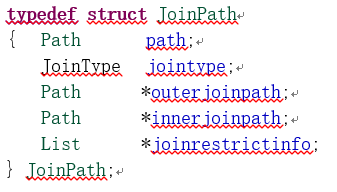


图13 JoinPath结构体

哈希连接的启动代价如下，前两项的启动代价由outerjoinpath和innerjoinpath传入。如果内表太大，哈希计算不能仅仅在内存进行，此时内表进行哈希计算的代价需要计算从磁盘读写页的代价。

可以看到，计算哈希函数的代价被算到哈希连接的启动代价中了。计算哈希函数的成本：必须对每个输入元组进行一次，为每列哈希函数收取一个cpu\_operator\_cost，为每个内部行收取一个cpu\_tuple\_cost，以模拟将行插入哈希表的成本。

而哈希连接的执行代价如下：

在计算代价时，为了计算在哈希桶内查找的代价，需要预估每个哈希桶的大小和分成的batch数量，batch数量是指当内存装不下整个表是分批处理的数量。当分批处理时，应该考虑哈希函数选择度的问题。

选择度（Selectivity）是GPDB定义的一个double型的变量，用来定量的地描述代价估算公式中的权重因子，其计算要综合考虑约束条件中的操作符、约束常量、索引中的元组数、某字段的最大值和最小值等等。比如，在计算内表关系桶的大小实际上是一个选择度的计算：如果假设散列键得分布在算完限制条件后还是相同的，数据也是完美分布，每个桶中的元组数量相同，桶大小就是内表大小的1/numOfbuckets，即选择度是1/numOfbuckets但是仅仅在满足下列两种情况下：（1）至少有numOfbuckets个不同的数据值。（2）数据分布不倾斜。否则，桶将不均匀地被占用。在实际的情况下，会使用统计值来进行预估，如果没有统计资料，系统会使用0.1作为预设值。

### 6.1.3 排序（Sort）

计算Sort算子代价函数是**voidcost\_sort**(Path \*path, PlannerInfo \*root,List \*pathkeys, Cost input\_cost, **double** tuples, **int** width,**double** limit\_tuples)。排序时，会传入一个输入代价作为启动代价，还有限制的元组数是在查询语句有limit条件下使用的。排序分为三个情况计算代价。

(1) 如果排序的数据总量是小于内存容量的，那么排序就可以完全在内存进行排序，使用快速排序那么就不需要I/O代价，只需要收取nlog(n)的元组比较的代价。

(2) 如果，排序是被限界的，也就是说只需要前K个元组，那么只需要比较n\*log(K)次。

(3) 如果排序的总量超过了内存，GPDB会转换成条带式的归并算法，这个算法也是需要收取nlog(n)的元组比较的代价，但是对于每一趟的归并，还会加上磁盘读写的代价（GPDB假设有1/4的是随机读写，3/4是顺序读写）。因为平均初始遍数应该大题是内存的两倍。

此外，提取每个元组还需要收取CPU运算的成本。所以，

其中，K要根据不同的情况选择，磁盘I/O也是根据情况加上。

### 6.1.4聚集（Aggregate）

计算Aggregate算子的代价函数是：**voidcost\_agg**(Path \*path, PlannerInfo \*root,AggStrategy aggstrategy, **int** numAggs,**int** numGroupCols, **double** numGroups,Cost input\_startup\_cost, Cost input\_total\_cost,**double** input\_tuples, **double** input\_width,**double** hash\_batches, **double** hashentry\_width,bool hash\_streaming)。对于每个输入和输出的元组，收取cpu\_operator\_cost，如果有分组的话，对于每组每元组要收取比较的代价cpu\_operator\_cost。如果没有分组的话，聚集的结果只有一个，但是如果有分组的话，聚集的结果就是组的个数。对于每个输出元组，收取cpu\_tuple\_cost。

聚集也分为三种不同情况来讨论代价。在GPDB中，有三种不同的聚集。第一种就是简单的聚集，没有分组，直接收取CPU处理的代价即可。另外两种，是排序聚集（AGG\_SORTED）和哈希聚集（AGG\_HASHED）。若输入的元组已经排好序了，排序聚集是更好的，因为这没有内存泄漏的风险。两种聚集的CPU消耗是一样的，但是排序聚集的启动代价更小。哈希聚集则有可能内存溢出，产生额外的I/O代价。

### 6.1.5物化代价的计算

前面已经提到，容错方案需要将哈希，排序，聚集算子的输出结果进行保存。本小节将会介绍各个算子的物化代价时如何计算的。

对于排序、聚集和哈希来说，物化代价很好计算。排序就是输出元组数所对应的页面大小；聚集一般是一个元组，当有grouping的时候是分组个数个元组所对应的页面数；哈希就是内表元组数所对应的页面数大小。可以调用**staticdouble page\_size**(**double** tuples, **int** width) 函数来将输出的元组数目转换成对应的页面数。width是根据统计算出的每个元组的平均宽度。

