

# Intelligent enGINE X: IGinX

## 用户手册-0.7.1

清华数为·多源异构数据一体化管理分析系统

清华大学 软件学院

大数据系统软件国家工程实验室

2024 年 8 月

## 目录

一、 IGINX 简介.....	6
1.1 系统特色.....	6
1.2 功能特点.....	6
1.3 系统架构.....	7
1.4 应用场景.....	7
二、 快速上手 .....	9
2.1 安装环境.....	9
2.2 基于发布包的安装与部署方法 .....	9
2.3 基于源码的安装与部署方法.....	9
2.4 参数配置.....	10
2.5 交互方式.....	11
2.5.1 API 交互.....	12
2.5.2 RESTful 交互.....	12
2.5.3 基于客户端的SQL 交互.....	12
三、 SQL 定义与使用.....	14
3.1 使用 IGINX-SQL.....	14
3.2 数据相关操作.....	14
3.2.1 插入数据.....	14
3.2.1.1 直接向指定的列插入数据.....	14
3.2.1.2 通过子查询插入数据.....	15
3.2.1.3 读取 csv 文件.....	15
3.2.2 删除数据.....	16
3.2.3 查询数据.....	16
3.2.3.1 范围查询.....	18
3.2.3.2 值过滤查询.....	18
3.2.3.2.1 带 UDF 的值过滤查询.....	18
3.2.3.3 聚合查询.....	18
3.2.3.4 降采样查询.....	18
3.2.3.5 数量限制查询.....	19
3.2.3.6 序列比较查询.....	19
3.2.3.6.1 带 UDF 的序列比较查询.....	19
3.2.3.7 模糊查询.....	20
3.2.3.8 序列重命名.....	20
3.2.3.9 嵌套查询.....	21
3.2.3.9.1 SELECT 子查询.....	21
3.2.3.9.2 FROM 子查询.....	21
3.2.3.9.3 WHERE 子查询.....	22
3.2.3.9.3.1 EXIST 子查询.....	22
3.2.3.9.3.2 IN 子查询.....	23
3.2.3.9.3.3 ALL 子查询.....	23
3.2.3.9.3.4 SOME(ANY)子查询.....	23
3.2.3.9.3.5 用标量子查询进行比较.....	24
3.2.3.9.4 HAVING 子查询.....	24
3.2.3.9.5 INSERT 子查询.....	24
3.2.3.10 查询序列.....	24
3.2.3.11 统计数据总量.....	25

3.2.3.12 清除数据 .....	25
3.2.3.13 Explain 查询 .....	25
3.2.3.14 分组查询 .....	25
3.2.3.15 查询排序 .....	27
3.2.3.16 集合操作 .....	30
3.2.3.17 DISTINCT 关键字 .....	31
3.2.3.18 SQL 查询结果导出到文件 .....	31
3.2.3.18.1 导出为 csv .....	31
3.2.3.18.1 以字节流形式导出 .....	32
3.2.3.19 公用表表达式 (WITH 子句) .....	32
3.2.3.20 VALUE2META .....	32
3.2.3.21 关联 (Join) 查询 .....	33
3.2.3.21.1 IGINX 内关联 (Join) 概念的说明 .....	33
3.2.3.21.2 支持的关联类型 .....	33
3.2.3.21.2.1 左连接 (left outer join) .....	34
3.2.3.21.2.2 右连接 (right outer join) .....	34
3.2.3.21.2.3 全连接 (full outer join) .....	34
3.2.3.21.2.4 内连接 (inner join) .....	34
3.2.3.21.2.5 自然连接 (natural join) .....	34
3.2.3.21.2.6 笛卡尔积 (cross join) .....	35
3.2.3.21.3 在关联查询中使用嵌套查询 .....	35
3.3 系统相关操作 .....	35
3.3.1 增加存储引擎 .....	35
3.3.2 查询副本数量 .....	40
3.3.3 查询集群信息 .....	40
3.3.4 用户权限管理 .....	40
3.3.5 移除堆叠分片 .....	41
3.3.6 配置管理 .....	42
3.3.7 查询当前的 session id .....	42
3.3.8 优化规则管理 .....	43
四、 基于 SQL 支持 TAGKV 定义 .....	44
4.1 概述 .....	44
4.2 语法 .....	44
4.2.1 插入数据 .....	44
4.2.2 查询数据 .....	44
4.3 例子 .....	45
五、 基于 PYTHON 的数据处理流程: TRANSFORM .....	48
5.1 环境准备 .....	48
5.2 脚本编写与注册 .....	48
5.3 运行 TRANSFORM 作业 .....	49
5.3.1 方式 1: IGINX-SQL .....	50
参数说明: .....	51
5.3.2 方式 2: IGINX-Session .....	52
六、 基于 PYTHON 的用户定义函数: UDF .....	54
6.1 概述 .....	54
6.1.1 UDTF (user-defined time series function) .....	54
6.1.2 UDAF (user-defined aggregate function) .....	54

