# 概述

## 背景

Hadoop提供了大数据存储和计算的一整套解决方案，完美地解决了大数据的存储和计算问题。但是Hadoop提供的大数据解决方案使用的是Map-Reduce计算框架，这种计算框架适用于大数据的离线和批量计算，因为该计算框架强调的是吞吐率而不是计算效率，所以其不能满足大数据快速实时Ad-Hoc查询计算的性能要求。

各大互联网公司和企业也发现了Map-Reduce在进行快速实时Ad-Hoc查询计算时所表现出来的弊端，因此，开源社区和各大互联网公司纷纷进行大数据实时Ad-Hoc查询计算产品的研发，Facebook于2012年秋季开始开发Presto，2013年正式开源。

Presto是Facebook开源的MPP（Massive Parallel Processing）SQL引擎，本身并不存储数据，但是可以接入多种数据源，并且支持跨数据源的级联查询。

Presto被设计为使用MapReduce作业管道（如Hive或Pig）查询HDFS的替代工具，用于对从GB到PB级别的各种大小的数据源运行交互式分析查询，并实现秒级和分钟级响应。它是一个符合ANSI SQL的查询引擎，允许用户将他们喜欢的数据工具（包括BI和ETL工具）与任何底层数据源集成。它本身并不存储数据，而是通过它自身的connector对接数据源拉取数据从而实现查询， 同时由于它可以支持丰富的connector，所以我们可以定义不同的connector来对接不同的数据源，从而实现从不同数据源获取数据实现跨数据源的关联查询。Presto旨在处理数据仓库和分析：数据分析、聚合大量数据并生成报告，即在线分析处理 (OLAP)。

## 概述

Presto是一个OLAP的工具，擅长对海量数据进行复杂的分析；但是对于OLTP场景，并不是presto擅长的，所以不要把presto当作数据库来使用。

Presto是Facebook开源的MPP (Massive Parallel Processing) SQL引擎，其理念来源于一个叫Volcano的并行数据库，该数据库提出了一个并行执行SQL的模型，核心思想就是Operator Model和Iterator Model。

### Operator Model

即**通过各种Operator组成一棵树，树的根节点负责结果输出，树的叶子节点是各种TableScan**。这棵树被称作Plan（执行计划），数据库里又被细分为逻辑执行计划和物理执行计划。这棵树是由SQL经过词法、语法分析及语义分析后，生成一个AST（Abstract Syntax Tree），一般经过Visitor模式遍历后生成。原始数据通过叶子节点TableScan读取数据，然后经过各个Operator的计算，包括（TableScan、Project、Filter、Exchange、Agg、Join、TaskOutput等）产出结果。

### Iterator Model

顾名思义就是一个递归迭代过程，Plan树的各节点有三个状态，Open、GetNext及Close。从根节点Open开始递归调用GetNext获取数据，即父节点递归调用子节点接口直到没有结果为止，然后Close。

## 分类

目前presto分为PrestoDB和PrestoSQL（Trino）两个版本，在2020年12月27日，prestosql与facebook正式分裂，由于版权问题，改名为trino。

### PrestoDB

### PrestoSQL

## 对比

相对于sparksql，presto是纯内存计算，尽量使用整个集群中的内存，sparksql依赖于分配的executor数量，在executor内存不足时会落盘；相对于sparksql，presto启动执行更迅速，sparksql要依赖yarn调度资源；相对于sparksql，presto带有分页功能，在分页查询时更有优势；相对于presto，sparksql运行更稳定，presto会经常出现oom。

druid和kylin更适合预聚合场景，其中druid是时序数据库，对所有维度列枚举值建立bitmap索引，所以维度枚举值如果较多的话，数据膨胀会比较厉害；kylin是通过MapReduce任务枚举维度组合，每一种维度组合对应一张hbase表，维度组合值为主键，通过hbase rowkey实现快速访问，因为实现要指定维度组合情况，所以kylin适合业务场景比较固定的情况，这也是hbase的特点，通过单一的key查询数据。

clickhouse可以对数据分区，建立主键，对命中主键的明细数据聚合查询较快，这种对于一些维度枚举值较多的数据比较适合，没有数据膨胀druid和kylin主要针对预聚合场景，因此查询较快，qps较高，但是因为数据做了聚合，无法查看明细数据，不支持join；clickhouse、presto、sparksql主要是扫描原始数据，实时聚合，因此查询较慢，qps较低，但是clickhouse有主键索引，数据都存在本地，因此查询速度明显快于presto和sparksql，但是presto、sparksql join性能更好。

### Persto VS MySQL

首先，MySQL是一个数据库，具备存储和计算分析能力，而presto只有计算分析能力；其次数据量方面，MySQL作为传统单点关系型数据库不能满足当前大数据量的需求，于是有各种大数据的存储和分析工具产生，presto就是这样一个可以满足大数据量分析计算需求的一个工具。

Presto需要从数据源获取数据来进行运算分析，它可以连接多种数据源，包括Hive、RDBMS（MySQL、Oracle、TiDB等）、Kafka、MongoDB、Redis等。

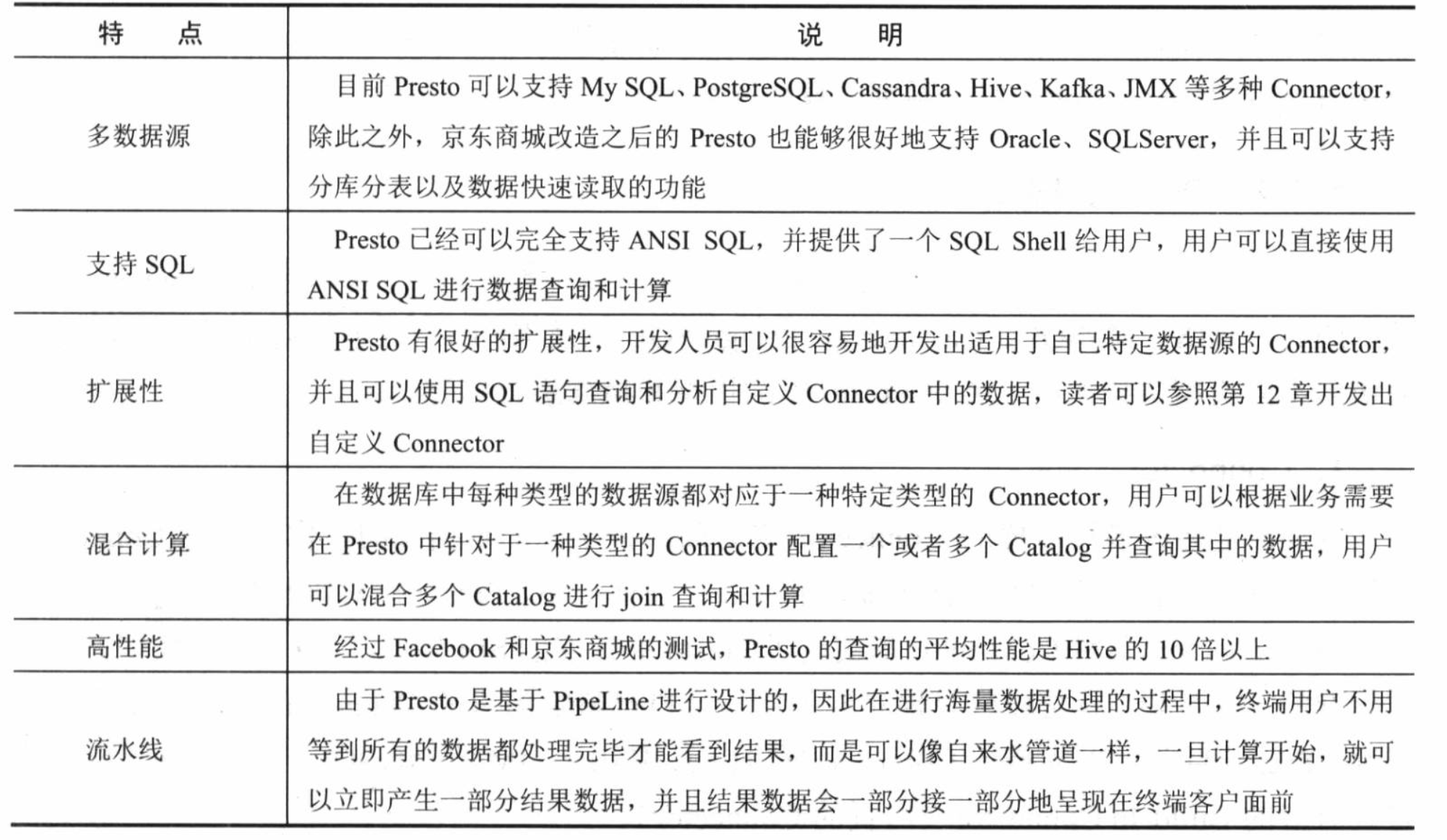
一个presto查询可以将多个数据源的数据进行合并分析。比如select \* from a join b where a.id=b.id;，其中a表可以来自Hive，b表可以来自MySQL。

### Presto VS ClickHouse

## 接入方式

Presto的接入方式有多种Lpresto-cli、pyhive、jdbc、http、golang、SQLAlchemy、PHP等，其中presto-cli是presto官方提供的。

## 特点



### 优点

1、支持多数据源及多数据源之间混合计算

数据源可以是多个不同种类的数据库，PB级海量数据复杂分析，交互式SQL查询。

2、大数据：完全的内存计算，数据量的大小取决于集群内存大小

3、低延迟高并发的内存计算引擎，相对hive，无论是mr还是tez还是spark执行引擎，至少提升10倍以上

4、扩展性：有众多的spi扩展点支持，开发人员可以编写udf、udtf，还可以实现自定义话的connector，实现索引下推，借助外置的索引能力，实现特殊场景下的mpp

5、流水线：presto是基于pipeline进行设计，在大量数据计算过程中，用户无需等到所有数据计算完成才能看到结果，一旦开始计算就可产生一部分结果返回，后续的计算结果以多个page返回给用户