## 6.2路径代价的传递和打印

现在代价已知，那么如何将新的哈希计算代价和物化代价保存到查询计划中呢？

在GPDB中用Path这个结构体来存储路径信息，路径信息中包含了这条路径的启动代价和总代价。如下图所示：

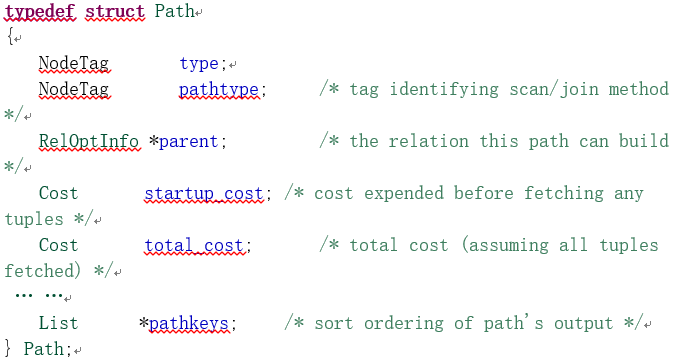


图14 Path结构体示例

type 是代表这个节点的类型（哈希、排序、重分布等等）；pathtype为了标识扫描或者连接，上面们看到连接有自己独特的结构体；parent 指向整个路径；

startup\_cost和total\_cost分别是上文提到的启动代价和总代价；pathkeys是输出结果的键值排序。

Path结构是只是用来保存路径信息，用于遍历所有路径的可能性。最终，规划器会找出一个最佳路径，存到Plan结构体中。Plan结构体也包含了计划中每个节点的启动代价和总代价，如下图所示：

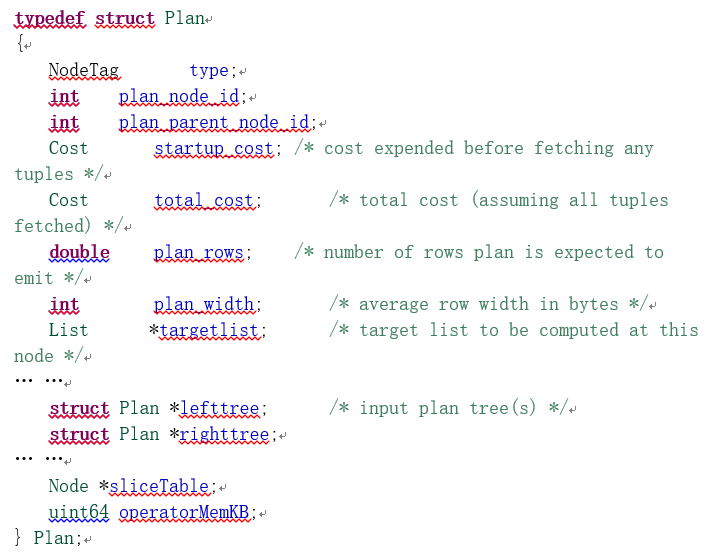


图15 Plan结构体示例

通过上一小节我们知道，不同的算子会有自己对应的代价计算函数，函数会将计算出的代价保存到Path结构体中。在生成查询计划时，会生成不同类型的计划节点。在创建节点时，需要进行代价的拷贝，这是需要调用函数copy\_path\_costsize(PlannerInfo \*root, Plan \*dest, Path \*src)来拷贝不同路径之间的代价。在查询计划执行后，需要通过对代价进行比较，得出最优路径。这时，需要调用函数copy\_plan\_costsize(Plan \*dest, Plan \*src)来复制Path路径中的代价。下图是创建连接计划时代价拷贝机制。

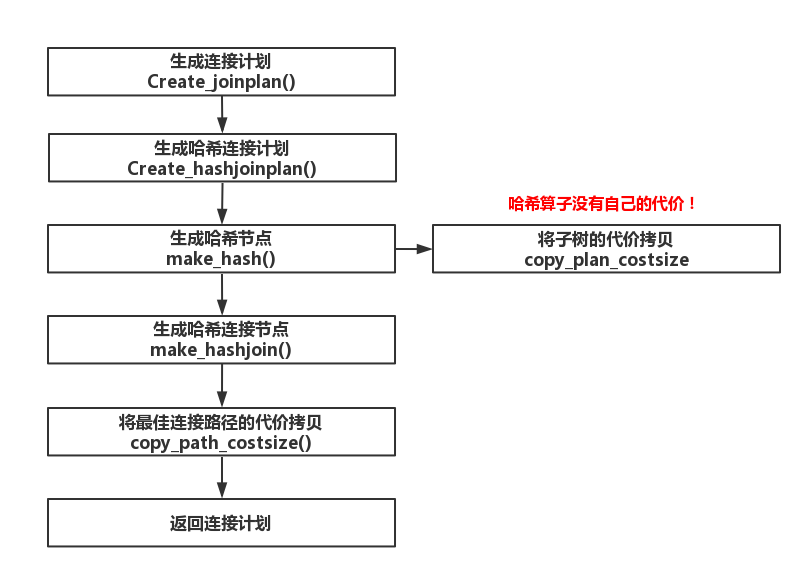


图16 代价拷贝机制示意图

为了不影响程序其他部分对Cost的使用，我在Path和Plan结构体中加入了两个Cost变量，new\_total\_cost和materialized\_cost分别记录总代价和物化代价。并且修改函数add\_sort\_cost()、add\_agg\_cost()、copy\_path\_costsize()、copy\_plan\_costsize()，将正确的cost写到Plan对应的节点中。