6.1.3 UDSF (user-defined set transform function)	55
6.2 环境准备	55
6.3 UDF 编写	56
6.4 UDF 注册	57
6.4.1 方式1: 注册	57
6.4.2 方式2: 配置文件	58
6.5 使用示例	59
6.5.1 UDTF	60
6.5.2 UDAF	61
6.5.3 UDSF	61
6.5.4 例: 基于UDTF 实现不定长参数的函数进行查询	62
6.5.4.1 Multiply 实现	62
6.5.4.2 使用上述函数定义进行查询的例子	62
七、 RESTFUL 访问接口	64
7.1 定义	64
7.2 描述	65
7.2.1 写入操作	65
7.2.1.1 插入数据点	65
7.2.2 查询操作	66
7.2.2.1 查询时间范围内的数据	66
7.2.2.2 包含聚合的查询	67
7.2.3 删除操作	77
7.2.3.1 删除数据点	77
7.2.3.2 删除 metric	78
7.3 特性	78
7.4 性能	78
八、 基于 RESTFUL 实现序列段打标记	79
8.1 添加标记	79
8.2 更新标记	80
8.3 查询标记	81
8.4 删除标记	82
九、 数据访问接口	84
9.1 定义	84
9.2 描述	86
9.3 特性	89
9.4 性能	89
十、 新版数据访问接口	90
10.1 安装方法	90
10.2 接口说明	90
10.3 初始化	90
10.4 数据写入	91
10.4.1 数据点	92
10.4.2 数据行	93
10.4.3 数据表	93
10.4.4 数据对象	94

10.5 数据查询	95
10.5.1 一般查询	96
10.5.2 对象映射	97
10.5.3 流式读取	97
10.6 数据删除	98
10.7 用户管理	99
10.8 集群操作	99
十一、存储扩容功能	101
11.1 IGINX 扩容操作	101
11.2 底层数据库扩容操作	101
11.3 数据库扩容操作参数说明	101
11.3.1 Parquet 存储	101
11.3.2 文件目录存储	102
11.3.3 MongoDB 对接层	103
11.3.4 Redis 对接层	103
11.3.5 Relational 对接层	104
十二、数据源接入功能	105
12.1 功能简介	105
12.2 配置参数	105
12.3 运行说明	106
十三、多数据库/存储扩展实现	107
13.1 需支持的功能	107
13.2 接口	108
13.3 异构数据库部署操作	108
13.4 同数据库多版本实现	109
十四、导入导出 CSV 工具	110
14.1 导出 CSV 工具	110
14.1.1 运行方法	110
14.1.2 运行示例	111
14.1.3 注意事项	111
14.2 导入 CSV 工具	112
14.2.1 运行方法	112
14.2.2 运行示例	112
14.2.3 注意事项	113
十五、曲线匹配功能	114
15.1 原理简介	114
15.2 使用说明	114
15.2.1 接口名称	114
15.2.2 参数介绍	114
15.2.3 应用示例	114
十六、部署指导原则	116
16.1 边缘端部署原则	116

