目录

Druid入门学习类文章

1. 十分钟了解Apache Druid

   Apache Druid是一个集时间序列数据库、数据仓库和全文检索系统特点于一体的分析性数据平台。本文将带你简单了解Druid的特性，使用场景，技术特点和架构。这将有助于你选型数据存储方案，深入了解Druid存储，深入了解时间序列存储等。

   原文链接

2. 勾叔谈大数据：大厂做法：Apache Druid在电商领域的实践应用

   Apache Druid虽然尚未在各个企业绝对普及，但是在互联网大厂是得到了较多应用的，毕竟它出道时间不长，还算作是新技术呢，而对于新技术，互联网一线大厂往往是践行者。

   原文链接

3. Kylin、Druid、ClickHouse核心技术对比

   Druid索引结构使用自定义的数据结构，整体上它是一种列式存储结构，每个列独立一个逻辑文件（实际上是一个物理文件，在物理文件内部标记了每个列的start和offset）

   原文链接

4. 适用于大数据的开源OLAP系统的比较：ClickHouse，Druid和Pinot

   ClickHouse，Druid和Pinot在效率和性能优化上具有大约相同的“极限”。没有“魔术药”可以使这些系统中的任何一个都比其他系统快得多。在当前状态下，这些系统在某些基准测试中的性能有很大不同，这一事实并不会让您感到困惑。

   原文链接

5. 有人说下kudu,kylin,druid,clickhouse的区别,使用场景么?

   Kylin 和 ClickHouse 都能通过 SQL 的方式在 PB 数据量级下,亚秒级(绝多数查询 5s内返回)返回 OLAP(在线分析查询) 查询结果

   原文链接

6. OLAP演进实战，Druid对比ClickHouse输在哪里?

   本文介绍eBay广告数据平台的基本情况，并对比分析了ClickHouse与Druid的使用特点。基于ClickHouse表现出的良好性能和扩展能力，本文介绍了如何将eBay广告系统从Druid迁移至ClickHouse，希望能为同业人员带来一定的启发。

   原文链接

7. clickhouse和druid实时分析性能总结

clickhouse 是俄罗斯的“百度”Yandex公司在2016年开源的，一款针对大数据实时分析的高性能分布式数据库，与之对应的有hadoop生态hive，Vertica和百度出品的palo。

原文链接

Druid源码解析类文章合集

1. Apache Druid源码导读--Google Guice DI框架

   在大数据应用组件中，有两款OLAP引擎应用广泛，一款是偏离线处理的Kylin，另一个是偏实时的Druid。Kylin是一款国人开源的优秀离线OLAP引擎，基本上是Hadoop领域离线OLAP事实标准，在离线报表，指标分析领域应用广泛。而Apache Druid则在实时OLAP领域独领风骚，优异的性能、高可用、易扩展。

   原文链接)

2. Apache Druid源码解析的一个合集

   原文链接

   - Druid中的Extension在启动时是如何加载的
   - Druid解析之管理用的接口大全
   - Druid原理分析之内存池管理
   - Druid源码解析之Segment
   - Druid源码解析之Column
   - Druid源码解析之HDFS存储
   - Druid源码解析之Coordinator
   - 让Druid实现事件设备数留存数的精准计算
   - 在Druid中定制自己的扩展【Extension】
   - Druid原理分析之“批”任务数据流转过程
   - Druid原理分析之“流”任务数据流转过程
   - Druid原理分析之Segment的存储结构
   - Druid索引与查询原理简析
   - Druid中的负载均衡策略分析
   - Druid中的Kafka Indexing Service源码分析
   - Druid源码分析之Query -- Sequence与Yielder
   - Druid原理分析之Segment的存储结构

各个大厂对Druid的优化与实践类文章合集

1. 快手 Druid 精确去重的设计和实现

   快手的业务特点包括超大数据规模、毫秒级查询时延、高数据实时性要求、高并发查询、高稳定性以及较高的 Schema 灵活性要求；因此快手选择 Druid 平台作为底层架构。由于 Druid 原生不支持数据精确去重功能，而快手业务中会涉及到例如计费等场景，有精确去重的需求。因此，本文重点讲述如何在 Druid 平台中实现精确去重。另一方面，Druid 对外的接口是 json 形式 (Druid 0.9 版本之后逐步支持 SQL)，对 SQL 并不友好，本文最后部分会简述 Druid 平台与 MySQL 交互方面做的一些改进。

   原文链接

2. 基于ApacheDruid的实时分析平台在爱奇艺的实践

   爱奇艺大数据服务团队评估了市面上主流的OLAP引擎，最终选择Apache Druid时序数据库来满足业务的实时分析需求。本文将介绍Druid在爱奇艺的实践情况、优化经验以及平台化建设的一些思考

   原文链接

3. 熵简技术谈 | 实时OLAP引擎之Apache Druid：架构、原理和应用实践

   本文以实时 OLAP 引擎的优秀代表 Druid 为研究对象，详细介绍 Druid 的架构思想和核心特性。在此基础上，我们介绍了熵简科技在数据智能分析场景下，针对私有化部署与实时响应优化的实践经验。

   原文链接

4. Apache Druid性能测评-云栖社区-阿里云

   原文链接

5. Druid在有赞的实践

   有赞作为一家 SaaS 公司，有很多的业务的场景和非常大量的实时数据和离线数据。在没有是使用 Druid 之前，一些 OLAP 场景的场景分析，开发的同学都是使用 SparkStreaming 或者 Storm 做的。用这类方案会除了需要写实时任务之外，还需要为了查询精心设计存储。带来问题是：开发的周期长；初期的存储设计很难满足需求的迭代发展；不可扩展。

   原文链接

6. Druid 在小米公司的技术实践

   Druid 作为一款开源的实时大数据分析软件，自诞生以来，凭借着自己优秀的特质，逐渐在技术圈收获了越来越多的知名度与口碑，并陆续成为了很多技术团队解决方案中的关键一环，从而真正在很多公司的技术栈中赢得了一席之地。 原文链接

Apache Druid是一个高性能的实时分析型数据库

- ## 云原生、流原生的分析型数据库

  Druid专为需要快速数据查询与摄入的工作流程而设计，在即时数据可见性、即席查询、运营分析以及高并发等方面表现非常出色。在实际中众多场景下数据仓库解决方案中，可以考虑将Druid当做一种开源的替代解决方案。

- ## 可轻松与现有的数据管道进行集成

  Druid原生支持从Kafka、Amazon Kinesis等消息总线中流式的消费数据，也同时支持从HDFS、Amazon S3等存储服务中批量的加载数据文件。

- ## 较传统方案提升近百倍的效率

  Druid创新地在架构设计上吸收和结合了数据仓库、时序数据库以及检索系统)的优势，在已经完成的基准测试中展现出来的性能远远超过数据摄入与查询的传统解决方案。

- ## 解锁了一种新型的工作流程

  Druid为点击流、APM、供应链、网络监测、市场营销以及其他事件驱动类型的数据分析解锁了一种新型的查询与工作流程，它专为实时和历史数据高效快速的即席查询而设计。

- ## 可部署在AWS/GCP/Azure,混合云,Kubernetes, 以及裸机上

  无论在云上还是本地，Druid可以轻松的部署在商用硬件上的任何*NIX环境。部署Druid也是非常简单的，包括集群的扩容或者下线都也同样很简单。

> [!TIP] 在国内Druid的使用者越来越多，但是并没有一个很好的中文版本的使用文档。 本文档根据Apache Druid官方文档0.18.0版本进行翻译，目前托管在Github上，欢迎更多的Druid使用者以及爱好者加入翻译行列，为国内的使用者提供一个高质量的中文版本使用文档。

Druid是什么

Apache Druid是一个实时分析型数据库，旨在对大型数据集进行快速的查询分析（"OLAP"查询)。Druid最常被当做数据库来用以支持实时摄取、高性能查询和高稳定运行的应用场景，同时，Druid也通常被用来助力分析型应用的图形化界面，或者当做需要快速聚合的高并发后端API，Druid最适合应用于面向事件类型的数据。

Druid通常应用于以下场景：

- 点击流分析（Web端和移动端）
- 网络监测分析（网络性能监控）
- 服务指标存储
- 供应链分析（制造类指标）
- 应用性能指标分析
- 数字广告分析
- 商务智能 / OLAP

Druid的核心架构吸收和结合了数据仓库、时序数据库以及检索系统)的优势，其主要特征如下：

1. **列式存储**，Druid使用列式存储，这意味着在一个特定的数据查询中它只需要查询特定的列，这样极地提高了部分列查询场景的性能。另外，每一列数据都针对特定数据类型做了优化存储，从而支持快速的扫描和聚合。
2. **可扩展的分布式系统**，Druid通常部署在数十到数百台服务器的集群中，并且可以提供每秒数百万条记录的接收速率，数万亿条记录的保留存储以及亚秒级到几秒的查询延迟。
3. **大规模并行处理**，Druid可以在整个集群中并行处理查询。
4. **实时或批量摄取**，Druid可以实时（已经被摄取的数据可立即用于查询）或批量摄取数据。
5. **自修复、自平衡、易于操作**，作为集群运维操作人员，要伸缩集群只需添加或删除服务，集群就会在后台自动重新平衡自身，而不会造成任何停机。如果任何一台Druid服务器发生故障，系统将自动绕过损坏。 Druid设计为7*24全天候运行，无需出于任何原因而导致计划内停机，包括配置更改和软件更新。
6. **不会丢失数据的云原生容错架构**，一旦Druid摄取了数据，副本就安全地存储在深度存储介质（通常是云存储，HDFS或共享文件系统）中。即使某个Druid服务发生故障，也可以从深度存储中恢复您的数据。对于仅影响少数Druid服务的有限故障，副本可确保在系统恢复时仍然可以进行查询。
7. **用于快速过滤的索引**，Druid使用CONCISE或Roaring压缩的位图索引来创建索引，以支持快速过滤和跨多列搜索。
8. **基于时间的分区**，Druid首先按时间对数据进行分区，另外同时可以根据其他字段进行分区。这意味着基于时间的查询将仅访问与查询时间范围匹配的分区，这将大大提高基于时间的数据的性能。
9. **近似算法**，Druid应用了近似count-distinct，近似排序以及近似直方图和分位数计算的算法。这些算法占用有限的内存使用量，通常比精确计算要快得多。对于精度要求比速度更重要的场景，Druid还提供了精确count-distinct和精确排序。
10. **摄取时自动汇总聚合**，Druid支持在数据摄取阶段可选地进行数据汇总，这种汇总会部分预先聚合您的数据，并可以节省大量成本并提高性能。

什么场景下应该使用Druid

许多公司都已经将Druid应用于多种不同的应用场景，详情可查看Powered by Apache Druid页面。

如果您的使用场景符合以下的几个特征，那么Druid是一个非常不错的选择：

- 数据插入频率比较高，但较少更新数据
- 大多数查询场景为聚合查询和分组查询（GroupBy），同时还有一定得检索与扫描查询
- 将数据查询延迟目标定位100毫秒到几秒钟之间
- 数据具有时间属性（Druid针对时间做了优化和设计）
- 在多表场景下，每次查询仅命中一个大的分布式表，查询又可能命中多个较小的lookup表
- 场景中包含高基维度数据列（例如URL，用户ID等），并且需要对其进行快速计数和排序
- 需要从Kafka、HDFS、对象存储（如Amazon S3）中加载数据

如果您的使用场景符合以下特征，那么使用Druid可能是一个不好的选择：

- 根据主键对现有数据进行低延迟更新操作。Druid支持流式插入，但不支持流式更新（更新操作是通过后台批处理作业完成）
- 延迟不重要的离线数据系统
- 场景中包括大连接（将一个大事实表连接到另一个大事实表），并且可以接受花费很长时间来完成这些查询

快速开始

在本快速入门教程中，我们将下载Druid并将其安装在一台服务器上，完成初始安装后，向集群中加载数据。

在开始快速入门之前，阅读Druid概述和数据摄取概述会很有帮助，因为当前教程会引用这些页面上讨论的概念。

预备条件

软件

- **Java 8(8u92+)**
- Linux, Mac OS X, 或者其他类UNIX系统（Windows不支持）

> [!WARNING] Druid服务运行依赖Java 8，可以使用环境变量 `DRUID_JAVA_HOME` 或 `JAVA_HOME` 指定在何处查找Java,有关更多详细信息，请运行 `verify-java` 脚本。

硬件

Druid安装包提供了几个单服务器配置的示例，以及使用这些配置启动Druid进程的脚本。

如果您正在使用便携式等小型计算机上运行服务，则配置为4CPU/16GB RAM环境的 `micro-quickstart` 配置是一个不错的选择。

如果您打算在本教程之外使用单机部署进行进一步试验评估，则建议使用比 `micro-quickstart` 更大的配置。

入门开始

下载Druid最新0.17.0release安装包

在终端中运行以下命令来提取Druid

```
tar -xzf apache-druid-0.17.0-bin.tar.gz
cd apache-druid-0.17.0
```

在安装包中有以下文件：

- `LICENSE` 和 `NOTICE` 文件
- `bin/*` - 启停等脚本
- `conf/*` - 用于单节点部署和集群部署的示例配置
- `extensions/*` - Druid核心扩展
- `hadoop-dependencies/*` - Druid Hadoop依赖
- `lib/*` - Druid核心库和依赖
- `quickstart/*` - 配置文件，样例数据，以及快速入门教材的其他文件

启动服务

以下命令假定您使用的是 `micro-quickstart` 单机配置，如果使用的是其他配置，在 `bin` 目录下有每一种配置对应的脚本，如 `bin/start-single-server-small`

在 `apache-druid-0.17.0` 安装包的根目录下执行命令：

```
./bin/start-micro-quickstart
```

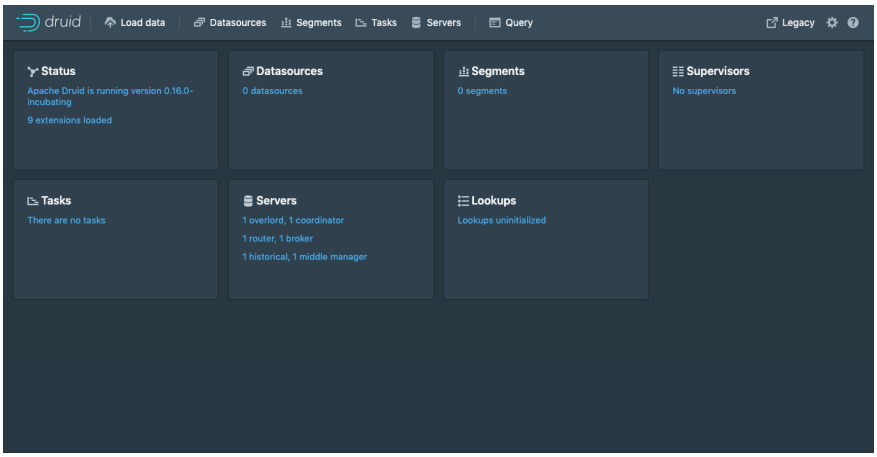
然后将在本地计算机上启动Zookeeper和Druid服务实例，例如：

```
$ ./bin/start-micro-quickstart
[Fri May  3 11:40:50 2019] Running command[zk], logging to[/apache-druid-0.17.
[Fri May  3 11:40:50 2019] Running command[coordinator-overlord], logging to[/
[Fri May  3 11:40:50 2019] Running command[broker], logging to[/apache-druid-0
[Fri May  3 11:40:50 2019] Running command[router], logging to[/apache-druid-0
[Fri May  3 11:40:50 2019] Running command[historical], logging to[/apache-dru
[Fri May  3 11:40:50 2019] Running command[middleManager], logging to[/apache-
```

所有的状态（例如集群元数据存储和服务的segment文件）将保留在 `apache-druid-0.17.0` 软件包根目录下的 `var` 目录中, 服务的日志位于 `var/sv` 。

稍后，如果您想停止服务，请按 `CTRL-C` 退出 `bin/start-micro-quickstart` 脚本，该脚本将终止Druid进程。

集群启动后，可以访问http://localhost:8888来Druid控制台，控制台由Druid Router进程启动。



所有Druid进程完全启动需要花费几秒钟。 如果在启动服务后立即打开控制台，则可能会看到一些可以安全忽略的错误。

（以下为广告，请见谅）

加载数据

教程使用的数据集

对于以下数据加载教程，我们提供了一个示例数据文件，其中包含2015年9月12日发生的Wikipedia页面编辑事件。

该样本数据位于Druid包根目录的 `quickstart/tutorial/wikiticker-2015-09-12-sampled.json.gz` 中,页面编辑事件作为JSON对象存储在文本文件中。

示例数据包含以下几列，示例事件如下所示：

- added
- channel
- cityName
- comment
- countryIsoCode
- countryName
- deleted
- delta
- isAnonymous
- isMinor
- isNew
- isRobot
- isUnpatrolled
- metroCode
- namespace
- page
- regionIsoCode
- regionName
- user

```
{
  "timestamp":"2015-09-12T20:03:45.018Z",
  "channel":"#en.wikipedia",
  "namespace":"Main",
  "page":"Spider-Man's powers and equipment",
  "user":"foobar",
  "comment":"/* Artificial web-shooters */",
  "cityName":"New York",
  "regionName":"New York",
  "regionIsoCode":"NY",
  "countryName":"United States",
  "countryIsoCode":"US",
  "isAnonymous":false,
  "isNew":false,
  "isMinor":false,
  "isRobot":false,
  "isUnpatrolled":false,
  "added":99,
  "delta":99,
  "deleted":0,
}
```

数据加载

以下教程演示了将数据加载到Druid的各种方法，包括批处理和流处理用例。 所有教程均假定您使用的是上面提到的 `micro-quickstart` 单机配置。

- 加载本地文件 - 本教程演示了如何使用Druid的本地批处理摄取来执行批文件加载
- 从Kafka加载流数据 - 本教程演示了如何从Kafka主题加载流数据
- 从Hadoop加载数据 - 本教程演示了如何使用远程Hadoop集群执行批处理文件加载
- 编写一个自己的数据摄取规范 - 本教程演示了如何编写新的数据摄取规范并使用它来加载数据

重置集群状态

如果要在清理服务后重新启动，请删除 `var` 目录，然后再次运行 `bin/start-micro-quickstart` 脚本。

一旦每个服务都启动，您就可以加载数据了。

重置Kafka

如果您完成了教程：从Kafka加载流数据并希望重置集群状态，则还应该清除所有Kafka状态。

在停止ZooKeeper和Druid服务之前，使用 `CTRL-C` 关闭 `Kafka Broker`，然后删除 `/tmp/kafka-logs` 中的Kafka日志目录：

```
rm -rf /tmp/kafka-logs
```

Docker

在这个部分中，我们将从 Docker Hub 下载Apache Druid镜像，并使用 Docker 和 Docker Compose 在一台机器上安装它。完成此初始设置后，集群将准备好加载数据。

在开始快速启动之前，阅读 Druid概述 和 摄取概述 是很有帮助的，因为教程将参考这些页面上讨论的概念。此外，建议熟悉Docker。

前提条件

- Docker

快速开始

Druid源代码包含一个 示例docker-compose.yml 它可以从Docker Hub中提取一个镜像，适合用作示例环境，并用于试验基于Docker的Druid配置和部署。

Compose文件

示例 `docker-compose.yml` 将为每个Druid服务创建一个容器，包括Zookeeper和作为元数据存储PostgreSQL容器。深度存储将是本地目录，默认配置为相对于 `docker-compose.yml` 文件的 `./storage` ，并将作为 `/opt/data` 挂载，并在需要访问深层存储的Druid容器之间共享。Druid容器是通过 环境文件 配置的。

配置

Druid Docker容器的配置是通过环境变量完成的，环境变量还可以指定到 标准Druid配置文件 的路径

特殊环境变量：

- `JAVA_OPTS` -- 设置 java options
- `DRUID_LOG4J` -- 设置完成的 `log4j.xml`
- `DRUID_LOG_LEVEL` -- 覆盖在log4j中的默认日志级别
- `DRUID_XMX` -- 设置 Java `Xmx`
- `DRUID_XMS` -- 设置 Java `Xms`
- `DRUID_MAXNEWSIZE` -- 设置 Java最大新生代大小
- `DRUID_NEWSIZE` -- 设置 Java 新生代大小
- `DRUID_MAXDIRECTMEMORYSIZE` -- 设置Java最大直接内存大小
- `DRUID_CONFIG_COMMON` -- druid "common"属性文件的完整路径
- `DRUID_CONFIG_${service}` -- druid "service"属性文件的完整路径

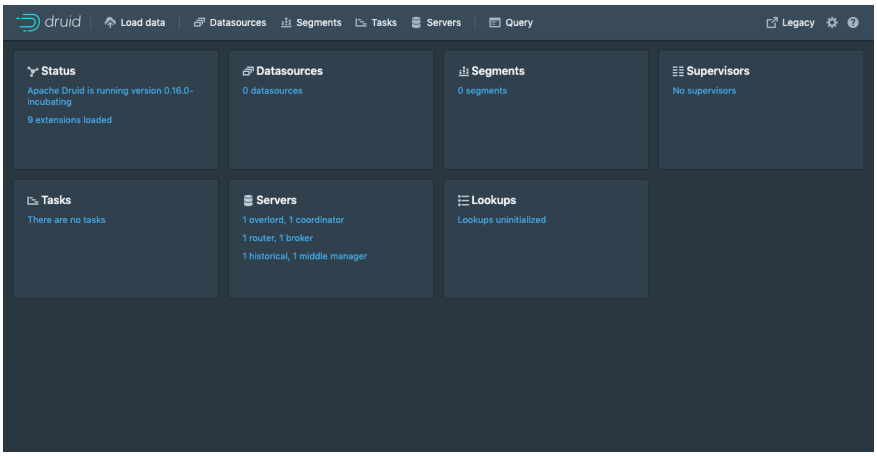
除了特殊的环境变量外，在容器中启动Druid的脚本还将尝试使用以 `druid_` 前缀开头的任何环境变量作为命令行配置。例如，Druid容器进程中的环境变量 `druid_metadata_storage_type=postgresql` 将被转换为 `-Ddruid.metadata.storage.type=postgresql`

Druid `docker-compose.yml` 示例使用单个环境文件来指定完整的Druid配置；但是，在生产用例中，我们建议使用 `DRUID_COMMON_CONFIG` 和 `DRUID_CONFIG_${service}` 或专门定制的特定于服务的环境文件。

启动集群

运行 `docker-compose up` 启动附加shell的集群，或运行 `docker-compose up -d` 在后台运行集群。如果直接使用示例文件，这个命令应该从Druid安装目录中的 `distribution/docker/` 运行。

启动集群后，可以导航到 http://localhost:8888 。服务于 Druid控制台 的 Druid路由进程 位于这个地址。



所有Druid进程需要几秒钟才能完全启动。如果在启动服务后立即打开控制台，可能会看到一些可以安全忽略的错误。

从这里你可以跟着 标准教程，或者详细说明你的 `docker-compose.yml` 根据需要添加任何其他外部服务依赖项。

单服务器部署

Druid包括一组参考配置和用于单机部署的启动脚本：

- `nano-quickstart`
- `micro-quickstart`
- `small`
- `medium`
- `large`
- `large`
- `xlarge`

`micro-quickstart` 适合于笔记本电脑等小型机器，旨在用于快速评估测试使用场景。

`nano-quickstart` 是一种甚至更小的配置，目标是具有1个CPU和4GB内存的计算机。它旨在在资源受限的环境（例如小型Docker容器）中进行有限的评估测试。

其他配置旨在用于一般用途的单机部署,它们的大小适合大致基于亚马逊i3系列EC2实例的硬件。

这些示例配置的启动脚本与Druid服务一起运行单个ZK实例,您也可以选择单独部署ZK。

通过Coordinator配置文档中描述的可选配置 `druid.coordinator.asOverlord.enabled = true` 可以在单个进程中同时运行Druid Coordinator和Overlord。

虽然为大型单台计算机提供了示例配置，但在更高规模下，我们建议在集群部署中运行Druid，以实现容错和减少资源争用。

（以下为广告，请见谅）

单服务器参考配置

Nano-Quickstart: 1 CPU, 4GB 内存

- 启动命令: `bin/start-nano-quickstart`
- 配置目录: `conf/druid/single-server/nano-quickstart`

Micro-Quickstart: 4 CPU, 16GB 内存

- 启动命令: `bin/start-micro-quickstart`
- 配置目录: `conf/druid/single-server/micro-quickstart`

Small: 8 CPU, 64GB 内存 (~i3.2xlarge)

- 启动命令: `bin/start-small`
- 配置目录: `conf/druid/single-server/small`

Medium: 16 CPU, 128GB 内存 (~i3.4xlarge)

- 启动命令: `bin/start-medium`
- 配置目录: `conf/druid/single-server/medium`

Large: 32 CPU, 256GB 内存 (~i3.8xlarge)

- 启动命令: `bin/start-large`
- 配置目录: `conf/druid/single-server/large`

X-Large: 64 CPU, 512GB 内存 (~i3.16xlarge)

- 启动命令: `bin/start-xlarge`
- 配置目录: `conf/druid/single-server/xlarge`

集群部署

Apache Druid旨在作为可伸缩的容错集群进行部署。

在本文档中，我们将安装一个简单的集群，并讨论如何对其进行进一步配置以满足您的需求。

这个简单的集群将具有以下特点：

- 一个Master服务同时起Coordinator和Overlord进程
- 两个可伸缩、容错的Data服务来运行Historical和MiddleManager进程
- 一个Query服务，运行Druid Broker和Router进程

在生产中，我们建议根据您的特定容错需求部署多个Master服务器和多个Query服务器，但是您可以使用一台Master服务器和一台Query服务器将服务快速运行起来，然后再添加更多服务器。

选择硬件

首次部署

如果您现在没有Druid集群，并打算首次以集群模式部署运行Druid，则本指南提供了一个包含预先配置的集群部署示例。

Master服务

Coordinator进程和Overlord进程负责处理集群的元数据和协调需求，它们可以运行在同一台服务器上。

在本示例中，我们将在等效于AWSm5.2xlarge实例的硬件环境上部署。

硬件规格为：

- 8核CPU
- 31GB内存

可以在 `conf/druid/cluster/master` 下找到适用于此硬件规格的Master示例服务配置。

Data服务

Historical和MiddleManager可以分配在同一台服务器上运行，以处理集群中的实际数据，这两个服务受益于CPU、内存和固态硬盘。

在本示例中，我们将在等效于AWSi3.4xlarge实例的硬件环境上部署。

硬件规格为：

- 16核CPU
- 122GB内存
- 2 * 1.9TB 固态硬盘

可以在 `conf/druid/cluster/data` 下找到适用于此硬件规格的Data示例服务配置。

Query服务

Druid Broker服务接收查询请求，并将其转发到集群中的其他部分，同时其可以可选的配置内存缓存。 Broker服务受益于CPU和内存。

在本示例中，我们将在等效于AWSm5.2xlarge实例的硬件环境上部署。

硬件规格为：

- 8核CPU
- 31GB内存

您可以考虑将所有的其他开源UI工具或者查询依赖等与Broker服务部署在同一台服务器上。

可以在 `conf/druid/cluster/query` 下找到适用于此硬件规格的Query示例服务配置。

其他硬件配置

上面的示例集群是从多种确定Druid集群大小的可能方式中选择的一个示例。

您可以根据自己的特定需求和限制选择较小/较大的硬件或较少/更多的服务器。

如果您的使用场景具有复杂的扩展要求，则还可以选择不将Druid服务混合部署（例如，独立的Historical Server）。

基本集群调整指南中的信息可以帮助您进行决策，并可以调整配置大小。

从单服务器环境迁移部署

如果您现在已有单服务器部署的环境，例如单服务器部署示例中的部署，并且希望迁移到类似规模的集群部署，则以下部分包含一些选择Master/Data/Query服务等效硬件的准则。

Master服务

Master服务的主要考虑点是可用CPU以及用于Coordinator和Overlord进程的堆内存。

首先计算出来在单服务器环境下Coordinator和Overlord已分配堆内存之和，然后选择具有足够内存的Master服务硬件，同时还需要考虑到为服务器上其他进程预留一些额外的内存。

对于CPU，可以选择接近于单服务器环境核数1/4的硬件。

Data服务

在为集群Data服务选择硬件时，主要考虑可用的CPU和内存，可行时使用SSD存储。

在集群化部署时，出于容错的考虑，最好是部署多个Data服务。

在选择Data服务的硬件时，可以假定一个分裂因子 `N` ，将原来的单服务器环境的CPU和内存除以 `N` ,然后在新集群中部署 `N` 个硬件规格缩小的Data服务。

Query服务

Query服务的硬件选择主要考虑可用的CPU、Broker服务的堆内和堆外内存、Router服务的堆内存。

首先计算出来在单服务器环境下Broker和Router已分配堆内存之和，然后选择可以覆盖Broker和Router内存的Query服务硬件，同时还需要考虑到为服务器上其他进程预留一些额外的内存。

对于CPU，可以选择接近于单服务器环境核数1/4的硬件。

基本集群调优指南包含有关如何计算Broker和Router服务内存使用量的信息。

选择操作系统

我们建议运行您喜欢的Linux发行版，同时还需要：

- **Java 8**

> [!WARNING] Druid服务运行依赖Java 8，可以使用环境变量 `DRUID_JAVA_HOME` 或 `JAVA_HOME` 指定在何处查找Java,有关更多详细信息，请运行 `verify-java` 脚本。

下载发行版

首先，下载并解压缩发布安装包。最好首先在单台计算机上执行此操作，因为您将编辑配置，然后将修改后的配置分发到所有服务器上。

下载Druid最新0.17.0release安装包

在终端中运行以下命令来提取Druid

```
tar -xzf apache-druid-0.17.0-bin.tar.gz
cd apache-druid-0.17.0
```

在安装包中有以下文件：

- `LICENSE` 和 `NOTICE` 文件
- `bin/*` - 启停等脚本
- `conf/druid/cluster/*` - 用于集群部署的模板配置
- `extensions/*` - Druid核心扩展
- `hadoop-dependencies/*` - Druid Hadoop依赖
- `lib/*` - Druid核心库和依赖
- `quickstart/*` - 与快速入门相关的文件

我们主要是编辑 `conf/druid/cluster/` 中的文件。

从单服务器环境迁移部署

在以下各节中，我们将在 `conf/druid/cluster` 下编辑配置。

如果您已经有一个单服务器部署，请将您的现有配置复制到 `conf/druid/cluster` 以保留您所做的所有配置更改。

配置元数据存储和深度存储

从单服务器环境迁移部署

如果您已经有一个单服务器部署，并且希望在整个迁移过程中保留数据，请在更新元数据/深层存储配置之前，按照元数据迁移和深层存储迁移中的说明进行操作。

这些指南针对使用Derby元数据存储和本地深度存储的单服务器部署。 如果您已经在单服务器集群中使用了非Derby元数据存储，则可以在新集群中可以继续使用当前的元数据存储。

这些指南还提供了有关从本地深度存储迁移段的信息。集群部署需要分布式深度存储，例如S3或HDFS。 如果单服务器部署已在使用分布式深度存储，则可以在新集群中继续使用当前的深度存储。

元数据存储

在 `conf/druid/cluster/_common/common.runtime.properties` 中，使用您将用作元数据存储的服务器地址来替换"metadata.storage.*":

- `druid.metadata.storage.connector.connectURI`
- `druid.metadata.storage.connector.host`

在生产部署中，我们建议运行专用的元数据存储，例如具有复制功能的MySQL或PostgreSQL，与Druid服务器分开部署。

MySQL扩展和PostgreSQL扩展文档包含有关扩展配置和初始数据库安装的说明。

深度存储

Druid依赖于分布式文件系统或大对象（blob）存储来存储数据，最常用的深度存储实现是S3（适合于在AWS上）和HDFS（适合于已有Hadoop集群）。

S3

在 `conf/druid/cluster/_common/common.runtime.properties` 中，

- 在 `druid.extension.loadList` 配置项中增加"druid-s3-extensions"扩展
- 注释掉配置文件中用于本地存储的"Deep Storage"和"Indexing service logs"
- 打开配置文件中关于"For S3"部分中"Deep Storage"和"Indexing service logs"的配置

上述操作之后，您将看到以下的变化：

```
druid.extensions.loadList=["druid-s3-extensions"]

#druid.storage.type=local
#druid.storage.storageDirectory=var/druid/segments

druid.storage.type=s3
druid.storage.bucket=your-bucket
druid.storage.baseKey=druid/segments
druid.s3.accessKey=...
druid.s3.secretKey=...

#druid.indexer.logs.type=file
#druid.indexer.logs.directory=var/druid/indexing-logs

druid.indexer.logs.type=s3
druid.indexer.logs.s3Bucket=your-bucket
druid.indexer.logs.s3Prefix=druid/indexing-logs
```

更多信息可以看S3扩展部分的文档。

HDFS

在 `conf/druid/cluster/_common/common.runtime.properties` 中，

- 在 `druid.extension.loadList` 配置项中增加"druid-hdfs-storage"扩展
- 注释掉配置文件中用于本地存储的"Deep Storage"和"Indexing service logs"
- 打开配置文件中关于"For HDFS"部分中"Deep Storage"和"Indexing service logs"的配置

上述操作之后，您将看到以下的变化：

```
druid.extensions.loadList=["druid-hdfs-storage"]

#druid.storage.type=local
#druid.storage.storageDirectory=var/druid/segments

druid.storage.type=hdfs
druid.storage.storageDirectory=/druid/segments

#druid.indexer.logs.type=file
#druid.indexer.logs.directory=var/druid/indexing-logs

druid.indexer.logs.type=hdfs
druid.indexer.logs.directory=/druid/indexing-logs
```

同时：

- 需要将Hadoop的配置文件（core-site.xml, hdfs-site.xml, yarn-site.xml, mapred-site.xml）放置在Druid进程的classpath中，可以将他们拷贝到 `conf/druid/cluster/_common` 目录中

更多信息可以看HDFS扩展部分的文档。

Hadoop连接配置

如果要从Hadoop集群加载数据，那么此时应对Druid做如下配置：

- 在 `conf/druid/cluster/_common/common.runtime.properties` 文件中更新 `druid.indexer.task.hadoopWorkingPath` 配置项，将其更新为您期望的一个用

于临时文件存储的HDFS路径。 通常会配置为 `druid.indexer.task.hadoopWorkingPath=/tmp/druid-indexing`

- 需要将Hadoop的配置文件（core-site.xml, hdfs-site.xml, yarn-site.xml, mapred-site.xml）放置在Druid进程的classpath中，可以将他们拷贝到 `conf/druid/cluster/_common` 目录中

请注意，您无需为了可以从Hadoop加载数据而使用HDFS深度存储。例如，如果您的集群在Amazon Web Services上运行，即使您使用Hadoop或Elastic MapReduce加载数据，我们也建议使用S3进行深度存储。

更多信息可以看基于Hadoop的数据摄取部分的文档。

Zookeeper连接配置

在生产集群中，我们建议使用专用的ZK集群，该集群与Druid服务器分开部署。

在 `conf/druid/cluster/_common/common.runtime.properties` 中，将 `druid.zk.service.host` 设置为包含用逗号分隔的host：port对列表的连接字符串，每个对与ZK中的ZooKeeper服务器相对应。（例如"127.0.0.1:4545"或"127.0.0.1:3000,127.0.0.1:3001、127.0.0.1:3002"）

您也可以选择在Master服务上运行ZK，而不使用专用的ZK集群。如果这样做，我们建议部署3个Master服务，以便您具有ZK仲裁。

配置调整

从单服务器环境迁移部署

Master服务

如果您使用的是单服务器部署示例中的示例配置，则这些示例中将Coordinator和Overlord进程合并为一个合并的进程。

`conf/druid/cluster/master/coordinator-overlord` 下的示例配置同样合并了Coordinator和Overlord进程。

您可以将现有的 `coordinator-overlord` 配置从单服务器部署复制到 `conf/druid/cluster/master/coordinator-overlord`

Data服务

假设我们正在从一个32CPU和256GB内存的单服务器部署环境进行迁移，在老的环境中，Historical和MiddleManager使用了如下的配置：

Historical（单服务器）

```
druid.processing.buffer.sizeBytes=500000000
druid.processing.numMergeBuffers=8
druid.processing.numThreads=31
```

MiddleManager（单服务器）

```
druid.worker.capacity=8
druid.indexer.fork.property.druid.processing.numMergeBuffers=2
druid.indexer.fork.property.druid.processing.buffer.sizeBytes=100000000
druid.indexer.fork.property.druid.processing.numThreads=1
```

在集群部署中，我们选择一个分裂因子（假设为2），则部署2个16CPU和128GB内存的Data服务，各项的调整如下：

Historical

- `druid.processing.numThreads` 设置为新硬件的（ `CPU核数 - 1` ）
- `druid.processing.numMergeBuffers` 使用分裂因子去除单服务部署环境的值
- `druid.processing.buffer.sizeBytes` 该值保持不变

MiddleManager:

- `druid.worker.capacity`：使用分裂因子去除单服务部署环境的值
- `druid.indexer.fork.property.druid.processing.numMergeBuffers`：该值保持不变
- `druid.indexer.fork.property.druid.processing.buffer.sizeBytes`：该值保持不变
- `druid.indexer.fork.property.druid.processing.numThreads`：该值保持不变

调整后的结果配置如下：

新的Historical(2 Data服务器)

```
druid.processing.buffer.sizeBytes=500000000
druid.processing.numMergeBuffers=8
druid.processing.numThreads=31
```

新的MiddleManager（2 Data服务器）

```
druid.worker.capacity=4
druid.indexer.fork.property.druid.processing.numMergeBuffers=2
druid.indexer.fork.property.druid.processing.buffer.sizeBytes=100000000
druid.indexer.fork.property.druid.processing.numThreads=1
```

Query服务

您可以将现有的Broker和Router配置复制到 `conf/druid/cluster/query` 下的目录中，无需进行任何修改.

首次部署

如果您正在使用如下描述的示例集群规格：

- 1 Master 服务器(m5.2xlarge)
- 2 Data 服务器(i3.4xlarge)
- 1 Query 服务器(m5.2xlarge)

`conf/druid/cluster` 下的配置已经为此硬件确定了，一般情况下您无需做进一步的修改。

如果您选择了其他硬件，则基本的集群调整指南可以帮助您调整配置大小。

开启端口(如果使用了防火墙)

如果您正在使用防火墙或其他仅允许特定端口上流量准入的系统，请在以下端口上允许入站连接：

Master服务

- 1527（Derby元数据存储，如果您正在使用一个像MySQL或者PostgreSQL的分离的元数据存储则不需要）
- 2181（Zookeeper，如果使用了独立的ZK集群则不需要）
- 8081（Coordinator）
- 8090（Overlord）

Data服务

- 8083（Historical）
- 8091，8100-8199（Druid MiddleManager，如果 `druid.worker.capacity` 参数设置较大的话，则需要更多高于8199的端口）

Query服务

- 8082（Broker）
- 8088（Router，如果使用了）

> [!WARNING] 在生产中，我们建议将ZooKeeper和元数据存储部署在其专用硬件上，而不是在Master服务器上。

启动Master服务

将Druid发行版和您编辑的配置文件复制到Master服务器上。

如果您一直在本地计算机上编辑配置，则可以使用rsync复制它们：

```
rsync -az apache-druid-0.17.0/ MASTER_SERVER:apache-druid-0.17.0/
```

不带Zookeeper启动

在发行版根目录中，运行以下命令以启动Master服务：

```
bin/start-cluster-master-no-zk-server
```

带Zookeeper启动

如果计划在Master服务器上运行ZK，请首先更新 `conf/zoo.cfg` 以标识您计划如何运行ZK，然后，您可以使用以下命令与ZK一起启动Master服务进程：

```
bin/start-cluster-master-with-zk-server
```

> [!WARNING] 在生产中，我们建议将ZooKeeper运行在其专用硬件上。

启动Data服务

将Druid发行版和您编辑的配置文件复制到您的Data服务器。

在发行版根目录中，运行以下命令以启动Data服务：

```
bin/start-cluster-data-server
```

您可以在需要的时候增加更多的Data服务器。

> [!WARNING] 对于具有复杂资源分配需求的集群，您可以将Historical和MiddleManager分开部署，并分别扩容组件。这也使您能够利用Druid的内置MiddleManager自动伸缩功能。

启动Query服务

将Druid发行版和您编辑的配置文件复制到您的Query服务器。

在发行版根目录中，运行以下命令以启动Query服务：

```
bin/start-cluster-query-server
```

您可以根据查询负载添加更多查询服务器。 如果增加了查询服务器的数量，请确保按照基本集群调优指南中的说明调整Historical和Task上的连接池。

加载数据

恭喜，您现在有了Druid集群！下一步是根据使用场景来了解将数据加载到Druid的推荐方法。

了解有关加载数据的更多信息。

加载本地文件

本教程演示了如何使用Apache Druid的本地批量数据摄取来执行批文件加载。

在本教程中，我们假设您已经按照快速入门中的规范下载了Druid，并使用 `micro-quickstart` 单机配置使其在本地计算机上运行。您不需要加载任何数据。

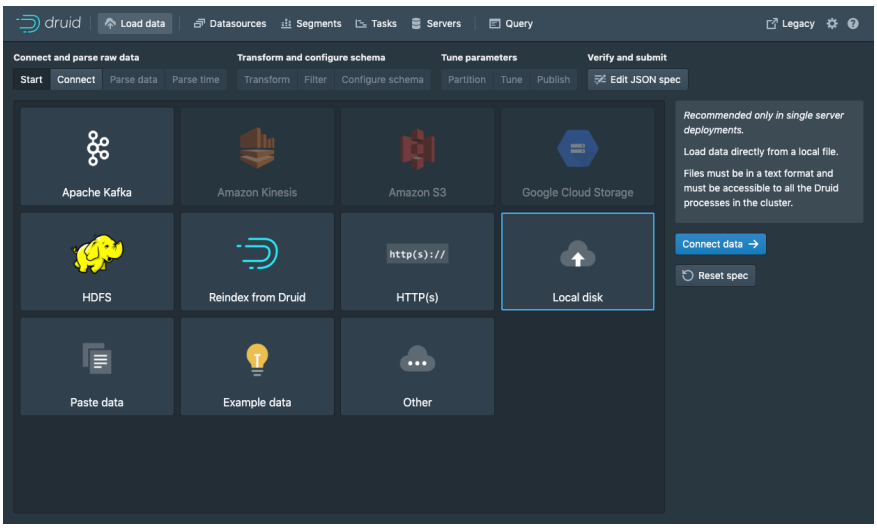
Druid的数据加载是通过向Overlord服务提交*摄取任务规范*来启动。对于本教程，我们将加载Wikipedia页面示例编辑数据。

*数据摄取任务规范*可以手动编写，也可以通过Druid控制台里内置的数据加载器编写。数据加载器可以通过采样摄入的数据并配置各种摄入参数来帮助您生成*摄取任务规范*。数据加载器当前仅支持本地批处理提取（将来的版本中将提供对流的支持，包括存储在Apache Kafka和AWS Kinesis中的数据）。目前只能通过手动书写摄入规范来进行流式摄入。

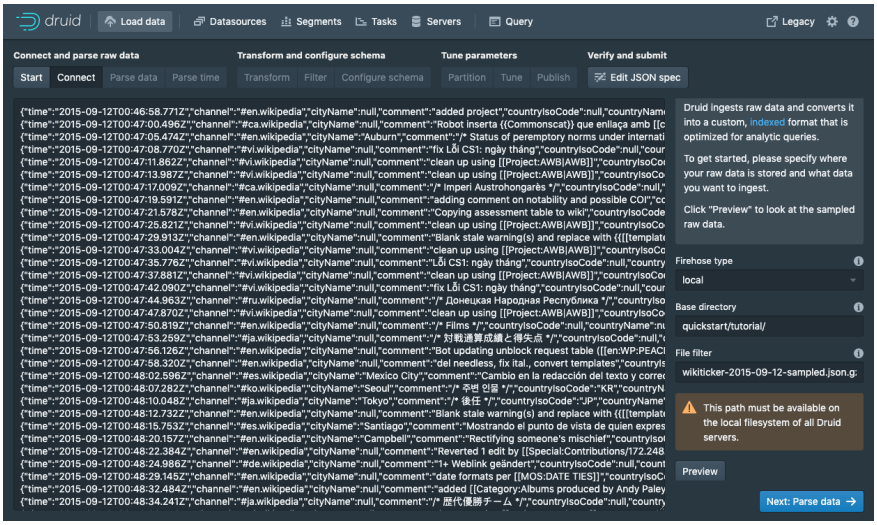
我们提供了2015年9月12日起对Wikipedia进行编辑的示例，以帮助您入门。

使用Data Loader来加载数据

浏览器访问 localhost:8888 然后点击控制台中的 `Load data`



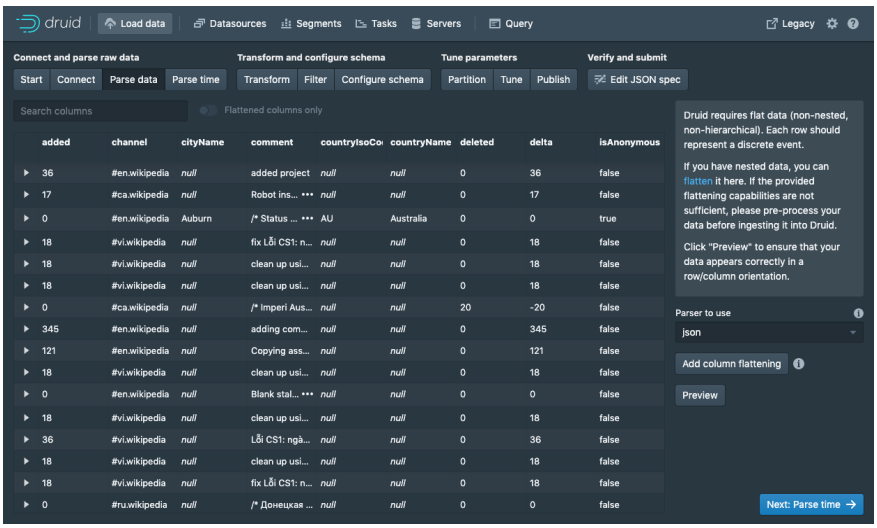
选择 `Local disk` 然后点击 `Connect data`



在 `Base directory` 中输入 `quickstart/tutorial/`，在 `File filter` 中输入 `wikiticker-2015-09-12-sampled.json.gz` 。 `Base directory` 和 `File filter` 分开是因为可能需要同时从多个文件中摄取数据。

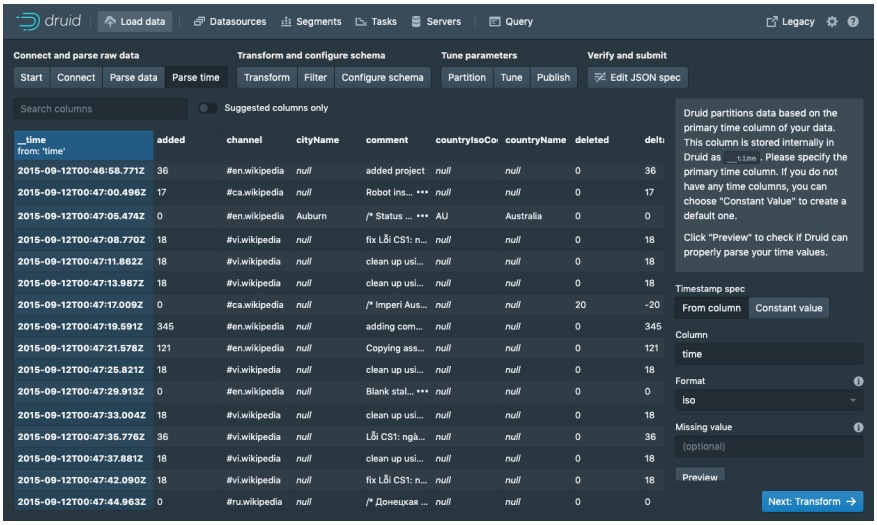
点击 `Preview`，确保您看到的数据是正确的。

数据定位后，您可以点击"Next: Parse data"来进入下一步。



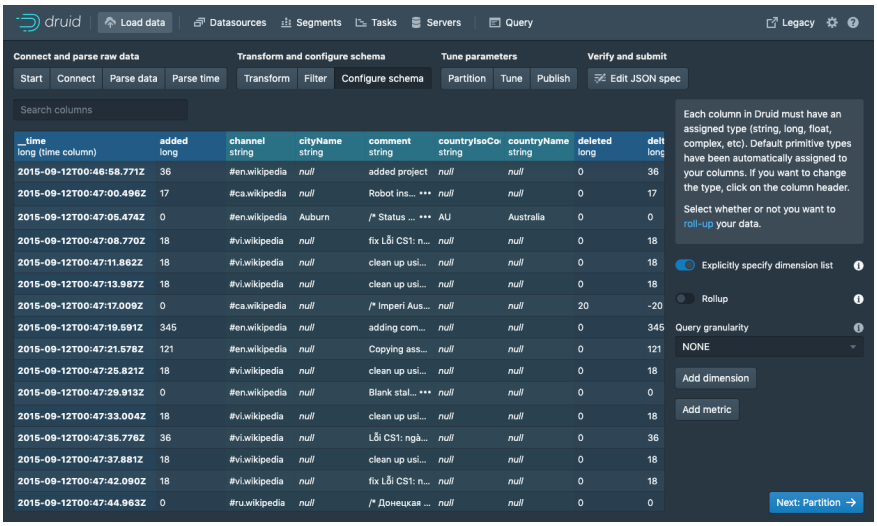
数据加载器将尝试自动为数据确定正确的解析器。在这种情况下，它将成功确定 `json` 。可以随意使用不同的解析器选项来预览Druid如何解析您的数据。

`json` 选择器被选中后，点击 `Next: Parse time` 进入下一步来决定您的主时间列。



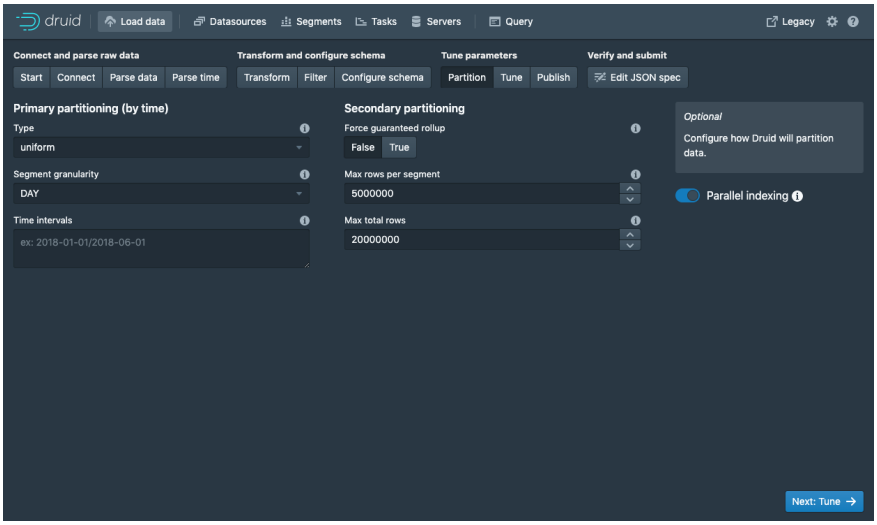
Druid的体系结构需要一个主时间列（内部存储为名为__time的列）。如果您的数据中没有时间戳，请选择 `固定值（Constant Value）` 。在我们的示例中，数据加载器将确定原始数据中的时间列是唯一可用作主时间列的候选者。

点击"Next:..."两次完成 `Transform` 和 `Filter` 步骤。您无需在这些步骤中输入任何内容，因为使用摄取时间变换和过滤器不在本教程范围内。



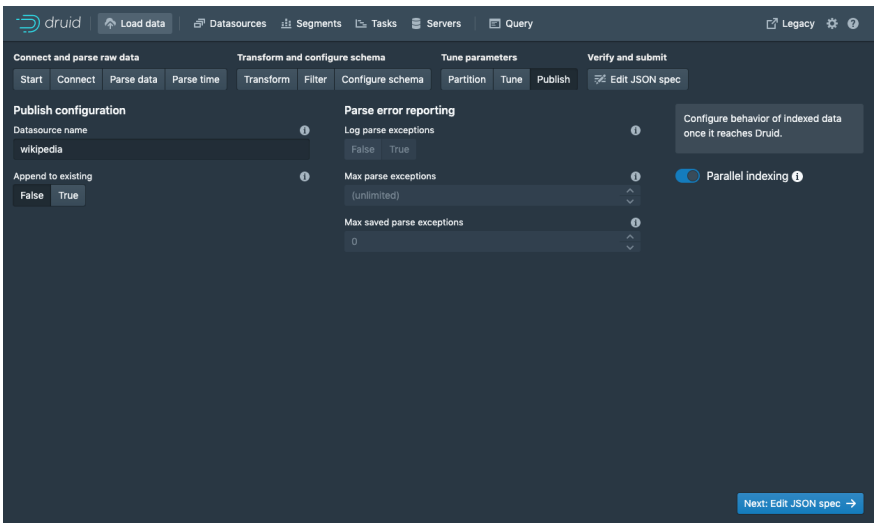
在 `Configure schema` 步骤中，您可以配置将哪些维度和指标摄入到Druid中，这些正是数据在被Druid中摄取后出现的样子。 由于我们的数据集非常小，关掉rollup、确认更改。

一旦对schema满意后，点击 `Next` 后进入 `Partition` 步骤，该步骤中可以调整数据如何划分为段文件的方式。

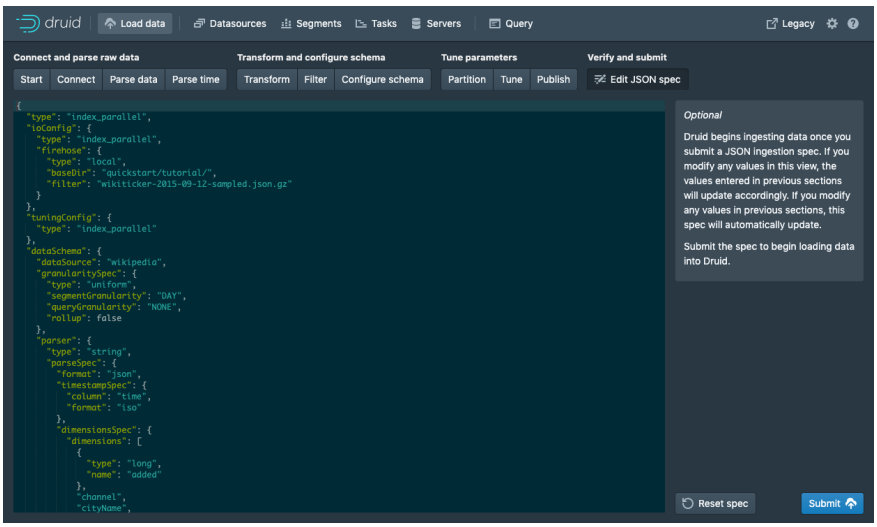


在这里，您可以调整如何在Druid中将数据拆分为多个段。 由于这是一个很小的数据集，因此在此步骤中无需进行任何调整。

点击完成 `Tune` 步骤，进入到 `Publish` 步。

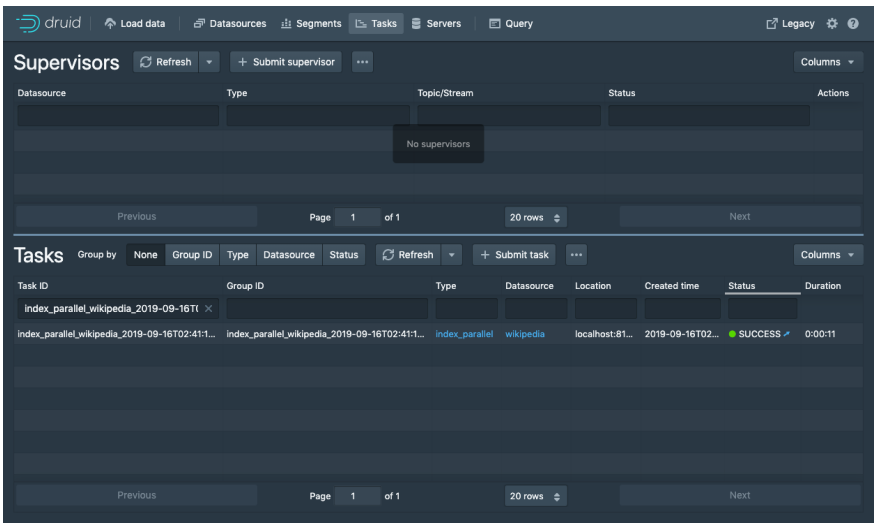


在 `Publish` 步骤中，我们可以指定Druid中的数据源名称,让我们将此数据源命名为 `Wikipedia` 。最后，单击 `Next` 来查看您的摄取规范。



这就是您构建的规范，为了查看更改将如何更新规范是可以随意返回之前的步骤中进行更改，同样，您也可以直接编辑规范，并在前面的步骤中看到它。

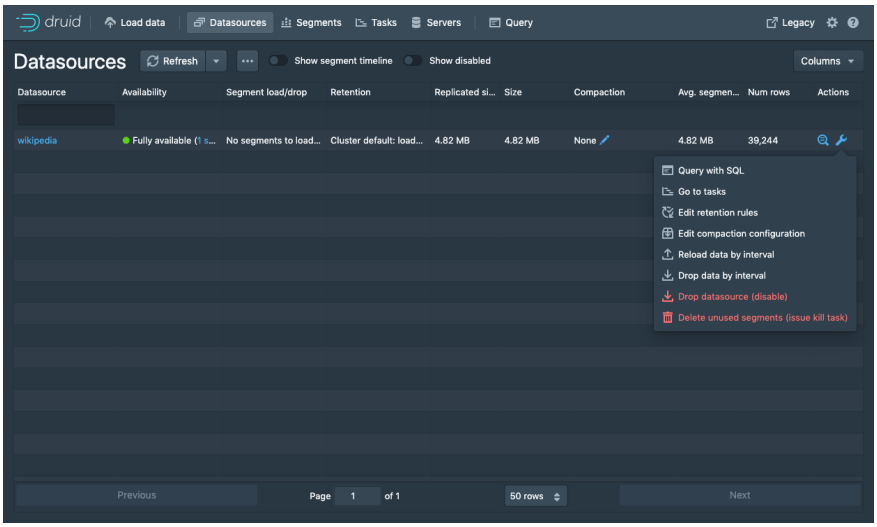
对摄取规范感到满意后，请单击 `Submit` ，然后将创建一个数据摄取任务。



您可以进入任务视图，重点关注新创建的任务。任务视图设置为自动刷新，请等待任务成功。

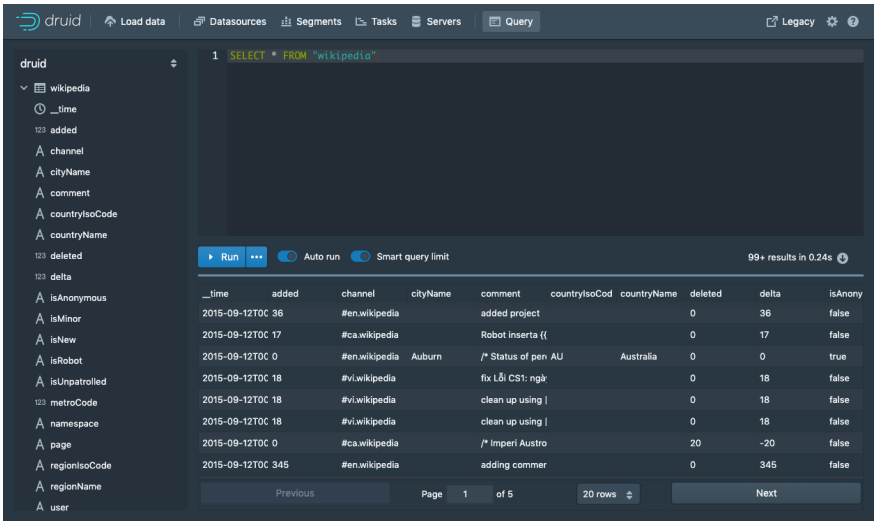
当一项任务成功完成时，意味着它建立了一个或多个段，这些段现在将由Data服务器接收。

从标题导航到 `Datasources` 视图。



等待直到您的数据源（Wikipedia）出现,加载段时可能需要几秒钟。

一旦看到绿色（完全可用）圆圈，就可以查询数据源。此时，您可以转到 `Query` 视图以对数据源运行SQL查询。



运行 `SELECT * FROM wikipedia` 查询可以看到详细的结果。

查看查询教程以对新加载的数据运行一些示例查询。

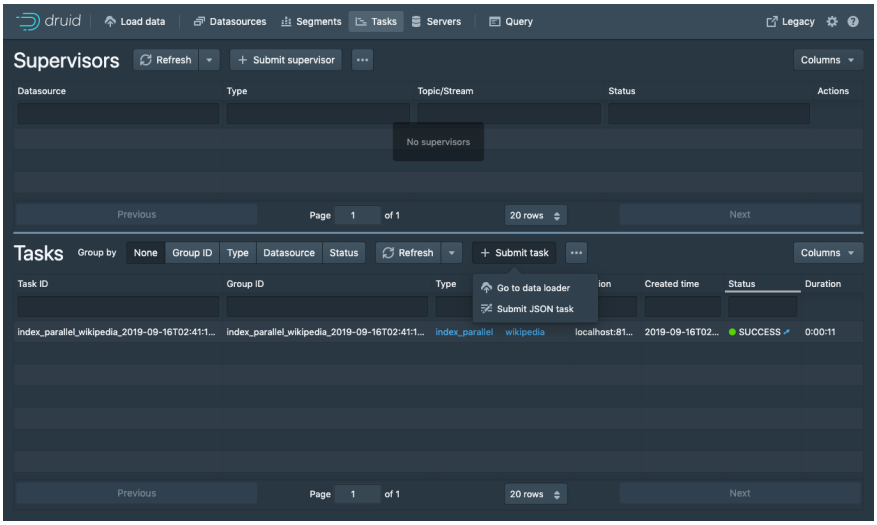
使用spec加载数据（通过控制台）

Druid的安装包中在 `quickstart/tutorial/wikipedia-index.json` 文件中包含了一个本地批摄入任务规范的示例。 为了方便我们在这里展示出来，该规范已经配置好读取 `quickstart/tutorial/wikiticker-2015-09-12-sampled.json.gz` 输入文件。

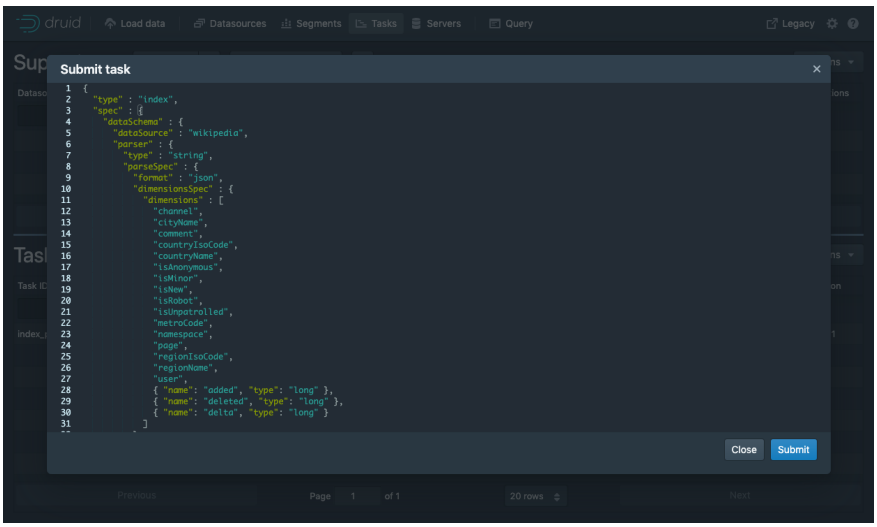
```
{
  "type" : "index_parallel",
  "spec" : {
    "dataSchema" : {
      "dataSource" : "wikipedia",
      "dimensionsSpec" : {
        "dimensions" : [
          "channel",
          "cityName",
          "comment",
          "countryIsoCode",
          "countryName",
          "isAnonymous",
          "isMinor",
          "isNew",
          "isRobot",
          "isUnpatrolled",
          "metroCode",
          "namespace",
          "page",
          "regionIsoCode",
          "regionName",
          "user",
          { "name": "added", "type": "long" },
          { "name": "deleted", "type": "long" },
          { "name": "delta", "type": "long" }
        ]
      },
      "timestampSpec": {
        "column": "time",
        "format": "iso"
      },
      "metricsSpec" : [],
      "granularitySpec" : {
        "type" : "uniform",
        "segmentGranularity" : "day",
        "queryGranularity" : "none",
        "intervals" : ["2015-09-12/2015-09-13"],
        "rollup" : false
      }
    },
    "ioConfig" : {
      "type" : "index_parallel",
      "inputSource" : {
        "type" : "local",
        "baseDir" : "quickstart/tutorial/",
        "filter" : "wikiticker-2015-09-12-sampled.json.gz"
      },
      "inputFormat" :  {
        "type": "json"
      },
      "appendToExisting" : false
    },
    "tuningConfig" : {
      "type" : "index_parallel",
      "maxRowsPerSegment" : 5000000,
      "maxRowsInMemory" : 25000
    }
  }
}
```

这份规范将创建一个命名为"wikipedia"的数据源。

在"Tasks"页面，点击 `Submit task` 后选择 `Raw JSON task`，



然后会在页面中弹出任务规范输入框，您可以在上面粘贴规范



提交任务规范后，您可以按照上述相同的规范等待数据加载然后查询。

使用spec加载数据（通过命令行）

为了方便，在Druid的软件包中提供了一个批摄取的帮助脚本 `bin/post-index-task`

该脚本会将数据摄取任务发布到Druid Overlord并轮询Druid，直到可以查询数据为止。

在Druid根目录运行以下命令：

```
bin/post-index-task --file quickstart/tutorial/wikipedia-index.json --url http
```

可以看到以下的输出：

```
Beginning indexing data for wikipedia
Task started: index_wikipedia_2018-07-27T06:37:44.323Z
Task log:     http://localhost:8081/druid/indexer/v1/task/index_wikipedia_2018
Task status:  http://localhost:8081/druid/indexer/v1/task/index_wikipedia_2018
Task index_wikipedia_2018-07-27T06:37:44.323Z still running...
Task index_wikipedia_2018-07-27T06:37:44.323Z still running...
Task finished with status: SUCCESS
Completed indexing data for wikipedia. Now loading indexed data onto the clust
wikipedia loading complete!  You may now query your data
```

提交任务规范后，您可以按照上述相同的规范等待数据加载然后查询。

不使用脚本来加载数据

我们简短地讨论一下如何在不使用脚本的情况下提交数据摄取任务，您将不需要运行这些命令。

要提交任务，可以在一个新的终端中通过以下方式提交任务到Druid：

```
curl -X 'POST' -H 'Content-Type:application/json' -d @quickstart/tutorial/wiki
```

当任务提交成功后会打印出来任务的ID

```
{"task":"index_wikipedia_2018-06-09T21:30:32.802Z"}
```

您可以如上所述从控制台监视此任务的状态

查询已加载数据

加载数据后，请按照查询教程的操作，对新加载的数据执行一些示例查询。

数据清理

如果您希望阅读其他任何入门教程，则需要关闭集群并通过删除druid软件包下的 `var` 目录的内容来重置集群状态，因为其他教程将写入相同的"wikipedia"数据源。

更多信息

更多关于加载批数据的信息可以查看原生批摄取文档

教程：从Kafka中加载流式数据

入门

本教程演示了如何使用Druid的Kafka索引服务将数据从Kafka流加载到Apache Druid中。

在本教程中，我们假设您已经按照快速入门中的说明下载了Druid并使用 `micro-quickstart` 单机配置使其在本地计算机上运行。您不需要加载任何数据。

下载并启动Kafka

Apache Kafka是一种高吞吐量消息总线，可与Druid很好地配合使用。在本教程中，我们将使用Kafka2.1.0。要下载Kafka，请在终端中执行以下命令：

```
curl -O https://archive.apache.org/dist/kafka/2.1.0/kafka_2.12-2.1.0.tgz
tar -xzf kafka_2.12-2.1.0.tgz
cd kafka_2.12-2.1.0
```

通过在终端中运行以下命令来启动一个Kafka Broker：

```
./bin/kafka-server-start.sh config/server.properties
```

执行以下命令来创建一个我们用来发送数据的Kafka主题，称为"*wikipedia*":

```
./bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor
```

发送数据到Kafka

让我们为该主题启动一个生产者并发送一些数据

在Druid目录下，运行下边的命令：

```
cd quickstart/tutorial
gunzip -c wikiticker-2015-09-12-sampled.json.gz > wikiticker-2015-09-12-sample
```

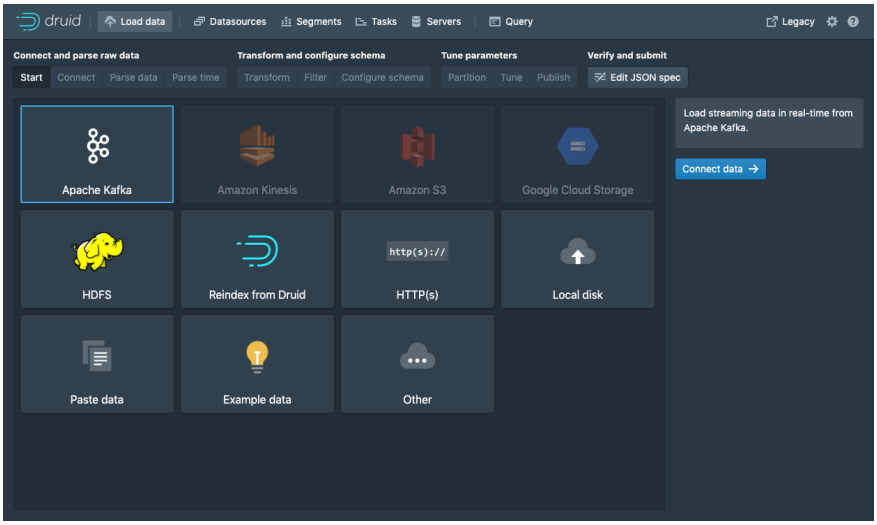
在Kafka目录下，运行以下命令，{PATH_TO_DRUID}替换为Druid目录的路径：

```
export KAFKA_OPTS="-Dfile.encoding=UTF-8"
./bin/kafka-console-producer.sh --broker-list localhost:9092 --topic wikipedia
```

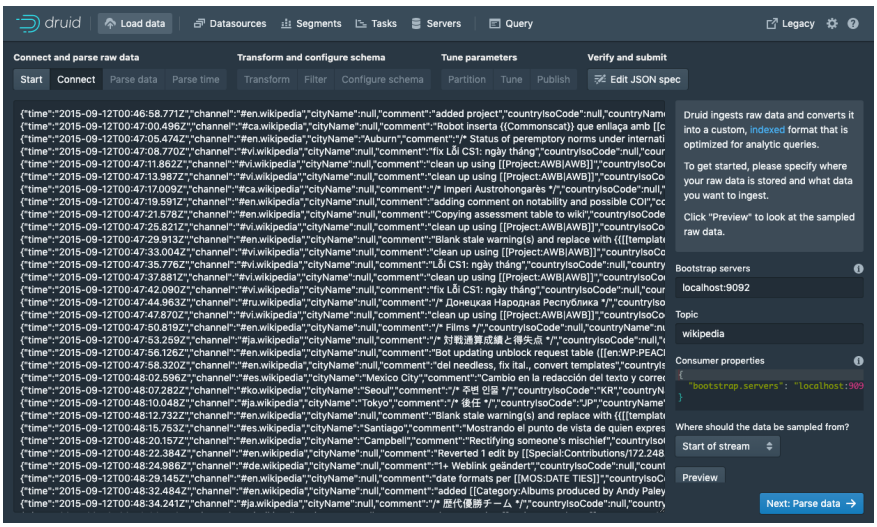
上一个命令将示例事件发布到名称为*wikipedia*的Kafka主题。现在，我们将使用Druid的Kafka索引服务从新创建的主题中提取消息。

使用数据加载器（Data Loader）

浏览器访问 localhost:8888 然后点击控制台中的 `Load data`



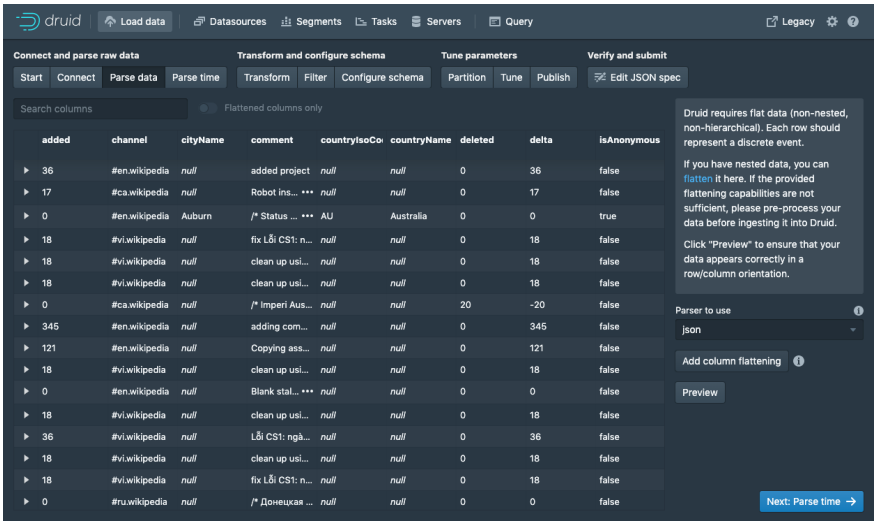
选择 `Apache Kafka` 然后点击 `Connect data`



在 `Bootstrap servers` 输入 `localhost:9092`，在 `Topic` 输入 `wikipedia`

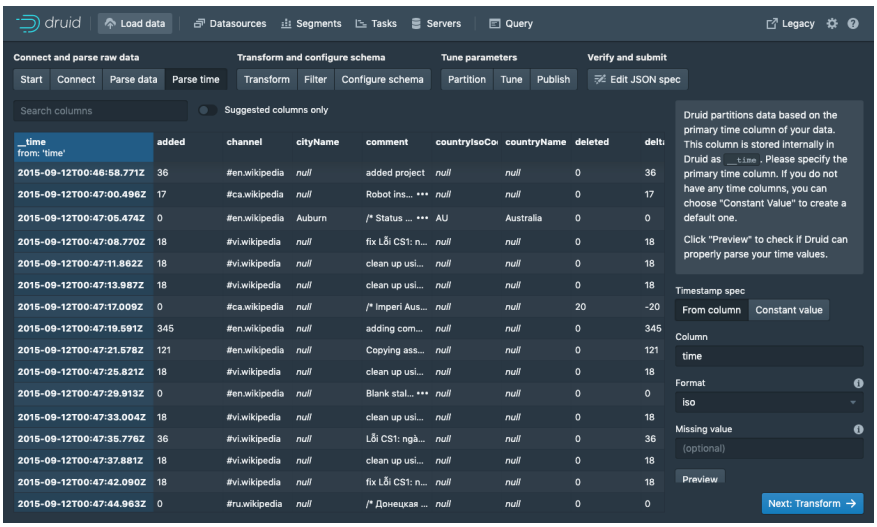
点击 `Preview` 后确保您看到的数据是正确的

数据定位后，您可以点击"Next: Parse data"来进入下一步。



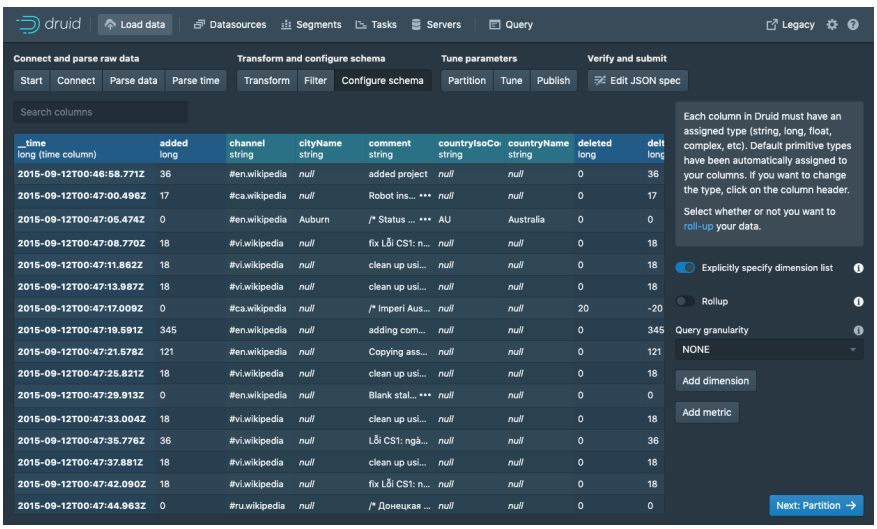
数据加载器将尝试自动为数据确定正确的解析器。在这种情况下，它将成功确定 `json` 。可以随意使用不同的解析器选项来预览Druid如何解析您的数据。

`json` 选择器被选中后，点击 `Next: Parse time` 进入下一步来决定您的主时间列。



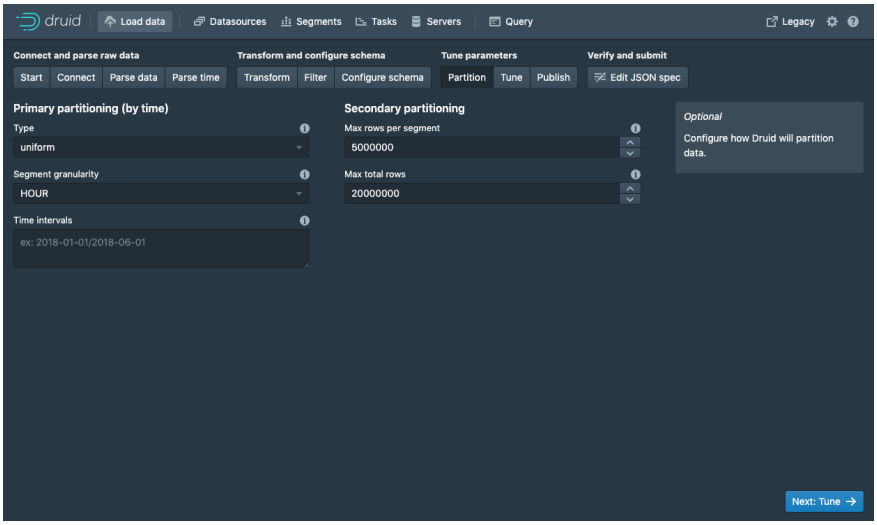
Druid的体系结构需要一个主时间列（内部存储为名为__time的列）。如果您的数据中没有时间戳，请选择 `固定值（Constant Value）` 。在我们的示例中，数据加载器将确定原始数据中的时间列是唯一可用作主时间列的候选者。

点击"Next:..."两次完成 `Transform` 和 `Filter` 步骤。您无需在这些步骤中输入任何内容，因为使用摄取时间变换和过滤器不在本教程范围内。



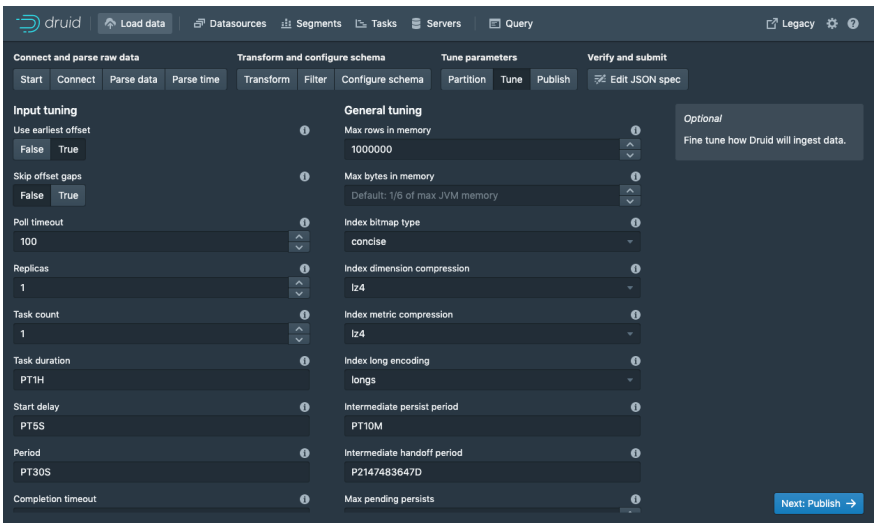
在 `Configure schema` 步骤中，您可以配置将哪些维度和指标摄入到Druid中，这些正是数据在被Druid中摄取后出现的样子。 由于我们的数据集非常小，关掉rollup、确认更改。

