尚硅谷大数据技术之Druid

(作者：尚硅谷大数据研发部)

版本：V1.0

1. 初识Druid

## 1.1 什么是Druid

Druid 是一个分布式的支持实时分析的数据存储系统（Data Store）。美国广告技术公司MetaMarkets 于2011 年创建了Druid 项目，并且于2012 年晚期开源了Druid 项目。Druid 设计之初的想法就是为分析而生，它在处理数据的规模、数据处理的实时性方面，比传统的OLAP 系统有了显著的性能改进，而且拥抱主流的开源生态，包括Hadoop 等。多年以来，Druid 一直是非常活跃的开源项目。

Druid 的官方网站是[http://druid.io](http://druid.io/)。   
　　另外，阿里巴巴也曾创建过一个开源项目叫作Druid（简称阿里Druid），它是一个数据库连接池的项目。阿里Druid 和本问讨论的Druid 没有任何关系，它们解决完全不同的问题。

1.2 Druid 的三个设计原则

* 快速查询（Fast Query）：部分数据的聚合（Partial Aggregate）+内存化（In-emory）+索引（Index）。
* 水平扩展能力（Horizontal Scalability）：分布式数据（Distributed Data）+ 并行化查询（Parallelizable Query）。
* 实时分析（Realtime Analytics）：不可变的过去，只追加的未来（Immutable Past，Append-Only Future）。

1.2.1 快速查询（Fast Query）

　　对于数据分析场景，大部分情况下，我们只关心一定粒度聚合的数据，而非每一行原始数据的细节情况。因此，数据聚合粒度可以是1 分钟、5 分钟、1 小时或1 天等。部分数据聚合（Partial Aggregate）给Druid 争取了很大的性能优化空间。   
　　数据内存化也是提高查询速度的杀手锏。内存和硬盘的访问速度相差近百倍，但内存的大小是非常有限的，因此在内存使用方面要精细设计，比如Druid 里面使用了Bitmap 和各种压缩技术。

另外，为了支持Drill-Down 某些维度，Druid 维护了一些倒排索引。这种方式可以加快AND 和OR 等计算操作。

1.2.2 水平扩展能力（Horizontal Scalability）

Druid 查询性能在很大程度上依赖于内存的优化使用。数据可以分布在多个节点的内存中，因此当数据增长的时候，可以通过简单增加机器的方式进行扩容。为了保持平衡，Druid按照时间范围把聚合数据进行分区处理。对于高基数的维度，只按照时间切分有时候是不够的（Druid 的每个Segment 不超过2000 万行），故Druid 还支持对Segment 进一步分区。

历史Segment 数据可以保存在深度存储系统中，存储系统可以是本地磁盘、HDFS 或远程的云服务。如果某些节点出现故障，则可借助Zookeeper 协调其他节点重新构造数据。   
　　Druid 的查询模块能够感知和处理集群的状态变化，查询总是在有效的集群架构中进行。集群上的查询可以进行灵活的水平扩展。

1.2.3 实时分析（Realtime Analytics）

Druid 提供了包含基于时间维度数据的存储服务，并且任何一行数据都是历史真实发生的事件，因此在设计之初就约定事件一但进入系统，就不能再改变。   
　　对于历史数据Druid 以Segment 数据文件的方式组织，并且将它们存储到深度存储系统中，例如文件系统或亚马逊的S3 等。当需要查询这些数据的时候，Druid 再从深度存储系统中将它们装载到内存供查询使用。

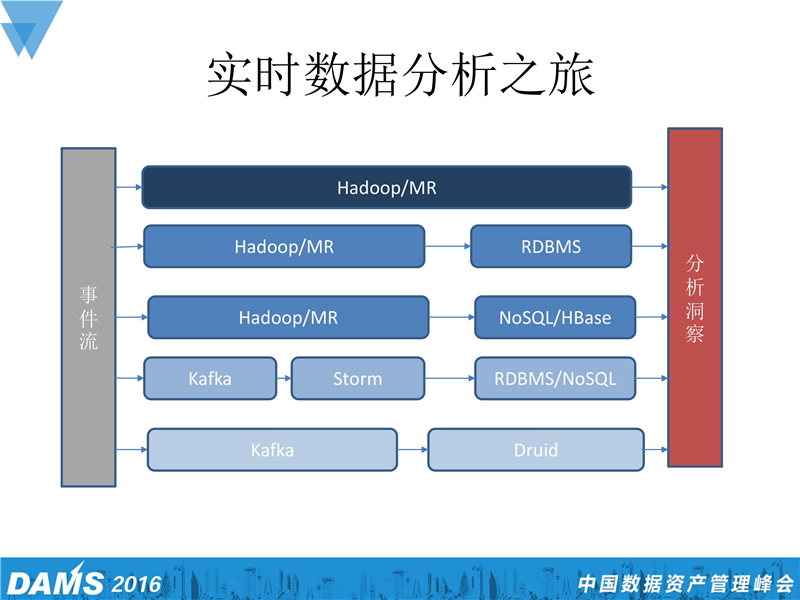
1.3 Druid 的主要特点

1. **列式存储格式**。Druid使用面向列的存储，这意味着，它只需要加载特定查询所需要的列。这为只差看几列的查询提供了巨大的速度提升。此外，每列都针对其特定的数据类型进行优化，支持快速扫描和聚合。
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，并且提供数百万条/秒的摄取率，保留数百万条记录，以及亚秒级到几秒钟的查询延迟。
3. **大规模的并行处理**。Druid可以在整个集群中进行大规模的并行查询。
4. **实时或批量摄取**。Druid可以实时摄取数据（实时获取的数据可立即用于查询）或批量处理数据。
5. **自愈，自平衡，易操作**。集群扩展和缩小，只需添加或删除服务器，集群将在后台自动重新平衡，无需任何停机时间。
6. **原生云、容错的架构，不会丢失数据**。一旦Druid吸收了您的数据，副本就安全地存储在深度存储中（通常是云存储、HDFS或共享文件系统）。即使每个Druid服务器都失败，也可以从深层存储恢复数据。对于仅影响少数Druid服务器的更有限的故障，复制确保在系统恢复时仍然可以执行查询。
7. **用于快速过滤的索引**。Druid使用CONCISE或Roaring压缩位图索引来创建索引，这些索引可以快速过滤和跨多个列搜索。
8. **近似算法**。Druid包括用于近似计数、近似排序以及计算近似直方图和分位数的算法。这些算法提供了有限的内存使用，并且通常比精确计算快得多。对于准确度比速度更重要的情况，Druid还提供精确的计数-明确和准确的排名。
9. **插入数据时自动聚合**。Druid可选地支持摄取时的数据自动汇总。预先汇总了您的数据，并且可以导致巨大的成本节约和性能提升。

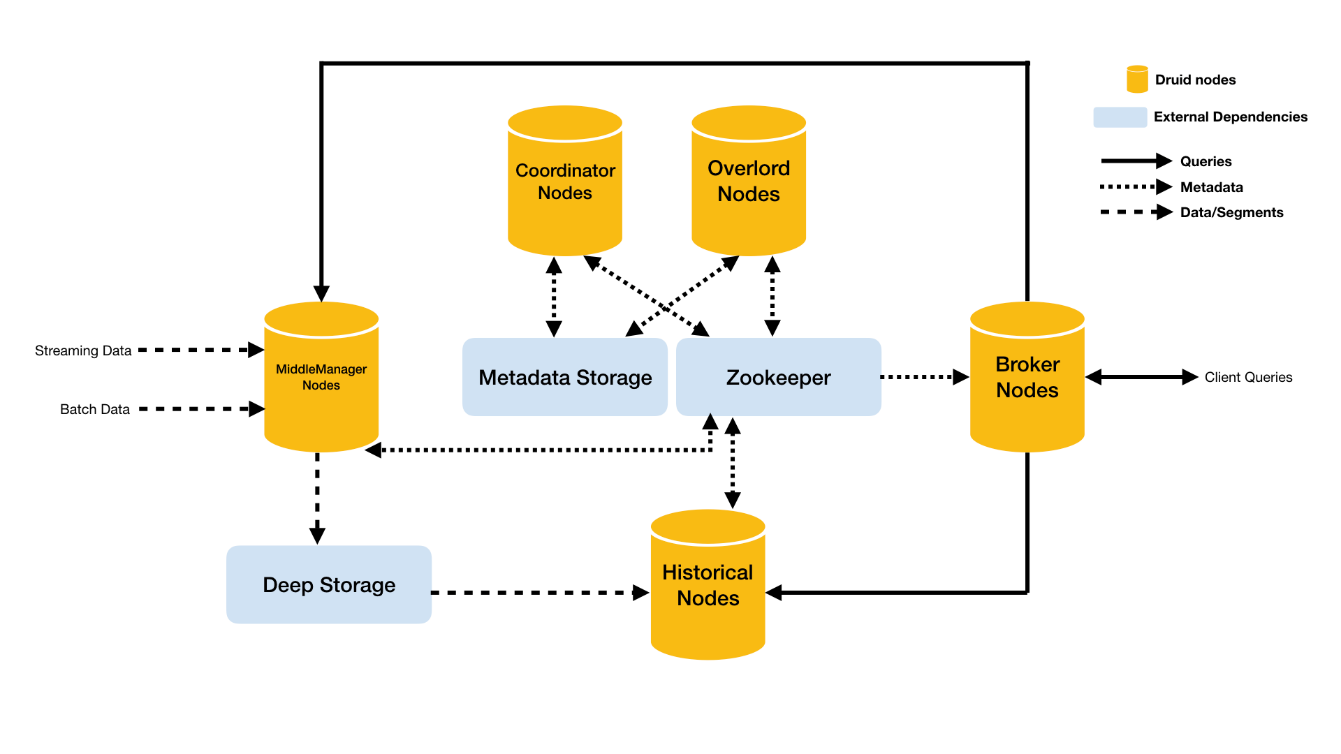
1.4 Druid的应用场景

Druid应用最多的是类似于广告分析创业公司MetaMarkets中的应用场景，如广告分析、互联网广告系统监控以及网络监控等。当业务中出现以下情况时，Druid是一个很好的技术方案选择：