查询计划的输出

GPDB为快速读写序列化的结构体写了专门的函数。在outfast.c、outfuncs.c和readfast.c有读写Path和Plan结构体的函数，要想在使用explain命令时输出相应的代价，需要修改这些函数。

## 6.3 物化节点选取算法

现在，我们已经获取了规划器产生的查询计划，并且知道每个节点的总代价和物化代价。接下来的任务就是在所有的物化节点中，选取合适的节点使查询在规定时间内的成功率最高。

如图，给定一个查询计划树（以哈希连接为例）。根据哈希连接的原理，查询计划树的基本单元是：小表的扫描，小表做哈希，大表扫描后与小表的哈希结果做哈希连接。如图中红框部分所示的1,2,3,4节点。当我们遍历所有策略组成的解空间时，节点数太多会增加复杂度，再加上哈希连接之间不会出现排序、聚集这样的物化节点，除了这三种节点之外的节点我们不考虑，所以需要将查询计划树压缩一下。压缩并不是简单的将没用的节点摘除，还需要将空白节点的代价进行合并。合并的规则如下：

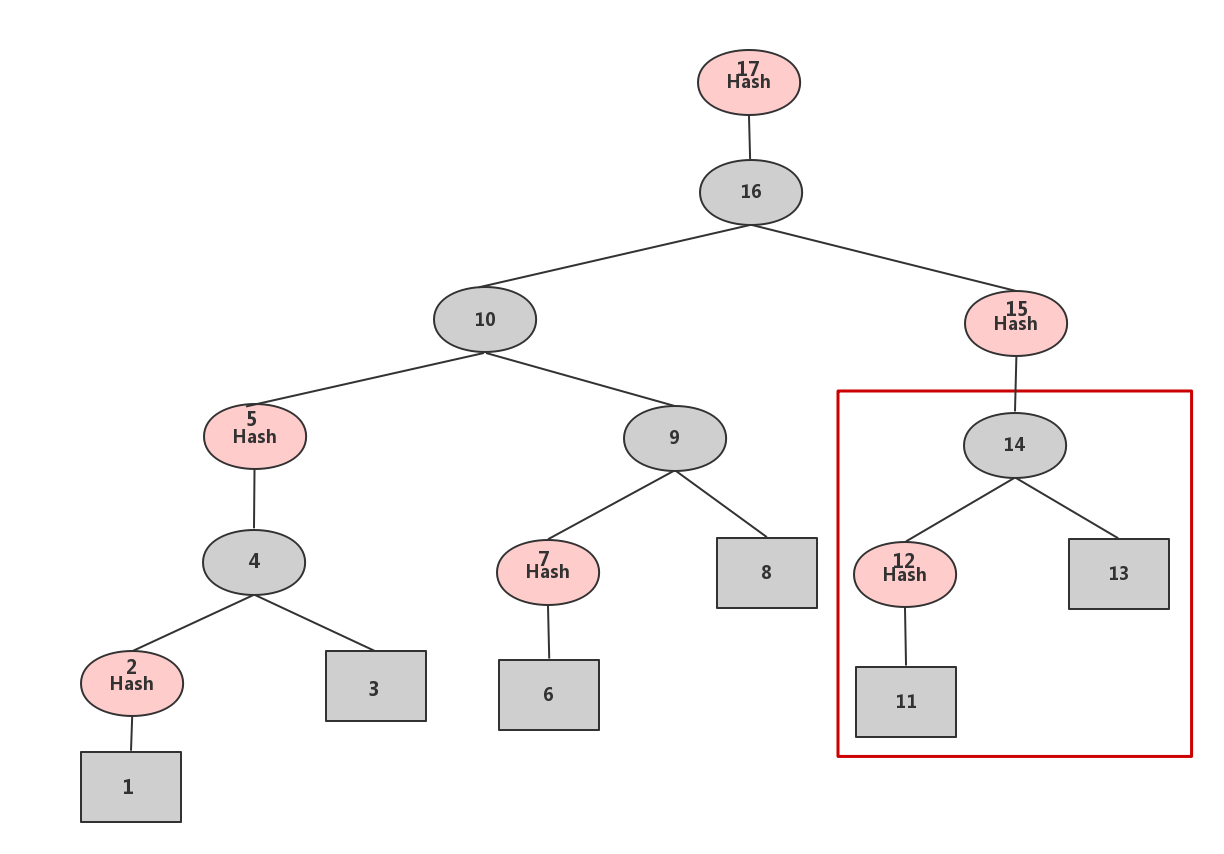


图17 简单查询计划树例图

(1) 按照后序遍历的方法遍历整棵树；

(2) 若节点类型是Hash，则保留左孩子和右孩子，并且返回node本身。

(3) 若一个节点有左右两个孩子，就返回节点本身；

(4) 若一个节点有左孩子或者右孩子，就返回左孩子或右孩子；

(5) 若一个节点没有左孩子也没有右孩子，即是叶子节点，就把他删除。

最后得到这样的简化树。

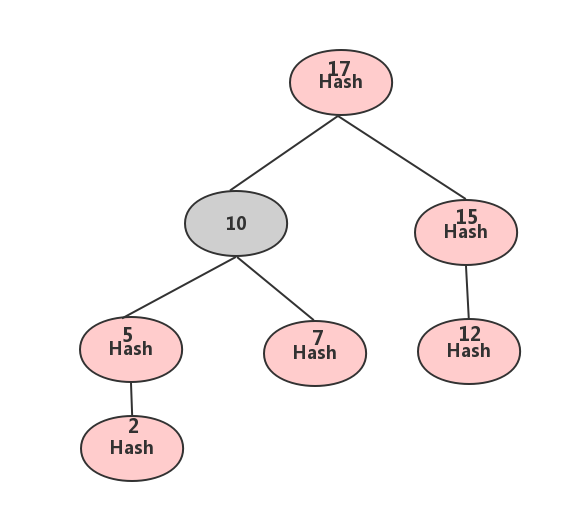


图18 压缩查询计划树示例图

在本模型中，我们用I/O来代替时间，I/O是线性的，这样就不会存在并行执行的问题。所以我们把简化树中的物化节点抽离出存在一个一维数组中。假设数组中存放了n个节点，那么这n个节点所组成的策略空间有2^n个。那么接下来的问题就是，给定一个策略如何评价其优劣呢？

在之前的部分提到，我们的目标是提高系统的可靠性，即在指定时间完成查询的概率。假设系统中某个节点在时间0~t时间内出现失败的概率符合概率分布。在整个查询中间不宕机的概率是。若是在查询中间宕机一次，还能在时间内完成的概率该怎么算呢？我们把整个查询从时间中点分成两段，每部分占用的时间是。那么，查询若要宕机一次，查询要在指定时间完成，只能在前的情况下宕机，否则就会超出时间，此时的概率是：

若是宕机2次，情况就多了起来，这时得出的概率公式就很复杂。这样算的目的是想通过概率公式，找出一个n能使概率达到一个最大值，但是这样的做法并没有多大意义。因为这样得出n之后，只能知道选取几个物化点，并不知道怎么选取。更何况，在假设中，物化点的选取是平均的，更不具实际意义。本文希望能选取更符合实际，更能用程序实现的方法。