一旦对schema满意后，点击 `Next` 后进入 `Partition` 步骤，该步骤中可以调整数据如何划分为段文件的方式。

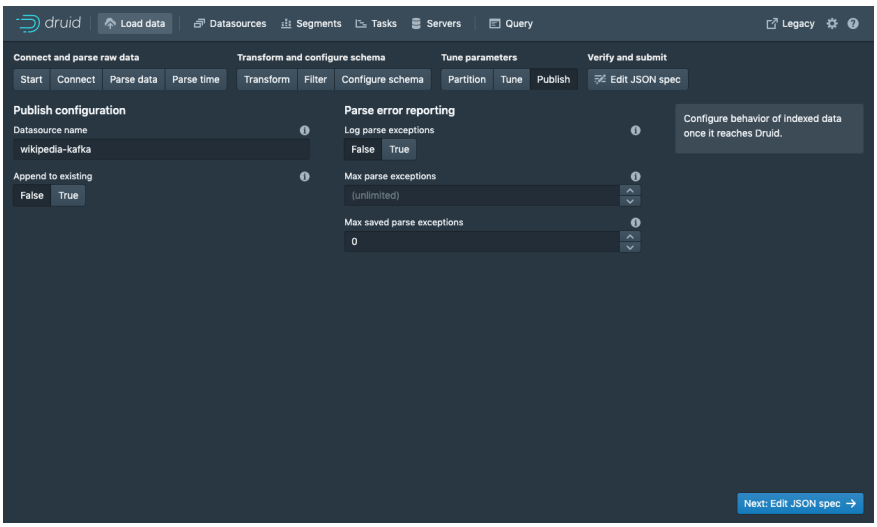


在这里，您可以调整如何在Druid中将数据拆分为多个段。 由于这是一个很小的数据集，因此在此步骤中无需进行任何调整。

点击完成 `Tune` 步骤。

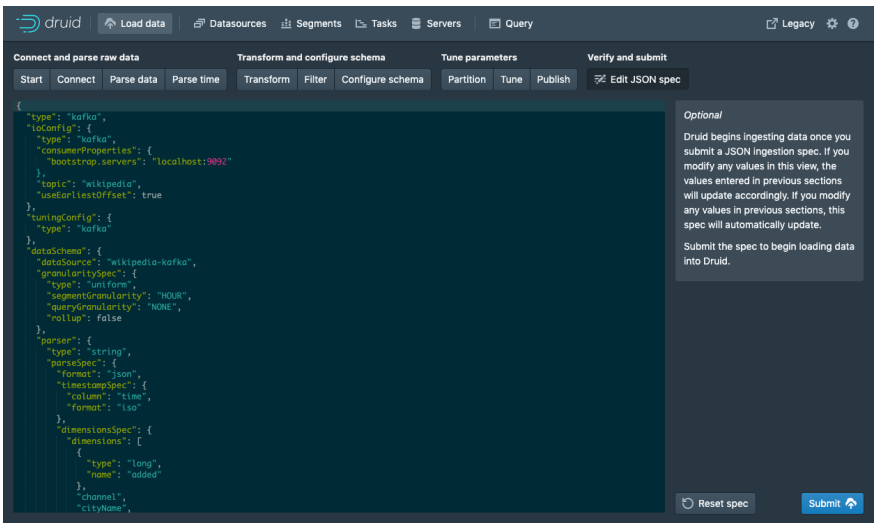


在 `Tune` 步骤中，将 `Use earliest offset` 设置为 `True` *非常重要*，因为我们需要从流的开始位置消费数据。 其他没有任何需要更改的地方，进入到 `Publish` 步



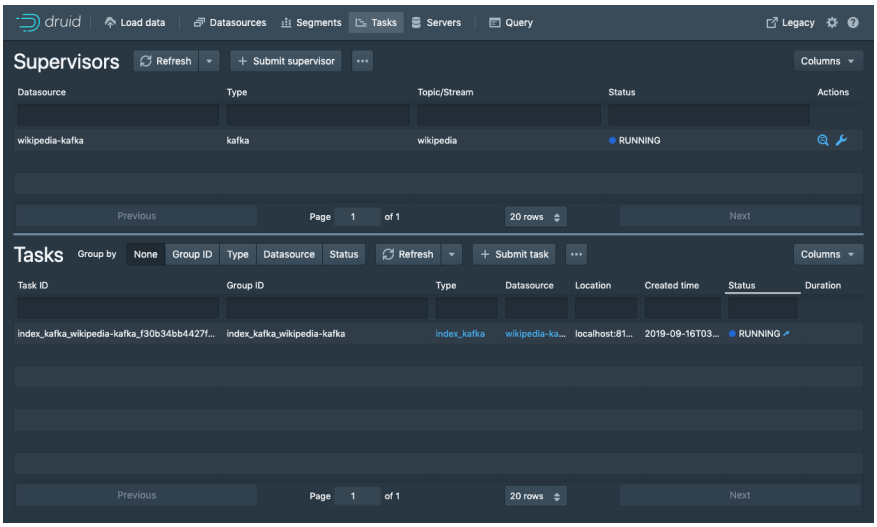
我们将该数据源命名为 `wikipedia-kafka`

最后点击 `Next` 预览摄入规范：



这就是您构建的规范，为了查看更改将如何更新规范是可以随意返回之前的步骤中进行更改，同样，您也可以直接编辑规范，并在前面的步骤中看到它。

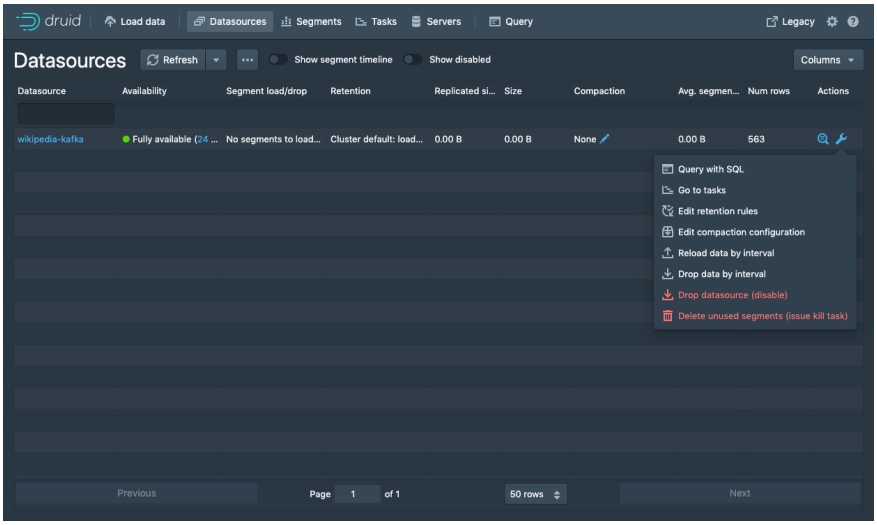
对摄取规范感到满意后，请单击 `Submit`，然后将创建一个数据摄取任务



您可以进入任务视图，重点关注新创建的supervisor。任务视图设置为自动刷新，请等待直到Supervisor启动了一个任务。

当一项任务开始运行时，它将开始处理其摄入的数据。

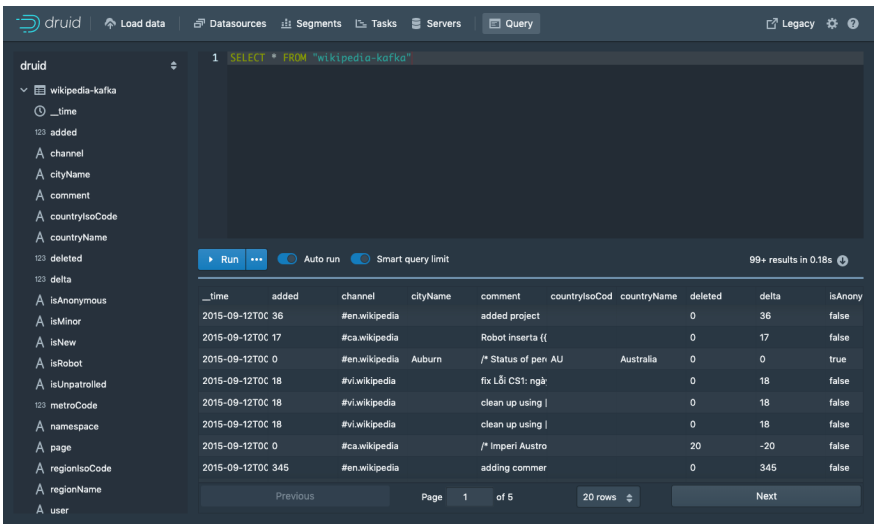
从标题导航到 `Datasources` 视图。



当 `wikipedia-kafka` 数据源出现在这儿的时候就可以被查询了。

> ![TIPS] 如果过了几分钟之后数据源还是没有出现在这里，可能是在 `Tune` 步骤中没有设置为从流的开始进行消费数据

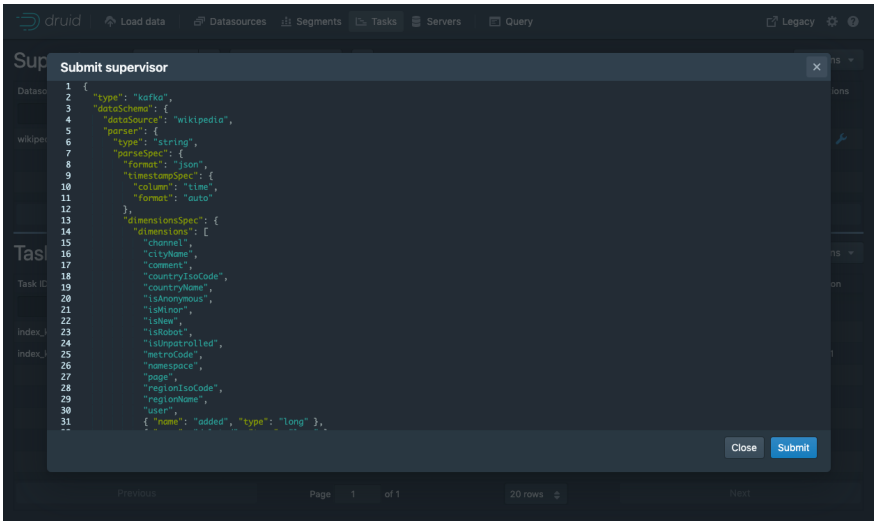
此时，就可以在 `Query` 视图中运行SQL查询了，因为这是一个小的数据集，你可以简单的运行 `SELECT * FROM "wikipedia-kafka"` 来查询结果。



查看查询教程以对新加载的数据运行一些示例查询。

通过控制台提交supervisor

在控制台中点击 `Submit supervisor` 打开提交supervisor对话框：



粘贴以下规范后点击 `Submit`

```
{
  "type": "kafka",
  "spec" : {
    "dataSchema": {
      "dataSource": "wikipedia",
      "timestampSpec": {
        "column": "time",
        "format": "auto"
      },
      "dimensionsSpec": {
        "dimensions": [
          "channel",
          "cityName",
          "comment",
          "countryIsoCode",
          "countryName",
          "isAnonymous",
          "isMinor",
          "isNew",
          "isRobot",
          "isUnpatrolled",
          "metroCode",
          "namespace",
          "page",
          "regionIsoCode",
          "regionName",
          "user",
          { "name": "added", "type": "long" },
          { "name": "deleted", "type": "long" },
          { "name": "delta", "type": "long" }
        ]
      },
      "metricsSpec" : [],
      "granularitySpec": {
        "type": "uniform",
        "segmentGranularity": "DAY",
        "queryGranularity": "NONE",
        "rollup": false
      }
    },
    "tuningConfig": {
      "type": "kafka",
      "reportParseExceptions": false
    },
    "ioConfig": {
      "topic": "wikipedia",
      "inputFormat": {
        "type": "json"
      },
      "replicas": 2,
      "taskDuration": "PT10M",
      "completionTimeout": "PT20M",
      "consumerProperties": {
        "bootstrap.servers": "localhost:9092"
      }
    }
  }
}
```

这将启动supervisor，该supervisor继而产生一些任务，这些任务将开始监听传入的数据。

直接提交supervisor

为了直接启动服务，我们可以在Druid的根目录下运行以下命令来提交一个supervisor规范到Druid Overlord中

```
curl -XPOST -H'Content-Type: application/json' -d @quickstart/tutorial/wikiped
```

如果supervisor被成功创建后，将会返回一个supervisor的ID，在本例中看到的是 `{"id":"wikipedia"}`

更详细的信息可以查看Druid Kafka索引服务文档

您可以在Druid控制台中查看现有的supervisors和tasks

数据查询

数据被发送到Kafka流之后，立刻就可以被查询了。

按照查询教程的操作，对新加载的数据执行一些示例查询

清理数据

如果您希望阅读其他任何入门教程，则需要关闭集群并通过删除druid软件包下的 `var` 目录的内容来重置集群状态，因为其他教程将写入相同的"wikipedia"数据源。

进一步阅读

更多关于从Kafka流加载数据的信息，可以查看Druid Kafka索引服务文档

使用Apache Hadoop加载批数据

本教程向您展示如何使用远程Hadoop集群将数据文件加载到Apache Druid中。

对于本教程，我们假设您已经使用快速入门中所述的 `micro-quickstart` 单机配置完成了前边的批处理摄取指南。

安装Docker

本教程要求将Docker安装在教程计算机上。

Docker安装完成后，请继续执行本教程中的后续步骤

构建Hadoop Docker镜像

在本教程中，我们为Hadoop 2.8.5集群提供了一个Dockerfile，我们将使用它运行批处理索引任务。

该Dockerfile和相关文件位于 `quickstart/tutorial/hadoop/docker` 。

从apache-druid-0.17.0软件包根目录中，运行以下命令以构建名为"druid-hadoop-demo"的Docker镜像，其版本标签为"2.8.5"：

```
cd quickstart/tutorial/hadoop/docker
docker build -t druid-hadoop-demo:2.8.5 .
```

该命令运行后开始构建Hadoop镜像。镜像构建完成后，可以在控制台中看到 `Successfully tagged druid-hadoop-demo:2.8.5` 的信息。

安装Hadoop Docker集群

创建临时共享目录

我们需要一个共享目录以便于主机和Hadoop容器之间进行传输文件

我们在 `/tmp` 下创建一些文件夹，稍后我们在启动Hadoop容器时会使用到它们：

```
mkdir -p /tmp/shared
mkdir -p /tmp/shared/hadoop_xml
```

配置 /etc/hosts

在主机的 `/etc/hosts` 中增加以下入口：

```
127.0.0.1 druid-hadoop-demo
```

启动Hadoop容器

在 `/tmp/shared` 文件夹被创建和 `/etc/hosts` 入口被添加后，运行以下命令来启动Hadoop容器：

```
docker run -it  -h druid-hadoop-demo --name druid-hadoop-demo -p 2049:2049 -p
```

容器启动后，您的终端将连接到容器内运行的bash shell：

```
Starting sshd:                                             [  OK  ]
18/07/26 17:27:15 WARN util.NativeCodeLoader: Unable to load native-hadoop lib
Starting namenodes on [druid-hadoop-demo]
druid-hadoop-demo: starting namenode, logging to /usr/local/hadoop/logs/hadoop
localhost: starting datanode, logging to /usr/local/hadoop/logs/hadoop-root-da
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /usr/local/hadoop/logs/hadoop-
18/07/26 17:27:31 WARN util.NativeCodeLoader: Unable to load native-hadoop lib
starting yarn daemons
starting resourcemanager, logging to /usr/local/hadoop/logs/yarn--resourcemana
localhost: starting nodemanager, logging to /usr/local/hadoop/logs/yarn-root-n
starting historyserver, logging to /usr/local/hadoop/logs/mapred--historyserve
bash-4.1#
```

`Unable to load native-hadoop library for your platform... using builtin-java classes where applicable` 这个信息可以安全地忽略掉。

进入Hadoop容器shell

运行下边命令打开Hadoop容器的另一个shell：

```
docker exec -it druid-hadoop-demo bash
```

拷贝数据到Hadoop容器

从apache-druid-0.17.0安装包的根目录拷贝 `quickstart/tutorial/wikiticker-2015-09-12-sampled.json.gz` 样例数据到共享文件夹

```
cp quickstart/tutorial/wikiticker-2015-09-12-sampled.json.gz /tmp/shared/wikit
```

设置Hadoop目录

在Hadoop容器shell中，运行以下命令来设置本次教程需要的HDFS目录，同时拷贝输入数据到HDFS上：

```
cd /usr/local/hadoop/bin
./hdfs dfs -mkdir /druid
./hdfs dfs -mkdir /druid/segments
./hdfs dfs -mkdir /quickstart
./hdfs dfs -chmod 777 /druid
./hdfs dfs -chmod 777 /druid/segments
./hdfs dfs -chmod 777 /quickstart
./hdfs dfs -chmod -R 777 /tmp
./hdfs dfs -chmod -R 777 /user
./hdfs dfs -put /shared/wikiticker-2015-09-12-sampled.json.gz /quickstart/wiki
```

如果在命令执行中遇到了namenode相关的错误，可能是因为Hadoop容器没有完成初始化，等一会儿后重新执行命令。

配置使用Hadoop的Druid

配置用于Hadoop批加载的Druid集群还需要额外的一些步骤。

拷贝Hadoop配置到Druid classpath

从Hadoop容器shell中，运行以下命令将Hadoop.xml配置文件拷贝到共享文件夹中：

```
cp /usr/local/hadoop/etc/hadoop/*.xml /shared/hadoop_xml
```

在宿主机上运行下边命令，其中{PATH_TO_DRUID}替换为Druid软件包的路径：

```
mkdir -p {PATH_TO_DRUID}/conf/druid/single-server/micro-quickstart/_common/had
cp /tmp/shared/hadoop_xml/*.xml {PATH_TO_DRUID}/conf/druid/single-server/micro
```

更新Druid段与日志的存储

在常用的文本编辑器中，打开 `conf/druid/single-server/micro-quickstart/_common/common.runtime.properties` 文件做如下修改：

禁用本地深度存储，启用HDFS深度存储

```
#
# Deep storage
#

# For local disk (only viable in a cluster if this is a network mount):
#druid.storage.type=local
#druid.storage.storageDirectory=var/druid/segments

# For HDFS:
druid.storage.type=hdfs
druid.storage.storageDirectory=/druid/segments
```

禁用本地日志存储，启动HDFS日志存储

```
#
# Indexing service logs
#

# For local disk (only viable in a cluster if this is a network mount):
#druid.indexer.logs.type=file
#druid.indexer.logs.directory=var/druid/indexing-logs

# For HDFS:
druid.indexer.logs.type=hdfs
druid.indexer.logs.directory=/druid/indexing-logs
```

重启Druid集群

Hadoop.xml文件拷贝到Druid集群、段和日志存储配置更新为HDFS后，Druid集群需要重启才可以让配置生效。

如果集群正在运行， `CTRL-C` 终止 `bin/start-micro-quickstart` 脚本，重新执行它使得Druid服务恢复。

加载批数据

我们提供了2015年9月12日起对Wikipedia编辑的示例数据，以帮助您入门。

要将数据加载到Druid中，可以提交指向该文件的*摄取任务*。我们已经包含了一个任务，该任务会加载存档中包含 `wikiticker-2015-09-12-sampled.json.gz` 文件。

通过以下命令进行提交 `wikipedia-index-hadoop.json` 任务：

```
bin/post-index-task --file quickstart/tutorial/wikipedia-index-hadoop.json --u
```

查询数据

加载数据后，请按照查询教程的操作，对新加载的数据执行一些示例查询.

清理数据

本教程只能与查询教程一起使用。

如果您打算完成其他任何教程，还需要：

- 关闭集群，通过删除Druid软件包中的 `var` 目录来重置集群状态
- 将 `conf/druid/single-server/micro-quickstart/_common/common.runtime.properties` 恢复深度存储与任务存储配置到本地类型
- 重启集群

这是必需的，因为其他摄取教程将写入相同的"wikipedia"数据源，并且以后的教程希望集群使用本地深度存储。

恢复配置示例：

```
#
# Deep storage
#

# For local disk (only viable in a cluster if this is a network mount):
druid.storage.type=local
druid.storage.storageDirectory=var/druid/segments

# For HDFS:
#druid.storage.type=hdfs
#druid.storage.storageDirectory=/druid/segments

#
# Indexing service logs
#

# For local disk (only viable in a cluster if this is a network mount):
druid.indexer.logs.type=file
druid.indexer.logs.directory=var/druid/indexing-logs

# For HDFS:
#druid.indexer.logs.type=hdfs
#druid.indexer.logs.directory=/druid/indexing-logs
```

进一步阅读

更多关于从Hadoop加载数据的信息，可以查看Druid Hadoop批量摄取文档

查询数据

本教程将以Druid SQL和Druid的原生查询格式的示例演示如何在Apache Druid中查询数据。

本教程假定您已经完成了摄取教程之一，因为我们将查询Wikipedia编辑样例数据。

- 加载本地文件
- 从Kafka加载数据
- 从Hadoop加载数据

Druid查询通过HTTP发送,Druid控制台包括一个视图，用于向Druid发出查询并很好地格式化结果。

Druid SQL查询

Druid支持SQL查询。

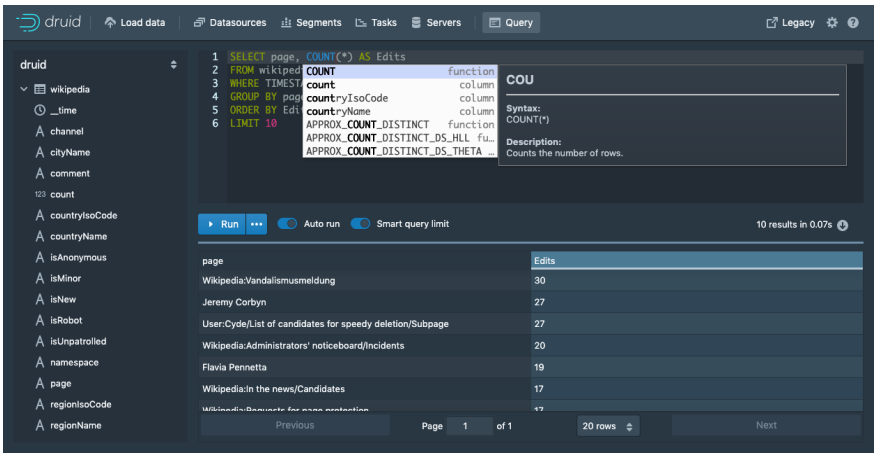
该查询检索了2015年9月12日被编辑最多的10个维基百科页面

```
SELECT page, COUNT(*) AS Edits
FROM wikipedia
WHERE TIMESTAMP '2015-09-12 00:00:00' <= "__time" AND "__time" < TIMESTAMP '20
GROUP BY page
ORDER BY Edits DESC
LIMIT 10
```

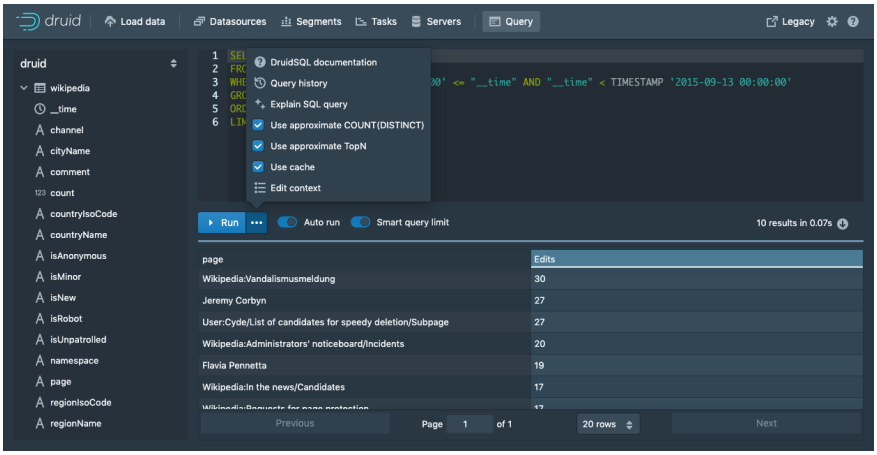
让我们来看几种不同的查询方法

通过控制台查询SQL

您可以通过在控制台中进行上述查询：

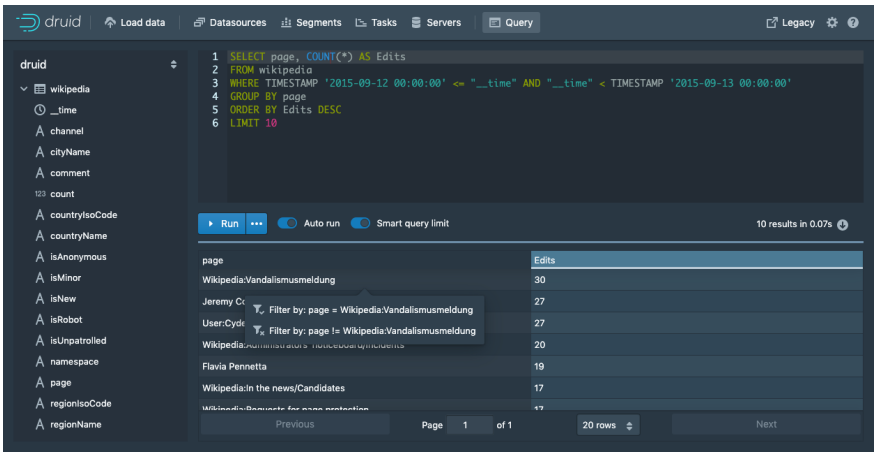


控制台查询视图通过内联文档提供自动补全功能。



您还可以从 `...` 选项菜单中配置要与查询一起发送的其他上下文标志。

请注意，控制台将（默认情况下）使用带Limit的SQL查询，以便可以完成诸如 `SELECT * FROM wikipedia` 之类的查询,您可以通过 `Smart query limit` 切换关闭此行为。



查询视图提供了可以为您编写和修改查询的上下文操作。

通过dsql查询SQL

为方便起见，Druid软件包中包括了一个SQL命令行客户端，位于Druid根目录中的 `bin/dsql`

运行 `bin/dsql` ，可以看到如下：

```
Welcome to dsql, the command-line client for Druid SQL.
Type "\h" for help.
dsql>
```

将SQl粘贴到 `dsql` 中提交查询：

```
dsql> SELECT page, COUNT(*) AS Edits FROM wikipedia WHERE "__time" BETWEEN TIM
┌──────────────────────────────────────────────────────────┬───────┐
│ page                                                     │ Edits │
├──────────────────────────────────────────────────────────┼───────┤
│ Wikipedia:Vandalismusmeldung                             │    33 │
│ User:Cyde/List of candidates for speedy deletion/Subpage │    28 │
│ Jeremy Corbyn                                            │    27 │
│ Wikipedia:Administrators' noticeboard/Incidents          │    21 │
│ Flavia Pennetta                                          │    20 │
│ Total Drama Presents: The Ridonculous Race               │    18 │
│ User talk:Dudeperson176123                               │    18 │
│ Wikipédia:Le Bistro/12 septembre 2015                    │    18 │
│ Wikipedia:In the news/Candidates                         │    17 │
│ Wikipedia:Requests for page protection                   │    17 │
└──────────────────────────────────────────────────────────┴───────┘
Retrieved 10 rows in 0.06s.
```

通过HTTP查询SQL

SQL查询作为JSON通过HTTP提交

教程包括一个示例文件, 该文件 `quickstart/tutorial/wikipedia-top-pages-sql.json` 包含上面显示的SQL查询, 我们将该查询提交给Druid Broker。

```
curl -X 'POST' -H 'Content-Type:application/json' -d @quickstart/tutorial/wiki
```

结果返回如下:

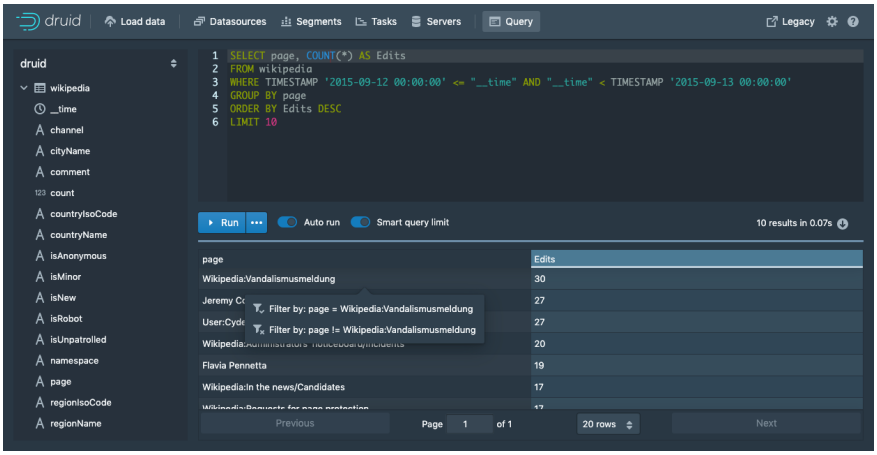
```
[
  {
    "page": "Wikipedia:Vandalismusmeldung",
    "Edits": 33
  },
  {
    "page": "User:Cyde/List of candidates for speedy deletion/Subpage",
    "Edits": 28
  },
  {
    "page": "Jeremy Corbyn",
    "Edits": 27
  },
  {
    "page": "Wikipedia:Administrators' noticeboard/Incidents",
    "Edits": 21
  },
  {
    "page": "Flavia Pennetta",
    "Edits": 20
  },
  {
    "page": "Total Drama Presents: The Ridonculous Race",
    "Edits": 18
  },
  {
    "page": "User talk:Dudeperson176123",
    "Edits": 18
  },
  {
    "page": "Wikipédia:Le Bistro/12 septembre 2015",
    "Edits": 18
  },
  {
    "page": "Wikipedia:In the news/Candidates",
    "Edits": 17
  },
  {
    "page": "Wikipedia:Requests for page protection",
    "Edits": 17
  }
]
```


更多Druid SQL示例

这是一组可尝试的查询:

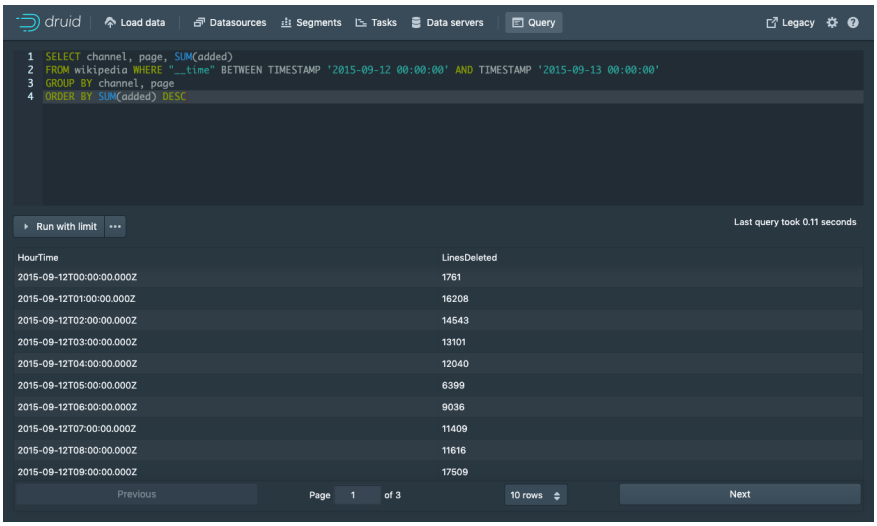
时间查询

```
SELECT FLOOR(__time to HOUR) AS HourTime, SUM(deleted) AS LinesDeleted
FROM wikipedia WHERE "__time" BETWEEN TIMESTAMP '2015-09-12 00:00:00' AND TIME
GROUP BY 1
```



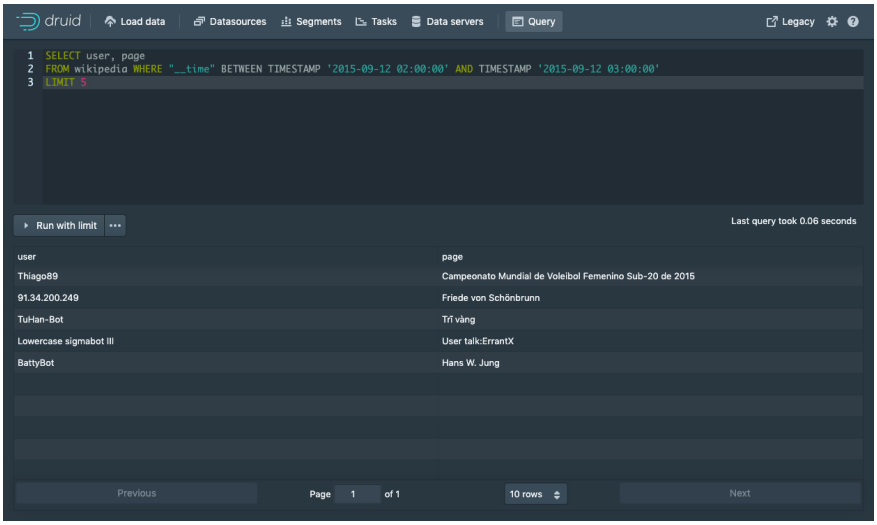
聚合查询

```
SELECT channel, page, SUM(added)
FROM wikipedia WHERE "__time" BETWEEN TIMESTAMP '2015-09-12 00:00:00' AND TIME
GROUP BY channel, page
ORDER BY SUM(added) DESC
```



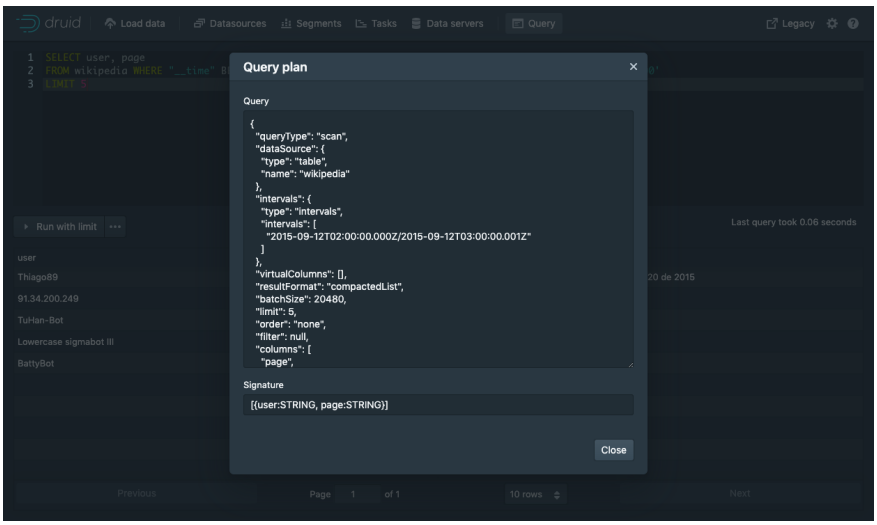
查询原始数据

```
SELECT user, page
FROM wikipedia WHERE "__time" BETWEEN TIMESTAMP '2015-09-12 02:00:00' AND TIME
LIMIT 5
```



SQL查询计划

Druid SQL能够解释给定查询的查询计划, 在控制台中，可以通过 `...` 按钮访问此功能。



如果您以其他方式查询，则可以通过在Druid SQL查询之前添加 `EXPLAIN PLAN FOR` 来获得查询计划。

使用上边的一个示例：

```
EXPLAIN PLAN FOR SELECT page, COUNT(*) AS Edits FROM wikipedia WHERE "__time"
BETWEEN TIMESTAMP '2015-09-12 00:00:00' AND TIMESTAMP '2015-09-13 00:00:00'
GROUP BY page ORDER BY Edits DESC LIMIT 10;
```

```
dsql> EXPLAIN PLAN FOR SELECT page, COUNT(*) AS Edits FROM wikipedia WHERE "__
┌──────────────────────────────────────────────────────────────────────────────
│ PLAN
├──────────────────────────────────────────────────────────────────────────────
│ DruidQueryRel(query=[{"queryType":"topN","dataSource":{"type":"table","name"
└──────────────────────────────────────────────────────────────────────────────
Retrieved 1 row in 0.03s.
```

原生JSON查询

Druid的原生查询格式以JSON表示。

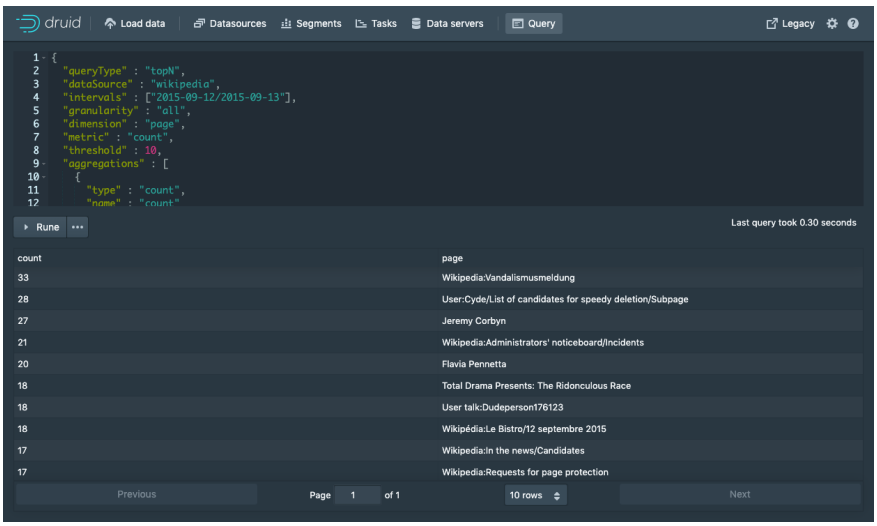
通过控制台原生查询

您可以从控制台的"Query"视图发出原生Druid查询。

这是一个查询，可检索2015-09-12上具有最多页面编辑量的10个wikipedia页面。

```
{
  "queryType" : "topN",
  "dataSource" : "wikipedia",
  "intervals" : ["2015-09-12/2015-09-13"],
  "granularity" : "all",
  "dimension" : "page",
  "metric" : "count",
  "threshold" : 10,
  "aggregations" : [
    {
      "type" : "count",
      "name" : "count"
    }
  ]
}
```

只需将其粘贴到控制台即可将编辑器切换到JSON模式。



通过HTTP原生查询

我们在 `quickstart/tutorial/wikipedia-top-pages.json` 文件中包括了一个示例原生TopN查询。

提交该查询到Druid：

```
curl -X 'POST' -H 'Content-Type:application/json' -d @quickstart/tutorial/wiki
```

您可以看到如下的查询结果：

```
[ {
  "timestamp" : "2015-09-12T00:46:58.771Z",
  "result" : [ {
    "count" : 33,
    "page" : "Wikipedia:Vandalismusmeldung"
  }, {
    "count" : 28,
    "page" : "User:Cyde/List of candidates for speedy deletion/Subpage"
  }, {
    "count" : 27,
    "page" : "Jeremy Corbyn"
  }, {
    "count" : 21,
    "page" : "Wikipedia:Administrators' noticeboard/Incidents"
  }, {
    "count" : 20,
    "page" : "Flavia Pennetta"
  }, {
    "count" : 18,
    "page" : "Total Drama Presents: The Ridonculous Race"
  }, {
    "count" : 18,
    "page" : "User talk:Dudeperson176123"
  }, {
    "count" : 18,
    "page" : "Wikipédia:Le Bistro/12 septembre 2015"
  }, {
    "count" : 17,
    "page" : "Wikipedia:In the news/Candidates"
  }, {
    "count" : 17,
    "page" : "Wikipedia:Requests for page protection"
  } ]
} ]
```


进一步阅读

查询文档有更多关于Druid原生JSON查询的信息 Druid SQL文档有更多关于Druid SQL查询的信息

Roll-up

Apache Druid可以通过roll-up在数据摄取阶段对原始数据进行汇总。 Roll-up是对选定列集的一级聚合操作，它可以减小存储数据的大小。

本教程中将讨论在一个示例数据集上进行roll-up的结果。

本教程我们假设您已经按照单服务器部署中描述下载了Druid，并运行在本地机器上。

完成加载本地文件和数据查询两部分内容也是非常有帮助的。

示例数据

对于本教程，我们将使用一个网络流事件数据的小样本，表示在特定时间内从源到目标IP地址的流量的数据包和字节计数。

```
{"timestamp":"2018-01-01T01:01:35Z","srcIP":"1.1.1.1", "dstIP":"2.2.2.2","pack
{"timestamp":"2018-01-01T01:01:51Z","srcIP":"1.1.1.1", "dstIP":"2.2.2.2","pack
{"timestamp":"2018-01-01T01:01:59Z","srcIP":"1.1.1.1", "dstIP":"2.2.2.2","pack
{"timestamp":"2018-01-01T01:02:14Z","srcIP":"1.1.1.1", "dstIP":"2.2.2.2","pack
{"timestamp":"2018-01-01T01:02:29Z","srcIP":"1.1.1.1", "dstIP":"2.2.2.2","pack
{"timestamp":"2018-01-01T01:03:29Z","srcIP":"1.1.1.1", "dstIP":"2.2.2.2","pack
{"timestamp":"2018-01-02T21:33:14Z","srcIP":"7.7.7.7", "dstIP":"8.8.8.8","pack
{"timestamp":"2018-01-02T21:33:45Z","srcIP":"7.7.7.7", "dstIP":"8.8.8.8","pack
{"timestamp":"2018-01-02T21:35:45Z","srcIP":"7.7.7.7", "dstIP":"8.8.8.8","pack
```

位于 `quickstart/tutorial/rollup-data.json` 的文件包含了样例输入数据

我们将使用 `quickstart/tutorial/rollup-index.json` 的摄入数据规范来摄取数据

```
{
  "type" : "index_parallel",
  "spec" : {
    "dataSchema" : {
      "dataSource" : "rollup-tutorial",
      "dimensionsSpec" : {
        "dimensions" : [
          "srcIP",
          "dstIP"
        ]
      },
      "timestampSpec": {
        "column": "timestamp",
        "format": "iso"
      },
      "metricsSpec" : [
        { "type" : "count", "name" : "count" },
        { "type" : "longSum", "name" : "packets", "fieldName" : "packets" },
        { "type" : "longSum", "name" : "bytes", "fieldName" : "bytes" }
      ],
      "granularitySpec" : {
        "type" : "uniform",
        "segmentGranularity" : "week",
        "queryGranularity" : "minute",
        "intervals" : ["2018-01-01/2018-01-03"],
        "rollup" : true
      }
    },
    "ioConfig" : {
      "type" : "index_parallel",
      "inputSource" : {
        "type" : "local",
        "baseDir" : "quickstart/tutorial",
        "filter" : "rollup-data.json"
      },
      "inputFormat" : {
        "type" : "json"
      },
      "appendToExisting" : false
    },
    "tuningConfig" : {
      "type" : "index_parallel",
      "maxRowsPerSegment" : 5000000,
      "maxRowsInMemory" : 25000
    }
  }
}
```

通过在 `granularitySpec` 选项中设置 `rollup : true` 来启用Roll-up

注意，我们将 `srcIP` 和 `dstIP` 定义为**维度**，将 `packets` 和 `bytes` 列定义为了 `longSum` 类型的**指标**，并将 `queryGranularity` 配置定义为 `minute` 。

加载这些数据后，我们将看到如何使用这些定义。

加载示例数据

在Druid的根目录下运行以下命令：

```
bin/post-index-task --file quickstart/tutorial/rollup-index.json --url http://
```

脚本运行完成以后，我们将查询数据。

查询示例数据

现在运行 `bin/dsql` 然后执行查询 `select * from "rollup-tutorial";` 来查看已经被摄入的数据。

```
$ bin/dsql
Welcome to dsql, the command-line client for Druid SQL.
Type "\h" for help.
dsql> select * from "rollup-tutorial";
┌──────────────────────────┬────────┬───────┬─────────┬─────────┬─────────┐
│ __time                   │ bytes  │ count │ dstIP   │ packets │ srcIP   │
├──────────────────────────┼────────┼───────┼─────────┼─────────┼─────────┤
│ 2018-01-01T01:01:00.000Z │  35937 │     3 │ 2.2.2.2 │     286 │ 1.1.1.1 │
│ 2018-01-01T01:02:00.000Z │ 366260 │     2 │ 2.2.2.2 │     415 │ 1.1.1.1 │
│ 2018-01-01T01:03:00.000Z │  10204 │     1 │ 2.2.2.2 │      49 │ 1.1.1.1 │
│ 2018-01-02T21:33:00.000Z │ 100288 │     2 │ 8.8.8.8 │     161 │ 7.7.7.7 │
│ 2018-01-02T21:35:00.000Z │   2818 │     1 │ 8.8.8.8 │      12 │ 7.7.7.7 │
└──────────────────────────┴────────┴───────┴─────────┴─────────┴─────────┘
Retrieved 5 rows in 1.18s.

dsql>
```

我们来看发生在 `2018-01-01T01:01` 的三条原始数据：

```
{"timestamp":"2018-01-01T01:01:35Z","srcIP":"1.1.1.1", "dstIP":"2.2.2.2","pack
{"timestamp":"2018-01-01T01:01:51Z","srcIP":"1.1.1.1", "dstIP":"2.2.2.2","pack
{"timestamp":"2018-01-01T01:01:59Z","srcIP":"1.1.1.1", "dstIP":"2.2.2.2","pack
```

这三条数据已经被roll up为以下一行数据：

```
┌──────────────────────────┬────────┬───────┬─────────┬─────────┬─────────┐
│ __time                   │ bytes  │ count │ dstIP   │ packets │ srcIP   │
├──────────────────────────┼────────┼───────┼─────────┼─────────┼─────────┤
│ 2018-01-01T01:01:00.000Z │  35937 │     3 │ 2.2.2.2 │     286 │ 1.1.1.1 │
└──────────────────────────┴────────┴───────┴─────────┴─────────┴─────────┘
```

这输入的数据行已经被按照时间列和维度列 `{timestamp, srcIP, dstIP}` 在指标列 `{packages, bytes}` 上做求和聚合

在进行分组之前，原始输入数据的时间戳按分钟进行标记/布局，这是由于摄取规范中的 `"queryGranularity": "minute"` 设置造成的。 同样， `2018-01-01T01:02` 期间发生的这两起事件也已经汇总。

```
{"timestamp":"2018-01-01T01:02:14Z","srcIP":"1.1.1.1", "dstIP":"2.2.2.2","pack
{"timestamp":"2018-01-01T01:02:29Z","srcIP":"1.1.1.1", "dstIP":"2.2.2.2","pack
```

```
┌──────────────────────────┬────────┬───────┬─────────┬─────────┬─────────┐
│ __time                   │ bytes  │ count │ dstIP   │ packets │ srcIP   │
├──────────────────────────┼────────┼───────┼─────────┼─────────┼─────────┤
│ 2018-01-01T01:02:00.000Z │ 366260 │     2 │ 2.2.2.2 │     415 │ 1.1.1.1 │
└──────────────────────────┴────────┴───────┴─────────┴─────────┴─────────┘
```

对于记录1.1.1.1和2.2.2.2之间流量的最后一个事件没有发生汇总，因为这是 `2018-01-01T01:03` 期间发生的唯一事件

```
{"timestamp":"2018-01-01T01:03:29Z","srcIP":"1.1.1.1", "dstIP":"2.2.2.2","pack
```

__time	bytes	count	dstIP	packets	srcIP
2018-01-01T01:03:00.000Z	10204	1	2.2.2.2	49	1.1.1.1

请注意， `计数指标 count` 显示原始输入数据中有多少行贡献给最终的"roll up"行。

配置数据保留规则

本教程演示如何在数据源上配置保留规则，以设置要保留或删除的数据的时间间隔

本教程我们假设您已经按照单服务器部署中描述下载了Druid，并运行在本地机器上。

完成加载本地文件和数据查询两部分内容也是非常有帮助的。

加载示例数据

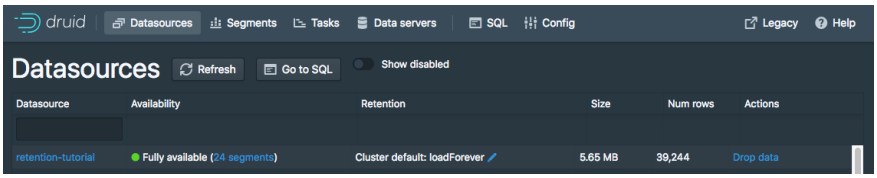
在本教程中，我们将使用Wikipedia编辑的示例数据，其中包含一个摄取任务规范，它将为输入数据每个小时创建一个单独的段

数据摄取规范位于 `quickstart/tutorial/retention-index.json`，提交这个规范，将创建一个名称为 `retention-tutorial` 的数据源

```
bin/post-index-task --file quickstart/tutorial/retention-index.json --url http
```

摄取完成后，在浏览器中转到http://localhost:8888/unified-console.html#datasources以访问Druid控制台的datasource视图

此视图显示可用的数据源以及每个数据源的保留规则摘要



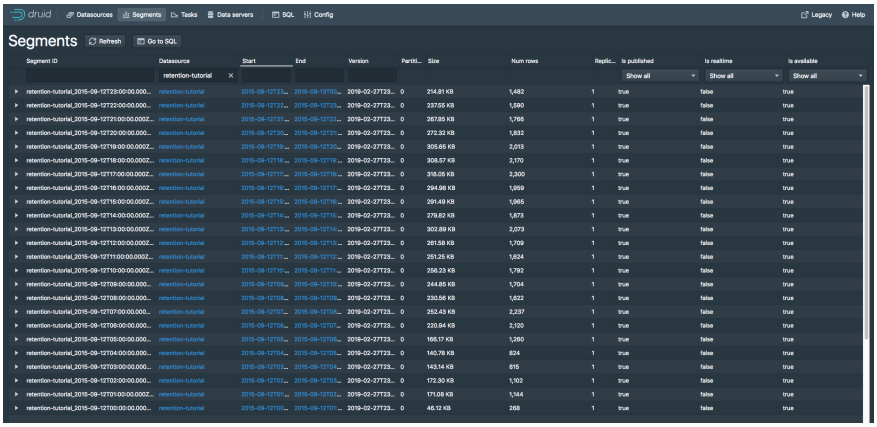
当前没有为 `retention-tutorial` 数据源设置规则。请注意，集群有默认规则：在 `_default_tier` 中永久加载2个副本

这意味着无论时间戳如何，所有数据都将加载，并且每个段将复制到两个Historical进程的 `_default_tier` 中

在本教程中，我们将暂时忽略分层和冗余概念

让我们通过单击"Fully Available"旁边的"24 Segments"链接来查看 `retention-tutorial` 数据源的段

Segment视图 提供了一个数据源包括的segment信息，本页显示有24个段，每一个段包括了2015-09-12特定小时的数据

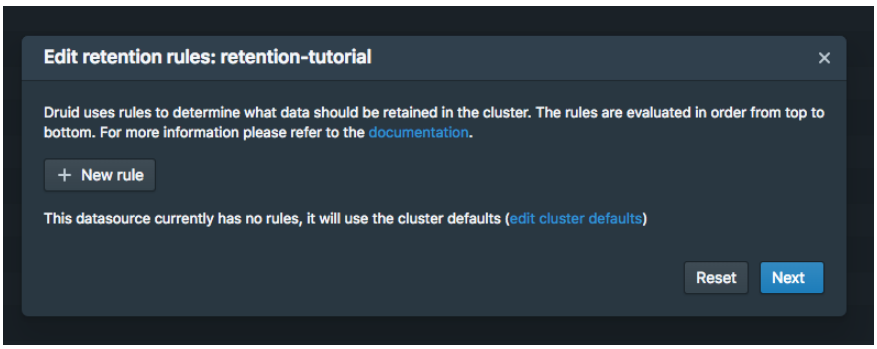


设置数据保留规则

假设我们想删除2015年9月12日前12小时的数据，保留2015年9月12日后12小时的数据。

进入到Datasources视图，点击 `retention-tutorial` 数据源的蓝色铅笔的图标 `Cluster default: loadForever`

一个规则配置窗口出现了：

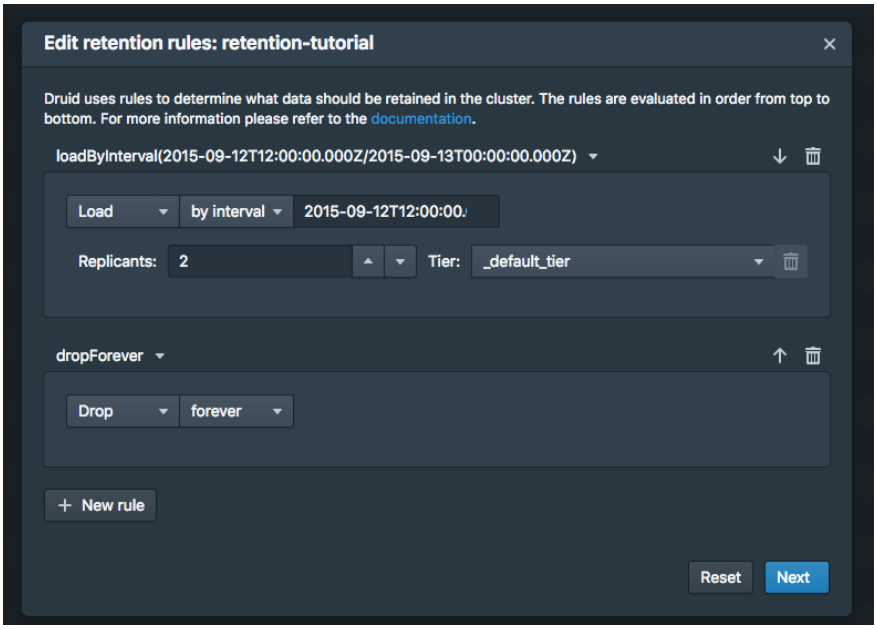


现在点击 `+ New rule` 按钮两次

在上边的规则框中，选择 `Load` 和 `by Interval` 然后输入在 `by Interval` 旁边的输入框中输入 `2015-09-12T12:00:00.000Z/2015-09-13T00:00:00.000Z`，副本可以选择保持2，在 `_default_tier` 中

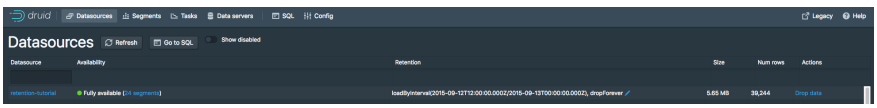
在下边的规则框中，选择 `Drop` 和 `forever`

规则看上去是这样的：

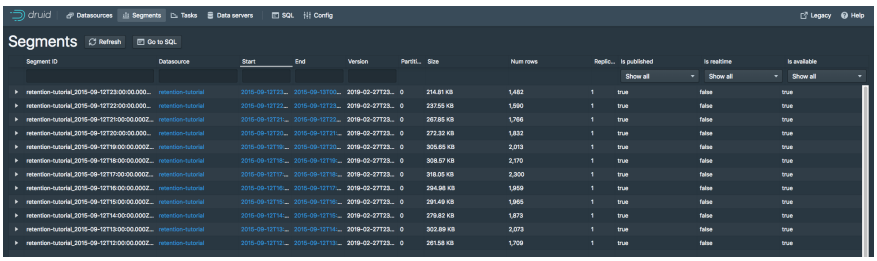


现在点击 `Next`，规则配置过程将要求提供用户名和注释，以便进行更改日志记录。您可以同时输入教程。

现在点击 `Save`，可以在Datasources视图中看到新的规则



给集群几分钟时间应用规则更改，然后转到Druid控制台中的segments视图。2015年9月12日前12小时的段文件现已消失



生成的保留规则链如下:

1. loadByInterval 2015-09-12T12/2015-09-13 (12 hours)
2. dropForever
3. loadForever (默认规则)

规则链是自上而下计算的，默认规则链始终添加在底部

我们刚刚创建的教程规则链在指定的12小时间隔内加载数据

如果数据不在12小时的间隔内，则规则链下一步将计算 `dropForever`，这将删除任何数据

`dropForever` 终止了规则链，有效地覆盖了默认的 `loadForever` 规则，在这个规则链中永远不会到达该规则

注意，在本教程中，我们定义了一个特定间隔的加载规则

相反，如果希望根据数据的生命周期保留数据（例如，保留从过去3个月到现在3个月的数据），则应定义一个周期性加载规则(Period Load Rule)。

进一步阅读

加载规则

数据更新

本教程演示如何更新现有数据，同时展示覆盖(Overwrite)和追加(append)的两个方式。

本教程我们假设您已经按照单服务器部署中描述下载了Druid，并运行在本地机器上。

完成加载本地文件、数据查询和roll-up部分内容也是非常有帮助的

数据覆盖

本节教程将介绍如何覆盖现有的指定间隔的数据

加载初始数据

本节教程使用的任务摄取规范位于 `quickstart/tutorial/updates-init-index.json`，本规范从 `quickstart/tutorial/updates-data.json` 输入文件创建一个名称为 `updates-tutorial` 的数据源

提交任务：

```
bin/post-index-task --file quickstart/tutorial/updates-init-index.json --url h
```

我们有三个包含"动物"维度和"数字"指标的初始行：

```
dsql> select * from "updates-tutorial";
┌──────────────────────────┬──────────┬───────┬────────┐
│ __time                   │ animal   │ count │ number │
├──────────────────────────┼──────────┼───────┼────────┤
│ 2018-01-01T01:01:00.000Z │ tiger    │     1 │    100 │
│ 2018-01-01T03:01:00.000Z │ aardvark │     1 │     42 │
│ 2018-01-01T03:01:00.000Z │ giraffe  │     1 │  14124 │
└──────────────────────────┴──────────┴───────┴────────┘
Retrieved 3 rows in 1.42s.
```

覆盖初始数据

为了覆盖这些数据，我们可以在相同的时间间隔内提交另一个任务，但是使用不同的输入数据。

`quickstart/tutorial/updates-overwrite-index.json` 规范将会对 `updates-tutorial` 数据进行数据重写

注意，此任务从 `quickstart/tutorial/updates-data2.json` 读取输入， `appendToExisting` 设置为false（表示这是一个覆盖）

提交任务：

```
bin/post-index-task --file quickstart/tutorial/updates-overwrite-index.json --
```

当Druid从这个覆盖任务加载完新的段时，"tiger"行现在有了值"lion"，"aardvark"行有了不同的编号，"giraffe"行已经被替换。更改可能需要几分钟才能生效：

```
dsql> select * from "updates-tutorial";
┌──────────────────────────┬──────────┬───────┬────────┐
│ __time                   │ animal   │ count │ number │
├──────────────────────────┼──────────┼───────┼────────┤
│ 2018-01-01T01:01:00.000Z │ lion     │     1 │    100 │
│ 2018-01-01T03:01:00.000Z │ aardvark │     1 │   9999 │
│ 2018-01-01T04:01:00.000Z │ bear     │     1 │    111 │
└──────────────────────────┴──────────┴───────┴────────┘
Retrieved 3 rows in 0.02s.
```

将旧数据与新数据合并并覆盖

现在我们尝试在 `updates-tutorial` 数据源追加一些新的数据，我们将从 `quickstart/tutorial/updates-data3.json` 增加新的数据

`quickstart/tutorial/updates-append-index.json` 任务规范配置为从现有的 `updates-tutorial` 数据源和 `quickstart/tutorial/updates-data3.json` 文件读取数据，该任务将组合来自两个输入源的数据，然后用新的组合数据覆盖原始数据。

提交任务：

```
bin/post-index-task --file quickstart/tutorial/updates-append-index.json --url
```

当Druid完成从这个覆盖任务加载新段时，新行将被添加到数据源中。请注意，“Lion”行发生了roll up：

```
dsql> select * from "updates-tutorial";
┌──────────────────────────┬──────────┬───────┬────────┐
│ __time                   │ animal   │ count │ number │
├──────────────────────────┼──────────┼───────┼────────┤
│ 2018-01-01T01:01:00.000Z │ lion     │     2 │    400 │
│ 2018-01-01T03:01:00.000Z │ aardvark │     1 │   9999 │
│ 2018-01-01T04:01:00.000Z │ bear     │     1 │    111 │
│ 2018-01-01T05:01:00.000Z │ mongoose │     1 │    737 │
│ 2018-01-01T06:01:00.000Z │ snake    │     1 │   1234 │
│ 2018-01-01T07:01:00.000Z │ octopus  │     1 │    115 │
└──────────────────────────┴──────────┴───────┴────────┘
Retrieved 6 rows in 0.02s.
```

追加数据

现在尝试另一种追加数据的方式

`quickstart/tutorial/updates-append-index2.json` 任务规范从 `quickstart/tutorial/updates-data4.json` 文件读取数据，然后追加到 `updates-tutorial` 数据源。注意到在规范中 `appendToExisting` 设置为 `true`

提交任务：

```
bin/post-index-task --file quickstart/tutorial/updates-append-index2.json --ur
```

加载新数据后，我们可以看到"octopus"后面额外的两行。请注意，编号为222的新"bear"行尚未与现有的bear-111行合并，因为新数据保存在单独的段中。

```
dsql> select * from "updates-tutorial";
┌──────────────────────────┬──────────┬───────┬────────┐
│ __time                   │ animal   │ count │ number │
├──────────────────────────┼──────────┼───────┼────────┤
│ 2018-01-01T01:01:00.000Z │ lion     │     2 │    400 │
│ 2018-01-01T03:01:00.000Z │ aardvark │     1 │   9999 │
│ 2018-01-01T04:01:00.000Z │ bear     │     1 │    111 │
│ 2018-01-01T05:01:00.000Z │ mongoose │     1 │    737 │
│ 2018-01-01T06:01:00.000Z │ snake    │     1 │   1234 │
│ 2018-01-01T07:01:00.000Z │ octopus  │     1 │    115 │
│ 2018-01-01T04:01:00.000Z │ bear     │     1 │    222 │
│ 2018-01-01T09:01:00.000Z │ falcon   │     1 │   1241 │
└──────────────────────────┴──────────┴───────┴────────┘
Retrieved 8 rows in 0.02s.
```

当我们执行一个GroupBy查询而非 `Select *`, 我们看到"beer"行将在查询时聚合在一起:

```
dsql> select __time, animal, SUM("count"), SUM("number") from "updates-tutoria
┌──────────────────────────┬──────────┬─────────┬─────────┐
│ __time                   │ animal   │ EXPR$2  │ EXPR$3  │
├──────────────────────────┼──────────┼─────────┼─────────┤
│ 2018-01-01T01:01:00.000Z │ lion     │       2 │     400 │
│ 2018-01-01T03:01:00.000Z │ aardvark │       1 │    9999 │
│ 2018-01-01T04:01:00.000Z │ bear     │       2 │     333 │
│ 2018-01-01T05:01:00.000Z │ mongoose │       1 │     737 │
│ 2018-01-01T06:01:00.000Z │ snake    │       1 │    1234 │
│ 2018-01-01T07:01:00.000Z │ octopus  │       1 │     115 │
│ 2018-01-01T09:01:00.000Z │ falcon   │       1 │    1241 │
└──────────────────────────┴──────────┴─────────┴─────────┘
Retrieved 7 rows in 0.23s.
```

合并段文件

本教程演示如何将现有段合并为较少但更大的段

因为每一个段都有一些内存和处理开销，所以有时减少段的总数是有益的。有关详细信息，请查阅段大小优化。

本教程我们假设您已经按照单服务器部署中描述下载了Druid，并运行在本地机器上。

完成加载本地文件和数据查询两部分内容也是非常有帮助的。

加载初始数据

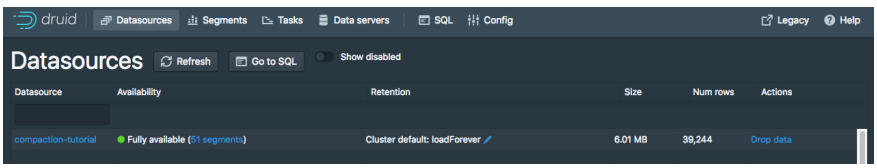
在本教程中，我们将使用wikipedia编辑的示例数据，其中的摄取任务规范将在输入数据中每小时创建1-3个段。

数据摄取规范位于 `quickstart/tutorial/compaction-init-index.json` ,提交这个任务规范将创建一个名称为 `compaction-tutorial` 的数据源：

```
bin/post-index-task --file quickstart/tutorial/compaction-init-index.json --ur
```

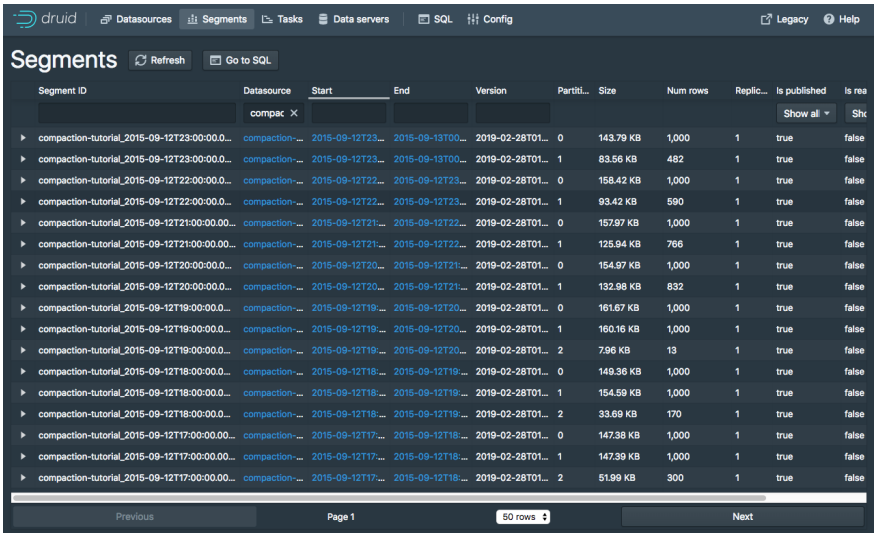
> ![WARNING] 请注意，摄取规范中的 `maxRowsPerSegment` 设置为1000, 这是为了每小时生成多个段，不建议在生产中使用。默认为5000000，可能需要进行调整以优化您的段文件。

摄取任务完成后，可以到 http://localhost:8888/unified-console.html#datasources Druid控制台查看新的数据源。



点击 `compaction-tutorial` 数据源中"Fully Available"旁边的 `51 Segments` 链接来查看数据源段的更多信息

该数据源有51个段文件，输入数据每小时1-3个段



对该数据源执行一个 `COUNT(*)` 查询可以看到39244行数据：

```
dsql> select count(*) from "compaction-tutorial";
┌────────┐
│ EXPR$0 │
├────────┤
│  39244 │
└────────┘
Retrieved 1 row in 1.38s.
```

合并数据

现在我们将合并这51个小的段

在 `quickstart/tutorial/compaction-keep-granularity.json` 文件中我们包含了一个本教程数据源的合并任务规范。

```
{
  "type": "compact",
  "dataSource": "compaction-tutorial",
  "interval": "2015-09-12/2015-09-13",
  "tuningConfig" : {
    "type" : "index_parallel",
    "maxRowsPerSegment" : 5000000,
    "maxRowsInMemory" : 25000
  }
}
```

该任务会合并 `compaction-tutorial` 数据源在 `2015-09-12/2015-09-13` 时间范围内的所有的段

`tuningConfig` 中的参数控制合并后的段文件集合中有多少个段。

在本教程示例中，每小时只创建一个合并段，因为每小时的行数少于5000000 `maxRowsPerSegment` （请注意，行总数为39244）。

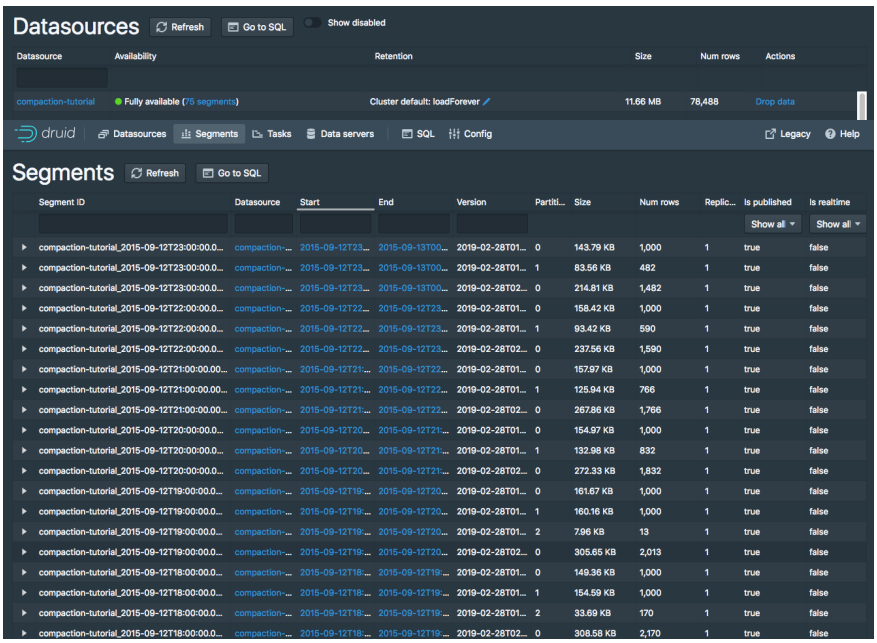
现在提交这个任务：

```
bin/post-index-task --file quickstart/tutorial/compaction-keep-granularity.jso
```

任务运行结束后，刷新 `Segments` 视图

最初的51个段最终将由Coordinator标记为"未使用"，并移除，保留新的合并段。

默认情况下，在Coordinator启动至少15分钟之前，Druid Coordinator不会将段标记为未使用，因此您可以在Druid控制台中同时看到旧段集合和新合并集，共有75个段：

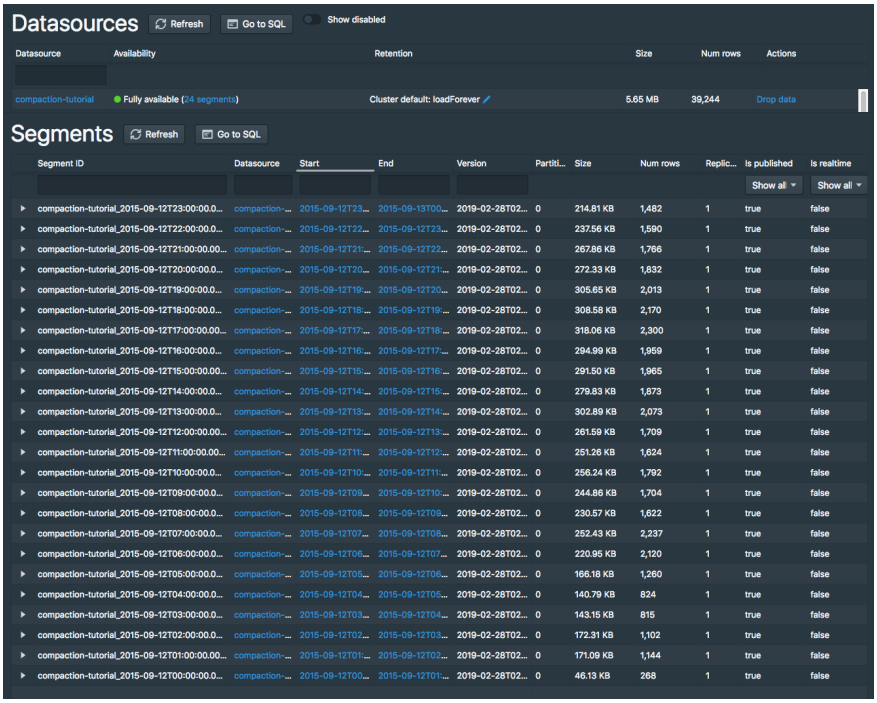


新的合并段比原来的段有一个更新的版本，所以即使两组段都显示在Druid控制台中，查询也只能从新的合并段中读取。

我们再次在 `compaction-tutorial` 数据源执行 `COUNT(*)` 查询可以看到，行数仍然是39244:

```
dsql> select count(*) from "compaction-tutorial";
┌────────┐
│ EXPR$0 │
├────────┤
│  39244 │
└────────┘
Retrieved 1 row in 1.30s.
```

Coordinator运行至少15分钟后，"Segments"视图应显示有24个分段，每小时一个：



用新的段粒度合并数据

合并任务还可以生成不同于输入段粒度的合并段

我们在 `quickstart/tutorial/compaction-day-granularity.json` 文件中包含了一个可以创建 `DAY` 粒度的合并任务摄取规范：

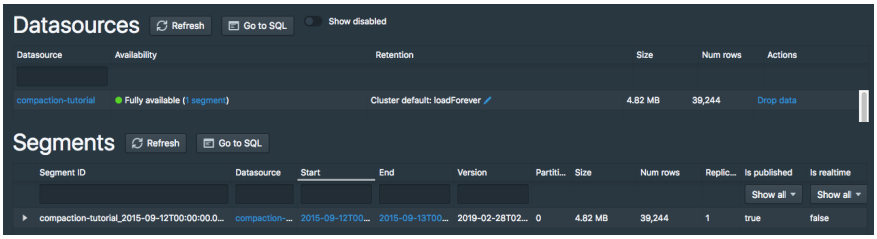
```
{
  "type": "compact",
  "dataSource": "compaction-tutorial",
  "interval": "2015-09-12/2015-09-13",
  "segmentGranularity": "DAY",
  "tuningConfig" : {
    "type" : "index_parallel",
    "maxRowsPerSegment" : 5000000,
    "maxRowsInMemory" : 25000,
    "forceExtendableShardSpecs" : true
  }
}
```

请注意这个合并任务规范中 `segmentGranularity` 配置项设置为了 `DAY`

现在提交这个任务：

```
bin/post-index-task --file quickstart/tutorial/compaction-day-granularity.json
```

Coordinator将旧的输入段标记为未使用需要一段时间，因此您可能会看到总共有25个段的中间状态。最终，只有一个天粒度的段



进一步阅读

任务文档

段优化

数据删除

本教程演示如何删除数据

本教程我们假设您已经按照单服务器部署中描述下载了Druid，并运行在本地机器上。

加载初始数据

在本教程中，我们将使用Wikipedia编辑数据，并使用创建每小时段的索引规范

这份规范位于 `quickstart/tutorial/deletion-index.json`，它将创建一个名称为 `deletion-tutorial` 的数据源

现在加载这份初始数据：

```
bin/post-index-task --file quickstart/tutorial/deletion-index.json --url http:
```

当加载完成后，在浏览器中访问http://localhost:8888/unified-console.html#datasources

如何永久删除数据

永久删除一个段需要两步：

1. 段必须首先标记为"未使用"。当用户通过Coordinator API手动禁用段时，就会发生这种情况
2. 在段被标记为"未使用"之后，一个Kill任务将从Druid的元数据存储和深层存储中删除任何“未使用”的段

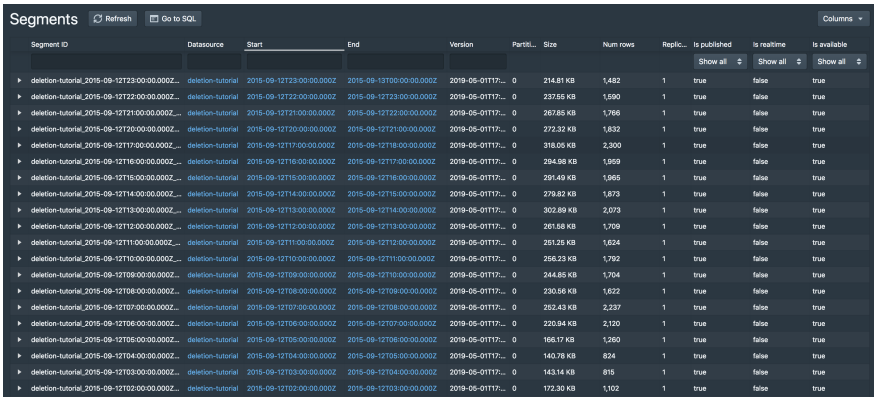
现在让我们通过使用Coordinator API按时间间隔和段id删除一些段。

通过时间间隔禁用段

让我们在指定的时间间隔内禁用段。这会将间隔中的所有段标记为"未使用"，但不会将它们从深层存储中移除。让我们禁用间隔 `2015-09-12T18:00:00.000Z/2015-09-12T20:00:00.000Z` 中的段，即在18到20小时之间

```
curl -X 'POST' -H 'Content-Type:application/json' -d '{ "interval" : "2015-09-
```

该命令完成后，您应该看到第18和19小时的段已被禁用：



请注意，第18小时和第19小时的数据段仍在深层存储中:

```
$ ls -l1 var/druid/segments/deletion-tutorial/
2015-09-12T00:00:00.000Z_2015-09-12T01:00:00.000Z
2015-09-12T01:00:00.000Z_2015-09-12T02:00:00.000Z
2015-09-12T02:00:00.000Z_2015-09-12T03:00:00.000Z
2015-09-12T03:00:00.000Z_2015-09-12T04:00:00.000Z
2015-09-12T04:00:00.000Z_2015-09-12T05:00:00.000Z
2015-09-12T05:00:00.000Z_2015-09-12T06:00:00.000Z
2015-09-12T06:00:00.000Z_2015-09-12T07:00:00.000Z
2015-09-12T07:00:00.000Z_2015-09-12T08:00:00.000Z
2015-09-12T08:00:00.000Z_2015-09-12T09:00:00.000Z
2015-09-12T09:00:00.000Z_2015-09-12T10:00:00.000Z
2015-09-12T10:00:00.000Z_2015-09-12T11:00:00.000Z
2015-09-12T11:00:00.000Z_2015-09-12T12:00:00.000Z
2015-09-12T12:00:00.000Z_2015-09-12T13:00:00.000Z
2015-09-12T13:00:00.000Z_2015-09-12T14:00:00.000Z
2015-09-12T14:00:00.000Z_2015-09-12T15:00:00.000Z
2015-09-12T15:00:00.000Z_2015-09-12T16:00:00.000Z
2015-09-12T16:00:00.000Z_2015-09-12T17:00:00.000Z
2015-09-12T17:00:00.000Z_2015-09-12T18:00:00.000Z
2015-09-12T18:00:00.000Z_2015-09-12T19:00:00.000Z
2015-09-12T19:00:00.000Z_2015-09-12T20:00:00.000Z
2015-09-12T20:00:00.000Z_2015-09-12T21:00:00.000Z
2015-09-12T21:00:00.000Z_2015-09-12T22:00:00.000Z
2015-09-12T22:00:00.000Z_2015-09-12T23:00:00.000Z
2015-09-12T23:00:00.000Z_2015-09-13T00:00:00.000Z
```

通过段ID禁用段

让我们按段id禁用一些段。这将再次将段标记为“未使用”，但不会将它们从深层存储中移除。您可以从UI中看到完整的段id，如下所述。

在"Segments"视图中，单击左侧的箭头以展开段条目：

信息框的顶部显示完整的段ID，例如 `deletion-tutorial_2015-09-12T14:00:00.000Z_2015-09-12T15:00:00.000Z_2019-02-28T01:11:51.606Z` ，第14小时的段。

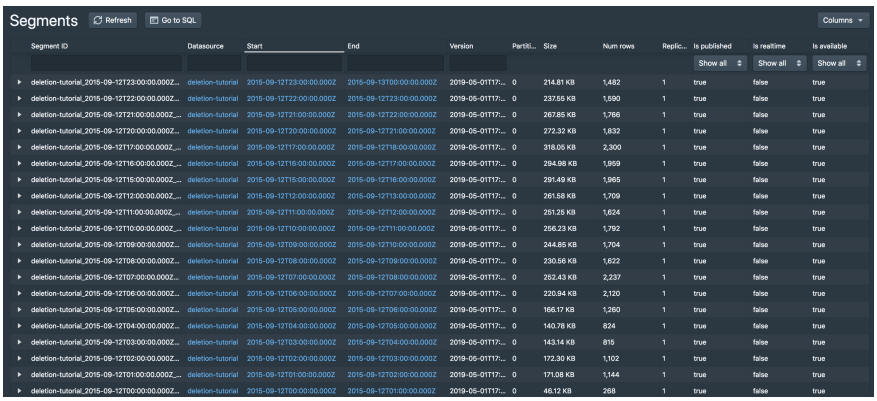
让我们向Coordinator发送一个POST请求来禁用13点和14点的段

```
{
  "segmentIds":
  [
    "deletion-tutorial_2015-09-12T13:00:00.000Z_2015-09-12T14:00:00.000Z_2019-
    "deletion-tutorial_2015-09-12T14:00:00.000Z_2015-09-12T15:00:00.000Z_2019-
  ]
}
```

json文件位于 `curl -X 'POST' -H 'Content-Type:application/json' -d @quickstart/tutorial/deletion-disable-segments.json http://localhost:8081/druid/coordinator/v1/datasources/deletion-tutorial/markUnused` ，如下向Coordinator提交一个POST请求：

```
curl -X 'POST' -H 'Content-Type:application/json' -d @quickstart/tutorial/dele
```

命令执行完成后，可以看到13时和14时的段已经被禁用：



注意到这时13时和14时的段仍然在深度存储中：

```
$ ls -l1 var/druid/segments/deletion-tutorial/
2015-09-12T00:00:00.000Z_2015-09-12T01:00:00.000Z
2015-09-12T01:00:00.000Z_2015-09-12T02:00:00.000Z
2015-09-12T02:00:00.000Z_2015-09-12T03:00:00.000Z
2015-09-12T03:00:00.000Z_2015-09-12T04:00:00.000Z
2015-09-12T04:00:00.000Z_2015-09-12T05:00:00.000Z
2015-09-12T05:00:00.000Z_2015-09-12T06:00:00.000Z
2015-09-12T06:00:00.000Z_2015-09-12T07:00:00.000Z
2015-09-12T07:00:00.000Z_2015-09-12T08:00:00.000Z
2015-09-12T08:00:00.000Z_2015-09-12T09:00:00.000Z
2015-09-12T09:00:00.000Z_2015-09-12T10:00:00.000Z
2015-09-12T10:00:00.000Z_2015-09-12T11:00:00.000Z
2015-09-12T11:00:00.000Z_2015-09-12T12:00:00.000Z
2015-09-12T12:00:00.000Z_2015-09-12T13:00:00.000Z
2015-09-12T13:00:00.000Z_2015-09-12T14:00:00.000Z
2015-09-12T14:00:00.000Z_2015-09-12T15:00:00.000Z
2015-09-12T15:00:00.000Z_2015-09-12T16:00:00.000Z
2015-09-12T16:00:00.000Z_2015-09-12T17:00:00.000Z
2015-09-12T17:00:00.000Z_2015-09-12T18:00:00.000Z
2015-09-12T18:00:00.000Z_2015-09-12T19:00:00.000Z
2015-09-12T19:00:00.000Z_2015-09-12T20:00:00.000Z
2015-09-12T20:00:00.000Z_2015-09-12T21:00:00.000Z
2015-09-12T21:00:00.000Z_2015-09-12T22:00:00.000Z
2015-09-12T22:00:00.000Z_2015-09-12T23:00:00.000Z
2015-09-12T23:00:00.000Z_2015-09-13T00:00:00.000Z
```

运行Kill任务

现在我们已经禁用了一些段，我们可以提交一个Kill任务，它将从元数据和深层存储中删除禁用的段。

在 `quickstart/tutorial/deletion-kill.json` 提供了一个Kill任务的规范，通过以下的命令将任务提交到Overlord：

```
curl -X 'POST' -H 'Content-Type:application/json' -d @quickstart/tutorial/dele
```

任务执行完成后，可以看到已经禁用的段已经被从深度存储中移除了：

```
$ ls -l1 var/druid/segments/deletion-tutorial/
2015-09-12T12:00:00.000Z_2015-09-12T13:00:00.000Z
2015-09-12T15:00:00.000Z_2015-09-12T16:00:00.000Z
2015-09-12T16:00:00.000Z_2015-09-12T17:00:00.000Z
2015-09-12T17:00:00.000Z_2015-09-12T18:00:00.000Z
2015-09-12T20:00:00.000Z_2015-09-12T21:00:00.000Z
2015-09-12T21:00:00.000Z_2015-09-12T22:00:00.000Z
2015-09-12T22:00:00.000Z_2015-09-12T23:00:00.000Z
2015-09-12T23:00:00.000Z_2015-09-13T00:00:00.000Z
```

编写一个摄取规范

本教程将指导读者定义摄取规范的过程，指出关键的注意事项和指导原则

本教程我们假设您已经按照单服务器部署中描述下载了Druid，并运行在本地机器上。

完成加载本地文件、数据查询和roll-up部分内容也是非常有帮助的

示例数据

假设我们有如下的网络流数据：

- `srcIP`：发送端的IP地址
- `srcPort`：发送端的端口
- `dstIP`：接收端的IP地址
- `dstPort`：接收端的端口
- `protocol`：IP协议号码
- `packets`：传输的数据包数
- `bytes`：传输的字节数
- `cost`：发送流量的成本

```
{"ts":"2018-01-01T01:01:35Z","srcIP":"1.1.1.1", "dstIP":"2.2.2.2", "srcPort":2
{"ts":"2018-01-01T01:01:51Z","srcIP":"1.1.1.1", "dstIP":"2.2.2.2", "srcPort":2
{"ts":"2018-01-01T01:01:59Z","srcIP":"1.1.1.1", "dstIP":"2.2.2.2", "srcPort":2
{"ts":"2018-01-01T01:02:14Z","srcIP":"1.1.1.1", "dstIP":"2.2.2.2", "srcPort":5
{"ts":"2018-01-01T01:02:29Z","srcIP":"1.1.1.1", "dstIP":"2.2.2.2", "srcPort":5
{"ts":"2018-01-01T01:03:29Z","srcIP":"1.1.1.1", "dstIP":"2.2.2.2", "srcPort":5
{"ts":"2018-01-01T02:33:14Z","srcIP":"7.7.7.7", "dstIP":"8.8.8.8", "srcPort":4

{"ts":"2018-01-01T02:33:45Z","srcIP":"7.7.7.7", "dstIP":"8.8.8.8", "srcPort":4
{"ts":"2018-01-01T02:35:45Z","srcIP":"7.7.7.7", "dstIP":"8.8.8.8", "srcPort":4
```

将上面的JSON内容保存到 `quickstart/` 中名为 `insertion-tutorial-data.json` 的文件中。

让我们来了解一下定义可加载此数据的摄取规范的过程。

对于本教程，我们将使用本地批索引任务。使用其他任务类型时，摄取规范的某些方面会有所不同，本教程将指出这些方面。

定义schema

Druid摄取规范中最核心的元素是 `dataSchema`。 `dataSchema` 定义了如何将输入数据解析为一组列，这些列将存储在Druid中。

让我们从一个空的 `dataSchema` 开始，并在完成教程的过程中向它添加字段。

在 `quickstart/` 中创建一个名为 `insertion-tutorial-index.json` 的新文件，其中包含以下内容：

```
"dataSchema" : {}
```

随着教程的深入，我们将对这个摄取规范进行连续的编辑。

数据源名称

数据源的名称通过 `dataSource` 字段来在 `dataSchema` 中被指定：

```
"dataSchema" : {
  "dataSource" : "ingestion-tutorial",
}
```

我们将该教程数据源命名为 `ingestion-tutorial`

时间列

`dataSchema` 需要知道如何从输入数据中提取主时间戳字段。

输入数据中的timestamp列名为"ts"，包含ISO 8601时间戳，因此让我们将包含该信息的 `timestampSpec` 添加到 `dataSchema`：

```
"dataSchema" : {
  "dataSource" : "ingestion-tutorial",
  "timestampSpec" : {
    "format" : "iso",
    "column" : "ts"
  }
}
```

列类型

现在我们已经定义了时间列，让我们看看其他列的定义。

Druid支持以下列类型：String、Long、Float和Double。我们将在下面的部分中了解如何使用它们。

在讨论如何定义其他非时间列之前，我们先讨论一下 `rollup` 。

Rollup

在接收数据时，我们必须考虑是否要使用rollup

- 如果启用了rollup，我们需要将输入列分为两类，"dimensions"和"metrics"。"dimensions" 是用于rollup的分组列，"metrics"是将要聚合的列。
- 如果禁用了rollup，则所有列都被视为"dimensions"，并且不会发生预聚合。

对于本教程，让我们启用rollup。这是用 `dataSchema` 上 `granularitySpec` 指定的。

```
"dataSchema" : {
  "dataSource" : "ingestion-tutorial",
  "timestampSpec" : {
    "format" : "iso",
    "column" : "ts"
  },
  "granularitySpec" : {
    "rollup" : true
  }
}
```

选择dimension和Metrics

对于这个示例数据集，以下是对"dimensions"和"metrics"的合理划分：

- Dimensions: srcIP, srcPort, dstIP, dstPort, protocol
- Metrics: packets, bytes, cost

这里的维度是一组属性，用于标识IP流量的单向流，而度量则表示由维度分组指定的IP流量的事实。

让我们看看如何在摄取规范中定义这些维度和度量。

Dimensions

在 `dataSchema` 中使用 `dimensionSpec` 指定维度。

```
"dataSchema" : {
  "dataSource" : "ingestion-tutorial",
  "timestampSpec" : {
    "format" : "iso",
    "column" : "ts"
  },
  "dimensionsSpec" : {
    "dimensions": [
      "srcIP",
      { "name" : "srcPort", "type" : "long" },
      { "name" : "dstIP", "type" : "string" },
      { "name" : "dstPort", "type" : "long" },
      { "name" : "protocol", "type" : "string" }
    ]
  },
  "granularitySpec" : {
    "rollup" : true
  }
}
```

每个维度都有一个 `名称` 和一个 `类型`，其中 `类型` 可以是"long"、"float"、"double"或"string"。

注意， `srcIP` 是一个"string"维度；对于string维度，只需指定一个维度名称就足够了，因为"string"是默认的维度类型。

还要注意， `protocol` 是输入数据中的一个数值，但我们将其作为"string"列摄取；Druid将在摄取期间将输入long强制为string。

Strings vs. Numerics

数字输入应该作为数字维度还是字符串维度接收?