* 需要交互式聚合和快速探究大量数据时；
* 具有大量数据时，如每天数亿事件的新增、每天数10T数据的增加；
* 对数据尤其是大数据进行实时分析时；
* 需要一个高可用、高容错、高性能数据库时。



1. Druid的架构



Druid总体包含以下5类节点：

1. 中间管理节点（middleManager node）：及时摄入实时数据，已生成Segment数据文件。
2. 历史节点（historical node）：加载已生成好的数据文件，以供数据查询。historical 节点是整个集群查询性能的核心所在，因为historical会承担绝大部分的segment查询。
3. 查询节点（broker node）：接收客户端查询请求，并将这些查询转发给Historicals和MiddleManagers。当Brokers从这些子查询中收到结果时，它们会合并这些结果并将它们返回给调用者。
4. 协调节点（coordinator node）：主要负责历史节点的数据负载均衡，以及通过规则（Rule）管理数据的生命周期。协调节点告诉历史节点加载新数据、卸载过期数据、复制数据、和为了负载均衡移动数据。
5. 统治者(overlord node） ：进程监视MiddleManager进程，并且是数据摄入Druid的控制器。他们负责将提取任务分配给MiddleManagers并协调Segement发布。

同时，Druid还包含3类外部依赖：

1. 数据文件存储库（DeepStorage）：存放生成的Segment数据文件，并供历史服务器下载，对于单节点集群可以是本地磁盘，而对于分布式集群一般是HDFS。
2. 元数据库（Metastore），存储Druid集群的元数据信息，比如Segment的相关信息，一般用MySQL或PostgreSQL。
3. Zookeeper：为Druid集群提供以执行协调服务。如内部服务的监控，协调和领导者选举。
4. Druid的数据结构

与Druid架构相辅相成的是其基于DataSource与Segment的数据结构，它们共同成就了 Druid的高性能优势。

3.1. DataSource结构

若与传统的关系型数据库管理系统（ RDBMS）做比较，Druid的DataSource可以理解为 RDBMS中的表（Table）。DataSource的结构包含以下几个方面。

1. 时间列（ TimeStamp）：表明每行数据的时间值，默认使用 UTC时间格式且精确到毫秒级别。这个列是数据聚合与范围查询的重要维度。
2. 维度列（Dimension）：维度来自于 OLAP的概念，用来标识数据行的各个类别信息。
3. 指标列（ Metric）：指标对应于 OLAP概念中的 Fact，是用于聚合和计算的列。这些指标列通常是一些数字，计算操作通常包括 Count、Sum和 Mean等。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/003t0w9pzy7lV2GwhUg04)﻿DataSource结构

​ 无论是实时数据消费还是批量数据处理， Druid在基于DataSource结构存储数据时即可选择对任意的指标列进行聚合（ RollUp）操作。该聚合操作主要基于维度列与时间范围两方面的情况。

下图显示的是执行聚合操作后 DataSource的数据情况。

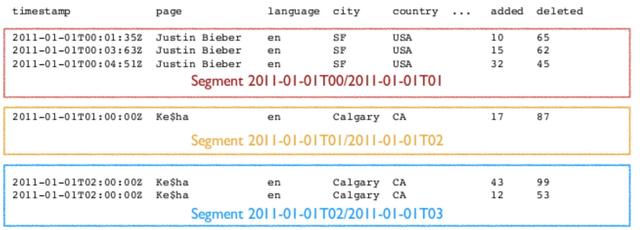
[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/003t0w9pzy7lV2Rsevibc)﻿DataSource聚合后的数

相对于其他时序数据库， Druid在数据存储时便可对数据进行聚合操作是其一大特点，该特点使得 Druid不仅能够节省存储空间，而且能够提高聚合查询的效率。

3.2. Segment结构

DataSource是一个逻辑概念， Segment却是数据的实际物理存储格式， Druid正是通过 Segment实现了对数据的横纵向切割（ Slice and Dice）操作。从数据按时间分布的角度来看，通过参数 segmentGranularity的设置，Druid将不同时间范围内的数据存储在不同的 Segment数据块中，这便是所谓的数据横向切割。

这种设计为 Druid带来一个显而易见的优点：按时间范围查询数据时，仅需要访问对应时间段内的这些 Segment数据块，而不需要进行全表数据范围查询，这使效率得到了极大的提高。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/003t0w9pzy7lV2Zux2565)﻿通过 Segment将数据按时间范围存储，同时，在 Segment中也面向列进行数据压缩存储，这便是所谓的数据纵向切割。而且在 Segment中使用了 Bitmap等技术对数据的访问进行了优化。

1. Druid的安装（单机版）

4.1 Jar包下载

从[https://imply.io/get-started](https://imply.io/get-started" \t "_blank) 下载最新版本安装包

4.2 Druid的安装部署

说明：imply集成了Druid，提供了Druid从部署到配置到各种可视化工具的完整的解决方案，imply有点类似于我们之前学过的Cloudera Manager

1.解压

tar -zxvf imply-2.7.10.tar.gz -C /opt/module

目录说明如下：

- bin/ - run scripts for included software.

- conf/ - template configurations for a clustered setup.

- conf-quickstart/\* - configurations for the single-machine quickstart.

- dist/ - all included software.

- quickstart/ - files related to the single-machine quickstart.

2.修改配置文件

1）修改Druid的ZK配置

[atguigu@hadoop102 \_common]$ pwd

/opt/module/imply/conf/druid/\_common

[atguigu@hadoop102 \_common]$ vi common.runtime.properties

druid.zk.service.host=hadoop102:2181,hadoop103:2181,hadoop104:2181

2）修改启动命令参数，使其不校验不启动内置ZK

[atguigu@hadoop102 supervise]$ pwd

/opt/module/imply/conf/supervise

:verify bin/verify-java

#:verify bin/verify-default-ports

#:verify bin/verify-version-check

:kill-timeout 10

#!p10 zk bin/run-zk conf-quickstart

3.启动

1）启动zookeeper

2）启动imply

[atguigu@hadoop102 imply]$ bin/supervise -c conf/supervise/quickstart.conf

说明：每启动一个服务均会打印出一条日志。可以通过/opt/module/imply-2.7.10/var/sv/查看服务启动时的日志信息

3）查看端口号9095的启动情况

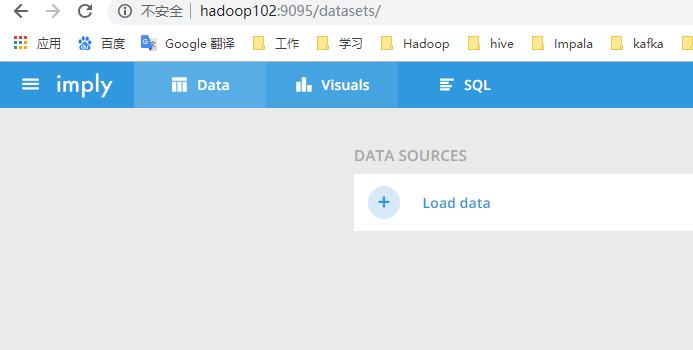
[atguigu@hadoop102 ~]$ netstat -anp | grep 9095

tcp 0 0 :::9095 :::\* LISTEN 3930/imply-ui-linux

tcp 0 0 ::ffff:192.168.1.102:9095 ::ffff:192.168.1.1:52567 ESTABLISHED 3930/imply-ui-linux

tcp 0 0 ::ffff:192.168.1.102:9095 ::ffff:192.168.1.1:52568 ESTABLISHED 3930/imply-ui-linux

