# 概述

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)。

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的设计和编写完全是为了解决像Facebook这样规模的商业数据仓库的交互式分析和处理速度的问题。目前国内的美团、京东、滴滴等在业务中都有使用presto。

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。

## 基本概念

### Presto服务进程

### Presto模型

### Presto查询执行模型

## 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>

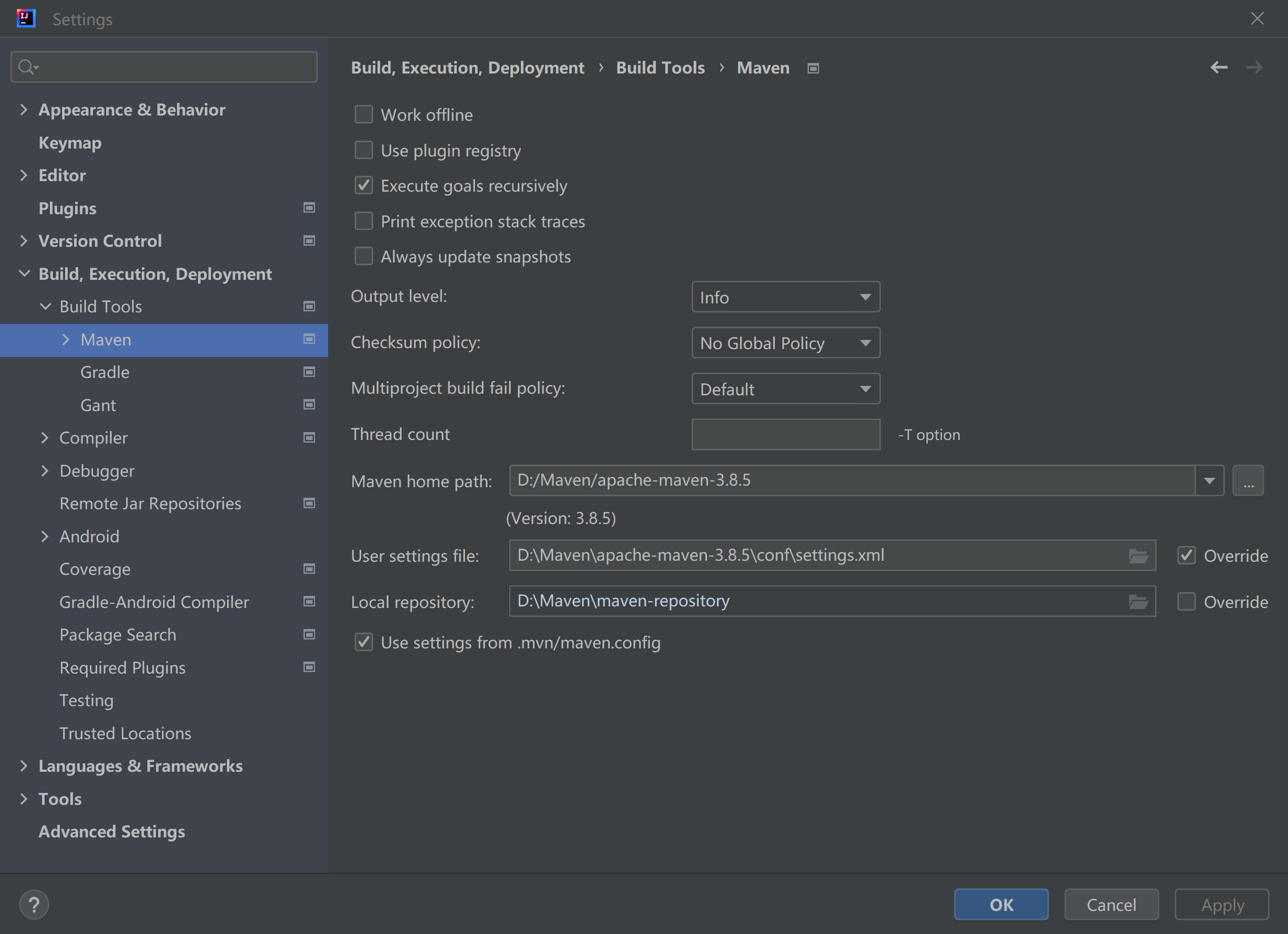
<id>central</id>

<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