相对于字符串维度，数字维度有以下优点/缺点：

- 优点：数值表示可以减少磁盘上的列大小，并在从列中读取值时降低处理开销
- 缺点：数字维度没有索引，因此对其进行筛选通常比对等效字符串维度（具有位图索引）进行筛选慢

Metrics

在 `dataSchema` 中使用 `metricsSpec` 来指定metrics

```
"dataSchema" : {
  "dataSource" : "ingestion-tutorial",
  "timestampSpec" : {
    "format" : "iso",
    "column" : "ts"
  },
  "dimensionsSpec" : {
    "dimensions": [
      "srcIP",
      { "name" : "srcPort", "type" : "long" },
      { "name" : "dstIP", "type" : "string" },
      { "name" : "dstPort", "type" : "long" },
      { "name" : "protocol", "type" : "string" }
    ]
  },
  "metricsSpec" : [
    { "type" : "count", "name" : "count" },
    { "type" : "longSum", "name" : "packets", "fieldName" : "packets" },
    { "type" : "longSum", "name" : "bytes", "fieldName" : "bytes" },
    { "type" : "doubleSum", "name" : "cost", "fieldName" : "cost" }
  ],
  "granularitySpec" : {
    "rollup" : true
  }
}
```

定义Metrics时，需要指定在rollup期间对该列执行何种类型的聚合

在这里，我们在两个Long类型的Metrics列（ `数据包` 和 `字节` ）上定义了longSum聚合，并为 `cost` 列定义了一个doubleSum聚合

注意， `metricsSpec` 与 `dimensionSpec` 或 `parseSpec` 位于不同的嵌套级别；它与 `dataSchema` 中的 `parser` 属于相同的嵌套级别

注意，我们还定义了一个 `count` 聚合器。计数聚合器将跟踪原始输入数据中有多少行贡献给最终接收数据中的"roll up"行。

No Rollup

如果我们没使用rollup，所有的列都将在 `dimensionsSpec` 中指定：

```
      "dimensionsSpec" : {
        "dimensions": [
          "srcIP",
          { "name" : "srcPort", "type" : "long" },
          { "name" : "dstIP", "type" : "string" },
          { "name" : "dstPort", "type" : "long" },
          { "name" : "protocol", "type" : "string" },
          { "name" : "packets", "type" : "long" },
          { "name" : "bytes", "type" : "long" },
          { "name" : "srcPort", "type" : "double" }
        ]
      },
```

定义粒度

在这一点上，我们已经在 `dataSchema` 中定义了 `parser` 和 `metricsSpec` ，并且我们几乎已经完成了摄取规范的编写

我们需要在 `granularitySpec` 中设置一些属性：

- `granularitySpec` 类型：`uniform` 和 `arbitrary` 是两种支持的类型。在本教程中，我们将使用 `uniform` 粒度规范，其中所有段都具有统一的间隔（例如：所有段都包含一小时的数据）
- 段粒度：单个段中包含的数据的时间间隔大小？例如：`DAY`，`WEEK`
- 时间列中时间戳的bucketing粒度（称为queryGranularity）

段粒度

段粒度由 `granularitySpec` 中的 `SegmentGranularity` 属性配置。对于本教程，我们将创建小时段：

```
"dataSchema" : {
  "dataSource" : "ingestion-tutorial",
  "timestampSpec" : {
    "format" : "iso",
    "column" : "ts"
  },
  "dimensionsSpec" : {
    "dimensions": [
      "srcIP",
      { "name" : "srcPort", "type" : "long" },
      { "name" : "dstIP", "type" : "string" },
      { "name" : "dstPort", "type" : "long" },
      { "name" : "protocol", "type" : "string" }
    ]
  },
  "metricsSpec" : [
    { "type" : "count", "name" : "count" },
    { "type" : "longSum", "name" : "packets", "fieldName" : "packets" },
    { "type" : "longSum", "name" : "bytes", "fieldName" : "bytes" },
    { "type" : "doubleSum", "name" : "cost", "fieldName" : "cost" }
  ],
  "granularitySpec" : {
    "type" : "uniform",
    "segmentGranularity" : "HOUR",
    "rollup" : true
  }
}
```

我们的输入数据有两个独立小时的事件，因此此任务将生成两个段。

查询粒度 查询粒度由 `granularitySpec` 中的 `queryGranularity` 属性配置。对于本教程，我们使用分钟粒度：

```
"dataSchema" : {
  "dataSource" : "ingestion-tutorial",
  "timestampSpec" : {
    "format" : "iso",
    "column" : "ts"
  },
  "dimensionsSpec" : {
    "dimensions": [
      "srcIP",
      { "name" : "srcPort", "type" : "long" },
      { "name" : "dstIP", "type" : "string" },
      { "name" : "dstPort", "type" : "long" },
      { "name" : "protocol", "type" : "string" }
    ]
  },
  "metricsSpec" : [
    { "type" : "count", "name" : "count" },
    { "type" : "longSum", "name" : "packets", "fieldName" : "packets" },
    { "type" : "longSum", "name" : "bytes", "fieldName" : "bytes" },
    { "type" : "doubleSum", "name" : "cost", "fieldName" : "cost" }
  ],
  "granularitySpec" : {
    "type" : "uniform",
    "segmentGranularity" : "HOUR",
    "queryGranularity" : "MINUTE",
    "rollup" : true
  }
}
```

要查看查询粒度的影响，让我们从原始输入数据中查看这一行

```
{"ts":"2018-01-01T01:03:29Z","srcIP":"1.1.1.1", "dstIP":"2.2.2.2", "srcPort":5
```

当这一行以分钟查询粒度摄取时，Druid会将该行的时间戳设置为分钟桶：

```
{"ts":"2018-01-01T01:03:00Z","srcIP":"1.1.1.1", "dstIP":"2.2.2.2", "srcPort":5
```

定义时间间隔（batch only）

对于批处理任务，需要定义时间间隔。时间戳超出时间间隔的输入行将不会被接收。

时间间隔指定在 `granularitySpec` 配置项中：

```
"dataSchema" : {
  "dataSource" : "ingestion-tutorial",
  "timestampSpec" : {
    "format" : "iso",
    "column" : "ts"
  },
  "dimensionsSpec" : {
    "dimensions": [
      "srcIP",
      { "name" : "srcPort", "type" : "long" },
      { "name" : "dstIP", "type" : "string" },
      { "name" : "dstPort", "type" : "long" },
      { "name" : "protocol", "type" : "string" }
    ]
  },
  "metricsSpec" : [
    { "type" : "count", "name" : "count" },
    { "type" : "longSum", "name" : "packets", "fieldName" : "packets" },
    { "type" : "longSum", "name" : "bytes", "fieldName" : "bytes" },
    { "type" : "doubleSum", "name" : "cost", "fieldName" : "cost" }
  ],
  "granularitySpec" : {
    "type" : "uniform",
    "segmentGranularity" : "HOUR",
    "queryGranularity" : "MINUTE",
    "intervals" : ["2018-01-01/2018-01-02"],
    "rollup" : true
  }
}
```

定义任务类型

我们现在已经完成了 `dataSchema` 的定义。剩下的步骤是将我们创建的数据模式放入一个摄取任务规范中，并指定输入源。

`dataSchema` 在所有任务类型之间共享，但每个任务类型都有自己的规范格式。对于本教程，我们将使用本机批处理摄取任务：

```
{
  "type" : "index_parallel",
  "spec" : {
    "dataSchema" : {
      "dataSource" : "ingestion-tutorial",
      "timestampSpec" : {
        "format" : "iso",
        "column" : "ts"
      },
      "dimensionsSpec" : {
        "dimensions": [
          "srcIP",
          { "name" : "srcPort", "type" : "long" },
          { "name" : "dstIP", "type" : "string" },
          { "name" : "dstPort", "type" : "long" },
          { "name" : "protocol", "type" : "string" }
        ]
      },
      "metricsSpec" : [
        { "type" : "count", "name" : "count" },
        { "type" : "longSum", "name" : "packets", "fieldName" : "packets" },
        { "type" : "longSum", "name" : "bytes", "fieldName" : "bytes" },
        { "type" : "doubleSum", "name" : "cost", "fieldName" : "cost" }
      ],
      "granularitySpec" : {
        "type" : "uniform",
        "segmentGranularity" : "HOUR",
        "queryGranularity" : "MINUTE",
        "intervals" : ["2018-01-01/2018-01-02"],
        "rollup" : true
      }
    }
  }
}
```

定义输入源

现在让我们定义输入源，它在 `ioConfig` 对象中指定。每个任务类型都有自己的 `ioConfig` 类型。要读取输入数据，我们需要指定一个 `inputSource` 。我们前面保存的示例网络流数据需要从本地文件中读取，该文件配置如下：

```
    "ioConfig" : {
      "type" : "index_parallel",
      "inputSource" : {
        "type" : "local",
        "baseDir" : "quickstart/",
        "filter" : "ingestion-tutorial-data.json"
      }
    }
```

定义数据格式

因为我们的输入数据为JSON字符串，我们使用json的 `inputFormat` ：

```
    "ioConfig" : {
      "type" : "index_parallel",
      "inputSource" : {
        "type" : "local",
        "baseDir" : "quickstart/",
        "filter" : "ingestion-tutorial-data.json"
      },
      "inputFormat" : {
        "type" : "json"
      }
    }
```

```
{
  "type" : "index_parallel",
  "spec" : {
    "dataSchema" : {
      "dataSource" : "ingestion-tutorial",
      "timestampSpec" : {
        "format" : "iso",
        "column" : "ts"
      },
      "dimensionsSpec" : {
        "dimensions": [
          "srcIP",
          { "name" : "srcPort", "type" : "long" },
          { "name" : "dstIP", "type" : "string" },
          { "name" : "dstPort", "type" : "long" },
          { "name" : "protocol", "type" : "string" }
        ]
      },
      "metricsSpec" : [
        { "type" : "count", "name" : "count" },
        { "type" : "longSum", "name" : "packets", "fieldName" : "packets" },
        { "type" : "longSum", "name" : "bytes", "fieldName" : "bytes" },
        { "type" : "doubleSum", "name" : "cost", "fieldName" : "cost" }
      ],
      "granularitySpec" : {
        "type" : "uniform",
        "segmentGranularity" : "HOUR",
        "queryGranularity" : "MINUTE",
        "intervals" : ["2018-01-01/2018-01-02"],
        "rollup" : true
      }
    },
    "ioConfig" : {
      "type" : "index_parallel",
      "inputSource" : {
        "type" : "local",
        "baseDir" : "quickstart/",
        "filter" : "ingestion-tutorial-data.json"
      },
      "inputFormat" : {
        "type" : "json"
      }
    }
  }
}
```

额外的调整

每一个摄入任务都有一个 `tuningConfig` 部分，该部分允许用户可以调整不同的摄入参数

例如，我们添加一个 `tuningConfig`，它为本地批处理摄取任务设置目标段大小：

```
    "tuningConfig" : {
      "type" : "index_parallel",
      "maxRowsPerSegment" : 5000000
    }
```

注意：每一类摄入任务都有自己的 `tuningConfig` 类型

最终形成的规范

我们已经定义了摄取规范，现在应该如下所示：

```
{
  "type" : "index_parallel",
  "spec" : {
    "dataSchema" : {
      "dataSource" : "ingestion-tutorial",
      "timestampSpec" : {
        "format" : "iso",
        "column" : "ts"
      },
      "dimensionsSpec" : {
        "dimensions": [
          "srcIP",
          { "name" : "srcPort", "type" : "long" },
          { "name" : "dstIP", "type" : "string" },
          { "name" : "dstPort", "type" : "long" },
          { "name" : "protocol", "type" : "string" }
        ]
      },
      "metricsSpec" : [
        { "type" : "count", "name" : "count" },
        { "type" : "longSum", "name" : "packets", "fieldName" : "packets" },
        { "type" : "longSum", "name" : "bytes", "fieldName" : "bytes" },
        { "type" : "doubleSum", "name" : "cost", "fieldName" : "cost" }
      ],
      "granularitySpec" : {
        "type" : "uniform",
        "segmentGranularity" : "HOUR",
        "queryGranularity" : "MINUTE",
        "intervals" : ["2018-01-01/2018-01-02"],
        "rollup" : true
      }
    },
    "ioConfig" : {
      "type" : "index_parallel",
      "inputSource" : {
        "type" : "local",
        "baseDir" : "quickstart/",
        "filter" : "ingestion-tutorial-data.json"
      },
      "inputFormat" : {
        "type" : "json"
      }
    },
    "tuningConfig" : {
      "type" : "index_parallel",
      "maxRowsPerSegment" : 5000000
    }
  }
}
```

提交任务和查询数据

在根目录运行以下命令：

```
bin/post-index-task --file quickstart/ingestion-tutorial-index.json --url http
```

脚本运行完成后我们可以查询数据。

运行 `bin/dsql` 中运行 `select * from "ingestion-tutorial"` 查询我们摄入什么样的数据

```
$ bin/dsql
Welcome to dsql, the command-line client for Druid SQL.
Type "\h" for help.
dsql> select * from "ingestion-tutorial";

┌──────────────────────────┬────────┬──────┬───────┬─────────┬─────────┬
│ __time                   │ bytes  │ cost │ count │ dstIP   │ dstPort │ packet
├──────────────────────────┼────────┼──────┼───────┼─────────┼─────────┼
│ 2018-01-01T01:01:00.000Z │   6000 │  4.9 │     3 │ 2.2.2.2 │    3000 │      6
│ 2018-01-01T01:02:00.000Z │   9000 │ 18.1 │     2 │ 2.2.2.2 │    7000 │      9
│ 2018-01-01T01:03:00.000Z │   6000 │  4.3 │     1 │ 2.2.2.2 │    7000 │      6
│ 2018-01-01T02:33:00.000Z │  30000 │ 56.9 │     2 │ 8.8.8.8 │    5000 │     30
│ 2018-01-01T02:35:00.000Z │  30000 │ 46.3 │     1 │ 8.8.8.8 │    5000 │     30
└──────────────────────────┴────────┴──────┴───────┴─────────┴─────────┴
Retrieved 5 rows in 0.12s.

dsql>
```

输入数据变换

本教程将演示如何使用转换规范在接收期间过滤和转换输入数据

本教程我们假设您已经按照单服务器部署中描述下载了Druid，并运行在本地机器上。

完成加载本地文件、数据查询和roll-up部分内容也是非常有帮助的

样例数据

我们在 `quickstart/tutorial/transform-data.json` 中包括了样例数据，为了方便我们展示一下：

```
{"timestamp":"2018-01-01T07:01:35Z","animal":"octopus",  "location":1, "number
{"timestamp":"2018-01-01T05:01:35Z","animal":"mongoose", "location":2,"number"
{"timestamp":"2018-01-01T06:01:35Z","animal":"snake", "location":3, "number":3
{"timestamp":"2018-01-01T01:01:35Z","animal":"lion", "location":4, "number":30
```

使用转换规范加载数据

我们将使用以下规范摄取示例数据，该规范演示了转换规范的使用：

```
{
  "type" : "index_parallel",
  "spec" : {
    "dataSchema" : {
      "dataSource" : "transform-tutorial",
      "timestampSpec": {
        "column": "timestamp",
        "format": "iso"
      },
      "dimensionsSpec" : {
        "dimensions" : [
          "animal",
          { "name": "location", "type": "long" }
        ]
      },
      "metricsSpec" : [
        { "type" : "count", "name" : "count" },
        { "type" : "longSum", "name" : "number", "fieldName" : "number" },
        { "type" : "longSum", "name" : "triple-number", "fieldName" : "triple-
      ],
      "granularitySpec" : {
        "type" : "uniform",
        "segmentGranularity" : "week",
        "queryGranularity" : "minute",
        "intervals" : ["2018-01-01/2018-01-03"],
        "rollup" : true
      },
      "transformSpec": {
        "transforms": [
          {
            "type": "expression",
            "name": "animal",
            "expression": "concat('super-', animal)"
          },
          {
            "type": "expression",
            "name": "triple-number",
            "expression": "number * 3"
          }
        ],
        "filter": {
          "type":"or",
          "fields": [
            { "type": "selector", "dimension": "animal", "value": "super-mongo
            { "type": "selector", "dimension": "triple-number", "value": "300"
            { "type": "selector", "dimension": "location", "value": "3" }
          ]
        }
      }
    },
    "ioConfig" : {
      "type" : "index_parallel",
      "inputSource" : {
        "type" : "local",
        "baseDir" : "quickstart/tutorial",
        "filter" : "transform-data.json"
      },
      "inputFormat" : {
        "type" :"json"
      },
      "appendToExisting" : false
    },
    "tuningConfig" : {
      "type" : "index_parallel",
      "maxRowsPerSegment" : 5000000,
      "maxRowsInMemory" : 25000
    }
  }
}
```

在转换规范中，我们有两个表达式转换：

- `super-animal`：在 `"animal"` 列的值前加上"super-"。这将用转换后的版本覆盖 `animal` 列，因为转换的名称是 `animal`
- `triple-number`：将数字列乘以3, 这将创建一个新的三位数列。注意，我们同时接收原始列和转换列

另外，我们有一个包含三个子句的OR过滤器：

- `super-animal` 值匹配"super-mongoose"
- `triple-number` 值匹配300
- `location` 值匹配3

这个过滤器选择前3行，它将排除输入数据中的最后一个"lion"行。请注意，过滤器是在转换之后应用的。

现在提交位于 `quickstart/tutorial/transform-index.json` 的任务：

```
bin/post-index-task --file quickstart/tutorial/transform-index.json --url http
```

查询已转换的数据

运行 `bin/dsql` 提交 `select * from "transform-tutorial"` 查询来看摄入的数据：

```
dsql> select * from "transform-tutorial";
┌──────────────────────────┬──────────────────┬───────┬──────────┬────────┬──────
│ __time                   │ animal           │ count │ location │ number │ trip
├──────────────────────────┼──────────────────┼───────┼──────────┼────────┼──────
│ 2018-01-01T05:01:00.000Z │ super-mongoose   │     1 │        2 │    200 │
│ 2018-01-01T06:01:00.000Z │ super-snake      │     1 │        3 │    300 │
│ 2018-01-01T07:01:00.000Z │ super-octopus    │     1 │        1 │    100 │


└──────────────────────────┴──────────────────┴───────┴──────────┴────────┴──────
Retrieved 3 rows in 0.03s.
```

"Lion"列被丢弃， `animal` 列被转换，我们既有原始列，也有转换后的数字列。

Kerberized HDFS存储

Hadoop设置

以下配置文件需要拷贝到Druid配置文件夹：

1. 对于HDFS作为深度存储， `hdfs-site.xml` , `core-site.xml`
2. 对于数据摄入， `mapred-site.xml` , `yarn-site.xml`

HDFS文件夹和权限

1. 选择用于Druid深度存储的文件夹，例如"druid"
2. 在hdfs中所需父文件夹下创建文件夹，例如 `hdfs dfs -mkdir /druid` 或者 `hdfs dfs -mkdir /apps/druid`
3. 授予druid进程访问此文件夹的适当权限。这将确保druid能够在HDFS中创建必要的文件夹，如数据和索引日志。例如，如果druid进程以用户"root"的身份运行，那么 `hdfs dfs -chown root:root /apps/druid` 或者 `hdfs dfs -chmod 777 /apps/druid`

Druid创建了必要的子文件夹，以便在这个新创建的文件夹下存储数据和索引。

Druid设置

在 `conf/druid/common/common.runtime.properties` 中编辑 `common.runtime.properties` 以包含HDFS属性,用于该位置的文件夹与上面示例中使用的文件夹相同

common.runtime.properties

```
# Deep storage
#
# For HDFS:
druid.storage.type=hdfs
druid.storage.storageDirectory=/druid/segments
# OR
# druid.storage.storageDirectory=/apps/druid/segments

#
# Indexing service logs
#

# For HDFS:
druid.indexer.logs.type=hdfs
druid.indexer.logs.directory=/druid/indexing-logs
# OR
# druid.storage.storageDirectory=/apps/druid/indexing-logs
```

注意：注释掉文件中的本地存储和S3存储参数

同时，需要在 `conf/druid/_common/common/common.runtime.properties` 中增加hdfs-storage核心扩展：

```
#
# Extensions
#

druid.extensions.directory=dist/druid/extensions
druid.extensions.hadoopDependenciesDir=dist/druid/hadoop-dependencies
druid.extensions.loadList=["mysql-metadata-storage", "druid-hdfs-storage", "dr
```

Hadoop Jars

确保Druid有必要的jar来支持Hadoop版本

通过 `hadoop version` 命令来查看hadoop版本

在其他使用hadoop的情况（如 `WanDisco` ）下，需要保证：

1. 必要的库是可用的
2. 在 `conf/druid/_common/common.runtime.properties` 中 `druid.extensions.loadList` 增加必要的扩展

Kerberos设置

创建一个无头的keytab，它可以访问druid数据和索引。

编辑 `conf/druid/_common/common.runtime.properties` 增加下列属性：

```
druid.hadoop.security.kerberos.principal
druid.hadoop.security.kerberos.keytab
```

例如：

```
druid.hadoop.security.kerberos.principal=hdfs-test@EXAMPLE.IO
druid.hadoop.security.kerberos.keytab=/etc/security/keytabs/hdfs.headless.keyt
```

重启Druid服务

完成上述更改后，重新启动Druid。这将确保Druid与Kerberized Hadoop一起工作

架构设计

Druid有一个多进程、分布式的架构，该架构设计为云友好且易于操作。每个Druid进程都可以独立配置和扩展，在集群上提供最大的灵活性。这种设计还提供了增强的容错能力：一个组件的中断不会立即影响其他组件。

进程与服务

Druid有若干不同类型的进程，简单描述如下：

- Coordinator 进程管理集群中数据的可用性
- Overlord 进程控制数据摄取负载的分配
- Broker 进程处理来自外部客户端的查询请求
- Router 进程是一个可选进程，可以将请求路由到Brokers、Coordinators和Overlords
- Historical 进程存储可查询的数据
- MiddleManager 进程负责摄取数据

Druid进程可以按照您喜欢的方式部署，但是为了便于部署，我们建议将它们组织成三种服务器类型：Master、Query和Data。

- **Master**: 运行Coordinator和Overlord进程，管理数据可用性和摄取
- **Query**: 运行Broker和可选的Router进程，处理来自外部客户端的请求
- **Data**: 运行Historical和MiddleManager进程，执行摄取负载和存储所有可查询的数据

关于进程和服务组织的更多信息，可以查看Druid进程与服务

外部依赖

除了内置的进程类型外，Druid同时有三个外部依赖，它们旨在利用现有的基础设施（如果有的话）。

深度存储

每个Druid服务器都可以访问的共享文件存储。在集群部署中，通常使用一个像S3或HDFS这样的分布式对象存储，或者是一个网络挂载的文件系统。在单服务器部署中，通常使用本地磁盘。Druid使用深度存储来存储任何已被系统接收的数据。

Druid只使用深度存储作为数据备份，并作为在后台Druid进程之间传输数据的一种方式。为了响应查询，Historical进程不会从深层存储中读取数据，而是从本地磁盘读取在执行任何查询之前预缓存的段，这意味着Druid在查询期间不需要访问深层存储，这有助于它提供尽可能最好的查询延迟。这也意味着您必须在深层存储和所有Historical进程中都有足够的磁盘空间来存储计划加载的数据。

深度存储是Druid弹性、容错设计的重要组成部分。即使每个数据服务器都丢失并重新配置，Druid也可以从深层存储启动。

有关更多详细信息，请参见深度存储

元数据存储

元数据存储包含各种共享的系统元数据，如段可用性信息和任务信息。在集群部署中，通常使用像PostgreSQL或MySQL这样的传统RDBMS。在单服务器部署中，通常使用本地存储的Apache Derby数据库。

有关更多详细信息，请参见元数据存储

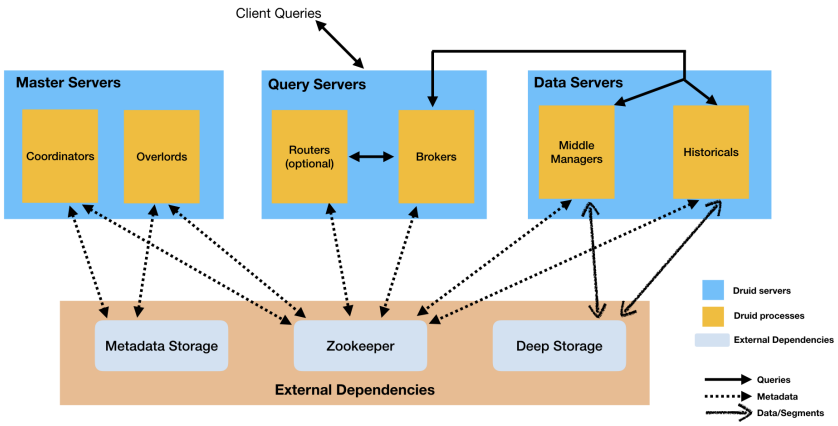
Zookeeper

用于内部服务发现、协调和领导选举。

有关更多详细信息，请参见Zookeeper

架构图

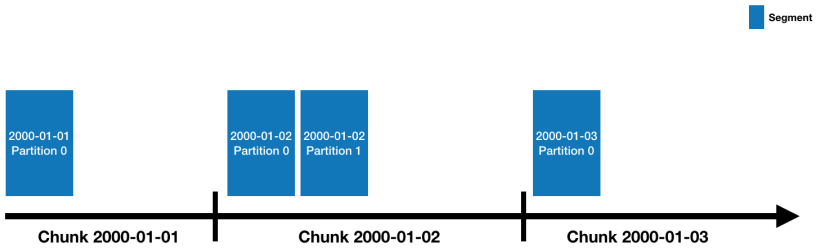
下图显示了使用建议的Master/Query/Data服务组织，查询和数据如何通过此体系结构流动：



存储设计

数据源和段

Druid数据被存储在"datasources"中，类似于传统RDBMS中的表。每一个数据源可以根据时间进行分区，可选地还可以进一步根据其他属性进行分区。每一个时间范围称为一个"块（chunk）"(例如，如果数据源按天分区，则为一天)。在一个块中，数据被分为一个或者多个"段（segments）"。每个段是一个单独的文件，一般情况下由数百万条数据组成。由于段被组织成时间块，因此有时将段视为存在于如下时间线上是有帮助的：



一个数据源可能有几个段到几十万甚至几百万个段。每个段都是在MiddleManager上创建的，但此时段是可变的和未提交的。段构建过程包括以下步骤，旨在生成一个紧凑且支持快速查询的数据文件：

- 转换为列格式
- 构建位图索引
- 使用不同的算法进行压缩
  - 字符串列id存储最小化的字典编码
  - 对位图索引做压缩
  - 所有列的类型感知压缩

周期性地，段被提交和发布，此时，它们将被写入到深度存储且变得不可更改，同时从MiddleManager移动到Historical进程。有关段的信息也写入到元数据存储中，这个信息是一个自描述的信息，包括段的schema、大小以及在深度存储中的位置，Coordinator可以根据这些信息来知道集群上应该有哪些数据是可用的

有关段文件格式的信息，请参见段文件

有关数据在Druid的建模，请参见schema设计

索引和切换(Indexing and handoff)

*索引(Indexing)*是创建新段的一种机制，*切换(handoff)*是发布新段并开始由Historical进程提供服务的机制。该机制在索引端的工作方式如下：

1. 索引任务开始运行并生成新段。必须先在索引任务构建段之前确定段的标识符，对于一个追加数据类型的任务（例如Kafka任务或者其他追加模式的索引任务），这将通过调用Overlord的"allocate" API来在现有的段集合中添加一个新的分区。对于一个重写类型的任务（例如Hadoop任务，或者一个非追加模式的索引任务）这是通过锁定间隔并创建新的版本号和新的段集来完成的。
2. 如果一个索引任务是实时任务（像kafka任务），那么段在此刻可以被立即查询，它是可用的，但是未发布。
3. 索引任务完成对段的数据读取后，会将其推送到深层存储，然后通过将记录写入元数据存储来发布。
4. 如果索引任务是实时任务，则此时它将等待Historical进程加载段。如果索引任务不是实时任务，它将立即退出。

在Coordinator和Historical方面：

1. 对于新发布的段，Coordinator会周期性（默认是1分钟）的拉取元数据存储信息
2. 当Coordinator发现一个段是发布且可以被使用的、但是不可用的状态时，它会选个一个Historical进程来加载这个段
3. Historical加载这个段并开始为其服务

4. 此时，如果索引任务正在等待切换，它将退出

段标识符

段都有一个由四部分组成的标识符，包含以下组件：

- 数据源名称
- 时间间隔（包含段的时间块，这与摄取时指定的 `segmentGranularity` 有关）
- 版本号（通常是ISO8601时间戳，对应于段集首次启动的时间）
- 分区号（整数，在datasource+interval+version中是唯一的,不一定是连续的）。

例如这个一个段标识符，数据源为 `clarity-cloud0`，时间块为 `2018-05-21T16:00:00.000Z/2018-05-21T17:00:00.000Z`，版本为 `2018-05-21T15:56:09.909Z` 以及分区编号为 `1`：

```
clarity-cloud0_2018-05-21T16:00:00.000Z_2018-05-21T17:00:00.000Z_2018-05-21T15
```

分区号为0的段（块中的第一个分区）忽略分区号，如下例所示，该段与上一个时间块位于同一时间块中，但分区号为0而不是1：

```
clarity-cloud0_2018-05-21T16:00:00.000Z_2018-05-21T17:00:00.000Z_2018-05-21T15
```

段版本

您可能想知道上一节中描述的“版本号”是用来做什么的。或者，你可能不想知道，在这种情况下对你有好处，你可以跳过这一节！

它支持批处理模式覆盖。在Druid中，如果您所做的只是附加数据，那么每个时间块只有一个版本。但是当您覆盖数据时，幕后发生的事情是，使用相同的数据源、相同的时间间隔、但更高的版本号创建一组新的段。这向Druid系统的其他部分发出了一个信号：旧版本应该从集群中删除，新版本应该替换它。

这个切换对用户来说似乎是瞬间发生的，因为Druid通过首先加载新数据（但不允许查询它）来处理这个问题，然后在新数据全部加载后，将所有新查询切换为使用这些新段。几分钟后，它会把旧的段卸载下来。

段生命周期

每个段都有一个生命周期，涉及以下三个主要领域：

1. **元数据存储**：段的元数据（一个小的JSON，通常不超过几个KB）在段构建完成后存储在元数据存储中，将段的记录插入到元数据存储中称为**发布(Publishing)**。这些元数据记录中有一个 `used` 的布尔标识，控制着段是否可查询。被实时任务创建的段在发布之前是可用的，因为它们仅在完成之时发布，并且不再接受额外的数据行
2. **深度存储**：一旦构建了一个段，在将元数据发布到元数据存储之前就立刻将段数据文件推送到深度存储
3. **可查询性**：在某些Druid数据服务器上段是可以进行查询的，如实时任务或Historical进程

可以使用Druid SQL查询 `sys.segments` 表检查当前活动段的状态，它包括以下标志：

- `is_published`：如果段元数据已发布到元数据存储且 `used` 是true的话，则为true
- `is_available`：如果段当前可用于查询（实时任务或Historical进程），则为true
- `is_realtime`：如果段仅在实时任务上可用，则为true。对于使用实时摄取的数据源，这通常从true开始，然后在发布和切换段时变为false
- `is_overshadowed`：如果段已发布（ `used` 设置为true），并且被某些其他已发布段完全覆盖，则为true。一般来说，这是一个过渡状态，处于该状态的段很快将其 `used` 标志自动设置为false

查询处理

查询首先进入Broker, Broker首先鉴别哪些段可能与本次查询有关。 段的列表总是按照时间进行筛选和修剪的，当然也可能由其他属性，具体取决于数据源的分区方式。然后，Broker将确定哪些Historical和MiddleManager为这些段提供服务、并向每个进程发送一个子查询。 Historical和MiddleManager进程接收查询、处理查询并返回结果，Broker将接收到的结果合并到一起形成最后的结果集返回给调用者。

Broker精简是Druid限制每个查询扫描数据量的一个重要方法，但不是唯一的方法。对于比Broker更细粒度级别的精简筛选器，每个段中的索引结构允许Druid在查看任何数据行之前，找出哪些行（如果有的话）与筛选器集匹配。一旦Druid知道哪些行与特定查询匹配，它就只访问该查询所需的特定列。在这些列中，Druid可以从一行跳到另一行，避免读取与查询过滤器不匹配的数据。

因此，Druid使用三种不同的技术来最大化查询性能：

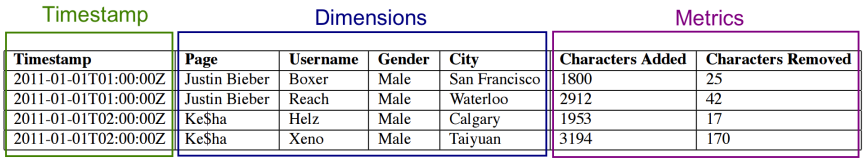
- 精简每个查询访问的段
- 在每个段中，使用索引标识必须访问哪些行
- 在每个段中，只读取与特定查询相关的特定行和列

段

ApacheDruid将索引存储在按时间分区的*段文件*中。在基本设置中，通常为每个时间间隔创建一个段文件，其中时间间隔可在 `granularitySpec` 的 `segmentGranularity` 参数中配置。为了使Druid在繁重的查询负载下运行良好，段文件大小必须在建议的300MB-700MB范围内。如果段文件大于此范围，请考虑更改时间间隔的粒度，或者对数据进行分区，并在 `partitionsSpec` 中调整 `targetPartitionSize` （此参数的建议起点是500万行）。有关更多信息，请参阅下面的**分片部分**和批处理摄取文档的**分区规范**部分。

段文件的核心数据结构

在这里，我们描述段文件的内部结构，它本质上是*列式*的：每列的数据在单独的数据结构中。通过分别存储每一列，Druid可以通过只扫描查询实际需要的列来减少查询延迟。有三种基本列类型：**时间戳列、维度列和指标列**，如下图所示：



Timestamp	Page	Username	Gender	City	Characters Added	Characters Removed
2011-01-01T01:00:00Z	Justin Bieber	Boxer	Male	San Francisco	1800	25
2011-01-01T01:00:00Z	Justin Bieber	Reach	Male	Waterloo	2912	42
2011-01-01T02:00:00Z	Ke$ha	Helz	Male	Calgary	1953	17
2011-01-01T02:00:00Z	Ke$ha	Xeno	Male	Taiyuan	3194	170

timestamp和metric列很简单：在底层，每个列都是用LZ4压缩的整数或浮点值数组。一旦查询知道需要选择哪些行，它只需解压缩这些行，提取相关行，然后使用所需的聚合运算符进行计算。与所有列一样，如果不查询一个列，则跳过该列的数据。

dimension列是不同的，因为它们支持过滤和聚合操作，所以每一个维度都需要以下三种数据结构：

1. 一个将值（通常被当做字符串）映射到整数id的字典
2. 一个使用第一步的字典进行编码的列值的列表
3. 对于列中每一个不同的值，标识哪些行包含该值的位图

为什么需要这三种数据结构？字典简单地将字符串值映射到整数id，以便于在（2）和（3）中可以紧凑的表示。（3）中的位图（也称*倒排索引*）可以进行快速过滤操作（特别是，位图便于快速进行AND和OR操作）。 最后， `GroupBy` 和 `TopN` 查询需要（2）中的值列表。换句话说，仅基于过滤器的聚合指标是不需要（2）中存储的维度值列表的。

要具体了解这些数据结构，请考虑上面示例数据中的"page"列,表示此维度的三个数据结构如下图所示：

```
1: Dictionary that encodes column values
  {
    "Justin Bieber": 0,
    "Ke$ha":         1
  }

2: Column data
  [0,
   0,
   1,
   1]

3: Bitmaps - one for each unique value of the column
  value="Justin Bieber": [1,1,0,0]
  value="Ke$ha":         [0,0,1,1]
```

注意，位图与前两个数据结构不同：前两个数据结构在数据大小上呈线性增长（在最坏的情况下），而位图部分的大小是数据大小 * 列基数的乘积。不过，压缩在这里会有帮助，因为我们知道对于"列数据"中的每一行，只有一个具有非零项的位图，这意味着高基数列将具有非常稀疏的、高度可压缩的位图。Druid使用特别适合位图的压缩算法（如Roaring位图压缩）来利用这一点特性。

多值列

如果数据源使用多值列，那么段文件中的数据结构看起来有点不同。让我们假设在上面的例子中，第二行同时标记了"Ke$ha"和"Justin Bieber"主题。在这种情况下，这三种数据结构现在看起来如下：

```
1: Dictionary that encodes column values
  {
    "Justin Bieber": 0,
    "Ke$ha":         1
  }

2: Column data
  [0,
   [0,1],  <--Row value of multi-value column can have array of values
   1,
   1]

3: Bitmaps - one for each unique value
  value="Justin Bieber": [1,1,0,0]
  value="Ke$ha":         [0,1,1,1]
                           ^
                           |
                           |
    Multi-value column has multiple non-zero entries
```

请注意列数据和Ke$ha位图中第二行的更改。如果一行中的某一列有多个值，则其在“列数据”中的条目是一个数组。此外，在“列数据”中有n个值的行在位图中将有n个非零值项。

SQL兼容的空值处理

默认情况下，Druid字符串维度列可以使用 `""` 或者 `null` ，数值列和指标列则不能表示为 `null` ，而是将 `null` 强制为 `0` 。但是，Druid还提供了一个与SQL兼容的空值处理模式，必须在系统级别通过 `Druid.generic.useDefaultValueForNull` 启用，当设置为 `false` 时，此设置将允许Druid在接收数据时创建的段中：字符串列区分 `""` 和 `null` ，数值列区分 `null` 和 `0` 。

在这种模式下，字符串维度列不包含额外的列结构，只是为 `null` 保留额外的字典条目。但是，数值列将与一个附加位图一起存储在段中，该位图标识哪些行是 `null` 值。除了略微增加段大小之外，由于需要检查 `null` 的位图，SQL兼容的空值处理在查询时也会导致性能损失，此性能开销仅对实际包含 `null` 列的场景中存在。

命名规则

段标识符通常使用数据源，时间区间的开始时间（ISO 8601格式），时间区间的结束时间（ISO 8601格式）和版本来构造。如果数据被额外的分片后超出了时间范围，则段标识符还将包含分区号。

一个示例段标识符可以是： `数据源名称_开始时间_结束时间_版本号_分区号`

段的组成

在底层，一个段由以下几个文件组成：

- `version.bin`

  4个字节，以整数表示当前段版本。 例如，对于v9段，版本为0x0、0x0、0x0、0x9

- `meta.smoosh`

  一个包含其他 `smoosh` 文件内容的元数据(文件名以及偏移量)文件

- `XXXXX.smoosh`

  这些文件中有一些是串联的二进制数据

  `smoosh` 文件代表 "smooshed" 在一起的多个文件，以减少必须打开用来容纳数据的文件描述符的数量，它们是最大为2GB的文件（以匹配Java中内存映射的ByteBuffer的限制）。 `smoosh` 文件为数据中的每个列提供单独的文件，并在 `index.drd` 文件提供有关该段的额外元数据。

  还有一个称为 `__time` 的特殊列，它表示该段的时间列。希望随着代码的发展，这种特殊性将越来越少，但就目前而言，它就像我妈妈一直告诉我的那样特殊。

在代码库中，段具有内部格式版本。当前的句段格式版本为 `v9` 。

列的格式

每列存储为两部分：

1. Jackson序列化的列描述符
2. 列二进制文件的其余部分

列描述符本质上是一个对象，它允许我们使用Jackson的多态反序列化来添加新的有趣的序列化方法，并且对代码的影响最小。它由关于列的一些元数据（它是什么类型的，它是多值的，等等）和一列序列化/反序列化逻辑组成，这些逻辑可以反序列化二进制文件的其余部分。

切分数据以创建段

数据分片

对于同一数据源，同一时间间隔内可能存在多个段。这些段在一段时间内形成一个 `块` 。根据用于切分数据的 `shardSpec` 的类型，只有当一个 `块` 完成时，Druid查询才能完成。也就是说，如果一个块由3个段组成，例如：

`sampleData_2011-01-01T02:00:00:00Z_2011-01-01T03:00:00:00Z_v1_0`

`sampleData_2011-01-01T02:00:00:00Z_2011-01-01T03:00:00:00Z_v1_1`

`sampleData_2011-01-01T02:00:00:00Z_2011-01-01T03:00:00:00Z_v1_2`

在完成对间隔 `2011-01-01T02:00:00:00Z_2011-01-01T03:00:00:00Z` 的查询之前，必须加载所有3个段。

此规则的例外是使用**线性切片规范**。线性切片规范不会强制"完整性"，即使系统中没有加载切片，查询也可以完成。例如，如果您的实时摄取任务创建了3个使用线性切片规范进行分段的段，并且系统中只加载了其中的两个段，那么查询将只返回这两个段的结果。

Schema更改

替换段

Druid使用数据源、间隔、版本和分区号唯一地标识段。只有在为某个时间粒度创建多个段时，分区号才在段id中可见。例如，如果有小时段，但一小时内的数据量超过单个段的容量，则可以为同一小时创建多个段。这些段将共享相同的数据源、间隔和版本，但具有线性增加的分区号。

```
foo_2015-01-01/2015-01-02_v1_0
foo_2015-01-01/2015-01-02_v1_1
foo_2015-01-01/2015-01-02_v1_2
```

在上述段的实例中， `dataSource = foo` , `interval = 2015-01-01/2015-01-02` , `version = v1` , and `partitionNum = 0` 。 如果在以后的某个时间点，使用新的schema重新索引数据，则新创建的段将具有更高的版本id。

```
foo_2015-01-01/2015-01-02_v2_0
foo_2015-01-01/2015-01-02_v2_1
foo_2015-01-01/2015-01-02_v2_2
```

Druid批量索引任务（基于Hadoop或基于IndexTask）保证了间隔内的的原子更新。在我们的例子中，在 `2015-01-01/2015-01-02` 的所有 `v2` 段加载到Druid集群之前，查询只使用 `v1` 段。加载完所有 `v2` 段并可查询后，所有查询都将忽略 `v1` 段并切换到 `v2` 段。不久之后， `v1` 段将从集群中卸载。

请注意，跨越多个段间隔的更新在每个间隔内都是原子的, 但是在整个更新过程中它们不是原子的。例如，您有如下段：

```
foo_2015-01-01/2015-01-02_v1_0
foo_2015-01-02/2015-01-03_v1_1
foo_2015-01-03/2015-01-04_v1_2
```

`v2` 段将在构建后立即加载到集群中，并在段重叠的时间段内替换 `v1` 段。在完全加载 `v2` 段之前，集群可能混合了 `v1` 和 `v2` 段。

```
foo_2015-01-01/2015-01-02_v1_0
foo_2015-01-02/2015-01-03_v2_1
foo_2015-01-03/2015-01-04_v1_2
```

在这种情况下，查询可能会命中 `v1` 和 `v2` 段的混合.

段之间的不同schemas

同一数据源的Druid段可能有不同的schema。如果一个字符串列（维度列）存在于一个段中而不是另一个段中，则涉及这两个段的查询仍然有效。对缺少维度的段的查询将表现为该维度只有空值。类似地，如果一个段有一个数值列（指标列），而另一个没有，那么查询缺少指标列的段通常会"做正确的事情", 在缺失的指标上做聚合操作也就是缺失的。

进程和服务

进程类型

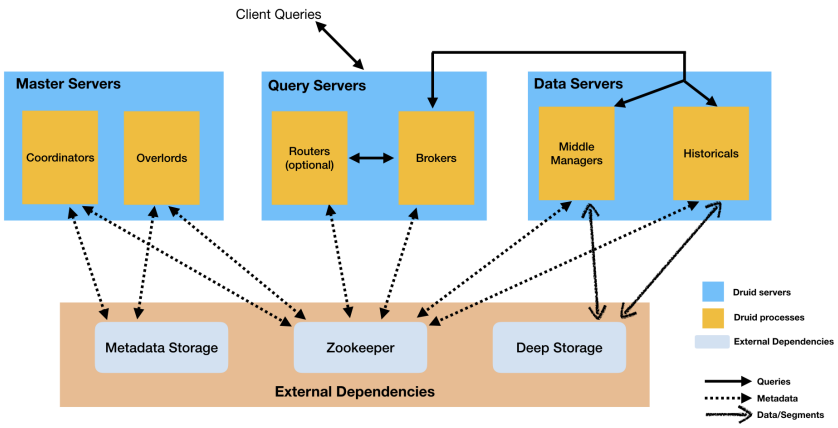
Druid有以下几种进程类型：

- Coordinator
- Overlord
- Broker
- Historical
- MiddleManager 和 Peons
- Indexer(可选)
- Router(可选)

服务类型

Druid进程可以按照您喜欢的任何方式部署，但是为了便于部署，我们建议将它们组织成三种服务器类型：

- **Master**
- **Query**
- **Data**



本节描述Druid进程和建议的Master/Query/Data server组织，如上面的架构图所示。

Master服务

Master服务管理数据的摄取和可用性：它负责启动新的摄取作业并协调下面描述的"Data服务"上数据的可用性。

在Master服务中，功能分为两个进程：Coordinator和Overlord。

Coordinator进程

Coordinator 监视Data服务中的Historical进程，它们负责将数据段分配给特定的服务器，并确保数据段在各个Historical之间保持良好的平衡。

Overlord进程

Overlord 监视Data服务中的MiddleManager进程，并且是Druid数据接收的控制器。它们负责将接收任务分配给MiddleManager，并协调数据段的发布。

Query服务

Query服务提供用户和客户端应用程序交互，将查询路由到Data服务或其他Query服务（以及可选的代理Master服务请求）。

在Query服务中，功能上分为两个进程：Broker和Router。

Broker进程

Broker从外部客户端接收查询并将这些查询转发到Data服务器，当Broker接收到子查询的结果时，它们会合并这些结果并将其返回给调用者。用户通常查询Broker，而不是直接查询Data服务中的Historical或MiddleManager进程。

Router进程（可选）

Router进程是*可选*的进程，相当于是为Druid Broker、Overlord和Coordinator提供一个统一的API网关。Router是可选的，因为也可以直接与Druid的Broker、Overlord和Coordinator。

Router还运行着Druid控制台，一个用于数据源、段、任务、数据进程（Historical和MiddleManager）和Coordinator动态配置的管理UI。用户还可以在控制台中运行SQL和本地Druid查询。

Data服务

Data服务执行摄取作业并存储可查询数据。

在Data服务中，根据功能被分为两个进程：Historical和MiddleManager。

Historical进程

Historical 进程是处理存储和查询"Historical"数据（包括系统中已提交足够长时间的任何流数据）的工作程序。Historical进程从深层存储下载段并响应有关这些段的查询，他们不接受写操作。

MiddleManager进程

MiddleManager 进程处理将新数据摄取到集群中的操作，他们负责读取外部数据源并发布新的Druid段。

Peon进程

Peon 进程是由MiddleManagers生成的任务执行引擎。每个Peon运行一个单独的JVM，负责执行一个任务。Peon总是和孕育它们的MiddleManager在同一个主机上运行。

Indexer进程（可选）

Indexer 进程是MiddleManager和Peon的替代方法。Indexer在单个JVM进程中作为单个线程运行任务，而不是为每个任务派生单独的JVM进程。

与MiddleManager + Peon系统相比，Indexer的设计更易于配置和部署，并且能够更好地实现跨任务的资源共享。Indexer是一种较新的功能，由于其内存管理系统仍在开发中，因此目前被指定为实验性的特性。它将在Druid的未来版本中继续成熟。

通常，您可以部署MiddleManagers或indexer，但不能同时部署两者。

服务混合部署的利弊

Druid进程可以基于上面描述的Master/Data/Query服务组织进行混合部署，这种部署方式通常会使大多数集群更好地利用硬件资源。

但是，对于非常大规模的集群，可以分割Druid进程，使它们在单独的服务器上运行，以避免资源争用。

本节介绍与进程混合部署相关的指南和配置参数。

Coordinator和Overlord

Coordinator进程的工作负载往往随着集群中段的数量而增加。Overlord的工作量也会根据集群中的分段数而增加，但程度要比Coordinator小。

在具有非常大量的段的集群中，可以将Coordinator进程和Overlord进程分开，以便为Coordinator进程的分段平衡工作负载提供更多资源。

统一进程

通过设置 `druid.Coordinator.asOverlord.enabled` 属性，Coordinator进程和Overlord进程可以作为单个组合进程运行。

有关详细信息，请参阅Coordinator配置。

Historical和MiddleManager

对于更高级别的数据摄取或查询负载，将Historical进程和MiddleManager进程部署在不同的主机上以避免CPU和内存争用。

Historical还受益于为**内存映射段**提供可用内存，这也是分别部署Historical和MiddleManager进程的另一个原因。

Coordinator进程

配置

对于Apache Druid的Coordinator进程配置，详见 Coordinator配置

HTTP

对于Coordinator的API接口，详见 Coordinator API

综述

Druid Coordinator程序主要负责段管理和分发。更具体地说，Druid Coordinator进程与Historical进程通信，根据配置加载或删除段。Druid Coordinator负责加载新段、删除过时段、管理段复制和平衡段负载。

Druid Coordinator定期运行，每次运行之间的时间是一个可配置的参数。每次运行Druid Coordinator时，它都会在决定要采取的适当操作之前评估集群的当前状态。与Broker和Historical进程类似，Druid Coordinator维护了一个用于当前集群信息的Zookeeper集群连接。Coordinator还维护到数据库的连接，该数据库包含有关可用段和规则的信息。可用段存储在段表中，并列出应加载到集群中的所有段。规则存储在规则表中，并指示应如何处理段。

在Historical进程为任何未分配的段提供服务之前，首先按容量对每个层的可用Historical进程进行排序，最小容量的服务器具有最高优先级。未分配的段总是分配给具有最小能力的进程，以保持进程之间的平衡级别。在为Historical进程分配新段时，Coordinator不直接与该进程通信；而是在ZK中的Historical进程加载队列路径下创建有关该新段的一些临时信息。一旦看到此请求，Historical进程将加载段并开始为其提供服务。

运行

```
org.apache.druid.cli.Main server coordinator
```

规则

可以根据一组规则自动从集群中加载和删除段。有关规则的详细信息，请参阅规则配置。

清理段

每次运行时，Druid Coordinator都会将数据库中可用段的列表与集群中的当前段进行比较,不在数据库中但仍在集群中服务的段将被标记并附加到删除列表中,被遮蔽的段（它们的版本太旧，它们的数据被更新的段替换）也会被丢弃。

段可用性

如果一个Historical进程由于任何原因重新启动或变得不可用，Druid Coordinator将注意到一个Historical进程已经丢失，并将该进程服务的所有段视为被删除。给定足够的时间后，这些段可以重新分配给集群中的其他Historical进程。然而，每个被删除的片段并不会立即被遗忘，取而代之的是一种过渡数据结构，它存储所有丢弃的具有相关生存期的段。生存期表示Coordinator不会重新分配丢弃的段的一段时间。因此，如果某个Historical进程在短时间内变得不可用并再次可用，则该Historical进程将启动并从其缓存中服务段，而不会在集群中重新分配任何这些段。

平衡段负载

为了确保在集群中跨Historical进程均匀分布段，Coordinator进程将在每次进程运行时查找每个Historical进程所服务的所有段的总大小。对于集群中的每个Historical进程层，Coordinator进程将确定利用率最高的Historical进程和利用率最低的Historical进程。计算两个进程之间利用率的百分比差异，如果结果超过某个阈值，则会将多个段从利用率最高的进程移动到利用率最低的进程。每次运行Coordinator时，可以从一个进程移动到另一个进程的段数有一个可配置的限制。要移动的段是随机选择的，只有在计算结果表明最高服务器和最低服务器之间的百分比差异减小时才会移动。

合并段

每次运行时，Druid Coordinator都通过合并小段或拆分大片段来压缩段。当您的段没有进行段大小（可能会导致查询性能下降）优化时，该操作非常有用。有关详细信息，请参见段大小优化。

Coordinator首先根据段搜索策略查找要压缩的段。找到某些段后，它会发出压缩任务来压缩这些段。运行压缩任务的最大数目为 `min(sum of worker capacity * slotRatio, maxSlots)` 。请注意，即使 `min(sum of worker capacity * slotRatio, maxSlots)` = 0，如果为数据源启用了压缩，则始终会提交至少一个压缩任务。请参阅压缩配置API和压缩配置以启用压缩。

压缩任务可能由于以下原因而失败:

- 如果压缩任务的输入段在开始前被删除或覆盖，则该压缩任务将立即失败。
- 如果优先级较高的任务获取与压缩任务的时间间隔重叠的时间块锁，则压缩任务失败。

一旦压缩任务失败，Coordinator只需再次检查失败任务间隔中的段，并在下次运行中发出另一个压缩任务。

段搜索策略

新段优先策略

在每次Coordinator运行时，此策略都会按从最新到最旧的顺序查找时间块，并检查这些时间块中的段是否需要压缩。如果满足以下所有条件，则需要研所一组段：

1. 时间块中段的总大小小于或等于配置的 `inputSegmentSizeBytes`
2. 段尚未被压缩，或者自上次压缩以来压缩规范(Compaction Spec)已更新，特别是 `maxRowsPerSegment` 、 `maxTotalRows` 和 `indexSpec`

下面是一些细节和一个例子。假设我们有两个数据源（`foo`，`bar`），如下所示：

- `foo`
  - `foo_2017-11-01T00:00:00.000Z_2017-12-01T00:00:00.000Z_VERSION`
  - `foo_2017-11-01T00:00:00.000Z_2017-12-01T00:00:00.000Z_VERSION_1`
  - `foo_2017-09-01T00:00:00.000Z_2017-10-01T00:00:00.000Z_VERSION`
- `bar`
  - `bar_2017-10-01T00:00:00.000Z_2017-11-01T00:00:00.000Z_VERSION`
  - `bar_2017-10-01T00:00:00.000Z_2017-11-01T00:00:00.000Z_VERSION_1`

假设每一个段大小为10MB且从未被压缩过，因为 `2017-11-01T00:00:00.000Z/2017-12-01T00:00:00.000Z` 是最近的时间段，所以该策略首先返回两个即将压缩的段 `foo_2017-11-01T00:00:00.000Z_2017-12-01T00:00:00.000Z_VERSION` 和 `foo_2017-11-01T00:00:00.000Z_2017-12-01T00:00:00.000Z_VERSION_1` 。

如果Coordinator还有足够的用于压缩任务的插槽，该策略则继续搜索剩下的段并返回 `bar_2017-10-01T00:00:00.000Z_2017-11-01T00:00:00.000Z_VERSION` 和 `bar_2017-10-01T00:00:00.000Z_2017-11-01T00:00:00.000Z_VERSION_1` 。最后，因为在 `2017-09-01T00:00:00.000Z/2017-10-01T00:00:00.000Z` 时间间隔中只有一个段，所以 `foo_2017-09-01T00:00:00.000Z_2017-10-01T00:00:00.000Z_VERSION` 段也会被选择。

搜索的起点可以通过 skipOffsetFromLatest 来更改设置。如果设置了此选项，则此策略将忽略范围内的时间段（最新段的结束时间 - `skipOffsetFromLatest`），该配置项主要是为了避免压缩任务和实时任务之间的冲突。请注意，默认情况下，实时任务的优先级高于压缩任务。如果两个任务的时间间隔重叠，实时任务将撤消压缩任务的锁，从而终止压缩任务。

> [!WARNING] 当有很多相同间隔的小段，并且它们的总大小超过 `inputSegmentSizeBytes` 时，此策略当前无法处理这种情况。如果它找到这样的段，它只会跳过它们。

Coordinator控制台

Druid Coordinator公开了一个web GUI，用于显示集群信息和规则配置。有关详细信息，请参阅Coordinator控制台。

FAQ

1. **客户端是否曾和Coordinator进行通信?**

   Coordinator进程不参与查询。

   Historical也不会直接与Coordinator进行通信。Coordinator通过Zookeeper来通知Historical来加载或者卸载段，但是Historical完全不知道Coordinator。

   Broker也不会直接与Coordinator进行通信。Broker通过Historical经ZK公开的元数据来理解数据拓扑，并且也完全不知道Coordinator。

2. **Coordinator进程在其他进程之前还是之后启动有关系吗?**

   不。如果Druid Coordinator没有启动，集群中将不会加载新的段，也不会删除过时的段。但是，Coordinator可以随时启动，并且在可配置的延迟之后，将开始运行协调任务。

这也意味着，如果有一个正在工作的集群，并且所有的Coordinator都死掉了，那么集群将继续工作，只是它的数据拓扑不会发生任何变化。

Overload进程

配置

对于Apache Druid的Overlord进程配置，详见 Overlord配置

HTTP

对于Overlord的API接口，详见 Overlord API

综述

Overlord进程负责接收任务、协调任务分配、创建任务锁并将状态返回给调用方。Overlord可以配置为本地模式运行或者远程模式运行（默认为本地）。在本地模式下，Overlord还负责创建执行任务的Peon， 在本地模式下运行Overlord时，还必须提供所有MiddleManager和Peon配置。本地模式通常用于简单的工作流。在远程模式下，Overlord和MiddleManager在不同的进程中运行，您可以在不同的服务器上运行每一个进程。如果要将索引服务用作所有Druid索引的单个端点，建议使用此模式。

Overlord控制台

Druid Overlord公开了一个web GUI，用于管理任务和worker。有关详细信息，请参阅Overlord控制台。

worker黑名单

如果一个MiddleManager的任务失败超过阈值，Overlord会将这些MiddleManager列入黑名单。不超过20%的MiddleManager可以被列入黑名单，被列入黑名单的MiddleManager将定期被列入白名单。

以下变量可用于设置阈值和黑名单超时：

```
druid.indexer.runner.maxRetriesBeforeBlacklist
druid.indexer.runner.workerBlackListBackoffTime
druid.indexer.runner.workerBlackListCleanupPeriod
druid.indexer.runner.maxPercentageBlacklistWorkers
```

自动缩放

目前采用的自动缩放机制与我们的部署基础设施紧密耦合，但框架应该适用于其他实现。我们对现有机制的新实现或扩展高度开放。在我们自己的部署中，MiddleManager进程是Amazon AWS EC2节点，它们被设置为在galaxy环境中注册自己。

如果启用了自动缩放，则当任务处于挂起状态太长时间时，可能会添加新的MiddleManagers。如果MiddleManager在一段时间内没有运行任何任务，则可能会被终止。

Historical

配置

对于Apache Druid Historical的配置，请参见 Historical配置

HTTP

Historical的API列表，请参见 Historical API

运行

```
org.apache.druid.cli.Main server historical
```

加载和服务段

每个Historical都保持与Zookeeper的持续连接，并监视一组可配置的Zookeeper路径以获取新的段信息。Historical不直接与Coordinator通信，而是依赖Zookeeper进行协调。

Coordinator 负责通过在与Historical关联的加载队列路径下创建一个短暂的Zookeeper条目来将新的段分配给Historical。有关Coordinator如何将段分配给Historical的更多信息，请参阅 Coordinator。

当Historical在其加载队列路径中注意到新的加载队列条目时，它将首先检查本地磁盘目录（缓存）以获取有关段的信息。如果缓存中不存在有关段的信息，Historical将从Zookeeper下载有关新段的元数据，此元数据包括段在深层存储中的位置以及如何解压缩和处理段的规范。有关段的元数据和一般的Druid段的更多信息，请参见 段。一旦一个Historical完成了对一个段的处理，这个段就会在Zookeeper中与该进程相关联的服务段路径下被宣布，同时，该段可供查询。

从缓存加载和服务段

回想一下，当Historical在其加载队列路径中注意到一个新的段条目时，Historical首先检查其本地磁盘上的一个可配置的缓存目录以查看该段以前是否下载过。如果本地缓存项已经存在，Historical将直接从磁盘读取段二进制文件并加载段。

当Historical首次启动时也会利用段缓存。Historical启动时，将搜索其缓存目录，并立即加载和服务找到的所有段。此功能允许Historical启动后皆可以对这些段进行查询。

查询段

有关查询Historical的详细信息，请参阅 数据查询。

Historical可以被配置记录和报告每个服务查询的指标。

MiddleManager进程

配置

对于Apache Druid MiddleManager配置，可以参见索引服务配置

HTTP

对于MiddleManager的API接口，详见 MiddleManager API

综述

MiddleManager进程是执行提交的任务的工作进程。MiddleManager将任务转发给运行在不同jvm中的Peon。我们为每个任务设置单独的jvm的原因是为了隔离资源和日志。每个Peon一次只能运行一个任务，但是，一个MiddleManager可能有多个Peon。

运行

```
org.apache.druid.cli.Main server middleManager
```

Broker

配置

对于Apache Druid Broker的配置，请参见 Broker配置

HTTP

对于Broker的API的列表，请参见 Broker API

综述

在分布式的Druid集群中，Broker是一个查询的路由进程。Broker了解所有已经发布到ZooKeeper的元数据，了解在哪些进程存在哪些段，然后将查询路由到以便它们可以正确命中的进程。Broker还将来自所有单个进程的结果集合并在一起。在启动时，Historical会在Zookeeper中注册它们自身以及它们所服务的段。

运行

```
org.apache.druid.cli.Main server broker
```

转发查询

大多数Druid查询都包含一个interval对象，该对象指示请求数据的时间跨度。同样，Druid段被划分为包含一定时间间隔的数据，并且段分布在集群中。假设有一个具有7个段的简单数据源，其中每个段包含一周中给定一天的数据，对数据源发出的任何查询如果超过一天的数据，都将命中多个段，这些段可能分布在多个进程中，因此，查询可能会命中多个进程。

为了确定将查询转发到哪个进程，Broker首先从Zookeeper中的信息构建一个全局视图。Zookeeper中维护有关 Historical 和流式摄取 Peon 过程及其所服务的段的信息。对于Zookeeper中的每个数据源，Broker都会构建一个段的时间线以及为它们提供服务的进程。当接收到针对特定数据源和间隔的查询时，Broker将对与查询数据源关联的查询间隔时间线执行查找，并检索包含所查询数据的进程。然后，Broker将查询向下转发到所选进程。

缓存

Broker使用具有LRU缓存失效策略的缓存，Broker缓存按段存储结果。缓存可以是每个Broker的本地缓存，也可以使用外部分布式缓存（如 memcached)跨多个进程共享。每次Broker进程收到查询时，它首先将查询映射到一组段，这些段结果的子集可能已经存在于缓存中，并且可以直接从缓存中提取结果。对于缓存中不存在的任何段结果，Broker将查询转发到Historical。一旦Historical返回其结果，Broker将把这些结果存储在缓存中。实时段从不被缓存，因此对实时数据的请求总是被转发到实时进程。实时数据一直在变化，缓存结果是不可靠的。

Router

> [!WARNING] Router是一个可选的和实验性的特性，因为它在Druid集群架构中的推荐位置仍在不断发展。然而，它已经在生产中经过了测试，并且承载了强大的Druid控制台，所以您应该放心地部署它。

Apache Druid Router用于将查询路由到不同的Broker。默认情况下，Broker根据规则 设置路由查询。例如，如果将最近1个月的数据加载到一个 `热集群` 中，则可以将最近一个月内的查询路由到一组专用的Broker,超出此范围的查询将路由到另一组Broker。该设置的主要功能是为了提供查询隔离，以便对较重要数据的查询不会受到对较不重要数据的查询的影响。

出于查询路由的目的，如果您有一个TB数据规模的Druid集群，您应该只使用Router进程。

除了查询路由，Router还运行 Druid控制台, 一个用于数据源、段、任务、数据进程（Historical和MiddleManager）和Coordinator动态配置的管理UI。用户还可以在控制台中运行SQL和本地Druid查询。

配置

对于Apache Druid Router的配置，请参见 Router 配置

HTTP

对于Router的API列表，请参见 Router API

运行

```
org.apache.druid.cli.Main server router
```

作为管理代理的Router

Router可以配置为将转发请求到活跃的Coordinator和Overlord。 该功能在非活跃 -> 活跃的Coordinator和Overlord的HTTP重定向机制无法正常工作时的情况下(服务器位于负载平衡器后面，重定向中使用的主机名只能在内部解析等)设置高可用集群很有帮助。

开启管理代理

要启用此功能，请在Router的 `runtime.properties` 中设置以下内容：

```
druid.router.managementProxy.enabled=true
```

管理代理路由

管理代理支持隐式和显式路由。隐式路由是根据Druid API路径从原始请求路径来确定目标的路由。对于Coordinator，约定是 `/druid/Coordinator/*` 。 对于Overlord，约定是 `/druid/indexer/*` 。由于使用管理代理不需要修改API请求，只

需要向Router而不是Coordinator或Overlord发出请求，所以是非常方便的。大多数Druid API请求都可以隐式路由。

显式路由是指到Router的请求包含路径前缀的路由，该前缀指示请求应路由到哪个进程。对于Coordinator，这个前缀是 `/proxy/coordinator`；对于Overlord，这个前缀是 `/proxy/overlord`。这对于目标不明确的API调用是必需的, 例如，所有Druid进程上都存在 `/status` API，因此需要使用显式路由来指示代理目标。

汇总如下：

请求路由	目标	重写路由	
`/druid/coordinator/*`	Coordinator	`/druid/coordinator/*`	`router coordi`
`/druid/indexer/*`	Overlord	`/druid/indexer/*`	`router overlo`
`/proxy/coordinator/*`	Coordinator	`/*`	`router coordi`
`/proxy/overlord/*`	Overlord	`/*`	`router -> ove`

Router策略

Router有一个可配置的策略列表，用于选择将查询路由到哪个Broker。策略的顺序很重要，因为一旦策略条件匹配，就会选择一个Broker。

timeBoundary

```
{
  "type":"timeBoundary"
}
```

包含这个策略的话意味着所有的 **timeRange** 查询将全部路由到高优先级的Broker

priority

```
{
  "type":"priority",
  "minPriority":0,
  "maxPriority":1
}
```

优先级设置为小于 `minPriority` 的查询将路由到最低优先级Broker, 优先级设置为大于 `maxPriority` 的查询将路由到最高优先级Broker。默认情况下， `minPriority` 为0， `maxPriority` 为1。使用这些默认值的时候，如果发送优先级为0（默认查询优先级为0）的查询，则查询将跳过优先级选择逻辑。

JavaScript 允许使用JavaScript函数定义任意路由规则。将配置和要执行的查询传递给函数，并返回它应该路由到的层（tier），或者对于默认层返回null。

示例：将包含三个以上聚合器的查询发送到最低优先级Broker的函数：

```
{
  "type" : "javascript",
  "function" : "function (config, query) { if (query.getAggregatorSpecs && que
}
```

> [!WARNING] 默认情况下禁用基于JavaScript的功能。有关使用Druid的JavaScript功能的指南，包括如何启用它的说明，请参阅Druid JavaScript编程指南。

Avatica查询平衡

具有给定连接ID的所有Avatica JDBC请求都必须路由到同一个Broker，因为Druid Broker之间不共享连接状态。

为了实现这一点，Druid提供了两个内置的平衡器，它们分别使用请求的连接ID的集合哈希和一致性哈希来将请求分配给Broker。

请注意，当使用多个Router时，所有Router应具有相同的平衡器配置，以确保它们做出相同的路由决策。

集合哈希平衡器

这个平衡器使用Avatica请求的连接ID上的 集合哈希 来将请求分配给Broker。

要使用此平衡器，请指定以下属性：

```
druid.router.avatica.balancer.type=consistentHash
```

如果 `druid.router.avatica.balancer` 配置项没有被设置，Router将同样默认使用集合哈希平衡器。

一致性哈希平衡器

这个平衡器使用Avatica请求的连接ID上的 集合哈希 来将请求分配给Broker。

要使用此平衡器，请指定以下属性：

```
druid.router.avatica.balancer.type=consistentHash
```

这是为实验目的而提供的非默认实现。一致哈希器在初始化和Brokers更改时的设置时间较长，但在使用5个Broker进行测试时，它的Broker分配时间比集合哈希器快。这两种实现的基准已经在 `ConsistentHasherBenchmark` 和 `RendezvousHasherBenchmark` 中提供。一致哈希器还需要锁定，而集合哈希器不需要。

生产环境配置示例

在本例中，我们的生产集群中有两个层： `hot` 层和 `默认层(_default_tier)` 。对 `hot` 层的查询通过 `broker-hot` 路由，对 `默认层` 的查询通过 `broker-cold` 路由。如果发生任何异常或网络问题，查询将被路由到 `broker-cold` 。在我们的示例中，我们运行的是一个c3.2xlarge EC2实例。我们假设 `common.runtime.properties` 已经存在。

JVM设置：

```
-server
-Xmx13g
-Xms13g
-XX:NewSize=256m
-XX:MaxNewSize=256m
-XX:+UseConcMarkSweepGC
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps
-XX:+UseLargePages
-XX:+HeapDumpOnOutOfMemoryError
-XX:HeapDumpPath=/mnt/galaxy/deploy/current/
-Duser.timezone=UTC
-Dfile.encoding=UTF-8
-Djava.io.tmpdir=/mnt/tmp

-Dcom.sun.management.jmxremote.port=17071
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
```

Runtime.properties:

```
druid.host=#{IP_ADDR}:8080
druid.plaintextPort=8080
druid.service=druid/router

druid.router.defaultBrokerServiceName=druid:broker-cold
druid.router.coordinatorServiceName=druid:coordinator
druid.router.tierToBrokerMap={"hot":"druid:broker-hot","_default_tier":"druid:
druid.router.http.numConnections=50
druid.router.http.readTimeout=PT5M

# Number of threads used by the Router proxy http client
druid.router.http.numMaxThreads=100

druid.server.http.numThreads=100
```

Indexer

> [!WARNING] 索引器是一个可选的和实验性的功能, 其内存管理系统仍在开发中，并将在以后的版本中得到显著增强。

Apache Druid索引器进程是MiddleManager + Peon任务执行系统的另一种可替代选择。索引器在单个JVM进程中作为单独的线程运行任务，而不是为每个任务派生单独的JVM进程。

与MiddleManager + Peon系统相比，Indexer的设计更易于配置和部署，并且能够更好地实现跨任务的资源共享。

配置

对于Apache Druid Indexer进程的配置，请参见 Indexer配置

HTTP

Indexer进程与MiddleManager共用API

运行

```
org.apache.druid.cli.Main server indexer
```

任务资源共享

以下资源在索引器进程内运行的所有任务中共享。

查询资源 查询处理线程和缓冲区在所有任务中共享。索引器将为来自所有任务共享的单个端点的查询提供服务。

如果启用了查询缓存，则查询缓存也将在所有任务中共享。

服务端HTTP线程 索引器维护两个大小相等的HTTP线程池。

一个池专门用于Overlord和Indexer之间的任务控制消息（"会话处理程序线程"），另一个池用于处理所有其他HTTP请求。

池的大小由 `druid.server.http.numThreads` 配置配置（例如，如果设置为10，则将有10个会话处理程序线程和10个非会话处理程序线程）。

除了这两个池之外，还为查找处理分配了两个单独的线程。如果不使用查找，则不会使用这些线程。

内存共享 索引器使用 `druid.worker.globalIngestionHeapLimitBytes` 配置对其运行的所有任务施加全局堆限制。

此全局限制平均分配给 `druid.worker.capacity` 配置的任务槽数。

要应用每个任务堆的限制，索引器将覆盖任务优化配置(task tuning)中的 `maxBytesInMemory` （即忽略默认值或任何用户配置的值）。 `maxRowsInMemory` 也将被重写为本质上不受限制的值：索引器不支持行限制。

默认情况下，`druid.worker.globalIngestionHeapLimitBytes` 设置为可用JVM堆的1/6。选择此默认值是为了在使用MiddleManager/Peon系统时与任务优化配置中 `maxBytesInMemory` 的默认值对齐，该系统也是JVM堆的1/6。

堆内存中保留的行的峰值使用量与任务优化配置中的 `maxBytesInMemory` 和 `maxPendingPersistent` 属性之间的交互有关。当任务在堆中保留的行数据量达到 `maxBytesInMemory` 指定的限制时，任务将持久化堆中的行数据。在持久化任务启动后，任务可以在持久化任务运行时再次摄取行数据的 `maxBytesInMemory` 字节。

这意味着行数据的堆使用峰值可以达到 `maxBytesInMemory * (2 + maxPendingResistent)` 。 `maxPendingPersistent` 的默认值为0，允许1一个持久化任务与摄取工作同时运行。

堆的其余部分保留用于查询处理、段持久/合并操作以及其他堆使用。

并发段持久/合并限制 为了帮助减少峰值内存使用，索引器对所有正在运行的任务中并发的段持久/合并操作的数量进行了限制。

默认情况下，并发持久性/合并操作的数量限制为 `(druid.worker.capacity/2)` ，四舍五入。可以使用 `druid.worker.numConcurrentMerges` 属性配置此限制。

当前限制

使用索引器时，当前不支持单独的任务日志；所有任务日志消息都将记录在索引器进程日志中。

索引器当前对每个任务施加相同的内存限制。在以后的版本中，将删除每个任务的内存限制，并且只应用全局限制。同时合并的限制也将被删除。

在以后的版本中，将动态管理每个任务的内存使用情况。请参阅 https://github.com/apache/druid/issues/7900 以了解有关索引器未来增强功能的详细信息。

Peons

配置

对于Apache Druid Peon配置，可以参见 Peon查询配置 和 额外的Peon配置

HTTP

对于Peon的API接口，详见 Peon API

Peon在单个JVM中运行单个任务。MiddleManager负责创建运行任务的Peon。Peon应该很少（如果为了测试目的）自己运行。

运行

Peon应该很少独立于MiddleManager，除非出于开发目的。

```
org.apache.druid.cli.Main internal peon <task_file> <status_file>
```

任务文件包含任务JSON对象。状态文件指示将输出任务状态的位置。

深度存储

Apache Druid不提供的存储机制，深度存储是存储段的地方。深度存储基础结构定义了数据的持久性级别，只要Druid进程能够看到这个存储基础结构并获得存储在上面的段，那么无论丢失多少Druid节点，都不会丢失数据。如果段在深度存储层消失了，则这些段中存储的任何数据都将丢失。

本地挂载

本地装载也可用于存储段。这使得您可以使用本地文件系统或任何可以在本地挂载的东西，如NFS、Ceph等来存储段。这是默认的深度存储实现。

为了使用本地装载进行深层存储，需要在公共配置中设置以下配置：

属性	可能的取值	描述	默认值
`druid.storage.type`	local		必须设置
`druid.storage.storageDirectory`		存储段的目录	必须设置

注意，通常应该将 `druid.storage.storageDirectory` 设置为与 `druid.segmentCache.locations` 和 `druid.segmentCache.infoDir` 不同的目录。

如果在本地模式下使用Hadoop Indexer，那么只需给它一个本地目录作为输出目录，它就可以工作了。

S3适配

请看druid-s3-extensions扩展文档

HDFS

请看druid-hdfs-extensions扩展文档

其他深度存储

对于另外的深度存储等，可以参见扩展列表

元数据存储

元数据存储是Apache Druid的一个外部依赖。Druid使用它来存储系统的各种元数据，但不存储实际的数据。下面有许多用于各种目的的表。

Derby是Druid的默认元数据存储，但是它不适合生产环境。MySQL和PostgreSQL是更适合生产的元数据存储。

> [!WARNING] 元数据存储存储了Druid集群工作所必需的整个元数据。对于生产集群，考虑使用MySQL或PostgreSQL而不是Derby。此外，强烈建议设置数据库的高可用，因为如果丢失任何元数据，将无法恢复。

使用Derby

将以下内容添加到您的Druid配置中：

```
druid.metadata.storage.type=derby
druid.metadata.storage.connector.connectURI=jdbc:derby://localhost:1527//opt/v
```

MySQL

参见mysql-metadata-storage扩展文档

PostgreSQL

参见postgresql-metadata-storage扩展文档

添加自定义的数据库连接池属性

注意：`username`、`password`、`connectURI`、`validationQuery`、`testOnBorrow`这些属性不能通过`druid.metadata.storage.connector.dbcp`属性设置，这些必须通过`druid.metadata.storage.connector`属性设置。