1. 登录hadoop102:9095查看



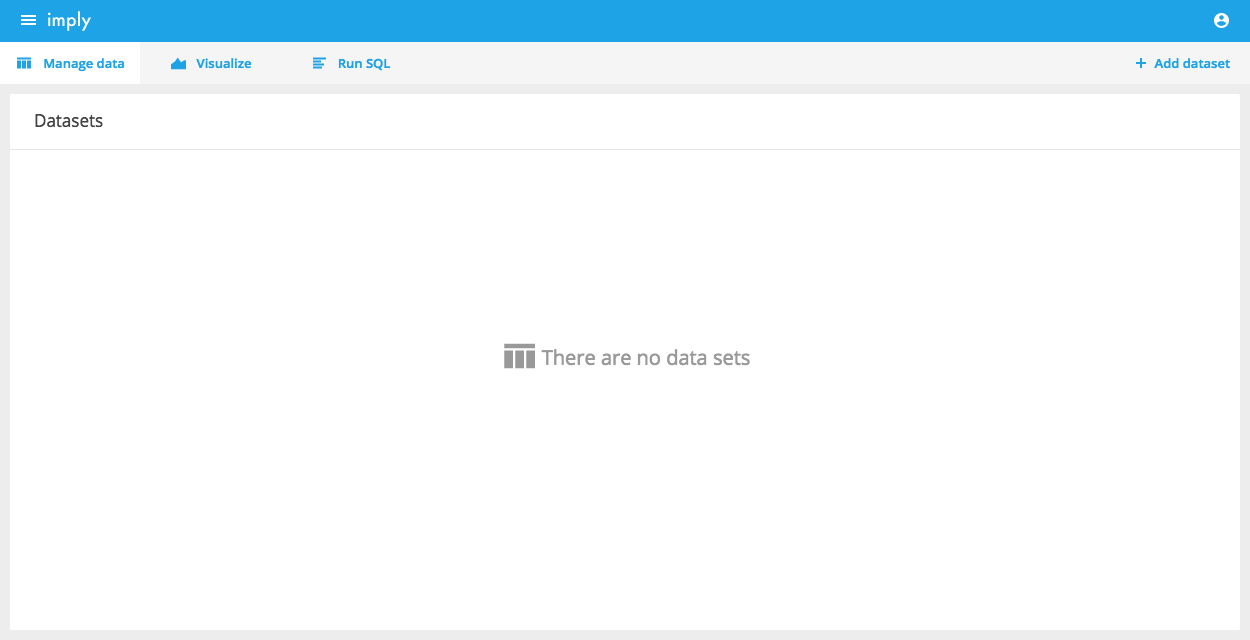
1. 停止服务

按Ctrl + c中断监督进程，如果想中断服务后进行干净的启动，请删除/opt/module/imply-2.7.10/var/目录。

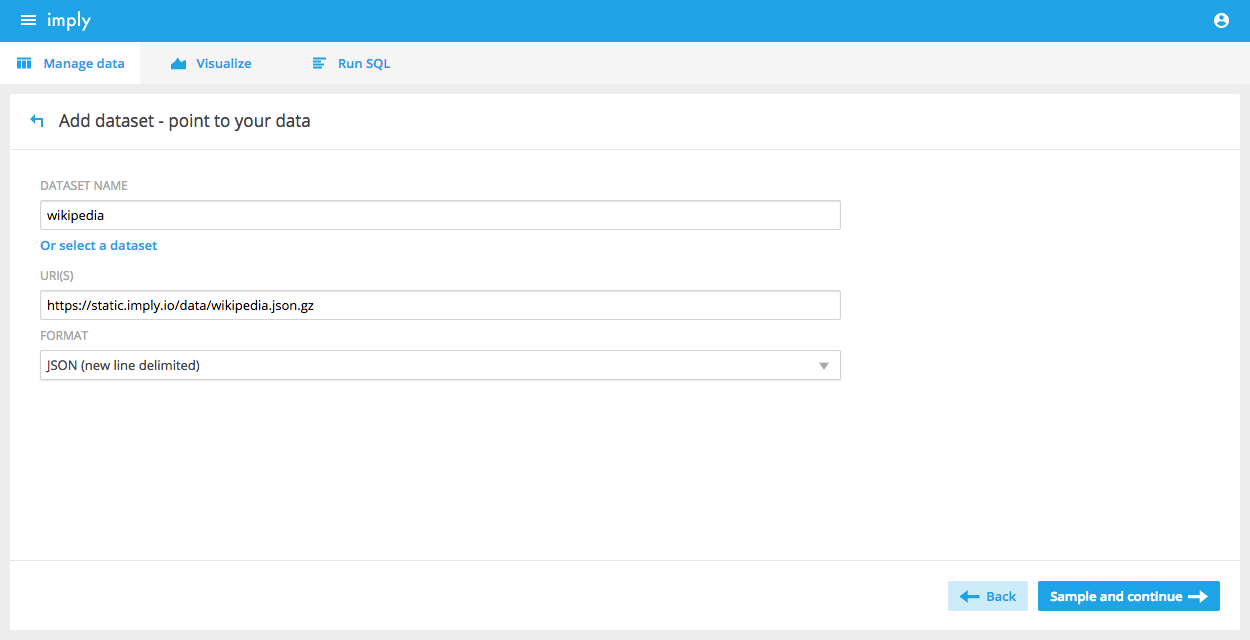
1. Druid的入门

5.1 在线加载样本数据

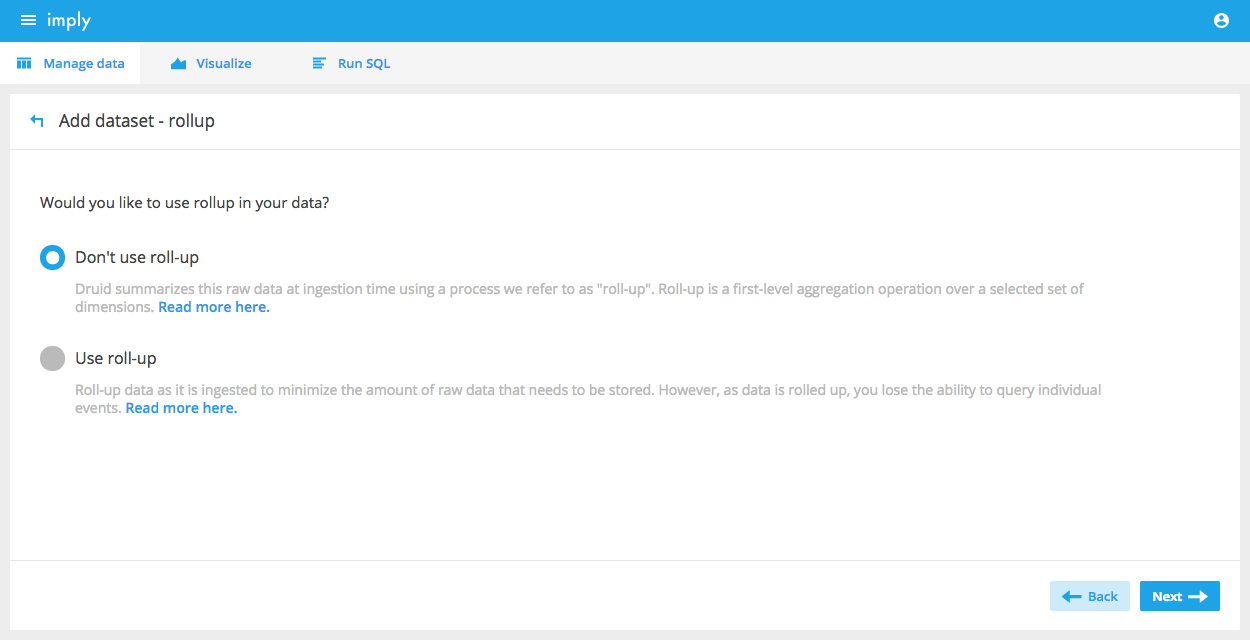
1.打开imply



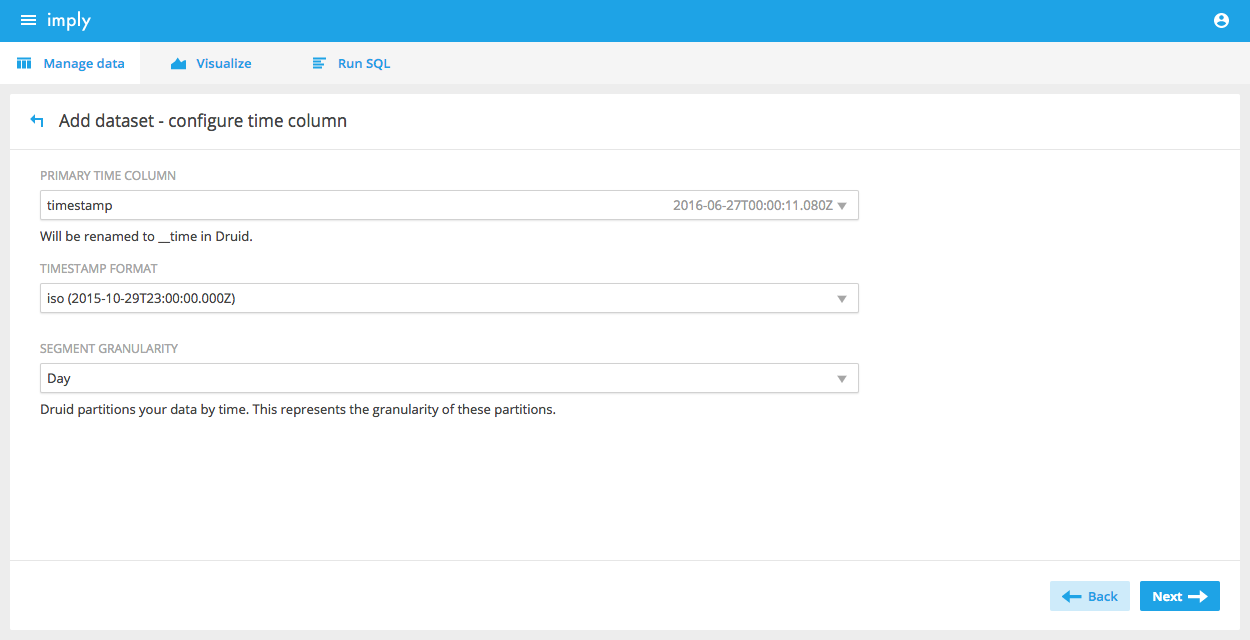
2.开始连接到实例Wikipedia数据集



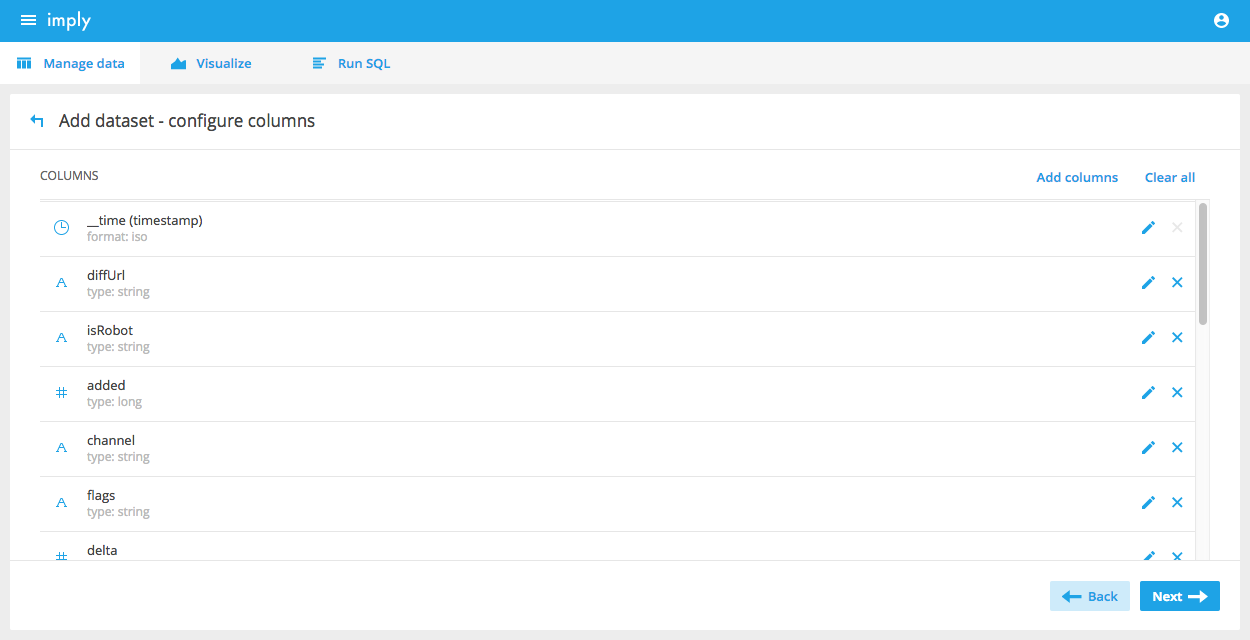
3.加载样本数据。Wikipedia示例使用Http数据加载器从URI路径读取数据，格式为json。可以通过点击采样并继续，对文件前几行的数据进行采样，以确保它是可解析的数据。