6、索引下推：Presto弃用Hadoop InputFormat，但采用类似的数据分区技术，并且可以把SQL经过解析后，把Where条件生成TupleDomain传递给Connector。Connector能根据字段元数据采用一定程度的索引下推，利用底层系统的索引能力，大大减少数据扫描区间和参与计算的数据量。

### 缺点

不擅长多个大表的JOIN操作，因为presto是基于内存的，多个大表JOIN计算会有多次内存读写，速度会变慢（类似于MySQL的innodb\_buffer\_pool\_size）。

## 应用场景

presto应用场景：

1、实时计算：实时查询工具上的重要选择

2、ad-hos查询：数据分析应用、presto根据特定条件查询返回结果和生成报表

3、etl：支持广泛的数据源，可用于不同数据库之间迁移、转换和清洗能力

4、实时数据流分析：presto-kafka connector使用sql对kafka的数据进行清洗、分析

5、作为MPP：Presto Connector 有非常好的扩展性，可进行扩展开发，可支持其他异构非SQL查询引擎转为SQL，支持索引下推

### ETL

### 实时数据计算

### Ad-Hoc查询

Ad-hoc查询就是即席查询，即席查询允许用户根据自己的需求随时调整和算则查询条件，计算平台或者系统能够根据用户的查询条件返回查询结果或者生成相应的报表。因此可见，即席查询和普通应用查询的最大不同是：普通的应用查询时定制开发的，其查询语句是固定或者限制在一定的变动范围之内的；而即席查询允许用户随意指定或者改变查询语句或者查询条件。由于普通的应用查询都是定制开发的，其查询语句几乎是固定的，因此，在系统实施时就可以通过建立索引或者分区等技术来优化这些查询，从而提高查询效率。但是即席查询是用户在使用时临时差生的、系统无法预知的，因此也无法对这些查询进行有针对性的优化和改进。