在实际中，物化点的间隔是不确定的。如图X所示的A、B、C三点。我们要用I/O代价来代替时间，所以要把A、B、C三点之间的代价进行转换成如下图所示的树：

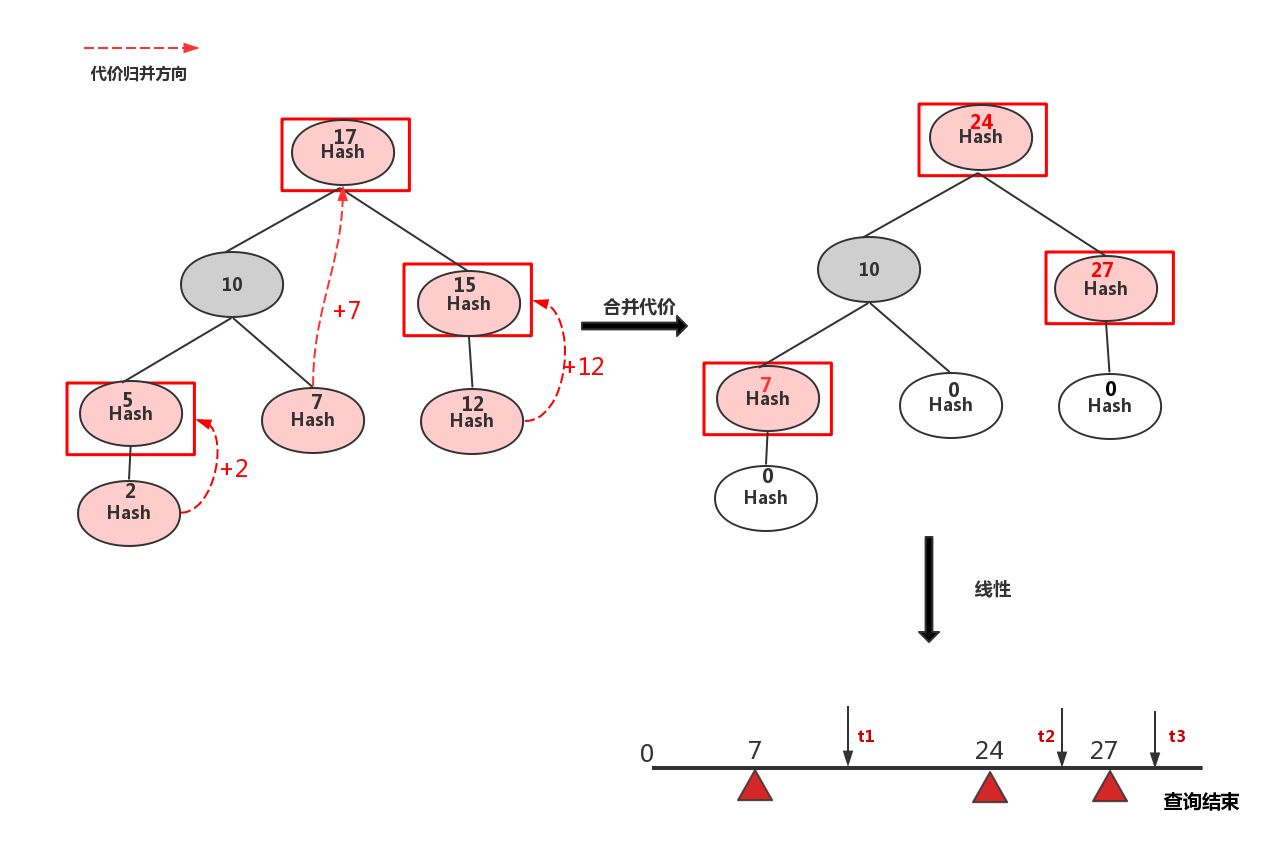


图19 合并代价示意图

因为选中的哈希节点所在的代价实际上相当于查询所经历的时间。这样，我们可以轻松地模拟出在t时刻宕机所发生的情况。如下图，在t1,t2,t3时刻宕机，只需要从代价为7，24，27处重分别重启，这时，查询的总执行时间T’为 t1 + (t2-7)+(t3-24)+(T-27)

可见，给定宕机时刻和保存点，我们可以算出这个查询究竟能不能再指定时间内完成。只需要将宕机时刻生成很多次，我们就就能利用统计学原理模拟实际的宕机情景，从而计算查询在规定时间内完成的概率。实际的宕机时间的分布函数为F(t)=1-e^(-λt)