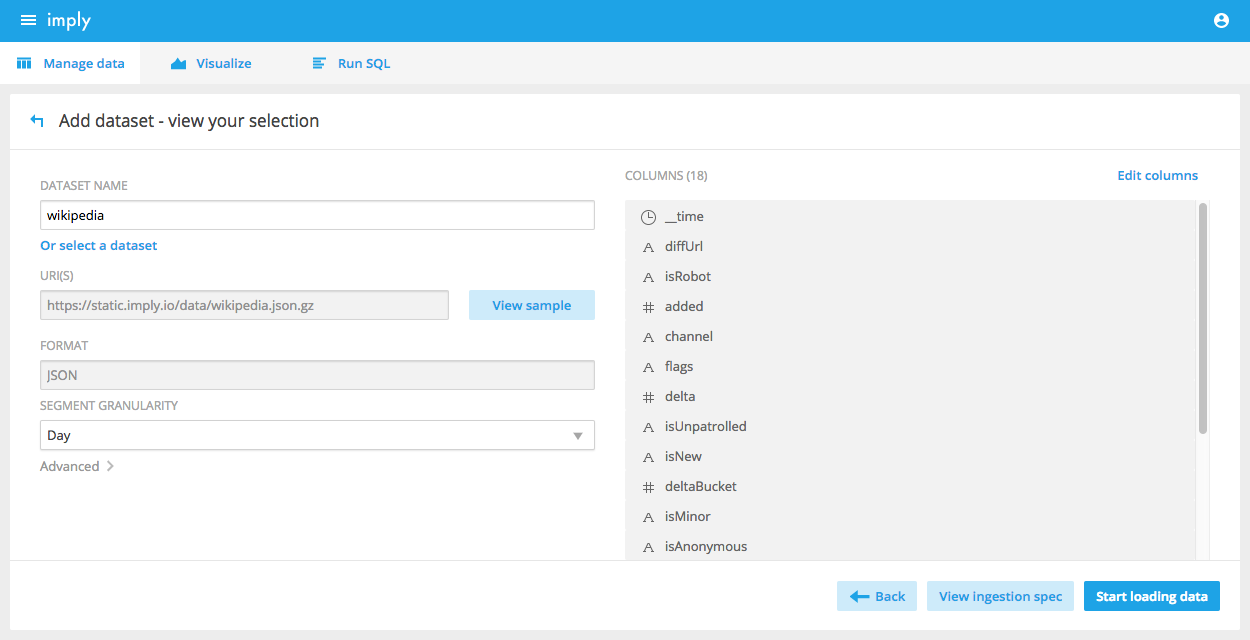
4.配置汇总



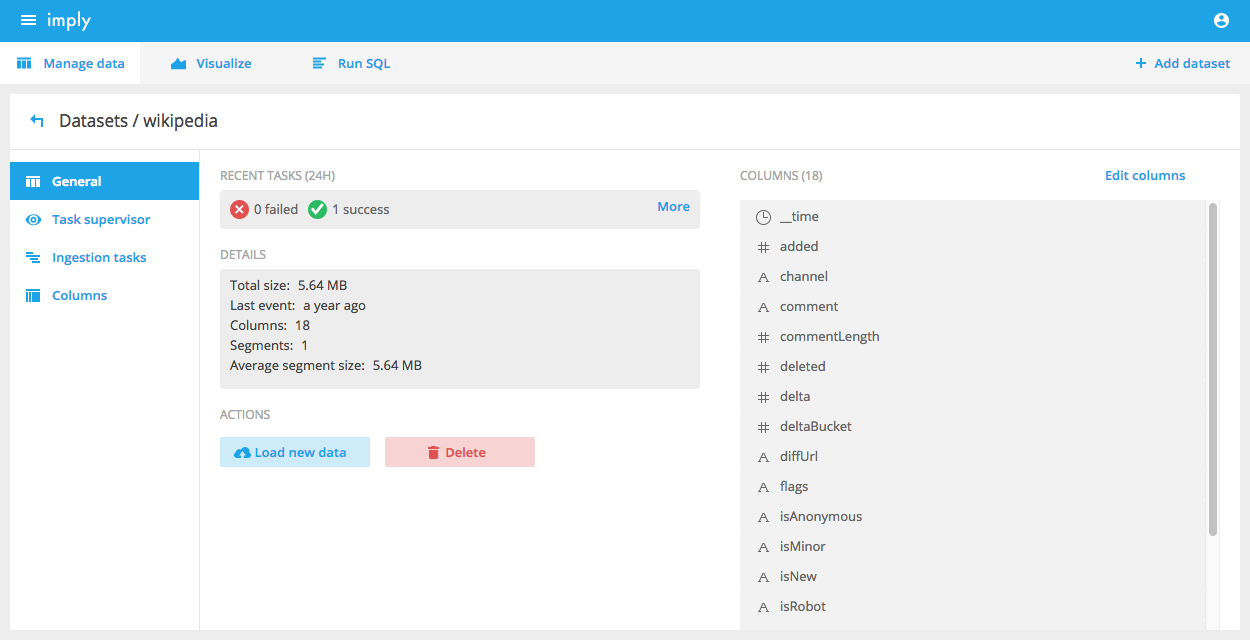
5.配置时间戳和分区



6.配置要加载的列。



7.确认并开始摄取！



一旦加载器指示数据已被索引，您就可以继续下一部分来定义数据立方体并开始可视化数据。

* 1. 离线加载样本数据

如果您无法访问公共Web服务器，则可以从本地文件加载相同的数据集。该quickstart目录包括一个样本数据集和一个摄取规范来处理数据，分别命名wikipedia-2016-06-27-sampled.json和wikipedia-index.json。

要为此摄取规范向Druid提交索引作业，请从Imply目录运行以下命令：

bin/post-index-task --file quickstart/wikipedia-index.json

成功运行将生成类似于以下内容的日志：

Beginning indexing data for wikipedia

Task started: index\_wikipedia\_2017-12-05T03:22:28.612Z

Task log: http://localhost:8090/druid/indexer/v1/task/index\_wikipedia\_2017-12-05T03:22:28.612Z/log

Task status: http://localhost:8090/druid/indexer/v1/task/index\_wikipedia\_2017-12-05T03:22:28.612Z/status

Task index\_wikipedia\_2017-12-05T03:22:28.612Z still running...

Task index\_wikipedia\_2017-12-05T03:22:28.612Z still running...

Task finished with status: SUCCESS

Completed indexing data for wikipedia. Now loading indexed data onto the cluster...

wikipedia is 0.0% finished loading...

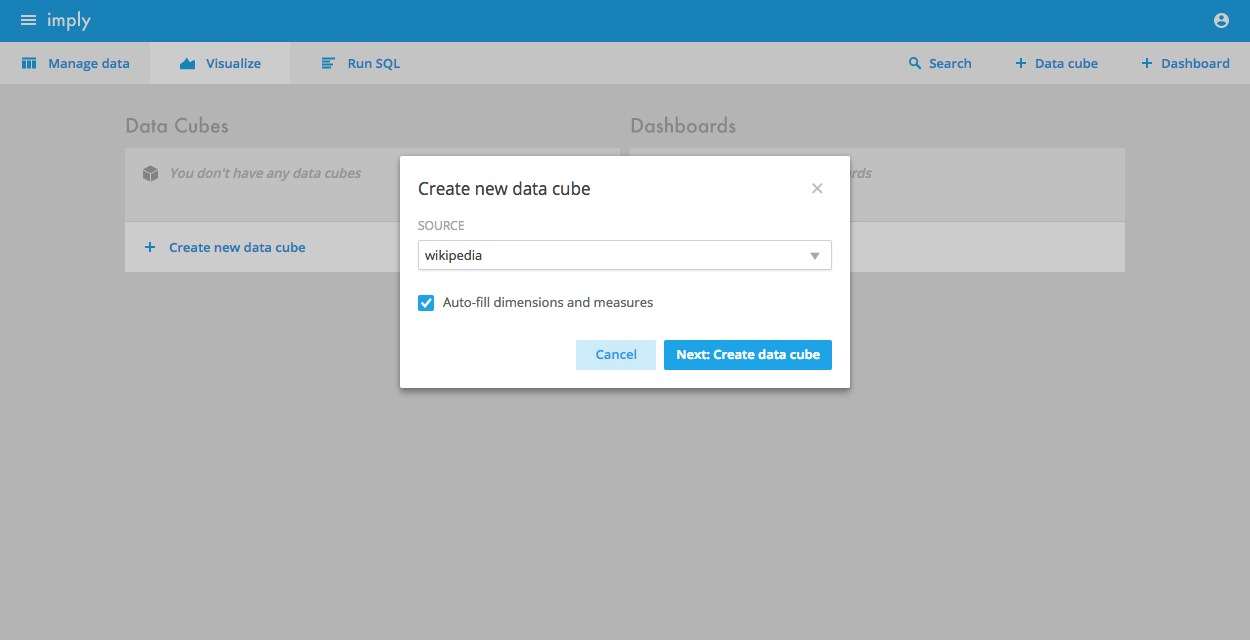
wikipedia is 0.0% finished loading...

wikipedia is 0.0% finished loading...