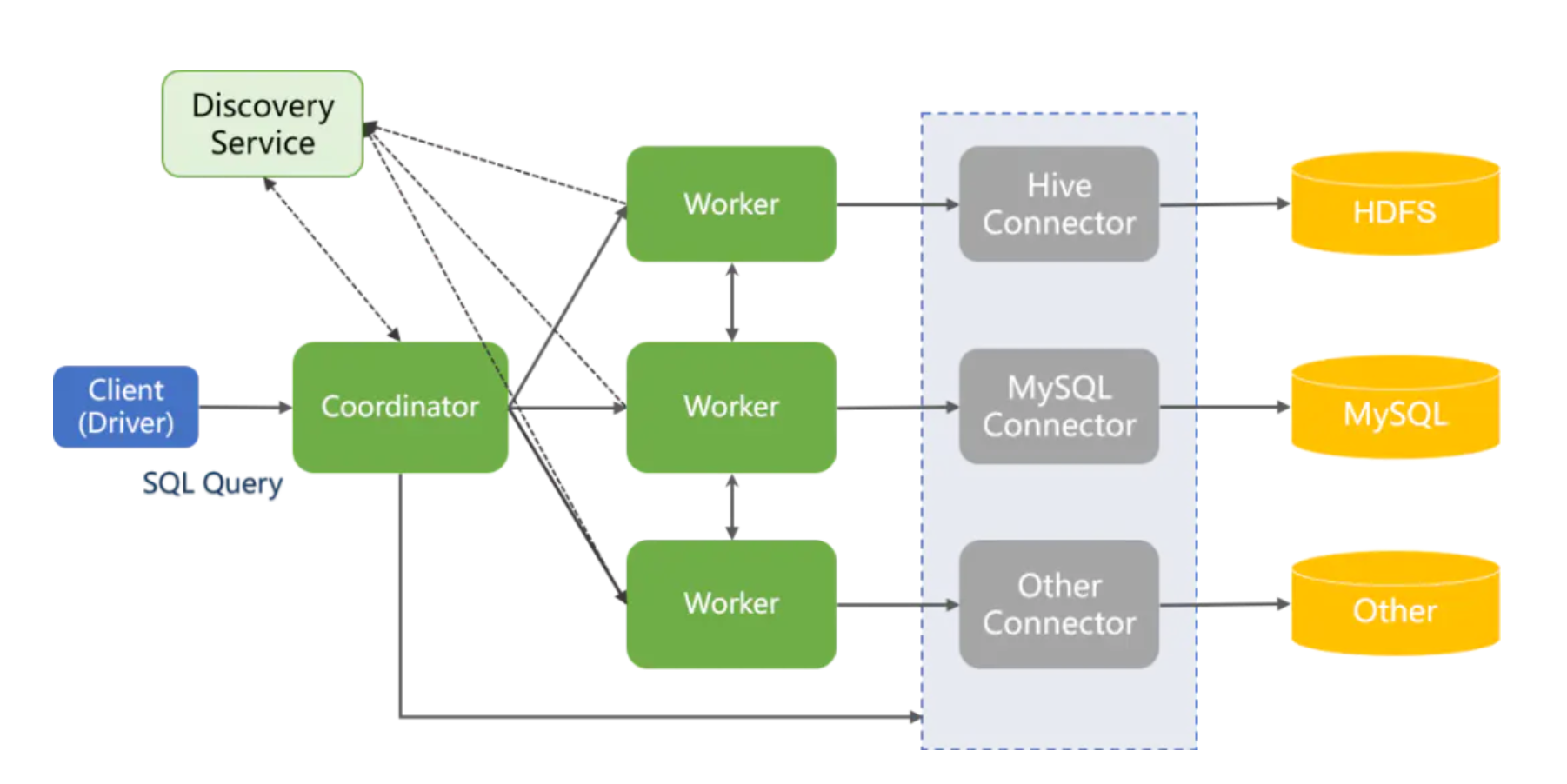
### Presto与ODBC

### 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

# 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>

## 数据模型

### 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。

#### 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

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；

Fixed：接受子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。

### Exchange

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。