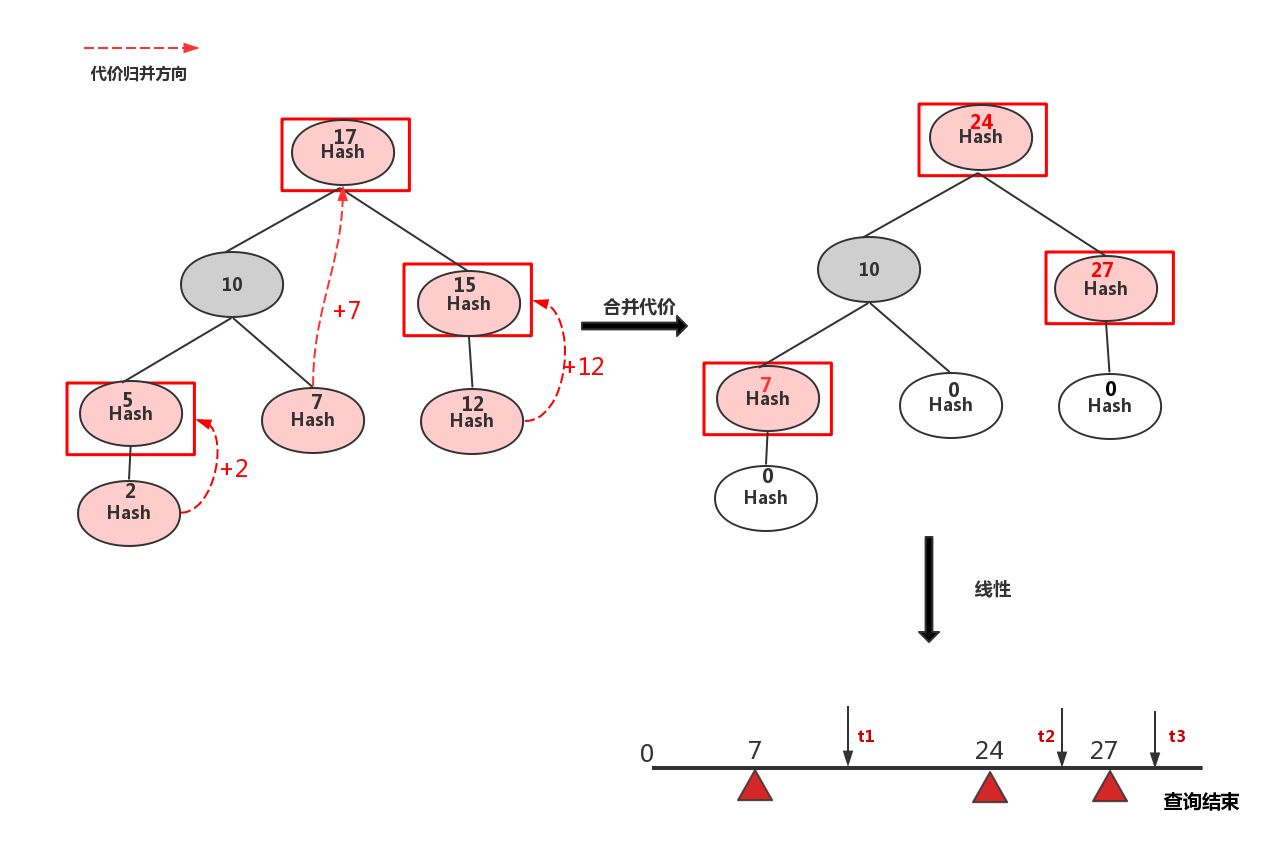


图20 蒙特卡罗在策略中的应用示意图

所以，本文提出方法是用蒙特卡罗方法按照分布函数F(t)模拟生成宕机时刻，计算出每种策略下的查询成功率。以此来筛选不同的策略。

算法步骤

1. 简化查询计划树，压缩节点，合并代价，尽量减少查询遍历时间。

2. 将其中需要物化的节点挑出，组成数组。

3. 枚举所有可能的策略组合，使用蒙特卡洛算法计算每种策略的成功率，当成功率达到指定值，就停止遍历。

# 7实验测试与结果分析

## 7.1 实验环境

本实验采用了七台配置一致的虚拟主机组成，一台作为 Master节点，其余六台作为 Slave节点。其中每台 Slave上配置两个 Primary Segment和两个 Mirror Segment。同一个 Segment的 Primary和 Mirror配置在不同的 Slave主机上。

每台机器配置如下：

表格 4实验环境配置

|  |  |
| --- | --- |
| 系统版本 | centos 6.5 |
| 内核版本 | Linux 2.6.32-431.el6.x86\_64 |
| 处理器 | CPU Genuine Intel（R）cpu MHz: 2399.318 |
| 核数 | 8 |
| vendor\_id | GenuineIntel |
| model name | Westmere E56xx/L56xx/X56xx (Nehalem-C) |
| cpu MHz | 2399.318 |
| cache size | 4096 KB |
| address sizes | 40 bits physical, 48 bits virtual |
| 内存总量 | 32880596 KB |
| 磁盘容量 | 500G |

## 7.2实验方案及结果

### 7.2.1 TPC-H 测试集

本实验采用 TPC-H 生成试验测试数据和查询。 TPC-H 基准测试是由 TPC-D (由 TPC 组织于 1994 年指定的标准,用于决策支持系统方面的测试基准)发展而来的。TPC-H 用 3NF 实现了一个数据仓库，共包含 8 个基本关系。TPC-H 基准测试包括 22 个查询，用户可以根据需要生成不同数据量的测试集。

### 7.2.2 查询重启性能测试

#### 7.2.2.1实验方案

1. 验证正确性。选定查询时间最长的查询7，使用不同方式关闭不同的slave，查看重启情况。宕机方式包括：切断网络连接、直接关机。
2. 验证重启性能。选取查询时间较长的1号，3号，7号，9号，17号，19号，21号，测试重启时间。