16.2 云端部署原则 .....	116
十七、常见问题 .....	117
17.1 如何知道当前有哪些 IGINX 节点? .....	117
17.2 如何知道当前有哪些时序数据库节点? .....	117
17.3 如何知道数据分片当前有几个副本? .....	117
17.4 如何加入 IGINX 的开发, 成为 IGINX 代码贡献者? .....	119
17.5 IGINX 集群版管理时序数据与 IoTDB-RAFT 版相比, 各自特色在何处? .....	119

# 一、IGinX 简介

世界上越来越多的企业意识到生产过程中的实时数据与历史数据是最有价值的信息财富，也是整个企业信息系统的核心和基础。随着工业互联网时代的到来，实时数据和历史数据的体量越来越大，历史数据积累的体量也越来越大，数据类型越来越丰富，单一的数据库系统都已经无法满足工业数据管理的全面需求——“*No one size fits all*”。我们可以见到，业界对于高可扩展多元大数据管理系统的需求越来越迫切。

二十年来，我们一直致力于企业信息及企业数据管理的相关工作，为满足上述需求，基于丰富的业界经验，在构建了非结构化数据管理系统、并于近年来打造了一款打破了国际著名测试基准组织 TPC 的 TPCx-IoT 基准性能榜榜首记录的分布时序数据库的基础上，精心打造出了一款高可扩展多元大数据管理系统 IGINX，实现异构多源数据的一体化管理分析。

## 1.1 系统特色

IGINX 当前发布版本，其主要特色包括：

1. **【平滑扩展能力】**在有高速写入和查询的条件下，可几乎不影响负载地进行数据库节点扩容。
2. **【快速响应负载能力】**IGINX 计算层组件采用无状态设计，可以快速响应多变的应用负载，也因此可以在资源允许的条件下、很好地确保体现出底层存储单机的高性能。
3. **【多副本下的高性能】**面向时序场景的大数据特性，副本基于弱一致性的多写多读实现，可以避开强一致性算法导致的性能问题。
4. **【异构数据源统一模型】**兼容 InfluxDB、IoTDB、PostgreSQL/TimescaleDB 等时序/关系数据库，以统一的数据模型，同时管理多种异构数据库，只需针对这些数据库实现一个简单接口。
5. **【数据分布可定制】**支持灵活分片，可以通过策略配置，来支持灵活的数据分布模式，实现非对称、多粒度、定制化的数据分布。

## 1.2 功能特点

IGINX 采用迭代周期式开发流程，目前已经完成以下版本的发布：首发版 v0.1.0，健壮版 v0.2.0、破世界记录版 v0.3.0、上线服务版 v0.4.0、技术引领版 v0.5.0。

- **【首发版 v0.1.0】**支持典型的工业互联网边缘端数据管理需求，即满足 TPCx-IoT 测试所对应的相关应用需求。
- **【健壮版 v0.2.0】**支持单机版时序数据库，尤其是 IoTDB，的数据访问功能全集。
- **【破世界记录版 v0.3.0】**性能打破国际数据库测试基准顶尖组织 TPC 的 TPCx-IoT 当前世界记录，取得 397.1 万 IoTps 以上的性能，比当前世界记录提升 16%。

- **【上线服务版 v0.4.0】**以 5 节点 IGINX+2 节点数据库，形成双副本，线上支持中车四方上海地铁日常监控数据管理；以 7 节点 IGINX+3 节点数据库，形成双副本，线上支持中车四方成都地铁日常监控数据管理；以 3 节点 IGINX+3 节点数据库，形成三副本，基于 MQTT 模式，线上支持商飞 5G 中心对时序数据的管理。
- **【技术引领版 v0.5.0】**增加 SQL 子查询能力、内置/外置 Python 计算能力、跨时序与关系数据库管理/计算能力、Annotation 标记能力、曲线匹配能力等当前绝大部分时序数据库都不具备的能力，方便支持用户多样化的应用需求。ss