支持的属性示例：

```
druid.metadata.storage.connector.dbcp.maxConnLifetimeMillis=1200000
druid.metadata.storage.connector.dbcp.defaultQueryTimeout=30000
```

全部列表请查看 基本数据源配置

元数据存储表

段表

这是由`druid.metadata.storage.tables.segments`属性决定的。

此表存储了系统中可用段的元数据。Coordinator 对表进行轮询，以确定可用于在系统中查询的段集。该表有两个主功能列，其他列用于索引。

`used` 列是布尔型标识。1表示集群应"使用"该段(即，应加载该段并可用于请求), 0表示不应将段主动加载到集群中。我们这样做是为了从集群中删除段，而不实际删除它们的元数据（这允许在出现问题时更简单地回滚）。

`payload` 列存储一个JSON blob，该blob包含该段的所有元数据（存储在该payload中的某些数据与表中的某些列是冗余的，这是有意的），信息如下：

```
{
 "dataSource":"wikipedia",
 "interval":"2012-05-23T00:00:00.000Z/2012-05-24T00:00:00.000Z",
 "version":"2012-05-24T00:10:00.046Z",
 "loadSpec":{
    "type":"s3_zip",
    "bucket":"bucket_for_segment",
    "key":"path/to/segment/on/s3"
 },
 "dimensions":"comma-delimited-list-of-dimension-names",
 "metrics":"comma-delimited-list-of-metric-names",
 "shardSpec":{"type":"none"},
 "binaryVersion":9,
 "size":size_of_segment,
 "identifier":"wikipedia_2012-05-23T00:00:00.000Z_2012-05-24T00:00:00.000Z_201
}
```

请注意，此blob的格式可以而且将不时地更改。

规则表

规则表用于存储有关段应在何处着陆的各种规则。Coordinator 在对集群进行段（重）分配决策时使用这些规则。

配置表

配置表用于存储运行时的配置对象。我们还没有很多这样的机制，我们也不确定是否会继续使用这种机制，但这是一种在运行时跨集群更改一些配置参数的方法的开始。

任务相关的表

在管理任务时，Overlord 和 MiddleManager 还创建和使用了许多表。

审计表

审核表用于存储配置更改的历史记录，例如Coordinator 所做的规则更改和其他配置更改。

只有以下角色才能访问元数据存储：

1. 索引服务进程（如果有）
2. 实时进程（如果有）
3. 协调程序

因此，您只需要为这些计算机授予访问元数据存储的权限（例如，在AWS安全组中）。

ZooKeeper

Apache Druid使用Apache ZooKeeper 来管理整个集群状态。通过ZK来进行的操作有：

1. Coordinator Leader选举
2. Historical 段发布协议
3. Coordinator 和 Historical 之间的段加载/删除
4. Overlord Leader选举
5. Overlord和MiddleManager任务管理

Coordinator Leader选举

我们使用 **Curator LeadershipLatch** 进行Leader选举：

```
${druid.zk.paths.coordinatorPath}/_COORDINATOR
```

Historical和Realtime之间的段发布

`announcementsPath` 和 `servedSegmentsPath` 这两个参数用于这个功能。

所有的 Historical 进程都将它们自身发布到 `announcementsPath`，具体来说它们将在以下路径创建一个临时的ZNODE：

```
${druid.zk.paths.announcementsPath}/${druid.host}
```

这意味着Historical节点可用。它们也将随后创建一个ZNODE:

```
${druid.zk.paths.servedSegmentsPath}/${druid.host}
```

当它们加载段时，它们将在以下路径附着的一个临时的ZNODE：

```
${druid.zk.paths.servedSegmentsPath}/${druid.host}/_segment_identifier_
```

然后，Coordinator 和 Broker 之类的进程可以监视这些路径，以查看哪些进程当前正在为哪些段提供服务。

Coordinator和Historical之间的段加载/删除

`loadQueuePath` 参数用于这个功能。

当 Coordiantor 决定一个 Historical 进程应该加载或删除一个段时，它会将一个临时znode写到:

```
${druid.zk.paths.loadQueuePath}/_host_of_historical_process/_segment_identifie
```

这个znode将包含一个payload，它向Historical进程指示它应该如何处理给定的段。当Historical进程完成任务时，它将删除znode，以便向Coordinator表示它已经完成处理。

数据摄入

综述

Druid中的所有数据都被组织成*段*，这些段是数据文件，通常每个段最多有几百万行。在Druid中加载数据称为*摄取或索引*，它包括从源系统读取数据并基于该数据创建段。

在大多数摄取方法中，加载数据的工作由Druid MiddleManager 进程（或 Indexer 进程）完成。一个例外是基于Hadoop的摄取，这项工作是使用Hadoop MapReduce作业在YARN上完成的（尽管MiddleManager或Indexer进程仍然参与启动和监视Hadoop作业）。一旦段被生成并存储在 深层存储 中，它们将被Historical进程加载。有关如何在引擎下工作的更多细节，请参阅Druid设计文档的存储设计 部分。

如何使用本文档

您**当前正在阅读的这个页面**提供了通用Druid摄取概念的信息，以及 所有摄取方法**通用的配置**信息。

每个摄取方法的单独页面提供了有关每个摄取方法**独有的概念和配置**的附加信息。

我们建议您先阅读（或至少略读）这个通用页面，然后参考您选择的一种或多种摄取方法的页面。

摄入方式

下表列出了Druid最常用的数据摄取方法，帮助您根据自己的情况选择最佳方法。每个摄取方法都支持自己的一组源系统。有关每个方法如何工作的详细信息以及特定于该方法的配置属性，请查看其文档页。

流式摄取

最推荐、也是最流行的流式摄取方法是直接从Kafka读取数据的 Kafka索引服务 。如果你喜欢Kinesis，Kinesis索引服务 也能很好地工作。

下表比较了主要可用选项：

Method	Kafka	Kinesis	Tranquility
Supervisor类型	`kafka`	`kinesis`	`N/A`
如何工作	Druid直接从Apache Kafka读取数据	Druid直接从Amazon Kinesis中读取数据	Tranquility, 一个独立于Druid的库，用来将数据推送到Druid
可以摄入迟到的数据	Yes	Yes	No(迟到的数据将会被基于 `windowPeriod` 的配置丢弃掉)
保证不重不丢（Exactly-once）	Yes	Yes	No

批量摄取

从文件进行批加载时，应使用一次性 任务，并且有三个选项： `index_parallel` （本地并行批任务）、 `index_hadoop` （基于hadoop）或 `index` （本地简单批任务）。

一般来说，如果本地批处理能满足您的需要时我们建议使用它，因为设置更简单（它不依赖于外部Hadoop集群）。但是，仍有一些情况下，基于Hadoop的批摄取可能是更好的选择，例如，当您已经有一个正在运行的Hadoop集群，并且希望使用现有集群的集群资源进行批摄取时。

此表比较了三个可用选项：

方式	本地批任务(并行)	基于Hadoop	本地批任务
任务类型	`index_parallel`	`index_hadoop`	`index`
并行?	如果 `inputFormat` 是可分割的且 `tuningConfig` 中的 `maxNumConcurrentSubTasks` > 1, 则 **Yes**	Yes	No，每个任务程的
支持追加或者覆盖	都支持	只支持覆盖	都支持
外部依赖	无	Hadoop集群，用来提交Map-Reduce任务	无
输入位置	任何 输入数据源	任何Hadoop文件系统或者Druid数据源	任何 输入数据
文件格式	任何 输入格式	任何Hadoop输入格式	任何 输入格式
Rollup modes	如果 `tuningConfig` 中的 `forceGuaranteedRollup` = true, 则为 **Perfect(最佳rollup)**	总是Perfect（最佳rollup）	如果 `tuningC` 的 `forceGuarant` = true, 则为 **P 佳rollup)**
分区选项	可选的有 `Dynamic` ， `hash-based` 和 `range-based` 三种分区方式，详情参见 分区规范	通过 partitionsSpec中指定 `hash-based` 和 `range-based` 分区	可选的有 `Dynamic` 和 `based` 二种分区 详情参见 分区

Druid数据模型

数据源

Druid数据存储在数据源中，与传统RDBMS中的表类似。Druid提供了一个独特的数据建模系统，它与关系模型和时间序列模型都具有相似性。

主时间戳列

Druid Schema必须始终包含一个主时间戳。主时间戳用于对 数据进行分区和排序。Druid查询能够快速识别和检索与主时间戳列的时间范围相对应的数据。Druid还可以将主时间戳列用于基于时间的数据管理操作，例如删除时间块、覆盖时间块和基于时间的保留规则。

主时间戳基于 `timestampSpec` 进行解析。此外， `granularitySpec` 控制基于主时间戳的其他重要操作。无论从哪个输入字段读取主时间戳，它都将作为名为 `__time` 的列存储在Druid数据源中。

如果有多个时间戳列，则可以将其他列存储为 辅助时间戳。

维度

维度是按原样存储的列，可以用于任何目的, 可以在查询时以特殊方式对维度进行分组、筛选或应用聚合器。如果在禁用了 rollup 的情况下运行，那么该维度集将被简单地视为要摄取的一组列，并且其行为与不支持rollup功能的典型数据库的预期完全相同。

通过 `dimensionSpec` 配置维度。

指标

Metrics是以聚合形式存储的列。启用 rollup 时，它们最有用。指定一个Metric允许您为Druid选择一个聚合函数，以便在摄取期间应用于每一行。这有两个好处：

1. 如果启用了 rollup，即使保留摘要信息，也可以将多行折叠为一行。在 Rollup 教程 中，这用于将netflow数据折叠为每（ `minute` ， `srcIP` ， `dstIP` ）元组一行，同时保留有关总数据包和字节计数的聚合信息。
2. 一些聚合器，特别是近似聚合器，即使在非汇总数据上，如果在接收时部分计算，也可以在查询时更快地计算它们。

Metrics是通过 `metricsSpec` 配置的。

Rollup

什么是rollup

Druid可以在接收过程中将数据进行汇总，以最小化需要存储的原始数据量。Rollup是一种汇总或预聚合的形式。实际上，Rollup可以极大地减少需要存储的数据的大小，从而潜在地减少行数的数量级。这种存储量的减少是有代价的：当我们汇总数据时，我们就失去了查询单个事件的能力。

禁用rollup时，Druid将按原样加载每一行，而不进行任何形式的预聚合。此模式类似于您对不支持汇总功能的典型数据库的期望。

如果启用了rollup，那么任何具有相同维度和时间戳的行（在基于 `queryGranularity` 的截断之后）都可以在Druid中折叠或汇总为一行。

rollup默认是启用状态。

启用或者禁用rollup

Rollup由 `granularitySpec` 中的 `rollup` 配置项控制。 默认情况下，值为 `true` (启用状态)。如果你想让Druid按原样存储每条记录，而不需要任何汇总，将该值设置为 `false` 。

rollup示例

有关如何配置Rollup以及该特性将如何修改数据的示例，请参阅Rollup教程。

最大化rollup比率

通过比较Druid中的行数和接收的事件数，可以测量数据源的汇总率。这个数字越高，从汇总中获得的好处就越多。一种方法是使用Druid SQL查询，比如：

```
SELECT SUM("cnt") / COUNT(*) * 1.0 FROM datasource
```

在这个查询中，`cnt` 应该引用在摄取时指定的"count"类型Metrics。有关启用汇总时计数工作方式的详细信息，请参阅"架构设计"页上的 计数接收事件数。

最大化Rollup的提示：

- 一般来说，拥有的维度越少，维度的基数越低，您将获得更好的汇总比率
- 使用 Sketches 避免存储高基数维度，因为会损害汇总比率
- 在摄入时调整 `queryGranularity` （例如，使用 `PT5M` 而不是 `PT1M` ）会增加Druid中两行具有匹配时间戳的可能性，并可以提高汇总率
- 将相同的数据加载到多个Druid数据源中是有益的。有些用户选择创建禁用汇总（或启用汇总，但汇总比率最小）的"完整"数据源和具有较少维度和较高汇总比率的"缩写"数据源。当查询只涉及"缩写"集里边的维度时，使用该数据源将导致更快的查询时间，这种方案只需稍微增加存储空间即可完成，因为简化的数据源往往要小得多。
- 如果您使用的 尽力而为的汇总(best-effort rollup) 摄取配置不能保证完全汇总(perfect rollup)，则可以通过切换到保证的完全汇总选项，或在初始摄取后在后台重新编制(reindex)数据索引，潜在地提高汇总比率。

最佳rollup VS 尽可能rollup

一些Druid摄取方法保证了*完美的汇总(perfect rollup)*，这意味着输入数据在摄取时被完美地聚合。另一些则提供了*尽力而为的汇总(best-effort rollup)*，这意味着输入数据可能无法完全聚合，因此可能有多个段保存具有相同时间戳和维度值的行。

一般来说，提供*尽力而为的汇总(best-effort rollup)*的摄取方法之所以这样做，是因为它们要么是在没有清洗步骤（这是*完美的汇总(perfect rollup)*所必需的）的情况下并行摄取，要么是因为它们在接收到某个时间段的所有数据（我们称之为*增量发布(incremental publishing)*）之前完成并发布段。在这两种情况下，理论上可以汇总的记录可能会以不同的段结束。所有类型的流接收都在此模式下运行。

保证*完美的汇总(perfect rollup)*的摄取方法通过额外的预处理步骤来确定实际数据摄取阶段之前的间隔和分区。此预处理步骤扫描整个输入数据集，这通常会增加摄取所需的时间，但提供完美汇总所需的信息。

下表显示了每个方法如何处理汇总： | **方法** | **如何工作** | | - | - | | 本地批 | 基于配置， `index_parallel` 和 `index` 可以是完美的，也可以是最佳的。 | | Hadoop批 | 总是 perfect | | Kafka索引服务 | 总是 best-effort | | Kinesis索引服务 | 总是 best-effort |

分区

为什么分区

数据源中段的最佳分区和排序会对占用空间和性能产生重大影响 。 Druid数据源总是按时间划分为*时间块*，每个时间块包含一个或多个段。此分区适用于所有摄取方法，并基于摄取规范的 `dataSchema` 中的 `segmentGranularity` 参数。

特定时间块内的段也可以进一步分区，使用的选项根据您选择的摄取类型而不同。一般来说，使用特定维度执行此辅助分区将改善局部性，这意味着具有该维度相同值的行存储在一起，并且可以快速访问。

通常，通过将数据分区到一些常用来做过滤操作的维度（如果存在的话）上，可以获得最佳性能和最小的总体占用空间。而且，这种分区通常会改善压缩性能而且还往往会提高查询性能（用户报告存储容量减少了三倍）。

> [!WARNING] 分区和排序是最好的朋友！如果您确实有一个天然的分区维度，那么您还应该考虑将它放在 `dimensionsSpec` 的 `dimension` 列表中的第一个维度，它告诉Druid按照该列对每个段中的行进行排序。除了单独分区所获得的改进之外，这通常还会进一步改进压缩。 但是，请注意，目前，Druid总是首先按时间戳对一个段内的行进行排序，甚至在 `dimensionsSpec` 中列出的第一个维度之前，这将使得维度排序达不到最大效率。如果需要，可以通过在 `granularitySpec` 中将 `queryGranularity` 设置为等于 `segmentGranularity` 的值来解决此限制，这将把段内的所有时间戳设置为相同的值，并将"真实"时间戳保存为辅助时间戳。这个限制可能在Druid的未来版本中被移除。

如何设置分区

并不是所有的摄入方式都支持显式的分区配置，也不是所有的方法都具有同样的灵活性。在当前的Druid版本中，如果您是通过一个不太灵活的方法（如Kafka）进行初始摄取，那么您可以使用 重新索引的技术(reindex)，在最初摄取数据后对其重新分区。这是一种强大的技术：即使您不断地从流中添加新数据, 也可以使用它来确保任何早于某个阈值的数据都得到最佳分区。

下表显示了每个摄取方法如何处理分区：

方法	如何工作
本地批	通过 `tuningConfig` 中的 `partitionsSpec`
Hadoop批	通过 `tuningConfig` 中的 `partitionsSpec`
Kafka索引服务	Druid中的分区是由Kafka主题的分区方式决定的。您可以在初次摄入后 重新索引的技术(reindex)以重新分区
Kinesis索引服务	Druid中的分区是由Kinesis流的分区方式决定的。您可以在初次摄入后 重新索引的技术(reindex)以重新分区

> [!WARNING]
>
> 注意，当然，划分数据的一种方法是将其加载到分开的数据源中。这是一种完全可行的方法，当数据源的数量不会导致每个数据源的开销过大时，它可以很好地工作。如果使用这种方法，那么可以忽略这一部分，因为这部分描述了如何在单个数据源中设置分区。
>
> 有关将数据拆分为单独数据源的详细信息以及潜在的操作注意事项，请参阅多租户注意事项。

摄入规范

无论使用哪一种摄入方式，数据要么是通过一次性tasks或者通过持续性的"supervisor"(运行并监控一段时间内的一系列任务)来被加载到Druid中。 在任一种情况下，task或者supervisor的定义都在*摄入规范*中定义。

摄入规范包括以下三个主要的部分：

- `dataSchema`，包含了 `数据源名称`，`主时间戳列`，`维度`，`指标` 和 `转换与过滤`
- `ioConfig`，该部分告诉Druid如何去连接数据源系统以及如何去解析数据。 更多详细信息，可以看摄入方法的文档。
- `tuningConfig`，该部分控制着每一种摄入方法的不同的特定调整参数

一个 `index_parallel` 类型任务的示例摄入规范如下：

```
{
  "type": "index_parallel",
  "spec": {
    "dataSchema": {
      "dataSource": "wikipedia",
      "timestampSpec": {
        "column": "timestamp",
        "format": "auto"
      },
      "dimensionsSpec": {
        "dimensions": [
          { "type": "string", "page" },
          { "type": "string", "language" },
          { "type": "long", "name": "userId" }
        ]
      },
      "metricsSpec": [
        { "type": "count", "name": "count" },
        { "type": "doubleSum", "name": "bytes_added_sum", "fieldName": "bytes_
        { "type": "doubleSum", "name": "bytes_deleted_sum", "fieldName": "byte
      ],
      "granularitySpec": {
        "segmentGranularity": "day",
        "queryGranularity": "none",
        "intervals": [
          "2013-08-31/2013-09-01"
        ]
      }
    },
    "ioConfig": {
      "type": "index_parallel",
      "inputSource": {
        "type": "local",
        "baseDir": "examples/indexing/",
        "filter": "wikipedia_data.json"
      },
      "inputFormat": {
        "type": "json",
        "flattenSpec": {
          "useFieldDiscovery": true,
          "fields": [
            { "type": "path", "name": "userId", "expr": "$.user.id" }
          ]
        }
      }
    },
    "tuningConfig": {
      "type": "index_parallel"
    }
  }
}
```

该部分中支持的特定选项依赖于选择的摄入方法。 更多的示例，可以参考每一种摄入方法的文档。

您还可以不用编写一个摄入规范，可视化的加载数据，该功能位于 Druid控制台 的 "Load Data" 视图中。 Druid可视化数据加载器目前支持 Kafka, Kinesis 和 本地批模式。

dataSchema

> [!WARNING]
>
> `dataSchema` 规范在0.17.0版本中做了更改，新的规范支持除*Hadoop摄取方式*外的所有方式。 可以在 过时的 `dataSchema` 规范查看老的规范

`dataSchema` 包含了以下部分：

- 数据源名称，主时间戳列，维度，指标 和 转换与过滤

一个 `dataSchema` 如下：

```
"dataSchema": {
  "dataSource": "wikipedia",
  "timestampSpec": {
    "column": "timestamp",
    "format": "auto"
  },
  "dimensionsSpec": {
    "dimensions": [
      { "type": "string", "page" },
      { "type": "string", "language" },
      { "type": "long", "name": "userId" }
    ]
  },
  "metricsSpec": [
    { "type": "count", "name": "count" },
    { "type": "doubleSum", "name": "bytes_added_sum", "fieldName": "bytes_adde
    { "type": "doubleSum", "name": "bytes_deleted_sum", "fieldName": "bytes_de
  ],
  "granularitySpec": {
    "segmentGranularity": "day",
    "queryGranularity": "none",
    "intervals": [
      "2013-08-31/2013-09-01"
    ]
  }
}
```

`dataSource`

`dataSource` 位于 `dataSchema` -> `dataSource` 中，简单的标识了数据将被写入的数据源的名称，示例如下：

```
"dataSource": "my-first-datasource"
```

`timestampSpec`

`timestampSpec` 位于 `dataSchema` -> `timestampSpec` 中，用来配置 主时间戳, 示例如下：

```
"timestampSpec": {
  "column": "timestamp",
  "format": "auto"
}
```

> [!WARNING] 概念上，输入数据被读取后，Druid会以一个特定的顺序来对数据应用摄入规范： 首先 `flattenSpec` (如果有)，然后 `timestampSpec`，然后 `transformSpec`，最后是 `dimensionsSpec` 和 `metricsSpec`。在编写摄入规范时需要牢记这一点

`timestampSpec` 可以包含以下的部分：

字段	描述	默认值
column	要从中读取主时间戳的输入行字段。 不管这个输入字段的名称是什么，主时间戳总是作为一个名为"__time"的列存储在您的Druid数据源中	timestamp
format	时间戳格式，可选项有： • `iso`：使用"T"分割的ISO8601，像"2000-01-01T01:02:03.456" • `posix`：自纪元以来的秒数 • `millis`：自纪元以来的毫秒数 • `micro`：自纪元以来的微秒数 • `nano`：自纪元以来的纳秒数 • `auto`：自动检测ISO或者毫秒格式 • 任何 Joda DateTimeFormat字符串	auto
missingValue	用于具有空或缺少时间戳列的输入记录的时间戳。应该是ISO8601格式，如 `"2000-01-01T01:02:03.456"` 。由于Druid需要一个主时间戳，因此此设置对于接收根本没有任何时间戳的数据集非常有用。	none

`dimensionSpec`

`dimensionsSpec` 位于 `dataSchema` -> `dimensionsSpec`，用来配置维度。示例如下：

```
"dimensionsSpec" : {
  "dimensions": [
    "page",
    "language",
    { "type": "long", "name": "userId" }
  ],
  "dimensionExclusions" : [],
  "spatialDimensions" : []
}
```

> [!WARNING] 概念上，输入数据被读取后，Druid会以一个特定的顺序来对数据应用摄入规范： 首先 `flattenSpec` (如果有)，然后 `timestampSpec`，然后 `transformSpec`，最后是 `dimensionsSpec` 和 `metricsSpec`。在编写摄入规范时需要牢记这一点

`dimensionsSpec` 可以包括以下部分：

字段	描述	默认值
dimensions	维度名称或者对象的列表，在 `dimensions` 和 `dimensionExclusions` 中不能包含相同的列。 如果该配置为一个空数组，Druid将会把所有未出现在 `dimensionExclusions` 中的非时间、非指标列当做字符串类型的维度列，参见Inclusions and exclusions。	`[]`
dimensionExclusions	在摄取中需要排除的列名称，在该配置中只支持名称，不支持对象。在 `dimensions` 和 `dimensionExclusions` 中不能包含相同的列。	`[]`
spatialDimensions	一个空间维度的数组	`[]`

`Dimension objects`

在 `dimensions` 列的每一个维度可以是一个名称，也可以是一个对象。 提供一个名称等价于提供了一个给定名称的 `string` 类型的维度对象。例如： `page` 等价于 `{"name": "page", "type": "string"}` 。

维度对象可以有以下的部分：

字段	描述	默认值
type	`string` , `long` , `float` 或者 `double`	`string`
name	维度名称，将用作从输入记录中读取的字段名，以及存储在生成的段中的列名。 注意： 如果想在摄取的时候重新命名列，可以使用 `transformSpec`	none（必填）
createBitmapIndex	对于字符串类型的维度，是否应为生成的段中的列创建位图索引。创建位图索引需要更多存储空间，但会加快某些类型的筛选（特别是相等和前缀筛选）。仅支持字符串类型的维度。	`true`

`Inclusions and exclusions`

Druid以两种可能的方式来解释 `dimensionsSpec` ：*normal* 和 *schemaless*

当 `dimensions` 或者 `spatialDimensions` 为非空时， 将会采用正常的解释方式。在该情况下， 前边说的两个列表结合起来的集合当做摄入的维度集合。

当 `dimensions` 和 `spatialDimensions` 同时为空或者null时候，将会采用无模式的解释方式。 在该情况下，维度集合由以下方式决定：

1. 首先，从 `inputFormat` (或者 `flattenSpec` , 如果正在使用)中所有输入字段集合开始

2. 排除掉任何在 `dimensionExclusions` 中的列
3. 排除掉在 `timestampSpec` 中的时间列
4. 排除掉 `metricsSpec` 中用于聚合器输入的列
5. 排除掉 `metricsSpec` 中任何与聚合器同名的列
6. 所有的其他字段都被按照默认配置摄入为 `string` 类型的维度

> [!WARNING] 注意：在无模式的维度解释方式中，由 `transformSpec` 生成的列当前并未考虑。

`metricsSpec`

`metricsSpec` 位于 `dataSchema` -> `metricsSpec` 中，是一个在摄入阶段要应用的聚合器 列表。 在启用了 rollup 时是很有用的，因为它将配置如何在摄入阶段进行聚合。

一个 `metricsSpec` 实例如下：

```
"metricsSpec": [
  { "type": "count", "name": "count" },
  { "type": "doubleSum", "name": "bytes_added_sum", "fieldName": "bytes_added"
  { "type": "doubleSum", "name": "bytes_deleted_sum", "fieldName": "bytes_dele
]
```

> [!WARNING] 通常，当 rollup 被禁用时，应该有一个空的 `metricsSpec` （因为没有rollup，Druid不会在摄取时进行任何的聚合，所以没有理由包含摄取时聚合器）。但是，在某些情况下，定义Metrics仍然是有意义的：例如，如果要创建一个复杂的列作为 近似聚合 的预计算部分，则只能通过在 `metricsSpec` 中定义度量来实现

`granularitySpec`

`granularitySpec` 位于 `dataSchema` -> `granularitySpec`，用来配置以下操作：

1. 通过 `segmentGranularity` 来将数据源分区到 时间块
2. 如果需要的话，通过 `queryGranularity` 来截断时间戳
3. 通过 `interval` 来指定批摄取中应创建段的时间块
4. 通过 `rollup` 来指定是否在摄取时进行汇总

除了 `rollup`，这些操作都是基于 主时间戳列

一个 `granularitySpec` 实例如下：

```
"granularitySpec": {
  "segmentGranularity": "day",
  "queryGranularity": "none",
  "intervals": [
    "2013-08-31/2013-09-01"
  ],
  "rollup": true
}
```

`granularitySpec` 可以有以下的部分：

字段	描述	默认值
type	`uniform` 或者 `arbitrary` ，大多数时候使用 `uniform`	`uniform`
segmentGranularity	数据源的 时间分块 粒度。每个时间块可以创建多个段, 例如，当设置为 `day` 时，同一天的事件属于同一时间块，该时间块可以根据其他配置和输入大小进一步划分为多个段。这里可以提供任何粒度。请注意，同一时间块中的所有段应具有相同的段粒度。 如果 `type` 字段设置为 `arbitrary` 则忽略	`day`
queryGranularity	每个段内时间戳存储的分辨率, 必须等于或比 `segmentGranularity` 更细。这将是您可以查询的最细粒度，并且仍然可以查询到合理的结果。但是请注意，您仍然可以在比此粒度更粗的场景进行查询，例如 " `minute` "的值意味着记录将以分钟的粒度存储，并且可以在分钟的任意倍数（包括分钟、5分钟、小时等）进行查询。 这里可以提供任何 粒度 。使用 `none` 按原样存储时间戳，而不进行任何截断。请注意，即使将 `queryGranularity` 设置为 `none` ，也将应用 `rollup` 。	`none`
rollup	是否在摄取时使用 rollup。 注意：即使 `queryGranularity` 设置为 `none` ，rollup也仍然是有效的，当数据具有相同的时间戳时数据将被汇总	`true`

字段	描述	默认值
interval	描述应该创建段的时间块的间隔列表。如果 `type` 设置为 `uniform`，则此列表将根据 `segmentGranularity` 进行拆分和舍入。如果 `type` 设置为 `arbitrary`，则将按原样使用此列表。 如果该值不提供或者为空值，则批处理摄取任务通常会根据在输入数据中找到的时间戳来确定要输出的时间块。 如果指定，批处理摄取任务可以跳过确定分区阶段，这可能会导致更快的摄取。批量摄取任务也可以预先请求它们的所有锁，而不是逐个请求。批处理摄取任务将丢弃任何时间戳超出指定间隔的记录。 在任何形式的流摄取中忽略该配置。	`null`

`transformSpec`

`transformSpec` 位于 `dataSchema` -> `transformSpec`，用来摄取时转换和过滤输入数据。 一个 `transformSpec` 实例如下：

```
"transformSpec": {
  "transforms": [
    { "type": "expression", "name": "countryUpper", "expression": "upper(count
  ],
  "filter": {
    "type": "selector",
    "dimension": "country",
    "value": "San Serriffe"
  }
}
```

> [!WARNING] 概念上，输入数据被读取后，Druid会以一个特定的顺序来对数据应用摄入规范： 首先 `flattenSpec` (如果有)，然后 `timestampSpec`，然后 `transformSpec`,最后是 `dimensionsSpec` 和 `metricsSpec`。在编写摄入规范时需要牢记这一点

过时的 `dataSchema` 规范

> [!WARNING]
>
> `dataSchema` 规范在0.17.0版本中做了更改，新的规范支持除*Hadoop摄取方式*外的所有方式。 可以在 `dataSchema` 查看老的规范

除了上面 `dataSchema` 一节中列出的组件之外，过时的 `dataSchema` 规范还有以下两个组件。

- input row parser, flatten of nested data

parser(已废弃) 在过时的 `dataSchema` 中， `parser` 位于 `dataSchema` -> `parser` 中，负责配置与解析输入记录相关的各种项。由于 `parser` 已经废弃，不推荐使用，强烈建议改用 `inputFormat` 。 对于 `inputFormat` 和支持的 `parser` 类型，可以参见 数据格式。

`parseSpec` 主要部分的详细，参见他们的子部分：

- `timestampSpec` , 配置 主时间戳列
- `dimensionsSpec` , 配置 维度
- `flattenSpec`

一个 `parser` 实例如下：

```
"parser": {
  "type": "string",
  "parseSpec": {
    "format": "json",
    "flattenSpec": {
      "useFieldDiscovery": true,
      "fields": [
        { "type": "path", "name": "userId", "expr": "$.user.id" }
      ]
    },
    "timestampSpec": {
      "column": "timestamp",
      "format": "auto"
    },
    "dimensionsSpec": {
      "dimensions": [
        { "type": "string", "page" },
        { "type": "string", "language" },
        { "type": "long", "name": "userId" }
      ]
    }
  }
}
```

flattenSpec 在过时的 `dataSchema` 中， `flattenSpec` 位于 `dataSchema` -> `parser` -> `parseSpec` -> `flattenSpec` 中，负责在潜在的嵌套输入数据（如JSON、Avro等）和Druid的数据模型之间架起桥梁。有关详细信息，请参见 flattenSpec 。

`ioConfig`

`ioConfig` 影响从源系统（如Apache Kafka、Amazon S3、挂载的文件系统或任何其他受支持的源系统）读取数据的方式。 `inputFormat` 属性适用于除Hadoop摄取之外的所有摄取方法。Hadoop摄取仍然使用过时的 `dataSchema` 中的[parser]。 `ioConfig` 的其余部分特定于每个单独的摄取方法。读取JSON数据的 `ioConfig` 示例如下：

```
"ioConfig": {
    "type": "<ingestion-method-specific type code>",
    "inputFormat": {
      "type": "json"
    },
    ...
}
```

详情可以参见每个 摄取方式 提供的文档。

`tuningConfig`

优化属性在 `tuningConfig` 中指定， `tuningConfig` 位于摄取规范的顶层。有些属性适用于所有摄取方法，但大多数属性特定于每个单独的摄取方法。 `tuningConfig` 将所有共享的公共属性设置为默认值的示例如下：

```
"tuningConfig": {
  "type": "<ingestion-method-specific type code>",
  "maxRowsInMemory": 1000000,
  "maxBytesInMemory": <one-sixth of JVM memory>,
  "indexSpec": {
    "bitmap": { "type": "concise" },
    "dimensionCompression": "lz4",
    "metricCompression": "lz4",
    "longEncoding": "longs"
  },
  <other ingestion-method-specific properties>
}
```

字段	描述	默认值
type	每一种摄入方式都有自己的类型，必须指定为与摄入方式匹配的类型。通常的选项有 `index`，`hadoop`，`kafka` 和 `kinesis`	
maxRowsInMemory	数据持久化到硬盘前在内存中存储的最大数据条数。 注意，这个数字是汇总后的，所以可能并不等于输入的记录数。 当摄入的数据达到 `maxRowsInMemory` 或者 `maxBytesInMemory` 时数据将被持久化到硬盘。	`1000000`
maxBytesInMemory	在持久化之前要存储在JVM堆中的数据最大字节数。这是基于对内存使用的粗略估计。当达到 `maxRowsInMemory` 或 `maxBytesInMemory` 时（以先发生的为准），摄取的记录将被持久化到磁盘。 将 `maxBytesInMemory` 设置为-1将禁用此检查，这意味着Druid将完全依赖 `maxRowsInMemory` 来控制内存使用。将其设置为零意味着将使用默认值（JVM堆大小的六分之一）。 请注意，内存使用量的估计值被设计为高估值，并且在使用复杂的摄取时聚合器（包括sketches）时可能特别高。如果这导致索引工作负载过于频繁地持久化到磁盘，则可以将 `maxBytesInMemory` 设置为-1并转而依赖 `maxRowsInMemory`。	JVM堆内存最大值的1/6
indexSpec	优化数据如何被索引，详情可以看下面的表格	看下面的表格
其他属性	每一种摄入方式都有其自己的优化属性。 详情可以查看每一种方法的文档。 Kafka索引服务, Kinesis索引服务, 本地批 和 Hadoop批	

`indexSpec`

上边表格中的 `indexSpec` 部分可以包含以下属性：

字段	描述	默认
bitmap	位图索引的压缩格式。 需要一个 `type` 设置为 `concise` 或者 `roaring` 的JSON对象。对于 `roaring` 类型，布尔属性 `compressRunOnSerialization` （默认为true）控制在确定运行长度编码更节省空间时是否使用该编码。	`{"type":"`
dimensionCompression	维度列的压缩格式。 可选项有 `lz4` , `lzf` 或者 `uncompressed`	`lz4`
metricCompression	Metrics列的压缩格式。可选项有 `lz4` , `lzf` , `uncompressed` 或者 `none` (`none` 比 `uncompressed` 更有效，但是在老版本的Druid不支持)	`lz4`
longEncoding	long类型列的编码格式。无论它们是维度还是Metrics，都适用，选项是 `auto` 或 `long` 。 `auto` 根据列基数使用偏移量或查找表对值进行编码，并以可变大小存储它们。 `longs` 按原样存储值，每个值8字节。	`longs`

除了这些属性之外，每个摄取方法都有自己的特定调整属性。有关详细信息，请参阅每个 摄取方法 的文档。

数据格式

Apache Druid可以接收JSON、CSV或TSV等分隔格式或任何自定义格式的非规范化数据。尽管文档中的大多数示例使用JSON格式的数据，但将Druid配置为接收任何其他分隔数据并不困难。我们欢迎对新格式的任何贡献。

此页列出了Druid支持的所有默认和核心扩展数据格式。有关社区扩展支持的其他数据格式，请参阅我们的 社区扩展列表。

格式化数据

下面的示例显示了在Druid中原生支持的数据格式：

JSON

```
{"timestamp": "2013-08-31T01:02:33Z", "page": "Gypsy Danger", "language" : "en
{"timestamp": "2013-08-31T03:32:45Z", "page": "Striker Eureka", "language" : "
{"timestamp": "2013-08-31T07:11:21Z", "page": "Cherno Alpha", "language" : "ru
{"timestamp": "2013-08-31T11:58:39Z", "page": "Crimson Typhoon", "language" :
{"timestamp": "2013-08-31T12:41:27Z", "page": "Coyote Tango", "language" : "ja
```

CSV

```
2013-08-31T01:02:33Z,"Gypsy Danger","en","nuclear","true","true","false","fals
2013-08-31T03:32:45Z,"Striker Eureka","en","speed","false","true","true","fals
2013-08-31T07:11:21Z,"Cherno Alpha","ru","masterYi","false","true","true","fal
2013-08-31T11:58:39Z,"Crimson Typhoon","zh","triplets","true","false","true","
2013-08-31T12:41:27Z,"Coyote Tango","ja","cancer","true","false","true","false
```

TSV (Delimited)

```
2013-08-31T01:02:33Z  "Gypsy Danger"  "en"  "nuclear" "true"  "true"  "false"
2013-08-31T03:32:45Z  "Striker Eureka"  "en"  "speed" "false" "true"  "true"
2013-08-31T07:11:21Z  "Cherno Alpha"  "ru"  "masterYi"  "false" "true"  "true"
2013-08-31T11:58:39Z  "Crimson Typhoon" "zh"  "triplets"  "true"  "false" "tru
2013-08-31T12:41:27Z  "Coyote Tango"  "ja"  "cancer"  "true"  "false" "true"
```

请注意，CSV和TSV数据不包含列标题。当您指定要摄取的数据时，这一点就变得很重要。

除了文本格式，Druid还支持二进制格式，比如 Orc 和 Parquet 格式。

定制格式

Druid支持自定义数据格式，可以使用 `Regex` 解析器或 `JavaScript` 解析器来解析这些格式。请注意，使用这些解析器中的任何一个来解析数据都不如编写原生Java解析器或使用外部流处理器那样高效。我们欢迎新解析器的贡献。

InputFormat

> [!WARNING] 输入格式是在0.17.0中引入的指定输入数据的数据格式的新方法。不幸的是，输入格式还不支持Druid支持的所有数据格式或摄取方法。特别是如果您想使用Hadoop接收，您仍然需要使用 解析器。如果您的数据是以本节未列出的某种格式格式化的，请考虑改用解析器。

所有形式的Druid摄取都需要某种形式的schema对象。要摄取的数据的格式是使用 `ioConfig` 中的 `inputFormat` 条目指定的。

JSON

JSON 一个加载JSON格式数据的 `inputFormat` 示例：

```
"ioConfig": {
  "inputFormat": {
    "type": "json"
  },
  ...
}
```

JSON `inputFormat` 有以下组件：

字段	类型	描述	是否必填
type	String	填 `json`	是
flattenSpec	JSON对象	指定嵌套JSON数据的展平配置。更多信息请参见flattenSpec	否
featureSpec	JSON对象	Jackson库支持的 JSON解析器特性 。这些特性将在解析输入JSON数据时应用。	否

CSV

一个加载CSV格式数据的 `inputFormat` 示例：

```
"ioConfig": {
  "inputFormat": {
    "type": "csv",
    "columns" : ["timestamp","page","language","user","unpatrolled","newPage",
  },
  ...
}
```

CSV `inputFormat` 有以下组件：

字段	类型	描述	是
type	String	填 `csv`	是
listDelimiter	String	多值维度的定制分隔符	否(默认ct
columns	JSON数组	指定数据的列。列的顺序应该与数据列的顺序相同。	如果 `findColu` 设置为 `f` 失， 则为
findColumnsFromHeader	布尔	如果设置了此选项，则任务将从标题行中查找列名。请注意，在从标题中查找列名之前，将首先使用 `skipHeaderRows` 。例如，如果将 `skipHeaderRows` 设置为2，将 `findColumnsFromHeader` 设置为 `true` ，则任务将跳过前两行，然后从第三行提取列信息。该项如果设置为true，则将忽略 `columns`	否（如果设置则默否则为nu
skipHeaderRows	整型数值	该项如果设置，任务将略过 `skipHeaderRows` 配置的行数	否（默认

TSV(Delimited)

```
"ioConfig": {
  "inputFormat": {
    "type": "tsv",
    "columns" : ["timestamp","page","language","user","unpatrolled","newPage",
    "delimiter":"| "
  },
  ...
}
```

TSV `inputFormat` 有以下组件：

字段	类型	描述	是
type	String	填 `tsv`	是
delimiter	String	数据值的自定义分隔符	否(默认为
listDelimiter	String	多值维度的定制分隔符	否(默认ct
columns	JSON数组	指定数据的列。列的顺序应该与数据列的顺序相同。	如果 `findColu` 设置为 `f` 失， 则为
findColumnsFromHeader	布尔	如果设置了此选项，则任务将从标题行中查找列名。请注意，在从标题中查找列名之前，将首先使用 `skipHeaderRows` 。例如，如果将 `skipHeaderRows` 设置为2，将 `findColumnsFromHeader` 设置为 `true` ，则任务将跳过前两行，然后从第三行提取列信息。该项如果设置为true，则将忽略 `columns`	否（如果设置则默否则为nu
skipHeaderRows	整型数值	该项如果设置，任务将略过 `skipHeaderRows` 配置的行数	否（默认

请确保将分隔符更改为适合于数据的分隔符。与CSV一样，您必须指定要索引的列和列的子集。

ORC

> [!WARNING] 使用ORC输入格式之前，首先需要包含 druid-orc-extensions
>
> [!WARNING] 如果您正在考虑从早于0.15.0的版本升级到0.15.0或更高版本，请仔细阅读 从contrib扩展的迁移。

一个加载ORC格式数据的 `inputFormat` 示例：

```
"ioConfig": {
  "inputFormat": {
    "type": "orc",
    "flattenSpec": {
      "useFieldDiscovery": true,
      "fields": [
        {
          "type": "path",
          "name": "nested",
          "expr": "$.path.to.nested"
        }
      ]
    }
    "binaryAsString": false
  },
  ...
}
```

ORC `inputFormat` 有以下组件：

字段	类型	描述	是否必填
type	String	填 `orc`	是
flattenSpec	JSON对象	指定嵌套JSON数据的展平配置。更多信息请参见flattenSpec	否
binaryAsString	布尔类型	指定逻辑上未标记为字符串的二进制orc列是否应被视为UTF-8编码字符串。	否（默认为false）

Parquet

> [!WARNING] 使用Parquet输入格式之前，首先需要包含 druid-parquet-extensions

一个加载Parquet格式数据的 `inputFormat` 示例：

```
"ioConfig": {
  "inputFormat": {
    "type": "parquet",
    "flattenSpec": {
      "useFieldDiscovery": true,
      "fields": [
        {
          "type": "path",
          "name": "nested",
          "expr": "$.path.to.nested"
        }
      ]
    }
    "binaryAsString": false
  },
  ...
}
```

Parquet `inputFormat` 有以下组件：

字段	类型	描述	是否必填
type	String	填 `parquet`	是
flattenSpec	JSON对象	定义一个 flattenSpec 从Parquet文件提取嵌套的值。注意，只支持"path"表达式（'jq'不可用）	否（默认自动发现根级别的属性）
binaryAsString	布尔类型	指定逻辑上未标记为字符串的二进制orc列是否应被视为UTF-8编码字符串。	否（默认为false）

FlattenSpec

`flattenSpec` 位于 `inputFormat` -> `flattenSpec` 中，负责将潜在的嵌套输入数据（如JSON、Avro等）和Druid的平面数据模型之间架起桥梁。 `flattenSpec` 示例如下：

```
"flattenSpec": {
  "useFieldDiscovery": true,
  "fields": [
    { "name": "baz", "type": "root" },
    { "name": "foo_bar", "type": "path", "expr": "$.foo.bar" },
    { "name": "first_food", "type": "jq", "expr": ".thing.food[1]" }
  ]
}
```

> [!WARNING] 概念上，输入数据被读取后，Druid会以一个特定的顺序来对数据应用摄入规范： 首先 `flattenSpec` (如果有)，然后 `timestampSpec` , 然后 `transformSpec` ,最后是 `dimensionsSpec` 和 `metricsSpec` 。在编写摄入规范时需要牢记这一点

展平操作仅仅支持嵌套的 数据格式, 包括： `avro` , `json` , `orc` 和 `parquet` 。

`flattenSpec` 有以下组件：

字段	描述
useFieldDiscovery	如果为true，则将所有根级字段解释为可用字段，供 `timestampSpec` 、 `transformSpec` 、 `dimensionsSpec` 和 `metricsSpec` 使用。 如果为false，则只有显式指定的字段（请参阅 `fields` ）才可供使用。
fields	指定感兴趣的字段及其访问方式, 详细请见下边

字段展平规范

`fields` 列表中的每个条目都可以包含以下组件：

字段	描述	默认值
type	可选项如下： • `root`，引用记录根级别的字段。只有当 `useFieldDiscovery` 为false时才真正有用。 • `path`，引用使用 JsonPath 表示法的字段，支持大多数提供嵌套的数据格式，包括 `avro`，`csv`，`json` 和 `parquet` • `jq`，引用使用 jackson-jq 表示法的字段， 仅仅支持 `json` 格式	none(必填)
name	展平后的字段名称。这个名称可以被 `timestampSpec`， `transformSpec`， `dimensionsSpec` 和 `metricsSpec` 引用	none(必填)
expr	用于在展平时访问字段的表达式。对于类型 `path`，这应该是 JsonPath。对于 `jq` 类型，这应该是 jackson-jq 表达式。对于其他类型，将忽略此参数。	none(对于 `path` 和 `jq` 类型的为必填)

展平操作的注意事项

- 为了方便起见，在定义根级字段时，可以只将字段名定义为字符串，而不是JSON对象。例如 `{"name": "baz", "type": "root"}` 等价于 `baz`
- 启用 `useFieldDiscovery` 只会在根级别自动检测与Druid支持的数据类型相对应的"简单"字段, 这包括字符串、数字和字符串或数字列表。不会自动检测到其他类型，其他类型必须在 `fields` 列表中显式指定
- 不允许重复字段名（`name`），否则将引发异常
- 如果启用 `useFieldDiscovery`，则将跳过与字段列表中已定义的字段同名的任何已发现字段，而不是添加两次
- http://jsonpath.herokuapp.com/ 对于测试 `path`-类型表达式非常有用
- jackson jq支持完整 `jq` 语法的一个子集。有关详细信息，请参阅 jackson jq 文档

Parser

> [!WARNING] parser在 本地批任务, Kafka索引任务 和 Kinesis索引任务 中已经废弃，在这些类型的摄入方式中考虑使用 inputFormat

该部分列出来了所有默认的以及核心扩展中的解析器。对于社区的扩展解析器，请参见 社区扩展列表

String Parser

`string` 类型的解析器对基于文本的输入进行操作，这些输入可以通过换行符拆分为单独的记录, 可以使用 `parseSpec` 进一步分析每一行。

字段	类型	描述	是否必须
type	string	一般是 `string`，在Hadoop索引任务中为 `hadoopyString`	是
parseSpec	JSON对象	指定格式，数据的timestamp和dimensions	是

Avro Hadoop Parser

> [!WARNING] 需要添加 druid-avro-extensions 来使用 Avro Hadoop解析器

该解析器用于 Hadoop批摄取。在 `ioConfig` 中， `inputSpec` 中的 `inputFormat` 必须设置为 `org.apache.druid.data.input.avro.AvroValueInputFormat` 。您可能想在 `tuningConfig` 中的 `jobProperties` 选项设置Avro reader的schema， 例如： `"avro.schema.input.value.path": "/path/to/your/schema.avsc"` 或者 `"avro.schema.input.value": "your_schema_JSON_object"` 。如果未设置Avro读取器的schema，则将使用Avro对象容器文件中的schema，详情可以参见 avro规范

字段	类型	描述	是否必填
type	String	应该填 `avro_hadoop`	是
parseSpec	JSON对象	指定数据的时间戳和维度。应该是“avro”语法规范。	是

Avro parseSpec可以包含使用"root"或"path"字段类型的 flattenSpec，这些字段类型可用于读取嵌套的Avro记录。Avro当前不支持“jq”字段类型。

例如，使用带有自定义读取器schema文件的Avro Hadoop解析器：

```
{
  "type" : "index_hadoop",
  "spec" : {
    "dataSchema" : {
      "dataSource" : "",
      "parser" : {
        "type" : "avro_hadoop",
        "parseSpec" : {
          "format": "avro",
          "timestampSpec": <standard timestampSpec>,
          "dimensionsSpec": <standard dimensionsSpec>,
          "flattenSpec": <optional>
        }
      }
    },
    "ioConfig" : {
      "type" : "hadoop",
      "inputSpec" : {
        "type" : "static",
        "inputFormat": "org.apache.druid.data.input.avro.AvroValueInputFormat"
        "paths" : ""
      }
    },
    "tuningConfig" : {
       "jobProperties" : {
          "avro.schema.input.value.path" : "/path/to/my/schema.avsc"
      }
    }
  }
}
```

ORC Hadoop Parser

> [!WARNING] 需要添加 druid-orc-extensions 来使用ORC Hadoop解析器
>
> [!WARNING] 如果您正在考虑从早于0.15.0的版本升级到0.15.0或更高版本，请仔细阅读 从contrib扩展的迁移。

该解析器用于 Hadoop批摄取。在 `ioConfig` 中， `inputSpec` 中的 `inputFormat` 必须设置为 `org.apache.orc.mapreduce.OrcInputFormat` 。

字段	类型	描述	是否必填
type	String	应该填 `orc`	是
parseSpec	JSON对象	指定数据(`timeAndDim` 和 `orc` 格式)的时间戳和维度和一个 `flattenSpec` （ `orc` 格式）	是

解析器支持两种 `parseSpec` 格式： `orc` 和 `timeAndDims`

`orc` 支持字段的自动发现和展平（如果指定了 flattenSpec。如果未指定展平规范，则默认情况下将启用 `useFieldDiscovery` 。如果启用了 `useFieldDiscovery` ，则指定 `dimensionSpec` 是可选的：如果提供了 `dimensionSpec` ，则它定义的维度列表将是摄取维度的集合，如果缺少发现的字段将构成该列表。

`timeAndDims` 解析规范必须通过 `dimensionSpec` 指定哪些字段将提取为维度。

支持所有 列类型 ，但 `union` 类型除外。 `list` 类型的列（如果用基本类型填充）可以用作多值维度，或者可以使用 flattenSpec 表达式提取特定元素。同样，可以用同样的方式从 `map` 和 `struct` 类型中提取基本字段。自动字段发现将自动为每个（非时间戳）基本类型或基本类型 `list` 以及 `flattenSpec` 中定义的任何展平表达式创建字符串维度。

Hadoop job属性

像大多数Hadoop作业，最佳结果是在 `tuningConfig` 中的 `jobProperties` 中添加 `"mapreduce.job.user.classpath.first": "true"` 或者 `"mapreduce.job.classloader": "true"` 。 注意，如果使用了 `"mapreduce.job.classloader": "true"` ，需要设置 `mapreduce.job.classloader.system.classes` 包含 `-org.apache.hadoop.hive.` 来让Hadoop从应用jars包中加载 `org.apache.hadoop.hive` 而非从系统jar中，例如：

```
...
    "mapreduce.job.classloader": "true",
    "mapreduce.job.classloader.system.classes" : "java., javax.accessibility.,
...
```

这是因为 `orc-mapreduce` 库的配置单元 `hive-storage-api` 依赖关系，它在 `org.apache.hadoop.hive` 包下提供了一些类。如果改为使用 `"mapreduce.job.user.classpath.first": "true"` 设置，则不会出现此问题。

示例

`orc` parser, `orc` parseSpec, 自动字段发现, 展平表达式

```
{
  "type": "index_hadoop",
  "spec": {
    "ioConfig": {
      "type": "hadoop",
      "inputSpec": {
        "type": "static",
        "inputFormat": "org.apache.orc.mapreduce.OrcInputFormat",
        "paths": "path/to/file.orc"
      },
      ...
    },
    "dataSchema": {
      "dataSource": "example",
      "parser": {
        "type": "orc",
        "parseSpec": {
          "format": "orc",
          "flattenSpec": {
            "useFieldDiscovery": true,
            "fields": [
              {
                "type": "path",
                "name": "nestedDim",
                "expr": "$.nestedData.dim1"
              },
              {
                "type": "path",
                "name": "listDimFirstItem",
                "expr": "$.listDim[1]"
              }
            ]
          },
          "timestampSpec": {
            "column": "timestamp",
            "format": "millis"
          }
        }
      },
      ...
    },
    "tuningConfig": <hadoop-tuning-config>
    }
  }
}
```

`orc` parser, `orc` parseSpec, 不具有 `flattenSpec` 或者 `dimensionSpec` 的字段发现

```
{
  "type": "index_hadoop",
  "spec": {
    "ioConfig": {
      "type": "hadoop",
      "inputSpec": {
        "type": "static",
        "inputFormat": "org.apache.orc.mapreduce.OrcInputFormat",
        "paths": "path/to/file.orc"
      },
      ...
    },
    "dataSchema": {
      "dataSource": "example",
      "parser": {
        "type": "orc",
        "parseSpec": {
          "format": "orc",
          "timestampSpec": {
            "column": "timestamp",
            "format": "millis"
          }
        }
      },
      ...
    },
    "tuningConfig": <hadoop-tuning-config>
    }
  }
}
```

`orc` parser, `orc` parseSpec, 非自动发现

```
{
  "type": "index_hadoop",
  "spec": {
    "ioConfig": {
      "type": "hadoop",
      "inputSpec": {
        "type": "static",
        "inputFormat": "org.apache.orc.mapreduce.OrcInputFormat",
        "paths": "path/to/file.orc"
      },
      ...
    },
    "dataSchema": {
      "dataSource": "example",
      "parser": {
        "type": "orc",
        "parseSpec": {
          "format": "orc",
          "flattenSpec": {
            "useFieldDiscovery": false,
            "fields": [
              {
                "type": "path",
                "name": "nestedDim",
                "expr": "$.nestedData.dim1"
              },
              {
                "type": "path",
                "name": "listDimFirstItem",
                "expr": "$.listDim[1]"
              }
            ]
          },
          "timestampSpec": {
            "column": "timestamp",
            "format": "millis"
          },
          "dimensionsSpec": {
            "dimensions": [
              "dim1",
              "dim3",
              "nestedDim",
              "listDimFirstItem"
            ],
            "dimensionExclusions": [],
            "spatialDimensions": []
          }
        }
      },
      ...
    },
    "tuningConfig": <hadoop-tuning-config>
    }
  }
}
```

`orc` parser, `timeAndDims` parseSpec

```
{
  "type": "index_hadoop",
  "spec": {
    "ioConfig": {
      "type": "hadoop",
      "inputSpec": {
        "type": "static",
        "inputFormat": "org.apache.orc.mapreduce.OrcInputFormat",
        "paths": "path/to/file.orc"
      },
      ...
    },
    "dataSchema": {
      "dataSource": "example",
      "parser": {
        "type": "orc",
        "parseSpec": {
          "format": "timeAndDims",
          "timestampSpec": {
            "column": "timestamp",
            "format": "auto"
          },
          "dimensionsSpec": {
            "dimensions": [
              "dim1",
              "dim2",
              "dim3",
              "listDim"
            ],
            "dimensionExclusions": [],
            "spatialDimensions": []
          }
        }
      },
      ...
    },
    "tuningConfig": <hadoop-tuning-config>
  }
}
```

Parquet Hadoop Parser

> [!WARNING] 需要添加 druid-parquet-extensions 来使用Parquet Hadoop解析器

该解析器用于 Hadoop批摄取。在 `ioConfig` 中， `inputSpec` 中的 `inputFormat` 必须设置为 `org.apache.druid.data.input.parquet.DruidParquetInputFormat` 。

Parquet Hadoop 解析器支持自动字段发现，如果提供了一个带有 `parquet` `parquetSpec` 的 `flattenSpec` 也支持展平。 Parquet嵌套 list 和 map 逻辑类型 应与所有受支持类型的JSON path表达式一起正确操作。

字段	类型	描述	是否必填
type	String	应该填 `parquet`	是
parseSpec	JSON对象	指定数据的时间戳和维度和一个可选的 `flattenSpec` 。有效的 `parseSpec` 格式是 `timeAndDims` 和 `parquet`	是
binaryAsString	布尔类型	指定逻辑上未标记为字符串的二进制orc列是否应被视为UTF-8编码字符串。	否（默认为false）

当时间维度是一个 date类型的列, 则无需指定一个格式。 当格式为UTF8的String，则要么指定为 `auto` ，或者显式的指定一个 时间格式。

Parquet Hadoop解析器 vs Parquet Avro Hadoop解析器 两者都是从Parquet文件中读取，但是又轻微的不同。主要不同之处是：

- Parquet Hadoop解析器使用简单的转换，而Parquet Avro Hadoop解析器首先使用 `parquet-avro` 库将Parquet数据转换为Avro记录，然后使用 `druid-avro-extensions` 模块将Avro数据解析为druid
- Parquet Hadoop解析器将Hadoop作业属性 `parquet.avro.add-list-element-records` 设置为false（通常默认为true），以便将原始列表元素"展开"为多值维度
- Parquet Hadoop解析器支持 `int96` Parquet值，而 Parquet Avro Hadoop解析器不支持。 `flatteSpec` 的JSON path表达式求值的行为也可能存在一些细微的差异

基于这些差异，我们建议在Parquet avro hadoop解析器上使用Parquet Hadoop解析器，以允许摄取超出Avro转换模式约束的数据。然而，Parquet Avro Hadoop解析器是支持Parquet格式的原始基础，因此它更加成熟。

示例

`parquet` parser, `parquet` parseSpec

```
{
  "type": "index_hadoop",
  "spec": {
    "ioConfig": {
      "type": "hadoop",
      "inputSpec": {
        "type": "static",
        "inputFormat": "org.apache.druid.data.input.parquet.DruidParquetInputF
        "paths": "path/to/file.parquet"
      },
      ...
    },
    "dataSchema": {
      "dataSource": "example",
      "parser": {
        "type": "parquet",
        "parseSpec": {
          "format": "parquet",
          "flattenSpec": {
            "useFieldDiscovery": true,
            "fields": [
              {
                "type": "path",
                "name": "nestedDim",
                "expr": "$.nestedData.dim1"
              },
              {
                "type": "path",
                "name": "listDimFirstItem",
                "expr": "$.listDim[1]"
              }
            ]
          },
          "timestampSpec": {
            "column": "timestamp",
            "format": "auto"
          },
          "dimensionsSpec": {
            "dimensions": [],
            "dimensionExclusions": [],
            "spatialDimensions": []
          }
        }
      },
      ...
    },
    "tuningConfig": <hadoop-tuning-config>
    }
  }
}
```

`parquet` parser, `timeAndDims` parseSpec

```
{
  "type": "index_hadoop",
  "spec": {
    "ioConfig": {
      "type": "hadoop",
      "inputSpec": {
        "type": "static",
        "inputFormat": "org.apache.druid.data.input.parquet.DruidParquetInputF
        "paths": "path/to/file.parquet"
      },
      ...
    },
    "dataSchema": {
      "dataSource": "example",
      "parser": {
        "type": "parquet",
        "parseSpec": {
          "format": "timeAndDims",
          "timestampSpec": {
            "column": "timestamp",
            "format": "auto"
          },
          "dimensionsSpec": {
            "dimensions": [
              "dim1",
              "dim2",
              "dim3",
              "listDim"
            ],
            "dimensionExclusions": [],
            "spatialDimensions": []
          }
        }
      },
      ...
    },
    "tuningConfig": <hadoop-tuning-config>
  }
}
```

Parquet Avro Hadoop Parser

> [!WARNING] 考虑在该解析器之上使用 Parquet Hadoop Parser 来摄取Parquet文件。 两者之间的不同之处参见 Parquet Hadoop解析器 vs Parquet Avro Hadoop解析器 部分
>
> [!WARNING] 使用Parquet Avro Hadoop Parser需要同时加入 druid-parquet-extensions 和 druid-avro-extensions

该解析器用于 Hadoop批摄取, 该解析器首先将Parquet数据转换为Avro记录，然后再解析它们后摄入到Druid。在 `ioConfig` 中， `inputSpec` 中的 `inputFormat` 必须设置为 `org.apache.druid.data.input.parquet.DruidParquetAvroInputFormat` 。

Parquet Avro Hadoop 解析器支持自动字段发现，如果提供了一个带有 `avro parquetSpec` 的 `flattenSpec` 也支持展平。 Parquet嵌套 list 和 map 逻辑类型 应与所有受支持类型的JSON path表达式一起正确操作。该解析器将Hadoop作业属性 `parquet.avro.add-list-element-records` 设置为false（通常默认为true），以便将原始列表元素"展开"为多值维度。

注意， `int96` Parquet值类型在该解析器中是不支持的。

字段	类型	描述	是否必填
type	String	应该填 `parquet-avro`	是
parseSpec	JSON对象	指定数据的时间戳和维度和一个可选的 `flattenSpec` , 应该是 `avro`	是
binaryAsString	布尔类型	指定逻辑上未标记为字符串的二进制orc列是否应被视为UTF-8编码字符串。	否（默认为false）

当时间维度是一个 date类型的列, 则无需指定一个格式。 当格式为UTF8的String, 则要么指定为 `auto` ， 或者显式的指定一个 时间格式。

示例

```
{
  "type": "index_hadoop",
  "spec": {
    "ioConfig": {
      "type": "hadoop",
      "inputSpec": {
        "type": "static",
        "inputFormat": "org.apache.druid.data.input.parquet.DruidParquetAvroIn
        "paths": "path/to/file.parquet"
      },
      ...
    },
    "dataSchema": {
      "dataSource": "example",
      "parser": {
        "type": "parquet-avro",
        "parseSpec": {
          "format": "avro",
          "flattenSpec": {
            "useFieldDiscovery": true,
            "fields": [
              {
                "type": "path",
                "name": "nestedDim",
                "expr": "$.nestedData.dim1"
              },
              {
                "type": "path",
                "name": "listDimFirstItem",
                "expr": "$.listDim[1]"
              }
            ]
          },
          "timestampSpec": {
            "column": "timestamp",
            "format": "auto"
          },
          "dimensionsSpec": {
            "dimensions": [],
            "dimensionExclusions": [],
            "spatialDimensions": []
          }
        }
      },
      ...
    },
    "tuningConfig": <hadoop-tuning-config>
    }
  }
}
```

Avro Stream Parser

> [!WARNING] 需要添加 druid-avro-extensions 来使用Avro Stream解析器

该解析器用于 流式摄取, 直接从一个流来读取数据。

字段	类型	描述	是否必须
type	String	`avro_stream`	否
avroBytesDecoder	JSON对象	指定如何对Avro记录进行解码	是
parseSpec	JSON对象	指定数据的时间戳和维度。 应该是一个 `avro` parseSpec	是

Avro parseSpec包含一个使用"root"或者"path"类型的 `flattenSpec` ，以便可以用来读取嵌套的avro数据。 "jq"类型在Avro中目前还不支持。

以下示例展示了一个具有**schema repo**avro解码器的 `Avro stream parser` ：

```
"parser" : {
  "type" : "avro_stream",
  "avroBytesDecoder" : {
    "type" : "schema_repo",
    "subjectAndIdConverter" : {
      "type" : "avro_1124",
      "topic" : "${YOUR_TOPIC}"
    },
    "schemaRepository" : {
      "type" : "avro_1124_rest_client",
      "url" : "${YOUR_SCHEMA_REPO_END_POINT}",
    }
  },
  "parseSpec" : {
    "format": "avro",
    "timestampSpec": <standard timestampSpec>,
    "dimensionsSpec": <standard dimensionsSpec>,
    "flattenSpec": <optional>
  }
}
```

Avro Bytes Decoder

如果 `type` 未被指定， `avroBytesDecoder` 默认使用 `schema_repo` 。

基于Avro Bytes Decoder的 `inline schema`

> [!WARNING] "schema_inline"解码器使用固定schema读取Avro记录，不支持schema迁移。如果将来可能需要迁移schema，请考虑其他解码器之一，所有解码器都使用一个消息头，该消息头允许解析器识别正确的Avro schema以读取记录。

如果可以使用同一schema读取所有输入事件，则可以使用此解码器。在这种情况下，在输入任务JSON本身中指定schema，如下所述:

```
...
"avroBytesDecoder": {
  "type": "schema_inline",
  "schema": {
    //your schema goes here, for example
    "namespace": "org.apache.druid.data",
    "name": "User",
    "type": "record",
    "fields": [
      { "name": "FullName", "type": "string" },
      { "name": "Country", "type": "string" }
    ]
  }
}
...
```

基于Avro Bytes Decoder的 `multiple inline schemas`

如果不同的输入事件可以有不同的读取schema，请使用此解码器。在这种情况下，在输入任务JSON本身中指定schema，如下所述:

```
...
"avroBytesDecoder": {
  "type": "multiple_schemas_inline",
  "schemas": {
    //your id -> schema map goes here, for example
    "1": {
      "namespace": "org.apache.druid.data",
      "name": "User",
      "type": "record",
      "fields": [
        { "name": "FullName", "type": "string" },
        { "name": "Country", "type": "string" }
      ]
    },
    "2": {
      "namespace": "org.apache.druid.otherdata",
      "name": "UserIdentity",
      "type": "record",
      "fields": [
        { "name": "Name", "type": "string" },
        { "name": "Location", "type": "string" }
      ]
    },
    ...
    ...
  }
}
...
```

注意，它本质上是一个整数Schema ID到avro schema对象的映射。此解析器假定记录具有以下格式。第一个1字节是版本，必须始终为1, 接下来的4个字节是使用大端字节顺序序列化的整数模式ID。其余字节包含序列化的avro消息。

基于Avro Bytes Decoder的 `SchemaRepo`

Avro Bytes Decorder首先提取输入消息的 `subject` 和 `id`， 然后使用她们去查找用来解码Avro记录的Avro schema，详情可以参见 Schema repo 和 AVRO-1124 。您需要一个类似schema repo的http服务来保存avro模式。有关在消息生成器端注册架构的信息，请见
`org.apache.druid.data.input.AvroStreamInputRowParserTest#testParse()`

字段	类型	描述	是否必须
type	String	`schema_repo`	否
subjectAndIdConverter	JSON对象	指定如何从消息字节中提取subject和id	是
schemaRepository	JSON对象	指定如何从subject和id查找Avro Schema	是

Avro-1124 Subject 和 Id 转换器 这部分描述了 `schema_avro` avro 字节解码器中的 `subjectAndIdConverter` 的格式

字段	类型	描述	是否必须
type	String	`avro_1124`	否
topic	String	指定Kafka流的主题	是

Avro-1124 Schema Repository 这部分描述了 `schema_avro` avro 字节解码器中的 `schemaRepository` 的格式

字段	类型	描述	是否必须
type	String	`avro_1124_rest_client`	否
url	String	指定Avro-1124 schema repository的http url	是

Confluent Schema Registry-based Avro Bytes Decoder

这个Avro字节解码器首先从输入消息字节中提取一个唯一的id，然后使用它在用于从字节解码Avro记录的模式注册表中查找模式。有关详细信息，请参阅schema注册 文档 和 存储库。

字段	类型	描述	是否必须
type	String	`schema_registry`	否
url	String	指定架构注册表的url	是
capacity	整型数字	指定缓存的最大值（默认为Integer.MAX_VALUE）	否

```
...
"avroBytesDecoder" : {
   "type" : "schema_registry",
   "url" : <schema-registry-url>
}
...
```

Protobuf Parser

> [!WARNING] 需要添加 druid-protobuf-extensions 来使用Protobuf解析器

此解析器用于 流接收，并直接从流中读取协议缓冲区数据。

字段	类型	描述	是否必须
type	String	`protobuf`	是
descriptor	String	类路径或URL中的Protobuf描述符文件名	是
protoMessageType	String	描述符中的Protobuf消息类型。可接受短名称和全限定名称。如果未指定，解析器将使用描述符中找到的第一个消息类型	否
parseSpec	JSON对象	指定数据的时间戳和维度。格式必须为JSON。有关更多配置选项，请参阅 JSON ParseSpec。请注意，不再支持timeAndDims parseSpec	是

样例规范：

```
"parser": {
  "type": "protobuf",
  "descriptor": "file:///tmp/metrics.desc",
  "protoMessageType": "Metrics",
  "parseSpec": {
    "format": "json",
    "timestampSpec": {
      "column": "timestamp",
      "format": "auto"
    },
    "dimensionsSpec": {
      "dimensions": [
        "unit",
        "http_method",
        "http_code",
        "page",
        "metricType",
        "server"
      ],
      "dimensionExclusions": [
        "timestamp",
        "value"
      ]
    }
  }
}
```

有关更多详细信息和示例，请参见 扩展说明。

ParseSpec

> [!WARNING] Parser 在 本地批任务, kafka索引任务 和Kinesis索引任务 中已经废弃，在这些类型的摄入中考虑使用 inputFormat

`parseSpec` 有两个目的：

- String解析器使用 `parseSpec` 来决定输入行的格式（例如： JSON，CSV，TSV）
- 所有的解析器使用 `parseSpec` 来决定输入行的timestamp和dimensions

如果 `format` 没有被包含， `parseSpec` 默认为 `tsv`

JSON解析规范

与字符串解析器一起用于加载JSON。

字段	类型	描述	是否必填
format	String	`json`	否
timestampSpec	JSON对象	指定timestamp的列和格式	是
dimensionsSpec	JSON对象	指定数据的dimensions	是
flattenSpec	JSON对象	指定嵌套的JSON数据的展平配置，详情可见 flattenSpec	否

示例规范：

```
"parseSpec": {
  "format" : "json",
  "timestampSpec" : {
    "column" : "timestamp"
  },
  "dimensionSpec" : {
    "dimensions" : ["page","language","user","unpatrolled","newPage","robot","
  }
}
```

JSON Lowercase解析规范

> [!WARNING] `JsonLowerCase` 解析器已经废弃，并可能在Druid将来的版本中移除

这是JSON ParseSpec的一个特殊变体，它将传入JSON数据中的所有列名小写。如果您正在从Druid 0.6.x更新到druid0.7.x，正在直接接收具有混合大小写列名的JSON，没有任何ETL来将这些列名转换大小写，并且希望进行包含使用0.6.x和0.7.x创建的数据的查询，则需要此parseSpec。

字段	类型	描述	是否必填
format	String	`jsonLowerCase`	是
timestampSpec	JSON对象	指定timestamp的列和格式	是
dimensionsSpec	JSON对象	指定数据的dimensions	是

CSV解析规范

与字符串解析器一起用于加载CSV， 字符串通过使用 `com.opencsv` 库来进行解析。

字段	类型	描述	是否必填
format	String	`csv`	是
timestampSpec	JSON对象	指定timestamp的列和格式	是
dimensionsSpec	JSON对象	指定数据的dimensions	是
listDelimiter	String	多值维度的定制分隔符	否（默认为 `ctrl + A`）
columns	JSON数组	指定数据的列	是

示例规范：

```
"parseSpec": {
  "format" : "csv",
  "timestampSpec" : {
    "column" : "timestamp"
  },
  "columns" : ["timestamp","page","language","user","unpatrolled","newPage","r
  "dimensionsSpec" : {
    "dimensions" : ["page","language","user","unpatrolled","newPage","robot","
  }
}
```

CSV索引任务

如果输入文件包含头，则 `columns` 字段是可选的，不需要设置。相反，您可以将 `hasHeaderRow` 字段设置为 `true` ，这将使Druid自动从标题中提取列信息。否则，必须设置 `columns` 字段，并确保该字段必须以相同的顺序与输入数据的列匹配。

另外，可以通过在parseSpec中设置 `skipHeaderRows` 跳过一些标题行。如果同时设置了 `skipHeaderRows` 和 `HashHeaderRow` 选项，则首先应用 `skipHeaderRows` 。例如，如果将 `skipHeaderRows` 设置为2， `hasHeaderRow` 设置为true，Druid将跳过前两行，然后从第三行提取列信息。

请注意，`hasHeaderRow` 和 `skipHeaderRows` 仅对非Hadoop批索引任务有效。其他类型的索引任务将失败，并出现异常。

其他CSV摄入任务

必须包含 `columns` 字段，并确保字段的顺序与输入数据的列以相同的顺序匹配。

TSV/Delimited解析规范

与字符串解析器一起使用此命令可加载不需要特殊转义的任何分隔文本。默认情况下，分隔符是一个制表符，因此这将加载TSV。

字段	类型	描述	是否必填
format	String	`csv`	是
timestampSpec	JSON对象	指定timestamp的列和格式	是
dimensionsSpec	JSON对象	指定数据的dimensions	是
delimiter	String	数据值的定制分隔符	否（默认为`\t`）
listDelimiter	String	多值维度的定制分隔符	否（默认为`ctrl + A`）
columns	JSON数组	指定数据的列	是

示例规范：

```
"parseSpec": {
  "format" : "tsv",
  "timestampSpec" : {
    "column" : "timestamp"
  },
  "columns" : ["timestamp","page","language","user","unpatrolled","newPage","r
  "delimiter":"| ",
  "dimensionsSpec" : {
    "dimensions" : ["page","language","user","unpatrolled","newPage","robot","
  }
}
```

请确保将 `delimiter` 更改为数据的适当分隔符。与CSV一样，您必须指定要索引的列和列的子集。

TSV(Delimited)索引任务

如果输入文件包含头，则 `columns` 字段是可选的，不需要设置。相反，您可以将 `hasHeaderRow` 字段设置为 `true`，这将使Druid自动从标题中提取列信息。否则，必须设置 `columns` 字段，并确保该字段必须以相同的顺序与输入数据的列匹配。

另外，可以通过在parseSpec中设置 `skipHeaderRows` 跳过一些标题行。如果同时设置了 `skipHeaderRows` 和 `HashHeaderRow` 选项，则首先应用 `skipHeaderRows`。例如，如果将 `skipHeaderRows` 设置为2，`hasHeaderRow` 设置为true，Druid将跳过前两行，然后从第三行提取列信息。

请注意， `hasHeaderRow` 和 `skipHeaderRows` 仅对非Hadoop批索引任务有效。其他类型的索引任务将失败，并出现异常。

其他TSV(Delimited)摄入任务

必须包含 `columns` 字段，并确保字段的顺序与输入数据的列以相同的顺序匹配。

多值维度

对于TSV和CSV数据，维度可以有多个值。要为多值维度指定分隔符，请在 `parseSpec` 中设置 `listDelimiter` 。

JSON数据也可以包含多值维度。维度的多个值必须在接收的数据中格式化为 `JSON数组` ，不需要额外的 `parseSpec` 配置。

正则解析规范

```
"parseSpec":{
  "format" : "regex",
  "timestampSpec" : {
    "column" : "timestamp"
  },
  "dimensionsSpec" : {
    "dimensions" : [<your_list_of_dimensions>]
  },
  "columns" : [<your_columns_here>],
  "pattern" : <regex pattern for partitioning data>
}
```

`columns` 字段必须以相同的顺序与regex匹配组的列匹配。如果未提供列，则默认列名称（“column_1”、“column2”、…”列“）将被分配, 确保列名包含所有维度

JavaScript解析规范

```
"parseSpec":{
  "format" : "javascript",
  "timestampSpec" : {
    "column" : "timestamp"
  },
  "dimensionsSpec" : {
    "dimensions" : ["page","language","user","unpatrolled","newPage","robot","
  },
  "function" : "function(str) { var parts = str.split(\"-\"); return { one: pa
}
```

注意: JavaScript解析器必须完全解析数据，并在JS逻辑中以 `{key:value}` 格式返回。这意味着任何展平或解析多维值都必须在这里完成。

> [!WARNING] 默认情况下禁用基于JavaScript的功能。有关使用Druid的JavaScript功能的指南，包括如何启用它的说明，请参阅 Druid JavaScript编程指南。

时间和维度解析规范

与非字符串解析器一起使用，为它们提供时间戳和维度信息。非字符串解析器独立处理所有格式化决策，而不使用ParseSpec。

字段	类型	描述	是否必填
format	String	`timeAndDims`	是
timestampSpec	JSON对象	指定timestamp的列和格式	是
dimensionsSpec	JSON对象	指定数据的dimensions	是

Orc解析规范

与Hadoop ORC解析器一起使用来加载ORC文件

字段	类型	描述	是否必填
format	String	`orc`	否
timestampSpec	JSON对象	指定timestamp的列和格式	是
dimensionsSpec	JSON对象	指定数据的dimensions	是
flattenSpec	JSON对象	指定嵌套的JSON数据的展平配置，详情可见 flattenSpec	否

Parquet解析规范

与Hadoop Parquet解析器一起使用来加载Parquet文件

字段	类型	描述	是否必填
format	String	`parquet`	否
timestampSpec	JSON对象	指定timestamp的列和格式	是
dimensionsSpec	JSON对象	指定数据的dimensions	是
flattenSpec	JSON对象	指定嵌套的JSON数据的展平配置，详情可见 flattenSpec	否

Schema设计

Druid数据模型

有关一般信息，请查看摄取概述页面上有关 Druid数据模型 的文档。本页的其余部分将讨论来自其他类型系统的用户的提示，以及一般提示和常见做法。

- Druid数据存储在 数据源 中，与传统RDBMS中的表类似。
- Druid数据源可以在摄取过程中使用或不使用 rollup 。启用rollup后，Druid会在接收期间部分聚合您的数据，这可能会减少其行数，减少存储空间，并提高查询性能。禁用rollup后，Druid为输入数据中的每一行存储一行，而不进行任何预聚合。
- Druid的每一行都必须有时间戳。数据总是按时间进行分区，每个查询都有一个时间过滤器。查询结果也可以按时间段（如分钟、小时、天等）进行细分。
- 除了timestamp列之外，Druid数据源中的所有列都是dimensions或metrics。这遵循 OLAP数据的标准命名约定。
- 典型的生产数据源有几十到几百列。
- dimension列 按原样存储，因此可以在查询时对其进行筛选、分组或聚合。它们总是单个字符串、字符串数组、单个long、单个double或单个float。
- Metrics列 是 预聚合 存储的，因此它们只能在查询时聚合（不能按筛选或分组）。它们通常存储为数字（整数或浮点数），但也可以存储为复杂对象，如 HyperLogLog草图或近似分位数草图。即使禁用了rollup，也可以在接收时配置metrics，但在启用汇总时最有用。

与其他设计模式类比

关系模型

（如 Hive 或者 PostgreSQL）

Druid数据源通常相当于关系数据库中的表。Druid的 lookups特性 可以类似于数据仓库样式的维度表，但是正如您将在下面看到的，如果您能够摆脱它，通常建议您进行非规范化。

关系数据建模的常见实践涉及 规范化 的思想：将数据拆分为多个表，从而减少或消除数据冗余。例如，在"sales"表中，最佳实践关系建模要求将"product id"列作为外键放入单独的"products"表中，该表依次具有"product id"、"product name"和"product category"列, 这可以防止产品名称和类别需要在"sales"表中引用同一产品的不同行上重复。

另一方面，在Druid中，通常使用在查询时不需要连接的完全平坦的数据源。在"sales"表的例子中，在Druid中，通常直接将"product_id"、"product_name"和"product_category"作为维度存储在Druid "sales"数据源中，而不使用单独的"products"表。完全平坦的模式大大提高了性能，因为查询时不需要连接。作为一个额外的速度提升，这也允许Druid的查询层直接操作压缩字典编码的数据。因为Druid使用字典编码来有效地为字符串列每行存储一个整数, 所以可能与直觉相反，这并*没有*显著增加相对于规范化模式的存储空间。

如果需要的话，可以通过使用 lookups 规范化Druid数据源，这大致相当于关系数据库中的维度表。在查询时，您将使用Druid的SQL `LOOKUP` 查找函数或者原生 `lookup` 提取函数，而不是像在关系数据库中那样使用JOIN关键字。由于lookup表会增加内存占用并在查询时产生更多的计算开销，因此仅当需要更新lookup表并立即反映主表中已摄取行的更改时，才建议执行此操作。

在Druid中建模关系数据的技巧：

- Druid数据源没有主键或唯一键，所以跳过这些。
- 如果可能的话，去规格化。如果需要定期更新dimensions/lookup并将这些更改反映在已接收的数据中，请考虑使用 lookups 进行部分规范化。
- 如果需要将两个大型的分布式表连接起来，则必须在将数据加载到Druid之前执行此操作。Druid不支持两个数据源的查询时间连接。lookup在这里没有帮助，因为每个lookup表的完整副本存储在每个Druid服务器上，所以对于大型表来说，它们不是一个好的选择。
- 考虑是否要为预聚合启用rollup，或者是否要禁用rollup并按原样加载现有数据。Druid中的Rollup类似于在关系模型中创建摘要表。

时序模型

（如 OpenTSDB 或者 InfluxDB）

与时间序列数据库类似，Druid的数据模型需要时间戳。Druid不是时序数据库，但它同时也是存储时序数据的自然选择。它灵活的数据模型允许它同时存储时序和非时序数据，甚至在同一个数据源中。

为了在Druid中实现时序数据的最佳压缩和查询性能，像时序数据库经常做的一样，按照metric名称进行分区和排序很重要。有关详细信息，请参见 分区和排序。

在Druid中建模时序数据的技巧：

- Druid并不认为数据点是"时间序列"的一部分。相反，Druid对每一点分别进行摄取和聚合
- 创建一个维度，该维度指示数据点所属系列的名称。这个维度通常被称为"metric"或"name"。不要将名为"metric"的维度与Druid Metrics的概念混淆。将它放在"dimensionsSpec"中维度列表的第一个位置，以获得最佳性能（这有助于提高局部性；有关详细信息，请参阅下面的 分区和排序）
- 为附着到数据点的属性创建其他维度。在时序数据库系统中，这些通常称为"标签"
- 创建与您希望能够查询的聚合类型相对应的 Druid Metrics。通常这包括"sum"、"min"和"max"（在long、float或double中的一种）。如果你想计算百分位数或分位数，可以使用Druid的 近似聚合器
- 考虑启用 rollup，这将允许Druid潜在地将多个点合并到Druid数据源中的一行中。如果希望以不同于原始发出的时间粒度存储数据，则这可能非常有用。如果要在同一个数据源中组合时序和非时序数据，它也很有用
- 如果您提前不知道要摄取哪些列，请使用空的维度列表来触发 维度列的自动检测