#### 7.2.2.2实验结果

（1）功能性验证

表格 5查询重启方案

|  |  |  |
| --- | --- | --- |
| 宕机方式 | Slave1 | Slave2 |
| 关机 | 成功重启 | 成功重启 |
| 切断网络连接 | 成功重启 | 成功重启 |

1. 重启时间

表格 6重启测试结果

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 查询号 | 1号 | 3号 | 7号 | 9号 | 17号 | 19号 | 21号 | 平均时间 |
| 重启时间 | 63s | 68s | 75s | 89s | 57s | 42s | 60s | 64.9s |

### 7.2.3 查询一致性验证

#### 7.2.3.1实验方案

（1）在查询1进行期间，开启另一个会话自定义一个查询2插入、删除或者更新一条元组。

（2）要保证查询1还未完成，此时关闭某一台slave节点。

（3）看查询结果有没有显示更新。

若有，则没有保证查询结果的一致性；若无，则保证一致性成功。

因为要保证查询1够长，所以本实验选取查询9和查询21。

#### 7.2.3.2测试结果。

（1）查询9查询结果

**nation | o\_year | sum\_profit**

**ALGERIA | 1998 | 28675397.8139**

1. 插入 INSERT

查询过程中执行下面插入语句成功之后宕机。

INSERT INTO orders values(22790,105043,'O',267041.09,'1998-03-1','4-NOT SPECIFIED','Clerk#000000265',0,'no commemts');

查询结果与原结果相同，再次查询结果：

**nation | o\_year | sum\_profit**

**ALGERIA | 1998 | 28694759.1923**

1. 删除

查询过程中执行下面删除语句成功之后宕机。

DELETE FROM orders where o\_orderkey = 22790;

查询结果与原结果相同，再次查询结果：

**nation | o\_year | sum\_profit**

**ALGERIA | 1998 | 28656036.4355**

1. 更新

查询过程中执行下面更新语句成功之后宕机。

UPDATE orders SET o\_orderdate = '1997-05-1' WHERE o\_orderkey = 22790;