同时，IGINX 支持多种不同的元数据管理模式，即 ZooKeeper 模式、Etcd 模式等，从而使得 IGINX 可以分布式部署，还可以与云上系统无缝接合、原生部署。

### 1.3 系统架构

IGINX 的整体框架视图如图 1.1 所示。自底向上，可以分成 2 层，下层是可以不相互关联通讯的单机版时序数据库集群层，上层是无状态的 IGINX 服务中间件层。整体框架的相关元数据信息都存储在一个高可用的元数据集群中。这样的架构充分学习、参考了谷歌的 Monarch 时序数据库系统的良好设计理念，以及其久经考验的实践经验。

其中，读写由 IGINX 进行分片解析，发送到底层的数据库进行处理。IGINX 通过元数据集群同步信息并进行配置。数据分片的分配由 IGINX 服务端来实现，并基于元数据集群进行冲突处理。

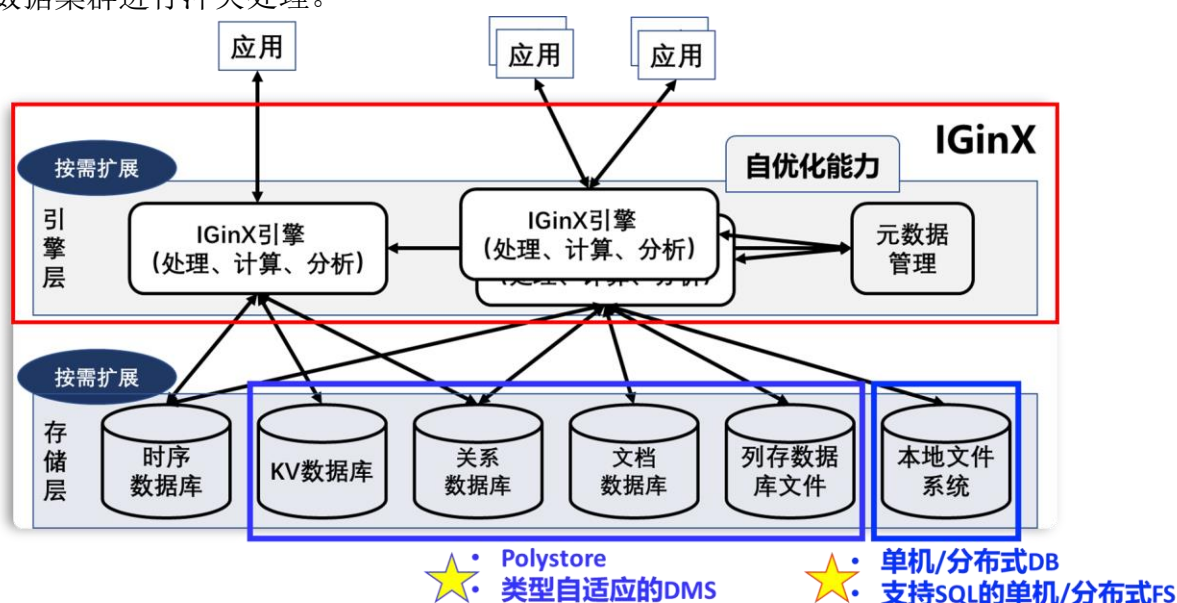


图 1.1 IGINX 总体架构

### 1.4 应用场景

**IGINX 是支持异构数据源“边云”协同、统一管理与原生分析的大数据系统软件。**



它可对关系数据、时序数据、键值数据、文档数据、文件等异构数据源，在无须 ETL 的情况下，实现统一数据模型管理的能力，基于中间层的高可扩展计算能力，实现对大数据，尤其是 IoT 数据的高效管理与计算。