### 实时数据流分析

实时数据流分析主要是指通过presto-kafka使用SQL语句对Kafka中的数据流进行清洗、分析和计算。

# 安装使用

## 部署

**Linux环境编译：**

1. 配置JDK：

yum install jdk

查看安装位置以及配置环境变量：

which java

1. 安装Maven：

yum install maven

修改系统环境变量（/etc/profile）：

export MAVEN\_HOME=/usr/local/maven/apache-maven-3.6.3

export PATH=$MAVEN\_HOME/bin:$PATH

加载环境变量：

source /etc/profile

验证：

mvn -v

修改Maven镜像库：<https://www.csdn.net/tags/OtDaYg3sMzg0NS1ibG9n.html>

修改项目pom.xml镜像库配置：

<https://blog.csdn.net/JosephThatwho/article/details/122732215>

增加内容如下：

<repositories>

<repository>

<id>central</id>

<url>https://maven.aliyun.com/repository/public</url>

</repository>

</repositories>

<pluginRepositories>

<pluginRepository>

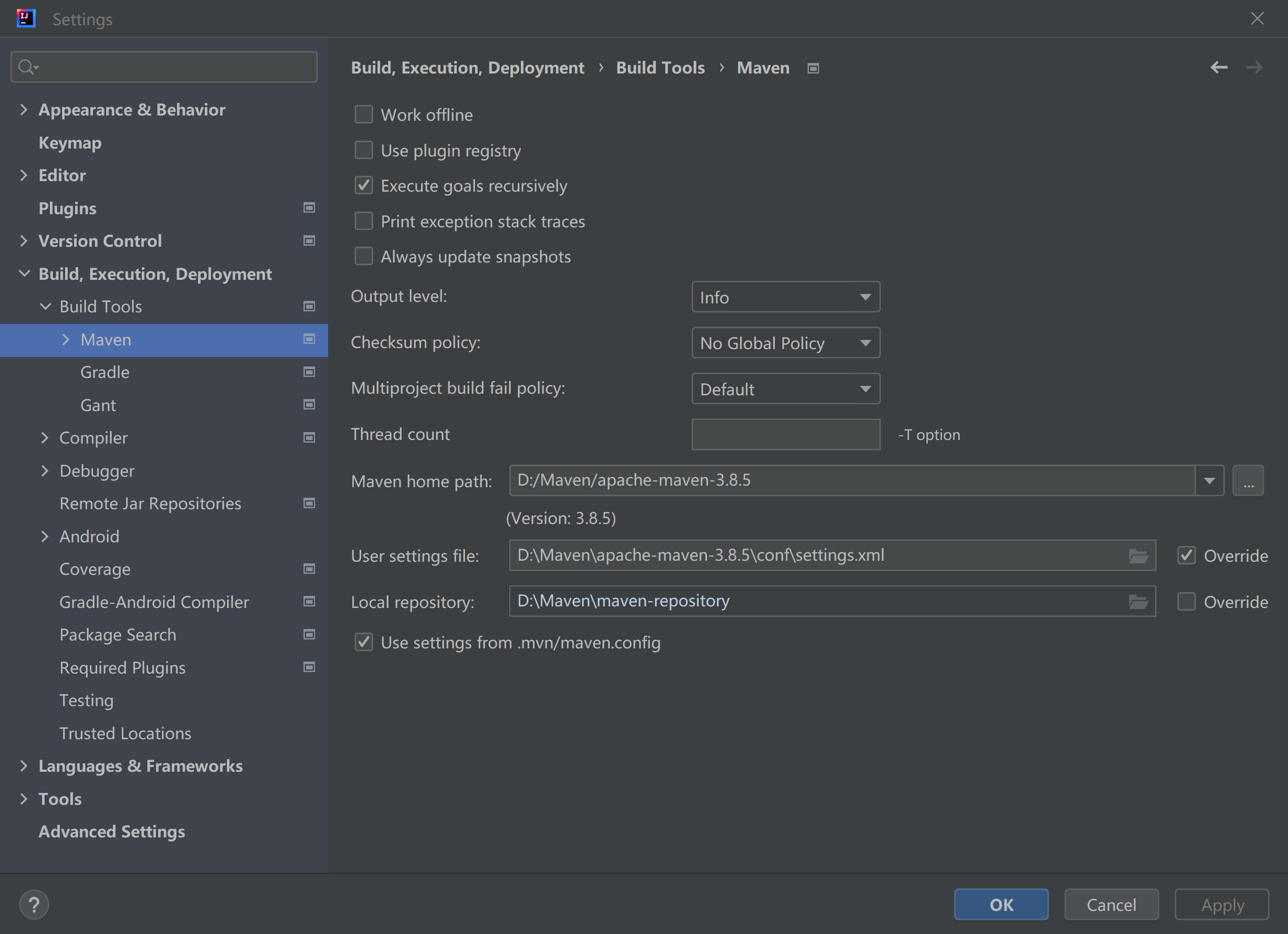
<id>central</id>

<url>https://maven.aliyun.com/repository/public</url>

</pluginRepository>

</pluginRepositories>

在Idea中File-settings中配置Maven：



1. 下载Presto：

Trino：git clone <https://github.com/yedushusheng/trino.git>

Presto：

1. 编译Presto

Window环境编译：

## 使用

### Presto CLI

### Presto JDBC驱动

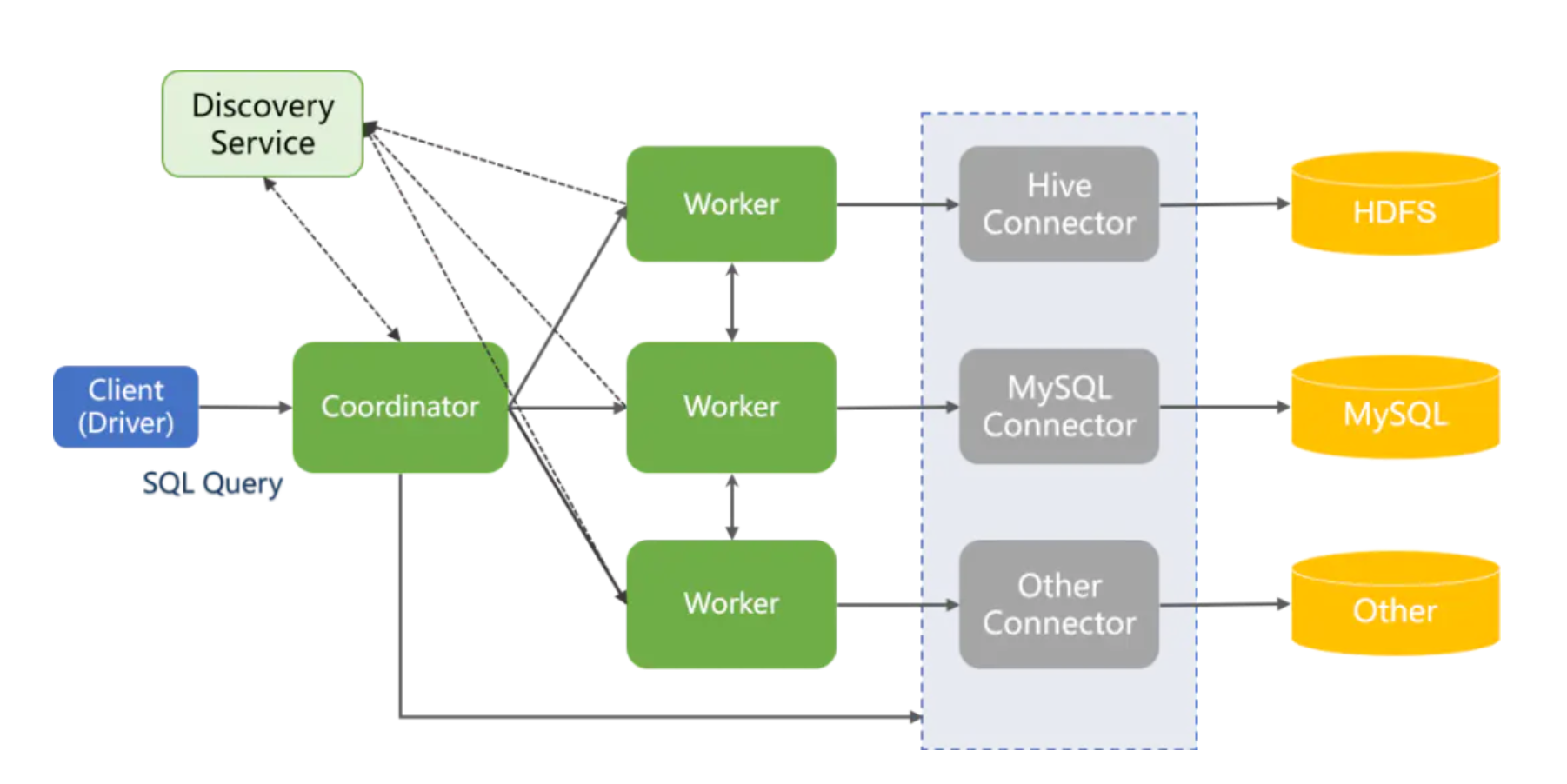
### Presto与ODBC

### 客户端库

### Presto Web UI

### 使用Presto执行SQL

# 架构



1、client：包含presto-cli客户端和jdbc驱动、odbc及其他语言实现的driver

2、discovery service：是将coordinator和work 结合到一起的服务。worker节点启动后向discovery server服务注册，coordinator从discovery server获得正常工作的worker节点

3、coordinator：主要用于接收客户端提交的查询，解析sql语句，生成查询执行计划，并生成stage和task进行调动，然后合并结果，把结果返回给客户端

4、worker：主要负责与数据的读写交互以及执行查询计划

**presto sql执行步骤：**

1、客户端通过http发送一个查询语句给presto集群的coordinator

2、coordinator接收到客户端的查询语句，对语句进行解析，生成查询执行计划，并根据生成的执行计划生成stage和task，并将task分发到需要处理数据的worker上进行分析

3、worker执行task，task通过connector从数据源中读取需要的数据

4、上游stage输出的结果给到下游stage作为输入，每个Stage的每个task在worker内存中进行计算和处理

5、client从提交查询后，就一直监听coordinator中的查询结果，一有结果就立即输出，直到轮训所有的结果都返回则本次查询结果结束

## 硬件架构

## 软件架构

# 原理

参考：

<https://jishuin.proginn.com/p/763bfbd62b9d>

<https://zhuanlan.zhihu.com/p/260399749>

<https://blog.csdn.net/weixin_42412601/article/details/112595357>

<https://wenku.baidu.com/view/b432f6f55322aaea998fcc22bcd126fff7055dfa.html>

<https://zhuanlan.zhihu.com/p/567510806?utm_id=0>

## Presto服务进程

Presto集群中一共有两种服务器进程：Coordinator服务进程和Worker服务进程，其中Coordinator服务进程的主要作用是：接收查询请求、解析查询语句、生成查询执行计划、任务调度和Worker管理。而Wroker服务进程则执行被分解后的查询执行任务：Task。

### Coordinator

参考：<https://www.infoq.cn/article/VNe0A9yKszPCmp32akCa>

<https://mayunlei.github.io/2020/08/16/深入理解Presto-Presto的内部架构/>

Coordinator 负责接受用户提交的查询请求，对这些请求进行分析，并调度到不同的 worker 节点上执行。

Coordinator的作用是：

从用户获得SQL语句。

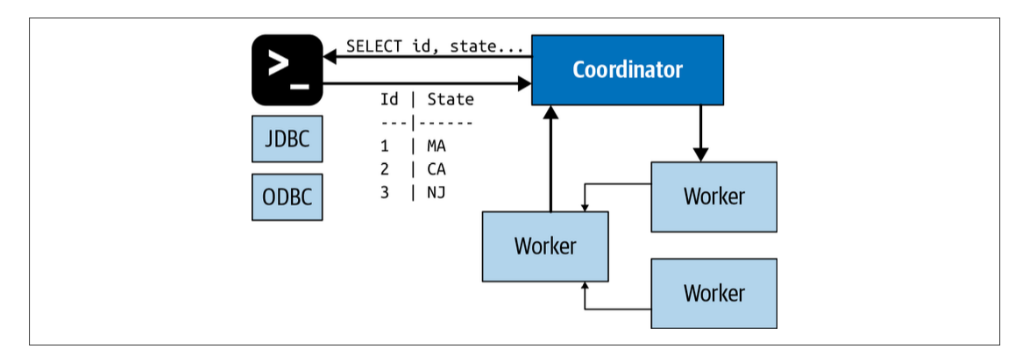
解析SQL语句。

规划查询的执行计划。

管理worker节点状态。

Coordinator是Presto集群的大脑，并且是负责和客户端沟通。用户通过PrestoCLI、JDBC、ODBC驱动、其他语言工具库等工具和coordinator进行交互。Coordinator从客户端接受SQL语句，例如select语句，才能进行计算。

每个Presto集群必须有一个coordinator，可以有一个或多个worker。在开发和测试环境中，一个Presto进程可以同时配置成两种角色。



Coordinator追踪每个worker上的活动，并且协调查询的执行过程。Coordinator给查询创建了一个包含多阶段的逻辑模型，一旦接受了SQL语句，Coordinator就负责解析、分析、规划、调度查询在多个worker节点上的执行过程，语句被翻译成一系列的任务，跑在多个worker节点上。worker一边处理数据，结果会被coordinator拿走并且放到output缓存区上，暴露给客户端。一旦输出缓冲区被客户完全读取，coordinator会代表客户端向worker读取更多数据（pipeline实质）。worker节点，和数据源打交道，从数据源获取数据。因此，客户端源源不断的读取数据，数据源源源不断的提供数据，直到查询执行结束。

Coordinator通过基于HTTP的协议和worker、客户端之间进行通信。

### Woker

## 数据模型

### Connector

Presto是通过多种多样的Connector来访问多种不同的数据源的。你可以将Connector当作Presto访问各种不同数据源的驱动程序。一般情况下，Presto针对每种数据源都有对应的Connector。每种Connector都实现了Presto中标准的SPI接口，因此只要你实现Presto中标准的SPI接口，就可以轻易地实现使用适合自己特定需求的Connector来访问特定的数据源。Presto目前可以支持的Connector有Hive、Kafka、JMX、MySQL、Cassandra等，都有其对应的Build-In Connector（内置的Connector）。

### 表结构

在Presto中我们定位一张数据表的完整路径为：catalog.schema.table。

Presto使用Catalog、Schema和Table这3层结构来管理数据。

Presto中的Catalog类似于MySQL中的一个数据库实例。而Schema就类似于MySQL中的一个Database。通过使用特定的Connector访问Catalog中指定的数据源，一个Catalog中可以包含多个Schema。

#### Catalog

Catalog：就是数据源。Hive是数据源，MySQL也是数据源，Hive和MySQL都是数据源类型，可以连接多个Hive和多个MySQL，每个连接都有一个名字。一个Catalog可以包含多个Schema，大家可以通过show catalogs命令看到presto连接的所有数据源。

#### Schema

Schema：相当于一个数据库实例，一个Schema包含多张数据表。Show schemas from ‘catalog\_name’可以列出catalog\_name下面的所有schema。

#### Table

Table：数据表，与一般意义上的数据库表相同。Show tables from ‘catalog\_name.schema\_name’可以查看catalog\_name.schema\_name下的所有表。

在presto中定位一张表，一般是catalog为根，例如：一张表的全称为hive.test\_data.test，标识hive（catalog）下的test\_date（schema）中的test表。可以理解为：数据源的大类.数据库.数据表。

### 存储单元

#### Page

#### Block

presto的存储单元包括：

1、Page：多行数据的集合，包含多个列的数据，内部仅提供逻辑行，实际以列式存储；

2、Block：一列数据，根据不同类型的数据，通常采取不同的编码方式，了解这些编码方式，有助于自己的存储系统对接presto。

**不同类型的block：**

1、array类型block，应用于固定宽度的类型，例如int，long，double。block由两部分组成。

boolean valueIsNull[]表示每一行是否有值。

T values[] 每一行的具体值。

2、可变宽度的block，应用于string类数据，由三部分信息组成。

Slice：所有行的数据拼接起来的字符串。

int offsets[] :每一行数据的起始偏移位置。每一行的长度等于下一行的起始便宜减去当前行的起始偏移。

boolean valueIsNull[] 表示某一行是否有值。如果有某一行无值，那么这一行的便宜量等于上一行的偏移量。

3、固定宽度的string类型的block，所有行的数据拼接成一长串Slice，每一行的长度固定。

4、字典block：对于某些列，distinct值较少，适合使用字典保存。主要有两部分组成：

字典，可以是任意一种类型的block(甚至可以嵌套一个字典block)，block中的每一行按照顺序排序编号。

int ids[]表示每一行数据对应的value在字典中的编号。在查找时，首先找到某一行的id，然后到字典中获取真实的值。

## 查询模型

Presto是一个分布式SQL查询引擎，在执行一条SQL语句时，这些SQL会被解析成对应的Task分发到不同的Worker机器上进行执行。

在Presto中，当一条SQL语句被提交到集群时会被转换成为一个可以由Presto解析执行的查询执行（Query）和相关的执行计划，一个 Query会被拆分成多个具有层级关系的Stage，每个Stage在逻辑上会被分成多个Task真正提交到Worker上进行计算，同样的，每个Task会划分成多个Driver，从而行的执行一个Task。而一个Task会分为多个Driver，一个Driver包含一系列Operator处理一个Split，一个Operator代表一个操作，而一个Split代表一个数据表的子集。

### Statement

在Presto中Statement就是我们输入的SQL语句。

### Query

在Presto中Query表示查询执行。当接收到一个SQL语句并执行的时候，Presto会将该SQL语句进行解析，将其转换成一个查询执行和相关的查询计划。一个查询执行代表可以在Presto集群执行的查询，它由多个Stage组成。