日志聚合模型

（如 ElasticSearch 或者 Splunk）

与日志聚合系统类似，Druid提供反向索引，用于快速搜索和筛选。Druid的搜索能力通常不如这些系统发达，其分析能力通常更为发达。Druid和这些系统之间的主要数据建模差异在于，在将数据摄取到Druid中时，必须更加明确。Druid列具有特定的类型，而Druid目前不支持嵌套数据。

在Druid中建模日志数据的技巧：

- 如果您提前不知道要摄取哪些列，请使用空维度列表来触发 维度列的自动检测
- 如果有嵌套数据，请使用 展平规范 将其扁平化
- 如果您主要有日志数据的分析场景，请考虑启用 rollup,这意味着您将失去从Druid中检索单个事件的能力，但您可能获得大量的压缩和查询性能提升

一般提示以及最佳实践

Rollup

Druid可以在接收数据时将其汇总，以最小化需要存储的原始数据量。这是一种汇总或预聚合的形式。有关更多详细信息，请参阅摄取文档的 汇总部分。

分区与排序

对数据进行最佳分区和排序会对占用空间和性能产生重大影响。有关更多详细信息，请参阅摄取文档的 分区部分。

Sketches高基维处理

在处理高基数列（如用户ID或其他唯一ID）时，请考虑使用草图(sketches)进行近似分析，而不是对实际值进行操作。当您使用草图(sketches)摄取数据时，Druid不存储原始原始数据，而是存储它的"草图(sketches)"，它可以在查询时输入到以后的计算中。草图(sketches)的常用场景包括 `count-distinct` 和分位数计算。每个草图都是为一种特定的计算而设计的。

一般来说，使用草图(sketches)有两个主要目的：改进rollup和减少查询时的内存占用。

草图(sketches)可以提高rollup比率，因为它们允许您将多个不同的值折叠到同一个草图(sketches)中。例如，如果有两行除了用户ID之外都是相同的（可能两个用户同时执行了相同的操作），则将它们存储在 `count-distinct sketch` 中而不是按原样，这意味着您可以将数据存储在一行而不是两行中。您将无法检索用户id或计算精确的非重复计数，但您仍将能够计算近似的非重复计数，并且您将减少存储空间。

草图(sketches)减少了查询时的内存占用，因为它们限制了需要在服务器之间洗牌的数据量。例如，在分位数计算中，Druid不需要将所有数据点发送到中心位置，以便对它们进行排序和计算分位数，而只需要发送点的草图。这可以将数据传输需要减少到仅千字节。

有关Druid中可用的草图的详细信息，请参阅 近似聚合器页面。

如果你更喜欢 视频，那就看一看吧！，一个讨论Druid Sketches的会议。

字符串 VS 数值维度

如果用户希望将列摄取为数值类型的维度（Long、Double或Float），则需要在 `dimensionsSpec` 的 `dimensions` 部分中指定列的类型。如果省略了该类型，Druid会将列作为默认的字符串类型。

字符串列和数值列之间存在性能折衷。数值列通常比字符串列更快分组。但与字符串列不同，数值列没有索引，因此可以更慢地进行筛选。您可能想尝试为您的用例找到最佳选择。

有关如何配置数值维度的详细信息，请参阅 `dimensionsSpec` 文档

辅助时间戳

Druid schema必须始终包含一个主时间戳, 主时间戳用于对数据进行 分区和排序，因此它应该是您最常筛选的时间戳。Druid能够快速识别和检索与主时间戳列的时间范围相对应的数据。

如果数据有多个时间戳，则可以将其他时间戳作为辅助时间戳摄取。最好的方法是将它们作为 毫秒格式的Long类型维度 摄取。如有必要，可以使用 `transformSpec` 和 `timestamp_parse` 等 表达式 将它们转换成这种格式，后者返回毫秒时间戳。

在查询时，可以使用诸如 `MILLIS_TO_TIMESTAMP` 、`TIME_FLOOR` 等 SQL时间函数 查询辅助时间戳。如果您使用的是原生Druid查询，那么可以使用 表达式。

嵌套维度

在编写本文时，Druid不支持嵌套维度。嵌套维度需要展平，例如，如果您有以下数据：

```
{"foo":{"bar": 3}}
```

然后在编制索引之前，应将其转换为：

```
{"foo_bar": 3}
```

Druid能够将JSON、Avro或Parquet输入数据展平化。请阅读 展平规格 了解更多细节。

计数接收事件数

启用rollup后，查询时的计数聚合器(count aggregator)实际上不会告诉您已摄取的行数。它们告诉您Druid数据源中的行数，可能小于接收的行数。

在这种情况下，可以使用*摄取时*的计数聚合器来计算事件数。但是，需要注意的是，在查询此Metrics时，应该使用 `longSum` 聚合器。查询时的 `count` 聚合器将返回时间间隔的Druid行数，该行数可用于确定rollup比率。

为了举例说明，如果摄取规范包含：

```
...
"metricsSpec" : [
      {
        "type" : "count",
        "name" : "count"
      },
...
```

您应该使用查询:

```
...
"aggregations": [
    { "type": "longSum", "name": "numIngestedEvents", "fieldName": "count" },
...
```

无schema的维度列

如果摄取规范中的 `dimensions` 字段为空，Druid将把不是timestamp列、已排除的维度和metric列之外的每一列都视为维度。

注意，当使用无schema摄取时，所有维度都将被摄取为字符串类型的维度。

包含与Dimension和Metric相同的列

一个具有唯一ID的工作流能够对特定ID进行过滤，同时仍然能够对ID列进行快速的唯一计数。如果不使用无schema维度，则通过将Metric的 `name` 设置为与维度不同的值来支持此场景。如果使用无schema维度，这里的最佳实践是将同一列包含两次，一次作为维度，一次作为 `hyperUnique` Metric。这可能涉及到ETL时的一些工作。

例如，对于无schema维度，请重复同一列：

```
{"device_id_dim":123, "device_id_met":123}
```

同时在 `metricsSpec` 中包含：

```
{ "type" : "hyperUnique", "name" : "devices", "fieldName" : "device_id_met" }
```

`device_id_dim` 将自动作为维度来被选取

数据管理

schema更新

数据源的schema可以随时更改，Apache Druid支持不同段之间的有不同的schema。

替换段文件

Druid使用数据源、时间间隔、版本号和分区号唯一地标识段。只有在某个时间粒度内创建多个段时，分区号才在段id中可见。例如，如果有小时段，但一小时内的数据量超过单个段的容量，则可以在同一小时内创建多个段。这些段将共享相同的数据源、时间间隔和版本号，但具有线性增加的分区号。

```
foo_2015-01-01/2015-01-02_v1_0
foo_2015-01-01/2015-01-02_v1_1
foo_2015-01-01/2015-01-02_v1_2
```

在上面的示例段中，`dataSource = foo`，`interval = 2015-01-01/2015-01-02`，version= `v1`，partitionNum= `0`。如果在以后的某个时间点，使用新的schema重新索引数据，则新创建的段将具有更高的版本id。

```
foo_2015-01-01/2015-01-02_v2_0
foo_2015-01-01/2015-01-02_v2_1
foo_2015-01-01/2015-01-02_v2_2
```

Druid批索引(基于Hadoop或基于IndexTask)保证了时间间隔内的原子更新。在我们的例子中，直到 `2015-01-01/2015-01-02` 的所有 `v2` 段加载到Druid集群中之前，查询只使用 `v1` 段，当加载完所有v2段并可查询后，所有查询都将忽略 `v1` 段并切换到 `v2` 段。不久之后，`v1` 段将从集群中卸载。

请注意，跨越多个段间隔的更新在每个间隔内都是原子的。在整个更新过程中它们不是原子的。例如，您有如下段：

```
foo_2015-01-01/2015-01-02_v1_0
foo_2015-01-02/2015-01-03_v1_1
foo_2015-01-03/2015-01-04_v1_2
```

`v2` 段将在构建后立即加载到集群中，并在段重叠的时间段内替换 `v1` 段。在完全加载 `v2` 段之前，集群可能混合了 `v1` 和 `v2` 段。

```
foo_2015-01-01/2015-01-02_v1_0
foo_2015-01-02/2015-01-03_v2_1
foo_2015-01-03/2015-01-04_v1_2
```

在这种情况下，查询可能会命中 `v1` 和 `v2` 段的混合。

在段中不同的schema

同一数据源的Druid段可能有不同的schema。如果一个字符串列（维度）存在于一个段中而不是另一个段中，则涉及这两个段的查询仍然有效。对缺少维度的段的查询将表现为该维度只有空值。类似地，如果一个段有一个数值列（metric），而另一个没有，那么查询缺少metric的段通常会"做正确的事情"。在此缺失的Metric上的使用聚合的行为类似于该Metric缺失。

压缩与重新索引

压缩合并是一种覆盖操作，它读取现有的一组段，将它们组合成一个具有较大但较少段的新集，并用新的压缩集覆盖原始集，而不更改它内部存储的数据。

出于性能原因，有时将一组段压缩为一组较大但较少的段是有益的，因为在接收和查询路径中都存在一些段处理和内存开销。

压缩任务合并给定时间间隔内的所有段。语法为：

```
{
    "type": "compact",
    "id": <task_id>,
    "dataSource": <task_datasource>,
    "ioConfig": <IO config>,
    "dimensionsSpec" <custom dimensionsSpec>,
    "metricsSpec" <custom metricsSpec>,
    "segmentGranularity": <segment granularity after compaction>,
    "tuningConfig" <parallel indexing task tuningConfig>,
    "context": <task context>
}
```

字段	描述	是否必须
`type`	任务类型，应该是 `compact`	是
`id`	任务id	否
`dataSource`	将被压缩合并的数据源名称	是
`ioConfig`	压缩合并任务的 `ioConfig`，详情见 Compaction ioConfig	是
`dimensionsSpec`	自定义 `dimensionsSpec`。压缩任务将使用此dimensionsSpec（如果存在），而不是生成dimensionsSpec。更多细节见下文。	否
`metricsSpec`	自定义 `metricsSpec`。如果指定了压缩任务，则压缩任务将使用此metricsSpec，而不是生成一个metricsSpec。	否
`segmentGranularity`	如果设置了此值，压缩合并任务将更改给定时间间隔内的段粒度。有关详细信息，请参阅 granularitySpec 的 `segmentGranularity`。行为见下表。	否
`tuningConfig`	并行索引任务的tuningConfig	否
`context`	任务的上下文	否

一个压缩合并任务的示例如下：

```
{
  "type" : "compact",
  "dataSource" : "wikipedia",
  "ioConfig" : {
    "type": "compact",
    "inputSpec": {
      "type": "interval",
      "interval": "2017-01-01/2018-01-01"
    }
  }
}
```

压缩任务读取时间间隔 `2017-01-01/2018-01-01` 的*所有分段*，并生成新分段。由于 `segmentGranularity` 为空，压缩后原始的段粒度将保持不变。要控制每个时间块的结果段数，可以设置 `maxRowsPerSegment` 或 `numShards`。请注意，您可以同时运行多个压缩任务。例如，您可以每月运行12个compactionTasks，而不是一整年只运行一个任务。

压缩任务在内部生成 `index` 任务规范，用于使用某些固定参数执行的压缩工作。例如，它的 `inputSource` 始终是 DruidInputSource，`dimensionsSpec` 和 `metricsSpec` 默认包含输入段的所有Dimensions和Metrics。

如果指定的时间间隔中没有加载数据段（或者指定的时间间隔为空），则压缩任务将以失败状态代码退出，而不执行任何操作。

除非所有输入段具有相同的元数据，否则输出段可以具有与输入段不同的元数据。

- Dimensions: 由于Apache Druid支持schema更改，因此即使是同一个数据源的一部分，各个段之间的维度也可能不同。如果输入段具有不同的维度，则输出段基本上包括输入段的所有维度。但是，即使输入段具有相同的维度集，维度顺序或维度的数据类型也可能不同。例如，某些维度的数据类型可以从 `字符串` 类型更改为基本类型，或者可以更改维度的顺序以获得更好的局部性。在这种情况下，在数据类型和排序方面，最近段的维度先于旧段的维度。这是因为最近的段更有可能具有所需的新顺序和数据类型。如果要使用自己的顺序和类型，可以在压缩任务规范中指定自定义 `dimensionsSpec` 。
- Roll-up: 仅当为所有输入段设置了 `rollup` 时，才会汇总输出段。有关详细信息，请参见 rollup。您可以使用 段元数据查询 检查段是否已被rollup。

压缩合并的IOConfig

压缩IOConfig需要指定 `inputSpec` ，如下所示。

字段	描述	是否必须
`type`	任务类型，固定为 `compact`	是
`inputSpec`	输入规范	是

目前有两种支持的 `inputSpec` ：

时间间隔 `inputSpec` ：

字段	描述	是否必须
`type`	任务类型，固定为 `interval`	是
`interval`	需要合并压缩的时间间隔	是

段 `inputSpec` ：

字段	描述	是否必须
`type`	任务类型，固定为 `segments`	是
`segments`	段ID列表	是

增加新的数据

Druid可以通过将新的段追加到现有的段集，来实现新数据插入到现有的数据源中。它还可以通过将现有段集与新数据合并并覆盖原始集来添加新数据。

Druid不支持按主键更新单个记录。

更新现有的数据

在数据源中摄取一段时间的数据并创建Apache Druid段之后，您可能需要对摄取的数据进行更改。有几种方法可以做到这一点。

使用lookups

如果有需要经常更新值的维度，请首先尝试使用 lookups。lookups的一个典型用例是，在Druid段中存储一个ID维度，并希望将ID维度映射到一个人类可读的字符串值，该字符串值可能需要定期更新。

重新摄取数据

如果基于lookups的技术还不够，您需要将想更新的时间块的数据重新索引到Druid中。这可以在覆盖模式（默认模式）下使用 批处理摄取 方法之一来完成。它也可以使用 流式摄取 来完成，前提是您先删除相关时间块的数据。

如果在批处理模式下进行重新摄取，Druid的原子更新机制意味着查询将从旧数据无缝地转换到新数据。

我们建议保留一份原始数据的副本，以防您需要重新摄取它。

使用基于Hadoop的摄取

本节假设读者理解如何使用Hadoop进行批量摄取。有关详细信息，请参见 Hadoop批处理摄取。Hadoop批量摄取可用于重新索引数据和增量摄取数据。

Druid使用 `ioConfig` 中的 `inputSpec` 来知道要接收的数据位于何处以及如何读取它。对于简单的Hadoop批接收， `static` 或 `granularity` 粒度规范类型允许您读取存储在深层存储中的数据。

还有其他类型的 `inputSpec` 可以启用重新索引数据和增量接收数据。

使用原生批摄取重新索引

本节假设读者了解如何使用 原生批处理索引 而不使用Hadoop的情况下执行批处理摄取（使用 `inputSource` 知道在何处以及如何读取输入数据）。 `DruidInputSource` 可以用来从Druid内部的段读取数据。请注意，**IndexTask**只用于原型设计，因为它必须在一个进程内完成所有处理，并且无法扩展。对于处理超过1GB数据的生产方案，请使用Hadoop批量摄取。

删除数据

Druid支持永久的将标记为"unused"状态（详情可见架构设计中的 段的生命周期）的段删除掉

杀死任务负责从元数据存储和深度存储中删除掉指定时间间隔内的不被使用的段

更多详细信息，可以看 杀死任务

永久删除一个段需要两步：

1. 段必须首先标记为"未使用"。当用户通过Coordinator API手动禁用段时，就会发生这种情况
2. 在段被标记为"未使用"之后，一个Kill任务将从Druid的元数据存储和深层存储中删除任何“未使用”的段

对于数据保留规则的文档，可以详细看 数据保留

对于通过Coordinator API来禁用段的文档，可以详细看 Coordinator数据源API

在本文档中已经包含了一个删除删除的教程，请看 数据删除教程

杀死任务

杀死任务删除段的所有信息并将其从深层存储中删除。在Druid的段表中，要杀死的段必须是未使用的（used==0）。可用语法为：

```
{
    "type": "kill",
    "id": <task_id>,
    "dataSource": <task_datasource>,
    "interval" : <all_segments_in_this_interval_will_die! >,
    "context": <task context>
}
```

数据保留

Druid支持保留规则，这些规则用于定义数据应保留的时间间隔和应丢弃数据的时间间隔。

Druid还支持将Historical进程分成不同的层，并且可以将保留规则配置为将特定时间间隔的数据分配给特定的层。

这些特性对于性能/成本管理非常有用；一个常见的场景是将Historical进程分为"热(hot)"层和"冷(cold)"层。

有关详细信息，请参阅 加载规则。

Apache Kafka 摄取数据

Kafka索引服务支持在Overlord上配置*supervisors*，supervisors通过管理Kafka索引任务的创建和生存期来便于从Kafka摄取数据。这些索引任务使用Kafka自己的分区和偏移机制读取事件，因此能够保证只接收一次（**exactly-once**）。supervisor监视索引任务的状态，以便于协调切换、管理故障，并确保维护可伸缩性和复制要求。

这个服务由 `druid-kafka-indexing-service` 这个druid核心扩展（详情请见 [扩展列表](../Development/extensions.md）所提供。

> [!WARNING] Kafka索引服务支持在Kafka 0.11.x中引入的事务主题。这些更改使Druid使用的Kafka消费者与旧的brokers不兼容。在使用此功能之前，请确保您的Kafka broker版本为0.11.x或更高版本。如果您使用的是旧版本的Kafka brokers，请参阅《Kafka升级指南》。

教程

本页包含基于Apache Kafka的摄取的参考文档。同样，您可以查看 Apache Kafka 教程 中的加载。

提交一个supervisor规范

Kafka索引服务需要同时在Overlord和MiddleManagers中加载 `druid-kafka-indexing-service` 扩展。 用于一个数据源的supervisor通过向 `http://<OVERLORD_IP>:<OVERLORD_PORT>/druid/indexer/v1/supervisor` 发送一个HTTP POST请求来启动，例如：

```
curl -X POST -H 'Content-Type: application/json' -d @supervisor-spec.json http
```

一个示例supervisor规范如下：

```
{
  "type": "kafka",
  "dataSchema": {
    "dataSource": "metrics-kafka",
    "timestampSpec": {
      "column": "timestamp",
      "format": "auto"
    },
    "dimensionsSpec": {
      "dimensions": [],
      "dimensionExclusions": [
        "timestamp",
        "value"
      ]
    },
    "metricsSpec": [
      {
        "name": "count",
        "type": "count"
      },
      {
        "name": "value_sum",
        "fieldName": "value",
        "type": "doubleSum"
      },
      {
        "name": "value_min",
        "fieldName": "value",
        "type": "doubleMin"
      },
      {
        "name": "value_max",
        "fieldName": "value",
        "type": "doubleMax"
      }
    ],
    "granularitySpec": {
      "type": "uniform",
      "segmentGranularity": "HOUR",
      "queryGranularity": "NONE"
    }
  },
  "tuningConfig": {
    "type": "kafka",
    "maxRowsPerSegment": 5000000
  },
  "ioConfig": {
    "topic": "metrics",
    "inputFormat": {
      "type": "json"
    },
    "consumerProperties": {
      "bootstrap.servers": "localhost:9092"
    },
    "taskCount": 1,
    "replicas": 1,
    "taskDuration": "PT1H"
  }
}
```

supervisor配置

字段	描述	是否必须
`type`	supervisor类型， 总是 `kafka`	是
`dataSchema`	Kafka索引服务在摄取时使用的schema。详情见 dataSchema	是
`ioConfig`	用于配置supervisor和索引任务的 KafkaSupervisorIOConfig，详情见以下	是
`tuningConfig`	用于配置supervisor和索引任务的 KafkaSupervisorTuningConfig，详情见以下	是

KafkaSupervisorTuningConfig

`tuningConfig` 是可选的， 如果未被配置的话，则使用默认的参数。

字段	类型	描述
`type`	String	索引任务类型， 总是 `kaf`
`maxRowsInMemory`	Integer	在持久化之前在内存中聚 数。该数值为聚合之后的 它不等于原始输入事件的 事件被聚合后的行数。 通 理所需的JVM堆内存。 使 `maxRowsInMemory * (2 + maxPendingPersists)` 来当 务的最大堆内存。通常用 置这个值，但是也需要根 点来决定，如果行的字节 户可能不想在内存中存储 应该设置这个值
`maxBytesInMemory`	Long	在持久化之前在内存中聚 节数。这是基于对内存使 估计，而不是实际使用量 在内部计算的，用户不需 索引任务的最大内存使用 `maxRowsInMemory * (2 + maxPendingPersists)`
`maxRowsPerSegment`	Integer	聚合到一个段中的行数， 合后的数值。 当 `maxRowsPerSegment` 或者 `maxTotalRows` 有一个值命 候，则触发handoff（数据 到深度存储）， 该动作也 `intermediateHandoffPerio` 隔发生一次。
`maxTotalRows`	Long	所有段的聚合后的行数， 后的行数。当 `maxRowsPer` 或者 `maxTotalRows` 有一 时候，则触发handoff（数 传到深度存储）， 该动作 每 `intermediateHandoffPe` 间隔发生一次。
`intermediateHandoffPeriod`	ISO8601 Period	确定触发持续化存储的周
`maxPendingPersists`	Integer	正在等待但启动的持久化 数量。 如果新的持久化任 此限制，则在当前运行的 之前，摄取将被阻止。索 大内存使用量是 `maxRowsI` `(2 + maxPendingPersists)`
`indexSpec`	Object	调整数据被如何索引。详 indexSpec

字段	类型	描述
`indexSpecForIntermediatePersists`		定义要在索引时用于中间持段的段存储格式选项。这中间段上的维度/度量压缩最终合并所需的内存。但段上禁用压缩可能会增加用，而在它们被合并到发之前使用它们，有关可能阅IndexSpec。
`reportParseExceptions`	Boolean	*已废弃*。如果为true，则在遇到的异常即停止摄取；false，则将跳过不可解析段。将 `reportParseExcept` 为 `true` 将覆盖 `maxParseExceptions` 和 `maxSavedParseExceptions` 置，将 `maxParseExceptions` `0` 并将 `maxSavedParseExc` 限制为不超过1。
`handoffConditionTimeout`	Long	段切换（持久化）可以等（超时时间）。该值要被于0的数，设置为0意味着等待不超时

字段	类型	描述

`resetOffsetAutomatically`	Boolean	控制当Druid需要读取Kafk
		的消息时的行为，比如当
		`OffsetOutOfRangeExceptio`
		时。
		如果为false，则异常将抛
		导致任务失败并停止接收
		这种情况，则需要手动干
		种情况；可能使用 重置 S
		API。此模式对于生产非常
		为它将使您意识到摄取的
		如果为true，Druid将根据
		`useEarliestOffset` 属性的
		（`true` 为 `earliest`，`f`
		`latest`）自动重置为Kafk
		的较早或最新偏移量。请
		能导致数据在您不知情的
		弃（如果 `useEarliestOffs`
		`false`）或 *重复*（如果
		`useEarliestOffset` 为 `tr`
		息将被记录下来，以标识
		置，但摄取将继续。这种
		生产环境非常有用，因为
		Druid尝试自动从问题中恢
		这些问题会导致数据被安
		复。
		该特性与Kafka的
		`auto.offset.reset` 消费者
		似

`workerThreads`	Integer	supervisor用于异步操作的
`chatThreads`	Integer	与索引任务的会话线程数
`chatRetries`	Integer	在任务没有响应之前，将 任务的HTTP请求的次数
`httpTimeout`	ISO8601 Period	索引任务的HTTP响应超时
`shutdownTimeout`	ISO8601 Period	supervisor尝试优雅的停 的超时时间
`offsetFetchPeriod`	ISO8601 Period	supervisor查询Kafka和索 获取当前偏移和计算滞后
`segmentWriteOutMediumFactory`	Object	创建段时要使用的段写入 信息见下文。

字段	类型	描述
`intermediateHandoffPeriod`	ISO8601 Period	段发生切换的频率。当 `maxRowsPerSegment` 或者 `maxTotalRows` 有一个值命候，则触发handoff（数据到深度存储）， 该动作也 `intermediateHandoffPerio` 隔发生一次。
`logParseExceptions`	Boolean	如果为true，则在发生解析 录错误消息，其中包含有 的行的信息。
`maxParseExceptions`	Integer	任务停止接收之前可发生 异常数。如果设置了 `reportParseExceptions` ， 被重写。
`maxSavedParseExceptions`	Integer	当出现解析异常时，Druic 最新的解析异 常。"maxSavedParseExc 定将保存多少个异常实例， 的异常将在 任务完成报告 完成后可用。如果设置 了 `reportParseExceptions` 会被重写。

IndexSpec

字段	类型	描述	是否必须
`bitmap`	Object	位图索引的压缩格式。 应该是一个JSON对象，详情见以下	否（默认为 `roaring`）
`dimensionCompression`	String	维度列的压缩格式。 从 `LZ4`，`LZF` 或者 `uncompressed` 选择	否（默认为 `LZ4`）
`metricCompression`	String	Metrics列的压缩格式。 从 `LZ4`，`LZF`，`uncompressed` 或者 `none` 选择	否（默认为 `LZ4`）
`longEncoding`	String	类型为long的Metric列和维度列的编码格式。从 `auto` 或者 `longs` 中选择。 `auto` 编码是根据列基数使用偏移量或查找表对值进行编码，并以可变大小存储它们。 `longs` 按原样存储值，每个值8字节。	否（默认为 `longs`）

Bitmap类型

对于Roaring位图：

字段	类型	描述	是否必须
`type`	String	必须为 `roaring`	是
`compressRunOnSerialization`	Boolean	使用一个运行长度编码，可以更节省空间	否（默认为 `true`）

对于Concise位图：

字段	类型	描述	是否必须
`type`	String	必须为 `concise`	是

SegmentWriteOutMediumFactory

字段	类型	描述	是否必须
`type`	String	对于可用选项，可以见 额外的Peon配置：SegmentWriteOutMediumFactory	是

KafkaSupervisorIOConfig

字段	类型	描
`topic`	String	要读取数据的Kafka主题 主题，因为不支持主题模
`inputFormat`	Object	`inputFormat` 指定如何解 部分 查看指定输入格式
`consumerProperties`	Map	传给Kafka消费者的一组 `bootstrap.servers` 的属 Broker列表，格式为: `<` `<BROKER_2>:<PORT_2>,...` 接， `keystore` ， `truststo` 被以一个字符串密码或者
`pollTimeout`	Long	Kafka消费者拉取消息记 秒单位
`replicas`	Integer	副本的数量，1意味着一 本）。副本任务将始终分 提供针对流程故障的恢复
`taskCount`	Integer	*一个副本集* 中 *读取*任务 读取任务的最大的数量将 `replicas` ，任务总数（*读* 数字的。 详情可以看下 `taskCount > {numKafkaPa` 数量会小于 `taskCount`
`taskDuration`	ISO8601 Period	任务停止读取数据、开始
`startDelay`	ISO8601 Period	supervisor开始管理任务
`useEarliestOffset`	Boolean	如果supervisor是第一次 Kafka获得一组起始偏移 Kafka中的最早偏移量还 情况下，后续任务将从分 因此此标志将仅在首次运
`completionTimeout`	ISO8601 Period	声明发布任务为失败并终 度。如果设置得太低，则 布。任务的发布时刻大约 持续)时间过后开始。
`lateMessageRejectionStartDateTime`	ISO8601 DateTime	用来配置一个时间，当消 间的时候，消息被拒绝。 `2016-01-01T11:00Z` ，sup `01T12:00Z` 创建了一个任 *01-01T11:00Z* 的消息将 据流有延迟消息，并且您 操作的管道（例如实时和 道），这可能有助于防止

字段	类型	描
`lateMessageRejectionPeriod`	ISO8601 Period	用来配置一个时间周期，期的时候，消息被拒绝。`PT1H`，supervisor 在 `20` 一个任务，则时间戳早于消息将被丢弃。 如果您且您有多个需要在同一段时和夜间批处理摄取管道并发问题。 **请特别注意**， `lateMessageRejecti` `lateMessageRejectionSta` 被指定。
`earlyMessageRejectionPeriod`	ISO8601 Period	用来配置一个时间周期，期的时候，消息被拒绝。`PT1H`,supervisor 在 `20` 一个任务，则时间戳晚于消息将被丢弃。**注意**，任续时间，例如，在super下。如果将 `earlyMessag` 得太低，则每当任务运行持续时间时，可能会导致

指定输入数据格式

Kafka索引服务同时支持通过 `inputFormat` 和 `parser` 来指定数据格式。`inputFormat` 是一种新的且推荐的用于Kafka索引服务中指定数据格式的方式，但是很遗憾的是目前它还不支持过时的 `parser` 所有支持的所有格式（未来会支持）。

`inputFormat` 支持的格式包括 `csv`，`delimited`），`json` 。可以使用 `parser` 来读取 `avro_stream`，`protobuf`，`thrift` 格式的数据。

操作

本节描述了一些supervisor API如何在Kafka索引服务中具体工作。对于所有的supervisor API，请查看 Supervisor APIs

获取supervisor的状态报告

`GET /druid/indexer/v1/supervisor/<supervisorId>/status` 返回由给定supervisor管理的任务当前状态的快照报告。报告中包括Kafka报告的最新偏移量、每个分区的使用者延迟，以及所有分区的聚合延迟。如果supervisor没有收到来自Kafka的最新偏移响应，则每个分区的使用者延迟可以报告为负值。聚合滞后值将始终大于等于0。

状态报告还包含supervisor的状态和最近引发的异常列表（报告为 `recentErrors`，其最大大小可以使用 `druid.supervisor.maxStoredExceptionEvents` 配置进行控制）。有两个字段与supervisor的状态相关- `state` 和 `detailedState` 。 `state` 字

段将始终是少数适用于任何类型的supervisor的通用状态之一，而 `detailedState` 字段将包含一个更具描述性的、特定实现的状态，该状态可以比通用状态字段更深入地了解supervisor的活动。

`state` 可能的值列表为：[`PENDING` , `RUNNING` , `SUSPENDED` , `STOPPING` , `UNHEALTHY_SUPERVISOR` , `UNHEALTHY_TASKS`]

`detailedState` 值与它们相应的 `state` 映射关系如下：

Detailed State	相应的State
UNHEALTHY_SUPERVISOR	UNHEALTHY_SUPERVISOR
UNHEALTHY_TASKS	UNHEALTHY_TASKS
UNABLE_TO_CONNECT_TO_STREAM	UNHEALTHY_SUPERVISOR
LOST_CONTACT_WITH_STREAM	UNHEALTHY_SUPERVISOR
PENDING（仅在第一次迭代中）	PENDING
CONNECTING_TO_STREAM（仅在第一次迭代中）	RUNNING
DISCOVERING_INITIAL_TASKS（仅在第一次迭代中）	RUNNING
CREATING_TASKS（仅在第一次迭代中）	RUNNING
RUNNING	RUNNING
SUSPENDED	SUSPENDED
STOPPING	STOPPING

在supervisor运行循环的每次迭代中，supervisor按顺序完成以下任务：

1. 从Kafka获取分区列表并确定每个分区的起始偏移量（如果继续，则基于最后处理的偏移量，如果这是一个新主题，则从流的开始或结束开始）。
2. 发现正在写入supervisor数据源的任何正在运行的索引任务，如果这些任务与supervisor的配置匹配，则采用这些任务，否则发出停止的信号。
3. 向每个受监视的任务发送状态请求，以更新我们对受监视任务的状态的视图。
4. 处理已超过 `taskDuration(任务持续时间)` 且应从读取状态转换为发布状态的任务。
5. 处理已完成发布的任务，并发出停止冗余副本任务的信号。
6. 处理失败的任务并清理supervisor的内部状态。
7. 将正常任务列表与请求的 `taskCount` 和 `replicas` 进行比较，并根据需要创建其他任务。

`detailedState` 字段将在supervisor启动后或从挂起恢复后第一次执行此运行循环时显示附加值（上述表格中那些标记为"仅限第一次迭代"的值）。这是为了解决初始化类型问题，即supervisor无法达到稳定状态（可能是因为它无法连接到Kafka，无法读取Kafka主题，或者无法与现有任务通信）。一旦supervisor稳定（也就是说，一旦完成完整的执行而没有遇到任何问题），`detailedState` 将显示 `RUNNING` 状态，直到它停止、挂起或达到故障阈值并过渡到不正常状态。

获取supervisor摄取状态报告

`GET /druid/indexer/v1/supervisor/<supervisorId>/stats` 返回由supervisor管理的每个任务的当前摄取行计数器的快照，以及行计数器的移动平均值。

可以在 任务报告：行画像 中查看详细信息。

supervisor健康检测

如果supervisor是健康的，则 `GET /druid/indexer/v1/supervisor/<supervisorId>/health` 返回 `200 OK`，如果是不健康的，则返回 `503 Service Unavailable` 。 健康状态是根据supervisor的 `state`（通过 `/status` 接口返回） 和 Overlord配置的阈值 `druid.supervisor.*` 来决定的。

更新现有的supervisor

`POST /druid/indexer/v1/supervisor` 可以被用来更新现有的supervisor规范。如果已存在同一数据源的现有supervisor，则调用此接口将导致：

- 正在运行的supervisor对其管理的任务发出停止读取并开始发布的信号
- 正在运行的supervisor退出
- 使用请求正文中提供的配置创建新的supervisor。该supervisor将保留现有的发布任务，并将从发布任务结束时的偏移开始创建新任务

因此，只需使用这个接口来提交新的schema，就可以实现无缝的schema迁移。

暂停和恢复supervisors

可以通过 `POST /druid/indexer/v1/supervisor/<supervisorId>/suspend` 和 `POST /druid/indexer/v1/supervisor/<supervisorId>/resume` 来暂停挂起和恢复一个supervisor。

注意，supervisor本身仍在运行并发出日志和metrics，它只会确保在supervisor恢复之前没有索引任务正在运行。

重置supervisors

`POST/druid/indexer/v1/supervisor/<supervisorId>/reset` 操作清除存储的偏移量，使supervisor开始从Kafka中最早或最新的偏移量读取偏移量（取决于 `useEarliestOffset` 的值）。清除存储的偏移量后，supervisor将终止并重新创建任务，以便任务开始从有效偏移量读取数据。

使用此操作时请小心！ 重置supervisor可能会导致跳过或读取Kafka消息两次，从而导致数据丢失或重复。

使用此操作的原因是：从由于缺少偏移而导致supervisor停止操作的状态中恢复。索引服务跟踪最新的持久化Kafka偏移量，以便跨任务提供准确的一次摄取保证。后续任务必须从上一个任务完成的位置开始读取，以便接受生成的段。如果Kafka中不再提供预期起始偏移量的消息（通常是因为消息保留期已过或主题已被删除并重新创建），supervisor将拒绝启动，在运行状态下的任务将失败。此操作使您能够从此情况中恢复。

请注意，要使此接口可用，必须运行supervisor。

终止supervisors

`POST /druid/indexer/v1/supervisor/<supervisorId>/terminate` 操作终止一个supervisor，并导致由该supervisor管理的所有关联的索引任务立即停止并开始发布它们的段。此supervisor仍将存在于元数据存储中，可以使用supervisor的历史API检索其历史记录，但不会在 "Get supervisor" API响应中列出，也无法检索其配置或状态报告。这个supervisor可以重新启动的唯一方法是向 "create" API提交一个正常工作的supervisor规范。

容量规划

Kafka索引任务运行在MiddleManager上，因此，其受限于MiddleManager集群的可用资源。 特别是，您应该确保有足够的worker（使用 `druid.worker.capacity` 属性配置）来处理supervisor规范中的配置。请注意，worker是在所有类型的索引任务之间共享的，因此，您应该计划好worker处理索引总负载的能力（例如批处理、实时任务、合并任务等）。如果您的worker不足，Kafka索引任务将排队并等待下一个可用的worker。这可能会导致查询只返回部分结果，但不会导致数据丢失（假设任务在Kafka清除这些偏移之前运行）。

正在运行的任务通常处于两种状态之一：*读取(reading)*或*发布(publishing)*。任务将在 `taskDuration(任务持续时间)` 内保持读取状态，在这时将转换为发布状态。只要生成段、将段推送到深层存储并由Historical进程加载和服务（或直到 `completionTimeout` 结束），任务将保持发布状态。

读取任务的数量由 `replicas` 和 `taskCount` 控制。 一般， 一共有 `replicas * taskCount` 个读取任务， 存在一个例外是当 taskCount > {numKafkaPartitions}, 在这种情况时 {numKafkaPartitions}个任务将被使用。 当 `taskDuration` 结束时，这些任务将被转换为发布状态并创建 `replicas * taskCount` 个新的读取任务。 因此，为了使得读取任务和发布任务可以并发的运行， 最小的容量应该是：

```
workerCapacity = 2 * replicas * taskCount
```

此值适用于这样一种理想情况：最多有一组任务正在发布，而另一组任务正在读取。在某些情况下，可以同时发布多组任务。如果发布时间（生成段、推送到深层存储、加载到历史记录中）> `taskDuration`，就会发生这种情况。这是一个有效的场景（正确性方面），但需要额外的worker容量来支持。一般来说，最好将 `taskDuration` 设置得足够大，以便在当前任务集开始之前完成上一个任务集的发布。

supervisor持久化

当通过 `POST /druid/indexer/v1/supervisor` 接口提交一个supervisor规范时，它将被持久化在配置的元数据数据库中。每个数据源只能有一个supervisor，为同一数据源提交第二个规范将覆盖前一个规范。

当一个Overlord获得领导地位时，无论是通过启动还是由于另一个Overlord失败，它都将为元数据数据库中的每个supervisor规范生成一个supervisor。然后，supervisor将发现正在运行的Kafka索引任务，如果它们与supervisor的配置兼容，则将尝试采用它们。如果它们不兼容，因为它们具有不同的摄取规范或分区分配，则任务将被终止，supervisor将创建一组新任务。这样，supervisor就可以在Overlord重启和故障转移期间坚持不懈地工作。

supervisor通过 `POST /druid/indexer/v1/supervisor/<supervisorId>/` 终止接口停止。这将在数据库中放置一个逻辑删除标记（以防止重新启动时重新加载supervisor），然后优雅地关闭当前运行的supervisor。当supervisor以这种方式关闭时，它将指示其托管的任务停止读取并立即开始发布其段。对关闭接口的调用将在所有任务发出停止信号后，但在任务完成其段的发布之前返回。

schema/配置变更

schema和配置更改是通过最初用于创建supervisor的 `POST /druid/indexer/v1/supervisor` 接口提交新的supervisor规范来处理的。Overlord将当前运行的supervisor优雅地关闭，这将导致由该supervisor管理的任务停止读取并开始发布其段。然后将启动一个新的supervisor，该supervisor将创建一组新的任务，这些任务将从先前发布任务关闭的偏移开始读取，但使用更新的schema。通过这种方式，可以在无需暂停摄取的条件下更新应用配置。

部署注意

每个Kafka索引任务将从分配给它的Kafka分区中消费的事件放在每个段粒度间隔的单个段中，直到达到 `maxRowsPerSegment` 、 `maxTotalRows` 或 `intermediateHandoffPeriod` 限制，此时将为进一步的事件创建此段粒度的新分区。Kafka索引任务还执行增量移交，这意味着任务创建的所有段在任务持续时间结束之前都不会被延迟。一旦达到 `maxRowsPerSegment` 、 `maxTotalRows` 或 `intermediateHandoffPeriod` 限制，任务在该时间点持有的所有段都将被传递，并且将为进一步的事件创建新的段集。这意味着任务可以运行更长的时间，而不必在MiddleManager进程的本地累积旧段，因此鼓励这样做。

Kafka索引服务可能仍然会产生一些小片段。假设任务持续时间为4小时，段粒度设置为1小时，supervisor在9:10启动，然后在13:10的4小时后，将启动新的任务集，并且间隔13:00-14:00的事件可以跨以前的和新的任务集拆分。如果您发现这成为一个问题，那么可以调度重新索引任务，以便将段合并到理想大小的新段中（每个段大约500-700 MB）。有关如何优化段大小的详细信息，请参见 "段大小优化"。还有一些工作正在进行，以支持碎片段的自动段压缩，以及不需要Hadoop的压缩(参见此处)。

本地批摄入

Apache Druid当前支持两种类型的本地批量索引任务， `index_parallel` 可以并行的运行多个任务， `index` 运行单个索引任务。 详情可以查看 基于Hadoop的摄取vs基于本地批摄取的对比 来了解基于Hadoop的摄取、本地简单批摄取、本地并行摄取三者的比较。

要运行这两种类型的本地批索引任务，请按以下指定编写摄取规范。然后将其发布到Overlord的 `/druid/indexer/v1/task` 接口，或者使用druid附带的 `bin/post-index-task` 。

教程

此页包含本地批处理摄取的参考文档。相反，如果要进行演示，请查看 加载文件教程，该教程演示了"简单"（单任务）模式

并行任务

并行任务（ `index_parallel` 类型）是用于并行批索引的任务。此任务只使用Druid的资源，不依赖于其他外部系统，如Hadoop。 `index_parallel` 任务是一个supervisor任务，它协调整个索引过程。supervisor分割输入数据并创建辅助任务来处理这些分割, 创建的worker将发布给Overlord，以便在MiddleManager或Indexer上安排和运行。一旦worker成功处理分配的输入拆分，它就会将生成的段列表报告给supervisor任务。supervisor定期检查工作任务的状态。如果其中一个失败，它将重试失败的任务，直到重试次数达到配置的限制。如果所有工作任务都成功，它将立即发布报告的段并完成摄取。

并行任务的详细行为是不同的，取决于 `partitionsSpec` ，详情可以查看 `partitionsSpec`

要使用此任务， `ioConfig` 中的 `inputSource` 应为*splittable(可拆分的)*， `tuningConfig` 中的 `maxNumConcurrentSubTasks` 应设置为大于1。否则，此任务将按顺序运行； `index_parallel` 任务将逐个读取每个输入文件并自行创建段。目前支持的可拆分输入格式有：

- `s3` 从AWS S3存储读取数据
- `gs` 从谷歌云存储读取数据
- `azure` 从Azure Blob存储读取数据
- `hdfs` 从HDFS存储中读取数据
- `http` 从HTTP服务中读取数据
- `local` 从本地存储中读取数据
- `druid` 从Druid数据源中读取数据

传统的 `firehose` 支持其他一些云存储类型。下面的 `firehose` 类型也是可拆分的。请注意， `firehose` 只支持文本格式。

- `static-cloudfiles`

您可能需要考虑以下事项：

- 您可能希望控制每个worker进程的输入数据量。这可以使用不同的配置进行控制，具体取决于并行摄取的阶段（有关更多详细信息，请参阅 `partitionsSpec` 。对于从 `inputSource` 读取数据的任务，可以在 `tuningConfig` 中设置 分割提示规范。对于合并无序段的任务，可以在 `tuningConfig` 中设置 `totalNumMergeTasks` 。
- 并行摄取中并发worker的数量由 `tuningConfig` 中的 `maxNumConcurrentSubTasks` 确定。supervisor检查当前正在运行的worker的数量，如果小于 `maxNumConcurrentSubTasks` ，则无论当前有多少任务槽可用，都会创建更多的worker。这可能会影响其他摄取性能。有关更多详细信息，请参阅下面的 容量规划部分。
- 默认情况下，批量摄取将替换它写入的任何段中的所有数据（在 `granularitySpec` 的间隔中）。如果您想添加到段中，请在 `ioConfig` 中设置 `appendToExisting` 标志。请注意，它只替换主动添加数据的段中的数据：如果 `granularitySpec` 的间隔中有段没有此任务写入的数据，则它们将被单独保留。如果任何现有段与 `granularitySpec` 的间隔部分重叠，则新段间隔之外的那些段的部分仍将可见。

任务符号

一个简易的任务如下所示：

```
{
  "type": "index_parallel",
  "spec": {
    "dataSchema": {
      "dataSource": "wikipedia_parallel_index_test",
      "timestampSpec": {
        "column": "timestamp"
      },
      "dimensionsSpec": {
        "dimensions": [
          "page",
          "language",
          "user",
          "unpatrolled",
          "newPage",
          "robot",
          "anonymous",
          "namespace",
          "continent",
          "country",
          "region",
          "city"
        ]
      },
      "metricsSpec": [
        {
          "type": "count",
              "name": "count"
            },
            {
              "type": "doubleSum",
              "name": "added",
              "fieldName": "added"
            },
            {
              "type": "doubleSum",
              "name": "deleted",
              "fieldName": "deleted"
            },
            {
              "type": "doubleSum",
              "name": "delta",
              "fieldName": "delta"
            }
        ],
        "granularitySpec": {
          "segmentGranularity": "DAY",
          "queryGranularity": "second",
          "intervals" : [ "2013-08-31/2013-09-02" ]
        }
    },
    "ioConfig": {
        "type": "index_parallel",
        "inputSource": {
          "type": "local",
          "baseDir": "examples/indexing/",
          "filter": "wikipedia_index_data*"
        },
        "inputFormat": {
          "type": "json"
        }
    },
    "tuningconfig": {
        "type": "index_parallel",
        "maxNumConcurrentSubTasks": 2
    }
  }
}
```

属性	描述	是否必须
`type`	任务类型，应当总是 `index_parallel`	是
`id`	任务ID。 如果该项没有显式的指定，Druid将使用任务类型、数据源名称、时间间隔、日期时间戳生成一个任务ID	否
`spec`	摄取规范包括数据schema、IOConfig 和 TuningConfig。详情见下边详细描述	是
`context`	Context包括了多个任务配置参数。详情见下边详细描述	否

`dataSchema`

该字段为必须字段。

可以参见 摄取规范中的dataSchema

如果在dataSchema的 `granularitySpec` 中显式地指定了 `intervals` ，则批处理摄取将锁定启动时指定的完整间隔，并且您将快速了解指定间隔是否与其他任务（例如Kafka摄取）持有的锁重叠。否则，在发现每个间隔时，批处理摄取将锁定该间隔，因此您可能只会在摄取过程中了解到该任务与较高优先级的任务重叠。如果显式指定 `intervals` ，则指定间隔之外的任何行都将被丢弃。如果您知道数据的时间范围，我们建议显式地设置 `intervals` ，以便锁定失败发生得更快，并且如果有一些带有意外时间戳的杂散数据，您不会意外地替换该范围之外的数据。

`ioConfig`

属性	描述	默认	是否必须
`type`	任务类型， 应当总是 `index_parallel`	none	是
`inputFormat`	`inputFormat` 用来指定如何解析输入数据	none	是
`appendToExisting`	创建段作为最新版本的附加分片，有效地附加到段集而不是替换它。仅当现有段集具有可扩展类型 `shardSpec` 时，此操作才有效。	false	否

`tuningConfig`

tuningConfig是一个可选项，如果未指定则使用默认的参数。 详情如下：

属性	描述	
`type`	任务类型，应当总是 `index_parallel`	n
`maxRowsPerSegment`	已废弃。使用 `partitionsSpec` 替代，被用来分片。 决定在每个段中有多少行。	5
`maxRowsInMemory`	用于确定何时应该从中间层持久化到磁盘。通常用户不需要设置此值，但根据数据的性质，如果行的字节数较短，则用户可能不希望在内存中存储一百万行，应设置此值。	1
`maxBytesInMemory`	用于确定何时应该从中间层持久化到磁盘。通常这是在内部计算的，用户不需要设置它。此值表示在持久化之前要在堆内存中聚合的字节数。这是基于对内存使用量的粗略估计，而不是实际使用量。用于索引的最大堆内存使用量为 `maxBytesInMemory * (2 + maxPendingResistent)`	最
`maxTotalRows`	已废弃。使用 `partitionsSpec` 替代。等待推送的段中的总行数。用于确定何时应进行中间推送。	2
`numShards`	已废弃。使用 `partitionsSpec` 替代。当使用 `hashed partitionsSpec` 时直接指定要创建的分片数。如果该值被指定了且在 `granularitySpec` 中指定了 `intervals` ，那么索引任务可以跳过确定间隔/分区传递数据。如果设置了 `maxRowsPerSegment` ，则无法指定 `numShards` 。	n
`splitHintSpec`	用于提供提示以控制每个第一阶段任务读取的数据量。根据输入源的实现，可以忽略此提示。有关更多详细信息，请参见 分割提示规范。	基
`partitionsSpec`	定义在每个时间块中如何分区数据。 参见 partitionsSpec	如 如 者
`indexSpec`	定义段在索引阶段的存储格式相关选项，参见 IndexSpec	n

属性	描述	
`indexSpecForIntermediatePersists`	定义要在索引时用于中间持久化临时段的段存储格式选项。这可用于禁用中间段上的维度/度量压缩，以减少最终合并所需的内存。但是，在中间段上禁用压缩可能会增加页缓存的使用，而在它们被合并到发布的最终段之前使用它们，有关可能的值，请参阅 IndexSpec。	与
`maxPendingPersists`	可挂起但未启动的最大持久化任务数。如果新的中间持久化将超过此限制，则在当前运行的持久化完成之前，摄取将被阻止。使用 `maxRowsInMemory * (2 + maxPendingResistents)` 索引扩展的最大堆内存使用量。	0 不
`forceGuaranteedRollup`	强制保证 最佳Rollup。最佳rollup优化了生成的段的总大小和查询时间，同时索引时间将增加。如果设置为true，则必须设置 `granularitySpec` 中的 `intervals` ，同时必须对 `partitionsSpec` 使用 `single_dim` 或者 `hashed` 。此标志不能与 `IOConfig` 的 `appendToExisting` 一起使用。有关更多详细信息，请参见下面的 "分段推送模式" 部分。	fa
`reportParseExceptions`	如果为true，则将引发解析期间遇到的异常并停止摄取；如果为false，则将跳过不可解析的行和字段。	fa
`pushTimeout`	段推送的超时毫秒时间。 该值必须设置为 >= 0, 0意味着永不超时	0
`segmentWriteOutMediumFactory`	创建段时使用的段写入介质。 参见 segmentWriteOutMediumFactory	未 (

属性	描述	
`maxNumConcurrentSubTasks`	可同时并行运行的最大worker数。无论当前可用的任务槽如何，supervisor都将生成最多为 `maxNumConcurrentSubTasks` 的worker。如果此值设置为1，supervisor将自行处理数据摄取，而不是生成worker。如果将此值设置为太大，则可能会创建太多的worker，这可能会阻止其他摄取。查看 容量规划 以了解更多详细信息。	1
`maxRetry`	任务失败后最大重试次数	3
`maxNumSegmentsToMerge`	单个任务在第二阶段可同时合并的段数的最大限制。仅在 `forceGuaranteedRollup` 被设置的时候使用。	1
`totalNumMergeTasks`	当 `partitionsSpec` 被设置为 `hashed` 或者 `single_dim` 时，在合并阶段用来合并段的最大任务数。	1
`taskStatusCheckPeriodMs`	检查运行任务状态的轮询周期（毫秒）。	1
`chatHandlerTimeout`	报告worker中的推送段超时。	P
`chatHandlerNumRetries`	重试报告worker中的推送段	5

分割提示规范

分割提示规范用于在supervisor创建输入分割时给出提示。请注意，每个worker处理一个输入拆分。您可以控制每个worker在第一阶段读取的数据量。

基于大小的分割提示规范 除HTTP输入源外，所有可拆分输入源都遵循基于大小的拆分提示规范。

属性	描述	默认值	是否必须
`type`	应当总是 `maxSize`	none	是
`maxSplitSize`	单个任务中要处理的输入文件的最大字节数。如果单个文件大于此数字，则它将在单个任务中自行处理（文件永远不会跨任务拆分）。	500MB	否

段分割提示规范

段分割提示规范仅仅用在 `DruidInputSource` (和过时的 `IngestSegmentFirehose`)

属性	描述	默认值	是否必须
`type`	应当总是 `segments`	none	是
`maxInputSegmentBytesPerTask`	单个任务中要处理的输入段的最大字节数。如果单个段大于此数字，则它将在单个任务中自行处理（输入段永远不会跨任务拆分）。	500MB	否

`partitionsSpec`

PartitionsSpec用于描述辅助分区方法。您应该根据需要的rollup模式使用不同的partitionsSpec。为了实现 最佳rollup，您应该使用 `hashed` （基于每行中维度的哈希进行分区）或 `single_dim` （基于单个维度的范围）。对于"尽可能rollup"模式，应使用 `dynamic` 。

三种 `partitionsSpec` 类型有着不同的特征。

PartitionsSpec	摄入速度	分区方式	支持的rollup模式	查询时的段修剪
`dynamic`	最快	基于段中的行数来进行分区	尽可能rollup	N/A
`hashed`	中等	基于分区维度的哈希值进行分区。此分区可以通过改进数据位置性来减少数据源大小和查询延迟。有关详细信息，请参见分区。	最佳rollup	N/A
`single_dim`	最慢	基于分区维度值的范围分区。段大小可能会根据分区键分布而倾斜。这可能通过改善数据位置性来减少数据源大小和查询延迟。有关详细信息，请参见分区。	最佳rollup	Broker可以使用分区信息提前修剪段以加快查询速度。由于Broker知道每个段中 `partitionDimension` 值的范围，因此，给定一个包含 `partitionDimension` 上的筛选器的查询，Broker只选取包含满足 `partitionDimension` 上的筛选器的行的段进行查询处理。

对于每一种partitionSpec，推荐的使用场景是：

- 如果数据有一个在查询中经常使用的均匀分布列，请考虑使用 `single_dim` partitionsSpec来最大限度地提高大多数查询的性能。
- 如果您的数据不是均匀分布的列，但在使用某些维度进行rollup时，需要具有较高的rollup汇总率，请考虑使用 `hashed` partitionsSpec。通过改善数据的局部性，可以减小数据源的大小和查询延迟。
- 如果以上两个场景不是这样，或者您不需要rollup数据源，请考虑使用 `dynamic` partitionsSpec。

Dynamic分区

属性	描述	默认值	是否必须
`type`	应该总是 `dynamic`	none	是
`maxRowsPerSegment`	用来分片。决定在每一个段中有多少行	5000000	否
`maxTotalRows`	等待推送的所有段的总行数。用于确定中间段推送的发生时间。	20000000	否

使用Dynamic分区，并行索引任务在一个阶段中运行：它将生成多个worker(`single_phase_sub_task` 类型)，每个worker都创建段。worker创建段的方式是：

- 每当当前段中的行数超过 `maxRowsPerSegment` 时，任务将创建一个新段。
- 一旦所有时间块中所有段中的行总数达到 `maxTotalRows` ，任务就会将迄今为止创建的所有段推送到深层存储并创建新段。

基于哈希的分区

属性	描述	默认值	是否必须
`type`	应该总是 `hashed`	none	是
`numShards`	直接指定要创建的分片数。如果该值被指定了，同时在 `granularitySpec` 中指定了 `intervals` ，那么索引任务可以跳过确定通过数据的间隔/分区	null	是
`partitionDimensions`	要分区的维度。留空可选择所有维度。	null	否

基于哈希分区的并行任务类似于 MapReduce。任务分为两个阶段运行，即 `部分段生成` 和 `部分段合并` 。

- 在 `部分段生成` 阶段，与MapReduce中的Map阶段一样，并行任务根据分割提示规范分割输入数据，并将每个分割分配给一个worker。每个worker（`partial_index_generate` 类型）从 `granularitySpec` 中的 `segmentGranularity（主分区键）` 读取分配的分割，然后按 `partitionsSpec` 中 `partitionDimensions（辅助分区键）` 的哈希值对行进行分区。分区数据存储在 MiddleManager 或 Indexer 的本地存储中。
- `部分段合并` 阶段类似于MapReduce中的Reduce阶段。并行任务生成一组新的worker（`partial_index_merge` 类型）来合并在前一阶段创建的分区数据。这里，分区数据根据要合并的时间块和分区维度的散列值进行洗牌；每个worker从多个MiddleManager/Indexer进程中读取落在同一时间块和同一散列值中的数据，并将其合并以创建最终段。最后，它们将最后的段一次推送到深层存储。

基于单一维度范围分区

> [!WARNING] 在并行任务的顺序模式下，当前不支持单一维度范围分区。尝试将 `maxNumConcurrentSubTasks` 设置为大于1以使用此分区方式。

属性	描述	默认值	是否必须
`type`	应该总是 `single_dim`	none	是
`partitionDimension`	要分区的维度。仅仅允许具有单一维度值的行	none	是
`targetRowsPerSegment`	在一个分区中包含的目标行数，应当是一个500MB ~ 1GB目标段的数值。	none	要么该值被设置，或者 `maxRowsPerSegment` 被设置。
`maxRowsPerSegment`	分区中要包含的行数的软最大值。	none	要么该值被设置，或者 `targetRowsPerSegment` 被设置。
`assumeGrouped`	假设输入数据已经按时间和维度分组。摄取将运行得更快，但如果违反此假设，则可能会选择次优分区	false	否

在 `single-dim` 分区下，并行任务分为3个阶段进行，即 `部分维分布` 、`部分段生成` 和 `部分段合并` 。第一个阶段是收集一些统计数据以找到最佳分区，另外两个阶段是创建部分段并分别合并它们，就像在基于哈希的分区中那样。

- 在 `部分维度分布` 阶段，并行任务分割输入数据，并根据分割提示规范将其分配给worker。每个worker任务（`partial_dimension_distribution` 类型）读取分配的分割并为 `partitionDimension` 构建直方图。并行任务从worker任务收集这些直方图，并根据 `partitionDimension` 找到最佳范围分区，以便在分区之间均匀分布行。请注意，`targetRowsPerSegment` 或 `maxRowsPerSegment` 将用于查找最佳分区。
- 在 `部分段生成` 阶段，并行任务生成新的worker任务（`partial_range_index_generate` 类型）以创建分区数据。每个worker任务都读取在前一阶段中创建的分割，根据 `granularitySpec` 中的 `segmentGranularity（主分区键）` 的时间块对行进行分区，然后根据在前一阶段中找到的范围分区对行进行分区。分区数据存储在 MiddleManager 或 Indexer的本地存储中。
- 在 `部分段合并` 阶段，并行索引任务生成一组新的worker任务（`partial_index_generic_merge` 类型）来合并在上一阶段创建的分区数据。这里，分区数据根据时间块和 `partitionDimension` 的值进行洗牌；每个工作任务从多个MiddleManager/Indexer进程中读取属于同一范围的同一分区中的段，并将它们合并以创建最后的段。最后，它们将最后的段推到深层存储。

> [!WARNING] 由于单一维度范围分区的任务在 `部分维度分布` 和 `部分段生成` 阶段对输入进行两次传递，因此如果输入在两次传递之间发生变化，任务可能会失败

HTTP状态接口

supervisor任务提供了一些HTTP接口来获取任务状态。

- `http://{PEON_IP}:{PEON_PORT}/druid/worker/v1/chat/{SUPERVISOR_TASK_ID}/mode`

如果索引任务以并行的方式运行，则返回 "parallel"，否则返回 "sequential"

- `http://{PEON_IP}:{PEON_PORT}/druid/worker/v1/chat/{SUPERVISOR_TASK_ID}/phase`

如果任务以并行的方式运行，则返回当前阶段的名称

- `http://{PEON_IP}:{PEON_PORT}/druid/worker/v1/chat/{SUPERVISOR_TASK_ID}/progress`

如果supervisor任务以并行的方式运行，则返回当前阶段的预估进度

一个示例结果如下：

```
{
  "running":10,
  "succeeded":0,
  "failed":0,
  "complete":0,
  "total":10,
  "estimatedExpectedSucceeded":10
}
```

- `http://{PEON_IP}:{PEON_PORT}/druid/worker/v1/chat/{SUPERVISOR_TASK_ID}/subtasks/running`

返回正在运行的worker任务的任务IDs，如果该supervisor任务以序列模式运行则返回一个空的列表

- `http://{PEON_IP}:{PEON_PORT}/druid/worker/v1/chat/{SUPERVISOR_TASK_ID}/subtaskspecs`

返回所有的worker任务规范，如果该supervisor任务以序列模式运行则返回一个空的列表

- `http://{PEON_IP}:{PEON_PORT}/druid/worker/v1/chat/{SUPERVISOR_TASK_ID}/subtaskspecs/running`

返回正在运行的worker任务规范，如果该supervisor任务以序列模式运行则返回一个空的列表

- `http://{PEON_IP}:{PEON_PORT}/druid/worker/v1/chat/{SUPERVISOR_TASK_ID}/subtaskspecs/complete`

返回已经完成的worker任务规范，如果该supervisor任务以序列模式运行则返回一个空的列表

- `http://{PEON_IP}:{PEON_PORT}/druid/worker/v1/chat/{SUPERVISOR_TASK_ID}/subtaskspec/{SUB_TASK_SPEC_ID}`

返回指定ID的worker任务规范，如果该supervisor任务以序列模式运行则返回一个HTTP 404

- `http://{PEON_IP}:{PEON_PORT}/druid/worker/v1/chat/{SUPERVISOR_TASK_ID}/subtaskspec/{SUB_TASK_SPEC_ID}/state`

返回指定ID的worker任务规范的状态，如果该supervisor任务以序列模式运行则返回一个HTTP 404。 返回的结果集中包括worker任务规范，当前任务状态(如果存在的话) 以及任务尝试历史记录。

一个示例结果如下：

```
{
  "spec": {
    "id": "index_parallel_lineitem_2018-04-20T22:12:43.610Z_2",
    "groupId": "index_parallel_lineitem_2018-04-20T22:12:43.610Z",
    "supervisorTaskId": "index_parallel_lineitem_2018-04-20T22:12:43.610Z",
    "context": null,
    "inputSplit": {
      "split": "/path/to/data/lineitem.tbl.5"
    },
    "ingestionSpec": {
      "dataSchema": {
        "dataSource": "lineitem",
        "timestampSpec": {
          "column": "l_shipdate",
          "format": "yyyy-MM-dd"
        },
        "dimensionsSpec": {
          "dimensions": [
            "l_orderkey",
            "l_partkey",
            "l_suppkey",
            "l_linenumber",
            "l_returnflag",
            "l_linestatus",
            "l_shipdate",
            "l_commitdate",
            "l_receiptdate",
            "l_shipinstruct",
            "l_shipmode",
            "l_comment"
          ]
        },
        "metricsSpec": [
          {
            "type": "count",
            "name": "count"
          },
          {
            "type": "longSum",
            "name": "l_quantity",
            "fieldName": "l_quantity",
            "expression": null
          },
          {
            "type": "doubleSum",
            "name": "l_extendedprice",
            "fieldName": "l_extendedprice",
            "expression": null
          },
          {
            "type": "doubleSum",
            "name": "l_discount",
            "fieldName": "l_discount",
            "expression": null
          },
          {
            "type": "doubleSum",
            "name": "l_tax",
            "fieldName": "l_tax",
            "expression": null
          }
        ],
        "granularitySpec": {
          "type": "uniform",
          "segmentGranularity": "YEAR",
          "queryGranularity": {
            "type": "none"
          },
          "rollup": true,
          "intervals": [
```

```
            "1980-01-01T00:00:00.000Z/2020-01-01T00:00:00.000Z"
          ]
        },
        "transformSpec": {
          "filter": null,
          "transforms": []
        }
      },
      "ioConfig": {
        "type": "index_parallel",
        "inputSource": {
          "type": "local",
          "baseDir": "/path/to/data/",
          "filter": "lineitem.tbl.5"
        },
        "inputFormat": {
          "format": "tsv",
          "delimiter": "|",
          "columns": [
            "l_orderkey",
            "l_partkey",
            "l_suppkey",
            "l_linenumber",
            "l_quantity",
            "l_extendedprice",
            "l_discount",
            "l_tax",
            "l_returnflag",
            "l_linestatus",
            "l_shipdate",
            "l_commitdate",
            "l_receiptdate",
            "l_shipinstruct",
            "l_shipmode",
            "l_comment"
          ]
        },
        "appendToExisting": false
      },
      "tuningConfig": {
        "type": "index_parallel",
        "maxRowsPerSegment": 5000000,
        "maxRowsInMemory": 1000000,
        "maxTotalRows": 20000000,
        "numShards": null,
        "indexSpec": {
          "bitmap": {
            "type": "roaring"
          },
          "dimensionCompression": "lz4",
          "metricCompression": "lz4",
          "longEncoding": "longs"
        },
        "indexSpecForIntermediatePersists": {
          "bitmap": {
            "type": "roaring"
          },
          "dimensionCompression": "lz4",
          "metricCompression": "lz4",
          "longEncoding": "longs"
        },
        "maxPendingPersists": 0,
        "reportParseExceptions": false,
        "pushTimeout": 0,
        "segmentWriteOutMediumFactory": null,
        "maxNumConcurrentSubTasks": 4,
        "maxRetry": 3,
        "taskStatusCheckPeriodMs": 1000,
        "chatHandlerTimeout": "PT10S",
        "chatHandlerNumRetries": 5,
        "logParseExceptions": false,
```

```
        "maxParseExceptions": 2147483647,
        "maxSavedParseExceptions": 0,
        "forceGuaranteedRollup": false,
        "buildV9Directly": true
      }
    }
  },
  "currentStatus": {
    "id": "index_sub_lineitem_2018-04-20T22:16:29.922Z",
    "type": "index_sub",
    "createdTime": "2018-04-20T22:16:29.925Z",
    "queueInsertionTime": "2018-04-20T22:16:29.929Z",
    "statusCode": "RUNNING",
    "duration": -1,
    "location": {
      "host": null,
      "port": -1,
      "tlsPort": -1
    },
    "dataSource": "lineitem",
    "errorMsg": null
  },
  "taskHistory": []
}
```


- `http://{PEON_IP}:{PEON_PORT}/druid/worker/v1/chat/{SUPERVISOR_TASK_ID}/subtaskspec/{SUB_TASK_SPEC_ID}/history`

返回被指定ID的worker任务规范的任务尝试历史记录，如果该supervisor任务以序列模式运行则返回一个HTTP 404

容量规划

不管当前有多少任务槽可用，supervisor任务最多可以创建 `maxNumConcurrentSubTasks` worker任务, 因此，可以同时运行的任务总数为 `(maxNumConcurrentSubTasks+1)(包括supervisor任务)` 。请注意，这甚至可以大于任务槽的总数（所有worker的容量之和）。如果 `maxNumConcurrentSubTasks` 大于 `n（可用任务槽）` ，则 `maxNumConcurrentSubTasks` 任务由supervisor任务创建，但只有 `n` 个任务将启动, 其他人将在挂起状态下等待，直到任何正在运行的任务完成。

如果将并行索引任务与流摄取一起使用，我们建议**限制批摄取的最大容量**，以防止流摄取被批摄取阻止。假设您同时有 `t` 个并行索引任务要运行， 但是想将批摄取的最大任务数限制在 `b` 。 那么， 所有并行索引任务的 `maxNumConcurrentSubTasks` 之和 + `t` (supervisor任务数) 必须小于 `b` 。

如果某些任务的优先级高于其他任务，则可以将其 `maxNumConcurrentSubTasks` 设置为高于低优先级任务的值。这可能有助于高优先级任务比低优先级任务提前完成，方法是为它们分配更多的任务槽。

简单任务

简单任务（ `index` 类型）设计用于较小的数据集。任务在索引服务中执行。

任务符号

一个示例任务如下：

```
{
  "type" : "index",
  "spec" : {
    "dataSchema" : {
      "dataSource" : "wikipedia",
      "timestampSpec" : {
        "column" : "timestamp",
        "format" : "auto"
      },
      "dimensionsSpec" : {
        "dimensions": ["page","language","user","unpatrolled","newPage","robot
        "dimensionExclusions" : []
      },
      "metricsSpec" : [
        {
          "type" : "count",
          "name" : "count"
        },
        {
          "type" : "doubleSum",
          "name" : "added",
          "fieldName" : "added"
        },
        {
          "type" : "doubleSum",
          "name" : "deleted",
          "fieldName" : "deleted"
        },
        {
          "type" : "doubleSum",
          "name" : "delta",
          "fieldName" : "delta"
        }
      ],
      "granularitySpec" : {
        "type" : "uniform",
        "segmentGranularity" : "DAY",
        "queryGranularity" : "NONE",
        "intervals" : [ "2013-08-31/2013-09-01" ]
      }
    },
    "ioConfig" : {
      "type" : "index",
      "inputSource" : {
        "type" : "local",
        "baseDir" : "examples/indexing/",
        "filter" : "wikipedia_data.json"
       },
       "inputFormat": {
         "type": "json"
       }
    },
    "tuningConfig" : {
      "type" : "index",
      "maxRowsPerSegment" : 5000000,
      "maxRowsInMemory" : 1000000
    }
  }
}
```

属性	描述	是否必须
`type`	任务类型， 应该总是 `index`	是
`id`	任务ID。如果该值为显式的指定，Druid将会使用任务类型、数据源名称、时间间隔以及日期时间戳生成一个任务ID	否
`spec`	摄入规范，包括dataSchema、IOConfig 和 TuningConfig。 详情见下边的描述	是
`context`	包含多个任务配置参数的上下文。 详情见下边的描述	否

`dataSchema`

该字段为必须字段。

详情可以见摄取文档中的 `dataSchema` 部分。

如果没有在 `dataSchema` 的 `granularitySpec` 中显式指定 `intervals`，本地索引任务将对数据执行额外的传递，以确定启动时要锁定的范围。如果显式指定 `intervals`，则指定间隔之外的任何行都将被丢弃。如果您知道数据的时间范围，我们建议显式设置 `intervals`，因为它允许任务跳过额外的过程，并且如果有一些带有意外时间戳的杂散数据，您不会意外地替换该范围之外的数据。

`ioConfig`

属性	描述	默认值	是否必须
`type`	任务类型，应该总是 `index`	none	是
`inputFormat`	`inputFormat` 指定如何解析输入数据	none	是
`appendToExisting`	创建段作为最新版本的附加分片，有效地附加到段集而不是替换它。仅当现有段集具有可扩展类型 `shardSpec` 时，此操作才有效。	false	否

`tuningConfig`

tuningConfig是一个可选项，如果未指定则使用默认的参数。 详情如下：

属性	描述	
`type`	任务类型，应当总是 `index`	
`maxRowsPerSegment`	已废弃。使用 `partitionsSpec` 替代，被用来分片。 决定在每个段中有多少行。	
`maxRowsInMemory`	用于确定何时应该从中间层持久化到磁盘。通常用户不需要设置此值，但根据数据的性质，如果行的字节数较短，则用户可能不希望在内存中存储一百万行，应设置此值。	
`maxBytesInMemory`	用于确定何时应该从中间层持久化到磁盘。通常这是在内部计算的，用户不需要设置它。此值表示在持久化之前要在堆内存中聚合的字节数。这是基于对内存使用量的粗略估计，而不是实际使用量。用于索引的最大堆内存使用量为 `maxBytesInMemory * (2 + maxPendingResistent)`	
`maxTotalRows`	已废弃。使用 `partitionsSpec` 替代。等待推送的段中的总行数。用于确定何时应进行中间推送。	
`numShards`	已废弃。使用 `partitionsSpec` 替代。当使用 `hashed partitionsSpec` 时直接指定要创建的分片数。如果该值被指定了且在 `granularitySpec` 中指定了 `intervals` ，那么索引任务可以跳过确定间隔/分区传递数据。如果设置了 `maxRowsPerSegment` ，则无法指定 `numShards` 。	
`partitionsSpec`	定义在每个时间块中如何分区数据。 参见 partitionsSpec	
`indexSpec`	定义段在索引阶段的存储格式相关选项，参见 IndexSpec	

属性	描述	
`indexSpecForIntermediatePersists`	定义要在索引时用于中间持久化临时段的段存储格式选项。这可用于禁用中间段上的维度/度量压缩，以减少最终合并所需的内存。但是，在中间段上禁用压缩可能会增加页缓存的使用，而在它们被合并到发布的最终段之前使用它们，有关可能的值，请参阅 IndexSpec。	
`maxPendingPersists`	可挂起但未启动的最大持久化任务数。如果新的中间持久化将超过此限制，则在当前运行的持久化完成之前，摄取将被阻止。使用 `maxRowsInMemory * (2 + maxPendingResistents)` 索引扩展的最大堆内存使用量。	
`forceGuaranteedRollup`	强制保证 最佳Rollup。最佳rollup优化了生成的段的总大小和查询时间，同时索引时间将增加。如果设置为true，则必须设置 `granularitySpec` 中的 `intervals` ，同时必须对 `partitionsSpec` 使用 `single_dim` 或者 `hashed` 。此标志不能与 `IOConfig` 的 `appendToExisting` 一起使用。有关更多详细信息，请参见下面的 "分段推送模式" 部分。	
`reportParseExceptions`	已废弃。如果为true，则将引发解析期间遇到的异常并停止摄取；如果为false，则将跳过不可解析的行和字段。将 `reportParseExceptions` 设置为true将覆盖 `maxParseExceptions` 和 `maxSavedParseExceptions` 的现有配置，将 `maxParseExceptions` 设置为0并将 `maxSavedParseExceptions` 限制为不超过1。	
`pushTimeout`	段推送的超时毫秒时间。 该值必须设置为 >= 0, 0意味着永不超时	
`segmentWriteOutMediumFactory`	创建段时使用的段写入介质。 参见 segmentWriteOutMediumFactory	

属性	描述	
`logParseExceptions`	如果为true，则在发生解析异常时记录错误消息，其中包含有关发生错误的行的信息。	f
`maxParseExceptions`	任务停止接收并失败之前可能发生的最大分析异常数。如果设置了 `reportParseExceptions` ，则该配置被覆盖。	u
`maxSavedParseExceptions`	当出现解析异常时，Druid可以跟踪最新的解析异常。"maxSavedParseExceptions"限制将保存多少个异常实例。这些保存的异常将在任务完成报告中的任务完成后可用。如果设置了 `reportParseExceptions` ，则该配置被覆盖。	(

`partitionsSpec`

PartitionsSpec用于描述辅助分区方法。您应该根据需要的rollup模式使用不同的partitionsSpec。为了实现 最佳rollup，您应该使用 `hashed` （基于每行中维度的哈希进行分区）

属性	描述	默认值	是否必须
`type`	应该总是 `hashed`	none	是
`maxRowsPerSegment`	用在分片中，决定在每个段中有多少行	5000000	否
`numShards`	直接指定要创建的分片数。如果该值被指定了，同时在 `granularitySpec` 中指定了 `intervals` ，那么索引任务可以跳过确定通过数据的间隔/分区	null	是
`partitionDimensions`	要分区的维度。留空可选择所有维度。	null	否

对于尽可能rollup模式，您应该使用 `dynamic`

属性	描述	默认值	是否必须
`type`	应该总是 `dynamic`	none	是
`maxRowsPerSegment`	用来分片。决定在每一个段中有多少行	5000000	否
`maxTotalRows`	等待推送的所有段的总行数。用于确定中间段推送的发生时间。	20000000	否

`segmentWriteOutMediumFactory`

字段	类型	描述	是否必须
`type`	String	配置解释和可选项可以参见 额外的Peon配置：SegmentWriteOutMediumFactory	是

分段推送模式

当使用简单任务摄取数据时，它从输入数据创建段并推送它们。对于分段推送，索引任务支持两种分段推送模式，分别是*批量推送模式*和*增量推送模式*，以实现 最佳rollup和尽可能rollup。

在批量推送模式下，在索引任务的最末端推送每个段。在此之前，创建的段存储在运行索引任务的进程的内存和本地存储中。因此，此模式可能由于存储容量有限而导致问题，建议不要在生产中使用。

相反，在增量推送模式下，分段是增量推送的，即可以在索引任务的中间推送。更准确地说，索引任务收集数据并将创建的段存储在运行该任务的进程的内存和磁盘中，直到收集的行总数超过 `maxTotalRows` 。一旦超过，索引任务将立即推送创建的所有段，直到此时为止，清除所有推送的段，并继续接收剩余的数据。

要启用批量推送模式，应在 `TuningConfig` 中设置 `forceGuaranteedRollup` 。请注意，此选项不能与 `IOConfig` 的 `appendToExisting` 一起使用。

输入源

输入源是定义索引任务读取数据的位置。只有本地并行任务和简单任务支持输入源。

S3输入源

> [!WARNING] 您需要添加 `druid-s3-extensions` 扩展以便使用S3输入源。

S3输入源支持直接从S3读取对象。可以通过S3 URI字符串列表或S3位置前缀列表指定对象，该列表将尝试列出内容并摄取位置中包含的所有对象。S3输入源是可拆分的，可以由 并行任务 使用，其中 `index_parallel` 的每个worker任务将读取一个或多个对象。

样例规范：

```
...
    "ioConfig": {
      "type": "index_parallel",
      "inputSource": {
        "type": "s3",
        "uris": ["s3://foo/bar/file.json", "s3://bar/foo/file2.json"]
      },
      "inputFormat": {
        "type": "json"
      },
      ...
    },
...
```

```
...
    "ioConfig": {
      "type": "index_parallel",
      "inputSource": {
        "type": "s3",
        "prefixes": ["s3://foo/bar", "s3://bar/foo"]
      },
      "inputFormat": {
        "type": "json"
      },
      ...
    },
...
```

```
...
    "ioConfig": {
      "type": "index_parallel",
      "inputSource": {
        "type": "s3",
        "objects": [
          { "bucket": "foo", "path": "bar/file1.json"},
          { "bucket": "bar", "path": "foo/file2.json"}
        ]
      },
      "inputFormat": {
        "type": "json"
      },
      ...
    },
...
```

属性	描述	默认	是否必须
`type`	应该是 `s3`	None	是
`uris`	指定被摄取的S3对象位置的URI JSON数组	None	`uris` 或者 `prefixes` 或者 `objects` 必须被设置。
`prefixes`	指定被摄取的S3对象所在的路径前缀的URI JSON数组	None	`uris` 或者 `prefixes` 或者 `objects` 必须被设置。
`objects`	指定被摄取的S3对象的JSON数组	None	`uris` 或者 `prefixes` 或者 `objects` 必须被设置。
`properties`	指定用来覆盖默认S3配置的对象属性，详情见下边	None	否（未指定则使用默认）

注意：只有当 `prefixes` 被指定时，S3输入源将略过空的对象。

S3对象：

属性	描述	默认	是否必须
`bucket`	S3 Bucket的名称	None	是
`path`	数据路径	None	是

属性对象：

属性	描述	默认	是否必须
`accessKeyId`	S3输入源访问密钥的 Password Provider 或纯文本字符串	None	如果 `secretAccessKey` 被提供的话，则为必须
`secretAccessKey`	S3输入源访问密钥的 Password Provider 或纯文本字符串	None	如果 `accessKeyId` 被提供的话，则为必须

注意： *如果 `accessKeyId` 和 `secretAccessKey` 未被指定的话， 则将使用默认的 S3认证*

谷歌云存储输入源

> [!WARNING] 您需要添加 `druid-google-extensions` 扩展以便使用谷歌云存储输入源。

谷歌云存储输入源支持直接从谷歌云存储读取对象，可以通过谷歌云存储URI字符串列表指定对象。谷歌云存储输入源是可拆分的，可以由 并行任务 使用，其中 `index_parallel` 的每个worker任务将读取一个或多个对象。

样例规范：

```
...
    "ioConfig": {
      "type": "index_parallel",
      "inputSource": {
        "type": "google",
        "uris": ["gs://foo/bar/file.json", "gs://bar/foo/file2.json"]
      },
      "inputFormat": {
        "type": "json"
      },
      ...
    },
...
```

```
...
    "ioConfig": {
      "type": "index_parallel",
      "inputSource": {
        "type": "google",
        "prefixes": ["gs://foo/bar", "gs://bar/foo"]
      },
      "inputFormat": {
        "type": "json"
      },
      ...
    },
...
```

```
...
    "ioConfig": {
      "type": "index_parallel",
      "inputSource": {
        "type": "google",
        "objects": [
          { "bucket": "foo", "path": "bar/file1.json"},
          { "bucket": "bar", "path": "foo/file2.json"}
        ]
      },
      "inputFormat": {
        "type": "json"
      },
      ...
    },
...
```

属性	描述	默认	是否必须
`type`	应该是 `google`	None	是
`uris`	指定被摄取的谷歌云存储对象位置的URI JSON数组	None	`uris` 或者 `prefixes` 或者 `objects` 必须被设置。
`prefixes`	指定被摄取的谷歌云存储对象所在的路径前缀的URI JSON数组。 以被给定的前缀开头的空对象将被略过	None	`uris` 或者 `prefixes` 或者 `objects` 必须被设置。
`objects`	指定被摄取的谷歌云存储对象的JSON数组	None	`uris` 或者 `prefixes` 或者 `objects` 必须被设置。

注意：只有当 `prefixes` 被指定时，谷歌云存储输入源将略过空的对象。

谷歌云存储对象：

属性	描述	默认	是否必须
`bucket`	谷歌云存储 Bucket的名称	None	是
`path`	数据路径	None	是

Azure输入源

> [!WARNING] 您需要添加 `druid-azure-extensions` 扩展以便使用Azure输入源。

Azure输入源支持直接从Azure读取对象，可以通过Azure URI字符串列表指定对象。Azure输入源是可拆分的，可以由 并行任务 使用，其中 `index_parallel` 的每个worker任务将读取一个或多个对象。

样例规范：

```
...
    "ioConfig": {
      "type": "index_parallel",
      "inputSource": {
        "type": "azure",
        "uris": ["azure://container/prefix1/file.json", "azure://container/pre
      },
      "inputFormat": {
        "type": "json"
      },
      ...
    },
...
```

```
...
    "ioConfig": {
      "type": "index_parallel",
      "inputSource": {
        "type": "azure",
        "prefixes": ["azure://container/prefix1", "azure://container/prefix2"]
      },
      "inputFormat": {
        "type": "json"
      },
      ...
    },
...
```

```
...
    "ioConfig": {
      "type": "index_parallel",
      "inputSource": {
        "type": "azure",
        "objects": [
          { "bucket": "container", "path": "prefix1/file1.json"},
          { "bucket": "container", "path": "prefix2/file2.json"}
        ]
      },
      "inputFormat": {
        "type": "json"
      },
      ...
    },
...
```

属性	描述	默认	是否必须
`type`	应该是 `azure`	None	是
`uris`	指定被摄取的azure对象位置的URI JSON数组, 格式必须为 `azure://<container>/<path-to-file>`	None	`uris` 或者 `prefixes` 或者 `objects` 必须被设置。
`prefixes`	指定被摄取的azure对象所在的路径前缀的URI JSON数组, 格式必须为 `azure://<container>/<prefix>` , 以被给定的前缀开头的空对象将被略过	None	`uris` 或者 `prefixes` 或者 `objects` 必须被设置。
`objects`	指定被摄取的azure对象的JSON数组	None	`uris` 或者 `prefixes` 或者 `objects` 必须被设置。