查询结果与原结果相同，再次查询结果：

**nation | o\_year | sum\_profit**

**ALGERIA | 1998 | 28656036.4355**

（2）查询21结果（部分）：

**s\_name | numwait**

**---------------------------+---------**

**Supplier#000006695 | 19**

**Supplier#000000409 | 18**

**Supplier#000005242 | 18**

**Supplier#000008520 | 18**

**Supplier#000001276 | 17**

**Supplier#000003799 | 17**

**Supplier#000005953 | 17**

1. 插入

查询过程中执行下面插入语句成功之后宕机。

INSERT into supplier values(6116,'Supplier#000006116','IC,4VtNEWWWWWW',7,'17-455-882-1999',8981.06,'no commemts')

查询结果与原结果相同，再次查询结果：

**s\_name | numwait**

**---------------------------+---------**

**Supplier#000006116 | 24**

**Supplier#000006695 | 19**

**Supplier#000000409 | 18**

**Supplier#000005242 | 18**

**Supplier#000008520 | 18**

**Supplier#000001276 | 17**

**Supplier#000003799 | 17**

**Supplier#000005953 | 17**

**Supplier#000007665 | 17**

1. 删除

查询过程中执行下面删除语句成功之后宕机。

DELETE FROM supplier where s\_suppkey = 6695

查询结果与原结果相同，再次查询结果：

**s\_name | numwait**

**---------------------------+---------**

**Supplier#000000409 | 18**

**Supplier#000005242 | 18**

**Supplier#000008520 | 18**

**Supplier#000001276 | 17**

**Supplier#000003799 | 17**

**Supplier#000005953 | 17**

**Supplier#000007665 | 17**

1. 更新

查询过程中执行下面更新语句成功之后宕机。

UPDATE supplier SET s\_nationkey = 10 WHERE s\_suppkey = 6695

查询结果与原结果相同，再次查询结果：

**s\_name | numwait**

**---------------------------+---------**

**Supplier#000000409 | 18**

**Supplier#000005242 | 18**

**Supplier#000008520 | 18**

**Supplier#000001276 | 17**

**Supplier#000003799 | 17**

**Supplier#000005953 | 17**

**Supplier#000007665 | 17**

### 7.2.4 保存点选取模型测试

7.2.4.1 实验方案

（1）代价保存和输出功能性验证。

执行TPCH测试集所有查询，依次查看EXPLAIN结果。是否正确的输出代价。

（2）蒙特卡罗方法时间测试

选取哈希、聚集、排序节点最多的查询2进行模拟测试。每次蒙特卡罗生成随机数个数设置为10万。平均故障时间选取英特尔2009年发布的SR1630GPMTBF。预估时间不能超过整个查询执行时间的20%。