### Stage

MPP的理念就是能尽量细粒度的将SQL并行执行，以一个SQL 2个表JOIN后Agg为例，那么每个表都可以单独并行执行去Scan数据（互不影响），然后进行Join和Agg。所以执行计划（Plan）将执行PlanFragment，即将一个树分块变为各个子树，每个子树可以并行的在多台机器上执行，这个Fragment被称为 Stage。

在Presto中Stage表示查询执行阶段。当Presto运行一个Query时，会将其拆分成多个Stage，一个Stage代表Query的一部分。

其中Stage被分为4类：

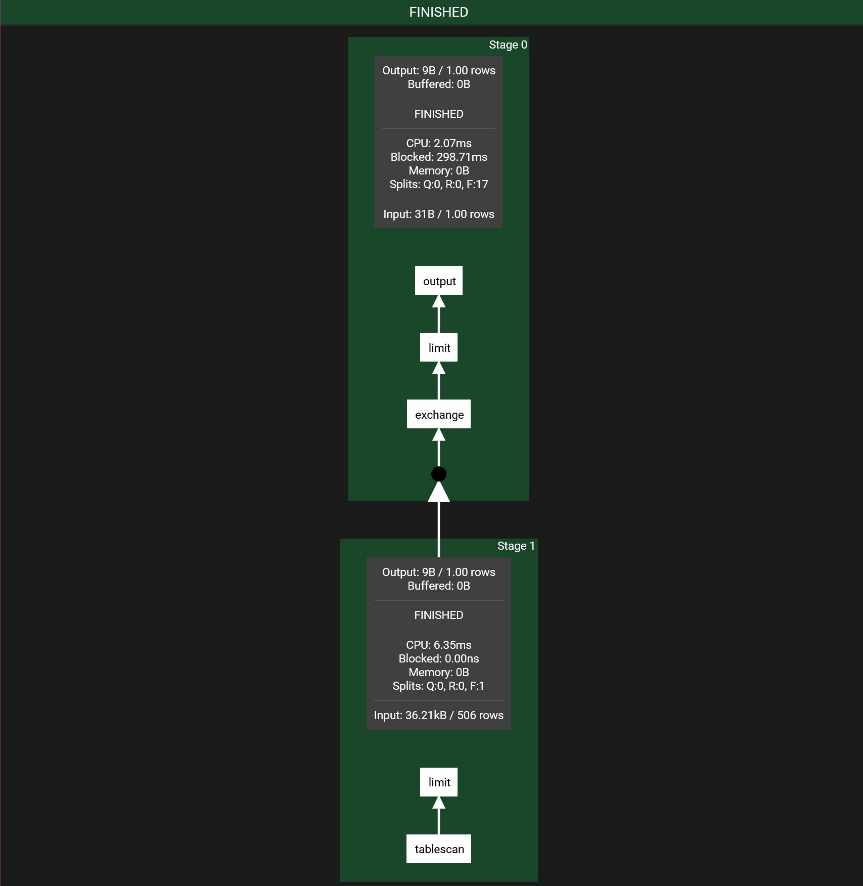
Coordinator\_Only：用于执行DDL（Data Definition Language）或DML（Data Manipulation Language）语句；

Single：用于**聚合子Stage数据**并返回给Coordinator，并由Coordinator返回给用户，比如每个查询中的根节点（Root Stage）；

Fixed：**接收子Stage数据并在集群中对这些数据进行分布式计算**；

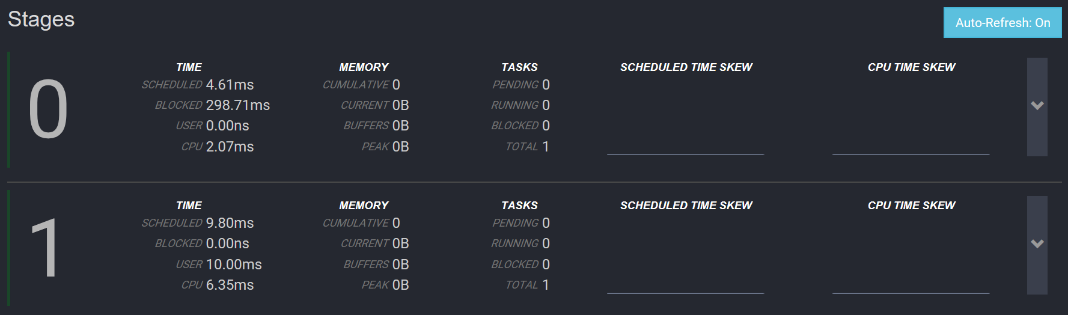
Source：直接连接数据源，从数据源读取数据，该阶段会根据执行计划进行相关的数据过滤以减少读取的数据量。

我们以简单的SQL查询为例，SQL为select id from table limit 1; 这个SQL简单来说，就干了2件事，一是Scan数据，另外是Limit，而这2件事，可以并行执行，所以如图所示，其分为2个Stage：



Stage 1为Scan数据和Limit，这里Limit是下推优化。Stage 0为最终结果输出。

同时Presto UI里可以看到每个Stage详细信息，以及每个Stage需要的Task数（可以认为Worker数），如图所示：



### Exchange

连接不同的Stage，用于不同Stage之间的数据交互。数据的交互有一些Operator实现，比如数据是Hash分发还是完全Replicate等。从上图可以看到Stage 1和Stage 0需要交互，通过Exchange实现。

在Presto中Exchange用来连接Stage，完成不同Stage之间的数据交换。

其中Exchange被分为两类：

Output Buffer : 用于将数据传送给下游Stage；

Exchange Client：用于从上游Stage接收数据。

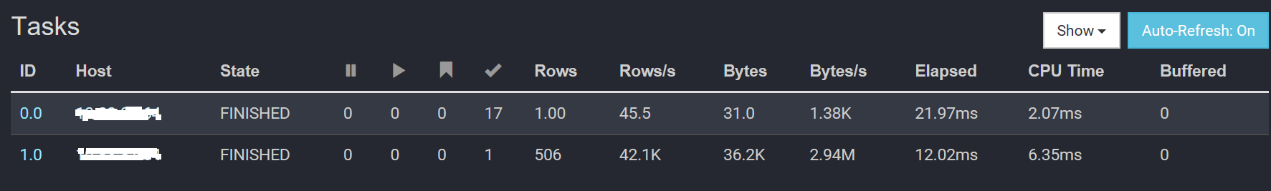
对于Source类型的Stage，该Stage使用Source Operator直接从数据源读取数据。

### Task

在Presto中，Stage被逻辑上分为一系列的Task，这些Task则是实际运行在Presto的各个Worker节点上。

Stage有多个Task组成。Stage并不会运行，其实个抽象的概念，其只是负责管理Task和封装建模。**Stage实际运行的是Task**。每个Task处理一个或者多个Split，每个Task被分配到每台机器上执行。每个Task都有对应的输入和输出。同一个Stage下的Task是个并行的概念，做的事情是相同的。

如下图所示，我们可以看到每个Tasks的相信信息，其中0.x表示Stage 0，1.x表示Stage 1，同时我们也可以看到每个Task执行花费的时间，读取的数据大小以及每个Task处理的Split数目。一个Stage包含一个或多个Task，每个Task做的事情是一样的，所以每个Stage的花费时间由最慢的Task决定，比如Scan HDFS数据，可能会因为某些Data Node阻塞导致Task阻塞。



### Driver

在Presto中，一个Task包含一个或多个Driver，一个Driver其实是作用于一个Split的一系列Operator集合。因此一个Dirver处理一个Split，并输出相应的数据，并由Task收集输出到下游Stage的一个Task。一个Driver拥有一个输入个一个输出。

### Operator

在Presto中，一个Operator代表一个Split的一种操作，并进行相应的操作，然后输出数据。每个Operator会以Page为最小单位分别读取和输出数据。每次只会读取及输出一个Page。

### Split

在Presto中，Split表示一个大数据集中的一个小的子集。当Presto执行一个查询的时候，首先会从Coordinator得到一个数据表对应的所以的Split，然后根据查询执行计划选择节点执行Task处理对应的Split。