注意：只有当 `prefixes` 被指定时，azure输入源将略过空的对象。

azure对象：

属性	描述	默认	是否必须
`bucket`	azure Bucket的名称	None	是
`path`	数据路径	None	是

HDFS输入源

> [!WARNING] 您需要添加 `druid-hdfs-extensions` 扩展以便使用HDFS输入源。

HDFS输入源支持直接从HDFS存储中读取文件，文件路径可以指定为HDFS URI字符串或者HDFS URI字符串列表。HDFS输入源是可拆分的，可以由 并行任务 使用，其中 `index_parallel` 的每个worker任务将读取一个或多个文件。

样例规范：

```
...
    "ioConfig": {
      "type": "index_parallel",
      "inputSource": {
        "type": "hdfs",
        "paths": "hdfs://foo/bar/", "hdfs://bar/foo"
      },
      "inputFormat": {
        "type": "json"
      },
      ...
    },
...
```

```
...
    "ioConfig": {
      "type": "index_parallel",
      "inputSource": {
        "type": "hdfs",
        "paths": ["hdfs://foo/bar", "hdfs://bar/foo"]
      },
      "inputFormat": {
        "type": "json"
      },
      ...
    },
...
```

```
...
    "ioConfig": {
      "type": "index_parallel",
      "inputSource": {
        "type": "hdfs",
        "paths": "hdfs://foo/bar/file.json", "hdfs://bar/foo/file2.json"
      },
      "inputFormat": {
        "type": "json"
      },
      ...
    },
...
```

```
...
    "ioConfig": {
      "type": "index_parallel",
      "inputSource": {
        "type": "hdfs",
        "paths": ["hdfs://foo/bar/file.json", "hdfs://bar/foo/file2.json"]
      },
      "inputFormat": {
        "type": "json"
      },
      ...
    },
...
```

属性	描述	默认	是否必须
`type`	应该总是 `hdfs`	None	是
`paths`	HDFS路径。可以是JSON数组或逗号分隔的路径字符串，这些路径支持类似*的通配符。给定路径之下的空文件将会被跳过。	None	是

您还可以使用HDFS输入源从云存储摄取数据。但是，如果您想从AWS S3或谷歌云存储读取数据，可以考虑使用 S3输入源 或 谷歌云存储输入源。

HTTP输入源

HTTP输入源支持直接通过HTTP从远程站点直接读取文件。 HTTP输入源是可拆分的，可以由 并行任务 使用，其中 `index_parallel` 的每个worker任务只能读取一个文件。

样例规范：

```
...
    "ioConfig": {
      "type": "index_parallel",
      "inputSource": {
        "type": "http",
        "uris": ["http://example.com/uri1", "http://example2.com/uri2"]
      },
      "inputFormat": {
        "type": "json"
      },
      ...
    },
...
```

使用DefaultPassword Provider的身份验证字段示例（这要求密码位于摄取规范中）：

```
...
    "ioConfig": {
      "type": "index_parallel",
      "inputSource": {
        "type": "http",
        "uris": ["http://example.com/uri1", "http://example2.com/uri2"],
        "httpAuthenticationUsername": "username",
        "httpAuthenticationPassword": "password123"
      },
      "inputFormat": {
        "type": "json"
      },
      ...
    },
...
```

您还可以使用其他现有的Druid PasswordProvider。下面是使用EnvironmentVariablePasswordProvider的示例：

```
...
    "ioConfig": {
      "type": "index_parallel",
      "inputSource": {
        "type": "http",
        "uris": ["http://example.com/uri1", "http://example2.com/uri2"],
        "httpAuthenticationUsername": "username",
        "httpAuthenticationPassword": {
          "type": "environment",
          "variable": "HTTP_INPUT_SOURCE_PW"
        }
      },
      "inputFormat": {
        "type": "json"
      },
      ...
    },
...
}
```

属性	描述	默认	是否必须
`type`	应该是 `http`	None	是
`uris`	输入文件的uris	None	是
`httpAuthenticationUsername`	用于指定uri的身份验证的用户名。如果规范中指定的uri需要基本身份验证头，则改属性是可选的。	None	否
`httpAuthenticationPassword`	用于指定uri的身份验证的密码。如果规范中指定的uri需要基本身份验证头，则改属性是可选的。	None	否

Inline输入源

Inline输入源可用于读取其规范内联的数据。它可用于演示或用于快速测试数据解析和schema。

样例规范：

```
...
    "ioConfig": {
      "type": "index_parallel",
      "inputSource": {
        "type": "inline",
        "data": "0,values,formatted\n1,as,CSV"
      },
      "inputFormat": {
        "type": "csv"
      },
      ...
    },
...
```

属性	描述	是否必须
`type`	应该是 `inline`	是
`data`	要摄入的内联数据	是

Local输入源

Local输入源支持直接从本地存储中读取文件，主要目的用于PoC测试。 Local输入源是可拆分的，可以由 并行任务 使用，其中 `index_parallel` 的每个worker任务读取一个或者多个文件。

样例规范：

```
...
    "ioConfig": {
      "type": "index_parallel",
      "inputSource": {
        "type": "local",
        "filter" : "*.csv",
        "baseDir": "/data/directory",
        "files": ["/bar/foo", "/foo/bar"]
      },
      "inputFormat": {
        "type": "csv"
      },
      ...
    },
...
```

属性	描述	是否必须
`type`	应该是 `local`	是
`filter`	文件的通配符筛选器, 详细信息 点击此处 查看	如果 `baseDir` 指定了，则为必须
`baseDir`	递归搜索要接收的文件的目录, 将跳过 `baseDir` 下的空文件。	`baseDir` 或者 `files` 至少需要被指定一个
`files`	要摄取的文件路径。如果某些文件位于指定的 `baseDir` 下，则可以忽略它们以避免摄取重复文件。该选项会跳过空文件。	`baseDir` 或者 `files` 至少需要被指定一个

Druid输入源

Druid输入源支持直接从现有的Druid段读取数据，可能使用新的模式，并更改段的名称、维度、Metrics、Rollup等。Druid输入源是可拆分的，可以由 并行任务 使用。这个输入源有一个固定的从Druid段读取的输入格式；当使用这个输入源时，不需要在摄取规范中指定输入格式字段。

属性	描述	是否必须
`type`	应该是 `druid`	是
`dataSource`	定义要从中获取行的Druid数据源	是
`interval`	ISO-8601时间间隔的字符串，它定义了获取数据的时间范围。	是
`dimensions`	包含要从Druid数据源中选择的维度列名称的字符串列表。如果列表为空，则不返回维度。如果为空，则返回所有维度。	否
`metrics`	包含要选择的Metric列名称的字符串列表。如果列表为空，则不返回任何度量。如果为空，则返回所有Metric。	否
`filter`	详情请查看 filters 如果指定，则只返回与筛选器匹配的行。	否

DruidInputSource规范的最小示例如下所示：

```
...
    "ioConfig": {
      "type": "index_parallel",
      "inputSource": {
        "type": "druid",
        "dataSource": "wikipedia",
        "interval": "2013-01-01/2013-01-02"
      }
      ...
    },
...
```

上面的规范将从 `wikipedia` 数据源中读取所有现有dimension和metric列，包括 `2013-01-01/2013-01-02` 时间间隔内带有时间戳（ `__time` 列）的所有行。

以下规范使用了筛选器并读取原始数据源列子集：

```
...
    "ioConfig": {
      "type": "index_parallel",
      "inputSource": {
        "type": "druid",
        "dataSource": "wikipedia",
        "interval": "2013-01-01/2013-01-02",
        "dimensions": [
          "page",
          "user"
        ],
        "metrics": [
          "added"
        ],
        "filter": {
          "type": "selector",
          "dimension": "page",
          "value": "Druid"
        }
      }
      ...
    },
...
```

上面的规范只返回 `page` 、 `user` 维度和 `added` 的Metric。只返回 `page` = `Druid` 的行。

Firehoses(已废弃)

StaticS3Firehose

HDFSFirehose

LocalFirehose

HttpFirehose

IngestSegmentFirehose

SqlFirehose

InlineFirehose

CombiningFirehose

基于Hadoop的摄入

Apache Druid当前支持通过一个Hadoop摄取任务来支持基于Apache Hadoop的批量索引任务， 这些任务被提交到 Druid Overlord的一个运行实例上。详情可以查看 基于Hadoop的摄取vs基于本地批摄取的对比 来了解基于Hadoop的摄取、本地简单批摄取、本地并行摄取三者的比较。

运行一个基于Hadoop的批量摄取任务，首先需要编写一个如下的摄取规范， 然后提交到Overlord的 `druid/indexer/v1/task` 接口，或者使用Druid软件包中自带的 `bin/post-index-task` 脚本。

教程

本章包括了基于Hadoop摄取的参考文档，对于粗略的查看，可以查看 从Hadoop加载数据 教程。

任务符号

以下为一个示例任务：

```
{
  "type" : "index_hadoop",
  "spec" : {
    "dataSchema" : {
      "dataSource" : "wikipedia",
      "parser" : {
        "type" : "hadoopyString",
        "parseSpec" : {
          "format" : "json",
          "timestampSpec" : {
            "column" : "timestamp",
            "format" : "auto"
          },
          "dimensionsSpec" : {
            "dimensions": ["page","language","user","unpatrolled","newPage","r
            "dimensionExclusions" : [],
            "spatialDimensions" : []
          }
        }
      },
      "metricsSpec" : [
        {
          "type" : "count",
          "name" : "count"
        },
        {
          "type" : "doubleSum",
          "name" : "added",
          "fieldName" : "added"
        },
        {
          "type" : "doubleSum",
          "name" : "deleted",
          "fieldName" : "deleted"
        },
        {
          "type" : "doubleSum",
          "name" : "delta",
          "fieldName" : "delta"
        }
      ],
      "granularitySpec" : {
        "type" : "uniform",
        "segmentGranularity" : "DAY",
        "queryGranularity" : "NONE",
        "intervals" : [ "2013-08-31/2013-09-01" ]
      }
    },
    "ioConfig" : {
      "type" : "hadoop",
      "inputSpec" : {
        "type" : "static",
        "paths" : "/MyDirectory/example/wikipedia_data.json"
      }
    },
    "tuningConfig" : {
      "type": "hadoop"
    }
  },
  "hadoopDependencyCoordinates": <my_hadoop_version>
}
```

属性	描述
`type`	任务类型，应该总是 `index_hadoop`
`spec`	Hadoop索引任务规范。 详见 ingestion
`hadoopDependencyCoordinates`	Druid使用的Hadoop依赖，这些属性会覆盖默认的Hadoop依赖。 如果该值被指定，Druid将在 `druid.extensions.hadoopDependenciesDir` 目录下查找指定的Hadoop依赖
`classpathPrefix`	为Peon进程准备的类路径。

还要注意，Druid会自动计算在Hadoop集群中运行的Hadoop作业容器的类路径。但是，如果Hadoop和Druid的依赖项之间发生冲突，可以通过设置 `druid.extensions.hadoopContainerDruidClasspath` 属性。请参阅 基本druid配置中的扩展配置 。

dataSchema

该字段是必须的。 详情可以查看摄取页中的 `dataSchema` 部分来看它应该包括哪些部分。

ioConfig

该字段是必须的。

字段	类型	描述	是否必须
`type`	String	应该总是 `hadoop`	是
`inputSpec`	Object	指定从哪里拉数据。详情见以下。	是
`segmentOutputPath`	String	将段转储到的路径	仅仅在 命令行Hadoop索引 中使用， 否则该字段必须为null
`metadataUpdateSpec`	Object	关于如何更新这些段所属的druid集群的元数据的规范	仅仅在 命令行Hadoop索引 中使用， 否则该字段必须为null

inputSpec

有多种类型的inputSec：

static

一种 `inputSpec` 的类型，该类型提供数据文件的静态路径。

字段	类型	描述
`inputFormat`	String	指定要使用的Hadoop输入格式的类，比如 `org.apache.hadoop.mapreduce.lib.input.SequenceFi`
`paths`	String 数组	标识原始数据位置的输入路径的字符串

例如，以下例子使用了静态输入路径：

```
"paths" : "hdfs://path/to/data/is/here/data.gz,hdfs://path/to/data/is/here/mor
```

也可以从云存储直接读取数据，例如AWS S3或者谷歌云存储。 前提是需要首先的所有Druid *MiddleManager进程或者Indexer进程*的类路径下安装必要的依赖库。对于S3，需要通过以下命令来安装 Hadoop AWS 模块

```
java -classpath "${DRUID_HOME}lib/*" org.apache.druid.cli.Main tools pull-deps
cp ${DRUID_HOME}/hadoop-dependencies/hadoop-aws/${HADOOP_VERSION}/hadoop-aws-$
```

一旦在所有的MiddleManager和Indexer进程中安装了Hadoop AWS模块，即可将S3路径放到 `inputSpec` 中，同时需要有任务属性。 对于更多配置，可以查看 Hadoop AWS 模块

```
"paths" : "s3a://billy-bucket/the/data/is/here/data.gz,s3a://billy-bucket/the/
```

```
"jobProperties" : {
  "fs.s3a.impl" : "org.apache.hadoop.fs.s3a.S3AFileSystem",
  "fs.AbstractFileSystem.s3a.impl" : "org.apache.hadoop.fs.s3a.S3A",
  "fs.s3a.access.key" : "YOUR_ACCESS_KEY",
  "fs.s3a.secret.key" : "YOUR_SECRET_KEY"
}
```

对于谷歌云存储，需要将 GCS connector jar 安装到*所有MiddleManager或者Indexer进程*的 `${DRUID_HOME}/hadoop-dependencies` 。 一旦在所有的MiddleManager和Indexer进程中安装了GCS连接器jar包，即可将谷歌云存储路径放到 `inputSpec` 中，同时需要有任务属性。对于更多配置，可以查看 instructions to configure Hadoop, GCS core default 和 GCS core template.

```
"paths" : "gs://billy-bucket/the/data/is/here/data.gz,gs://billy-bucket/the/da
```

```
"jobProperties" : {
  "fs.gs.impl" : "com.google.cloud.hadoop.fs.gcs.GoogleHadoopFileSystem",
  "fs.AbstractFileSystem.gs.impl" : "com.google.cloud.hadoop.fs.gcs.GoogleHado
}
```

`granularity`

一种 `inputSpec` 类型，该类型期望数据已经按照日期时间组织到对应的目录中，路径格式为： `y=XXXX/m=XX/d=XX/H=XX/M=XX/S=XX` (其中日期用小写表示，时间用大写表示)。

字段	类型	描述
`dataGranularity`	String	指定期望的数据粒度，例如，hour意味着期望 `y=XXXX/m=XX/d=XX/H=XX`
`inputFormat`	String	指定要使用的Hadoop输入格式的类，比如 `org.apache.hadoop.mapreduce.lib.input.Sequen`
`inputPath`	String	要将日期时间路径附加到的基路径。
`filePattern`	String	要包含的文件应匹配的模式
`pathFormat`	String	每个目录的Joda datetime目录。 默认值为： `"'y'=yyyy/'m'=MM/'d'=dd/'H'=HH"` ,详情可以看

例如， 如果示例配置具有 2012-06-01/2012-06-02 时间间隔，则数据期望的路径是：

```
s3n://billy-bucket/the/data/is/here/y=2012/m=06/d=01/H=00
s3n://billy-bucket/the/data/is/here/y=2012/m=06/d=01/H=01
...
s3n://billy-bucket/the/data/is/here/y=2012/m=06/d=01/H=23
```

`dataSource`

一种 `inputSpec` 的类型, 该类型读取已经存储在Druid中的数据。 该类型被用来"re-indexing"(重新索引)数据和下边描述 `multi` 类型 `inputSpec` 的 "delta-ingestion"(增量摄取)。

字段	类型	描述	是否必须
`type`	String	应该总是 `dataSource`	是
`ingestionSpec`	JSON对象	要加载的Druid段的规范。详情见下边内容。	是
`maxSplitSize`	Number	允许根据段的大小将多个段合并为单个Hadoop InputSplit。使用-1，druid根据用户指定的映射任务数计算最大拆分大小(`mapred.map.tasks` 或者 `mapreduce.job.maps`). 默认情况下，对一个段进行一次拆分。 `maxSplitSize` 以字节为单位指定。	否
`useNewAggs`	Boolean	如果"false"，则hadoop索引任务的"metricsSpec"中的聚合器列表必须与接收原始数据时在原始索引任务中使用的聚合器列表相同。默认值为"false"。当"inputSpec"类型为"dataSource"而不是"multi"时，可以将此字段设置为"true"，以便在重新编制索引时启用任意聚合器。请参阅下面的"multi"类型增量摄取支持。	否

下表中为 `ingestionSpec` 中的一些选项：

字段	类型	描述
`dataSource`	String	Druid数据源名称，从该数据源读取数据
`intervals`	List	ISO-8601时间间隔的字符串List
`segments`	List	从中读取数据的段的列表，默认情况下以通过向Coordinator的接口 `/druid/Coordinator/v1/metadata/dataso full` 进行POST查询来获取要放在这里如["2012-01-01T00:00:00.000/2012-01 03T00:00:00.000"，"2012-01-05T00:0 01-07T00:00:00.000"]. 您可能希望手动以确保读取的段与任务提交时的段完全提供的列表与任务实际运行时的数据库任务将失败
`filter`	JSON	查看 Filter
`dimensions`	String 数组	要加载的维度列的名称。默认情况下，`parseSpec` 构造。如果 `parseSpec` 没有表，则将读取存储数据中的所有维度列
`metrics`	String 数组	要加载的Metric列的名称。默认情况下，有已配置聚合器的"name"构造。
`ignoreWhenNoSegments`	boolean	如果找不到段，是否忽略此 `ingestionS` 为是在找不到段时引发错误。

示例：

```
"ioConfig" : {
  "type" : "hadoop",
  "inputSpec" : {
    "type" : "dataSource",
    "ingestionSpec" : {
      "dataSource": "wikipedia",
      "intervals": ["2014-10-20T00:00:00Z/P2W"]
    }
  },
  ...
}
```

`multi`

这是一个组合类型的 `inputSpec`，来组合其他 `inputSpec`。此inputSpec用于增量接收。您还可以使用一个 `multi` 类型的inputSpec组合来自多个数据源的数据。但是，每个特定的数据源只能指定一次。注意，"useNewAggs"必须设置为默认值false以支持增量摄取。

字段	类型	描述	是否必须
`children`	JSON对象数组	一个JSON对象List，里边包含了其他类型的inputSpec	是

示例：

```
"ioConfig" : {
  "type" : "hadoop",
  "inputSpec" : {
    "type" : "multi",
    "children": [
      {
        "type" : "dataSource",
        "ingestionSpec" : {
          "dataSource": "wikipedia",
          "intervals": ["2012-01-01T00:00:00.000/2012-01-03T00:00:00.000", "20
          "segments": [
            {
              "dataSource": "test1",
              "interval": "2012-01-01T00:00:00.000/2012-01-03T00:00:00.000",
              "version": "v2",
              "loadSpec": {
                "type": "local",
                "path": "/tmp/index1.zip"
              },
              "dimensions": "host",
              "metrics": "visited_sum,unique_hosts",
              "shardSpec": {
                "type": "none"
              },
              "binaryVersion": 9,
              "size": 2,
              "identifier": "test1_2000-01-01T00:00:00.000Z_3000-01-01T00:00:0
            }
          ]
        }
      },
      {
        "type" : "static",
        "paths": "/path/to/more/wikipedia/data/"
      }
    ]
  },
  ...
}
```

强烈建议显式地在 `dataSource` 中的 `inputSpec` 中提供段列表，以便增量摄取任务是幂等的。您可以通过对Coordinator进行以下调用来获取该段列表，POST `/druid/coordinator/v1/metadata/datasources/{dataSourceName}/segments?full`，请求体：[interval1，interval2，…]， 例如["2012-01-01T00:00:00.000/2012-01-03T00:00:00.000"，"2012-01-05T00:00:00.000/2012-01-07T00:00:00.000"]

tuningConfig

`tuningConfig` 是一个可选项，如果未指定的话，则使用默认的参数。

字段	类型	描述
`workingPath`	String	用于存储中间结果（Hado 结果）的工作路径
`version`	String	创建的段的版本。 对于Ha 一般是忽略的，除非 `useExplicitVersion` 被设
`partitionsSpec`	Object	指定如何将时间块内的分区 属性意味着不会发生分区。 `partitionsSpec`
`maxRowsInMemory`	Integer	在持久化之前在堆内存中聚 意：由于rollup操作，该值 数，可能不等于输入的行数 来管理需要的JVM堆内存大 下，用户并不需要设置该值 据自身。 如果数据是非常小 望在内存存储上百万行数据 设置该值。
`maxBytesInMemory`	Long	在持久化之前在堆内存中聚 通常这是在内部计算的，用 它。此值表示在持久化之前 聚合的字节数。这是基于对 粗略估计，而不是实际使用 的最大堆内存使用量为 `ma` `* (2 + maxPendingResistent`
`leaveIntermediate`	Boolean	作业完成时，不管通过还是 作路径中留下中间文件（用
`cleanupOnFailure`	Boolean	当任务失败时清理中间文件 `leaveIntermediate` 设置为
`overwriteFiles`	Boolean	在索引过程中覆盖找到的现
`ignoreInvalidRows`	Boolean	**已废弃**。忽略发现有问题的 false，解析过程中遇到的任 引发并停止摄取；如果为tr 可解析的行和字段。如果定 `maxParseExceptions` ，则忽
`combineText`	Boolean	使用CombineTextInputFor 件合并为一个文件拆分。这 量小文件时加快Hadoop作

字段	类型	描述
`useCombiner`	Boolean	如果可能的话，使用Hado 在mapper阶段合并行
`jobProperties`	Object	增加到Hadoop作业配置的 情见下边。
`indexSpec`	Object	调整数据如何被索引。 详 位于摄取页的 `indexSpec`
`indexSpecForIntermediatePersists`	Object	定义要在索引时用于中间持 段存储格式选项。这可用于 的dimension/metric压缩， 并所需的内存。但是，在中 缩可能会增加页缓存的使用 它们被合并到发布的最终段 们，有关可能的值，请参阅 `indexSpec` 。
`numBackgroundPersistThreads`	Integer	用于增量持久化的新后台线 功能会显著增加内存压力和 率，但会使任务更快完成。 0（对持久性使用当前线程 将其设置为1。
`forceExtendableShardSpecs`	Boolean	强制使用可扩展的shardSp 的分区总是使用可扩展的s 于单维分区，此选项应设置 可扩展shardSpec。对于分 区规范
`useExplicitVersion`	Boolean	强制HadoopIndexTask使用
`logParseExceptions`	Boolean	如果为true，则在发生解析 误消息，其中包含有关发生 息。
`maxParseExceptions`	Integer	任务停止接收并失败之前可 分析异常数。如果设置 了 `reportParseExceptions` ， 覆盖。
`useYarnRMJobStatusFallback`	Boolean	如果索引任务创建的Hado JobHistory服务器检索其完 此参数为true，则索引任务 `http://<varn rm address>/ws/v1/cluster/app id>` 获取应用程序状态，其 `address>` 是Hadoop配置中 `yarn.resourcemanager.weba` 地址。此标志用于索引任务 JobHistory服务器不可用的 退，从而导致索引任务失败 确定作业状态。

jobProperties

```
  "tuningConfig" : {
    "type": "hadoop",
    "jobProperties": {
      "<hadoop-property-a>": "<value-a>",
      "<hadoop-property-b>": "<value-b>"
    }
  }
```

Hadoop的 MapReduce文档 列出来了所有可能的配置参数。

在一些Hadoop分布式环境中，可能需要设置 `mapreduce.job.classpath` 或者 `mapreduce.job.user.classpath.first` 来避免类加载相关的问题。 更多详细信息可以参见 使用不同Hadoop版本的文档

partitionsSpec

段总是基于时间戳进行分区（根据 `granularitySpec` ），并且可以根据分区类型以其他方式进一步分区。Druid支持两种类型的分区策略： `hashed` （基于每行中所有维度的hash）和 `single_dim` （基于单个维度的范围）。

在大多数情况下，建议使用哈希分区，因为相对于单一维度分区，哈希分区将提高索引性能并创建更统一大小的数据段。

基于哈希的分区

```
  "partitionsSpec": {
    "type": "hashed",
    "targetRowsPerSegment": 5000000
  }
```

哈希分区的工作原理是首先选择多个段，然后根据每一行中所有维度的哈希对这些段中的行进行分区。段的数量是根据输入集的基数和目标分区大小自动确定的。

配置项为：

字段	描述	是否必须
`type`	使用的partitionsSpec的类型	"hashed"
`targetRowsPerSegment`	要包含在分区中的目标行数，应为500MB~1GB段的数。如果未设置 `numShards` ，则默认为5000000。	为该配置或者 `numShards`
`targetPartitionSize`	已弃用。重命名为 `targetRowsPerSegment` 。要包含在分区中的目标行数，应为500MB~1GB段的数。	为该配置或者 `numShards`
`maxRowsPerSegment`	已弃用。重命名为 `targetRowsPerSegment` 。要包含在分区中的目标行数，应为500MB~1GB段的数。	为该配置或者 `numShards`
`numShards`	直接指定分区数，而不是目标分区大小。摄取将运行得更快，因为它可以跳过自动选择多个分区所需的步骤。	为该配置或者 `maxRowsPerSegment`
`partitionDimensions`	要划分的维度。留空可选择所有维度。仅与 `numShard` 一起使用，在设置 `targetRowsPerSegment` 时将被忽略。	否

单一维度范围分区

```
"partitionsSpec": {
   "type": "single_dim",
   "targetRowsPerSegment": 5000000
 }
```

单一维度范围分区的工作原理是首先选择要分区的维度，然后将该维度分隔成连续的范围，每个段将包含该维度值在该范围内的所有行。例如，可以在维度"host"上对段进行分区,范围为"a.example.com"到"f.example.com"和"f.example.com"到"z.example.com"。 默认情况下，将自动确定要使用的维度，但可以使用特定维度替代它。

配置项为：

字段	描述	是否必须
`type`	使用的partitionsSpec的类型	"single_dim"
`targetRowsPerSegment`	要包含在分区中的目标行数，应为500MB~1GB段的数。	是
`targetPartitionSize`	已弃用。重命名为 `targetRowsPerSegment` 。要包含在分区中的目标行数，应为500MB~1GB段的数。	否
`maxRowsPerSegment`	要包含在分区中的最大行数。默认值为比 `targetRowsPerSegment` 大50%。	否
`maxPartitionSize`	已弃用。请改用 `maxRowsPerSegment` 。要包含在分区中的最大行数, 默认为比 `targetPartitionSize` 大50%。	否
`partitionDimension`	要分区的维度。留空可自动选择维度。	否
`assumeGrouped`	假设输入数据已经按时间和维度分组。摄取将运行得更快，但如果违反此假设，则可能会选择次优分区。	否

远程Hadoop集群

如果已经有了一个远程的Hadoop集群，确保在Druid的 `_common` 配置目录中包含 `*.xml` 文件。

如果Hadoop与Druid的版本存在依赖等问题，请查看 这些文档

Elastic MapReduce

如果集群运行在AWS上，可以使用Elastic MapReduce(EMR)来从S3中索引数据。需要以下几步：

- 创建一个 持续运行的集群
- 创建集群时，请输入以下配置。如果使用向导，则应在"编辑软件设置"下处于高级模式：

```
classification=yarn-site,properties=[mapreduce.reduce.memory.mb=6144,mapreduce
```

- 按照 Hadoop连接配置 指导，使用EMR master中 `/etc/hadoop/conf` 的XML文件。

Kerberized Hadoop集群

默认情况下，druid可以使用本地kerberos密钥缓存中现有的TGT kerberos票证。虽然TGT票证的生命周期有限，但您需要定期调用 `kinit` 命令以确保TGT票证的有效性。为了避免这个额外的外部cron作业脚本周期性地调用 `kinit` ，您可以提供主体名称和keytab位置，druid将在启动和作业启动时透明地执行身份验证。

属性	可能的值
`druid.hadoop.security.kerberos.principal`	`druid@EXAMPLE.COM`
`druid.hadoop.security.kerberos.keytab`	`/etc/security/keytabs/druid.he`

从具有EMR的S3加载

- 在Hadoop索引任务中 `tuningConfig` 部分的 `jobProperties` 字段中添加一下内容：

```
"jobProperties" : {
   "fs.s3.awsAccessKeyId" : "YOUR_ACCESS_KEY",
   "fs.s3.awsSecretAccessKey" : "YOUR_SECRET_KEY",
   "fs.s3.impl" : "org.apache.hadoop.fs.s3native.NativeS3FileSystem",
   "fs.s3n.awsAccessKeyId" : "YOUR_ACCESS_KEY",
   "fs.s3n.awsSecretAccessKey" : "YOUR_SECRET_KEY",
   "fs.s3n.impl" : "org.apache.hadoop.fs.s3native.NativeS3FileSystem",
   "io.compression.codecs" : "org.apache.hadoop.io.compress.GzipCodec,org.apac
}
```

注意，此方法使用Hadoop的内置S3文件系统，而不是Amazon的EMRFS，并且与Amazon的特定功能（如S3加密和一致视图）不兼容。如果您需要使用这些特性，那么您将需要通过 其他Hadoop发行版 一节中描述的机制之一，使Amazon EMR Hadoop JARs对Druid可用。

使用其他的Hadoop

Druid在许多Hadoop发行版中都是开箱即用的。

如果Druid与您当前使用的Hadoop版本发生依赖冲突时，您可以尝试在 Druid用户组 中搜索解决方案， 或者阅读 Druid不同版本Hadoop文档

命令行版本

运行：

```
java -Xmx256m -Duser.timezone=UTC -Dfile.encoding=UTF-8 -classpath lib/*:<hado
```

可选项

- "--coordinate" - 提供要使用的Apache Hadoop版本。此属性将覆盖默认的Hadoop。一旦指定，Apache Druid将从 `druid.extensions.hadoopDependenciesDir` 位置寻找Hadoop依赖。
- "--no-default-hadoop" - 不要下拉默认的hadoop版本

规范文件

spec文件需要包含一个JSON对象，其中的内容与Hadoop索引任务中的"spec"字段相同。有关规范格式的详细信息，请参见 Hadoop批处理摄取。

另外， `metadataUpdateSpec` 和 `segmentOutputPath` 字段需要被添加到ioConfig中：

```
      "ioConfig" : {
        ...
        "metadataUpdateSpec" : {
          "type":"mysql",
          "connectURI" : "jdbc:mysql://localhost:3306/druid",
          "password" : "druid",
          "segmentTable" : "druid_segments",
          "user" : "druid"
        },
        "segmentOutputPath" : "/MyDirectory/data/index/output"
      },
```

同时， `workingPath` 字段需要被添加到tuningConfig：

```
  "tuningConfig" : {
   ...
    "workingPath": "/tmp",
    ...
  }
```

Metadata Update Job Spec

这是一个属性规范，告诉作业如何更新元数据，以便Druid集群能够看到输出段并加载它们。

字段	类型	描述	是否必须
`type`	String	"metadata"是唯一可用的值	是
`connectURI`	String	连接元数据存储的可用的JDBC	是
`user`	String	DB的用户名	是
`password`	String	DB的密码	是
`segmentTable`	String	DB中使用的表	是

这些属性应该模仿您为 Coordinator 配置的内容。

segmentOutputPath配置

字段	类型	描述	是否必须
`segmentOutputPath`	String	将段转储到的路径	是

workingPath配置

字段	类型	描述	是否必须
`workingPath`	String	用于中间结果（Hadoop作业之间的结果）的工作路径。	否（默认为 `/tmp/druid-indexing` ）

请注意，命令行Hadoop indexer不具备索引服务的锁定功能，因此如果选择使用它，则必须注意不要覆盖由实时处理创建的段（如果设置了实时管道）。

任务参考文档

任务在Druid中完成所有与 摄取 相关的工作。

对于批量摄取，通常使用 任务api 直接将任务提交给Druid。对于流式接收，任务通常被提交给supervisor。

任务API

任务API主要在两个地方是可用的：

- Overlord 进程提供HTTP API接口来进行提交任务、取消任务、检查任务状态、查看任务日志与报告等。 查看 任务API文档 可以看到完整列表
- Druid SQL包括了一个 `sys.tasks` ，保存了当前任务运行的信息。 此表是只读的，并且可以通过Overlord API查询完整信息的有限制的子集。

任务报告

报告包含已完成的任务和正在运行的任务中有关接收的行数和发生的任何分析异常的信息的报表。

报告功能支持 简单的本地批处理任务、Hadoop批处理任务以及Kafka和Kinesis摄取任务支持报告功能。

任务结束报告

任务运行完成后，一个完整的报告可以在以下接口获取：

```
http://<OVERLORD-HOST>:<OVERLORD-PORT>/druid/indexer/v1/task/<task-id>/reports
```

一个示例输出如下：

```
{
  "ingestionStatsAndErrors": {
    "taskId": "compact_twitter_2018-09-24T18:24:23.920Z",
    "payload": {
      "ingestionState": "COMPLETED",
      "unparseableEvents": {},
      "rowStats": {
        "determinePartitions": {
          "processed": 0,
          "processedWithError": 0,
          "thrownAway": 0,
          "unparseable": 0
        },
        "buildSegments": {
          "processed": 5390324,
          "processedWithError": 0,
          "thrownAway": 0,
          "unparseable": 0
        }
      },
      "errorMsg": null
    },
    "type": "ingestionStatsAndErrors"
  }
}
```


任务运行报告

当一个任务正在运行时， 任务运行报告可以通过以下接口获得，包括摄取状态、未解析事件和过去1分钟、5分钟、15分钟内处理的平均事件数。

```
http://<OVERLORD-HOST>:<OVERLORD-PORT>/druid/indexer/v1/task/<task-id>/reports
```

和

```
http://<middlemanager-host>:<worker-port>/druid/worker/v1/chat/<task-id>/liveR
```

一个示例输出如下：

```
{
  "ingestionStatsAndErrors": {
    "taskId": "compact_twitter_2018-09-24T18:24:23.920Z",
    "payload": {
      "ingestionState": "RUNNING",
      "unparseableEvents": {},
      "rowStats": {
        "movingAverages": {
          "buildSegments": {
            "5m": {
              "processed": 3.392158326408501,
              "unparseable": 0,
              "thrownAway": 0,
              "processedWithError": 0
            },
            "15m": {
              "processed": 1.736165476881023,
              "unparseable": 0,
              "thrownAway": 0,
              "processedWithError": 0
            },
            "1m": {
              "processed": 4.206417693750045,
              "unparseable": 0,
              "thrownAway": 0,
              "processedWithError": 0
            }
          }
        },
        "totals": {
          "buildSegments": {
            "processed": 1994,
            "processedWithError": 0,
            "thrownAway": 0,
            "unparseable": 0
          }
        }
      },
      "errorMsg": null
    },
    "type": "ingestionStatsAndErrors"
  }
}
```


字段的描述信息如下：

`ingestionStatsAndErrors` 提供了行数和错误数的信息

`ingestionState` 标识了摄取任务当前达到了哪一步，可能的取值包括：

- `NOT_STARTED`：任务还没有读取任何行
- `DETERMINE_PARTITIONS`：任务正在处理行来决定分区信息
- `BUILD_SEGMENTS`：任务正在处理行来构建段
- `COMPLETED`：任务已经完成

只有批处理任务具有 `DETERMINE_PARTITIONS` 阶段。实时任务（如由Kafka索引服务创建的任务）没有 `DETERMINE_PARTITIONS` 阶段。

`unparseableEvents` 包含由不可解析输入引起的异常消息列表。这有助于识别有问题的输入行。对于 `DETERMINE_PARTITIONS` 和 `BUILD_SEGMENTS` 阶段，每个阶段都有一个列表。请注意，Hadoop批处理任务不支持保存不可解析事件。

`rowStats` map包含有关行计数的信息。每个摄取阶段有一个条目。不同行计数的定义如下所示：

- `processed`：成功摄入且没有报错的行数
- `processedWithErro`：摄取但在一列或多列中包含解析错误的行数。这通常发生在输入行具有可解析的结构但列的类型无效的情况下，例如为数值列传入非数值字符串值
- `thrownAway`：跳过的行数。 这包括时间戳在摄取任务定义的时间间隔之外的行，以及使用 `transformSpec` 过滤掉的行，但不包括显式用户配置跳过的行。例如，CSV格式的 `skipHeaderRows` 或 `hasHeaderRow` 跳过的行不计算在内
- `unparseable`：完全无法分析并被丢弃的行数。这将跟踪没有可解析结构的输入行，例如在使用JSON解析器时传入非JSON数据。

`errorMsg` 字段显示一条消息，描述导致任务失败的错误。如果任务成功，则为空

实时报告

行画像

非并行的 简单本地批处理任务、Hadoop批处理任务以及Kafka和kinesis摄取任务支持在任务运行时检索行统计信息。

可以通过运行任务的Peon上的以下URL访问实时报告：

```
http://<middlemanager-host>:<worker-port>/druid/worker/v1/chat/<task-id>/rowSt
```

示例报告如下所示。 `movingAverages` 部分包含四行计数器的1分钟、5分钟和15分钟移动平均增量，其定义与结束报告中的定义相同。 `totals` 部分显示当前总计。

```
{
  "movingAverages": {
    "buildSegments": {
      "5m": {
        "processed": 3.392158326408501,
        "unparseable": 0,
        "thrownAway": 0,
        "processedWithError": 0
      },
      "15m": {
        "processed": 1.736165476881023,
        "unparseable": 0,
        "thrownAway": 0,
        "processedWithError": 0
      },
      "1m": {
        "processed": 4.206417693750045,
        "unparseable": 0,
        "thrownAway": 0,
        "processedWithError": 0
      }
    }
  },
  "totals": {
    "buildSegments": {
      "processed": 1994,
      "processedWithError": 0,
      "thrownAway": 0,
      "unparseable": 0
    }
  }
}
```

对于Kafka索引服务，向Overlord API发送一个GET请求，将从supervisor管理的每个任务中检索实时行统计报告，并提供一个组合报告。

```
http://<OVERLORD-HOST>:<OVERLORD-PORT>/druid/indexer/v1/supervisor/<supervisor
```

未解析的事件

可以对Peon API发起一次Get请求，从正在运行的任务中检索最近遇到的不可解析事件的列表：

```
http://<middlemanager-host>:<worker-port>/druid/worker/v1/chat/<task-id>/unpar
```

注意：并不是所有的任务类型支持该功能。 当前，该功能只支持非并行的 本地批任务 (`index` 类型) 和由Kafka、Kinesis索引服务创建的任务。

任务锁系统

本节介绍Druid中的任务锁定系统。Druid的锁定系统和版本控制系统是紧密耦合的，以保证接收数据的正确性。

段与段之间的"阴影"

可以运行任务覆盖现有数据。覆盖任务创建的段将*覆盖*现有段。请注意，覆盖关系只适用于**同一时间块和同一数据源**。在过滤过时数据的查询处理中，不考虑这些被遮盖的段。

每个段都有一个*主*版本和一个*次*版本。主版本表示为时间戳，格式为"yyyy-MM-dd'T'hh:MM:ss"，次版本表示为整数。这些主版本和次版本用于确定段之间的阴影关系，如下所示。

在以下条件下，段 `s1` 将会覆盖另一个段 `s2`：

- `s1` 比 `s2` 有一个更高的主版本
- `s1` 和 `s2` 有相同的主版本，但是有更高的次版本

以下是一些示例：

- 一个主版本为 `2019-01-01T00:00:00.000Z` 且次版本为 `0` 的段将覆盖另一个主版本为 `2018-01-01T00:00:00.000Z` 且次版本为 `1` 的段
- 一个主版本为 `2019-01-01T00:00:00.000Z` 且次版本为 `1` 的段将覆盖另一个主版本为 `2019-01-01T00:00:00.000Z` 且次版本为 `0` 的段

锁

如果您正在运行两个或多个 Druid任务，这些任务为同一数据源和同一时间块生成段，那么生成的段可能会相互覆盖，从而导致错误的查询结果。

为了避免这个问题，任务将在Druid中创建任何段之前尝试获取锁, 有两种类型的锁，即 *时间块锁* 和 *段锁*。

使用时间块锁时，任务将锁定生成的段将写入数据源的整个时间块。例如，假设我们有一个任务将数据摄取到 `wikipedia` 数据源的时间块 `2019-01-01T00:00:00.000Z/2019-01-02T00:00:00.000Z` 中。使用时间块锁，此任务将在创建段之前锁定wikipedia数据源的 `2019-01-01T00:00:00.000Z/2019-01-02T00:00:00.000Z` 整个时间块。只要它持有锁，任何其他任务都将无法为同一数据源的同一时间块创建段。使用时间块锁创建的段的主版本*高于*现有段，它们的次版本总是 `0` 。

使用段锁时，任务锁定单个段而不是整个时间块。因此，如果两个或多个任务正在读取不同的段，则它们可以同时为同一时间创建同一数据源的块。例如，Kafka索引任务和压缩合并任务总是可以同时将段写入同一数据源的同一时间块中。原因是，Kafka索引任务总是附加新段，而压缩合并任务总是覆盖现有段。使用段锁创建的段具有*相同的*主版本和较高的次版本。

> [!WARNING] 段锁仍然是实验性的。它可能有未知的错误，这可能会导致错误的查询结果。

要启用段锁定，可能需要在 task context(任务上下文) 中将 `forceTimeChunkLock` 设置为 `false` 。一旦 `forceTimeChunkLock` 被取消设置，任务将自动选择正确的锁类型。**请注意**，段锁并不总是可用的。使用时间块锁的最常见场景是当覆盖任务更改段粒度时。此外，只有本地索引任务和Kafka/kinesis索引任务支持段锁。Hadoop索引任务和索引实时(`index_realtime`)任务（被 Tranquility使用）还不支持它。

任务上下文中的 `forceTimeChunkLock` 仅应用于单个任务。如果要为所有任务取消设置，则需要在 Overlord配置 中设置 `druid.indexer.tasklock.forceTimeChunkLock` 为false。

如果两个或多个任务尝试为同一数据源的重叠时间块获取锁，则锁请求可能会相互冲突。**请注意，**锁冲突可能发生在不同的锁类型之间。

锁冲突的行为取决于 任务优先级。如果冲突锁请求的所有任务具有相同的优先级，则首先请求的任务将获得锁，其他任务将等待任务释放锁。

如果优先级较低的任务请求锁的时间晚于优先级较高的任务，则此任务还将等待优先级较高的任务释放锁。如果优先级较高的任务比优先级较低的任务请求锁的时间晚，则此任务将*抢占*优先级较低的另一个任务。优先级较低的任务的锁将被撤销，优先级较高的任务将获得一个新锁。

锁抢占可以在任务运行时随时发生，除非它在关键的*段发布阶段*。一旦发布段完成，它的锁将再次成为可抢占的。

请注意，锁由同一groupId的任务共享。例如，同一supervisor的Kafka索引任务具有相同的groupId，并且彼此共享所有锁。

锁优先级

每个任务类型都有不同的默认锁优先级。下表显示了不同任务类型的默认优先级。数字越高，优先级越高。

任务类型	默认优先级
实时索引任务	75
批量索引任务	50
合并/追加/压缩任务	25
其他任务	0

通过在任务上下文中设置优先级，可以覆盖任务优先级，如下所示。

```
"context" : {
  "priority" : 100
}
```

上下文参数

任务上下文用于各种单独的任务配置。以下参数适用于所有任务类型。

属性	默认值	描述
`taskLockTimeout`	300000	任务锁定超时（毫秒）。更多详细信息，可以查看 锁 部分
`forceTimeChunkLock`	true	*将此设置为false仍然是实验性的* 。强制始终使用时间块锁。如果未设置，则每个任务都会自动选择要使用的锁类型。如果设置了，它将覆盖 Overlord配置 部分。
`priority`	不同任务类型是不同的。 参见 锁优先级	任务优先级

> [!WARNING] 当任务获取锁时，它通过HTTP发送请求并等待，直到它收到包含锁获取结果的响应为止。因此，如果 `taskLockTimeout` 大于 Overlord 的 `druid.server.http.maxIdleTime` 将会产生HTTP超时错误。

所有任务类型

`index`

参见 本地批量摄取(简单任务)

`index_parallel`

参见 本地批量社区(并行任务)

`index_sub`

由 `index_parallel` 代表您自动提交的任务。

`index_hadoop`

参见 基于Hadoop的摄取

`index_kafka`

由 `Kafka摄取supervisor` 代表您自动提交的任务。

`index_kinesis`

由 `Kinesis摄取supervisor` 代表您自动提交的任务。

`index_realtime`

由 `Tranquility` 代表您自动提交的任务。

`compact`

压缩任务合并给定间隔的所有段。有关详细信息，请参见有关 压缩 的文档。

`kill`

Kill tasks删除有关某些段的所有元数据，并将其从深层存储中删除。有关详细信息，请参阅有关 删除数据 的文档。

`append`

附加任务将段列表附加到单个段中（一个接一个）。语法是：

```
{
    "type": "append",
    "id": <task_id>,
    "dataSource": <task_datasource>,
    "segments": <JSON list of DataSegment objects to append>,
    "aggregations": <optional list of aggregators>,
    "context": <task context>
}
```

`merge`

合并任务将段列表合并在一起。合并任何公共时间戳。如果在接收过程中禁用了rollup，则不会合并公共时间戳，并按其时间戳对行重新排序。

> [!WARNING] `compact` 任务通常是比 `merge` 任务更好的选择。

语法是：

```
{
    "type": "merge",
    "id": <task_id>,
    "dataSource": <task_datasource>,
    "aggregations": <list of aggregators>,
    "rollup": <whether or not to rollup data during a merge>,
    "segments": <JSON list of DataSegment objects to merge>,
    "context": <task context>
}
```

same_interval_merge

同一间隔合并任务是合并任务的快捷方式，间隔中的所有段都将被合并。

> [!WARNING] `compact` 任务通常是比 `same_interval_merge` 任务更好的选择。

语法是:

```
{
    "type": "same_interval_merge",
    "id": <task_id>,
    "dataSource": <task_datasource>,
    "aggregations": <list of aggregators>,
    "rollup": <whether or not to rollup data during a merge>,
    "interval": <DataSegment objects in this interval are going to be merged>,
    "context": <task context>
}
```

数据摄取相关问题FAQ

实时摄取

最常见的原因是事件被摄取是在Druid的窗口时段 `windowPeriod` 范围之外。Druid实时摄取只接受当前时间的可配置窗口时段内的事件。您可以通过查看包含 `ingest/events/*` 日志行的实时进程日志来验证这是什么情况。这些z指标将标识接收、拒绝的事件等。

我们建议对生产中的历史数据使用批量摄取方法。

批量摄取

如果尝试批量加载历史数据，但没有事件被加载到，请确保摄取规范的时间间隔实际上包含了数据的间隔。此间隔之外的事件将被删除。

Druid支持什么样的数据类型

Druid可以摄取JSON、CSV、TSV和其他分隔数据。Druid支持一维值或多维值（字符串数组）。Druid支持long、float和double数值列。

并非所有的事件都被摄取了

Druid会拒绝时间窗口之外的事件， 确认事件是否被拒绝了的最佳方式是查看Druid摄取指标

如果摄取的事件数似乎正确，请确保查询的格式正确。如果在摄取规范中包含 `count` 聚合器，则需要使用 `longSum` 聚合器查询此聚合的结果。使用count聚合器发出查询将计算Druid行的数量，包括 rollup。

摄取之后段存储在哪里

段的存储位置由 `druid.storage.type` 配置决定的，Druid会将段上传到 深度存储。本地磁盘是默认的深度存储位置。

流摄取任务没有发生段切换递交

首先，确保摄取过程的日志中没有异常，如果运行的是分布式集群，还要确保 `druid.storage.type` 被设置为非本地的深度存储。

移交失败的其他常见原因如下：

1. Druid无法写入元数据存储，确保您的配置正确
2. Historical进程容量不足，无法再下载任何段。如果发生这种情况，您将在Coordinator日志中看到异常，Coordinator控制台将显示历史记录接近容量
3. 段已损坏，无法下载。如果发生这种情况，您将在Historical进程中看到异常
4. 深度存储配置不正确。确保您的段实际存在于深度存储中，并且Coordinator日志没有错误

如何让HDFS工作

确保在类路径中包含 `druid-hdfs-storage` 和所有的hadoop配置、依赖项（可以通过在安装了hadoop的计算机上运行 `hadoop classpath` 命令获得）。并且，提供必要的HDFS设置，如 深度存储 中所述。

没有在Historical进程中看到Druid段

您可以查看位于 `<Coordinator_IP>:<PORT>` 的Coordinator控制台，确保您的段实际上已加载到 Historical进程中。如果段不存在，请检查Coordinator日志中有关复制错误容量的消息。不下载段的一个原因是，Historical进程的 `maxSize` 太小，使它们无法下载更多数据。您可以使用（例如）更改它：

```
-Ddruid.segmentCache.locations=[{"path":"/tmp/druid/storageLocation","maxSize"
-Ddruid.server.maxSize=500000000000
```

查询返回来了空结果

您可以对为数据源创建的dimension和metric使用段 元数据查询。确保您在查询中使用的聚合器的名称与这些metric之一匹配，还要确保指定的查询间隔与存在数据的有效时间范围匹配。

schema变化时如何在Druid中重新索引现有数据

您可以将 DruidInputSource 与 并行任务 一起使用，以使用新schema摄取现有的druid段，并更改该段的name、dimensions、metrics、rollup等。有关详细信息，请参阅 DruidInputSource。或者，如果使用基于hadoop的摄取，那么可以使用"dataSource"输入规范来重新编制索引。

有关详细信息，请参阅 数据管理 页的 更新现有数据 部分。

如果更改Druid中现有数据的段粒度

在很多情况下，您可能希望降低旧数据的粒度。例如，任何超过1个月的数据都只有小时级别的粒度，而较新的数据只有分钟级别的粒度。此场景与重新索引相同。

为此，使用 DruidInputSource 并运行一个 并行任务。DruidInputSource 将允许你从Druid中获取现有的段并将它们聚合并反馈给Druid。它还允许您在反馈数据时过滤这些段中的数据，这意味着，如果有要删除的行，可以在重新摄取期间将它们过滤掉。通常，上面的操作将作为一个批处理作业运行，即每天输入一大块数据并对其进行聚合。或者，如果使用基于hadoop的摄取，那么可以使用"dataSource"输入规范来重新编制索引。

有关详细信息，请参阅 数据管理 页的 更新现有数据 部分。

实时摄取似乎被卡住了

有几种方法可以做到这一点。如果中间持久化消耗太长时间或如果移交消耗太长事件，Druid将限制摄入，以防止内存不足的问题。如果您的流程日志表明某些列的生成时间非常长（例如，如果您的段粒度是每小时一次，但是创建一个列需要30分钟），那么您应该重新评估您的配置或扩展您的实时接收

更多信息

对于第一次使用Druid的用户来说，将数据输入Druid是非常困难的。请不要犹豫，在我们的IRC频道或在我们的 google群组 页面上提问。

SQL

> [!WARNING] Apache Druid支持两种查询语言： Druid SQL和 原生查询。本文档讲述SQL查询。

Druid SQL是一个内置的SQL层，是Druid基于JSON的本地查询语言的替代品，它由基于 Apache Calcite 的解析器和规划器提供支持。Druid SQL将SQL转换为查询Broker(查询的第一个进程)上的原生Druid查询，然后作为原生Druid查询传递给数据进程。除了在Broker上 转换SQL 的（轻微）开销之外，与原生查询相比，没有额外的性能损失。

查询符号

Druid SQL支持如下结构的SELECT查询：

```
[ EXPLAIN PLAN FOR ]
[ WITH tableName [ ( column1, column2, ... ) ] AS ( query ) ]
SELECT [ ALL | DISTINCT ] { * | exprs }
FROM { <table> | (<subquery>) | <o1> [ INNER | LEFT ] JOIN <o2> ON condition }
[ WHERE expr ]
[ GROUP BY [ exprs | GROUPING SETS ( (exprs), ... ) | ROLLUP (exprs) | CUBE (e
[ HAVING expr ]
[ ORDER BY expr [ ASC | DESC ], expr [ ASC | DESC ], ... ]
[ LIMIT limit ]
[ UNION ALL <another query> ]
```

FROM

FROM子句可以引用下列任何一个：

- 来自 `druid` schema中的 表数据源。 这是默认schema，因此可以将Druid表数据源引用为 `druid.dataSourceName` 或者简单的 `dataSourceName` 。
- 来自 `lookup` schema的 lookups, 例如 `lookup.countries` 。 注意：lookups还可以使用 Lookup函数 来查询。
- 子查询
- 列表中任何内容之间的 joins，本地数据源（table、lookup、query）和系统表之间的联接除外。连接条件必须是连接左侧和右侧的表达式之间的相等。
- 来自于 `INFORMATION_SCHEMA` 或者 `sys` schema的 元数据表

有关table、lookup、query和join数据源的更多信息，请参阅 数据源文档。

WHERE

WHERE子句引用FROM表中的列，并将转换为 原生过滤器。WHERE子句还可以引用子查询，比如 `WHERE col1 IN (SELECT foo FROM ...)` 。像这样的查询作为子查询的连接执行，如下在 查询转换 部分所述。

GROUP BY

GROUP BY子句引用FROM表中的列。使用 **GROUP BY**、**DISTINCT** 或任何聚合函数都将使用Druid的 三种原生聚合查询类型之一触发聚合查询。**GROUP BY**可以引用表达式或者select子句的序号位置（如 `GROUP BY 2` 以按第二个选定列分组）。

GROUP BY子句还可以通过三种方式引用多个分组集。 最灵活的是 **GROUP BY GROUPING SETS**，例如 `GROUP BY GROUPING SETS ( (country, city), () )`，该实例等价于一个 `GROUP BY country, city` 然后 `GROUP BY ()`。 对于**GROUPING SETS**，底层数据只扫描一次，从而提高了效率。其次，**GROUP BY ROLLUP**为每个级别的分组表达式计算一个分组集，例如 `GROUP BY ROLLUP (country, city)` 等价于 `GROUP BY GROUPING SETS ( (country, city), (country), () )` ，将为每个country/city对生成分组行，以及每个country的小计和总计。最后，**GROUP BY CUBE**为每个分组表达式组合计算分组集，例如 `GROUP BY CUBE (country, city)` 等价于 `GROUP BY GROUPING SETS ( (country, city), (country), (city), () )`。对不适用于特定行的列进行分组将包含 `NULL`，例如，当计算 `GROUP BY GROUPING SETS ( (country, city), () )`，与 `()` 对应的总计行对于"country"和"city"列将为 `NULL`。

使用 **GROUP BY GROUPING SETS**, **GROUP BY ROLLUP**, 或者 **GROUP BY CUBE**时，请注意，可能不会按照在查询中指定分组集的顺序生成结果。如果需要按特定顺序生成结果，请使用**ORDER BY**子句。

HAVING

HAVING子句引用在执行**GROUP BY**之后出现的列，它可用于对分组表达式或聚合值进行筛选，它只能与**GROUP BY**一起使用。

ORDER BY

ORDER BY子句引用执行**GROUP BY**后出现的列。它可用于根据分组表达式或聚合值对结果进行排序。**ORDER BY**可以引用表达式或者select子句序号位置（例如 `ORDER BY 2` 根据第二个选定列进行排序）。对于非聚合查询，**ORDER BY**只能按 `__time` 排序。对于聚合查询，**ORDER BY**可以按任何列排序。

LIMIT

LIMIT子句可用于限制返回的行数。它可以用于任何查询类型。对于使用原生TopN查询类型（而不是原生GroupBy查询类型）运行的查询，它被下推到数据进程。未来的Druid版本也将支持使用原生GroupBy查询类型来降低限制。如果您注意到添加一个限制并不会对性能产生很大的影响，那么很可能Druid并没有降低查询的限制。

UNION ALL

UNION ALL操作符可用于将多个查询融合在一起。它们的结果将被连接起来，每个查询将单独运行，背对背（不并行）。Druid现在不支持没有"All"的"UNION"。**UNION ALL**必须出现在SQL查询的最外层（它不能出现在子查询或FROM子句中）。

请注意，尽管名称相似，UNION ALL与 union datasource 并不是一回事。**UNION ALL**允许联合查询结果，而UNION数据源允许联合表。

EXPLAIN PLAN

在任何查询的开头添加"EXPLAIN PLAN FOR"，以获取有关如何转换的信息。在这种情况下，查询实际上不会执行。有关解释**EXPLAIN PLAN**输出的帮助，请参阅查询转换文档。

标识符和字面量

可以选择使用双引号引用数据源和列名等标识符。要在标识符中转义双引号，请使用另一个双引号，如 `"My ""very own"" identifier"` 。所有标识符都区分大小写，不执行隐式大小写转换。

字面量字符串应该用单引号引起来，如 `'foo'` 。带Unicode转义符的文本字符串可以像 `U&'fo\00F6'` 一样写入，其中十六进制字符代码的前缀是反斜杠。字面量数字可以写成 `100` （表示整数）、 `100.0` （表示浮点值）或 `1.0e5` （科学表示法）等形式。字面量时间戳可以像 `TIMESTAMP '2000-01-01 00:00:00'` 一样写入, 用于时间算术的字面量间隔可以写成 `INTERVAL '1' HOUR` 、 `INTERVAL '1 02:03' DAY TO MINUTE, INTERVAL '1-2' YEAR TO MONTH` 等等。

动态参数

Druid SQL支持使用问号 `(?)` 的动态参数语法，动态参数在执行时绑定到占位符 `?` 中。若要使用动态参数，请将查询中的任何文本替换为 `?` 字符，并在执行查询时提供相应的参数值, 参数按传递顺序绑定到占位符。HTTP POST和JDBC APIs都支持参数。

数据类型

标准类型

Druid原生支持五种列类型："long"(64位有符号整型),"float"(32位浮点型),"double"(64位浮点型),"string"(UTF-8编码的字符串或者字符串数组)和"complex"(获取更多奇异的数据类型，如hyperUnique列和approxHistogram列)

时间戳（包括 `__time` 列）被Druid视为long，其值是1970-01-01T00:00:00 UTC以来的毫秒数，不包括闰秒。因此，Druid中的时间戳不携带任何时区信息，而只携带关于它们所代表的确切时间的信息。有关时间戳处理的更多信息，请参阅 时间函数部分。

下表描述了Druid如何在查询运行时将SQL类型映射到原生类型。在具有相同Druid运行时类型的两个SQL类型之间进行强制转换不会产生任何影响，除非表中指出了异常。两个具有不同Druid运行时类型的SQL类型之间的转换将在Druid中生成一个运行时转换。如果一个值不能正确地转换为另一个值，如 `CAST('foo' AS BIGINT)` ，则运行时将替换默认值。 NULL转换为不可为空类型时将替换为默认值（例如，NULL转为数字将转换为零）。

SQL类型	Druid运行时类型	默认值	注意事项
CHAR	STRING	`''`	
VARCHAR	STRING	`''`	Druid STRING列报告为VARCHAR，包括 多值字符串
DECIMAL	DOUBLE	`0.0`	DECIMAL使用浮点，非定点
FLOAT	FLOAT	`0.0`	Druid FLOAT列报告为FLOAT
REAL	DOUBLE	`0.0`	
DOUBLE	DOUBLE	`0.0`	Druid DOUBLE列报告为DOUBLE
BOOLEAN	LONG	`false`	
TINYINT	LONG	`0`	
SMALLINT	LONG	`0`	
INTEGER	LONG	`0`	
BIGINT	LONG	`0`	Druid LONG列(除了 `__time` 报告为BIGINT
TIMESTAMP	LONG	`0`，意思是1970-01-01 00:00:00 UTC	Druid的 `__time` 列被报告为TIMESTAMP。 string和timestamp类型的转换都是假定为标准格式，例如 `2000-01-02 03:04:05`，而非ISO8601格式。 有关时间戳处理的更多信息，请参阅 时间函数部分。
DATE	LONG	`0`，意思是1970-01-01	转换TIMESTAMP为DATE时间戳将时间戳舍入到最近的一天。string和date类型的转换都是假定为标准格式，例如 `2000-01-02` 。 有关时间戳处理的更多信息，请参阅 时间函数部分。
OTHER	COMPLEX	none	可以表示各种Druid列类型，如hyperUnique、approxHistogram等

多值字符串

Druid的原生类型系统允许字符串可能有多个值。这些 多值维度 将被报告为SQL中的 `VARCHAR` 类型，可以像任何其他VARCHAR一样在语法上使用。引用多值字符串维度的常规字符串函数将分别应用于每行的所有值，多值字符串维度也可以通过特殊的 多值字符串函数 作为数组处理，该函数可以执行强大的数组操作。

按多值表达式分组将observe原生Druid多值聚合行为，这与某些其他SQL语法中 `UNNEST` 的功能类似。有关更多详细信息，请参阅有关 多值字符串维度 的文档。

NULL

runtime property 中的 `druid.generic.useDefaultValueForNull` 配置控制着Druid的NULL处理模式。

在默认模式(`true`)下，Druid将NULL和空字符串互换处理，而不是根据SQL标准。在这种模式下，Druid SQL只部分支持NULL。例如，表达式 `col IS NULL` 和 `col = ''` 等效，如果 `col` 包含空字符串，则两者的计算结果都为true。类似地，如果 `col1` 是空字符串，则表达式 `COALESCE(col1, col2)` 将返回 `col2` 。当 `COUNT(*)` 聚合器计算所有行时， `COUNT(expr)` 聚合器将计算expr既不为空也不为空字符串的行数。此模式中的数值列不可为空；任何空值或缺少的值都将被视为零。

在SQL兼容模式(`false`)中，NULL的处理更接近SQL标准，该属性同时影响存储和查询，因此为了获得最佳行为，应该在接收时和查询时同时设置该属性。处理空值的能力会带来一些开销；有关更多详细信息，请参阅 段文档。

聚合函数

聚合函数可以出现在任务查询的SELECT子句中，任何聚合器都可以使用 `AGG(expr) FILTER(WHERE whereExpr)` 这样的表达式进行过滤。 被过滤的聚合器仅仅聚合那些匹配了过滤器的行。 同一个SQL查询中的两个聚合器可能有不同的过滤器。

只有COUNT聚合支持使用DISTINCT

函数	描述
`COUNT(*)`	计算行数
`COUNT( DISTINCT expr)`	唯一值的计数，表达式可以是string、numeric或者hyperUnique。默认情况下，这是近似值，使用HyperLogLog 的变体。若要获取精确计数，请将"useApproximateCountDistinct"设置为"false"。如果这样做，expr必须是字符串或数字，因为使用hyperUnique列不可能精确计数。另请参见 `APPROX_COUNT_DISTINCT(expr)` 。在精确模式下，每个查询只允许一个不同的计数。
`SUM(expr)`	求和
`MIN(expr)`	取数字的最小值
`MAX(expr)`	取数字的最大值
`AVG(expr)`	取平均值
`APPROX_COUNT_DISTINCT(expr)`	唯一值的计数，该值可以是常规列或hyperUnique。这始终是近似值，而不考虑"useApproximateCountDistinct"的值。该函数使用了Druid内置的"cardinality"或"hyperUnique"聚合器。另请参见 `COUNT(DISTINCT expr)`
`APPROX COUNT DISTINCT_DS_HLL(expr, [lgK, tgtHllType])`	唯一值的计数，该值可以是常规列或HLL sketch。 `lgk` 和 `tgtHllType` 参数在HLL Sketch文档中做了描述。 该值也始终是近似值，而不考虑"useApproximateCountDistinct"的值。另请参见 `COUNT(DISTINCT expr)` , 使用该函数需要加载DataSketches扩展
`APPROX COUNT_DISTINCT_DS_THETA(expr, [size])`	唯一值的计数，该值可以是常规列或Theta sketch。 `size` 参数在Theta Sketch文档中做了描述。 该值也始终是近似值，而不考虑"useApproximateCountDistinct"的值。另请参见 `COUNT(DISTINCT expr)` , 使用该函数需要加载DataSketches扩展

函数	描述
`DS_HLL(expr, [lgK, tgtHllType])`	在表达式的值上创建一个 `HLL sketch`，该值可以是常规列或者包括HLL Sketch的列。 `lgk` 和 `tgtHllType` 参数在HLL Sketch文档中做了描述。使用该函数需要加载 DataSketches扩展
`DS_THETA(expr, [size])`	在表达式的值上创建一个 `Theta sketch`，该值可以是常规列或者包括Theta Sketch的列。 `size` 参数在Theta Sketch文档中做了描述。使用该函数需要加载 DataSketches扩展
`APPROX_QUANTILE(expr, probability, [resolution])`	在数值表达式或者近似图 表达式上计算近似分位数，"probability"应该是位于0到1之间（不包括1），"resolution"是用于计算的centroids，更高的resolution将会获得更精确的结果，默认值为50。使用该函数需要加载 近似直方图扩展
`APPROX_QUANTILE_DS(expr, probability, [k])`	在数值表达式或者 Quantiles sketch 表达式上计算近似分位数，"probability"应该是位于0到1之间（不包括1），`k` 参数在Quantiles Sketch文档中做了描述。使用该函数需要加载 DataSketches扩展
`APPROX_QUANTILE_FIXED_BUCKETS(expr, probability, numBuckets, lowerLimit, upperLimit, [outlierHandlingMode])`	在数值表达式或者fixed buckets直方图 表达式上计算近似分位数，"probability"应该是位于0到1之间（不包括1），`numBuckets`，`lowerLimit`，`upperLimit` 和 `outlierHandlingMode` 参数在fixed buckets直方图文档中做了描述。使用该函数需要加载 近似直方图扩展
`DS_QUANTILES_SKETCH(expr, [k])`	在表达式的值上创建一个 `Quantiles sketch`，该值可以是常规列或者包括Quantiles Sketch的列。 `k` 参数在Quantiles Sketch文档中做了描述。使用该函数需要加载 DataSketches扩展
`BLOOM_FILTER(expr, numEntries)`	根据 `expr` 生成的值计算bloom筛选器，其中 `numEntries` 在假阳性率增加之前具有最大数量的不同值。详细可以参见 Bloom过滤器扩展

函数	描述
`TDIGEST QUANTILE(expr, quantileFraction, [compression])`	根据 `expr` 生成的值构建一个T-Digest sketch，并返回分位数的值。"compression"（默认值100）确定sketch的精度和大小。更高的compression意味着更高的精度，但更多的空间来存储sketch。有关更多详细信息，请参阅 t-digest扩展文档
`TDIGEST GENERATE_SKETCH(expr, [compression])`	根据 `expr` 生成的值构建一个T-Digest sketch。"compression"（默认值100）确定sketch的精度和大小。更高的compression意味着更高的精度，但更多的空间来存储sketch。有关更多详细信息，请参阅 t-digest扩展文档
`VAR_POP(expr)`	计算 `expr` 的总体方差, 额外的信息参见 stats扩展文档
`VAR_SAMP(expr)`	计算表达式的样本方差，额外的信息参见 stats扩展文档
`VARIANCE(expr)`	计算表达式的样本方差，额外的信息参见 stats扩展文档
`STDDEV_POP(expr)`	计算 `expr` 的总体标准差, 额外的信息参见 stats扩展文档
`STDDEV_SAMP(expr)`	计算表达式的样本标准差，额外的信息参见 stats扩展文档
`STDDEV(expr)`	计算表达式的样本标准差，额外的信息参见 stats扩展文档
`EARLIEST(expr)`	返回 `expr` 的最早值，该值必须是数字。如果 `expr` 来自一个与timestamp列（如Druid数据源）的关系，那么"earliest"是所有被聚合值的最小总时间戳最先遇到的值。如果 `expr` 不是来自带有时间戳的关系，那么它只是遇到的第一个值。
`ARLIEST(expr, maxBytesPerString)`	与 `EARLIEST(expr)` 相似，但是面向string。 `maxBytesPerString` 参数确定每个字符串要分配多少聚合空间, 超过此限制的字符串将被截断。这个参数应该设置得尽可能低，因为高值会导致内存浪费。

函数	描述
`LATEST(expr)`	返回 `expr` 的最新值，该值必须是数字。如果 `expr` 来自一个与timestamp列（如Druid数据源）的关系，那么"latest"是最后一次遇到的值，它是所有被聚合的值的最大总时间戳。如果 `expr` 不是来自带有时间戳的关系，那么它只是遇到的最后一个值。
`LATEST(expr, maxBytesPerString)`	与 `LATEST(expr)` 类似，但是面向string。 `maxBytesPerString` 参数确定每个字符串要分配多少聚合空间，超过此限制的字符串将被截断。这个参数应该设置得尽可能低，因为高值会导致内存浪费。
`ANY_VALUE(expr)`	返回 `expr` 的任何值，包括null。 `expr` 必须是数字，此聚合器可以通过返回第一个遇到的值（包括空值）来简化和优化性能
`ANY_VALUE(expr, maxBytesPerString)`	与 `ANY_VALUE(expr)` 类似，但是面向string。 `maxBytesPerString` 参数确定每个字符串要分配多少聚合空间，超过此限制的字符串将被截断。这个参数应该设置得尽可能低，因为高值会导致内存浪费。

对于近似聚合函数，请查看 近似聚合文档

扩展函数

数值函数

对于数学运算，如果表达式中涉及的所有操作数都是整数，Druid SQL将使用整数数学。否则，Druid将切换到浮点数学，通过将一个操作数转换为浮点，可以强制执行此操作。在运行时，对于大多数表达式，Druid将把32位浮点扩展到64位。

函数	描述
`ABS(expr)`	绝对值
`CEIL(expr)`	向上取整
`EXP(expr)`	次方
`FLOOR(expr)`	向下取整
`LN(expr)`	对数（以e为底）
`LOG10(expr)`	对数（以10为底）
`POWER(expr,power)`	次方
`SQRT(expr)`	开方
`TRUNCATE(expr[, digits])`	将 `expr` 截断为指定的小数位数。如果数字为负数，则此操作会截断小数点左侧的许多位置。如果未指定，则数字默认为零。
`ROUND(expr[, digits])`	`ROUND(x, y)` 将返回x的值，并四舍五入到y小数位。虽然x可以是整数或浮点数，但y必须是整数。返回值的类型由x的类型指定。如果省略，则默认为0。当y为负时，x在y小数点的左侧四舍五入。
`x + y`	加
`x - y`	减
`x * y`	乘
`x / y`	除
`MOD(x, y)`	模除
`SIN(expr)`	正弦
`COS(expr)`	余弦
`TAN(expr)`	正切
`COT(expr)`	余切
`ASIN(expr)`	反正弦
`ACOS(expr)`	反余弦
`ATAN(expr)`	反正切
`ATAN2(y, x)`	从直角坐标（x，y）到极坐标（r，θ）的转换角度θ。
`DEGREES(expr)`	将以弧度测量的角度转换为以度测量的近似等效角度
`RADIANS(expr)`	将以度为单位测量的角度转换为以弧度为单位测量的近似等效角度

字符串函数

字符串函数接受字符串，并返回与该函数相应的类型。

函数	描述		
`x		y`	拼接字符
`CONCAT(expr, expr, ...)`	拼接一系列表达式		
`TEXTCAT(expr, expr)`	两个参数版本的CONCAT		
`STRING_FORMAT(pattern[, args...])`	返回以Java的 方式格式化 的字符串字符串格式		
`LENGTH(expr)`	UTF-16代码单位的长度或表达式		
`CHAR_LENGTH(expr)`	`LENGTH` 的同义词		
`CHARACTER_LENGTH(expr)`	`LENGTH` 的同义词		
`STRLEN(expr)`	`LENGTH` 的同义词		
`LOOKUP(expr, lookupName)`	已注册的 查询时Lookup表的Lookup表达式。 注意：lookups也可以直接使用 `lookup schema` 来查询		
`LOWER(expr)`	返回的expr的全小写		
`PARSE_LONG(string[, radix])`	将字符串解析为具有给定基数的长字符串（BIGINT），如果未提供基数，则解析为10（十进制）。		
`POSITION(needle IN haystack [FROM fromIndex])`	返回haystack中指针的索引，索引从1开始。搜索将从fromIndex开始，如果未指定fromIndex，则从1开始。如果找不到针，则返回0。		
`REGEXP_EXTRACT(expr, pattern, [index])`	应用正则表达式模式并提取捕获组，如果没有匹配，则为空。如果index未指定或为零，则返回与模式匹配的子字符串。		

函数	描述
`REPLACE(expr, pattern, replacement)`	在expr中用replacement替换pattern，并返回结果。
`STRPOS(haystack, needle)`	返回haystack中指针的索引，索引从1开始。如果找不到针，则返回0。
`SUBSTRING(expr, index, [length])`	返回从索引开始的expr子字符串，最大长度均以UTF-16代码单位度量。
`RIGHT(expr, [length])`	从expr返回最右边的长度字符。
`LEFT(expr, [length])`	返回expr中最左边的长度字符。
`SUBSTR(expr, index, [length])`	SUBSTRING的同义词
`TRIM([BOTH LEADING TRAILING] [ FROM] expr)`	返回exp果字符在"chars则从"exp开头、结两端删除符。如果供"chars默认为""格）。如提供方向数，则默为"BOTH
`BTRIM(expr[, chars])`	`TRIM(BOTH <chars> FROM <expr>)` 的替代格式
`LTRIM(expr[, chars])`	`TRIM(LEADING <chars> FROM <expr>)` 的替代格式
`RTRIM(expr[, chars])`	`TRIM(TRAILING <chars> FROM <expr>)` 的替代格式
`UPPER(expr)`	返回全大写的expr
`REVERSE(expr)`	反转expr
`REPEAT(expr, [N])`	将expr重复N次

函数	描述
`LPAD(expr, length[, chars])`	从"expr"中返回一个用"chars"填充的"length"字符串。如果"length"小于"expr"的长度，则结果为"expr"，并被截断为"length"。如果"expr"或"chars"为空，则结果为空。
`RPAD(expr, length[, chars])`	从"expr"返回一个用"chars"填充的"length"字符串。如果"length"小于"expr"的长度，则结果为"expr"，并被截断为"length"。如果"expr"或"chars"为空，则结果为空。

时间函数

时间函数可以与Druid的时 `__time` 一起使用，任何存储为毫秒时间戳的列都可以使用 `MILLIS_TO_TIMESTAMP` 函数，或者任何存储为字符串时间戳的列都可以使用 `TIME_PARSE` 函数。默认情况下，时间操作使用UTC时区。您可以通过将连接上下文参数"sqlTimeZone"设置为另一个时区的名称（如"America/Los_Angeles"）或设置为偏移量（如"-08:00"）来更改时区。如果需要在同一查询中混合多个时区，或者需要使用连接时区以外的时区，则某些函数还接受时区作为参数。这些参数始终优先于连接时区。

连接时区中的字面量时间戳可以使用 `TIMESTAMP '2000-01-01 00:00:00'` 语法编写。在其他时区写入字面量时间戳的最简单方法是使用TIME_PARSE，比如 `TIME_PARSE('2000-02-01 00:00:00', NULL, 'America/Los_Angeles')` 。

函数	描述
`CURRENT_TIMESTAMP`	在连接时区的当前时间戳
`CURRENT_DATE`	在连接时区的当期日期
`DATE_TRUNC(<unit>, <timestamp_expr>)`	截断时间戳，将其作为新时间戳返回秒"、"秒"、"分"、"时"、"日"、"周"、"年"、"世纪"或"千年"。
`TIME_CEIL(<timestamp_expr>, <period>, [<origin>, [<timezone>]])`	对时间戳进行向上取整，并将其作为期可以是任何ISO8601周期，如P3M PT12H（半天）。时区（如果提供）如"America/Los_Angeles"或偏移量，类似于 `CEIL` ，但更灵活。
`TIME_FLOOR(<timestamp_expr>, <period>, [<origin>, [<timezone>]])`	对时间戳进行向下取整，将其作为新以是任何ISO8601周期，如P3M（季天）。时区（如果提供）应为时区名如"America/Los_Angeles"或偏移量，类似于 `FLOOR` ，但更灵活。
`TIME_SHIFT(<timestamp_expr>, <period>, <step>, [<timezone>])`	将时间戳移动一个周期（步进时间）戳返回。 `period` 可以是任何ISO86为负。时区（如果提供）应为时区名如"America/Los_Angeles"或偏移量，
`TIME_EXTRACT(<timestamp_expr>, [<unit>, [<timezone>]])`	从expr中提取时间部分，并将其作为是EPOCH、SECOND、MINUTE、H日）、DOW（周的日）、DOY（年的周）、MONTH（1到12）、QUARTE YEAR。时区（如果提供）应为时区名如"America/Los_Angeles"或偏移量，类似于 `EXTRACT` ，但更灵活。单位和并且必须提供引号，如时间提取 `TIM 'HOUR')` 或 `TIME_EXTRACT(__time, 'HO 'America/Los_Angeles')` 。
`TIME_PARSE(<string_expr>, [<pattern>, [<timezone>]])`	如果未提供该 `pattern` ，使用给定的 DateTimeFormat模式 或ISO8601（例 `02T03:04:05Z` ）将字符串解析为时间戳供）应为时区名称，如"America/Los_量，如"-08:00"，并将用作不包括时区时区。模式和时区必须是字面量。无效符串将返回空值。
`TIME_FORMAT(<timestamp_expr>, [<pattern>, [<timezone>]])`	如果 `pattern` 未提供，使用给定的 J DateTimeFormat模式 或ISO8601（例 `02T03:04:05Z` ）将时间戳格式化为字符供）应为时区名称，如"America/Los_量，如"-08:00"，并将用作不包括时区时区。模式和时区必须是字面量。无效符串将返回空值。
`MILLIS_TO_TIMESTAMP(millis_expr)`	将纪元后的毫秒数转换为时间戳。

函数	描述
`TIMESTAMP_TO_MILLIS(timestamp_expr)`	将时间戳转换为自纪元以来的毫秒数
`EXTRACT(<unit> FROM timestamp_expr)`	从expr中提取时间部分，并将其作为 是EPOCH, MICROSECOND, MILLIS MINUTE, HOUR, DAY (day of month week), ISODOW (ISO day of week), WEEK (week of year), MONTH, QUA ISOYEAR, DECADE, CENTURY or 提供未加引号的单位，如 `EXTRACT(HO`
`FLOOR(timestamp_expr TO <unit>)`	向下取整时间戳，将其作为新时间戳 SECOND, MINUTE, HOUR, DAY, W QUARTER, 或者YEAR
`CEIL(timestamp_expr TO <unit>)`	向上取整时间戳，将其作为新时间戳 SECOND, MINUTE, HOUR, DAY, W QUARTER, 或者YEAR
`TIMESTAMPADD(<unit>, <count>, <timestamp>)`	等价于 `timestamp + count * INTERVAL`
`TIMESTAMPDIFF(<unit>, <timestamp1>, <timestamp2>)`	返回 `timestamp1` 和 `timestamp2` 之间
`timestamp_expr { +/- } <interval_expr>`	从时间戳中加上或减去时间量。`inte` `INTERVAL '2' HOUR` 之类的区间字面量 算法。该操作将天数统一视为86400秒 时。要计算夏时制时间，请使用 `TIM`

归约函数

归约函数对零个或多个表达式进行操作，并返回单个表达式。如果没有表达式作为参数传递，则结果为 `NULL` 。表达式必须全部转换为公共数据类型，即结果的类型：

- 如果所有的参数都是 `NULL` , 结果是 `NULL` , 否则， `NULL` 参数被忽略
- 如果所有的参数包含了数字和字符串的混合，参数都被解释为字符串
- 如果所有的参数是整型数字，参数都被解释为长整型
- 如果所有的参数是数值且至少一个参数是double，则参数都被解释为double

函数	描述
`GREATEST([expr1, ...])`	计算零个或多个表达式，并根据上述比较返回最大值。
`LEAST([expr1, ...])`	计算零个或多个表达式，并根据上述比较返回最小值。

IP地址函数

对于IPv4地址函数，地址参数可以是IPv4点分十进制字符串（例如"192.168.0.1"）或表示为整数的IP地址（例如3232235521）。 `subnet` 参数应该是一个字符串，格式为CIDR表示法中的IPv4地址子网（例如"192.168.0.0/16"）。

函数	描述
`IPV4 MATCH(address, subnet)`	如果 `address` 属于 `subnet` 文本，则返回true，否则返回false。如果 `address` 不是有效的IPv4地址，则返回false。如果 `address` 是整数而不是字符串，则此函数更效率。
`IPV4_PARSE(address)`	将 `address` 解析为存储为整数的IPv4地址。如果 `address` 是有效的IPv4地址的整数，则它将被可以解析。如果 `address` 不能表示为IPv4地址，则返回null。
`IPV4_STRINGIFY(address)`	将 `address` 转换为以点分隔的IPv4地址十进制字符串。如果 `address` 是有效的IPv4地址的字符串，则它将解析。如果 `address` 不能表示为IPv4地址，则返回null。

比较操作符

函数	描述
`x = y`	等于
`x <> y`	不等于
`x > y`	大于
`x >= y`	大于等于
`x < y`	小于
`x <= y`	小于等于
`x BETWEEN y AND z`	等价于 `x >= y AND x <= z`
`x NOT BETWEEN y AND z`	等价于 `x >= y OR x <= z`
`x LIKE pattern [ESCAPE esc]`	如果x匹配上了一个SQL LIKE模式则返回true
`x NOT LIKE pattern [ESCAPE esc]`	如果x没有匹配上了一个SQL LIKE模式则返回true
`x IS NULL`	如果x是NULL或者空串，返回true
`x IS NOT NULL`	如果x不是NULL也不是空串，返回true
`x IS TRUE`	如果x是true，返回true
`x IS NOT TRUE`	如果x不是true，返回true
`x IS FALSE`	如果x是false，返回true
`x IS NOT FALSE`	如果x不是false，返回true
`x IN (values)`	如果x是列出的值之一，则为True
`x NOT IN (values)`	如果x不是列出的值之一，则为True
`x IN (subquery)`	如果子查询返回x，则为True。这将转换为联接；有关详细信息，请参阅 查询转换
`x NOT IN (subquery)`	如果子查询没有返回x，则为True。这将转换为联接；有关详细信息，请参阅 查询转换
`x AND y`	与
`x OR y`	或
`NOT x`	非