wikipedia loading complete! You may now query your data

* 1. 创建数据立方体

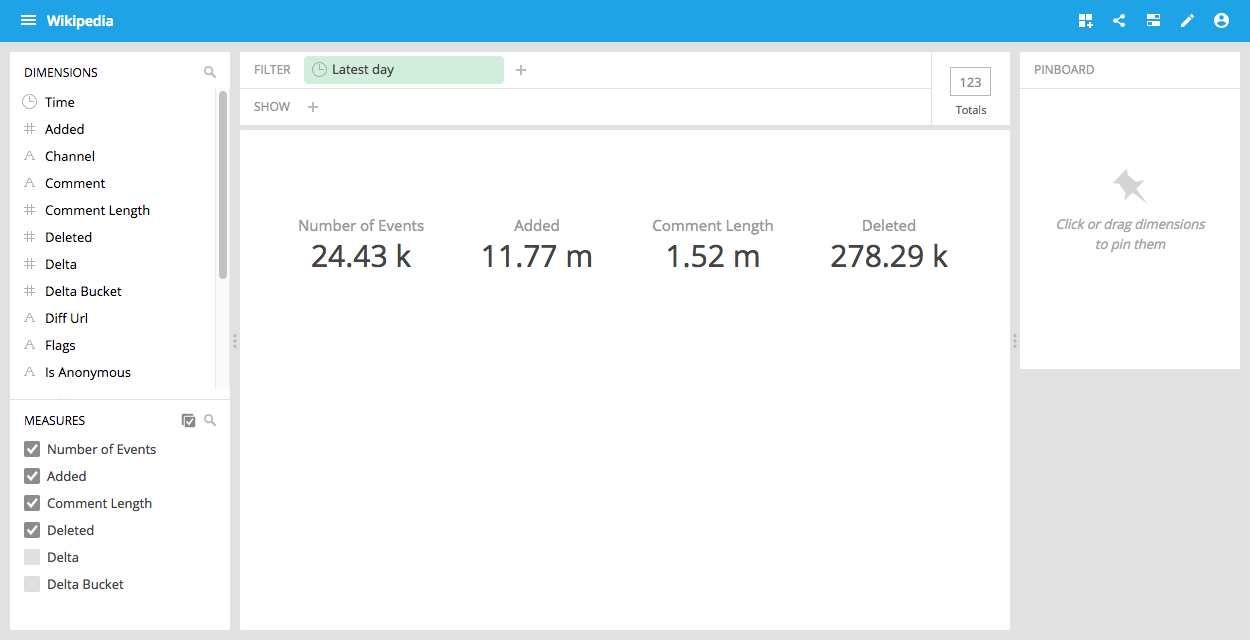
通过单击顶部栏上的相应按钮切换到Imply 的“ 可视化”部分。从这里，您可以创建数据立方体来建模数据，浏览这些立方体，并将视图组织到仪表板中。首先单击+创建新数据多维数据集。



在出现的对话框中，确保wikipedia选中此源并选择自动填充尺寸和度量。单击下一步继续：创建数据立方体。

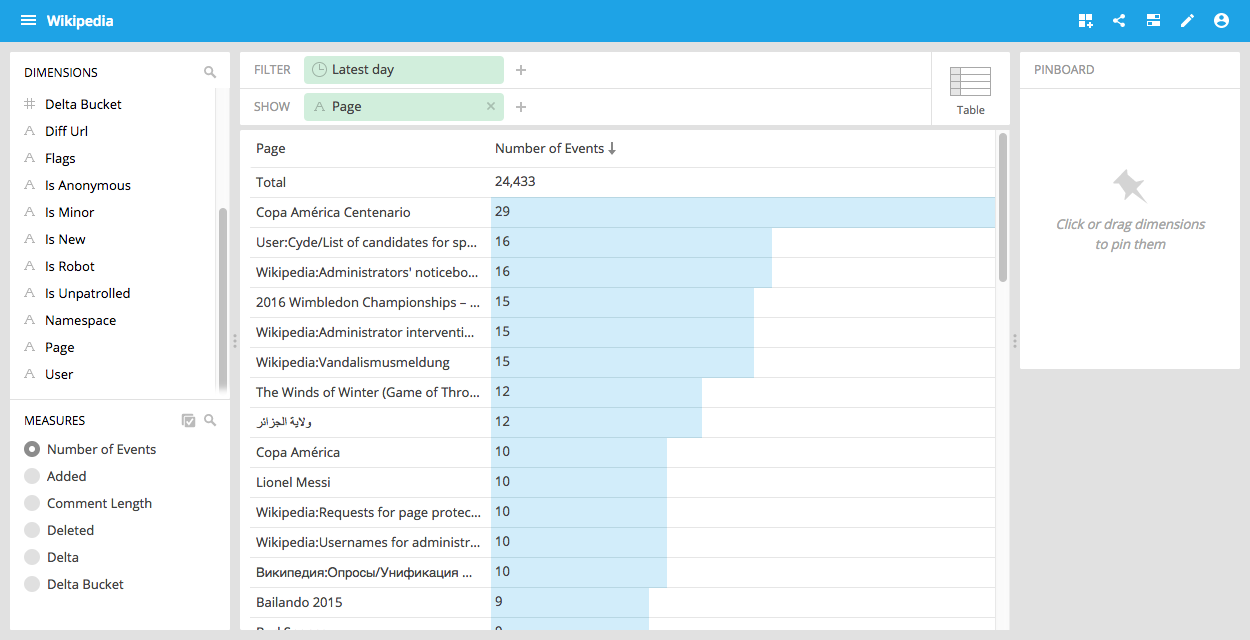
* 1. 可视化数据立方体

单击“ 保存”后，将自动加载此新数据多维数据集的数据立方体视图。将来，还可以通过从“ 可视化”屏幕单击数据立方体的名称（在此示例中为“Wikipedia”）来加载此视图。

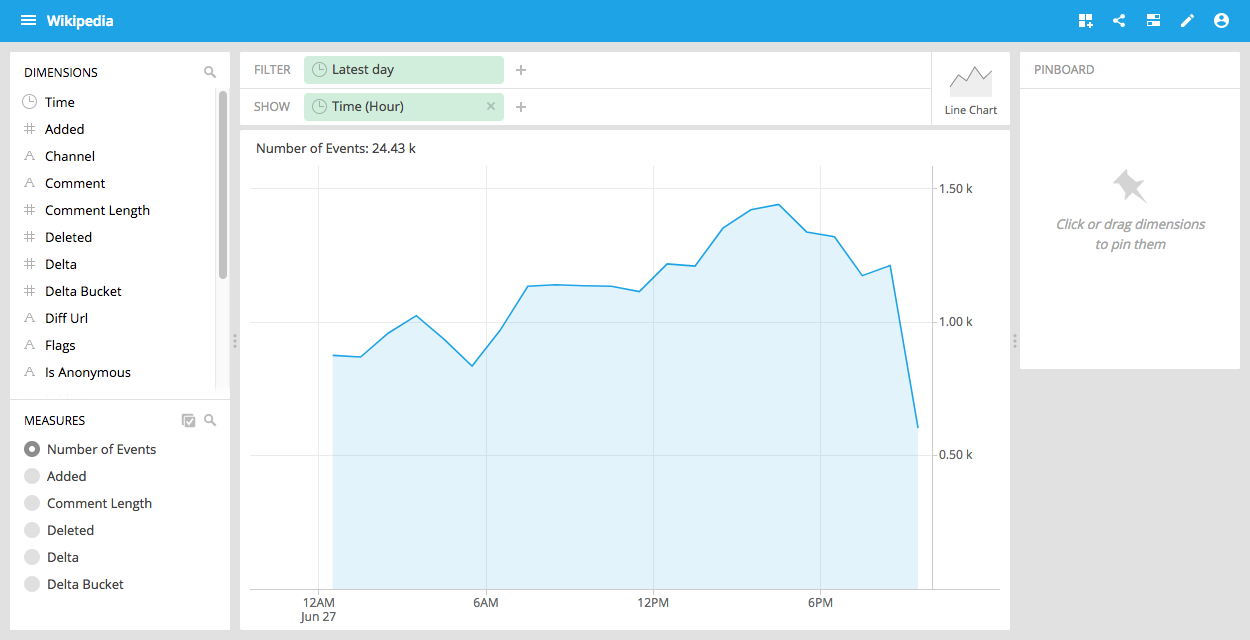


在这里，您可以通过过滤并在任何维度上拆分数据集来探索数据集。对于数据的每次过滤拆分，您将看到所选度量的总值。

例如，在维基百科数据集上，通过在page上拆分和按事件数排序查看最常编辑的page）。



数据立方体视图根据您分割数据的方式建议不同的可视化。如果拆分字符串列，则数据最初将显示为表格。如果按时间拆分，数据立方体视图将推荐时间序列图，如果在数字列上拆分，则会得到条形图。



* 1. 运行SQL

访问SQL编辑器。

SELECT page, COUNT(\*) AS Edits

FROM wikipedia

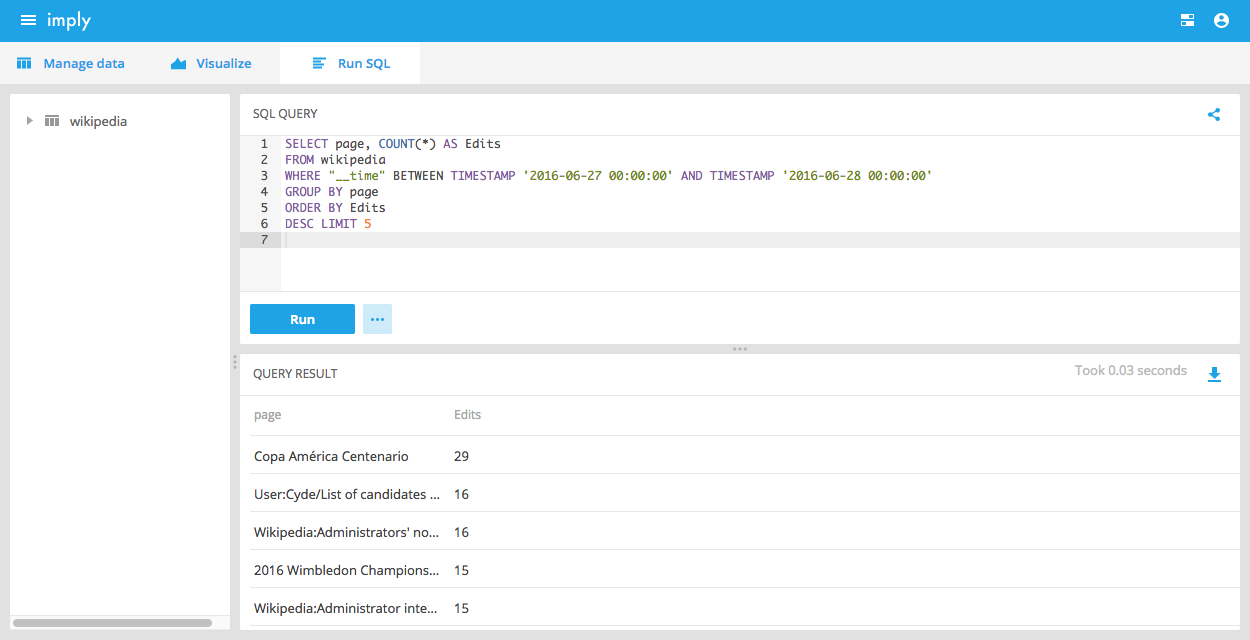
WHERE "\_\_time" BETWEEN TIMESTAMP '2016-06-27 00:00:00' AND TIMESTAMP '2016-06-28 00:00:00'