### 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。

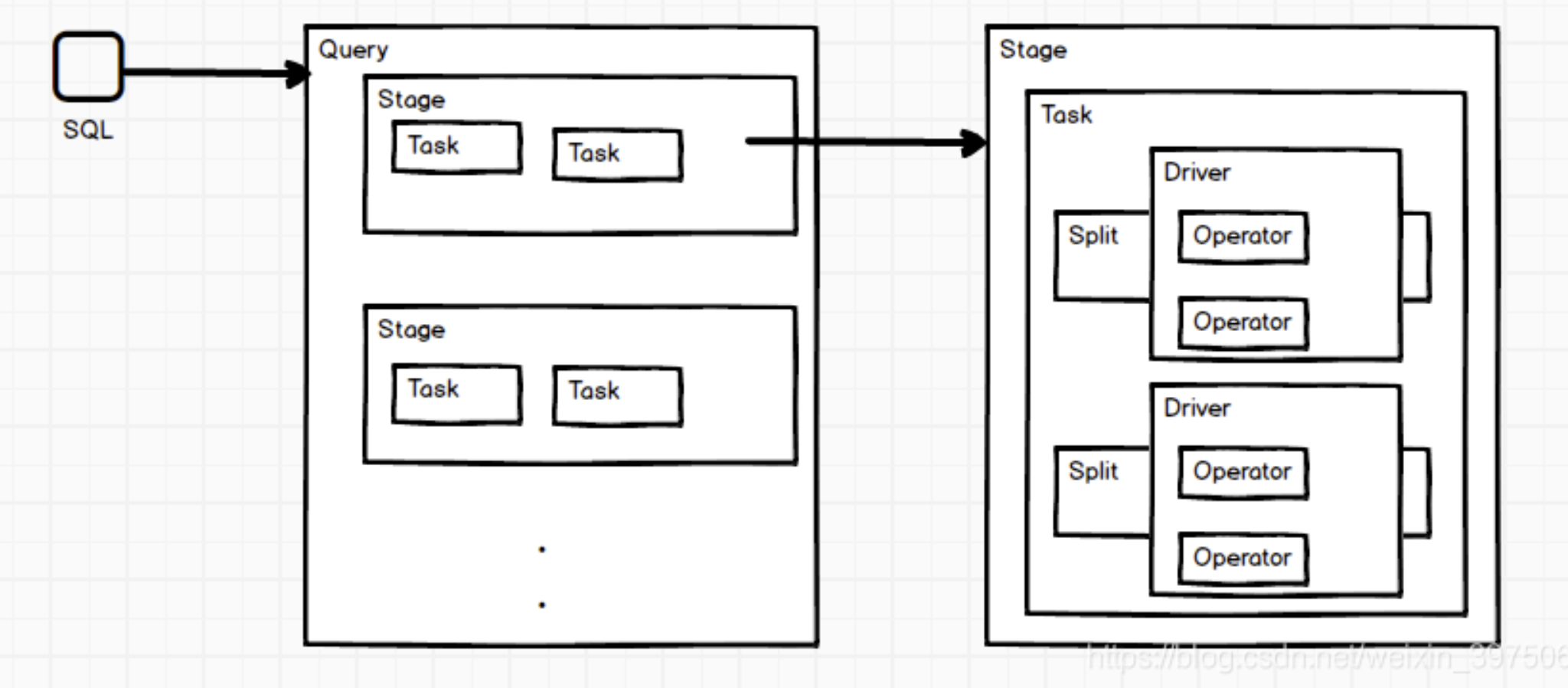
由于Driver是处理一个Split的一系列操作的集合，而一个Task包含多个Driver，所以一个Task可以操作多个Split。

### 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：



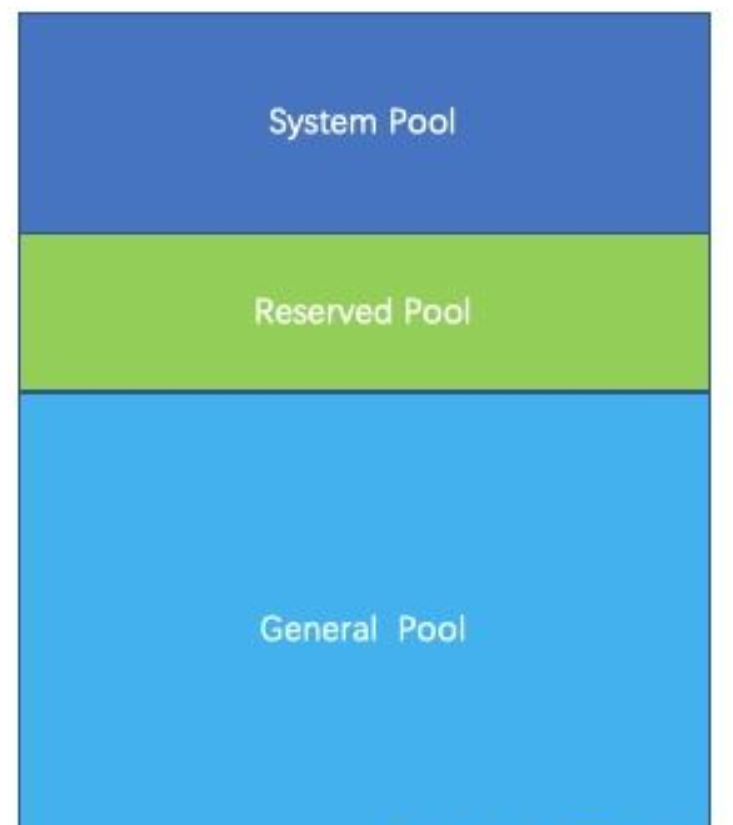
## 内存管理

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。

# 优化器

# 连接器

# 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

# 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