Sketch函数

这些函数对返回sketch对象的表达式或列进行操作。

HLL Sketch函数

以下函数操作在 DataSketches HLL sketches 之上，使用这些函数之前需要加载 DataSketches扩展

函数	描述
`HLL_SKETCH_ESTIMATE(expr, [round])`	从HLL草图返回非重复计数估计值。`expr` 必须返回HLL草图。可选的 `round` 布尔参数如果设置为 `true` 将舍入估计值，默认值为 `false` 。
`HLL SKETCH ESTIMATE_WITH_ERROR_BOUNDS(expr, [numStdDev])`	从HLL草图返回不同的计数估计值和错误边界。 `expr` 必须返回HLL草图。可以提供可选的 `numStdDev` 参数。
`HLL SKETCH UNION([lgK, tgtHllType], expr0, expr1, ...)`	返回HLL草图的并集，其中每个输入表达式必须返回HLL草图。可以选择将 `lgK` 和 `tgtHllType` 指定为第一个参数；如果提供了，则必须同时指定这两个可选参数。
`HLL_SKETCH_TO_STRING(expr)`	返回用于调试的HLL草图的可读字符串表示形式。 `expr` 必须返回HLL草图。

Theta Sketch函数

以下函数操作在 theta sketches 之上，使用这些函数之前需要加载 DataSketches 扩展

函数	描述
`THETA_SKETCH_ESTIMATE(expr)`	从theta草图返回不同的计数估计值。`expr` 必须返回theta草图。
`THETA SKETCH ESTIMATE_WITH_ERROR_BOUNDS(expr, errorBoundsStdDev)`	从theta草图返回不同的计数估计值和错误边界。 `expr` 必须返回theta草图。
`THETA_SKETCH_UNION([size], expr0, expr1, ...)`	返回theta草图的并集，其中每个输入表达式必须返回theta草图。可以选择将 `size` 指定为第一个参数。
`THETA_SKETCH_INTERSECT([size], expr0, expr1, ...)`	返回theta草图的交集，其中每个输入表达式必须返回theta草图。可以选择将 `size` 指定为第一个参数。
`THETA_SKETCH_NOT([size], expr0, expr1, ...)`	返回theta草图的集合差，其中每个输入表达式必须返回theta草图。可以选择将 `size` 指定为第一个参数。

Quantiles Sketch函数

以下函数操作在 quantiles sketches 之上，使用这些函数之前需要加载 DataSketches扩展

函数	描述
`DS_GET_QUANTILE(expr, fraction)`	返回与来自分位数草图的 `fraction` 相对应的分位数估计。 `expr` 必须返回分位数草图。
`DS_GET_QUANTILES(expr, fraction0, fraction1, ...)`	返回一个字符串，该字符串表示与分位数草图的分数列表相对应的分位数估计数组。 `expr` 必须返回分位数草图
`DS_HISTOGRAM(expr, splitPoint0, splitPoint1, ...)`	返回一个字符串，该字符串表示给定一个分割点列表的直方图近似值，该列表定义了分位数草图中的直方图箱。 `expr` 必须返回分位数草图。
`DS_CDF(expr, splitPoint0, splitPoint1, ...)`	返回一个字符串，该字符串表示给定的分割点列表（该列表定义了来自分位数草图的容器边缘）的累积分布函数的近似值。 `expr` 必须返回分位数草图。
`DS_RANK(expr, value)`	返回对给定值的秩的近似值，该值是分布的分数，小于来自分位数草图的该值。 `expr` 必须返回分位数草图。
`DS_QUANTILE_SUMMARY(expr)`	返回分位数草图的字符串摘要，用于调试。 `expr` 必须返回分位数草图。

其他扩展函数

函数	描述
`CAST(value AS TYPE)`	将值转换为其他类型。 可以查看 数据类型 来了解在Druid SQL中如何传利CAST
`CASE expr WHEN value1 THEN result1 \[ WHEN value2 THEN result2 ... \] \[ ELSE resultN \] END`	简单CASE
`CASE WHEN boolean expr1 THEN result1 \[ WHEN boolean expr2 THEN result2 ... \] \[ ELSE resultN \] END`	搜索CASE
`NULLIF(value1, value2)`	如果value1和value2匹配，则返回NULL，否则返回value1
`COALESCE(value1, value2, ...)`	返回第一个既不是NULL也不是空字符串的值。
`NVL(expr,expr-for-null)`	如果'expr'为空（或字符串类型为空字符串），则返回 `expr for null`
`BLOOM_FILTER_TEST(<expr>, <serialized-filter>)`	如果值包含在Base64序列化bloom筛选器中，则返回true。详情查看 Bloom Filter扩展

多值字符串函数

多值字符串函数文档中的所有"array"引用都可以引用多值字符串列或数组字面量。

函数	描述
`ARRAY(expr1,expr ...)`	从表达式参数构造SQL数组字面量，使用第一个参数的类型作为输出数组类型
`MV_LENGTH(arr)`	返回数组表达式的长度
`MV_OFFSET(arr,long)`	返回所提供的基于0的索引处的数组元素，或对于超出范围的索引返回null
`MV_ORDINAL(arr,long)`	返回所提供的基于1的索引处的数组元素，或对于超出范围的索引返回null
`MV_CONTAINS(arr,expr)`	如果数组包含expr指定的元素，则返回1；如果expr是数组，则返回expr指定的所有元素，否则返回0
`MV_OVERLAP(arr1,arr2)`	如果arr1和arr2有任何共同元素，则返回1，否则返回0
`MV_OFFSET_OF(arr,expr)`	返回数组中expr第一次出现的基于0的索引，或 `-1` 或 `null` 。如果 `druid.generic.useDefaultValueForNull=false` 如果数组中不存在匹配元素。
`MV_ORDINAL_OF(arr,expr)`	返回数组中expr第一次出现的基于1的索引，或 `-1` 或 `null` 。如果 `druid.generic.useDefaultValueForNull=false` 如果数组中不存在匹配元素。
`MV_PREPEND(expr,arr)`	在开头将expr添加到arr，结果数组类型由数组类型决定
`MV_APPEND(arr,expr)`	将expr追加到arr，结果数组类型由第一个数组的类型决定
`MV_CONCAT(arr1,arr2)`	连接2个数组，结果数组类型由第一个数组的类型决定
`MV_SLICE(arr,start,end)`	将arr的子数组从基于0的索引start（inclusive）返回到end（exclusive），如果start小于0，大于arr的长度或小于end，则返回空
`MV_TO_STRING(arr,str)`	用str指定的分隔符连接arr的所有元素
`STRING_TO_MV(str1,str2)`	将str1拆分为str2指定的分隔符上的数组

查询转换

在运行之前，Druid SQL将SQL查询转换为 原生查询，理解这种转换是获得良好性能的关键。

最佳实践

在研究如何将SQL查询转换为原生查询的性能影响时，请考虑以下（非详尽）要注意的事项列表。

1. 如果在主时间列 `__time` 上写了一个过滤器，需要确保可以正确的转换为原生的 `"interval"` 过滤器，如下边部分中描述的 时间过滤器。否则，您可能需要更改编写筛选器的方式。
2. 尽量避免连接后的子查询：它们会影响性能和可伸缩性。这包括由不匹配类型上的条件生成的隐式子查询，以及由使用表达式引用右侧的条件生成的隐式子查询。
3. 阅读 查询执行页面，了解如何执行各种类型的原生查询。
4. 解释**执行计划**输出时要小心，如果有疑问，请使用请求日志记录。请求日志将显示运行的确切原生查询。有关更多详细信息，请参见 下一节。
5. 如果您遇到一个可以计划得更好的查询，可以在 GitHub上提出一个问题。一个可重复的测试用例总是值得赞赏的。

解释EXPLAIN PLAN输出

EXPLAIN PLAN功能可以帮助您理解如何将给定的SQL查询转换为原生查询。对于不涉及子查询或联接的简单查询，EXPLAIN PLAN的输出易于解释。将运行的原生查询作为JSON嵌入到"DruidQueryRel"行中：

```
> EXPLAIN PLAN FOR SELECT COUNT(*) FROM wikipedia

DruidQueryRel(query=[{"queryType":"timeseries","dataSource":"wikipedia","inter
```

对于涉及子查询或联接的更复杂查询，解释计划稍微更难解释。例如，考虑以下查询：

```
> EXPLAIN PLAN FOR
> SELECT
>     channel,
>     COUNT(*)
> FROM wikipedia
> WHERE channel IN (SELECT page FROM wikipedia GROUP BY page ORDER BY COUNT(*)
> GROUP BY channel

DruidJoinQueryRel(condition=[=($1, $3)], joinType=[inner], query=[{"queryType"
  DruidQueryRel(query=[{"queryType":"scan","dataSource":{"type":"table","name"
  DruidQueryRel(query=[{"queryType":"topN","dataSource":{"type":"table","name"
```

这里，有一个带有两个输入的连接。阅读这篇文章的方法是将EXPLAIN计划输出的每一行看作可能成为一个查询，或者可能只是一个简单的数据源。它们都拥有的 `query` 字段称为"部分查询"，并表示如果该行本身运行，将在该行所表示的数据源上运行的查询。在某些情况下，比如本例第二行中的"scan"查询，查询实际上并没有运行，最终被转换为一个简单的表数据源。有关如何工作的更多详细信息，请参见 Join转换 部分

我们可以使用Druid的 请求日志功能 看到这一点。在启用日志记录并运行此查询之后，我们可以看到它实际上作为以下原生查询运行。

```
{
  "queryType": "groupBy",
  "dataSource": {
    "type": "join",
    "left": "wikipedia",
    "right": {
      "type": "query",
      "query": {
        "queryType": "topN",
        "dataSource": "wikipedia",
        "dimension": {"type": "default", "dimension": "page", "outputName": "d
        "metric": {"type": "numeric", "metric": "a0"},
        "threshold": 10,
        "intervals": "-146136543-09-08T08:23:32.096Z/146140482-04-24T15:36:27.
        "granularity": "all",
        "aggregations": [
          { "type": "count", "name": "a0"}
        ]
      }
    },
    "rightPrefix": "j0.",
    "condition": "(\"page\" == \"j0.d0\")",
    "joinType": "INNER"
  },
  "intervals": "-146136543-09-08T08:23:32.096Z/146140482-04-24T15:36:27.903Z",
  "granularity": "all",
  "dimensions": [
    {"type": "default", "dimension": "channel", "outputName": "d0"}
  ],
  "aggregations": [
    { "type": "count", "name": "a0"}
  ]
}
```

查询类型

Druid SQL使用四种不同的原生查询类型。

- Scan 操作被用来做不进行聚合的查询（非GroupBy和DISTINCT）
- Timeseries 操作被用来查询GROUP BY `FLOOR(__time TO <unit>)` 或者 `TIME_FLOOR(__time, period)`，不再有其他分组表达式，也没有HAVING或者LIMIT子句，没有嵌套，要么是没有ORDER BY、要么是有与GROUP BY表达式相同的ORDER BY。它还将Timeseries用于具有聚合函数但没有分组依据的"总计"查询。这种查询类型利用了Druid段是按时间排序的这一事实。
- TopN 默认情况下用于按单个表达式分组、具有ORDER BY和LIMIT子句、没有HAVING子句和不嵌套的查询。但是，在某些情况下，TopN查询类型将提供近似的排名和结果；如果要避免这种情况，请将"useApproximateTopN"设置为"false"。TopN结果总是在内存中计算的。有关详细信息，请参阅TopN文档。
- GroupBy 用于所有其他聚合，包括任何嵌套的聚合查询。Druid的GroupBy是一个传统的聚合引擎：它提供精确的结果和排名，并支持多种功能。GroupBy可以在内存中聚合，但如果没有足够的内存来完成查询，它可能会溢出到磁盘。如果您在GROUP BY子句中使用相同的表达式进行ORDER BY，或者根本没有ORDER BY，则结果将通过Broker从数据进程中流回。如果查询具有未出现在GROUP BY子句（如聚合函数）中的ORDER BY引用表达式，则Broker将在内存中具体化结果列表，最大值不超过LIMIT（如果有的话）。有关优化性能和内存使用的详细信息，请参阅GroupBy文档。

时间过滤器

对于所有原生查询类型，只要有可能，`__time` 列上的过滤器将被转换为顶级查询的"interval"，这允许Druid使用其全局时间索引来快速调整必须扫描的数据集。请考虑以下（非详尽）时间过滤器列表，这些时间过滤器将被识别并转换为"intervals"：

- `__time >= TIMESTAMP '2000-01-01 00:00:00'` (与绝对时间相比)
- `__time >= CURRENT_TIMESTAMP - INTERVAL '8' HOUR` (与相对时间相比)
- `FLOOR(__time TO DAY) = TIMESTAMP '2000-01-01 00:00:00'` (指定的一天)

请参阅 解释执行计划输出 部分，以了解有关确认时间筛选器按预期翻译的详细信息。

连接

SQL连接运算符转换为原生连接数据源，如下所示：

1. 原生层可以直接处理的连接将被逐字翻译为 join数据源，其 `left` 、 `right` 和 `condition` 是原始SQL的直接翻译。这包括任何SQL连接，其中右边是 `lookup` 或 `子查询` ，条件是等式，其中一边是基于左边表的表达式，另一边是对右边表的简单列引用，等式的两边是相同的数据类型。
2. 如果一个连接不能够被直接处理为原生的 join数据源, Druid SQL将插入一个子查询使得其可运行。 例如： `foo INNER JOIN bar ON foo.abc = LOWER(bar.def)` 因为右边是一个表达式而非简单的列引用，所以不能够被直接转换，这时会插入一个子查询有效的转换为 `INNER JOIN (SELECT LOWER(def) AS def FROM bar) t ON foo.abc = t.def`
3. Druid SQL目前不重新排序连接以优化查询。

请参阅 解释执行计划输出部分，以了解有关确认连接是否按预期转换的详细信息。

有关如何执行连接操作的信息，请参阅 查询执行页。

子查询

SQL中的子查询一般被转换为原生的查询数据源。有关如何执行子查询操作的信息，请参阅 查询执行页。

> [!WARNING] WHERE子句中的子查询，如： `WHERE col1 IN (SELECT foo FROM ...)` ，被转化为内连接

近似

Druid SQL在一些场景中使用近似算法：

- 默认情况下， `COUNT(DISTINCT col)` 聚合函数使用 HyperLogLog 的变体，HyperLogLog是一种快速近似的DISTINCT计数算法。如果通过查询上下文或通过Broker配置将"useApproximateCountDistinct"设置为"false"，Druid SQL将切换到精确计数
- 对于具有ORDER BY和LIMIT的单列GROUP BY查询，可以采用使用了近似算法的TopN引擎执行查询。如果通过查询上下文或通过Broker配置将"useApproximateTopN"设置为"false"，Druid SQL将切换到精确的分组算法

- 标记为使用草图或近似（例如近似计数不同）的聚合函数不管配置如何,始终是近似的

不支持的特性

Druid SQL并非支持所有的SQL特性。 以下特性不支持：

- 原生数据源（table, lookup, subquery）与系统表的JOIN操作
- 左侧和右侧的表达式之间不相等的JOIN条件
- OVER子句， `LAG` 和 `LEAD` 等分析型函数
- OFFSET子句
- DDL和DML
- 在 元数据表 上使用Druid特性的函数，比如 `TIME_PARSE` 和 `APPROX_QUANTILE_DS`

另外，一些Druid原生查询中的特性目前还不被SQL支持。 不支持的特性如下：

- UNION数据源
- INLINE数据源
- 空间过滤器
- 查询取消

客户端API

HTTP POST

在Druid SQL查询中，可以通过HTTP方式发送POST请求到 `/druid/v2/sql` 来执行SQL查询。该请求应该是一个带有 "query" 字段的JSON对象，例如： `{"query" : "SELECT COUNT(*) FROM data_source WHERE foo = 'bar'"}`

Request

属性	描述	默认值
`query`	SQL	必填，无
`resultFormat`	查询结果的格式，详情查看下边的response部分	object
`header`	是否包含一个请求头，详情查看下边的response部分	false
`context`	包括 连接上下文 参数JSON对象	{}(空)
`parameters`	参数化查询的查询参数列表。列表中的每个参数都应该是一个JSON对象，比如 `{"type": "VARCHAR", "value": "foo"}` 。 `type` 应为SQL类型；有关支持的SQL类型的列表，请参见 数据类型	

可以在命令行中使用 *curl* 来发送SQL查询：

```
$ cat query.json
{"query":"SELECT COUNT(*) AS TheCount FROM data_source"}

$ curl -XPOST -H'Content-Type: application/json' http://BROKER:8082/druid/v2/s
[{"TheCount":24433}]
```

可以提供一个"context"的参数来添加 连接上下文 变量，例如：

```
{
  "query" : "SELECT COUNT(*) FROM data_source WHERE foo = 'bar' AND __time > T
  "context" : {
    "sqlTimeZone" : "America/Los_Angeles"
  }
}
```

参数化SQL查询也是支持的：

```
{
  "query" : "SELECT COUNT(*) FROM data_source WHERE foo = ? AND __time > ?",
  "parameters": [
    { "type": "VARCHAR", "value": "bar"},
    { "type": "TIMESTAMP", "value": "2000-01-01 00:00:00" }
  ]
}
```

通过对 元数据表 进行HTTP POST请求可以获得元数据

Responses

Druid SQL的HTTP POST API支持一个可变的结果格式，可以通过"resultFormat"参数来指定，例如：

```
{
  "query" : "SELECT COUNT(*) FROM data_source WHERE foo = 'bar' AND __time > T
  "resultFormat" : "object"
}
```

支持的结果格式为：

格式	描述	Content-Type
`object`	默认值，JSON对象的JSON数组。每个对象的字段名都与SQL查询返回的列匹配，并且按与SQL查询相同的顺序提供。	application/json
`array`	JSON数组的JSON数组。每个内部数组按顺序都有与SQL查询返回的列匹配的元素。	application/json
`objectLines`	与"object"类似，但是JSON对象由换行符分隔，而不是包装在JSON数组中。如果您没有流式JSON解析器的现成访问权限，这可以使将整个响应集解析为流更加容易。为了能够检测到被截断的响应，此格式包含一个空行的尾部。	text/plain
`arrayLines`	与"array"类似，但是JSON数组由换行符分隔，而不是包装在JSON数组中。如果您没有流式JSON解析器的现成访问权限，这可以使将整个响应集解析为流更加容易。为了能够检测到被截断的响应，此格式包含一个空行的尾部。	text/plain
`csv`	逗号分隔的值，每行一行。单个字段值可以用双引号括起来进行转义。如果双引号出现在字段值中，则通过将它们替换为双引号（如 `""this""` ）来对其进行转义。为了能够检测到被截断的响应，此格式包含一个空行的尾部。	text/csv

您还可以通过在请求中将"header"设置为true来请求头，例如：

```
{
  "query" : "SELECT COUNT(*) FROM data_source WHERE foo = 'bar' AND __time > T
  "resultFormat" : "arrayLines",
  "header" : true
}
```

在这种情况下，返回的第一个结果将是头。对于 `csv` 、 `array` 和 `arrayline` 格式，标题将是列名列表。对于 `object` 和 `objectLines` 格式，头将是一个对象，其中键是列名，值为空。

在发送响应体之前发生的错误将以JSON格式报告，状态代码为HTTP 500，格式与原生Druid查询错误 相同。如果在发送响应体时发生错误，此时更改HTTP状态代码或报告JSON错误已经太迟，因此响应将简单地结束流，并且处理您的请求的Druid服务器将记录一个错误。

作为调用者，正确处理响应截断非常重要。这对于"object"和"array"格式很容易，因为的截断响应将是无效的JSON。对于面向行的格式，您应该检查它们都包含的尾部：结果集末尾的一个空行。如果通过JSON解析错误或缺少尾随的换行符检测

到截断的响应，则应假定响应由于错误而未完全传递。

JDBC

您可以使用 Avatica JDBC Driver 来进行Druid SQL查询。 下载Avatica客户端jar包后加到类路径下，使用如下连接串：

`jdbc:avatica:remote:url=http://BROKER:8082/druid/v2/sql/avatica/`

示例代码为：

```
// Connect to /druid/v2/sql/avatica/ on your Broker.
String url = "jdbc:avatica:remote:url=http://localhost:8082/druid/v2/sql/avati

// Set any connection context parameters you need here (see "Connection contex
// Or leave empty for default behavior.
Properties connectionProperties = new Properties();

try (Connection connection = DriverManager.getConnection(url, connectionProper
  try (
      final Statement statement = connection.createStatement();
      final ResultSet resultSet = statement.executeQuery(query)
  ) {
    while (resultSet.next()) {
      // Do something
    }
  }
}
```

表的元数据信息在JDBC中也是可以查询的，通过 `connection.getMetaData()` 或者查询 信息Schema

连接粘性

Druid的JDBC服务不在Broker之间共享连接状态。这意味着，如果您使用JDBC并且有多个Druid Broker，您应该连接到一个特定的Broker，或者使用启用了粘性会话的负载平衡器。Druid Router进程在平衡JDBC请求时提供连接粘性，即使使用普通的非粘性负载平衡器，也可以用来实现必要的粘性。请参阅 Router文档 以了解更多详细信息

注意：非JDBC的 HTTP POST 是无状态的，不需要粘性

动态参数

在JDBC代码中也可以使用参数化查询，例如：

```
PreparedStatement statement = connection.prepareStatement("SELECT COUNT(*) AS
statement.setString(1, "abc");
statement.setString(2, "def");
final ResultSet resultSet = statement.executeQuery();
```

连接上下文

Druid SQL支持在客户端设置连接参数，下表中的参数会影响SQL执行。您提供的所有其他上下文参数都将附加到Druid查询，并可能影响它们的运行方式。关于可能选项的详情可以参见 查询上下文

请注意，要为SQL查询指定唯一标识符，请使用 `sqlQueryId` 而不是 `queryId` 。为SQL请求设置 `queryId` 没有任何效果，所有SQL底层的原生查询都将使用自动生成的queryId。

连接上下文可以指定为JDBC连接属性，也可以指定为JSON API中的 "context" 对象。

参数	描述	
`sqlQueryId`	本次SQL查询的唯一标识符。对于HTTP客户端，它在 `X-Druid-SQL-Query-Id` 头中返回	自动生成
`sqlTimeZone`	设置此连接的时区，这将影响时间函数和时间戳文本的行为。应该是时区名称，如"America/Los_Angeles"或偏移量，如"-08:00"	Broker配置 默认为：U
`useApproximateCountDistinct`	是否对 `COUNT(DISTINCT foo)` 使用近似基数算法	Broker配置 `druid.sql.` 默认为：tru
`useApproximateTopN`	当SQL查询可以这样表示时，是否使用近似TopN查询。如果为false，则将使用精确的 GroupBy查询。	Broker配置 `druid.sql.` 为：true

元数据表

Druid Broker从集群中加载的段推断每个数据源的表和列元数据，并使用此来计划SQL查询。该元数据在Broker启动时缓存，并通过 段元数据查询 在后台定期更新。后台元数据刷新由进入和退出集群的段触发，也可以通过配置进行限制。

Druid通过特殊的系统表公开系统信息。有两种schema可用：Information Schema和Sys Schema。Information Schema提供有关表和列类型的详细信息, Sys Schema提供了有关Druid内部的信息，比如段/任务/服务器。

INFORMATION SCHEMA

您可以使用JDBC连接 `connection.getMetaData()` 访问表和列元数据，或通过下面描述的INFORMATION_SCHEMA表。例如，要检索Druid数据源"foo"的元数据，请使用查询：

```
SELECT * FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_SCHEMA = 'druid' AND TABL
```

> [!WARNING] INFORMATION_SCHEMA表目前不支持Druid特定的函数，如： `TIME_PARSE` 和 `APPROX_QUANTILE_DS` , 只有标准的SQL函数是可用的。

SCHEMATA表

Column	Notes
CATALOG_NAME	Unused
SCHEMA_NAME	
SCHEMA_OWNER	Unused
DEFAULT_CHARACTER_SET_CATALOG	Unused
DEFAULT_CHARACTER_SET_SCHEMA	Unused
DEFAULT_CHARACTER_SET_NAME	Unused
SQL_PATH	Unused

TABLES表

Column	Notes
TABLE_CATALOG	Unused
TABLE_SCHEMA	
TABLE_NAME	
TABLE_TYPE	"TABLE" or "SYSTEM_TABLE"

COLUMNS表

Column	Notes
TABLE_CATALOG	Unused
TABLE_SCHEMA	
TABLE_NAME	
OLUMN_NAME	
ORDINAL_POSITION	
COLUMN_DEFAULT	Unused
IS_NULLABLE	
DATA_TYPE	
CHARACTER_MAXIMUM_LENGTH	Unused
CHARACTER_OCTET_LENGTH	Unused
NUMERIC_PRECISION	
NUMERIC_PRECISION_RADIX	
NUMERIC_SCALE	
DATETIME_PRECISION	
CHARACTER_SET_NAME	
COLLATION_NAME	
JDBC_TYPE	Type code from java.sql.Types (Druid extension)

SYSTEM SCHEMA

"sys"模式提供了对Druid段、服务器和任务的可见性。

> [!WARNING] 注意： "sys"表当前不支持Druid特定的函数，例如 `TIME_PARSE` 和 `APPROX_QUANTILE_DS` 。仅仅标准SQL函数可以被使用。

SEGMENTS表

segments表提供了所有Druid段的详细信息，无论该段是否被发布

字段	类型	注意
`segment_id`	STRING	唯一的段标识符
`datasource`	STRING	数据源名称
`start`	STRING	Interval开始时间（ISO8601格式）
`end`	STRING	Interval结束时间（ISO8601格式）
`size`	LONG	段大小，单位为字节
`version`	STRING	版本字符串（通常是ISO8601时间戳，对应于段集首次启动的时间）。较高的版本意味着最近创建的段。版本比较基于字符串比较。
`partition_num`	LONG	分区号（整数，在数据源+间隔+版本中是唯一的；不一定是连续的）
`num_replicas`	LONG	当前正在服务的此段的副本数
`num_rows`	LONG	当前段中的行数，如果查询时Broker未知，则此值可以为空
`is_published`	LONG	布尔值表示为long类型，其中1=true，0=false。1表示此段已发布到元数据存储且 `used=1` 。详情查看 架构页面
`is_available`	LONG	布尔值表示为long类型，其中1=true，0=false。1表示此段当前由任何进程（Historical或Realtime）提供服务。详情查看 架构页面
`is_realtime`	LONG	布尔值表示为long类型，其中1=true，0=false。如果此段仅由实时任务提供服务，则为1；如果任何Historical进程正在为此段提供服务，则为0。
`is_overshadowed`	LONG	布尔值表示为long类型，其中1=true，0=false。如果此段已发布，并且被其他已发布的段完全覆盖则为1。目前，对于未发布的段，`is_overshadowed` 总是false，尽管这在未来可能会改变。可以通过过滤 `is_published=1` 和 `is_overshadowed=0` 来筛选"应该发布"的段。如果段最近被替换，它们可以短暂地被发布，也可以被掩盖，但还没有被取消发布。详情查看 架构页面
`payload`	STRING	JSON序列化数据段负载

例如，要检索数据源"wikipedia"的所有段，请使用查询：

```
SELECT * FROM sys.segments WHERE datasource = 'wikipedia'
```

另一个检索每个数据源的段总大小、平均大小、平均行数和段数的示例：

```
SELECT
    datasource,
    SUM("size") AS total_size,
    CASE WHEN SUM("size") = 0 THEN 0 ELSE SUM("size") / (COUNT(*) FILTER(WHERE
    CASE WHEN SUM(num_rows) = 0 THEN 0 ELSE SUM("num_rows") / (COUNT(*) FILTER
    COUNT(*) AS num_segments
FROM sys.segments
GROUP BY 1
ORDER BY 2 DESC
```

注意：请注意，一个段可以由多个流摄取任务或Historical进程提供服务，在这种情况下，它将有多个副本。当由多个摄取任务提供服务时，这些副本彼此之间的一致性很弱，直到某个片段最终由一个Historical段提供服务，此时该段是不可变的。Broker更喜欢从Historical中查询段，而不是从摄取任务中查询段。但如果一个段有多个实时副本，例如Kafka索引任务，同时一个任务比另一个慢，然后 `sys.segments` 结果在任务的持续时间内可能会有所不同，因为Broker只查询一个摄取任务，并且不能保证每次都选择相同的任务。段表的 `num_rows` 列在此期间可能有不一致的值。关于与流摄取任务的不一致性，有一个 公开问题 。

SERVERS表

Servers表列出集群中发现的所有服务器

字段	类型	注意
`server`	STRING	服务名称，格式为host:port
`host`	STRING	服务的hostname
`plaintext_port`	LONG	服务器的不安全端口，如果禁用明文通信，则为-1
`tls_port`	LONG	服务器的TLS端口，如果禁用了TLS，则为-1
`server_type`	STRING	Druid服务的类型，可能的值包括：COORDINATOR, OVERLORD, BROKER, ROUTER, HISTORICAL, MIDDLE_MANAGER 或者 PEON
`tier`	STRING	分布层，查看 druid.server.tier。仅对Historical有效，对于其他类型则为null
`current_size`	LONG	此服务器上以字节为单位的段的当前大小。仅对Historical有效，对于其他类型则为0
`max_size`	LONG	此服务器建议分配给段的最大字节大小，请参阅 druid.server.maxSize 文件, 仅对Historical有效，对于其他类型则为0

要检索有关所有服务器的信息，请使用查询：

```
SELECT * FROM sys.servers;
```

SERVER_SEGMENTS表

SERVER_SEGMENTS表用来连接服务与段表

字段	类型	注意
`server`	STRING	格式为host:port的服务名称， SERVER表 的主键
`segment_id`	STRING	段标识符，SEGMENTS表 的主键

"SERVERS"和"SEGMENTS"之间的联接可用于查询特定数据源的段数，按SERVER分组，示例查询：

```
SELECT count(segments.segment_id) as num_segments from sys.segments as segment
INNER JOIN sys.server_segments as server_segments
ON segments.segment_id  = server_segments.segment_id
INNER JOIN sys.servers as servers
ON servers.server = server_segments.server
WHERE segments.datasource = 'wikipedia'
GROUP BY servers.server;
```

TASKS表

"TASKS"表提供有关活跃的和最近完成的索引任务的信息。有关更多信息，请查看摄取任务的文档。

字段	类型	注意
task_id	STRING	唯一的任务标识符
group_id	STRING	本任务的任务组ID，值依赖于任务的 type，例如，对于原生索引任务， 它与 task_id 相同，对于子任务，该值为父任务的ID
type	STRING	任务类型，例如该值为"index"表示为索引任务。 可以查看 任务概述
datasource	STRING	被索引的数据源名称
created_time	STRING	ISO8601格式的时间戳，与创建摄取任务的时间相对应。请注意，此值是为已完成和正在等待的任务填充的。对于正在运行和挂起的任务，此值设置为1970-01-01T00:00:00Z
queue_insertion_time	STRING	ISO8601格式的时间戳，与此任务添加到Overlord上的队列时对应
status	STRING	任务状态，可以是RUNNING、FAILED、SUCCESS
runner_status	STRING	对于已完成任务的运行状态为NONE，对于进行中的任务，值可以为RUNNING、WAITING、PENDING
duration	LONG	完成任务所用的时间（毫秒），此值仅对已完成的任务显示
location	STRING	运行此任务的服务器名称，格式为主机：端口，此信息仅对正在运行的任务显示
host	STRING	运行此任务的服务器名称
plaintext_port	LONG	服务器的不安全端口，如果明文通信被禁用，则为-1
tls_port	LONG	服务器的TLS端口，如果TLS被禁用，则为-1
error_msg	STRING	FAILED任务的详细错误信息

例如，要检索按状态筛选的任务信息，请使用查询

```
SELECT * FROM sys.tasks WHERE status='FAILED';
```

SUPERVISORS表

SUPERVISORS表提供supervisor的详细信息

字段	类型	注意
`supervisor_id`	STRING	supervisor任务的标识符
`state`	STRING	supervisor的基本状态，可用状态有：`UNHEALTHY_SUPERVISOR`，`UNHEALTHY_TASKS`，`PENDING`，`RUNNING`，`SUSPENDED`，`STOPPING`。详情可以查看Kafka摄取文档
`detailed_state`	STRING	supervisor特定的状态。(详情查看特定的supervisor状态的文档)
`healthy`	LONG	布尔值表示为long类型，其中1=true，0=false。1表示supervisor健康
`type`	STRING	supervisor的类型，例如 `kafka`，`kinesis` 或者 `materialized_view`
`source`	STRING	supervisor的源，例如 Kafka Topic或者 Kinesis Stream
`suspended`	LONG	布尔值表示为long类型，其中1=true，0=false。1表示supervisor处于暂停状态
`spec`	STRING	JSON序列化的supervisor说明

例如：要检索按运行状况筛选的supervisor任务信息，请使用查询：

```
SELECT * FROM sys.supervisors WHERE healthy=0;
```

服务配置

Druid SQL计划发生在Broker上，由 Broker runtime properties 配置。

安全性

有关进行SQL查询需要哪些权限的信息，请参阅基本安全文档中的 定义SQL权限 。

原生查询

> [!WARNING] Apache Druid支持两种查询语言，Druid SQL 和 原生查询，该文档描述原生查询语言。有关Druid SQL如何选择运行SQL查询时要使用的原生查询类型的信息，请查看 SQL文档。

Druid中的原生查询是JSON对象，通常发送给Broker或Router进程。查询可以这样发布：

```
curl -X POST '<queryable_host>:<port>/druid/v2/?pretty' -H 'Content-Type:appli
```

Druid的原生查询语言是JSON over HTTP，尽管社区的许多成员已经用其他语言提供了不同的 客户端库 来查询Druid。

Content-Type/Accept Header也可以采用"application/x-jackson-smile":

```
curl -X POST '<queryable_host>:<port>/druid/v2/?pretty' -H 'Content-Type:appli
```

注意：如果未提供Accept header，则默认为"Content Type" header的值。

Druid的原生查询级别相对较低，与内部执行计算的方式密切相关。Druid查询被设计成轻量级的，并且非常快速地完成。这意味着对于更复杂的分析，或者构建更复杂的可视化，可能需要多个Druid查询。

即使查询通常是向Broker或Router发出的，但是它们也可以被 Historical进程 和运行流摄取任务的 peon(任务jvm) 接受。如果您想查询由特定进程提供服务的特定段的结果，这可能很有价值。

可用的查询

Druid有许多不同场景的查询类型。查询由各种JSON属性组成，Druid针对不同的场景有不同类型的查询。各种查询类型的文档描述了可以设置的所有JSON属性。

聚合查询

- Timeseries
- TopN
- GroupBy

元数据查询

- TimeBoundary
- SegmentMetadata
- DatasourceMetadata

其他查询

- Scan
- Search

应该使用哪种类型的查询

对于聚合查询，如果有多个查询可以满足您的需求，我们通常建议尽可能使用Timeseries或TopN，因为Druid针对它们的场景做了专门优化的。如果两者都不适合，则应该使用GroupBy查询，这是最灵活的。

查询取消

可以使用查询的唯一标识符显式地取消查询。如果查询标识符是在查询时设置的，或者是已知的，那么可以在Broker或Router上使用以下接口来取消查询。

```
DELETE /druid/v2/{queryId}
```

例如：如果查询ID是 `abc123`，查询可以如下取消：

```
curl -X DELETE "http://host:port/druid/v2/abc123"
```

查询错误

如果查询失败，您将得到一个HTTP 500响应，其中包含具有以下结构的JSON对象：

```
{
  "error" : "Query timeout",
  "errorMessage" : "Timeout waiting for task.",
  "errorClass" : "java.util.concurrent.TimeoutException",
  "host" : "druid1.example.com:8083"
}
```

如果查询请求由于受到 query scheduler laning configuration 的限制而失败，则为HTTP 429响应，该响应具有与错误响应相同的JSON对象架构，但 `errorMessage` 格式为："Total query capacity exceeded"或"query capacity exceeded for lane 'low'"。

响应中的字段是:

字段	描述
`error`	定义明确的错误代码（如下表格）
`errorMessage`	包含有关错误的详细信息的自由格式消息。可能为空。
`errorClass`	导致此错误的异常的类。可能为空。
`host`	发生此错误的主机。可能为空。

可能的错误码字段如下：

错误码	描述
`Query timeout`	查询超时
`Query interrupted`	查询被中断，可能是因为JVM关闭
`Query cancelled`	查询通过"查询取消API"取消
`Resource limit exceeded`	查询超出了配置的资源限制（例如groupBy maxResults）
`Unauthorized request`	由于安全策略，查询被拒绝。用户已被识别，但未识别出用户的访问权限。
`Unsupported operation`	查询试图执行不受支持的操作。当使用未记录的功能或使用未完全实现的扩展时，可能会发生这种情况
`Unknown exception`	发生了其他异常。请检查errorMessage和errorClass以获取详细信息，但请记住，这些字段的内容是自由格式的，并且可能会随着版本的不同而变化

查询执行

> [!WARNING] 本文档描述了Druid如何执行 原生查询，但是由于 Druid SQL 查询被转换为原生查询，因此本文档也适用于SQL运行时。有关如何将SQL查询转换为原生查询的信息，请参阅 SQL查询转换 页面。

Druid的查询执行方法因查询的 数据源类型 而异。

数据源类型

`table`

直接在 表数据源 上操作的查询使用由Broker进程引导的**分散-聚集**方法执行。过程如下：

1. Broker根据 `"interval"` 参数确定哪些 段 与查询相关。段总是按时间划分的，因此任何间隔与查询间隔重叠的段都可能是相关的。
2. 如果输入数据使用 `single_dim` partitionsSpec 按范围分区，并且过滤器与用于分区的维度匹配，则Broker还可以根据 `"filter"` 进一步修剪段列表。
3. Broker在删除了查询的段列表之后，将查询转发到当前为这些段提供服务的数据服务器（如Historical或者运行在MiddleManagers的任务）。
4. 对于除 Scan 之外的所有查询类型，数据服务器并行处理每个段，并为每个段生成部分结果。所做的具体处理取决于查询类型。如果启用了 查询缓存，则可以缓存这些部分结果。对于Scan查询，段由单个线程按顺序处理，结果不被缓存。
5. Broker从每个数据服务器接收部分结果，将它们合并到最终结果集中，并将它们返回给调用方。对于Timeseries和Scan查询，以及没有排序的GroupBy查询，Broker可以以流式方式执行此操作。否则，Broker将在返回任何内容之前完全计算结果集。

`lookup`

直接对 Lookup数据源(没有联接) 进行操作的查询使用查询的本地副本在接收查询的Broker上执行。所有注册的Lookup表都预加载到Broker的内存中。查询运行单线程。

使用Lookup作为联接的右端输入的查询的执行是以依赖于其"base"（最左下角）数据源的方式执行的，如下面的 join 部分所述。

`union`

直接在 union数据源 上操作的查询在Broker上被拆分为属于union的每个表的单独查询。这些查询中的每一个都单独运行，Broker将它们的结果合并在一起。

`inline`

直接在 内联数据源 上操作的查询在接收查询的Broker上执行。查询运行单线程。

使用内联数据源作为联接的右端输入的查询的执行方式取决于它们的"base"（最左下角）数据源，如下面的 join 部分所述。

`query`

[query数据源] 是子查询, 每个子查询都被当作它自己的查询来执行, 结果会返回给Broker。然后，Broker继续处理查询的其余部分，就像子查询被内联数据源替换一样。

在大多数情况下，子查询结果在其余查询继续之前在Broker上的内存中完全缓冲，这意味着子查询按顺序执行。以这种方式在给定查询的所有子查询中缓冲的行总数不能超过 druid.server.http.maxSubQueryRows 属性。

有一个例外：如果外部查询和所有子查询都是 groupBy 类型，则可以以流式方式处理子查询结果，并且 `druid.server.http.maxSubQueryRows` 限制不适用。

`join`

使用广播hash-join方法处理 联接数据源。

1. Broker执行输入join的任何子查询，如 `query` 部分所述，并用inline数据源替换它们。
2. Broker将连接树（如果存在）展平为"基本"数据源（最左下角的一个）和其他叶数据源（其余部分）。
3. 使用与基本数据源相同结构的查询执行将单独继续执行。如果基数据源是一个表，则像往常一样根据"interval"修剪段，并通过将查询并行地转发到所有相关的数据服务器，在集群上执行查询。如果基数据源是Lookup或Join数据源（包括将内联子查询作为结果的内联数据源），则查询将在Broker本身上执行。基查询不能是Union，因为当前不支持Union作为Join的输入。
4. 在开始处理基数据源之前，将执行查询的服务器首先检查所有非基叶数据源，以确定是否需要为即将到来的哈希联接生成新的哈希表。目前，Lookup不需要构建新的哈希表（因为它们是预加载的），但Inline数据源需要。
5. 使用与基本数据源相同结构的查询执行将单独继续执行, 但有一个附加条件：在处理基本数据源时，Druid服务器将使用从其他连接输入构建的哈希表来逐行生成联接结果，查询引擎将对联接的行而不是基本行进行操作。

数据源

在Apache Druid中，数据源是被查询的对象。 最常见的数据源类型是一个表数据源，本文档在很多场景中"dataSource"就是指代表数据源，尤其是在 数据摄取 部分中，在数据摄取中，总是创建一个表数据源或者往表数据源中写入数据。但是在查询时，有许多种类型的数据源可用。

出现在API请求和响应中的"datasource"一般拼写为 `dataSource` ，注意是大写的S。

数据源类型

`table`

SQL

```
SELECT column1, column2 FROM "druid"."dataSourceName"
```

原生

```
{
  "queryType": "scan",
  "dataSource": "dataSourceName",
  "columns": ["column1", "column2"],
  "intervals": ["0000/3000"]
}
```

表数据源是最常见的类型，该类数据源可以在 数据摄取 后获得。它们被分成若干段，分布在集群中，并且并行地进行查询。

在 Druid SQL 中，表数据源位于 `druid` schema中。 这是默认schema，表数据源可以被指定为 `druid.dataSourceName` 或者简单的 `dataSourceName`

在原生查询中，可以使用表数据源的名称作为字符串（如上面的示例中所示）或使用以下形式的JSON对象来引用表数据源：

```
"dataSource": {
  "type": "table",
  "name": "dataSourceName"
}
```

为了看到所有的表数据源列表，可以通过SQL查询 `SELECT * FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_SCHEMA = 'druid'`

`lookup`

SQL

```
SELECT k, v FROM lookup.countries
```

原生

```
{
  "queryType": "scan",
  "dataSource": {
    "type": "lookup",
    "lookup": "countries"
  },
  "columns": ["k", "v"],
  "intervals": ["0000/3000"]
}
```

Lookup数据源对应于Druid的键值 lookup 对象。在Druid SQL 中，它们驻留在 `lookup` schema中。它们会被预加载到所有服务器的内存中，因此可以快速访问它们。可以使用 join运算符 将它们连接到常规表上。

Lookup数据源是面向键值的，总是正好有两列： `k` （键）和 `v` （值），而且这两列始终是字符串。

要查看所有的Lookup数据源的列表，请使用SQL查询 `SELECT * FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_SCHEMA='lookup'` 。

> [!WARNING] 性能提示：Lookup可以通过显式联接或使用SQL LOOKUP函数 与基表联接。但是，join运算符必须对每一行计算条件，而 `LOOKUP` 函数可以将计算推迟到聚合阶段之后。这意味着 `LOOKUP` 函数通常比联接到lookup数据源要快

有关使用表数据源时如何执行查询的更多详细信息，请参阅 查询执行 页面。

`union`

原生

```
{
  "queryType": "scan",
  "dataSource": {
    "type": "union",
    "dataSources": ["<tableDataSourceName1>", "<tableDataSourceName2>", "<tabl
  },
  "columns": ["column1", "column2"],
  "intervals": ["0000/3000"]
}
```

Union数据源允许您将两个或多个表数据源视为单个数据源。联合的数据源不需要具有相同的结构。如果它们不完全匹配，那么存在于一个表中而不是另一个表中的列将被视为在**它们不存在的表中包含所有空值**。

union数据源在Druid SQL中不可用。

有关使用union数据源时如何执行查询的更多详细信息，请参阅 查询执行 页面。

`inline`

原生

```
{
  "queryType": "scan",
  "dataSource": {
    "type": "inline",
    "columnNames": ["country", "city"],
    "rows": [
      ["United States", "San Francisco"],
      ["Canada", "Calgary"]
    ]
  },
  "columns": ["country", "city"],
  "intervals": ["0000/3000"]
}
```

Inline数据源允许可以查询嵌入在查询体本身中的一小波数据。 当想要查询一小部分数据但是还没有摄取加载时，这是非常有用的，而且可以很有效的用在 join 的输入。 Druid也允许在内部使用它们来操作需要在Broker嵌入的子查询。 详情可以查看 `query` 数据源 的文档。

在Inline数据源中有两个字段，名为 `columnNames` 的数组和名为 `rows` 的二维数组，每一行必须是一个长度与 `columnNames` 同长度的数组。每行中的第一个元素对应于 `columnNames` 中的第一列，依此类推。

Inline数据源目前在Druid SQL中是不可用的。

有关使用Inline数据源时如何执行查询的更多详细信息，请参阅 查询执行 页面。

`query`

SQL

```
-- Uses a subquery to count hits per page, then takes the average.
SELECT
  AVG(cnt) AS average_hits_per_page
FROM
  (SELECT page, COUNT(*) AS hits FROM site_traffic GROUP BY page)
```

原生

```
{
  "queryType": "timeseries",
  "dataSource": {
    "type": "query",
    "query": {
      "queryType": "groupBy",
      "dataSource": "site_traffic",
      "intervals": ["0000/3000"],
      "granularity": "all",
      "dimensions": ["page"],
      "aggregations": [
        { "type": "count", "name": "hits" }
      ]
    }
  },
  "intervals": ["0000/3000"],
  "granularity": "all",
  "aggregations": [
    { "type": "longSum", "name": "hits", "fieldName": "hits" },
    { "type": "count", "name": "pages" }
  ],
  "postAggregations": [
    { "type": "expression", "name": "average_hits_per_page", "expression": "hi
  ]
}
```

Query数据源允许您发出子查询。在原生查询中，它们可以出现在接受 `dataSource` 的任何地方。在SQL中，它们可以出现在以下位置，始终用括号括起来：

- FROM子句： `FROM (<subquery>)`
- 作为JOIN的输入： `<table-or-subquery-1> t1 INNER JOIN <table-or-subquery-2> t2 ON t1.<col1> = t2.<col2>`
- 在WHERE子句中： `WHERE <column> {IN | NOT IN} (<subquery>)` 。在SQL计划器中这些都被转换为joins

> [!WARNING] 性能提示：在大多数情况下，子查询结果在Broker的内存中完全缓冲，然后在Broker本身上进行进一步的处理。这意味着具有大型结果集的子查询可能会导致性能瓶颈或在Broker上遇到内存使用限制。有关使用Query数据源时如何执行查询的更多详细信息，请参阅 查询执行 页面。

`join`

SQL

```
-- Joins "sales" with "countries" (using "store" as the join key) to get sales
SELECT
  store_to_country.v AS country,
  SUM(sales.revenue) AS country_revenue
FROM
  sales
  INNER JOIN lookup.store_to_country ON sales.store = store_to_country.k
GROUP BY
  countries.v
```

原生

```
{
  "queryType": "groupBy",
  "dataSource": {
    "type": "join",
    "left": "sales",
    "right": {
      "type": "lookup",
      "lookup": "store_to_country"
    },
    "rightPrefix": "r.",
    "condition": "store == \"r.k\"",
    "joinType": "INNER"
  },
  "intervals": ["0000/3000"],
  "granularity": "all",
  "dimensions": [
    { "type": "default", "outputName": "country", "dimension": "r.v" }
  ],
  "aggregations": [
    { "type": "longSum", "name": "country_revenue", "fieldName": "revenue" }
  ]
}
```

Join数据源允许您对两个数据源执行SQL样式的联接。相互堆叠连接允许您任意连接多个数据源。

在Druid 0.18.1版本中，joins使用**broadcast hash-join algorithm**来实现，这意味着除了左边的"base"表之外的表都需要在内存中，同时也意味着连接条件必须是等于。 此特性主要用于将常规Druid表与 lookup、inline 和 query 数据源连接起来。

有关使用Join数据源时如何执行查询的更多详细信息，请参阅 查询执行 页面。

SQL中的Joins

SQL中的join格式如下：

```
<o1> [ INNER | LEFT [OUTER] ] JOIN <o2> ON <condition>
```

条件必须是等于，但是函数是可以使用的，而且可以使用多个AND。 像 `t1.x = t2.x` 或者 `LOWER(t1.x) = t2.x` 或者 `t1.x = t2.x AND t1.y = t2.y` 这些都可以被处理。 像 `t1.x <> t2.x` 这样的条件是不可以被处理的。

注意，Druid SQL没有原生join数据源所能处理的严格。在SQL查询执行与原生join数据源不允许的操作的情况下，Druid SQL将生成一个子查询。这可能会对性能和可伸缩性产生重大影响，因此需要注意。SQL层何时生成子查询的示例包括：

- 将一个普通的Druid表连接到自己，或者另一个普通的Druid表。原生Join数据源可以接受左侧的表，但不能接受右侧的表，因此需要子查询。
- 联接条件的两边的表达式属于不同类型。
- 联接条件的右表达式不是可直接访问的列。

对于Druid如何将SQL转化为原生查询的更多信息，可以参考 Druid SQL 文档

原生中的Joins

原生的Join查询需要以下属性，且都是必须的。

字段	描述
`left`	左侧数据源。 类型必须是 `table`，`join`，`lookup`，`query` 或者 `inline`。将另一个join作为左数据源放置允许您任意连接多个数据源。
`right`	右侧数据源。 类型必须是 `lookup`，`query` 或者 `inline`。注意：这一点比Druid SQL更加严格
`rightPrefix`	字符串前缀，将应用于右侧数据源中的所有列，以防止它们与左侧数据源中的列发生冲突。可以是任何字符串，只要它不是空的并且不是 `__time` 字符串的前缀。左侧以 `rightPrefix` 开头的任何列都将被隐藏。您需要提供一个前缀，它不会从左侧隐藏任何重要的列。
`condition`	表达式，该表达式必须是相等的，其中一侧是左侧的表达式，另一侧是对右侧的简单列引用。注意，这比Druid SQL所要求的更严格：这里，右边的引用必须是一个简单的列引用；在SQL中，它可以是一个表达式。
`joinType`	`INNER` 或者 `LEFT`

Join的性能

Join是一个可以显著影响查询性能的功能。一些性能提示和注意事项：

1. Joins与lookup数据源一起使用是非常有用的，但是在大多数场景下，LOOKUP函数 的性能比Join更优。 如果使用场景非常近似的话，可以考虑使用 `LOOKUP`
2. 当在Druid SQL中使用join的时候，它可以生成未在查询中显式指定的子查询。关于何时发生以及何时探测到它可以在 Druid SQL 查看详细信息
3. 一个生成显式子查询的常见原因是：等号两边的类型不一致。 例如：因为lookup的key总是字符串，当 `d.field` 是字符串的时候 `druid.d JOIN lookup.l ON d.field = l.field` 的性能最佳
4. 从druid0.18.1开始，join操作符必须为每一行计算条件。在未来，我们希望实现早期和延迟的条件评估，这将大大提高常见用例的性能。

Join的下一步工作

Join是Druid开发中比较活跃发展的一个领域。目前缺少以下功能，但可能会在将来的版本中出现：

- 比Lookup更宽的预加载维度表（例如支持单个键和多个值）
- RIGHT OUTER和FULL OUTER连接。目前，它们已部分实施，查询会运行，但结果并不总是正确的
- 上一节中提到的与性能相关的优化
- broadcast hash-join以外的连接算法

Joins

Druid有两个与数据连接相关的特性：

1. join 运算符。可以在原生查询中使用 join数据源 ，或者在Druid SQL中使用 join操作符 。有关join在Druid中如何工作的信息，请参阅 join数据源文档。
2. 查询时Lookup，简单的键到值映射。所有涉及查询的服务器上都预加载了这些查询，可以使用或不使用显式join运算符进行访问。有关更多详细信息，请参阅 Lookup 文档。

只要可能，为了获得最佳性能，最好在查询时避免使用Join，通常可以在数据加载到Druid之前通过数据处理阶段来完成。但是，在某些情况下，尽管存在性能开销，但joins或lookups是可用的最佳解决方案，包括：

- fact-to-dimension的情况：您需要在初始摄取之后更改维度值，并且无法重新导入来执行此操作。在这种情况下，可以使用维度表的Lookup。
- 查询需要对子查询进行join或filter。

Lookups

> [!WARNING] Lookups是一个 实验性的特性

Lookups是Apache Druid中的一个概念，在Druid中维度值(可选地)被新值替换，从而允许类似join的功能。在Druid中应用Lookup类似于在数据仓库中的联接维度表。有关详细信息，请参见 维度说明。在这些文档中，"key"是指要匹配的维度值，"value"是指其替换的目标值。所以如果你想把 `appid-12345` 映射到 `Super Mega Awesome App` ，那么键应该是 `appid-12345` ，值就是 `Super Mega Awesome App` 。

值得注意的是，Lookups不仅支持键一对一映射到唯一值（如国家代码和国家名称）的场景，还支持多个ID映射到同一个值的场景，例如多个应用程序ID映射到一个客户经理。当Lookup是一对一的时候，Druid能够在查询时应用额外的优化；有关更多详细信息，请参阅下面的 查询执行。

Lookups没有历史记录，总是使用当前的数据。这意味着，如果特定应用程序id的首席客户经理发生更改，并且您发出了一个查询，其中存储了应用程序id与客户经理之间的关系，则无论您查询的时间范围如何，它都将返回该应用程序id的当前客户经理。

如果您需要进行对数据时间范围敏感的Lookups，那么目前在查询时不支持这样的场景，并且这些数据属于原始的非规范化数据中，以便在Druid中使用。

在所有服务器上，Lookup通常都预加载在内存中。但是，对于非常小的Lookup（大约几十到几百个条目）也可以使用"map"Lookup类型在原生查询时内联传递。有关详细信息，请参见 维度说明。

其他的Lookup类型在扩展中是可用的，例如：

- 来自本地文件、远程URI或JDBC的全局缓存Lookup，使用 lookups-cached-global扩展
- 来自Kafka Topic的全局缓存Lookup，使用 kafka-extraction-namespace扩展

查询符号

在Druid SQL 中，Lookups可以使用 `LOOKUP` 函数 来进行查询，例如：

```
SELECT
  LOOKUP(store, 'store_to_country') AS country,
  SUM(revenue)
FROM sales
GROUP BY 1
```

也可以使用 JOIN运算符:

```
SELECT
  store_to_country.v AS country,
  SUM(sales.revenue) AS country_revenue
FROM
  sales
  INNER JOIN lookup.store_to_country ON sales.store = store_to_country.k
GROUP BY 1
```

在原生查询中，lookups可以使用 维度规范或者提取函数

查询执行

当执行涉及Lookup函数（如SQL中的 `LOOKUP` 函数）的聚合查询时，Druid可以决定在扫描和聚合行时应用它们，或者在聚合完成后应用它们。在聚合完成后应用Lookup更为有效，所以如果可以的话，Druid会这样做。Druid通过检查Lookup是否标记为"injective"来决定这一点。一般来说，您应该为任何自然的一对一的Lookup设置此属性，以使得Druid尽可能快地运行查询。

"injective"(内部映射式)Lookup应该包括*所有*可能出现在数据集中的键，还应该将所有键映射到*唯一值*。这一点很重要，因为非内部映射式Lookup可能将不同的键映射到同一个值，在聚合过程中必须考虑到这一点，以免查询结果包含两个应该聚合为一个的结果值。

以下Lookup为内部映射式（假设它包含了数据中所有可能的键）：

```
1 -> Foo
2 -> Bar
3 -> Billy
```

但是以下的并不是，因为"2"和"3"映射到同一个键：

```
1 -> Foo
2 -> Bar
3 -> Bar
```

可以通过在Druid配置中指定 `"injective" : true` 来告诉Druid该Lookup为内部映射式。Druid并不会自动的检测。

> [!WARNING] 目前，当Lookup是 Join数据源 的输入时，不会触发内射查找优化。它只在直接使用查找函数时使用，而不使用联接运算符。

动态配置

> [!WARNING] 动态Lookup配置是一个 实验特性, 不再支持静态配置。下面的文档说明了集群范围的配置，该配置可以通过Coordinator进行访问。配置通过服务器的"tier"概念传播。"tier"被定义为一个应该接收一组Lookup的服务集合。例如，您可以让所有Historical都是 `_default`，而Peon是它们所负责的数据源的各个层的一部分。Lookups的tier完全独立于Historical tiers。

这些配置都可以通过以下URI模板来使用JSON获取到：

```
http://<COORDINATOR_IP>:<PORT>/druid/coordinator/v1/lookups/config/{tier}/{id}
```

假设下面的所有URI都预先被添加到了 `http://<COORDINATOR_IP>:<PORT>`

如果您此前**从未**配置过lookups，**必须**首先通过POST请求发送一个Json Object `{}` 到 `/druid/coordinator/v1/lookups/config` 来进行初始化。

该接口可能返回以下几个结果：

- 资源不存在的时返回404

- 请求格式存在问题时返回400
- 请求(`POST` 和 `DELETE`)被异步接收返回202
- 请求(仅针对 `GET`)成功返回200

配置传播行为

配置由Coordinator传播到查询服务进程（Broker / Router / Peon / Historical）。查询服务进程有一个内部API，用于管理进程上的Lookup，这些查询由Coordinator使用。Coordinator定期检查是否有任何进程需要加载/删除Lookup并适当地更新它们。

请注意，一个查询服务进程只能同时处理两个同步的Lookup配置传播请求。该限制是为了防止Lookup处理消耗过多的服务器HTTP连接。

配置Lookups的API

批量更新Lookup

Lookups可以通过发送一个POST请求到 `/druid/coordinator/v1/lookups/config` 进行批量更新， 数据格式为：

```
{
    "<tierName>": {
        "<lookupName>": {
          "version": "<version>",
          "lookupExtractorFactory": {
            "type": "<someExtractorFactoryType>",
            "<someExtractorField>": "<someExtractorValue>"
          }
        }
    }
}
```

请注意，"version"是用户指定的任意字符串，当更新现有Lookup时，用户需要指定一个字典级别更高的版本。

例如，配置可能看起来像：

```
{
  "__default": {
    "country_code": {
      "version": "v0",
      "lookupExtractorFactory": {
        "type": "map",
        "map": {
          "77483": "United States"
        }
      }
    },
    "site_id": {
      "version": "v0",
      "lookupExtractorFactory": {
        "type": "cachedNamespace",
        "extractionNamespace": {
          "type": "jdbc",
          "connectorConfig": {
            "createTables": true,
            "connectURI": "jdbc:mysql:\/\/localhost:3306\/druid",
            "user": "druid",
            "password": "diurd"
          },
          "table": "lookupTable",
          "keyColumn": "country_id",
          "valueColumn": "country_name",
          "tsColumn": "timeColumn"
        },
        "firstCacheTimeout": 120000,
        "injective": true
      }
    },
    "site_id_customer1": {
      "version": "v0",
      "lookupExtractorFactory": {
        "type": "map",
        "map": {
          "847632": "Internal Use Only"
        }
      }
    },
    "site_id_customer2": {
      "version": "v0",
      "lookupExtractorFactory": {
        "type": "map",
        "map": {
          "AHF77": "Home"
        }
      }
    }
  },
  "realtime_customer1": {
    "country_code": {
      "version": "v0",
      "lookupExtractorFactory": {
        "type": "map",
        "map": {
          "77483": "United States"
        }
      }
    },
    "site_id_customer1": {
      "version": "v0",
      "lookupExtractorFactory": {
        "type": "map",
        "map": {
          "847632": "Internal Use Only"
        }
      }
    }
  }
```

```
  },
  "realtime_customer2": {
    "country_code": {
      "version": "v0",
      "lookupExtractorFactory": {
        "type": "map",
        "map": {
          "77483": "United States"
        }
      }
    },
    "site_id_customer2": {
      "version": "v0",
      "lookupExtractorFactory": {
        "type": "map",
        "map": {
          "AHF77": "Home"
        }
      }
    }
  }
}
```

map中所有的条目都将会更新，没有条目被删除。

更新Lookup

通过发送一个 `POST` 请求到 `/druid/coordinator/v1/lookups/config/{tier}/{id}` ，可以根据特定的 `lookupExtractorFactory` 来更新Lookup。

例如，一个POST `/druid/coordinator/v1/lookups/config/realtime_customer1/site_id_customer1` 可能包含以下信息：

```
{
  "version": "v1",
  "lookupExtractorFactory": {
    "type": "map",
    "map": {
      "847632": "Internal Use Only"
    }
  }
}
```

该操作会使用上边定义的配置来更新 `realtime_customer1` 的 `site_id_customer1` Lookup

获取所有Lookups

对 `/druid/coordinator/v1/lookups/config/all` 的 `GET` 请求会返回所有tier的已知Lookups

获取Lookup

对 `/druid/coordinator/v1/lookups/config/{tier}/{id}` 的 `GET` 请求会返回一个特定的Lookup

针对前边的例子， `GET` 请求 `/druid/coordinator/v1/lookups/config/realtime_customer2/site_id_customer2` 会返回：

```
{
  "version": "v1",
  "lookupExtractorFactory": {
    "type": "map",
    "map": {
      "AHF77": "Home"
    }
  }
}
```

删除Lookup

对 `/druid/coordinator/v1/lookups/config/{tier}/{id}` 的 `DELETE` 请求会删除掉集群中的Lookup，如果该Lookup是该tier的最有一个，则tier也被删除

删除tier

对 `/druid/coordinator/v1/lookups/config/{tier}` 的 `DELETE` 请求会删除掉集群中的指定tier

列出所有tier名称

对 `/druid/coordinator/v1/lookups/config` 的 `GET` 请求将返回动态配置中所有已知的tier名称列表， 在请求中加上 `discover=true` 参数（即 `/druid/coordinator/v1/lookups/config?discover=true` ）可以查找集群中除动态配置中已知tier之外当前活动的tier列表

列出所有Lookup名称

对 `/druid/coordinator/v1/lookups/config/{tier}` 的 `GET` 请求将返回该tier的所有已知Lookup的名称。

这些接口可用于获取已配置的Lookup的传播状态，以使用Historical之类的查找来处理进程。

Lookup状态的API

列出所有Lookups的加载状态

`GET /druid/coordinator/v1/lookups/status` ,参数 `detailed` 是一个可选的查询参数

列出一个tier中的Lookups的加载状态

`GET /druid/coordinator/v1/lookups/status/{tier}` ,参数 `detailed` 是一个可选的查询参数

列出单个Lookup的加载状态

`GET /druid/coordinator/v1/lookups/status/{tier}/{lookup}` ,参数 `detailed` 是一个可选的查询参数

列出所有进程的Lookup状态

`GET /druid/coordinator/v1/lookups/nodeStatus` , 参数 `discover` 为可选的查询参数，用来发现tiers或者已列出tier的Lookup

列出某个tier中进程的Lookup状态

`GET /druid/coordinator/v1/lookups/nodeStatus/{tier}`

列出单一进程中Lookup的状态

`GET /druid/coordinator/v1/lookups/nodeStatus/{tier}/{host:port}`

内部API

在Peon、Router、Broker和Historical进程中都可以消费到Lookup配置。`/druid/listen/v1/lookups` 是一个内部API，这些进程都使用该API进行list/load/drop 它们的Lookups。它们遵循与集群范围动态配置相同的返回值约定。以下接口可用于调试目的，但不能用于其他目的。

获取Lookups

在一个进程上对 `/druid/listen/v1/lookups` 的 `GET` 请求将返回当前进程上活跃的lookup的一个json map。

```
{
  "site_id_customer2": {
    "version": "v1",
    "lookupExtractorFactory": {
      "type": "map",
      "map": {
        "AHF77": "Home"
      }
    }
  }
}
```

获取Lookup

在一个进程上对 `/druid/listen/v1/lookups/some_lookup_name` 的 `GET` 请求将返回由 `some_lookup_name` 标识的LookupExtractorFactory。

```
{
  "version": "v1",
  "lookupExtractorFactory": {
    "type": "map",
    "map": {
      "AHF77": "Home"
    }
  }
}
```

配置

可以查看Coordinator配置中的 Lookups动态配置

使用以下属性来配置Broker/Router/Historical/Peon来宣告它自身作为一个lookup tier的部分。

属性	描述	默
`druid.lookup.lookupTier`	该进程上lookups的tier。 独立于其他tier	`__d`
`druid.lookup.lookupTierIsDatasource`	对于索引服务任务之类的某些操作，数据源是在任务的运行时属性中传递的。此选项从与任务的数据源相同的值中获取tier名称。建议只将其用作索引服务的Peon可选项（如果有的话）。如果为true，则 `druid.lookup.lookupTier` 必须指定。	`fal`

在Coordinator上使用以下属性来配置动态配置管理器的行为：

属性	描述	默认值
`druid.manager.lookups.hostTimeout`	每台主机处理请求的超时时间，毫秒单位	`2000` (2s)
`druid.manager.lookups.allHostTimeout`	在所有进程上完成Lookup管理的超时时间，毫秒单位	`900000` (15mins)
`druid.manager.lookups.period`	管理周期中可以暂停多久	`120000` (2mins)
`druid.manager.lookups.threadPoolSize`	可以并行的管理的服务进程数量	`10`

重启时保存配置

可以在重新启动时保存配置，这样进程就不必等待Coordinator操作来重新填充其Lookup。为此，将设置以下属性：

属性	描述	默认值
`druid.lookup.snapshotWorkingDir`	用于存储当前Lookup配置的快照的工作路径，将此属性留空将禁用快照/引导实用程序	null
`druid.lookup.enableLookupSyncOnStartup`	启动时使用Coordinator启用Lookup同步进程。可查询进程将从Coordinator获取并加载Lookup，而不是等待Coordinator加载Lookup。如果集群中没有配置Lookup，用户可以选择禁用此选项。	true
`druid.lookup.numLookupLoadingThreads`	启动时并行加载Lookup的线程数。启动完成后，此线程池将被销毁。它不会在JVM的生命周期内保留	可用的处理器/2
`druid.lookup.coordinatorFetchRetries`	在启动时同步期间，重试从Coordinator获取Lookup bean列表的次数。	3
`druid.lookup.lookupStartRetries`	在启动时同步期间或运行时，重试启动每个Lookup的次数。	3
`druid.lookup.coordinatorRetryDelay`	启动时同步期间从Coordinator获取Lookups列表的重试之间延迟的时间（毫秒）。	60000

Lookup反射

如果lookup类型实现了 `LookupIntrospectHandler` 接口，Broker提供了一个Lookup反射的API。

对 `/druid/v1/lookups/introspect/{lookupId}` 发送一个 `GET` 请求将返回完整值的map，例如： `GET /druid/v1/lookups/introspect/nato-phoneti` ：

```
{
    "A": "Alfa",
    "B": "Bravo",
    "C": "Charlie",
    ...
    "Y": "Yankee",
    "Z": "Zulu",
    "-": "Dash"
}
```

key的列表可以通过 `GET /druid/v1/lookups/introspect/{lookupId}/keys` 来获取到，例如： `GET /druid/v1/lookups/introspect/nato-phonetic/keys`

```
[
    "A",
    "B",
    "C",
    ...
    "Y",
    "Z",
    "-"
]
```

values的列表可以通过 `GET /druid/v1/lookups/introspect/{lookupId}/values` 来获取到。例如： `GET /druid/v1/lookups/introspect/nato-phonetic/values`

```
[
    "Alfa",
    "Bravo",
    "Charlie",
    ...
    "Yankee",
    "Zulu",
    "Dash"
]
```

多值维度

Apache Druid支持多值字符串维度。当输入字段中包括一个数组值而非单一值（例如，JSON数组，或者包括多个 `listDelimiter` 分割的TSV字段）时即可生成多值维度。

本文档描述了对一个维度进行聚合时，多值维度上的GroupBy查询行为（TopN很类似）。对于多值维度的内部详细信息可以查看 Segments 文档的多值列部分。本文档中的示例都为 原生Druid查询格式，对于多值维度在SQL中的使用情况请查阅 Druid SQL 文档

查询多值维度

假设，您已经有一个具有一个段的数据源，该段包含以下几行，其中 `tags` 是一个多值维度。

```
{"timestamp": "2011-01-12T00:00:00.000Z", "tags": ["t1","t2","t3"]}  #row1
{"timestamp": "2011-01-13T00:00:00.000Z", "tags": ["t3","t4","t5"]}  #row2
{"timestamp": "2011-01-14T00:00:00.000Z", "tags": ["t5","t6","t7"]}  #row3
{"timestamp": "2011-01-14T00:00:00.000Z", "tags": []}                #row4
```

过滤(Filtering)

所有的查询类型，包括 Filtered Aggregator，都可以在多值维度上进行过滤。 在多值维度上进行使用Filter遵循以下规则：

- 当多值维度的任何一个值匹配到值过滤器（例如 "selector", "bound" 和 "in"），该行即被匹配上
- 如果维度有重叠，则列比较过滤器会匹配该行
- 值过滤器中的 `null` 或者 `""` (空字符串)将匹配多值维度中的空
- 逻辑表达过滤器在多值维度上的行为方式与它们在单值维度上的行为相同：“and"标识所有基础过滤器都匹配该行; "or"表示有任意一个基础过滤器匹配该行; "not"表示基础过滤器与行不匹配。

例如，以下"or"过滤器将会匹配上边数据集中的第一行和第二行，但不会匹配第三行：

```
{
  "type": "or",
  "fields": [
    {
      "type": "selector",
      "dimension": "tags",
      "value": "t1"
    },
    {
      "type": "selector",
      "dimension": "tags",
      "value": "t3"
    }
  ]
}
```

以下"and"过滤器将会仅仅匹配上边数据集中的第一行：

```
{
  "type": "and",
  "fields": [
    {
      "type": "selector",
      "dimension": "tags",
      "value": "t1"
    },
    {
      "type": "selector",
      "dimension": "tags",
      "value": "t3"
    }
  ]
}
```

以下"selector"过滤器将匹配到上边数据集中的第四行：

```
{
  "type": "selector",
  "dimension": "tags",
  "value": null
}
```

分组(Grouping)

topN和groupBy查询可以在多值维度上进行分组。当在多值维度上进行分组时，匹配到的行中的*所有*值都会根据单值生成一个分组。可以看作相当于 `UNNEST` 运算符，它用于许多SQL支持的数组类型上, 这就意味着一个查询可以返回多于行数的分组。例如，在 `tags` 维度上使用过滤器 `"t1" AND "t3"` 将只会匹配到第一行，同时生成一个具有三个分组 `t1`，`t2` 和 `t3` 。如果您只需要包括匹配过滤器的值，可以使用 Filtered DimensionSpec, 该操作也可以提升性能。

示例：不带过滤的GroupBy

详情可以查看 GroupBy查询

```
{
  "queryType": "groupBy",
  "dataSource": "test",
  "intervals": [
    "1970-01-01T00:00:00.000Z/3000-01-01T00:00:00.000Z"
  ],
  "granularity": {
    "type": "all"
  },
  "dimensions": [
    {
      "type": "default",
      "dimension": "tags",
      "outputName": "tags"
    }
  ],
  "aggregations": [
    {
      "type": "count",
      "name": "count"
    }
  ]
}
```

返回如下结果：


```
[
  {
    "timestamp": "1970-01-01T00:00:00.000Z",
    "event": {
      "count": 1,
      "tags": "t1"
    }
  },
  {
    "timestamp": "1970-01-01T00:00:00.000Z",
    "event": {
      "count": 1,
      "tags": "t2"
    }
  },
  {
    "timestamp": "1970-01-01T00:00:00.000Z",
    "event": {
      "count": 2,
      "tags": "t3"
    }
  },
  {
    "timestamp": "1970-01-01T00:00:00.000Z",
    "event": {
      "count": 1,
      "tags": "t4"
    }
  },
  {
    "timestamp": "1970-01-01T00:00:00.000Z",
    "event": {
      "count": 2,
      "tags": "t5"
    }
  },
  {
    "timestamp": "1970-01-01T00:00:00.000Z",
    "event": {
      "count": 1,
      "tags": "t6"
    }
  },
  {
    "timestamp": "1970-01-01T00:00:00.000Z",
    "event": {
      "count": 1,
      "tags": "t7"
    }
  }
]
```


需要注意原始的行是如何扩散为多行并合并的。

示例：带选择查询过滤的GroupBy

详情可以查看 query过滤器 中的select查询过滤器。

```
{
  "queryType": "groupBy",
  "dataSource": "test",
  "intervals": [
    "1970-01-01T00:00:00.000Z/3000-01-01T00:00:00.000Z"
  ],
  "filter": {
    "type": "selector",
    "dimension": "tags",
    "value": "t3"
  },
  "granularity": {
    "type": "all"
  },
  "dimensions": [
    {
      "type": "default",
      "dimension": "tags",
      "outputName": "tags"
    }
  ],
  "aggregations": [
    {
      "type": "count",
      "name": "count"
    }
  ]
}
```

返回以下结果：

```
[
  {
    "timestamp": "1970-01-01T00:00:00.000Z",
    "event": {
      "count": 1,
      "tags": "t1"
    }
  },
  {
    "timestamp": "1970-01-01T00:00:00.000Z",
    "event": {
      "count": 1,
      "tags": "t2"
    }
  },
  {
    "timestamp": "1970-01-01T00:00:00.000Z",
    "event": {
      "count": 2,
      "tags": "t3"
    }
  },
  {
    "timestamp": "1970-01-01T00:00:00.000Z",
    "event": {
      "count": 1,
      "tags": "t4"
    }
  },
  {
    "timestamp": "1970-01-01T00:00:00.000Z",
    "event": {
      "count": 1,
      "tags": "t5"
    }
  }
]
```


您可能惊讶于结果集中包含了 "t1", "t2", "t4" 和 "t5"，这是因为查询过滤器先进行执行。 对于多值维度，在"t3"的selector过滤器会首先匹配到第一行和第二行，然后进行后续。 对于多值维度，如果多个值中的任何单个值与查询过滤器匹配，则查询过滤器将匹配到这一行。

示例：带一个选择查询过滤和另外的维度属性过滤的 GroupBy

为了解决上述问题，使得仅仅返回"t3"，可以使用下边所述查询中的"filtered dimension spec"。详情可以查看Filtered DimensionSpec部分。

```
{
  "queryType": "groupBy",
  "dataSource": "test",
  "intervals": [
    "1970-01-01T00:00:00.000Z/3000-01-01T00:00:00.000Z"
  ],
  "filter": {
    "type": "selector",
    "dimension": "tags",
    "value": "t3"
  },
  "granularity": {
    "type": "all"
  },
  "dimensions": [
    {
      "type": "listFiltered",
      "delegate": {
        "type": "default",
        "dimension": "tags",
        "outputName": "tags"
      },
      "values": ["t3"]
    }
  ],
  "aggregations": [
    {
      "type": "count",
      "name": "count"
    }
  ]
}
```

返回以下结果：

```
[
  {
    "timestamp": "1970-01-01T00:00:00.000Z",
    "event": {
      "count": 2,
      "tags": "t3"
    }
  }
]
```

注意, 对于具有 having spec的GroupBy查询也是存在类似的结果，使用一个带filtered dimensionSpec是非常高效率的，因为它使用在查询处理管道的最底层，而Having Spec使用在在groupBy查询处理的最外层。

多租户考虑

Apache Druid经常被用于支持面向用户的数据应用程序，其中多租户是一个重要的需求。本文概述了Druid的多租户存储和查询功能。

共享数据源还是每个租户一个数据源

Druid中的数据源等价于关系型数据库中的表。 对于多租户场景，可以为每一个租户创建独立的数据源，也可以多个租户之间通过使用一个租户ID的维度来共享一个或者多个数据源。在决定走哪条路时，要考虑每一条路都有利弊。

为租户独立数据源的优势：

- 每个数据源可以有自己的schema、自己的backfills、自己的分区规则以及自己的数据加载和过期规则。
- 查询可以更快，因为对于典型的租户查询，需要检查的段更少。
- 你得到了最大的灵活性。

多租户共享数据源的优势：

- 每个数据源都需要自己的JVM来进行实时索引。
- 对于Hadoop批处理作业，每个数据源都需要自己的YARN资源。
- 每个数据源在磁盘上都需要自己的段文件。
- 由于这些原因，拥有大量的小数据源可能是浪费。

一个折衷方案是使用多个数据源，但数量要比租户少。例如，您可以让一些租户使用分区规则A，而另一些租户使用分区规则B；您可以使用两个数据源并在它们之间拆分租户。

对共享数据源进行分区

如果您的多租户集群使用共享数据源，那么您的大多数查询可能会在"tenant_id"维度上过滤。当数据被租户很好地分区时，这类查询的性能最好。有几种方法可以做到这一点。

使用批处理索引，您可以使用 单维分区 按租户ID对数据进行分区。Druid总是先按时间进行分区，但每个时间段内的辅助分区将位于租户ID上。

通过实时索引，你可以通过调整发送给Druid的数据流来实现这一点。例如，如果您使用的是Kafka，那么您可以让Kafka生产者按照租户ID的哈希对您的Topic进行分区。

定制数据的分布

Druid还通过提供可配置的数据分发方式来支持多租户。Druid的Historical进程可以配置成多个层次(tier)，并且可以设置 规则(rule) 来决定哪些部分进入哪些层。其中一个场景是最近的数据比旧的数据更容易被访问，分层使较新的数据段能够托管在功能更强大的硬件上，以获得更好的性能。最近的段的第二个副本可以复制到更便宜的硬件上（另一层），旧的段也可以存储在这个层上。

支持高查询并发

Druid的基本计算单位是段。进程并行地扫描段，给定进程可以根据 `druid.processing.numThreads` 的配置并发扫描。为了并行处理更多的数据并提高性能，可以向集群中添加更多的核。Druid段的大小应该使任何给定段上的计算都能在最多500毫秒内完成。

Druid在内部将扫描段的请求存储在优先队列中。如果一个给定的查询需要扫描比集群中可用处理器总数更多的段，并且许多类似昂贵的查询同时运行，我们不希望任何查询都被耗尽。Druid的内部处理逻辑将扫描一个查询中的一组段，扫描完成后立即释放资源，允许继续扫描来自另一个查询的第二组段。通过保持段计算时间非常小，我们确保不断地产生资源，并且与不同查询相关的段都被处理。

Druid查询可以选择在查询上下文中设置 `priority` 标志。已知速度较慢的查询（下载或报告样式的查询）可以取消优先级，交互程度更高的查询可以具有更高的优先级。

Broker进程也可以专用于给定的层。例如，一组Broker进程可以专用于快速交互查询，另一组Broker进程可以专用于较慢的报告查询。Druid还提供了一个Router进程，可以根据各种查询参数（datasource、interval等）将查询路由到不同的Broker。

查询缓存

Apache Druid支持两种级别的结果缓存，分别是：段缓存和整个查询结果的缓存。缓存数据既可以存储在本地JVM堆内存中，也可以存储在一个外部的分布式kv存储中。在所有场景中，Druid的缓存是查询结果的缓存，唯一的差别是特定段的*部分结果*还是全部结果。 在所有情况下，只要数据发生变化，缓存即失效，Druid永远不会返回过期的结果。

段级缓存中即使某些底层段是可变的并且正在进行实时摄取也允许使用缓存。在这种情况下，Druid可能会缓存不可变历史段的查询结果，同时重新计算每个查询的实时段的结果。在这种情况下，连续的缓存是没有用的，因为它的结果是无效的。

段级缓存需要Druid在每次查询时合并每个段的结果，即使每个数据段的缓存结果都是从Druid缓存读取时。因此，如果不存在由于实时摄取而导致的失效的问题，则整个查询结果级缓存可以更高效。

使用和填充缓存

所有缓存都有一对参数，用于控制单个查询如何与缓存交互的行为，"use"缓存参数和"populate"缓存参数。必须通过运行时属性(runtime properties)在服务级别启用这些设置以利用缓存，但可以通过在查询上下文(query context)中设置它们来控制每个查询。"use"参数显然控制查询是否将使用缓存结果, "populate"参数控制查询是否更新缓存的结果。这些是单独的参数，目的是使得不常见数据(例如大型报表或非常旧的数据)的查询不会污染被其他查询重用的缓存结果。

Brokers上边查询缓存

Broker同时支持段级缓存与全部查询结果级缓存。 段级缓存通过参数 `useCache` 和 `populateCache` 来控制。全部结果级缓存通过参数 `useResultLevelCache` 和 `populateResultLevelCache` 来控制，这些参数都在运行时属性(runtime properties)中的 `druid.broker.cache.*`

对于小集群，在Broker上启用段级缓存比在Historical上启用查询缓存的结果更快。对于较小的生产集群（<5台服务器），建议使用此设置。对于大型生产集群，**不建议**在Broker上填充段级缓存，因为当属性 `druid.broker.cache.populateCache` 设置为 `true` （并且查询上下文参数 `populateCache` 未设置为 `false` ），则将会按段返回Historical的结果，Historical将无法进行任何本地结果合并。这会削弱Druid集群的扩展能力。

Historical上边查询缓存

Historical仅仅支持段级缓存。段级缓存通过上下文参数 `useCache` 和 `populateCache` 以及运行时属性(runtime properties)中的 `druid.historical.cache.*` 来控制。

大型集群应该仅仅在Historical上（非Broker）启用段级缓存填充，这可以避免在Broker上合并所有的查询结果。在Historical上而非Broker上启用缓存填充使得Historical可以在自己本地进行结果合并，然后将较少的数据传递给Broker。

摄取任务上的查询缓存

任务执行进程，如Peon进程或者实验性的Indexer进程仅仅支持段级缓存。段级缓存通过上下文参数 `useCache` 和 `populateCache` 以及运行时属性(runtime properties)中的 `druid.realtime.cache.*` 来控制。

大型集群应该仅仅在任务执行进程上（非Broker）启用段级缓存填充，这可以避免在Broker上合并所有的查询结果。在任务执行进程上而非Broker上启用缓存填充使得任务执行进程可以在自己本地进行结果合并，然后将较少的数据传递给Broker。

注意：任务执行进程仅仅支持将段缓存在本地，例如 `caffeine` 缓存。 存在此限制是因为这些缓存是在摄取任务生成的中间部分段级别存储结果，这些中间部分段在任务副本中不一定相同，因此任务执行进程将忽略 `memcached` 等远程缓存类型。