GROUP BY page

ORDER BY Edits

DESC LIMIT 5

您应该看到如下结果：



1. 数据摄入

6.1 数据格式

1. 摄入规范化数据：JSON、CSV、TSV
2. 自定义格式
3. 其他格式

6.2 配置

主要是摄入的规则ingestion Spec  
摄入规则主要包含3个部分

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Description | Required |
| dataSchema | JSON Object | 标识摄入数据的schema，不同specs可共享 | yes |
| ioConfig | JSON Object | 标识data从哪来，到哪去。根据不同的ingestion method不同 | yes |
| tuningConfig | JSON Object | 标识如何调优不同的ingestion parameters。根据不同的ingestion method不同 | no |

{

"dataSchema" : {...},

"ioConfig" : {...},

"tuningConfig" : {...}

}

6.2.1 DataSchema

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Description | Required |
| dataSource | String | 要摄入的datasource名称，Datasources可看做为表 | yes |
| parser | JSON Object | ingested data如何解析 | yes |
| metricsSpec | JSON Object array | [aggregators](http://druid.io/docs/0.10.1/querying/aggregations.html)器列表 | yes |
| granularitySpec | JSON Object | 指定segment的存储粒度和查询粒度 | yes |

1、parser

（1）string parser

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Description | Required |
| type | String | 一般为string，或在Hadoop indexing job中使用hadoopyString | no |
| parseSpec | JSON Object | 标识格式format和、imestamp、dimensions | yes |

parseSpec两个功能:

* String Parser用parseSpec判定将要处理rows的数据格式（ JSON, CSV, TSV）
* 所有的Parsers 用parseSpec判定将要处理rows的 timestamp  和 dimensionsAll

format字段默认为tsv格式

JSON ParseSpec

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Description | Required |
| format | String | json | no |
| timestampSpec | JSON Object | 指明时间戳列名和格式 | yes |
| dimensionsSpec | JSON Object | 指明维度的设置 | yes |
| flattenSpec | JSON Object | 若json有嵌套层级，则需要指定 | no |

CSV ParseSpec

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Description | Required |
| format | String | csv. | yes |
| timestampSpec | JSON Object | 指明时间戳列名和格式 | yes |
| dimensionsSpec | JSON Object | 指明维度的设置 | yes |
| listDelimiter | String | 多值dimensions的分割符 | no (default == ctrl+A) |
| columns | JSON array | csv的数据列名 | yes |

TSV ParseSpec

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Description | Required |
| format | String | tsv. | yes |
| timestampSpec | JSON Object | 指明时间戳列名和格式 | yes |
| dimensionsSpec | JSON Object | 指明维度的设置 | yes |
| listDelimiter | String | 多值dimensions的分割符 | no (default == ctrl+A) |
| columns | JSON array | tsv的数据列名 | yes |
| delimiter | String | 数据之间的分隔符，默认是\t | no |

TimestampSpec

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Description | Required |
| column | String | timestamp的列 | yes |
| format | String | iso, millis, posix, auto or Joda time，时间戳格式 | no (default == 'auto') |

DimensionsSpec

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Description | Required |
| dimensions | JSON array | dimension schema 对象或dimension names，标识维度列，否则将timestamp列外的所以string列作为维度列 | yes |
| dimensionExclusions | JSON String array | 剔除的维度名列表 | no (default == []) |
| spatialDimensions | JSON Object array | 空间维度名列表，主要用于地理几何运算 | no (default == []) |

2、metricsSpec

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Description | Required |
| dimensions | string | count，longSum等聚合函数类型 | yes |
| fieldName | string | 聚合函数运用的列名 | no |
| name | string | 聚合后指标的列名 | yes |

一些简单的聚合函数：

count 、longSum、longMin、longMax、doubleSum、doubleMin、doubleMax

3、GranularitySpec

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Description | Required |
| type | string | uniform | yes |
| segmentGranularity | string | segment的存储粒度，HOUR DAY等 | yes |
| queryGranularity | string | 最小查询粒度 MINUTE HOUR | yes |
| intervals | JSON Object array | 输入数据的时间段，可选，对于流式数据pull方式而言可以忽略 | no |

"dataSchema" : {

**"dataSource"** : "wikipedia",

**"parser"** : {

"type" : "string",

"parseSpec" : {

"format" : "json",

"dimensionsSpec" : {

"dimensions" : [

"channel",

"cityName",

"comment",

"countryIsoCode",

"countryName",

"isAnonymous",

"isMinor",

"isNew",

"isRobot",

"isUnpatrolled",

"metroCode",

"namespace",

"page",

"regionIsoCode",

"regionName",

"user",

{ "name" : "commentLength", "type" : "long" },

{ "name" : "deltaBucket", "type" : "long" },

"flags",

"diffUrl",

{ "name": "added", "type": "long" },

{ "name": "deleted", "type": "long" },

{ "name": "delta", "type": "long" }

]

},

"timestampSpec": {

"column": "timestamp",

"format": "iso"

}

}

},

**"metricsSpec"** : [],

**"granularitySpec"** : {

"type" : "uniform",

"segmentGranularity" : "day",

"queryGranularity" : "none",

"intervals" : ["2016-06-27/2016-06-28"],

"rollup" : false

}

}

6.2.2 ioConfig

ioConfig  指明了真正具体的数据源

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Description | Required |
| type | String | always be 'realtime'. | yes |
| firehose | JSON Object | 指明数据源，例如本地文件 kafka | yes |
| plumber | JSON Object | Where the data is going. | yes |

不同的firehose的格式不太一致，以kafka为例

{

    firehose : {

        consumerProps : {

            auto.commit.enable : false

            auto.offset.reset : largest

            fetch.message.max.bytes : 1048586

            group.id : druid-example

            zookeeper.connect : localhost:2181

                       zookeeper.connect.timeout.ms : 15000

                        zookeeper.session.timeout.ms : 15000

                        zookeeper.sync.time.ms : 5000

        },

        feed : wikipedia

        type : kafka-0.8

    }

}

ioConfig 的案例：

"ioConfig" : {

"type" : "index",

"firehose" : {

"type" : "local",

"baseDir" : "quickstart/",

"filter" : "wikipedia-2016-06-27-sampled.json"

},

"appendToExisting" : false

}

6.2.3 tuningConfig

tuningConfig 这部分配置是优化数据输入的过程

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Description | Required |
| type | String | realtime | no |
| maxRowsInMemory | Integer | 在存盘之前内存中最大的存储行数，指的是聚合后的行数indexing所需Maximum heap memory= maxRowsInMemory \* (2 + maxPendingPersists). | no (default == 75000) |
| windowPeriod | ISO 8601 Period String | 默认10分钟，最大可容忍时间窗口，超过窗口，数据丢弃 | no (default == PT10m) |
| intermediatePersistPeriod | ISO8601 Period String | 多长时间数据临时存盘一次 | no (default == PT10m) |
| basePersistDirectory | String | 临时存盘目录 | no (default == java tmp dir) |
| versioningPolicy | Object | 如何为segment设置版本号 | no (default == based on segment start time) |
| rejectionPolicy | Object | 数据丢弃策略 | no (default == 'serverTime') |
| maxPendingPersists | Integer | 最大同时存盘请求数，达到上限，输入将会暂停 | no (default == 0) |
| shardSpec | Object | 分片设置 | no (default == 'NoneShardSpec') |
| buildV9Directly | Boolean | 是否直接构建V9版本的索引 | no (default == true) |
| persistThreadPriority | int | 存盘线程优先级 | no (default == 0) |
| mergeThreadPriority | int | 存盘归并线程优先级 | no (default == 0) |
| reportParseExceptions | Boolean | 是否汇报数据解析错误 | no (default == false) |