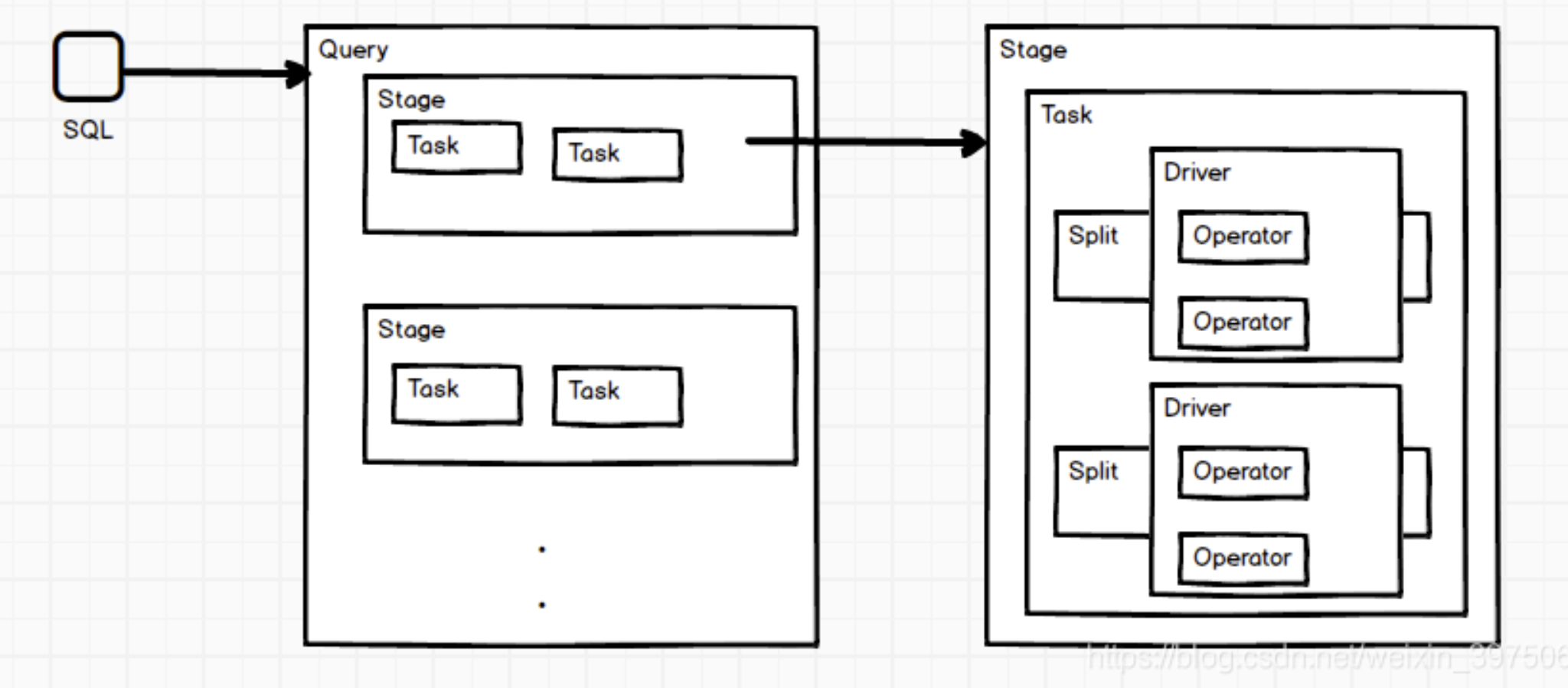
由于Driver是处理一个Split的一系列操作的集合，而一个Task包含多个Driver，所以一个Task可以操作多个Split。

### Page

在Presto中，Page是处理数据的最小单元，一个Page对象包含多个Block对象，每个Block对象是一个字节数组，存储一个字段的若干行。多个Block横切的一行就是真实的一行数据。

一个Page最大1MB，最多16\*1024行数据。

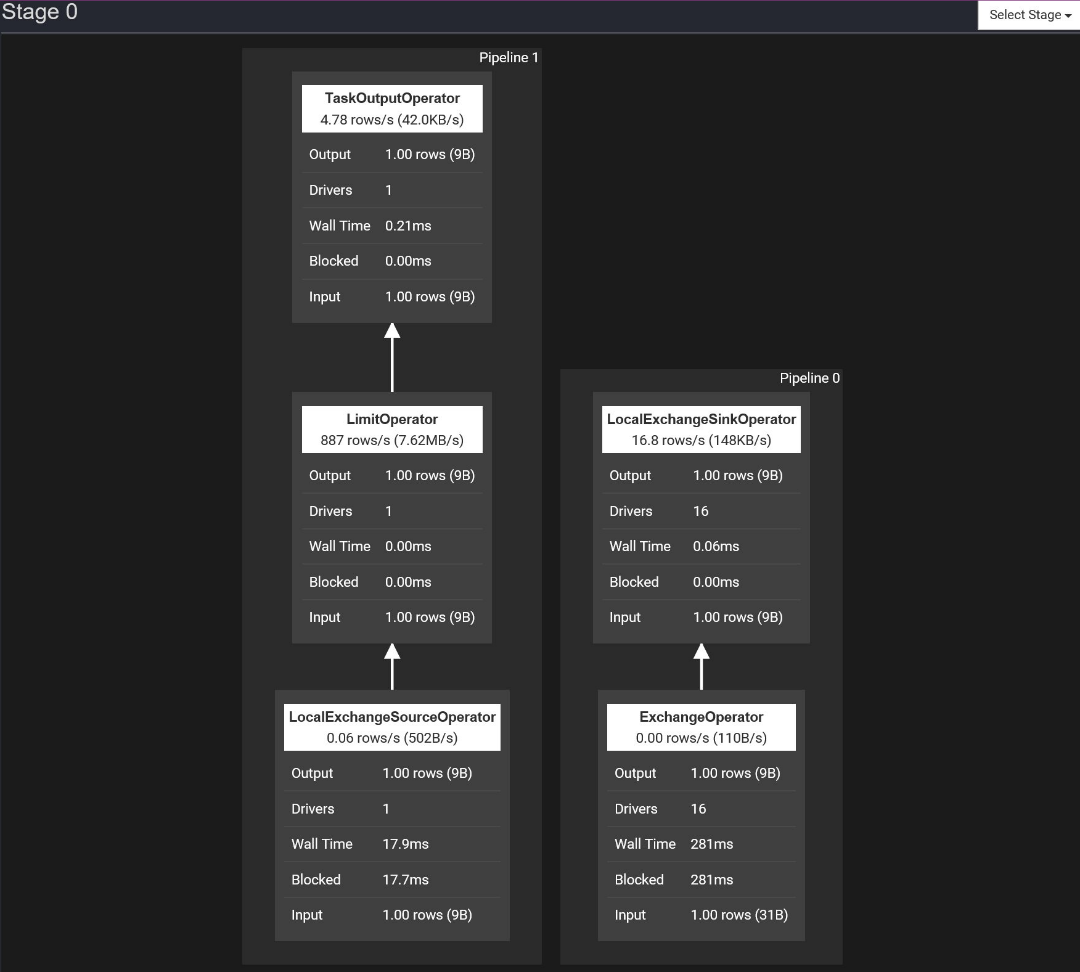
Presto查询模型图示大致如下：



### Pipeline

Stage里有很多Operator，这些Operator可能并行度是不一样的，比如Scan数据并行就很大，但是最后聚合数据，并行一般为1。所以PlanFragment又会被切分为若干Pipeline，每个Pipeline由一组Operator组成，这些Operator被设置同样的并行度。Pipeline之间会通过LocalExchangeOperator来传递数据。

在Presto UI上我们可以看到Pipeline信息，如下图所示，Stage 0主要是将Exchange的数据，做最后的limit，所以其可以细分为2个步骤，LocalExchangeOperator及LimitOperator，这2个动作的并行度是不一样的，Exchange 可以多个线程去做，而Limit只需要一个线程。从图中我们可以看到Driver和Operator信息，其中Driver的数目就是这个Pipeline的并行度。



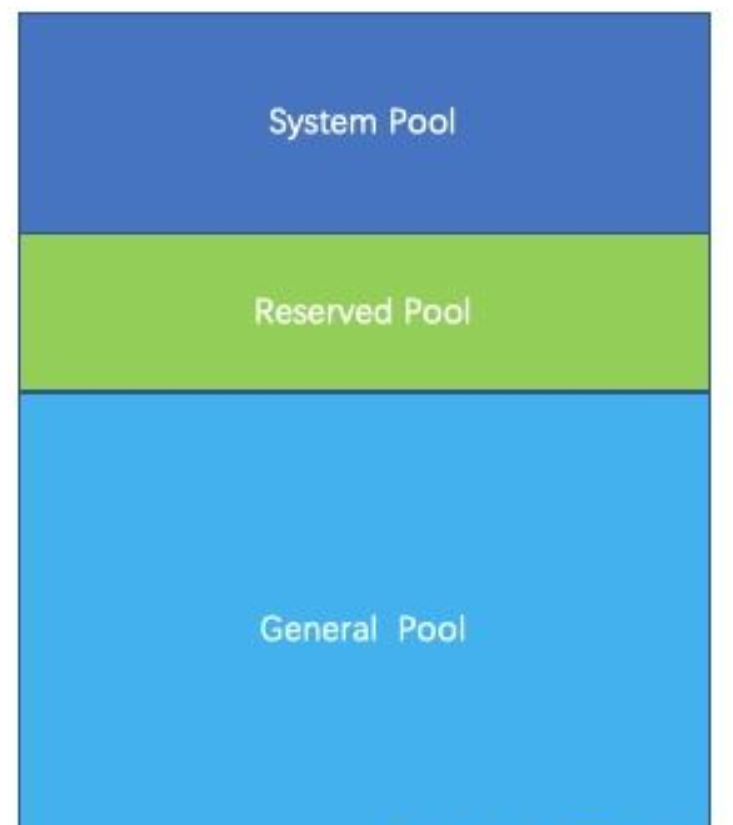
## 内存管理

Presto是一款内存计算型的引擎，所以对于内存管理必须做到精细，才能保证query有序、顺利的执行，部分发生饿死、死锁等情况。

### 内存池

Presto采用逻辑的内存池，来管理不同类型的内存需求。

Presto把整个内存划分成三个内存池，分别是System Pool ,Reserved Pool, General Pool。



1、System Pool 是用来保留给系统使用的，默认为40%的内存空间留给系统使用。

2、Reserved Pool和General Pool 是用来分配query运行时内存的。

3、其中大部分的query使用general Pool。 而最大的一个query，使用Reserved Pool， 所以Reserved Pool的空间等同于一个query在一个机器上运行使用的最大空间大小，默认是10%的空间。

4、General则享有除了System Pool和General Pool之外的其他内存空间。

**为什么要使用内存池？**

System Pool用于系统使用的内存，例如机器之间传递数据，在内存中会维护buffer，这部分内存挂载system名下。

那么，为什么需要保留区内存呢？并且保留区内存正好等于query在机器上使用的最大内存？

如果没有Reserved Pool， 那么当query非常多，并且把内存空间几乎快要占完的时候，某一个内存消耗比较大的query开始运行。但是这时候已经没有内存空间可供这个query运行了，这个query一直处于挂起状态，等待可用的内存。 但是其他的小内存query跑完后，又有新的小内存query加进来。由于小内存query占用内存小，很容易找到可用内存。 这种情况下，大内存query就一直挂起直到饿死。