IGinX 可应用于广泛的大数据管理与计算场景，结合用户方的领域知识，可方便支持用户从多样化的大数据中提炼价值。

## 二、快速上手

多源异构数据一体化管理分析系统 IGINX 由两部分构成，一部分是元数据系统 ZooKeeper（可替换为 Etcd），用于存储整个集群的元信息，另一部分是 IGINX 集群系统，使用 Parquet 存储。

### 2.1 安装环境

IGINX 运行时所需的硬件最小配置：

- CPU：单核 2.0Hz 以上
- 内存：4GB 以上
- 网络：100Mbps 以上

IGINX 运行时所依赖的软件配置：

- 操作系统：Linux、MacOS 或 Windows
- JVM：1.8+
- ZooKeeper：3.7.2+

### 2.2 基于发布包的安装与部署方法

JDK 是 Java 程序的开发的运行环境，由于 ZooKeeper 以及 IGINX 都是使用 Java 开发的，因此首先需要安装 Java。下列步骤假设 JAVA 环境已经安装妥当。

1. 下载安装包（要求版本在 3.7.2+）或使用发布包 include 目录下的安装包，解压 ZooKeeper，下载地址为：

<https://zookeeper.apache.org/releases.html>

2. 配置 ZooKeeper（创建文件 conf/zoo.cfg）：

- tickTime=1000
- dataDir=data
- clientPort=2181

3. 启动 ZooKeeper

如果是 Windows 操作系统，使用以下命令：

```
> bin\zkServer.cmd
```

否则，使用以下命令：

```
> bin/zkServer.sh start
```

4. 在 IGINX 配置文件中配置数据库信息等，见章节 2.4

5. 使用以下命令启动一个或多个 IGINX 实例

如果是 Windows 操作系统，则使用以下命令进行启动：

```
> start_iginx.bat
```

否则，使用以下命令：

```
> ./start_iginx.sh
```

### 2.3 基于源码的安装与部署方法

在下列部署步骤之前，要求安装好 Maven 和 Java 运行环境。

1. 安装 Java 运行环境

<https://www.java.com/zh-CN/download>

2. 安装 Maven

<http://maven.apache.org/download.cgi>

然后，按以下步骤进行系统安装部署：

1. 下载并安装 ZooKeeper，要求版本在 3.7.2+

<https://zookeeper.apache.org/releases.html>

2. 配置 ZooKeeper（创建文件 conf/zoo.cfg）

- tickTime=1000
- dataDir=data
- clientPort=2181

3. 启动 ZooKeeper

4. 启动 ZooKeeper

如果是 Windows 操作系统，使用以下命令：

```
> bin\zkServer.cmd
```

否则，使用以下命令：

```
> bin/zkServer.sh start
```

5. 下载 IGINX 源代码项目

<https://github.com/thulab/IGINX/>

6. 编译安装 IGINX

```
> mvn clean install -DskipTests
```

7. 在 IGINX 配置文件中配置数据库信息等，见章节 2.4

8. 使用以下命令启动一个或多个 IGINX 实例

如果是 Windows 操作系统，则使用以下命令进行启动：

```
> start_iginx.bat
```

否则，使用以下命令：

```
> ./start_iginx.sh
```

## 2.4 参数配置

配置文件位于 conf/config.properties，以下为配置文件的内容及其各项的具体含义：

- IGINX 绑定的 ip  
ip=0.0.0.0
- IGINX 绑定的端口  
port=6888
- IGINX 本身的用户名  
username=root
- IGINX 本身的密码  
password=root
- 时序数据库列表，使用 “,” 分隔不同实例  
storageEngineList=127.0.0.1#6667#parquet#dir=/path/to/your/parquet#dummy\_dir=/path/to/your/data#iginx\_port=6888#has\_data=false
- 写入的副本个数，目前建议是 {0,1,2,3}，如果为 0，则没有副本。系统不要求