"tuningConfig" : {

"type" : "index",

"targetPartitionSize" : 5000000,

"maxRowsInMemory" : 25000,

"forceExtendableShardSpecs" : true

}

6.3 从hadoop加载数据

6.3.1加载数据

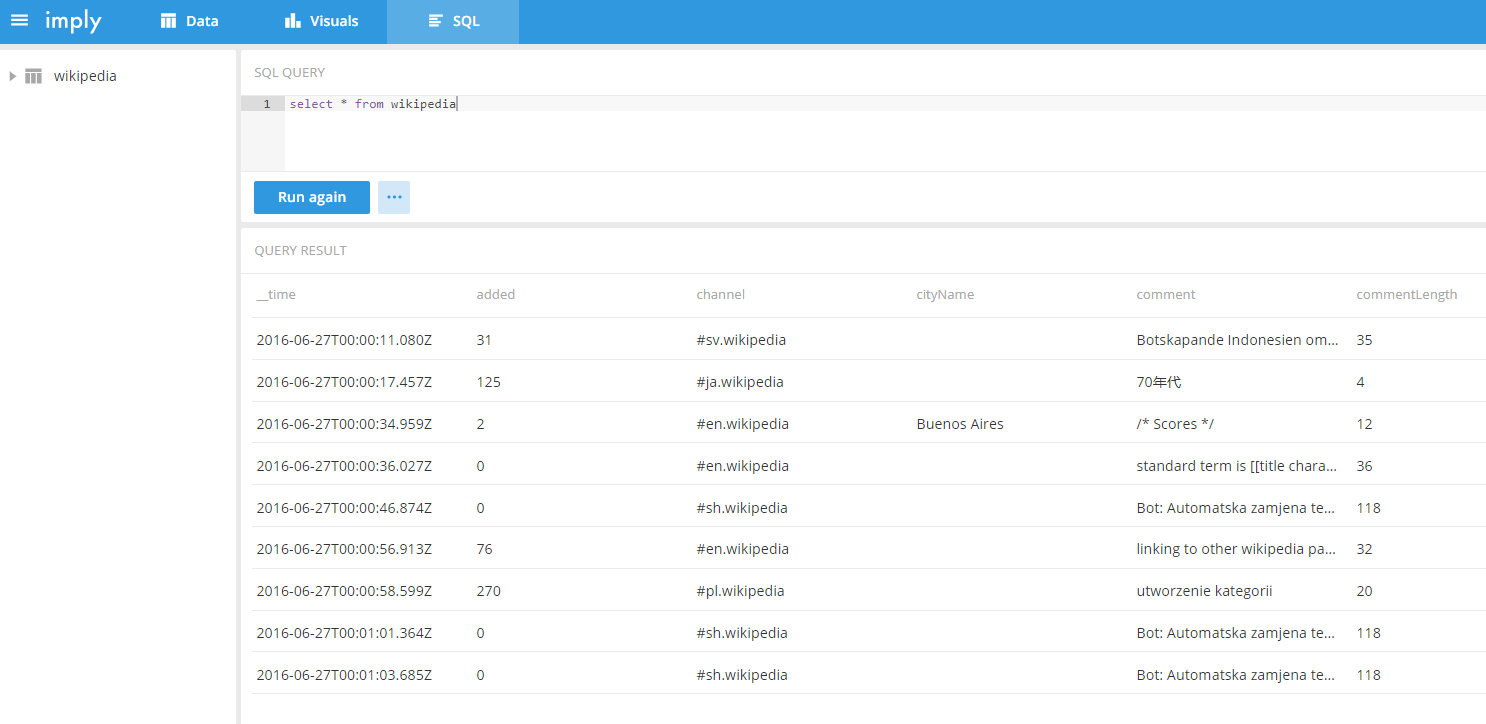
批量摄取维基百科样本数据，文件位于quickstart/wikipedia-2016-06-27-sampled.json。使用quickstart/wikipedia-index-hadoop.json 摄取任务文件。

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

此命令将启动Druid Hadoop摄取任务。

摄取任务完成后，数据将由历史节点加载，并可在一两分钟内进行查询。

6.3.2查询数据



6.4 从kafka加载数据

6.4.1 准备kafka

1.启动kafka

[atguigu@hadoop102 kafka]$ bin/kafka-server-start.sh config/server.properties

[atguigu@hadoop103 kafka]$ bin/kafka-server-start.sh config/server.properties

[atguigu@hadoop104 kafka]$ bin/kafka-server-start.sh config/server.properties

2.创建wikipedia主题

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --zookeeper hadoop102:2181 –topic

wikipedia --partitions 1 --replication-factor 1 –create

Created topic "wikipedia".

3.查看主题是否创建成功

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --zookeeper hadoop102:2181 --list

\_\_consumer\_offsets

first

wikipedia

6.4.2 启动索引服务

我们将使用Druid的Kafka索引服务从我们新创建的维基百科主题中提取消息。要启动该服务，我们需要通过从Imply目录运行以下命令向Druid的overlord提交supervisor spec

[atguigu@hadoop102 imply-2.7.10]$ curl -XPOST -H'Content-Type: application/json' -d

@quickstart/wikipedia-kafka-supervisor.json http://hadoop102:8090/druid/indexer/v1/supervisor

 说明：

curl是一个利用URL规则在命令行下工作的文件传输工具。它支持文件的上传和下载，所以是综合传输工具。

* -X 为 HTTP 数据包指定一个方法，比如 PUT、DELETE。默认的方法是 GET 6.4.3
* -H 为 HTTP 数据包指定 Header 字段内容
* -d 为 POST 数据包指定要向 HTTP 服务器发送的数据并发送出去，如果<data>的内容以符号 @ 开头，其后的字符串将被解析为文件名，curl 命令会从这个文件中读取数据发送。

6.4.3 加载历史数据

启动kafka生产者生产数据

[atguigu@hadoop102 kafka]$ bin/kafka-console-producer.sh --broker-list hadoop102:9092 –

-topic wikipedia < /opt/module/imply-2.7.10/quickstart/wikipedia-2016-06-27-sampled.json

说明：

< 将文件作为命令输入

可在kafka本地看到相应的数据生成

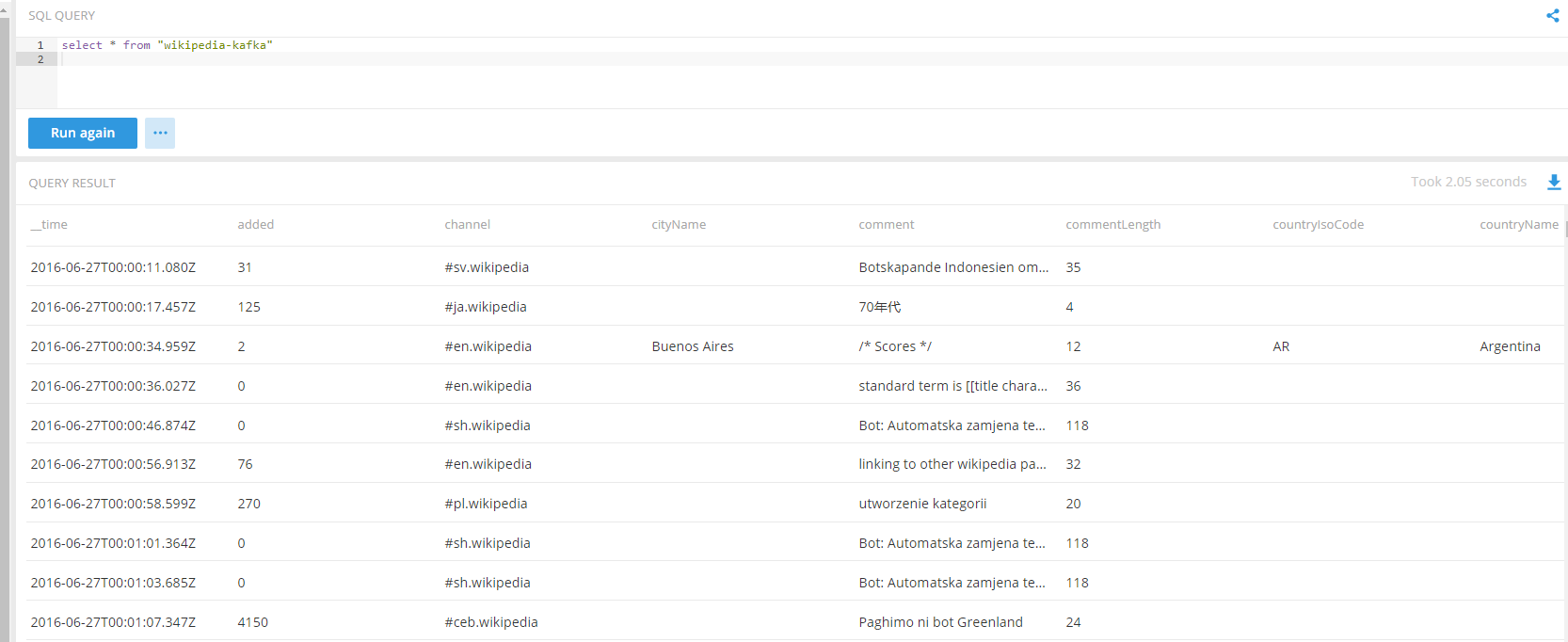
[atguigu@hadoop103 logs]$ ll

drwxrwxr-x. 2 atguigu atguigu 4096 3月 30 11:16 wikipedia-0

[atguigu@hadoop103 logs]$ pwd

/opt/module/kafka/logs

将样本事件发布到Kafka的wikipedia主题，然后由Kafka索引服务将其提取到Druid中。你现在准备运行一些查询了！



6.4.4 加载实时数据

下载一个帮助应用程序，该应用程序将解析维基媒体的IRC提要中的event，并将这些event发布到我们之前设置的Kafka的wikipedia主题中。

[atguigu@hadoop102 imply-2.7.10]$ curl -O

https://static.imply.io/quickstart/wikiticker-0.4.tar.gz

说明：

-O 在本地保存获取的数据时，使用她们在远程服务器上的文件名进行保存。

[atguigu@hadoop102 imply-2.7.10]$ tar -zxvf wikiticker-0.4.tar.gz

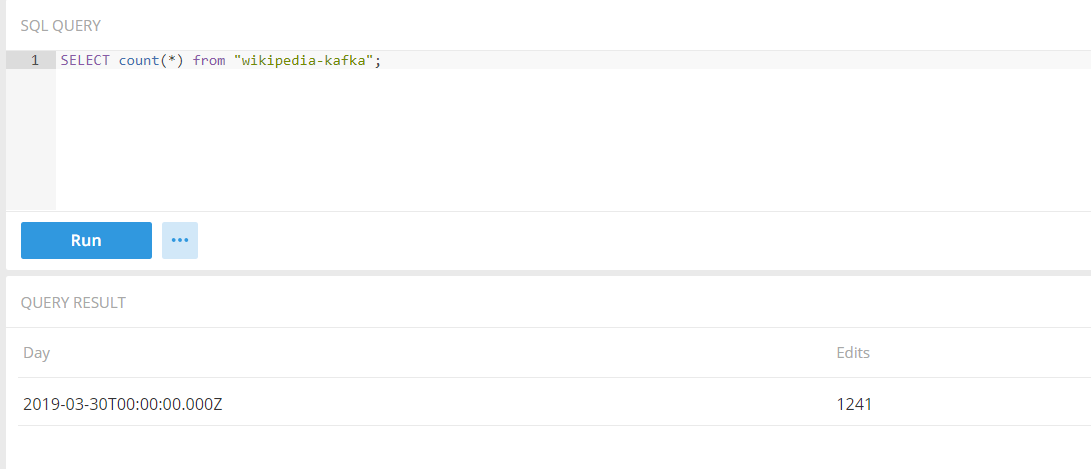
[atguigu@hadoop102 imply-2.7.10]$ cd wikiticker-0.4