所以为了防止出现这种饿死的情况，必须预留出来一块空间，共大内存query运行。 预留的空间大小等于query允许使用的最大内存。Presto每秒钟，挑出来一个内存占用最大的query，允许它使用reserved pool，避免一直没有可用内存供该query运行。

### 内存管理



Presto内存管理，分两部分：

1、query内存管理：

query划分成很多task， 每个task会有一个线程循环获取task的状态，包括task所用内存。汇总成query所用内存。

如果query的汇总内存超过一定大小，则强制终止该query。

2、机器内存管理

coordinator有一个线程，定时的轮训每台机器，查看当前的机器内存状态。

当query内存和机器内存汇总之后，coordinator会挑选出一个内存使用最大的query，分配给Reserved Pool。

内存管理是由coordinator来管理的， coordinator每秒钟做一次判断，指定某个query在所有的机器上都能使用reserved 内存。那么问题来了，如果某台机器上，没有运行该query，那岂不是该机器预留的内存浪费了？为什么不在单台机器上挑出来一个最大的task执行。原因还是死锁，假如query，在其他机器上享有reserved内存，很快执行结束。但是在某一台机器上不是最大的task，一直得不到运行，导致该query无法结束。

# PrestoRESTful框架

在Presto中，几乎所有的操作都需要依赖于基于AirLift框架构建的RESTful服务来完成，包括Worker节点的管理、查询语句的提交、查询执行状态的显示、各个Task之间数据的传递等。因此Presot中的RESTful服务是Presto集群的基础。

Presto总共提供了4类RESTful接口，分别是Statement接口、Query接口、Stage接口和Task接口。

## Statement服务接口

与SQL语句相关的请求均由该服务接口处理，包括接受提交的SQL语句、获取查询执行的结果、取消查询等。

## Query服务接口

与查询执行相关的RESTful请求均由Query服务接口处理，包括SQL语句的提交、获取查询执行的结果、查询取消等。

## Stage服务接口

与Stage相关的RESTful请求均由Stage服务接口处理，其实该接口只提供了一个功能，就是取消获取结束一个指定的Stage。

## Task服务接口

与Task相关的RESTful请求均由Task服务接口处理，包括Task的创建、更新、状态查询和结果查询等。

## 用户执行查询相关Restful接口

用户执行查询相关的Restful接口，在类QueuedStatementResource和ExecutingStatementResource中实现。

从用户的角度看来，一个数据库查询在结束前主要经历两个状态，Queued（已提交，排队中）和 Executing（执行中）。对于处于不同状态的查询，Presto 把相关的 Restful 接口放在两个类中：

QueuedStatementResource 类负责处理处于Queued状态的查询，主要接口有：

* tStatement（POST /v1/statement）：提交一个新的查询，若成功，会返回新查询的id，nextUri和新查询的状态等。其中，nextUri可以直接用于下面的getStatus和cancelQuery接口；
* tStatus（GET /v1/statement/queued/{queryId}/{slug}/{token}）：获取查询当前的状态，返回的响应中包含一个nextUri。根据查询的当前状态，这个nextUri的路径以/v1/statement/queued或者/v1/statement/executing为前缀。当以/v1/statement/queued为前缀时，可以用于重复调用getStatus接口或cancelQuery接口。当以/v1/statement/executing为前缀时，可以用于调用ExecutingStatementResource的相关接口；
* ncelQuery（DELETE /v1/statement/queued/{queryId}/{slug}/{token}）：取消已经提交的查询；

ExecutingStatementResource类负责处理处于Executing状态的查询，主要接口有：

* tQueryResults（GET /v1/statement/executing/{queryId}/{slug}/{token}）：获取查询结果，响应中，主要的元素有nextUri（用于进一步调用getQueryResults接口或cancelQuery接口），partialCancelUri（用于取消已经部分执行的查询，即下面的cancelPartial 接口），columns（结果的列信息），data（查询结果数据）；
* ncelQuery（DELETE /v1/statement/executing/{queryId}/{slug}/{token}）：用于取消已经处于Executing的查询；
* tialCancel（DELETE /v1/statement/partialCancel/{queryId}/{stage}/{slug}/{token}）：用于取消已经部分执行的查询；

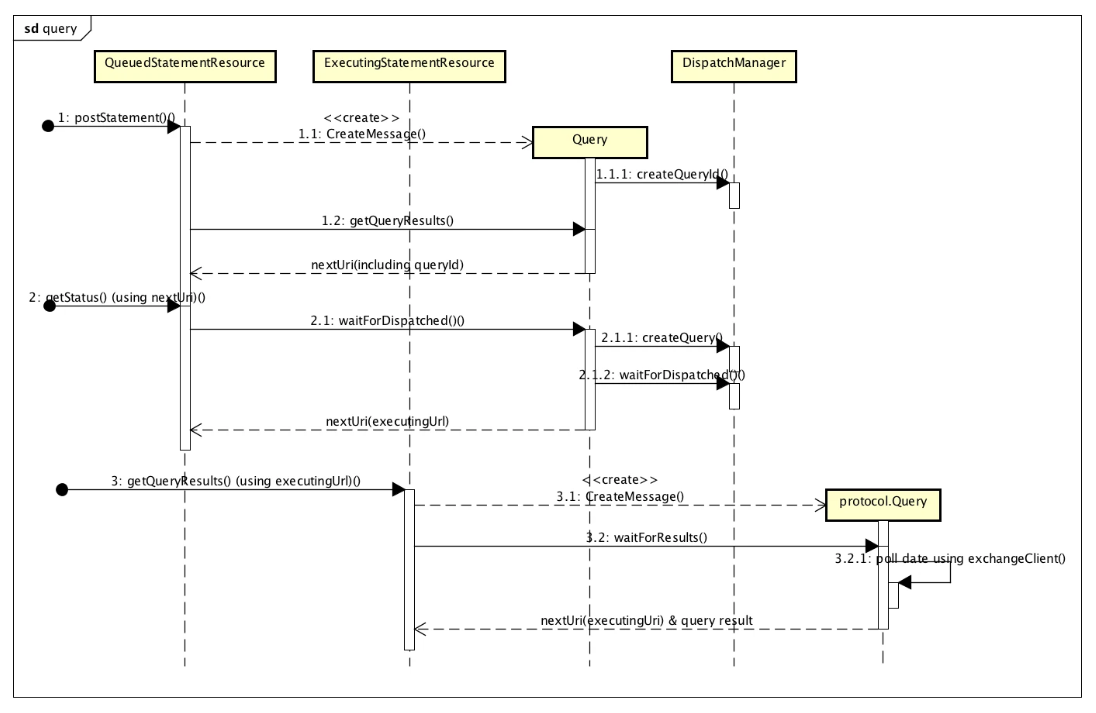
## 小结

Presto集群中的数据传输、节点通信、心跳感应、计算监控、计算调度和计算分布全部都是基于RESTful服务实现的，因此Presto中的RESTful服务就是Presto所有服务的基石。

# 查询总体流程

参考：<https://www.infoq.cn/article/VNe0A9yKszPCmp32akCa>

基于Restful 接口，一个查询的总体执行流程如图所示，分为以下几个步骤：



1、用户通过postStatement接口提交一个查询请求，QueuedStatementResource 接受到该请求后，创建一个Query对象维护该请求的状态，Query对象的创建过程中，会请求DispatchManager分配一个全局唯一的queryId；

2、Query创建后，QueuedStatementResource立即请求它的getQueryResults方法，此时由于DispatchManager仅仅只是为该查询分配了ID，而尚未进行分发，因此会立即返回一个nextUri（queuedUri），这个nextUri指向的其实就是 QueuedStatementResource的getStatus接口；

3、用户通过上一步的nextUri调用getStatus接口之后，实际上调用的是Query 对象的waitForDispatched接口，该接口则请求DispatchManager新建一个查询，并等待查询被分发，一旦分发完成，则返回一个nextUri（executingUri），这个nextUri指向的其实就是ExecutingStatementResource的getQueryResults接口；

4、一旦getQueryResults接口被调用，ExecutingStatementResource将创建一个protocol.Query对象以维护查询状态，并异步调用该对象的waitForResults以获取查询结果，当查询结果未就绪或者未全部返回，getQueryResults仍然会返回一个nextUri（executingUri），用户可以通过这个nextUri循环获取所有结果数据。

# 提交查询

## 提交查询的步骤

Presto客户端对查询语句的提交主要分为以下3个步骤：

1. 从指定的文件、命令行参数或者Cli窗口中获取需要执行的SQL语句；
2. 将得到的SQL语句组装成一个RESTful请求，发送给Coordinator，并处理返回的response；
3. Cli会不停地循环**分批**读取查询结果并在屏幕进行动态显示，直到查询结果完全显示完毕。