强一致性，因此在节点个数少于副本个数时，也可正常运行  
replicaNum=0

- 默认写入时间戳的单位，s/ms/us/ns  
timePrecision="ms"
- 底层数据库类名，使用“,”分隔  
databaseClassNames=iotdb12=cn.edu.tsinghua.iginx.iotdb.IoTDBStorage,parquet=cn.edu.tsinghua.iginx.parquet.ParquetStorage
- 逻辑层优化策略  
queryOptimizer=rbo
- 优化器规则  
ruleBasedOptimizer=RemoveNotRule=on,FilterFragmentRule=on
- 约束  
constraintChecker=naïve
- 物理层优化策略  
physicalOptimizer=naive
- 内存任务执行线程池  
memoryTaskThreadPoolSize=200
- 每个存储节点对应的工作线程数  
physicalTaskThreadPoolSizePerStorage=100
- 每个存储节点任务最大堆积数  
maxCachedPhysicalTaskPerStorage=500
- 策略类名  
policyClassName=cn.edu.tsinghua.IGinX.policy.naive.NaivePolicy
- 是否允许通过环境变量设置参数  
enableEnvParameter=false
- rest 绑定的 ip  
restIp=0.0.0.0
- rest 绑定的端口  
restPort=6666
- 是否启用 rest 服务  
enableRestService=true
- rest 异步执行并发数  
asyncRestThreadPool=100
- 元数据后端类型，目前支持 zookeeper, file, etcd  
metaStorage=zookeeper
- 如果使用 zookeeper 作为元数据存储后端，需要提供  
zookeeperConnectionString=127.0.0.1:2181
- 是否开启元数据内存管理  
enableMetaCacheControl=false
- 分片缓存最大内存限制，单位为 KB，默认 128 MB  
fragmentCacheThreshold=131072

## 2.5 交互方式

IGinX 交互一共有 3 种方式。

## 2.5.1 API 交互

第一种是基于 IGINX API 开发的客户端程序，与 IGINX 进行交互（见章节 9），目前在 example 目录下有相关示例程序：

<https://github.com/IGinX-THU/IGinX/tree/main/example>

在使用 Java API 与 IGINX 进行交互时，可以通过源码在本地 Maven 仓库安装依赖包（见章节 2.3）。IGinX 在 Github 中提供了远程 Maven 仓库，也可以直接在 Maven 项目的 pom.xml 文件中添加

```
<repositories>
  <repository>
    <id>iginx</id>
    <name>IGinX GitHub repository</name>
    <url>https://iginx-thu.github.io/IGinX/maven-repo</url>
  </repository>
</repositories>
```

然后添加所需依赖，例如 iginx-session

```
<dependencies>
  <dependency>
    <groupId>cn.edu.tsinghua</groupId>
    <artifactId>iginx-session</artifactId>
    <version>0.7.1</version>
  </dependency>
</dependencies>
```

## 2.5.2 RESTful 交互

第二种是基于 RESTful 接口，与 IGINX 进行交互，主要 RESTful 接口见章节 7。RESTful 命令可通过命令行工具 Curl 执行。

## 2.5.3 基于客户端的 SQL 交互

第三种是基于 IGINX 自带的客户端进行交互（具体命令见章节 9）：

- 基于发布包安装的 IGINX，直接运行 startCli.sh（Windows 环境中应当使用 start\_cli.bat 脚本），即可通过 SQL 命令与 IGINX 交互。
- 基于源码编译安装的 IGINX，则客户端在 IGINX 项目下 client/target/iginx-client-{\$VERSION} 目录下，进入该目录可见 sbin 目录，运行 sbin 目录中的 start\_cli.sh 脚本（Windows 环境中应当使用 start\_cli.bat 脚本），即可通过 SQL 命令与 IGINX 交互。