查询上下文

通用参数

查询上下文用于各种查询配置参数。可以通过以下方式指定查询上下文参数：

- 对于Druid SQL，上下文参数要么是通过命名为 `context` 的JSON对象来调HTTP POST接口提供，要么是作为JDBC连接的属性。
- 对于原生查询，上下文参数通过命名为 `context` 的JSON对象来提供。

以下参数可以使用在所有的查询类型中：

属性	默认值
timeout	`druid.server.http.defaultQueryTimeout`
priority	`0`
lane	`null`
queryId	自动生成
useCache	`true`
populateCache	`true`
useResultLevelCache	`true`
populateResultLevelCache	`true`
bySegment	`false`
finalize	`true`
maxScatterGatherBytes	`druid.server.http.maxScatterGatherBytes`
maxQueuedBytes	`druid.broker.http.maxQueuedBytes`
serializeDateTimeAsLong	`false`
serializeDateTimeAsLongInner	`false`
enableParallelMerge	`false`
parallelMergeParallelism	`druid.processing.merge.pool.parallelism`

属性	默认值
parallelMergeInitialYieldRows	`druid.processing.merge.task.initialYieldNu`
parallelMergeSmallBatchRows	`druid.processing.merge.task.smallBatchNumF`
useFilterCNF	`false`

查询类型特定的参数

另外，一些特定的查询类型提供了特定的上下文参数。

TopN

属性	默认值	描述
minTopNThreshold	`1000`	返回每个段的 `minTopNThreshold` 局部结果，以便合并以确定全局topN。

Timeseries

属性	默认值	描述
skipEmptyBuckets	`false`	禁用Timeseries查询中零填充行为，因此只返回包含结果的bucket。

GroupBy

GroupBy的查询上下文参数可以专门查看GroupBy查询页

矢量化参数

GroupBy和Timeseries查询类型可以在*矢量化*模式下运行，通过一次处理多个行来加快查询执行。并非所有查询都可以矢量化。特别是矢量化目前有以下要求：

- 所有查询级筛选器必须能够在位图索引上运行，或者必须提供矢量化的行匹配器。其中包括"selector"、"bound"、"in"、"like"、"regex"、"search"、"and"、"or"和"not"
- 筛选聚合器中的所有筛选器都必须提供矢量化的行匹配器
- 所有聚合器必须提供矢量化实现。其中包括"count"、"doubleSum"、"floatSum"、"longSum"、"hyperUnique"和"filtered"

- 没有虚拟列
- 对于GroupBy：所有维度spec都必须是"default"（没有提取函数或过滤的维度spec）
- 对于GroupBy：没有多值维度
- 对于时间序列：没有"降序"顺序
- 只有不可变的片段（不是实时的）
- 仅表数据源（不包括联接、子查询、查找或内联数据源）

其他查询类型（如TopN、Scan、Select和Search）忽略"vectorize"参数，将在不进行矢量化的情况下执行。这些查询类型将忽略"vectorize"参数，即使它被设置为"force"。

属性	默认值	描述
vectorize	`true`	启用或者禁用矢量化查询执行。 可能的值有 `false` (禁用), `true` （如果可能则启用，否则禁用）和 `force` (已启用，无法矢量化的groupBy和Timeseries查询将会失败)。"force"设置的目的是帮助测试，在生产中通常不起作用（因为实时段永远不能通过矢量化执行进行处理，因此对实时数据的任何查询都将失败）。设置该值将覆盖 `druid.query.vectorize`
vectorSize	`512`	设置一个特定查询的行数批量大小，设置该值将覆盖 `druid.query.vectorSize` 的值

Timeseries查询

> [!WARNING] Apache Druid支持两种查询语言： Druid SQL 和 原生查询。该文档描述了原生查询中的一种查询方式。 对于Druid SQL中使用的该种类型的信息，可以参考 SQL文档。

该类型的查询将会得到一个时间序列的查询结果，返回的是一个JSON对象数组，数组中的每一个对象表示被Timeseries查询所查的值。

一个Timeseries查询的实例如下：

```
{
  "queryType": "timeseries",
  "dataSource": "sample_datasource",
  "granularity": "day",
  "descending": "true",
  "filter": {
    "type": "and",
    "fields": [
      { "type": "selector", "dimension": "sample_dimension1", "value": "sample
      { "type": "or",
        "fields": [
          { "type": "selector", "dimension": "sample_dimension2", "value": "sa
          { "type": "selector", "dimension": "sample_dimension3", "value": "sa
        ]
      }
    ]
  },
  "aggregations": [
    { "type": "longSum", "name": "sample_name1", "fieldName": "sample_fieldNam
    { "type": "doubleSum", "name": "sample_name2", "fieldName": "sample_fieldN
  ],
  "postAggregations": [
    { "type": "arithmetic",
      "name": "sample_divide",
      "fn": "/",
      "fields": [
        { "type": "fieldAccess", "name": "postAgg__sample_name1", "fieldName":
        { "type": "fieldAccess", "name": "postAgg__sample_name2", "fieldName":
      ]
    }
  ],
  "intervals": [ "2012-01-01T00:00:00.000/2012-01-03T00:00:00.000" ]
}
```

时间序列查询主要包括7个主要部分：

属性	描述	是否必须
`queryType`	该字符串总是"timeseries"; 该字段告诉Apache Druid如何去解释这个查询	是
`dataSource`	用来标识查询的的字符串或者对象，与关系型数据库中的表类似。查看数据源可以获得更多信息	是
`descending`	是否对结果集进行降序排序,默认是 `false` , 也就是升序排列	否
`intervals`	ISO-8601格式的JSON对象，定义了要查询的时间范围	是
`granularity`	定义了查询结果的粒度，参见 Granularity	是
`filter`	参见 Filters	否
`aggregations`	参见 聚合	否
`postAggregations`	参见Post Aggregations	否
`limit`	限制返回结果数量的整数值，默认是unlimited	否
`context`	可以被用来修改查询行为，包括 Grand Total 和 Zero-filling。详情可以看 上下文参数部分中的所有参数类型	否

为了将所有数据集中起来，上面的查询将从"sample_datasource"表返回2个数据点，在 2012-01-01 和 2012-01-03 期间每天一个。每个数据点将是sample_fieldName1的longSum、sample_fieldName2的doubleSum以及sample_fieldName1除以sample_fieldName2的double结果。输出如下：

```
[
  {
    "timestamp": "2012-01-01T00:00:00.000Z",
    "result": { "sample_name1": <some_value>, "sample_name2": <some_value>, "s
  },
  {
    "timestamp": "2012-01-02T00:00:00.000Z",
    "result": { "sample_name1": <some_value>, "sample_name2": <some_value>, "s
  }
]
```

Grand Total(共计)

Druid可以在时间序列查询的结果集中增加一个额外的"总计"行，通过在上下文中增加 `"grandTotal":true` 来启用该功能，例如：

```
{
  "queryType": "timeseries",
  "dataSource": "sample_datasource",
  "intervals": [ "2012-01-01T00:00:00.000/2012-01-03T00:00:00.000" ],
  "granularity": "day",
  "aggregations": [
    { "type": "longSum", "name": "sample_name1", "fieldName": "sample_fieldNam
    { "type": "doubleSum", "name": "sample_name2", "fieldName": "sample_fieldN
  ],
  "context": {
    "grandTotal": true
  }
}
```

总计行将显示为结果数组中的最后一行，并且没有时间戳。即使查询以"降序"模式运行，它也将是最后一行。总计行中的后聚合将基于总计聚合计算。

Zero-filling(0填充)

Timeseries查询通常用零填充空的内部时间。例如，如果对间隔2012-01-01/2012-01-04发出"Day"粒度时间序列查询，并且2012-01-02不存在数据，则将收到：

```
[
  {
    "timestamp": "2012-01-01T00:00:00.000Z",
    "result": { "sample_name1": <some_value> }
  },
  {
   "timestamp": "2012-01-02T00:00:00.000Z",
   "result": { "sample_name1": 0 }
  },
  {
    "timestamp": "2012-01-03T00:00:00.000Z",
    "result": { "sample_name1": <some_value> }
  }
]
```

完全位于数据间隔之外的时间不是零填充的。

可以使用上下文标志"skipEmptyBuckets"禁用所有零填充。在此模式下，将从结果中省略2012-01-02的数据点。

设置了此上下文标志的查询如下所示：

```
{
  "queryType": "timeseries",
  "dataSource": "sample_datasource",
  "granularity": "day",
  "aggregations": [
    { "type": "longSum", "name": "sample_name1", "fieldName": "sample_fieldNam
  ],
  "intervals": [ "2012-01-01T00:00:00.000/2012-01-04T00:00:00.000" ],
  "context" : {
    "skipEmptyBuckets": "true"
  }
}
```

TopN查询

> [!WARNING] Apache Druid支持两种查询语言： Druid SQL 和 原生查询。该文档描述了原生查询中的一种查询方式。 对于Druid SQL中使用的该种类型的信息，可以参考 SQL文档。

Apache Druid TopN查询根据某些条件返回给定维度中值的排序结果集。从概念上讲，可以将它们看作是一个具有排序的、在单个维度上的近似GroupByQuery, 在该场景下TopN查询比GroupBy查询更加的效率。 这些类型的查询获取一个topN查询对象并返回一个JSON对象数组，其中每个对象代表topN查询所请求的值。

TopN是近似查询，因为每个数据进程将对其前K个结果进行排序，并且只将那些前K个结果返回给Broker。在Druid中K的默认值是 `max(1000, threshold)` 。在实践中，这意味着，如果你要求查询前1000个数据，前900个数据的正确性将为100%，之后的结果排序将无法保证。通过增加阈值可以使TopNs更加精确。

TopN的查询对象如下所示：

```
{
  "queryType": "topN",
  "dataSource": "sample_data",
  "dimension": "sample_dim",
  "threshold": 5,
  "metric": "count",
  "granularity": "all",
  "filter": {
    "type": "and",
    "fields": [
      {
        "type": "selector",
        "dimension": "dim1",
        "value": "some_value"
      },
      {
        "type": "selector",
        "dimension": "dim2",
        "value": "some_other_val"
      }
    ]
  },
  "aggregations": [
    {
      "type": "longSum",
      "name": "count",
      "fieldName": "count"
    },
    {
      "type": "doubleSum",
      "name": "some_metric",
      "fieldName": "some_metric"
    }
  ],
  "postAggregations": [
    {
      "type": "arithmetic",
      "name": "average",
      "fn": "/",
      "fields": [
        {
          "type": "fieldAccess",
          "name": "some_metric",
          "fieldName": "some_metric"
        },
        {
          "type": "fieldAccess",
          "name": "count",
          "fieldName": "count"
        }
      ]
    }
  ],
  "intervals": [
    "2013-08-31T00:00:00.000/2013-09-03T00:00:00.000"
  ]
}
```

对于TopN查询，有11个部分，如下：

属性	描述	是否必须
queryType	该字符串总是"TopN"，Druid根据该值来确定如何解析查询	是
dataSource	定义将要查询的字符串或者对象，与关系型数据库中的表类似。 详情可以查看 数据源 部分。	是
intervals	ISO-8601格式的时间间隔，定义了查询的时间范围	是
granularity	定义查询粒度， 参见 Granularities	是
filter	参见 Filters	否
aggregations	参见Aggregations	对于数值类型的metricSpec，aggregations或者postAggregations必须指定，否则非必须
postAggregations	参见postAggregations	对于数值类型的metricSpec，aggregations或者postAggregations必须指定，否则非必须
dimension	一个string或者json对象，用来定义topN查询的维度列，详情参见DimensionSpec	是
threshold	在topN中定义N的一个整型数字，例如：在top列表中返回多少个结果	是
metric	一个string或者json对象，用来指定top列表的排序。更多信息可以参见TopNMetricSpec	是
context	参见Context	否

请注意，context JSON对象也可用于topN查询，应该像timeseries一样谨慎使用。结果的格式如下：

```
[
  {
    "timestamp": "2013-08-31T00:00:00.000Z",
    "result": [
      {
        "dim1": "dim1_val",
        "count": 111,
        "some_metrics": 10669,
        "average": 96.11711711711712
      },
      {
        "dim1": "another_dim1_val",
        "count": 88,
        "some_metrics": 28344,
        "average": 322.09090909090907
      },
      {
        "dim1": "dim1_val3",
        "count": 70,
        "some_metrics": 871,
        "average": 12.442857142857143
      },
      {
        "dim1": "dim1_val4",
        "count": 62,
        "some_metrics": 815,
        "average": 13.14516129032258
      },
      {
        "dim1": "dim1_val5",
        "count": 60,
        "some_metrics": 2787,
        "average": 46.45
      }
    ]
  }
]
```


多值维度上的TopN

topN查询可以按多值维度分组。在多值维度上分组时，来自匹配行的所有值将为每个值生成一个组。查询返回的组可能多于行数。例如，在维度 `tags` 上带有过滤器 `"t1" AND "t3"` 的topN将只匹配row1，并生成包含三个组的结果： `t1` 、 `t2` 和 `t3` 。如果只需要包含与过滤器匹配的值，则可以使用 filtered dimensionSpec, 这也可以提高性能。

更过详细信息还可以参见多值维度

混淆之处

目前的TopN算法是一种近似算法，返回每个段的前1000个局部结果以进行合并，以确定全局topN。因此，topN算法在秩和结果上都是近似的。近似结果*仅适用于维度值超过1000的情况*， 唯一维度值小于1000的维度上的topN在秩和聚合上都可以被认为是精确的。

阈值可以通过服务参数 `druid.query.topN.minTopNThreshold` 从默认值1000修改，它需要重新启动服务才能生效，或者在查询上下文中设置 `minTopNThreshold` ，该查询上下文对每个查询生效。

如果您想要一个高基数、均匀分布维度的前100个维度按某个低基数、均匀分布的维度排序，那么您可能会得到丢失数据的聚合。

换言之，topN的最佳用例是当您能够确信总体结果一致地位于顶层时。例如，如果某个特定站点的ID在某个指标中每天每小时都在前10位，那么它可能会在多天内精确到topN。但是，如果一个站点在任何给定的小时内几乎不在前1000名之内，但在整个查询粒度上却在前500名（例如：一个站点在数据集中获得高度一致的流量，并且站点具有高度周期性的数据），则top500查询可能没有该特定站点的确切排名，对于那个特定站点的聚合可能并不准确。

在继续本节之前，请考虑是否确实需要确切的结果。获得准确的结果是一个非常耗费资源的过程。对于绝大多数"有用"的数据结果，近似topN算法提供了足够的精度。

如果用户希望在一个维度上获得精确的排名和精确的topN聚合，那么应该发出groupBy查询并自行对结果进行排序。对于高基数维，这在计算上非常昂贵。

如果用户能够容忍超过1000个唯一值的维度上的近似秩topN，但需要精确的聚合，则可以发出两个查询。一个用于获取近似的topN维度值，另一个具有维度选择过滤器的topN只使用第一个的topN结果。

首次查询的示例

```
{
    "aggregations": [
         {
             "fieldName": "L_QUANTITY_longSum",
             "name": "L_QUANTITY_",
             "type": "longSum"
         }
    ],
    "dataSource": "tpch_year",
    "dimension":"l_orderkey",
    "granularity": "all",
    "intervals": [
        "1900-01-09T00:00:00.000Z/2992-01-10T00:00:00.000Z"
    ],
    "metric": "L_QUANTITY_",
    "queryType": "topN",
    "threshold": 2
}
```

第二次查询的示例

```
{
    "aggregations": [
         {
             "fieldName": "L_TAX_doubleSum",
             "name": "L_TAX_",
             "type": "doubleSum"
         },
         {
             "fieldName": "L_DISCOUNT_doubleSum",
             "name": "L_DISCOUNT_",
             "type": "doubleSum"
         },
         {
             "fieldName": "L_EXTENDEDPRICE_doubleSum",
             "name": "L_EXTENDEDPRICE_",
             "type": "doubleSum"
         },
         {
             "fieldName": "L_QUANTITY_longSum",
             "name": "L_QUANTITY_",
             "type": "longSum"
         },
         {
             "name": "count",
             "type": "count"
         }
    ],
    "dataSource": "tpch_year",
    "dimension":"l_orderkey",
    "filter": {
        "fields": [
            {
                "dimension": "l_orderkey",
                "type": "selector",
                "value": "103136"
            },
            {
                "dimension": "l_orderkey",
                "type": "selector",
                "value": "1648672"
            }
        ],
        "type": "or"
    },
    "granularity": "all",
    "intervals": [
        "1900-01-09T00:00:00.000Z/2992-01-10T00:00:00.000Z"
    ],
    "metric": "L_QUANTITY_",
    "queryType": "topN",
    "threshold": 2
}
```

GroupBy查询

> [!WARNING] Apache Druid支持两种查询语言： Druid SQL 和 原生查询。该文档描述了原生查询中的一种查询方式。 对于Druid SQL中使用的该种类型的信息，可以参考 SQL文档。

这些类型的Apache Druid查询获取一个GroupBy查询对象，并返回一个JSON对象数组，其中每个对象表示查询所请求的分组。

> [!WARNING] 如果您正在使用时间作为唯一的分组进行聚合，或者在单个维度上使用有序的GroupBy，请考虑 Timeseries 和 TopN 查询以及GroupBy。在某些情况下，他们的表现可能会更好。更多详细信息，请参阅下面的备选方案。

GroupBy查询对象的示例如下所示:

```
{
  "queryType": "groupBy",
  "dataSource": "sample_datasource",
  "granularity": "day",
  "dimensions": ["country", "device"],
  "limitSpec": { "type": "default", "limit": 5000, "columns": ["country", "dat
  "filter": {
    "type": "and",
    "fields": [
      { "type": "selector", "dimension": "carrier", "value": "AT&T" },
      { "type": "or",
        "fields": [
          { "type": "selector", "dimension": "make", "value": "Apple" },
          { "type": "selector", "dimension": "make", "value": "Samsung" }
        ]
      }
    ]
  },
  "aggregations": [
    { "type": "longSum", "name": "total_usage", "fieldName": "user_count" },
    { "type": "doubleSum", "name": "data_transfer", "fieldName": "data_transfe
  ],
  "postAggregations": [
    { "type": "arithmetic",

      "name": "avg_usage",
      "fn": "/",
      "fields": [
        { "type": "fieldAccess", "fieldName": "data_transfer" },
        { "type": "fieldAccess", "fieldName": "total_usage" }
      ]
    }
  ],
  "intervals": [ "2012-01-01T00:00:00.000/2012-01-03T00:00:00.000" ],
  "having": {
    "type": "greaterThan",
    "aggregation": "total_usage",
    "value": 100
  }
}
```

下表内容为一个GroupBy查询的主要部分：

属性	描述	是否必须
queryType	该字符串应该总是"groupBy", Druid根据该值来确定如何解析查询	是
dataSource	定义将要查询的字符串或者对象，与关系型数据库中的表类似。 详情可以查看 数据源 部分。	是
dimension	一个用来GroupBy的json List，详情参见DimensionSpec来了解提取维度的方式	是
limitSpec	参见limitSpec	否
having	参见Having	否
granularity	定义查询粒度，参见 Granularities	是
filter	参见Filters	否
aggregations	参见Aggregations	否
postAggregations	参见Post Aggregations	否
intervals	ISO-8601格式的时间间隔，定义了查询的时间范围	是
subtotalsSpec	一个JSON数组，返回顶级维度子集分组的附加结果集。稍后将更详细地描述它。	否
context	参见Context	否

把它们放在一起，上面的查询将返回n*m个数据点，最多5000个点，其中n是 `country` 维度的基数，m是 `device` 维度的基数，在2012-01-01和2012-01-03之间的每一天，都会从 `sample_datasource` 表返回。如果数据点的值大于100，则每个数据点包含 `longSum total_usage` ，对于特定的 `country` 和 `device` 分组，每个数据点都包含 `double total_usage` 除以 `data_transfer` 的结果。输出如下：

```
[
  {
    "version" : "v1",
    "timestamp" : "2012-01-01T00:00:00.000Z",
    "event" : {
      "country" : <some_dim_value_one>,
      "device" : <some_dim_value_two>,
      "total_usage" : <some_value_one>,
      "data_transfer" :<some_value_two>,
      "avg_usage" : <some_avg_usage_value>
    }
  },
  {
    "version" : "v1",
    "timestamp" : "2012-01-01T00:00:12.000Z",
    "event" : {
      "dim1" : <some_other_dim_value_one>,
      "dim2" : <some_other_dim_value_two>,
      "sample_name1" : <some_other_value_one>,
      "sample_name2" :<some_other_value_two>,
      "avg_usage" : <some_other_avg_usage_value>
    }
  },
...
]
```

多值维度上的GroupBy

GroupBy查询可以按多值维度分组。在多值维度上分组时，来自匹配行的所有值将用于为每个值生成一个组，查询返回的组可能多于行数。例如，带有过滤器"t1"和"t3"的 `tags` 维度上的GroupBy将只匹配row1，并生成包含三个组的结果： `t1` 、 `t2` 和 `t3` 。如果只需要包含与过滤器匹配的值，则可以使用过滤的dimensionSpec, 这也可以提高性能。

有关详细信息，请参见多值维度。

关于subtotalSpec

小计功能允许在单个查询中计算多个子分组。要使用此功能，请在查询中添加"subtotalsSpec"，它应该是子组维度集的列表。它应该包含"dimensions"属性中维度的"outputName"，顺序与它们在"dimensions"属性中出现的顺序相同（当然，您可以跳过一些）。例如，考虑这样一个groupBy查询：

```
{
"type": "groupBy",
 ...
 ...
"dimensions": [
  {
  "type" : "default",
  "dimension" : "d1col",
  "outputName": "D1"
  },
  {
  "type" : "extraction",
  "dimension" : "d2col",
  "outputName" :  "D2",
  "extractionFn" : extraction_func
  },
  {
  "type":"lookup",
  "dimension":"d3col",
  "outputName":"D3",
  "name":"my_lookup"
  }
],
...
...
"subtotalsSpec":[ ["D1", "D2", D3"], ["D1", "D3"], ["D3"]],
..

}
```

返回的响应相当于将"dimensions"字段为["D1"、"D2"、"D3"]、["D1"、"D3"]和["D3"]的3个groupBy查询的结果与上面查询中使用的适当 `DimensionSpec` 连接起来。上述查询的响应如下所示:

```
[
  {
    "version" : "v1",
    "timestamp" : "t1",
    "event" : { "D1": "..", "D2": "..", "D3": ".." }
    }
  },
    {
    "version" : "v1",
    "timestamp" : "t2",
    "event" : { "D1": "..", "D2": "..", "D3": ".." }
    }
  },
  ...
  ...

   {
    "version" : "v1",
    "timestamp" : "t1",
    "event" : { "D1": "..", "D3": ".." }
    }
  },
    {
    "version" : "v1",
    "timestamp" : "t2",
    "event" : { "D1": "..", "D3": ".." }
    }
  },
  ...
  ...

  {
    "version" : "v1",
    "timestamp" : "t1",
    "event" : { "D3": ".." }
    }
  },
    {
    "version" : "v1",
    "timestamp" : "t2",
    "event" : { "D3": ".." }
    }
  },
...
]
```

详细实现

策略

GroupBy查询可以使用两种不同的策略执行。默认策略由Broker上的"druid.query.groupBy.defaultStrategy"运行时属性来决定，也可以在查询上下文中使用"groupByStrategy"重写。如果上下文字段和属性都未设置，则将使用"v2"策略。

- 默认设置为"v2"，旨在提供更好的性能和内存管理。此策略使用完全堆外映射生成每段结果。数据处理使用完全堆外并发事实映射和堆内字符串字典合并每个段的结果，这可能包括溢出到磁盘。数据进程将已排序的结果返回给Broker，Broker使用N-way来合并已合并的结果流。Broker在必要时将结果具体化（例如，如果查询对列而不是维度进行排序）。否则，在合并结果时，它会将结果流式返回

- "v1"是一个遗留引擎，它使用一个部分在堆上（维度键和映射本身）和部分在堆外（聚合值）的映射在数据处理（Historical、Realtime、MiddleManager）上生成每段结果。数据处理然后使用Druid的索引机制合并每个片段的结果。默认情况下，此合并是多线程的，但也可以是单线程的。Broker再次使用Druid的索引机制合并最终结果集，Broker合并总是单线程的。因为Broker使用索引机制合并结果，所以它必须在返回任何结果之前具体化完整的结果集。在数据进程和Broker上，默认情况下合并索引完全在堆上，但它可以选择将聚合值存储在堆外。

v1和v2之间的差别

两个引擎之间的查询API和结果是兼容的；但是，从集群配置的角度来看，有一些不同：

- groupBy v1使用基于行的限制（maxResults）控制资源使用，而groupBy v2使用基于字节的限制。此外，groupBy v1在堆上合并结果，而groupBy v2在堆外合并结果。这些因素意味着内存调优和资源限制在v1和v2之间表现不同。特别是，由于这一点，一些可以在一个引擎中成功完成的查询可能会超出资源限制，并在另一个引擎中失败。有关详细信息，请参阅内存调整和资源限制部分。
- groupBy v1对并发运行的查询数量没有限制，而groupBy v2通过使用有限大小的合并缓冲池来控制内存使用。默认情况下，合并缓冲区的数量是处理线程数的1/4。您可以根据需要进行调整，以平衡并发性和内存使用。
- groupBy v1支持在Broker或Historical进程上进行缓存，而groupBy v2只支持对Historical进程进行缓存。
- groupBy v2支持基于数组的聚合和基于哈希的聚合。仅当分组键是单个索引字符串列时，才使用基于数组的聚合。在基于数组的聚合中，使用字典编码的值作为索引，这样就可以直接访问数组中的聚合值，而无需基于哈希查找桶。

内存优化与资源限制

当使用groupBy v2版本时候，通过三个参数来控制资源使用和限制：

- `druid.processing.buffer.sizeBytes`，每个查询用于聚合的堆外哈希表的大小（以字节为单位），一次最多创建 `druid.processing.numMergeBuffers` 个哈希表，这也是并发运行的groupBy查询数量的上限。
- `druid.query.groupBy.maxMergingDictionarySize`，对每个查询的字符串进行分组时使用的堆上字典的大小（以字节为单位）。注意，这是基于对字典大小的粗略估计，而不是实际大小。
- `druid.query.groupBy.maxOnDiskStorage`:每个查询用于聚合的磁盘空间量（以字节为单位）。默认情况下，这是0，这意味着聚合将不使用磁盘。

如果 `maxOnDiskStorage` 为0（默认值），则超出堆内字典限制或堆外聚合表限制的查询将失败，并出现"Resource limit exceeded"错误，说明超出的限制。

如果 `maxOnDiskStorage` 大于0，则超出内存限制的查询将开始使用磁盘进行聚合。在这种情况下，当堆内字典或堆外哈希表填满时，部分聚合的记录将被排序并刷新到磁盘。然后，两个内存中的结构都将被清除，以便进一步聚合。然后继续超过 `maxOnDiskStorage` 的查询将失败，并出现"Resource limit exceeded"错误，指示它们的磁盘空间不足。

对于groupBy v2，集群操作符应该确保堆外哈希表和堆内合并字典不会超过最大可能并发查询负载的可用内存（由 `druid.processing.numMergeBuffers` 控制)。有关直接内存使用（按Druid进程类型组织）的更多详细信息，请参阅基本集群调优指南。

Broker对基础的groupBy查询不需要合并缓冲区。包含子查询的查询（使用 `query` 数据源）需要一个合并缓冲区（如果有一个子查询），如果有多个嵌套子查询层，则需要两个合并缓冲区。包含 `subtotals` 的查询需要一个合并缓冲区。它们可以相互堆叠：一个包含多层嵌套子查询的groupBy查询，也使用小计，将需要三个合并缓冲区。

Historical和摄取任务需要为每个groupBy查询提供一个合并缓冲区，除非启用了并行组合，在这种情况下，每个查询需要两个合并缓冲区。

使用groupBy v1时，所有聚合都在堆上完成，资源限制通过参数 `druid.query.groupBy.maxResults` 来决定，这是对结果集中最大结果数的限制。超过此限制的查询将失败，并显示"Resource limit exceeded"错误，指示它们超出了行限制。集群操作应该确保堆上聚合不会超过预期并发查询负载的可用JVM堆空间。

v2版本的性能优化

限制下推优化

Druid将groupBy查询中的 `limit` 规范推到Historical数据段中，以便尽早删除不必要的中间结果，并将传输给Broker的数据量降到最低。默认情况下，仅当 `orderBy` 规范中的所有字段都是分组键的子集时，才应用此技术。这是因为如果 `orderBy` 规范包含任何不在分组键中的字段， `limitPushDown` 不能保证准确的结果。但是, 即使在这种情况下, 如果您可以牺牲一些准确性来快速处理topN查询, 您也可以启用此技术。可以在高级配置部分中查看 `forceLimitPushDown` 。

优化哈希表

groupBy v2通过寻址打开哈希引擎来用于聚合。哈希表使用给定的初始bucket编号初始化，并在缓冲区满时逐渐增长。在哈希冲突中，使用了线性探测技术。

初始bucket的默认数目是1024个，哈希表的默认最大负载因子是0.7。如果在哈希表中可以看到太多的冲突，可以调整这些数字。可以在高级配置部分中查看 `bufferGrouperInitialBuckets` 和 `bufferGrouperMaxLoadFactor` 。

并行合并

一旦Historical使用哈希表完成数据聚合，它将对聚合结果进行排序并将其合并，然后再发送到Broker进行N路合并聚合。默认情况下，Historical使用其所有可用的处理线程（由 `druid.processing.numThreads` 配置)用于聚合，但使用单个线程对聚合结果进行排序和合并，这是一个http线程，用于向代Broker发送数据。

这是为了防止一些繁重的groupBy查询阻塞其他查询。在Druid中，处理线程在所有提交的查询之间共享，它们是*不可中断的*。这意味着，如果一个重查询占用了所有可用的处理线程，那么所有其他查询都可能被阻塞，直到重查询完成。GroupBy查询通常比timeseries或topN查询花费更长的时间，因此它们应该尽快释放处理线程。

但是，您可能会关心一些非常繁重的groupBy查询的性能。通常，重groupBy查询的性能瓶颈是合并排序的聚合。在这种情况下，也可以使用处理线程，这叫做*并行合并*。要启用并行合并，请参阅高级配置中的 `numParallelCombineThreads` 参数。

启用并行合并后，groupBy v2引擎可以创建一个合并树来合并排序的聚合。树的每个中间节点都是一个线程，它合并了来自子节点的聚合。叶节点线程从哈希表（包括溢出的哈希表）读取和合并聚合。通常，叶进程比中间节点慢，因为它们需要从磁盘读取数据。因此，默认情况下，中间节点使用的线程较少。可以更改中间节点的阶数。请参阅高级配置中的 `intermediateCombineDegree` 。

请注意，每个Historical都需要两个合并缓冲区来处理带有并行合并的groupBy v2查询：一个用于计算每个段的中间聚合，另一个用于并行合并中间聚合。

备选方案

在某些情况下，其他查询类型可能比groupBy更好。

- 对于没有"维度"的查询（即仅按时间分组），Timeseries查询通常比groupBy快。主要的区别在于，它是以完全流的方式实现的（利用段已经按时排序的事实），并且不需要使用哈希表进行合并。
- 对于具有单个"维度"元素的查询（即按一个字符串维度分组），TopN查询通常会比groupBy快。这是特别真实的，如果你是按指标排序，并发现近似结果可以接受。

嵌套的GroupBy查询

嵌套的groupby(类型为"query"的数据源)对"v1"和"v2"的执行方式不同。broker首先以通常的方式运行内部groupBy查询,"v1"策略然后用Druid的索引机制在堆上具体化内部查询的结果，并对这些具体化的结果运行外部查询。"v2"策略在内部查询的结果流上运行外部查询，其中包含堆外事实映射和堆内字符串字典，这些字典可能溢出到磁盘。这两种策略都以单线程方式对Broker执行外部查询。

配置

本节介绍groupBy查询的配置。可以在Broker、Historical和MiddleManager进程上设置运行时属性运行 `runtime.properties` ，可以通过查询上下文设置查询上下文参数。

groupBy v2的一些配置

支持的运行时属性：

属性	描述	默认值
`druid.query.groupBy.maxMergingDictionarySize`	合并期间用于字符串字典的最大堆空间量（大约）。当字典超过此大小时，将触发溢出到磁盘。	100000000
`druid.query.groupBy.maxOnDiskStorage`	当合并缓冲区或字典已满时，每个查询用于将结果集溢出到磁盘的最大磁盘空间量。超过此限制的查询将失败。设置为零以禁用磁盘溢出。	0(禁用)

支持的查询上下文：

key	描述
`maxMergingDictionarySize`	在本次查询中取与 `druid.query.groupBy.maxMergingDictionarySize` 较小的值
`maxOnDiskStorage`	在本次查询中取与 `` `druid.query.groupBy.maxOnDiskStorage` 比较小值

高级配置

所有GroupBy策略的通用配置

支持的运行时属性：

属性	描述	默认值
`druid.query.groupBy.defaultStrategy`	默认的GroupBy查询策略	v2
`druid.query.groupBy.singleThreaded`	使用单线程合并结果	false

支持的查询上下文：

key	描述
`groupByStrategy`	覆盖 `druid.query.groupBy.defaultStrategy` 的值
`groupByIsSingleThreaded`	覆盖 `druid.query.groupBy.singleThreaded` 的值

GroupBy V2配置

支持的运行时属性：

属性	
`druid.query.groupBy.bufferGrouperInitialBuckets`	堆外哈希表中用于分组结（1024）。
`druid.query.groupBy.bufferGrouperMaxLoadFactor`	用于分组结果的堆外哈希溢出到磁盘。设置为0以
`druid.query.groupBy.forceHashAggregation`	强制使用基于哈希的聚合
`druid.query.groupBy.intermediateCombineDegree`	合并树中合并在一起的中度将需要更少的线程，这
`druid.query.groupBy.numParallelCombineThreads`	并行合并线程数的提示。程数 为 `druid.query.groupBy.r` 较小的数
`druid.query.groupBy.applyLimitPushDownToSegment`	如果Broker将限制向下推期间限制结果。如果数据会违反直觉地降低性能。

支持的查询上下文：

key	描述
`bufferGrouperInitialBuckets`	覆盖本次查询 `druid.query.groupBy.bufferGrouperInitialB` 值
`bufferGrouperMaxLoadFactor`	覆盖本次查询 `druid.query.groupBy.bufferGrouperMaxLoadF` 值
`forceHashAggregation`	覆盖本次查询 `druid.query.groupBy.forceHashAggregation`
`intermediateCombineDegree`	覆盖本次查询 `druid.query.groupBy.intermediateCombineDe` 值
`numParallelCombineThreads`	覆盖本次查询 `druid.query.groupBy.numParallelCombineThr` 值
`sortByDimsFirst`	首先按维度值排序结果，然后按时间戳排序。
`forceLimitPushDown`	当orderby中的所有字段都是分组键的一部分时 broker将把limit操作下推到Historical。当排序 不在分组键中的字段时，使用此优化可能会导 知的近似结果，因此在这种情况下默认情况下 化。启用此上下文标志将为包含非分组键列的 limit/orderbys启用limit下推。
`applyLimitPushDownToSegment`	如果Broker将limit向下推到可查询的节点（Hi Peon），则在段扫描期间限制结果。这个上 将覆盖 `druid.query.groupBy.applyLimitPushDownToSeg`

GroupBy V1配置

支持的运行时属性：

属性	描述	默认值
`druid.query.groupBy.maxIntermediateRows`	每段分组引擎的最大中间行数。这是一个优化参数，它不会强加硬限制；相反，它可能会将合并工作从每段引擎转移到整个合并索引。超过此限制的查询不会失败。	50000
`druid.query.groupBy.maxResults`	最大结果数。超过此限制的查询将失败。	500000

支持的查询上下文：

key	描述	
`maxIntermediateRows`	在本次查询中取与 `druid.query.groupBy.maxIntermediateRows` 比较小的值	N
`maxResults`	在本次查询中取与 `druid.query.groupBy.maxResults` 比较小的值	N
`useOffheap`	设置为true可在合并结果时将聚合存储在堆外。	fa

基于数组的结果行

在内部，Druid总是使用基于数组的groupBy结果行表示，但在默认情况下，它在broker处被转换为基于map的结果格式。为了减少这种转换的开销，如果在查询上下文中将 `resultAsArray` 设置为 `true`，则还可以直接以基于数组的格式从broker返回结果。

每一行都是位置行，并按顺序包含以下字段：

- 时间戳（可选；仅适用于粒度！=全部）
- 维度（按顺序）
- 聚合器（按顺序）
- 后聚合器（可选；按顺序，如果存在）

此架构在响应中不可用，因此必须从发出的查询中计算它才能正确读取结果。

Scan查询

> [!WARNING] Apache Druid支持两种查询语言： Druid SQL 和 原生查询。该文档描述了原生查询中的一种查询方式。 对于Druid SQL中使用的该种类型的信息，可以参考 SQL文档。

Scan查询以流模式返回原始Apache Druid行。

除了向Broker发出扫描查询的简单用法之外，还可以直接向Historical或流式摄取任务发出扫描查询。如果您希望并行地检索大量数据，这将非常有用。

扫描查询对象示例如下所示：

```
{
  "queryType": "scan",
  "dataSource": "wikipedia",
  "resultFormat": "list",
  "columns":[],
  "intervals": [
    "2013-01-01/2013-01-02"
  ],
  "batchSize":20480,
  "limit":3
}
```

以下为Scan查询的主要参数：

属性	描述	是否必须
`queryType`	该字符串始终为"scan", Druid根据该字段来确定如何执行该查询	是
`dataSource`	要查询的数据源， 类似于关系型数据库的表。 可以通过 数据源 来查看更多信息	是
`intervals`	表示ISO-8601间隔的JSON对象。这定义了运行查询的时间范围。	是
`resultFormat`	结果集如何呈现：当前仅支持 `list` 和 `compactedList` ， 默认为 `list`	否
`filter`	参考 Filters	否
`columns`	要扫描的维度和指标的字符串数组。如果留空，则返回所有维度和指标。	否
`batchSize`	返回到客户端之前缓冲的最大行数。默认值为 `20480`	否
`limit`	返回多少行。如果未指定，则返回所有行。	否
`offset`	返回结果时跳过这么多行。跳过的行仍然需要在内部生成，然后丢弃，这意味着将偏移量提高到较高的值可能会导致查询使用额外的资源。"limit"和"offset"一起可以用来实现分页。但是，请注意，如果在页面获取之间修改基础数据源的方式会影响整个查询结果，那么不同的页面不一定会彼此对齐。	否
`order`	基于时间戳对返回行的排序。支持升序、降序和无（默认）。目前，"升序"和"降序"仅支持在 `columns` 字段中包含 `__time` 列并且满足时间顺序部分中列出的要求的查询。	NONE
`legacy`	返回与旧版scan-query扩展一致的结果。默认为由设置的值 `druid.query.scan.legacy` ，然后默认为false。有关详细信息，请参见传统模式。	否
`context`	一个附加的JSON对象，可用于指定某些标志（请参阅下面的查询上下文属性部分）。	否

示例结果

当resultFormat为 `list` 时，结果格式如下：

```
[{
  "segmentId" : "wikipedia_editstream_2012-12-29T00:00:00.000Z_2013-01-10T08
  "columns" : [
    "timestamp",
    "robot",
    "namespace",
    "anonymous",
    "unpatrolled",
    "page",
    "language",
    "newpage",
    "user",
    "count",
    "added",
    "delta",
    "variation",
    "deleted"
  ],
  "events" : [ {
      "timestamp" : "2013-01-01T00:00:00.000Z",
      "robot" : "1",
      "namespace" : "article",
      "anonymous" : "0",
      "unpatrolled" : "0",
      "page" : "11._korpus_(NOVJ)",
      "language" : "sl",
      "newpage" : "0",
      "user" : "EmausBot",
      "count" : 1.0,
      "added" : 39.0,
      "delta" : 39.0,
      "variation" : 39.0,
      "deleted" : 0.0
  }, {
      "timestamp" : "2013-01-01T00:00:00.000Z",
      "robot" : "0",
      "namespace" : "article",
      "anonymous" : "0",
      "unpatrolled" : "0",
      "page" : "112_U.S._580",
      "language" : "en",
      "newpage" : "1",
      "user" : "MZMcBride",
      "count" : 1.0,
      "added" : 70.0,
      "delta" : 70.0,
      "variation" : 70.0,
      "deleted" : 0.0
  }, {
      "timestamp" : "2013-01-01T00:00:00.000Z",
      "robot" : "0",
      "namespace" : "article",
      "anonymous" : "0",
      "unpatrolled" : "0",
      "page" : "113_U.S._243",
      "language" : "en",
      "newpage" : "1",
      "user" : "MZMcBride",
      "count" : 1.0,
      "added" : 77.0,
      "delta" : 77.0,
      "variation" : 77.0,
      "deleted" : 0.0
  } ]
} ]
```


当resultFormat为 `compactedList` 时，结果格式如下：

```
[{
  "segmentId" : "wikipedia_editstream_2012-12-29T00:00:00.000Z_2013-01-10T08
  "columns" : [
    "timestamp", "robot", "namespace", "anonymous", "unpatrolled", "page", "
  ],
  "events" : [
    ["2013-01-01T00:00:00.000Z", "1", "article", "0", "0", "11._korpus_(NOVJ)
    ["2013-01-01T00:00:00.000Z", "0", "article", "0", "0", "112_U.S._580", "e
    ["2013-01-01T00:00:00.000Z", "0", "article", "0", "0", "113_U.S._243", "e
  ]
} ]
```

时间排序

对于非传统模式，Scan查询当前是支持基于时间戳的排序。请注意，使用时间戳进行排序将产生并不标识来源于哪个段中的行的结果(`segmentId` 将展示为 `null`)。此外，仅当结果集限制小于 `druid.query.scan.maxRowsQueuedForOrdering` 行或者所有被扫描的段小于 `druid.query.scan.maxSegmentPartitionsOrderedInMemory` 时，才支持时间排序。 另外，除非指定了段列表，否则直接向Historical发出的查询不支持时间排序。这些限制背后的原因是，时间排序的实现使用了两种策略，如果不受限制，这两种策略可能会消耗太多堆内存。这些策略（如下所列）是根据查询结果集限制和扫描的段数在每个Historical进行选择的。

1. 优先级队列：按顺序打开Historical上的每个段。每一行都被添加到一个按时间戳排序的有界优先级队列中。对于超过结果集限制的每一行，时间戳最早（如果降序）或最晚（如果升序）的行将被取消排队。在处理完每一行之后，优先级队列的排序内容将以批处理的方式流回Broker。试图将太多的行加载到内存中会有Historical内存不足的风险。这个 `druid.query.scan.maxRowsQueuedForOrdering` 属性通过在使用时间排序时限制查询结果集中的行数来防止此问题。
2. N路合并：对于每个段，并行打开每个分区。由于每个分区的行已经按时间顺序排列，因此可以对每个分区的结果执行n路合并。这种方法不会将整个结果集持久化到内存中（像优先级队列那样），因为它会在批处理从merge函数返回时将它们流式返回。但是，由于需要为每个分区打开解压缩和解码缓冲区，尝试查询太多分区也可能导致内存使用率高。这个 `druid.query.scan.maxSegmentPartitionsOrderedInMemory` 通过在使用时间排序的任何时候限制打开的分区数来防止出现这种情况。

无论是 `druid.query.scan.maxRowsQueuedForOrdering` 还是 `druid.query.scan.maxSegmentPartitionsOrderedInMemory` ，都可以根据硬件规格和查询的维度数量来进行优化调整，这些属性也可以在查询上下文中通过 `maxRowsQueuedForOrdering` 和 `maxSegmentPartitionsOrderedInMemory` 进行覆盖，可以查看 查询上下文属性 部分来查看。

传统模式

Scan查询支持一个传统模式，该模式是为与以前的Scan查询contrib扩展的协议兼容性而设计的。在传统模式下，您可以预期以下行为更改：

- `__time` 列返回为 `timestamp` 而不是 `__time` 。 它将优先于任何其他名称为 `timestamp` 的列
- `__time` 列包含在列的列表中，即使没有特别的指定

- 时间戳以ISO8601时间字符串而不是整数的形式返回（自1970-01-01 00:00:00 UTC以来的毫秒）。

传统模式可以通过两种方式来进行触发： 在查询的JSON中传入 `"legacy":true` ，在Druid进程中设置 `druid.query.scan.legacy = true` 配置。 如果以前使用的是scan查询contrib扩展，迁移的最佳方法是在滚动升级期间激活传统模式，然后在升级完成后将其关闭。

配置属性

属性	描述	
`druid.query.scan.maxRowsQueuedForOrdering`	当使用时间排序时，返回的最大行数	1到2147之间整数
`druid.query.scan.maxSegmentPartitionsOrderedInMemory`	当使用时间排序时， 每个Historical上被扫描的最大的段数	1到2147之间整数
`druid.query.scan.legacy`	在scan查询中是否打开传统模式	true或false

查询上下文属性

属性	描述	值	
`maxRowsQueuedForOrdering`	当使用时间排序时，返回的最大行数, 覆盖同名配置	1到2147483647之间的一个整数	`dru`
`maxSegmentPartitionsOrderedInMemory`	当使用时间排序时， 每个Historical上被扫描的最大的段数	1到2147483647之间的一个整数	`dru`

示例查询上下文的JSON对象：

```
{
  "maxRowsQueuedForOrdering": 100001,
  "maxSegmentPartitionsOrderedInMemory": 100
}
```

Search查询

> [!WARNING] Apache Druid支持两种查询语言： Druid SQL 和 原生查询。该文档描述了仅仅在原生查询中的一种查询方式。

搜索查询返回与搜索规范匹配的维度值。

```
{
  "queryType": "search",
  "dataSource": "sample_datasource",
  "granularity": "day",
  "searchDimensions": [
    "dim1",
    "dim2"
  ],
  "query": {
    "type": "insensitive_contains",
    "value": "Ke"
  },
  "sort" : {
    "type": "lexicographic"
  },
  "intervals": [
    "2013-01-01T00:00:00.000/2013-01-03T00:00:00.000"
  ]
}
```

对于一个搜索查询，有以下几个主要部分：

属性	描述	是否必须
`queryType`	该字符串始终为"search", Druid根据该字段来确定如何执行该查询	是
`dataSource`	要查询的数据源， 类似于关系型数据库的表。 可以通过 数据源 来查看更多信息	是
`granularity`	定义查询粒度，参见 Granularities	是
`filter`	参考 Filters	否
`limit`	定义搜索结果的每个Historical进程返回最大多少行。	否（默认为1000）
`intervals`	表示ISO-8601间隔的JSON对象。这定义了运行查询的时间范围。	是
`searchDimensions`	运行搜索的维度列，没有设置该字段则意味着在所有的列中搜索	否
`query`	参见 Search查询说明	否
`sort`	标识搜索结果集如何排序的对象， 可能的类型有 `lexicographic` , `alphanumeric` , `strlen` , `numeric` 。 详情参见 字符串比较部分	否
`context`	参见 查询上下文	否

结果格式如下：

```
[
  {
    "timestamp": "2013-01-01T00:00:00.000Z",
    "result": [
      {
        "dimension": "dim1",
        "value": "Ke$ha",
        "count": 3
      },
      {
        "dimension": "dim2",
        "value": "Ke$haForPresident",
        "count": 1
      }
    ]
  },
  {
    "timestamp": "2013-01-02T00:00:00.000Z",
    "result": [
      {
        "dimension": "dim1",
        "value": "SomethingThatContainsKe",
        "count": 1
      },
      {
        "dimension": "dim2",
        "value": "SomethingElseThatContainsKe",
        "count": 2
      }
    ]
  }
]
```

搜索查询的详细实现策略

搜索查询可以使用两种策略来进行执行，默认策略由Broker的运行时配置 `druid.query.search.searchStrategy` 来决定，该值可以被查询上下文中的 `searchStrategy` 值覆盖。 如果查询上下文和运行时配置都有指定，默认使用 `useIndexes` 策略。

- 默认的"useIndexes"策略首先根据对位图索引的支持将搜索维度分为两组。然后，它分别对支持位图的维组和其他维组应用 `index-only` 和 `cursor-based` 的执行计划。 `index-only` 仅使用索引进行搜索查询处理。对于每个维度，它读取每个维度值的位图索引，计算搜索，最后检查时间间隔和筛选器。对于 `cursor-based` 的执行计划，请参考"cursorOnly"策略。对于基数较大(大多数搜索维度的值都是唯一的)的搜索维度， `index-only` 的性能较低。
- "cursorOnly"策略生成一个基于cursor的执行计划。这个计划创建一个游标，从**QueryableIndexSegment**中读取一行，然后计算搜索。如果某些过滤器支持位图索引，则光标只能读取满足这些过滤器的行，从而节省I/O成本。然而，对于低选择性的过滤器，它可能是缓慢的。
- "auto"策略使用基于成本的计划来选择最佳搜索策略。它估计了基于索引和游标的执行计划的成本，并选择了最优的执行计划。目前，由于成本估算的开销，默认情况下不启用。

服务端配置

下列属性将在运行时生效：

属性	描述	是否必须
`druid.query.search.searchStrategy`	默认的搜索查询策略	`useIndexes`

查询上下文

下列属性将在查询时生效：

属性	描述
`searchStrategy`	覆盖本次查询中 `druid.query.search.searchStrategy` 的值

Search查询说明

`insensitive_contains`

如果维度值的任何部分包含此搜索查询规范中指定的值（无论大小写），则会出现"匹配"。语法是：

```
{
  "type"  : "insensitive_contains",
  "value" : "some_value"
}
```

`fragment`

如果维度值的任何部分包含此搜索查询规范中指定的所有值（默认情况下不区分大小写），则会出现"匹配"。语法是：

```
{
  "type" : "fragment",
  "case_sensitive" : false,
  "values" : ["fragment1", "fragment2"]
}
```

`contains`

如果维度值的任何部分包含此搜索查询规范中指定的值，则会出现“匹配”。语法是：

```
{
  "type"  : "contains",
  "case_sensitive" : true,
  "value" : "some_value"
}
```

`regex`

如果维度值的任何部分包含此搜索查询规范中指定的模式，则会出现"匹配"。语法是：

```
{
  "type"  : "regex",
  "pattern" : "some_pattern"
}
```

TimeBoundary查询

> [!WARNING] Apache Druid支持两种查询语言： Druid SQL 和 原生查询。该文档描述了仅仅在原生查询中的一种查询方式。

时间边界查询返回数据集的最早和最新数据点。语法是：

```
{
    "queryType" : "timeBoundary",
    "dataSource": "sample_datasource",
    "bound"     : < "maxTime" | "minTime" > # optional, defaults to returning
    "filter"    : { "type": "and", "fields": [<filter>, <filter>, ...] } # opt
}
```

时间边界查询有3个主要部分：

属性	描述	是否必须
`queryType`	该字符串始终为"search", Druid根据该字段来确定如何执行该查询	是
`dataSource`	要查询的数据源， 类似于关系型数据库的表。 可以通过 数据源 来查看更多信息	是
`bound`	可选，设置为 `maxTime` 或 `minTime` 仅返回最新或最早的时间戳。如果未设置，则默认为返回两者	否
`filter`	参考 Filters	否
`context`	参见 查询上下文	否

结果的格式为：

```
[ {
  "timestamp" : "2013-05-09T18:24:00.000Z",
  "result" : {
    "minTime" : "2013-05-09T18:24:00.000Z",
    "maxTime" : "2013-05-09T18:37:00.000Z"
  }
} ]
```

SegmentMetaData查询

> [!WARNING] Apache Druid支持两种查询语言： Druid SQL 和 原生查询。该文档描述了仅仅在原生查询中的一种查询方式。同时，Druid SQL在元数据表部分包括了相似的功能。

段的元数据查询返回每个段的如下信息：

- 段中存储的行数
- 段包含的时间间隔
- 如果存储的平铺格式的数据（如：csv文件）估计一个段的大小
- 段id
- 段是否上卷
- 每一列的详细信息，比如：
  - 类型
  - 基数
  - 最大最小值
  - 存在空值
  - 估计的“平面格式”字节大小

```
{
  "queryType":"segmentMetadata",
  "dataSource":"sample_datasource",
  "intervals":["2013-01-01/2014-01-01"]
}
```

对于一个段元数据查询主要有以下几个主要部分：

属性	描述	是否必须
`queryType`	该字符串始终为"segmentMetadata"，Druid根据该字段来确定如何执行该查询	是
`dataSource`	要查询的数据源， 类似于关系型数据库的表。 可以通过 数据源 来查看更多信息	是
`intervals`	表示ISO-8601间隔的JSON对象。这定义了运行查询的时间范围。	是
`toInclude`	标识哪些列该被包含在结果中的JSON对象，默认为 `all`	否
`merge`	合并所有独立的段元数据结果到一个单一的结果中	否
`context`	详情参见 Context	否
`analysisTypes`	字符串列表，指定应计算哪些列属性（例如基数、大小）并在结果中返回。默认为 ["cardinality"，"interval"，"minmax"]，但可以使用段元数据查询配置进行覆盖。有关详细信息，请参见 analysisTypes部分	否
`lenientAggregatorMerge`	如果为true，并且启用了"聚合器" `analysisType` ，则聚合器将轻松合并。详见下文。	否

结果集的格式为：

```
[ {
  "id" : "some_id",
  "intervals" : [ "2013-05-13T00:00:00.000Z/2013-05-14T00:00:00.000Z" ],
  "columns" : {
    "__time" : { "type" : "LONG", "hasMultipleValues" : false, "hasNulls": fal
    "dim1" : { "type" : "STRING", "hasMultipleValues" : false, "hasNulls": fal
    "dim2" : { "type" : "STRING", "hasMultipleValues" : true, "hasNulls": true
    "metric1" : { "type" : "FLOAT", "hasMultipleValues" : false, "hasNulls": fa
  },
  "aggregators" : {
    "metric1" : { "type" : "longSum", "name" : "metric1", "fieldName" : "metri
  },
  "queryGranularity" : {
    "type": "none"
  },
  "size" : 300000,
  "numRows" : 5000000
} ]
```

维度列有以下类型 `STRING` ， `FLOAT` ， `DOUBLE` 或者 `LONG` 。指标列有以下类型 `FLOAT` ， `DOUBLE` ，或者 `LONG` ，或者如 `hyperUnique` 复杂类型的名字。时间戳列的类型为 `LONG` 。

如果 `errorMessage` 字段为非null，则不应信任响应中的其他字段。它们的内容没有定义。

仅仅只有字典编码的列才有基数（如 `STRING` 类型），其余类型的列（时间戳和指标列）的基数字段都是 `null` 。

intervals

如果未指定间隔，查询将使用默认间隔，该间隔跨越最近段结束时间之前的可配置时段。

默认时间周期的长度可以在Broker配置中的 `druid.query.segmentMetadata.defaultHistory` 来设置。

toInclude

有三个类型的toInclude对象。

All

语法如下：

```
"toInclude": { "type": "all"}
```

None

语法如下：

```
"toInclude": { "type": "none"}
```

List

语法如下：

```
"toInclude": { "type": "list", "columns": [<string list of column names>]}
```

analysisTypes

这是一个属性列表，用于确定返回的有关列的信息量，即对列执行的分析。

默认情况下，"cardinality", "interval" 和 "minmax" 类型被使用。

如果不需要某个属性，则从该列表中省略该属性将导致更高效的查询。

默认的分析类型可以通过Broker配置中的 `druid.query.segmentMetadata.defaultAnalysisTypes` 来设置。

列分析的类型如下描述：

cardinality

- 结果中的 `cardinality` 将返回每列基数的估计下限。仅与维度列相关。

minmax

- 预估每一列的最大最小值，仅与维度列相关

size

- 如果以文本格式存储数据，结果中的 `size` 包括段的字节大小

interval

- 结果中的 `intervals` 包括查询段相关的时间间隔

timestampSpec

- 结果中的 `timestampSpec` 包括段中存储数据的时间说明。 如果段的时间说明为未知或者未合并，该值可以为null

queryGranularity

- 结果中的 `queryGranularity` 包括段中存储的数据的查询粒度。 如果段的查询粒度为未知或者未合并，该值可以为null

aggregators

- 结果中的 `aggregators` 包括用于查询指标列使用的聚合器。 如果段的聚合器为未知或者未合并，该值可以为null
- 合并可以是严格的，也可以是宽松的。 详情可以看下边的 lenientAggregatorMerge
- 结果的格式为一个列名到聚合器的map

rollup

- 结果中的 `rollup` 为true/false/null
- 当合并开启的时候，如果某些有rollup，某些没有，则结果是null

lenientAggregatorMerge

如果某些段具有未知的聚合器，或者两个段对同一列使用不兼容的聚合器（例如，longSum更改为doubleSum），则会发生跨段聚合器元数据之间的冲突。

聚合器可以严格合并（默认）或轻松合并。在严格合并中，如果存在具有未知聚合器的任何段，或任何类型的冲突，则合并的聚合器列表将为空。通过宽松合并，具有未知聚合器的段将被忽略，聚合器之间的冲突只会使该特定列的聚合器失效。

特别是，通过宽松合并，单个列的聚合器可能为空。严格合并不会发生这种情况

DatasourceMetaData查询

> [!WARNING] Apache Druid支持两种查询语言： Druid SQL 和 原生查询。该文档描述了仅仅在原生查询中的一种查询方式。

数据源元信息查询返回一个数据源的元数据信息。这些查询返回如下信息：

- 数据源的最新摄取事件的时间戳。这是不考虑Rollup的摄取事件。

这些查询的语法为：

```
{
    "queryType" : "dataSourceMetadata",
    "dataSource": "sample_datasource"
}
```

对于一个数据源元数据查询主要有以下几个主要部分：

属性	描述	是否必须
queryType	该字符串始终为"dataSourceMetadata", Druid根据该字段来确定如何执行该查询	是
dataSource	要查询的数据源， 类似于关系型数据库的表。 可以通过 数据源 来查看更多信息	是
context	详情参见 Context	否

结果的格式为：

```
[ {
  "timestamp" : "2013-05-09T18:24:00.000Z",
  "result" : {
    "maxIngestedEventTime" : "2013-05-09T18:24:09.007Z"
  }
} ]
```

查询过滤器（Query Filters）

> [!WARNING] Apache Druid支持两种查询语言： Druid SQL 和 原生查询。该文档描述了原生查询中的一种查询方式。 对于Druid SQL中使用的该种类型的信息，可以参考 SQL文档。

Filter是一个JSON对象，指示查询的计算中应该包含哪些数据行。它本质上相当于SQL中的WHERE子句。Apache Druid支持以下类型的过滤器。

注意

过滤器通常情况下应用于维度列，但是也可以使用在聚合后的指标上，例如，参见 filtered-aggregator 和 having-filter

选择过滤器(Selector Filter)

最简单的过滤器就是选择过滤器。 选择过滤器使用一个特定的值来匹配一个特定的维度。 选择过滤器可以被用来当做更复杂的布尔表示式过滤器的基础过滤器。

选择过滤器的语法如下：

```
"filter": { "type": "selector", "dimension": <dimension_string>, "value": <dim
```

该表达式等价于 `WHERE <dimension_string> = '<dimension_value_string>'`

选择过滤器支持使用提取函数，详情可见 带提取函数的过滤器

列比较过滤器(Column Comparison Filter)

列比较过滤器与选择过滤器很相似，不同的是比较的不同的维度。例如：

```
"filter": { "type": "columnComparison", "dimensions": [<dimension_a>, <dimensi
```

该表达式等价于 `WHERE <dimension_a> = <dimension_b>`

`dimensions` 为 DimensionSpecs中的list， 需要的话还可以使用提取函数。

正则表达式过滤器(Regular expression Filter)

正则表达式过滤器与选择过滤器相似，不同的是使用正则表达式。 它使用一个给定的模式来匹配特性的维度。 模式可以是任意的标准 Java正则表达式

```
"filter": { "type": "regex", "dimension": <dimension_string>, "pattern": <patt
```

正则表达式支持使用提取函数，详情可见 带提取函数的过滤器

逻辑表达式过滤器(Logical expression Filter)

AND

AND过滤器的语法如下：

```
"filter": { "type": "and", "fields": [<filter>, <filter>, ...] }
```

该过滤器的fields字段中可以是本页中的任何一个过滤器

OR

OR过滤器的语法如下：

```
"filter": { "type": "or", "fields": [<filter>, <filter>, ...] }
```

该过滤器的fields字段中可以是本页中的任何一个过滤器

NOT

NOT过滤器的语法如下：

```
"filter": { "type": "not", "field": <filter> }
```

该过滤器的field字段中可以是本页中的任何一个过滤器

JavaScript过滤器

JavaScript过滤器使用一个特定的js函数来匹配维度。 该过滤器匹配到函数返回为true的值。

JavaScript函数需要一个维度值的参数，返回值要么是true或者false

```
{
  "type" : "javascript",
  "dimension" : <dimension_string>,
  "function" : "function(value) { <...> }"
}
```

例如， 下边的表达式将匹配维度 `name` 在 `bar` 和 `foo` 之间的维度值。

```
{
  "type" : "javascript",
  "dimension" : "name",
  "function" : "function(x) { return(x >= 'bar' && x <= 'foo') }"
}
```

JavaScript过滤器支持使用提取函数，详情可见 带提取函数的过滤器

> [!WARNING] 基于JavaScript的功能默认是禁用的。 如何启用它以及如何使用Druid JavaScript功能，参考 JavaScript编程指南。

提取过滤器(Extraction Filter)

> [!WARNING] 提取过滤器当前已经废弃。 指定了提取函数的选择器过滤器提供了相同的功能，应该改用它。

提取过滤器使用一个特定的 提取函数 来匹配维度。 以下筛选器匹配提取函数具有转换条目 `input_key=output_value` 的值，其中 `output_value` 等于过滤器 `value` ， `input_key` 作为维度显示。

例如， 下列例子匹配 `product` 列中维度值为 `[product_1, product_3, product_5]` 的数据：

```
{
    "filter": {
        "type": "extraction",
        "dimension": "product",
        "value": "bar_1",
        "extractionFn": {
            "type": "lookup",
            "lookup": {
                "type": "map",
                "map": {
                    "product_1": "bar_1",
                    "product_5": "bar_1",
                    "product_3": "bar_1"
                }
            }
        }
    }
}
```

搜索过滤器(Search Filter)

搜索过滤器可以使用在部分字符串上进行过滤匹配

```
{
    "filter": {
        "type": "search",
        "dimension": "product",
        "query": {
          "type": "insensitive_contains",
          "value": "foo"
        }
    }
}
```

属性	描述	是否必须
`type`	该值始终为 `search`	是
`dimension`	要执行搜索的维度	是
`query`	搜索类型的详细JSON对象。 详情可看下边	是
`extractionFn`	对维度使用的 提取函数	否

搜索过滤器支持使用提取函数，详情可见 带提取函数的过滤器

搜索查询规格

Contains

属性	描述	是否必须
`type`	该值始终为 `contains`	是
`value`	要执行搜索的字符串值	是
`caseSensitive`	两个字符串比较时是否忽略大小写	否（默认为false）

Insensitive Contains

属性	描述	是否必须
`type`	该值始终为 `insensitive_contains`	是
`value`	要执行搜索的字符串值	是

注意：一个"insensitive_contains"搜索等价于一个具有值为false或者未提供的"caseSensitive"的"contains"搜索

Fragment

属性	描述	是否必须
`type`	该值始终为 `fragment`	是
`values`	要执行搜索的字符串值数组	是
`caseSensitive`	两个字符串比较时是否忽略大小写	否（默认为false）

In过滤器

In过滤器可以用来表达以下SQL查询：

```
SELECT COUNT(*) AS 'Count' FROM `table` WHERE `outlaw` IN ('Good', 'Bad', 'Ug
```

In过滤器的语法如下：

```
{
   "type": "in",
   "dimension": "outlaw",
   "values": ["Good", "Bad", "Ugly"]
}
```

In过滤器支持使用提取函数，详情可见 带提取函数的过滤器

如果一个空的 `values` 传给了In过滤器，则简单的返回一个空的结果。 如果 `dimension` 为多值维度，则当维度中的一个值在 `values` 数组中时In过滤器将返回true

Like过滤器

Like过滤器被用于基本的通配符搜索，等价于SQL中的LIKE语句。 特定的符号支持"%"(匹配任意数量的字符)和"_"(匹配任意单个字符)

属性	类型	描述	是否必须
`type`	String	该值始终为 `fragment`	是
`dimension`	String	需要过滤的维度	是
`pattern`	String	LIKE模式， 例如"foo%"或者"__bar"	是
`escape`	String	可以用来转义特殊字符的转义符号	否
`extractionFn`	提取函数	对维度使用的 提取函数	否

Like过滤器支持使用提取函数，详情可见 带提取函数的过滤器

下边的Like过滤器表达了条件 `last_name LIKE "D%"` , 即： last_name以D开头

```
{
    "type": "like",
    "dimension": "last_name",
    "pattern": "D%"
}
```

边界过滤器(Bound Filter)

边界过滤器可以过滤一定范围内的维度值， 它可以用来比较大于、小于、大于等于、小于等于等

属性	类型	描述	是否必须
`type`	String	该值始终为 `fragment`	是
`dimension`	String	需要过滤的维度	是
`lower`	String	边界过滤的下边界	是
`upper`	String	边界过滤的上边界	是
`lowerStrict`	Boolean	下边界严格比较，是">"而非">="	否，默认为false
`upperStrict`	Boolean	上边界严格比较，是"<"而非"<="	否，默认为false
`ordering`	String	指定将值与边界进行比较时要使用的排序顺序。值可以为以下值之一："lexicographic", "alphanumeric", "numeric", "strlen", "version"。 详情可以查看 Sorting-Orders	否，默认为"lexicographic"
`extractionFn`	提取函数	对维度使用的 提取函数	否

边界过滤器支持使用提取函数，详情可见 带提取函数的过滤器

以下边界过滤器表达式指的是 `21 <= age <= 31`：

```
{
    "type": "bound",
    "dimension": "age",
    "lower": "21",
    "upper": "31" ,
    "ordering": "numeric"
}
```

以下表达式表达的是 `foo <= name <= hoo`，使用默认的排序方式:

```
{
    "type": "bound",
    "dimension": "name",
    "lower": "foo",
    "upper": "hoo"
}
```

使用严格边界，以下表达式表示 `21 < age < 31`：

```
{
    "type": "bound",
    "dimension": "age",
    "lower": "21",
    "lowerStrict": true,
    "upper": "31" ,
    "upperStrict": true,
    "ordering": "numeric"
}
```

以下表达式表示了一个单一边界，表示 `age < 31`：

```
{
    "type": "bound",
    "dimension": "age",
    "upper": "31" ,
    "upperStrict": true,
    "ordering": "numeric"
}
```

相反，以下表达式表示 `age >= 18`：

```
{
    "type": "bound",
    "dimension": "age",
    "lower": "18" ,
    "ordering": "numeric"
}
```

间隔过滤器(Interval Filter)

间隔过滤器针对包含长毫秒值的列启用范围过滤，边界指定为ISO-8601时间间隔。它适用于 `__time` 列， Long类型的指标列，和值可以解析为长毫秒的维度列。

该过滤器将ISO-8601的时间间隔转换为开始/结束范围，同时将这些毫秒范围转换为边界过滤器的OR操作。 边界过滤器为左闭右开匹配，（例如，start <= time < end）

属性	类型	描述	是否必须
`type`	String	该值始终为 `interval`	是
`dimension`	String	需要过滤的维度	是
`interval`	Array	包含了ISO-8601间隔字符串的JSON数组，该字段定义了要过滤的时间范围	是
`extractionFn`	提取函数	对维度使用的 提取函数	否

间隔过滤器支持使用提取函数，详情可见 带提取函数的过滤器

如果在该过滤器中使用提取函数，则提取函数的输出值应该是可以解析成长毫秒类型的。

下列实例展示了过滤时间范围在2014年10月1日到7日、2014年11月15日-16日的数据：

```
{
    "type" : "interval",
    "dimension" : "__time",
    "intervals" : [
      "2014-10-01T00:00:00.000Z/2014-10-07T00:00:00.000Z",
      "2014-11-15T00:00:00.000Z/2014-11-16T00:00:00.000Z"
    ]
}
```

上述过滤器等价于下边的OR连接的边界过滤器：

```
{
    "type": "or",
    "fields": [
      {
        "type": "bound",
        "dimension": "__time",
        "lower": "1412121600000",
        "lowerStrict": false,
        "upper": "1412640000000" ,
        "upperStrict": true,
        "ordering": "numeric"
      },
      {
         "type": "bound",
         "dimension": "__time",
         "lower": "1416009600000",
         "lowerStrict": false,
         "upper": "1416096000000" ,
         "upperStrict": true,
         "ordering": "numeric"
      }
    ]
}
```

带提取函数的过滤

除了"spatial"之外的所有的过滤器都支持提取函数，提取函数通过在过滤器的 `extractionFn` 字段中设置。 关于提取函数更多的信息可以参见提取函数

如果指定了，提取函数将在过滤器之前对输入数据进行转换。 下边的实例展示了一个带有提取函数的选择过滤器，该过滤器首先根据预定义的lookup值来转换输入值，然后转换后的值匹配到了 `bar_1` 。

实例： 下边的例子对于 `product` 列来进行匹配 `[product_1, product_3, product_5]` 中的维度值。

```
{
    "filter": {
        "type": "selector",
        "dimension": "product",
        "value": "bar_1",
        "extractionFn": {
            "type": "lookup",
            "lookup": {
                "type": "map",
                "map": {
                    "product_1": "bar_1",
                    "product_5": "bar_1",
                    "product_3": "bar_1"
                }
            }
        }
    }
}
```

列类型

Druid支持在时间戳、字符串、长整型和浮点数列上进行过滤。

注意：仅仅是字符串类型的列有位图索引。 因此，在其他类型的列上进行过滤将需要扫描列数据。

在数值列上进行过滤

在对数值列进行过滤时，可以将过滤器当作字符串来编写。在大多数情况下，筛选器将转换为数值，并将直接应用于数值列值。在某些情况下（如正则过滤器），数字列值将在扫描期间转换为字符串。

例如，在一个特定值过滤，`myFloatColumn = 10.1`：

```
"filter": {
  "type": "selector",
  "dimension": "myFloatColumn",
  "value": "10.1"
}
```

在一个范围值进行过滤，`10 <= myFloatColumn < 20`：

```
"filter": {
  "type": "bound",
  "dimension": "myFloatColumn",
  "ordering": "numeric",
  "lower": "10",
  "lowerStrict": false,
  "upper": "20",
  "upperStrict": true
}
```

在时间戳列上进行过滤

查询过滤器同时也可以应用于时间戳列。 时间戳列有长毫秒值。 时间戳列的使用是通过字符串 `__time` 来当做维度名称的。 和数值型维度类似， 当时间戳的值为字符串时，时间戳过滤器需要被指定。

如果我们希望以一个特定的格式（时区等）来解释时间戳，时间格式转换函数 是非常有用的。

例如，对一个长时间戳值进行过滤：

```
"filter": {
  "type": "selector",
  "dimension": "__time",
  "value": "124457387532"
}
```

对一周的一天进行过滤：

```
"filter": {
  "type": "selector",
  "dimension": "__time",
  "value": "Friday",
  "extractionFn": {
    "type": "timeFormat",
    "format": "EEEE",
    "timeZone": "America/New_York",
    "locale": "en"
  }
}
```

对一个ISO-8601时间间隔集合进行过滤：

```
{
    "type" : "interval",
    "dimension" : "__time",
    "intervals" : [
      "2014-10-01T00:00:00.000Z/2014-10-07T00:00:00.000Z",
      "2014-11-15T00:00:00.000Z/2014-11-16T00:00:00.000Z"
    ]
}
```

True过滤器

true过滤器是匹配所有值的过滤器。它可以用来暂时禁用其他过滤器而不删除过滤器。

```
{ "type" : "true" }
```

查询粒度(Query granularities)

> [!WARNING] Apache Druid支持两种查询语言： Druid SQL 和 原生查询。该文档描述了原生查询中的一种查询方式。 对于Druid SQL中使用的该种类型的信息，可以参考 SQL文档。

粒度字段决定了数据如何被根据时间维度组织，或者如何按小时、天、分钟等进行聚合

对于简单粒度可以使用字符串进行指定，或者对于任意粒度使用对象进行指定

简单粒度(Simple Granularities)

简单粒度被指定为字符串和bucket时间戳（按UTC时间）（例如 days开始在UTC00：00）

当前支持的粒度字符串有： `all`，`none`，`second`，`minute`，`fifteen_minute`，`thirty_minute`，`hour`，`day`，`week`，`month`，`quater` 和 `year`

- `all` 表示所有的数据都写入到一个bucket
- `none` 不存储数据（它实际上使用索引的粒度-这里的最小值是 `none`，这意味着毫秒粒度）。目前不建议在TimeseriesQuery 中使用 `none`（系统将尝试为所有不存在的毫秒生成0值，这通常是非常多的）。

实例：

假设有以下数据按秒的粒度摄入到Druid中：

```
{"timestamp": "2013-08-31T01:02:33Z", "page": "AAA", "language" : "en"}
{"timestamp": "2013-09-01T01:02:33Z", "page": "BBB", "language" : "en"}
{"timestamp": "2013-09-02T23:32:45Z", "page": "CCC", "language" : "en"}
{"timestamp": "2013-09-03T03:32:45Z", "page": "DDD", "language" : "en"}
```

当提交一个 `hour` 粒度的GroupBy查询时：

```
{
   "queryType":"groupBy",
   "dataSource":"my_dataSource",
   "granularity":"hour",
   "dimensions":[
      "language"
   ],
   "aggregations":[
      {
         "type":"count",
         "name":"count"
      }
   ],
   "intervals":[
      "2000-01-01T00:00Z/3000-01-01T00:00Z"
   ]
}
```

将得到以下结果：

```
[ {
  "version" : "v1",
  "timestamp" : "2013-08-31T01:00:00.000Z",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
}, {
  "version" : "v1",
  "timestamp" : "2013-09-01T01:00:00.000Z",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
}, {
  "version" : "v1",
  "timestamp" : "2013-09-02T23:00:00.000Z",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
}, {
  "version" : "v1",
  "timestamp" : "2013-09-03T03:00:00.000Z",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
} ]
```

可以注意到所有的空的buckets都被丢弃。

如果查询粒度变为 `day`，将会得到：

```
[ {
  "version" : "v1",
  "timestamp" : "2013-08-31T00:00:00.000Z",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
}, {
  "version" : "v1",
  "timestamp" : "2013-09-01T00:00:00.000Z",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
}, {
  "version" : "v1",
  "timestamp" : "2013-09-02T00:00:00.000Z",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
}, {
  "version" : "v1",
  "timestamp" : "2013-09-03T00:00:00.000Z",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
} ]
```

如果查询粒度为 `none`，将会得到和摄入的数据粒度一样的数据：

```
[ {
  "version" : "v1",
  "timestamp" : "2013-08-31T01:02:33.000Z",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
}, {
  "version" : "v1",
  "timestamp" : "2013-09-01T01:02:33.000Z",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
}, {
  "version" : "v1",
  "timestamp" : "2013-09-02T23:32:45.000Z",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
}, {
  "version" : "v1",
  "timestamp" : "2013-09-03T03:32:45.000Z",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
} ]
```

注意：当查询时的 `granularity` 小于 数据摄取 时候设置的 `queryGranularity` 是不合理的，因为在存储的数据中没有更细粒度的数据了。 所以，当查询时设置的粒度小于摄取时设置的粒度时，Druid将基于 `granularity` 与 `queryGranularity` 相同的基础上进行生产结果。

如果查询粒度更改为 `all` ,将会在一个bucket中查到所以数据：

```
[ {
  "version" : "v1",
  "timestamp" : "2000-01-01T00:00:00.000Z",
  "event" : {
    "count" : 4,
    "language" : "en"
  }
} ]
```

持续时间粒度

持续时间粒度指定为精确的持续时间（毫秒），时间戳返回为UTC。持续时间粒度值以毫秒为单位。

它们还支持指定可选的原点，该原点定义从何处开始计算时间段（默认为1970-01-01T00:00:00Z）。

```
{"type": "duration", "duration": 7200000}
```

每两小时就有一次

```
{"type": "duration", "duration": 3600000, "origin": "2012-01-01T00:30:00Z"}
```

在每小时30分时每一小时就有一次。

实例：

还是使用上边摄入的数据的例子，当提交一个24小时持续的GroupBy查询：

```
{
   "queryType":"groupBy",
   "dataSource":"my_dataSource",
   "granularity":{"type": "duration", "duration": "86400000"},
   "dimensions":[
      "language"
   ],
   "aggregations":[
      {
         "type":"count",
         "name":"count"
      }
   ],
   "intervals":[
      "2000-01-01T00:00Z/3000-01-01T00:00Z"
   ]
}
```

将会得到：

```
[ {
  "version" : "v1",
  "timestamp" : "2013-08-31T00:00:00.000Z",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
}, {
  "version" : "v1",
  "timestamp" : "2013-09-01T00:00:00.000Z",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
}, {
  "version" : "v1",
  "timestamp" : "2013-09-02T00:00:00.000Z",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
}, {
  "version" : "v1",
  "timestamp" : "2013-09-03T00:00:00.000Z",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
} ]
```

如果设置了查询粒度的起始时间为 `2012-01-01T00:30:00Z` ：

```
   "granularity":{"type": "duration", "duration": "86400000", "origin":"2012-0
```

将会得到：

```
[ {
  "version" : "v1",
  "timestamp" : "2013-08-31T00:30:00.000Z",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
}, {
  "version" : "v1",
  "timestamp" : "2013-09-01T00:30:00.000Z",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
}, {
  "version" : "v1",
  "timestamp" : "2013-09-02T00:30:00.000Z",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
}, {
  "version" : "v1",
  "timestamp" : "2013-09-03T00:30:00.000Z",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
} ]
```


可以注意到每个Bucket的起始时间都在30分钟。

周期性粒度

周期粒度以 ISO8601 格式指定为年、月、周、小时、分钟和秒（如P2W、P3M、PT1H30M、PT0.750S）的任意周期组合。它们支持指定一个时区来确定时段边界的起始位置以及返回的时间戳的时区。默认情况下，年份从1月1日开始，月份从1月1日开始，周从周一开始，除非指定了原点。

时区是可选的，默认为UTC。 起始时间也是可选的，默认为在给定时区的1970-01-01T00:00:00


```
{"type": "period", "period": "P2D", "timeZone": "America/Los_Angeles"}
```


这将在太平洋时区持续两天。


```
{"type": "period", "period": "P3M", "timeZone": "America/Los_Angeles", "origin
```


在太平洋时区，三个月的季度定义为从2月份开始，这将是三个月的时间段。

同样使用上边的示例数据，在太平洋时区下提交一个一天的周期的GroupBy查询：

```
{
   "queryType":"groupBy",
   "dataSource":"my_dataSource",
   "granularity":{"type": "period", "period": "P1D", "timeZone": "America/Los_
   "dimensions":[
      "language"
   ],
   "aggregations":[
      {
         "type":"count",
         "name":"count"
      }
   ],
   "intervals":[
      "1999-12-31T16:00:00.000-08:00/2999-12-31T16:00:00.000-08:00"
   ]
}
```

将会得到：

```
[ {
  "version" : "v1",
  "timestamp" : "2013-08-30T00:00:00.000-07:00",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
}, {
  "version" : "v1",
  "timestamp" : "2013-08-31T00:00:00.000-07:00",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
}, {
  "version" : "v1",
  "timestamp" : "2013-09-02T00:00:00.000-07:00",
  "event" : {
    "count" : 2,
    "language" : "en"
  }
} ]
```

注意： 每一个bucket的时间戳都已经转换成太平洋时间。 `{"timestamp": "2013-09-02T23:32:45Z", "page": "CCC", "language" : "en"}` 和 `{"timestamp": "2013-09-03T03:32:45Z", "page": "DDD", "language" : "en"}` 两行被合并到一个bucket中，是因为在太平洋时区下是同一天。

同时也可以注意到，groupBy查询中的 `intervals` 不会被转换成指定的时区，时区只会在查询结果中生效。

如果设置了粒度的起始时间为： `1970-01-01T20:30:00-08:00` ：

```
    "granularity":{"type": "period", "period": "P1D", "timeZone": "America/Los_
```

将会得到：

```
[ {
  "version" : "v1",
  "timestamp" : "2013-08-29T20:30:00.000-07:00",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
}, {
  "version" : "v1",
  "timestamp" : "2013-08-30T20:30:00.000-07:00",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
}, {
  "version" : "v1",
  "timestamp" : "2013-09-01T20:30:00.000-07:00",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
}, {
  "version" : "v1",
  "timestamp" : "2013-09-02T20:30:00.000-07:00",
  "event" : {
    "count" : 1,
    "language" : "en"
  }
} ]
```

注意到，查询中指定的 `origin` 与时区无关，它仅仅决定了第一个粒度bucket的起始点，在这种情况下， `{"timestamp": "2013-09-02T23:32:45Z", "page": "CCC", "language" : "en"}` 和 `{"timestamp": "2013-09-03T03:32:45Z", "page": "DDD", "language" : "en"}` 数据行就不在一个bucket中了。

支持的时区

时区是由Joda Time Library提供的， 其使用的是标准IANA时区。 详情可以查看 Joda Time时区支持

查询维度(Query dimensions)

> [!WARNING] Apache Druid支持两种查询语言： Druid SQL 和 原生查询。该文档描述了原生查询中的一种查询方式。 对于Druid SQL中使用的该种类型的信息，可以参考 SQL文档。

下边的JSON字段可以被用于查询中来对维度值进行操作

DimensionSpec

`DimensionSpecs` 定义在聚合之前如何转换维度值。

默认的DimensionSpec

按原样返回维度值，并可选地重命名维度

```
{
  "type" : "default",
  "dimension" : <dimension>,
  "outputName": <output_name>,
  "outputType": <"STRING"| "LONG"| "FLOAT">
}
```

带过滤的DimensionSpec

近似聚合

唯一计数

直方图与中位数

其他聚合

过滤聚合器

Having过滤器(groupBy)

配置

Coordinator

操作指南

开发指南

其他