说明：正是这里的分批读取，使得虽然后端是按照一批一批处理结果并返回部分结果的处理方式，在客户端看来没有明显的感知。

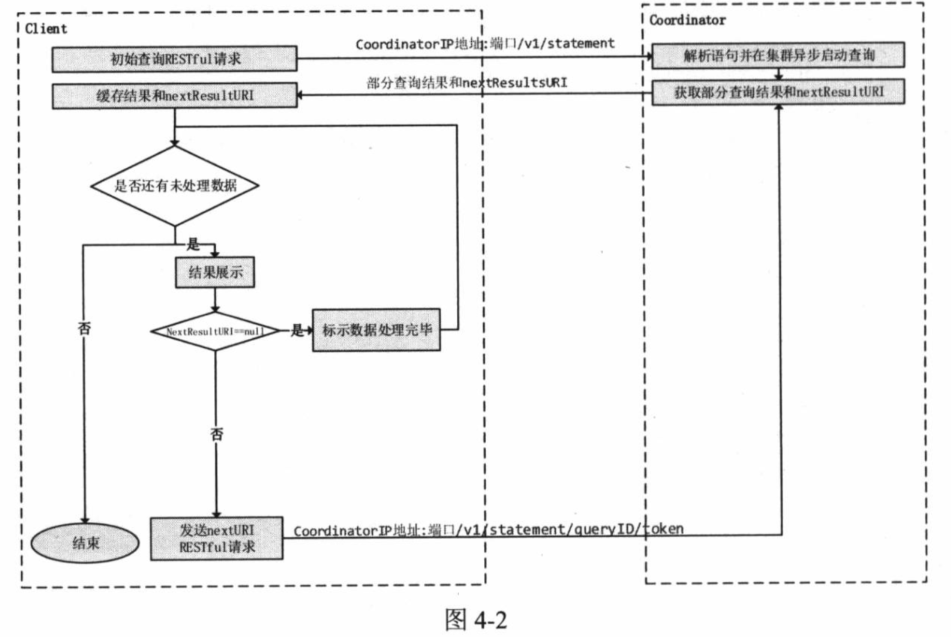
## 源码解析

## 提交查询的流程

提交查询的流程如下：

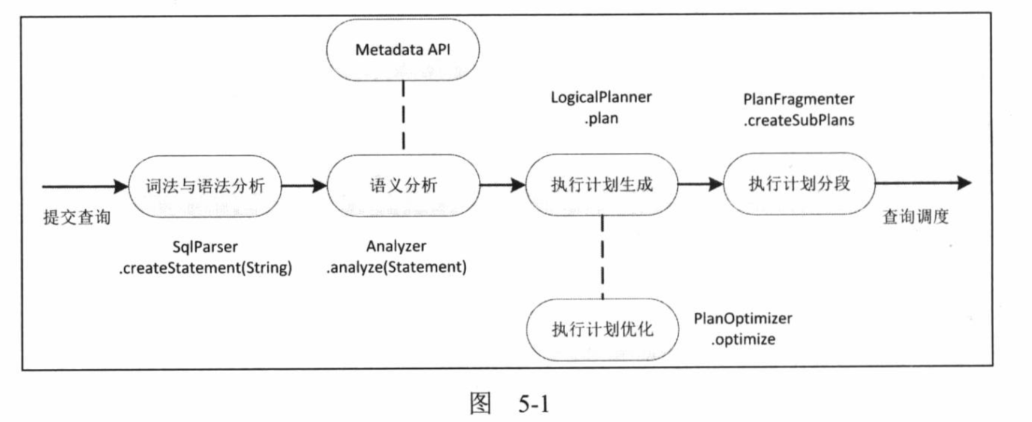
1. 生成查询的初始RESTful请求，并发送给Coordinator；
2. Coordinator处理RESTful请求，执行查询，并将部分查询结果和用于Client获取下一批查询结果的RESTful请求地址nextResultsURI一起返回给客户端；
3. Client端获得Coordinator返回的部分查询结果并进行显示，若所有查询结果均已经获取完毕，则直接退出；否则就向Coordinator发送URI为nextResultsURI的RESTful请求获取下一批查询执行结果；
4. 反复执行步骤3，直至所有查询结果均已获取、处理完毕。

查询流程图如图4-2所示：



# 生成查询执行计划

生成执行计划的主要流程如图所示：



可以看到，从提交查询到执行计划生成包含了词法与语法分析、语义分析、执行计划生成、执行计划优化、执行计划分段等部分。

## 基本概念

### Node

查询语句经过词法与语法分析之后，会生成抽象语法树（AST），该语法树中的每一个节点都是一个Node（SQL语句的一部分，例如Select部分、Where部分、Group by部分等），Node是一个抽象类。

### Metadata API

Metadata API即Metadata接口，其提供了对元数据进行各种操作的接口，例如列出所有的库名（listSchemaNames）、表名（listTables）等。这些接口在对SQL进行语义分析以及某些DDL操作（如create table）的执行过程中会用到。

Metadata API将不同Connector对其元数据的各种操作抽象成统一的接口，使得在使用这些接口时无需考虑具体的底层Connector实现。

Metadata API除了提供对元数据操作的接口，还提供了一些通用的与Connector无关的方法，例如列出Presto支持的自定义函数（listFunctions）等。

## 词语与语法分析

### 语法规则

Presto使用ANTLR4编写SQL语法，语法规则的定义在presto-parser项目的SqlBase.g4文件。

### 词法分析

### 语法分析

Presto使用Visitor模式对SQL语句进行语法分析。

## 获取QueryExecution

CLI提交SQL之后，查询请求会被封装成一个SqlQueryExecution对象提交给 Coordinator去执行。其中在 SqlQueryExecution#startExecution#analyzeQuery里，会生成逻辑执行计划，之后会被优化成一个优化后的逻辑执行计划，代码如下：

private void startExecution()

{

// analyze query

PlanRoot plan = analyzeQuery();

metadata.beginQuery(getSession(), plan.getConnectors());

// plan distribution of query

planDistribution(plan);

// transition to starting

if (!stateMachine.transitionToStarting()) {

// query already started or finished

return;

}

// if query is not finished, start the scheduler, otherwise cancel it

SqlQueryScheduler scheduler = queryScheduler.get();

if (!stateMachine.isDone()) {

scheduler.start();

}

}

下面的过程在 analyzeQuery 中。

// 构造逻辑执行计划

LogicalPlanner logicalPlanner = new LogicalPlanner(

stateMachine.getSession(),

planOptimizers,

idAllocator,

metadata,

sqlParser,

statsCalculator, // 用于获取或计算代价模型估算所需要的统计信息

costCalculator, // 调用代价模型估算计划代价

stateMachine.getWarningCollector());

// 优化后的执行计划

Plan plan = logicalPlanner.plan(analysis);

LogicalPlanner 负责整个SQL语句执行计划的生成，根据SQL语句类型的不同，生成不同的执行计划，然后针对生成的执行计划，分别使用注册进来的优化器进行优化，生成优化好的逻辑执行计划。

其中在 logicalPlanner.plan里面，会进行逻辑执行计划的优化。通过调试可以看见的采用的优化是 IterativeOptimizer。详细的关于 IterativeOptimizer的介绍，可以参考CMU的Andy Pavlo在数据库课程中有清晰详细的介绍：https://15721.courses.cs.cmu.edu/spring2018/slides/16-optimizer2.pdf

在AST绑定相应元数据后（Analysis 主要是用于绑定元数据），将把AST转换成逻辑计划树PlanNode。

// 生成逻辑计划树

public PlanNode planStatement(Analysis analysis, Statement statement)

{

if (statement instanceof CreateTableAsSelect && analysis.isCreateTableAsSelectNoOp()) {

Symbol symbol = symbolAllocator.newSymbol("rows", BIGINT);

PlanNode source = new ValuesNode(idAllocator.getNextId(), ImmutableList.of(symbol), ImmutableList.of(ImmutableList.of(new LongLiteral("0"))));

return new OutputNode(idAllocator.getNextId(), source, ImmutableList.of("rows"), ImmutableList.of(symbol));

}

// 这里会根据不同类型的SQL生成不同类型的 RelationPlan 传入第一个参数

// 比如 statement 是一个 Insert、Delete 或者还是一个 Query 等等，

// 比如 delete 语句，那么就会生成一个 DeleteNode，

return createOutputPlan(planStatementWithoutOutput(analysis, statement), analysis);

}

接下来该 PlanNode 所表示的逻辑执行计划会被 PlanOptimizer 进行优化，生成一颗优化后的PlanNode Tree。

public interface PlanOptimizer

{

PlanNode optimize(PlanNode plan,

Session session,

TypeProvider types,

SymbolAllocator symbolAllocator,

PlanNodeIdAllocator idAllocator,

WarningCollector warningCollector);

}

这里可以用一条SQL语句来验证优化前后的逻辑计划树长啥样，我采用 tpc-h catalog做为进入Presto的CLI，假设此时的SQL为:

select \* from customer limit 12;

没有被优化前的逻辑计划树如下：

OutputNode

ProjectNode

LimitNode

TableScanNode

执行优化后的逻辑计划树如下，具体如何做PlanNode的优化，则不在本篇文章讨论的范围内，有兴趣的可以查看具体优化器的代码。

OutputNode

LimitNode

ExchangeNode

最后会把优化后的 PlanNode Tree 生成 Plan 执行计划。

QueryExecution表示一次查询执行，用于启动、停止与管理一个查询，以及统计这个查询的相关信息。QueryExecution是一个接口，其实现类有DataDefinitionExecution、SqlQueryExecution和FailedQueryExecution。

### 获取QueryExecutionFactory

### 创建QueryExecution

### 启动QueryExecution

## 语义分析

### Statement分析

### Relation分析

### 表达式分析

## 执行计划生成

参考：<https://zhuanlan.zhihu.com/p/57395047>

### 执行计划节点

### SQL执行计划

### Relation执行计划

### Query执行计划

## 执行计划优化

参考：<https://zhuanlan.zhihu.com/p/52154130>

Presto的LogicalPlanner会调用一堆Optimizer对PlanTree进行优化。实际上Presto将所有对PlanTree的改写逻辑都加到了planOptimizers里面。跟Calcite相比，Presto的优化器做的还是挺业余的，参考价值不大。如果不是要往里面加优化器的话，值得看的几条优化规则有ReorderJoins（影响Join的执行顺序）， AddExchanges （影响SubPlan的划分）和DetermineJoinDistributionType （影响Join结点的类型）。Predict下沉到数据源的逻辑似乎不是在Optimize 的过程做的。

Presto 里的优化规则可以在 com.facebook.presto.sql.planner#PlanOptimizers 找到。规则分两类，第一类是在 com.facebook.presto.sql.planner.optimizations 里，都是通过 visitor 模式来对树进行改写；第二类在 com.facebook.presto.sql.planner.iterative.rule 里，这里的规则都是通过 pattern match 来触发。Presto一开始都是手写 visitor，后面规则一多就复杂了，于是加了个pattern match的模块来触发规则，后期的规则就都用 pattern 来触发了。可以在Optimizers里发现数十个形如 PruneXXXColumns， PushLimitThroughXXX 的优化规则，实际上列裁剪和 limit 下推这种规则还是写 visitor 好一些。