7.2.4.2实验结果

（1）EXPLAIN输出部分结果如下图红框部分所示：

哈希聚集总代价是：18151.95；物化代价是：958

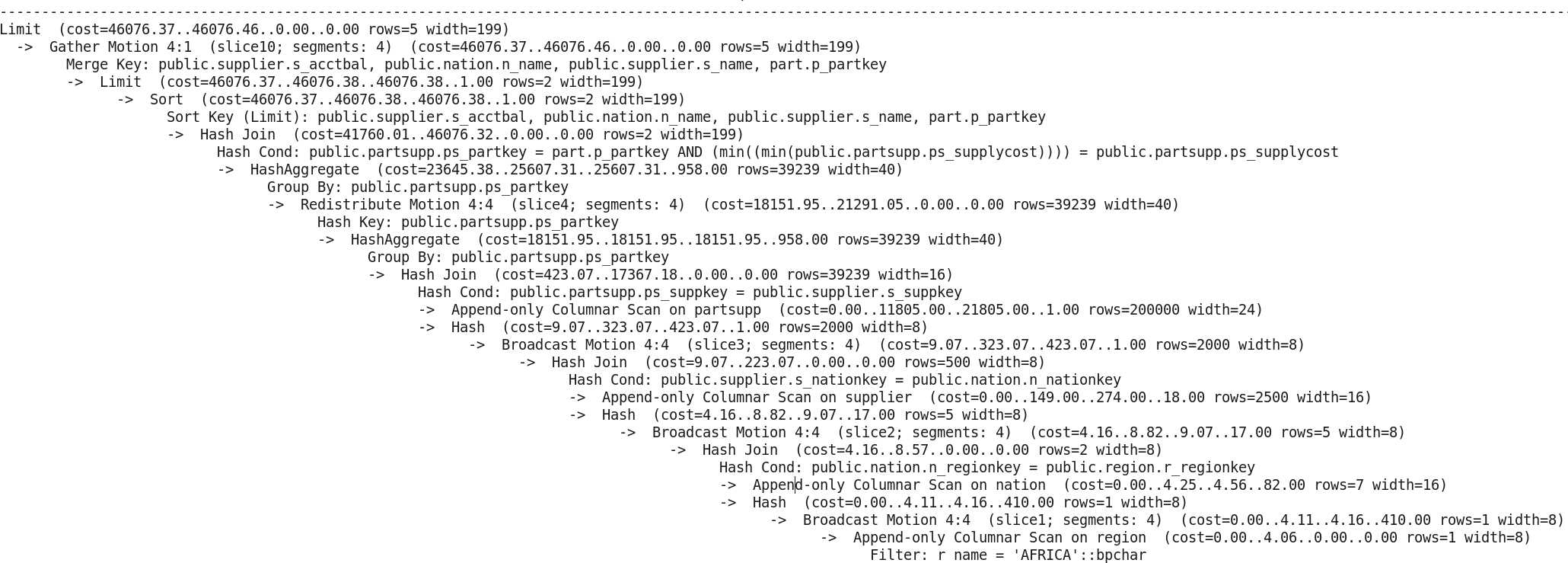


图 21 代价输出结果图示

（2）查询2共有11个阻塞节点。

表格 7 物化策略性能测试结果

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 次数 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 平均时间 |
| 时间 | 10 | 9 | 7 | 7 | 6 | 4 | 5 | 7 | 8 | 10 | 7.3 |

查询2总时间是48s,7.3s的生成查询计划时间可以接受。因为查询2的时间略短，实际情况下有很多长查询没有很多阻塞节点，枚举空间会大大减少，平均花费时间也会降低。

7.2.5 整体系统性能测试

由于保存点选取策略只是模拟实现的，在系统的实际代码中，我们选择物化所有哈希算子节点。

7.2.5.1 实验方案

（1）查询正常进行时修改代码增加的开销对比。

先后使用原版和修改版代码运行12个查询对比结果。

（2）出现宕机时，修改版代码的查询续作时间。

在查询中间宕机一次，对比查看修改版代码的查询时间。

7.2.5.2实验结果

表格 8整体实验结果

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 查询号 | 宕机时查询已执行时间 | 重启后查询续作时间 | 查询正常执行时间 | 宕机后查询总时间 | 与正常查询时间比例 |
| 7 | 257 | 412 | 456 | 669 | 1.46 |
| 9 | 415 | 168 | 417 | 583 | 1.39 |
| 17 | 509 | 96 | 578 | 605 | 1.046 |
| 21 | 497 | 204 | 687 | 701 | 1.039 |

查询7，9的大部分工作是在中间结果物化之后完成的，而这部分工作在查询重启后不得不重做，因此容错机制在Q7上的作用并不大。

## 7.3结论

经验证，本文提出的查询重启机制是正确可行的，而且保持了查询结果的一致性。由于在实际中查询计划中的阻塞节点不是非常多，运用蒙特卡罗方法模拟实际宕机情况是可行的。

# 8 总结

首先，本文提出对时间敏感查询增加系统可靠性的需求。为了适应这种需求，本文通过详细了解GPDB查询控制流程，提出了重启查询的可靠方案；同时，为了保证数据库状态和查询结果的一致性，详细了解保证事务的一致性的MVCC的方法和分布式事务与本地事务的对应关系，并提出保证查询结果一致性的方案；最后，为了提高查询在规定时间内完成的概率，利用蒙特卡罗方法模拟实际故障发生情境，计算每种情境下查询成功的概率，从而选取较为合适的物化节点的策略。最后，结合中间结果保存和同步方面的工作，整个容错系统的性能是可以接受的。

# 参考文献

[1] Yang C, Yen C, Tan C, et al. Osprey: Implementing MapReduce-style fault tolerance in a shared-nothing distributed database[C]// IEEE, International Conference on Data Engineering. IEEE, 2010:657-668.

[2] Babu S. Massively Parallel Databases and MapReduce Systems[J]. Foundations & Trends in Databases, 2012, 5(1):1-104.

[3] Binnig C, Salama A, Zamanian E. DoomDB: kill the query[M]. ACM, 2014.

[4] Salama A, Binnig C, Kraska T, et al. Cost-based Fault-tolerance for Parallel Data Processing[J]. 2015:285-297.

[5] Upadhyaya P, Kwon Y C, Balazinska M. A latency and fault-tolerance optimizer for online parallel query plans[C]// ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June. DBLP, 2011:241-252.

[6] Waas F M. Beyond Conventional Data Warehousing — Massively Parallel Data Processing with Greenplum Database[C]// Informal Proceedings of the Second International Workshop on Business Intelligence for the Real-Time Enterprise, BIRTE 2008, in conjunction with VLDB'08, August 24, 2008, Auckland, New Zealand. DBLP, 2008:89-96.

[7] Isard M, Budiu M, Yu Y, et al. Dryad:distributed data-parallel programs from sequential building blocks[J]. Acm Sigops Operating Systems Review, 2007, 41(3):59-72.

[8] Greenplum database. http://www.greenplum.com/.

[9]Hadoop. http://hadoop.apache.org/.

[10]Shenker S, Stoica I, Zaharia M, et al. Shark: SQL and Rich Analytics at Scale[J]. Computer Science, 2013:13-24.

[11]M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In NSDI, pages 15–28, 2012.