现在运行带有参数的wikiticker，指示它将输出写入我们的Kafka主题：

[atguigu@hadoop102 wikiticker-0.4]$ bin/wikiticker -J-Dfile.encoding=UTF-8 -out kafka –

topic Wikipedia

查询多次，对比结果的变化



6.4.5 加载自定义kafka主题数据

可以通过编写自定义supervisor spec来加载自己的数据集。要自定义受监督的Kafka索引服务提取，您可以将包含的quickstart/wikipedia-kafka-supervisor.json 规范复制到自己的文件中，根据需要进行编辑，并根据需要创建或关闭管理程序。没有必要自己重启Imply或Druid服务。

1. Druid数据摄入之Tranquility

在上面的内容中，我们学习了如何通过索引服务器摄取数据，然而索引服务器的API太过底层，运用起来比较麻烦。Tranquility对索引服务的API进行了封装，可以方便创建任务，处理分片，复制、服务器发现以及无缝的数据结构调整。

向Druid发送数据，需将Tranquility库依赖到程序中。

地址：<https://github.com/druid-io/tranquility>

Tranquility提供了两种Core API：

* Tranquilizer，一个高级API，发送单个message。
* Beam，一种低级API，发送批次数据。

7.1 高级API

7.1.1环境准备

依赖

<dependencies>

<dependency>

<groupId>io.druid</groupId>

<artifactId>tranquility-core\_2.11</artifactId>

<version>0.8.2</version>

</dependency>

<dependency>

<groupId>com.metamx</groupId>

<artifactId>java-util</artifactId>

<version>1.3.2</version>

</dependency>

</dependencies>

数据格式的配置：

{

"dataSources": [

{

"spec": {

"dataSchema": {

"dataSource": "wikipedia02",

"parser": {

"type": "string",

"parseSpec": {

"format": "json",

"timestampSpec": {

"column": "timestamp",

"format": "auto"

},

"dimensionsSpec": {

"dimensions": [

"page",

"language",

"user",

"unpatrolled",

"newPage",

"robot",

"anonymous",

"namespace",

"continent",

"country",

"region",

"city"

],

"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"

}

},

"tuningConfig": {

"type": "realtime",

"maxRowsInMemory": 100000,

"intermediatePersistPeriod": "PT10m",

"windowPeriod": "PT10M"

}

}

}

],

"properties": {

"zookeeper.connect": "hadoop102:2181",

"druid.selectors.indexing.serviceName": "druid/overlord",

"druid.discovery.curator.path": "/druid/discovery",

"druidBeam.taskLocator": "overlord",

"druidBeam.overlordPollPeriod": "PT5S"

}

}

7.1.2 Java代码编写

package com.atguigu.tranquility;

import com.google.common.collect.ImmutableMap;

import com.metamx.common.logger.Logger;

import com.metamx.tranquility.config.DataSourceConfig;

import com.metamx.tranquility.config.PropertiesBasedConfig;

import com.metamx.tranquility.config.TranquilityConfig;

import com.metamx.tranquility.druid.DruidBeams;

import com.metamx.tranquility.tranquilizer.MessageDroppedException;

import com.metamx.tranquility.tranquilizer.Tranquilizer;

import com.twitter.util.FutureEventListener;

import org.joda.time.DateTime;

import scala.runtime.BoxedUnit;

import java.io.InputStream;

import java.util.Map;

public class JavaExample {

private static final Logger log = new Logger(JavaExample.class);

public static void main(String[] args)

{

// Read config from "example.json" on the classpath.

final InputStream configStream = JavaExample.class.getClassLoader().getResourceAsStream("example.json");

final TranquilityConfig<PropertiesBasedConfig> config = TranquilityConfig.read(configStream);

final DataSourceConfig<PropertiesBasedConfig> wikipediaConfig = config.getDataSource("wikipedia02");

final Tranquilizer<Map<String, Object>> sender = DruidBeams.fromConfig(wikipediaConfig)

.buildTranquilizer(wikipediaConfig.tranquilizerBuilder());

sender.start();

try {

// Send 10000 objects

for (int i = 0; i < 10; i++) {

// Build a sample event to send; make sure we use a current date

final Map<String, Object> obj = ImmutableMap.<String, Object>of(

"timestamp", new DateTime().toString(),

"page", "foo",

"added", i

);

// Asynchronously send event to Druid:

sender.send(obj).addEventListener(

new FutureEventListener<BoxedUnit>()

{

@Override

public void onSuccess(BoxedUnit value)

{

log.info("Sent message: %s", obj);

}

@Override

public void onFailure(Throwable e)

{

if (e instanceof MessageDroppedException) {

log.warn(e, "Dropped message: %s", obj);

} else {

log.error(e, "Failed to send message: %s", obj);

}

}

}

);

}

}

finally {

sender.flush();

sender.stop();

}

}

}

7.1.3 scala代码编写

package com.atguigu.tranquility

import com.metamx.common.scala.Logging

import com.metamx.tranquility.config.DataSourceConfig

import com.metamx.tranquility.config.PropertiesBasedConfig

import com.metamx.tranquility.config.TranquilityConfig

import com.metamx.tranquility.druid.DruidBeams

import com.metamx.tranquility.tranquilizer.MessageDroppedException

import com.metamx.tranquility.tranquilizer.Tranquilizer

import com.twitter.util.Return

import com.twitter.util.Throw

import org.joda.time.DateTime

import scala.collection.JavaConverters.\_

object ScalaExample extends Logging

{

def main(args: Array[String]) {

// Read config from "example.json" on the classpath.

val configStream = getClass.getClassLoader.getResourceAsStream("example.json")

val config: TranquilityConfig[PropertiesBasedConfig] = TranquilityConfig.read(configStream)

val wikipediaConfig: DataSourceConfig[PropertiesBasedConfig] = config.getDataSource("wikipedia03")

val sender: Tranquilizer[java.util.Map[String, AnyRef]] = DruidBeams

.fromConfig(config.getDataSource("wikipedia03"))

.buildTranquilizer(wikipediaConfig.tranquilizerBuilder())

sender.start()

try {

// Send 10000 objects.

for (i <- 0 until 10) {

val obj = Map[String, AnyRef](

"timestamp" -> new DateTime().toString,

"page" -> "foo",

"added" -> Int.box(i)

)

// Asynchronously send event to Druid:

sender.send(obj.asJava) respond {

case Return(\_) =>

log.info("Sent message: %s", obj)

case Throw(e: MessageDroppedException) =>

log.warn(e, "Dropped message: %s", obj)

case Throw(e) =>

log.error(e, "Failed to send message: %s", obj)

}

}

}

finally {

sender.flush()

sender.stop()

}

}

}

7.2 低级API

Druid 整合SparkStreaming

7.2.1 环境准备

<dependencies>

<dependency>

<groupId>io.druid</groupId>

<artifactId>tranquility-core\_2.11</artifactId>

<version>0.8.2</version>

</dependency>

<dependency>

<groupId>com.metamx</groupId>

<artifactId>java-util</artifactId>

<version>1.3.2</version>

</dependency>

<dependency>

<groupId>io.druid</groupId>

<artifactId>tranquility-spark\_2.11</artifactId>

<version>0.8.2</version>

</dependency>

<dependency>

<groupId>org.apache.spark</groupId>

<artifactId>spark-streaming\_2.11</artifactId>

<version>2.1.1</version>

</dependency>

</dependencies>

7.2.2 代码编写

class SimpleEventBeamFactory extends BeamFactory[SimpleEvent]

{

// Return a singleton, so the same connection is shared across all tasks in the same JVM.

def makeBeam: Beam[SimpleEvent] = SimpleEventBeamFactory.BeamInstance

}

object SimpleEventBeamFactory

{

val BeamInstance: Beam[SimpleEvent] = {

// Tranquility uses ZooKeeper (through Curator framework) for coordination.

val curator = CuratorFrameworkFactory.newClient(

"localhost:2181",

new BoundedExponentialBackoffRetry(100, 3000, 5)

)

curator.start()

val indexService = "druid/overlord" // Your overlord's druid.service, with slashes replaced by colons.

val discoveryPath = "/druid/discovery" // Your overlord's druid.discovery.curator.path

val dataSource = "foo"

val dimensions = IndexedSeq("bar")

val aggregators = Seq(new LongSumAggregatorFactory("baz", "baz"))

val isRollup = true

// Expects simpleEvent.timestamp to return a Joda DateTime object.

DruidBeams

.builder((simpleEvent: SimpleEvent) => simpleEvent.timestamp)

.curator(curator)

.discoveryPath(discoveryPath)

.location(DruidLocation(indexService, dataSource))

.rollup(DruidRollup(SpecificDruidDimensions(dimensions), aggregators, QueryGranularities.MINUTE, isRollup))

.tuning(

ClusteredBeamTuning(

segmentGranularity = Granularity.HOUR,

windowPeriod = new Period("PT10M"),

partitions = 1,

replicants = 1

)

)

.buildBeam()

}

}

// Add this import to your Spark job to be able to propagate events from any RDD to Druid

import com.metamx.tranquility.spark.BeamRDD.\_

// Now given a Spark DStream, you can send events to Druid.

dstream.foreachRDD(rdd => rdd.propagate(new SimpleEventBeamFactory))