### ReorderJoins

SQL 的优化规则可以分成两种：Join重排和其他。Join重排的计算量过于巨大，与其它规则根本不是一个量级，有的数据库甚至使用了遗传算法来计算最优顺序。Presto目前的Join重排算法是基于动态规划的，还设置了超时机制。据我所知，大部分公司对单条SQL中join的数量都是有限制的，动态规划倒也能解决大部分场景的需求。

### AddExchange

Presto中，AddExchangesSubPlan是根据Exchange划分的。因此 AddExchanges其实是划分SubPlan的过程。阅读这部分代码可以帮助我们了解 Presto 的分布式执行逻辑。代码很长，不过基本都是看一下当前结点的种类和meta，根据一定的规则插结点。AddExchanges将optimize的过程分成了两个阶段：

第一阶段大部分都可以在单机上做，是比较传统的规则，包括谓词下推，列消除，Join 重排以及一些简单的结点变换。

第二阶段由于Exchange 结点将执行计划分成了多个阶段，因此又跑了一些通用优化，除此之外还多了一些针对 SubPlan 的优化，比如 PushPartialAggregationThroughJoin 等等。

不同优化规则的执行顺序是个很有意思的问题，感兴趣的读者可以看一下 com.facebook.presto.sql.planner$PlanOptimizers，注释还是挺多的，作者想通过注释将 optimize 的过程分成几个阶段。

### DetermineJoinDistributionType

这个优化规则决定了Join的类型，还有一个DetermineSemiJoinDistributionType的规则，前者是典型的CBO规则，后者是RBO规则，代码都很简洁，也挺重要的。想弄清楚DetermineJoinDistributionType的逻辑还是要看CostProvider的具体实现。Presto的cost计算机制还是有点复杂的，用到了cache，计算了子树的cost， 最终的计算逻辑在com.facebook.presto.cost$CostCalculatorWithEstimatedExchanges#calculateJoinCost 里。

### 总结

Presto的optimizer有着浓浓的半成品的气息，另外作者是将大部分对Plan 的改写逻辑丢到optimizer里边了。不知道把AddExchange这种划分子图的逻辑当成优化规则是不是通用做法。考虑到针对异构计算节点的优化和一些编译优化，划分子图之后最好单独做一些优化工作，甚至可以将划分后的子图丢到 worker 上做优化。总而言之Optimizer是Presto相当重要的一个部分，直接决定了各个子计划的形态。

## 执行计划分段

经过执行计划生成与执行计划优化之后，最后对执行计划进行分段：

SubPlan subplan = new PlanFragmenter().createSubPlans(plan);

Presto根据执行计划的操作分为如下几个阶段：

### Source

Source阶段是从数据源的表中读取数据的阶段，一般包括tableScanNode和ProjectNode，以及可能存在的FilterNode等。

### Fixed

Fixed阶段位于Source阶段之后，该阶段将Source阶段读取的数据分散到多个节点上进行处理，主要处理的操作有局部聚合、局部Join、局部数据写入表等。

### Single

Single阶段位于Fixed阶段之后，只在单个节点上执行，用于汇总所有的查询结果，例如针对局部聚合的数据进行最终聚合，并将结果传输给Coordinator。

另外全局排序的操作也是在Single节点进行。

### Coordinator\_only

Coordinator\_only阶段只在Coordinator上进行，对insert和create table操作进行Commit的TableCommitNode属于Coordinator\_only阶段。

# 查询调度

Presto根据生成的逻辑执行计划将其拆分成多个且有层级关系的Stage，Presto的各个Stage都可以理解为将整个查询的执行计划分为若干段，实际就是将整个SQL处理过程拆分成多个具有各自功能的执行阶段。每个执行阶段都会被进一步分解为若干个Task，Presto将查询请求解析成各个执行阶段后，便会将各个阶段分配到各个计算节点中执行，这个分配的过程实际是基于Stage进行的，每个Stage的调度过程是基于Split分配Worker Node的过程，不同的Stage有不同的调度策略。

## 生成调度执行器

## 查询调度过程

### NodeManager

### NodeSelector

# 查询执行

## 查询执行逻辑

## Task调度

### Source Task调度

### Fixed Task调度

### Single Task调度

### Coordinator\_Only Task调度

## Task执行

### 创建Task

### 更新Task

### 运行Task

# 队列

Presto队列是用于控制查询并发量和可接收的SQL数量，可针对用户、提交来源、Session等信息进行个性化配置。队列规则定义在一个JSON文件内，可在$PRESTO\_HOME/etc/config.properties中通过配置参数query.queue-config-file来指定队列规则的JSON文件路径。

## 配置说明

## 队列加载

## 队列匹配

# System Connector

System Connector提供了Presto集群的运行指标以及相关的元数据信息，这些集群信息也可以通过标准的SQL语句进行查询。System Connector不需要配置专门的System Catalog，Presto Server启动后，System Connector默认可用。

## System Connector使用

使用presto-cli进入到命令行后，用show schemas from system显示出System Catalog所提供的信息。

### Information\_schema

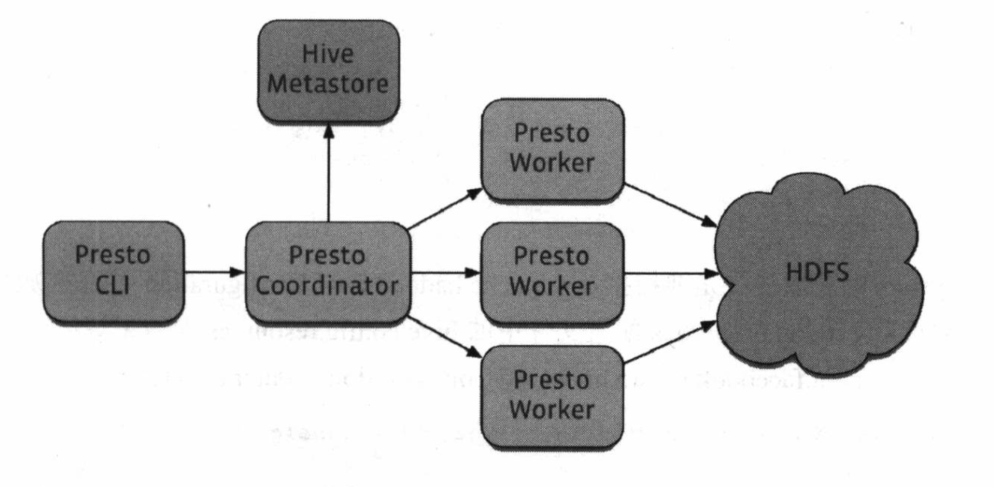
### Metadata

### Runtime

## System Connector实现

# Hive Connector

在Presto提供的众多数据源Connector中，最常用的为Hive Connector，并且也是查询性能最好的数据源之一。Hive Connector使用Hive的元数据，Coordinator节点通过Hive Metastore加载元数据信息，Presto的计算节点读取Hive表对应的HDFS数据，整体架构如下：



## 与Hive的结合

## Split分片管理

## 数据读取

## Create table as select的实现

# Kafk Connector

## 概述

### 配置

### 配置属性

### 内置字段

### 表定义文件

### Kafka中的key和message

### 行解码

### 日期和时间解码器

### 文本解码器

### 数值解码器

## Kafka连接器使用教程

### 安装Apache Kafka

### 下载数据

### 在Presto中配置Kafka topics

### 基本数据查询

### 添加表定义文件

### 将message中所有值映射到不同列

### 使用实时数据

## Kafka Connector获取数据

### Split分片管理

### 数据读取

# Connector开发

## 创建Maven工程

## 注册Plugin

## Connector

## Metadata

## SplitManager

## RecordSetProvider

# Functions开发

Presto的Functions是使用在SQL语中，用于实现特定功能的函数。Presto内部实现了一系列的Functions，提供了非常丰富的Function操作，常用的Functions包括以下几类。

比较类型函数，例如：greatest，least。

条件类型函数，例如：case，if。

类型转换函数，例如：cast，type \_cast。

数学函数，例如：abs，ceil，floor。

字符相关操作函数，例如：concat，length，replace。

时间函数，例如：current date，now，date add。

正则函数，例如：regexp like，regexp\_replace。

JSON 相关函数，例如：json array\_length，json extract。

URL相关函数，例如：url extract fragment，url extract host。

聚合函数，例如：sum，count。

窗口函数，例如：row number，rank。0

复杂类型操作函数，例如：Array/Map。

## Function注册

## 窗口函数

## 聚合函数

# Presto使用SQL

## Presto语句

## Presto系统表

## Catalog

## Schema

## information Schema

## 表

## 视图

## 会话信息和配置

## 数据类型

## SELECT语句基础

## WHERE子句

## GROUP BY和HAVING子句

## ORDER BY子句和LIMIT子句

## JOIN语句

## UNION、INTERSECT和EXCEPT子句

## 分组操作

## WITH子句

## 子查询

## 从表中删除数据

# 高级SQL特性

## 函数和运算符介绍

## 标量函数和运算符

## 布尔运算符

## 逻辑运算符

## 用BETWEEN语句选择范围

## 用IS(NOT) NULL检测值的存在

## 数学函数和运算符

## 三角函数

## 常数和随机函数

## 字符串函数和运算符

## 字符串和映射

## Unicode

## 正则表达式

## 解嵌套复杂数据类型

## JSON函数

## 日期和时间函数及运算符

## 直方图

## 聚合函数

## 窗函数

## lambda表达式

## 地理空间函数

## Prepared Statement

# JD-Presto功能改造

## PDBO功能开发

## DDL及DML支持

## 动态增加、修改、删除Catalog

# 性能调优

## 合理设计分区

## Group by字段优化

## 使用模糊聚合函数

## 合并多条like子句为一条regexp\_like子句

## 大表放在join子句左边

## 关闭distributed hash join

## 使用ORC存储

# 安全