另外，启动客户端时还可以通过 -h 和 -p 参数指定 IP 和端口，默认为 127.0.0.1 和 6888。

客户端交互界面截图如图 2.1 所示，看到该界面时，即可输入 IGINX 的 SQL 命令进行交互。

```

-----
Starting IGINX Client
-----

  -----  -----  -      --  --
  |_  _| / _ _ _| ( _ )      \ \ / /
    | | | | _ _ _ _ _ _ _ _ \ V /
    | | | | | | | | | | | _ \   > <
  _| | _ | | _ _| | | | | | | | / . \
| _ _ _ _| \ _ _ _ _| | _| | _| | _| / _ / \ _ \

version 0.7.1

IGinx> █

```

图 2.1

## 三、SQL 定义与使用

### 3.1 使用 I GinX-SQL

I GinX 支持通过 I GinX-Client 使用 I GinX-SQL 与 I GinX 进行直接交互。I GinX-Client 的运行方式见章节 2.5.3。

### 3.2 数据相关操作

#### 3.2.1 插入数据

##### 3.2.1.1 直接向指定的列插入数据

```
INSERT INTO <prefixPath> ((KEY) (, <suffixPath>)+) VALUES (KeyID (, constant)+);
```

使用插入数据语句可以向指定的一条或有公共前缀的多条时间序列中插入数据。参数具体说明如下：

1. 完整路径 <fullPath> = <prefixPath>.<suffixPath>，完整路径对应的时间序列若没有创建则会自动创建。
2. KEY 为必填关键字，代表插入数据的 ID。
3. KeyID 为该条数据的 ID 值，其默认类型为长整数，常用场景是时间戳，因此，也支持时间戳多种表示方式，本文档以将 KeyID 用作时间戳为例，进行描述：
  - a) long，将会被视为默认的纳秒时间戳，如 1665103338000000000 则会被视为插入时间戳 1665103338000000000 ns
  - b) 带单位的时间戳，如 1665103338s 、 1665103338000ms 、 1665103338000000000ns 等
  - c) now()函数
  - d) yyyy-MM-dd HH:mm:ss 或 yyyy/MM/dd HH:mm:ss
  - e) 基于上述三者的加减 duration 运算表达式，如 now() + 1ms、2021-09-01 12:12:12 - 1ns
  - f) duration 支持 Y|M|D|H|M|S|MS|NS 等单位
4. CONSTANT 为具体的数据。
  - a) 包括 boolean、float、double、int、long、string、空值(NaN、NULL)
  - b) 默认情况下对于整数类型和浮点数类型按照 long 和 double 解析
5. 当路径中的某一级为 4 中提到的 CONSTANT 时，应使用反引号 ` 将其包裹，例如，要创建路径 test.'string'.123.true.null.test123，实际应输入的路径名应为 test.`string`.`123`.`true`.`null`.test123。

以下是使用示例。我们向 <test.test\_insert.status> 和 <test.test\_insert.version> 两条路径分别插入多条数据：

```
I GinX> INSERT INTO test.test_insert (key, status, version) VALUES (1633421949, false, "v2");
I GinX> INSERT INTO test.test_insert (key, status, version) VALUES (1633421950, true, "v3"), (1633421951, false, "v4"), (1633421952, true, "v5");
```

查询我们刚刚插入的结果：

```
IGinX> SELECT * FROM test.test_insert;
```

### 3.2.1.2 通过子查询插入数据

```
INSERT INTO <prefixPath> ((KEY) (, <suffixPath>)+) VALUES (<queryclause>)  
[TIME_OFFSET = constant];
```

IGinX 还支持通过 SQL 查询语句查询到的数据进行插入，并能通过设置 TIME\_OFFSET 来使得插入的数据相对于原数据有一个偏移。

### 3.2.1.3 读取 csv 文件

```
LOAD DATA FROM INFILE <filepath> AS CSV  
[ FIELDS TERMINATED BY ","  
  [OPTIONALLY] ENCLOSED BY ""  
  ESCAPED BY "\\"]  
[ LINES TERMINATED BY "\\n" ]  
[ SKIPPING HEADER ]  
INTO <prefixPath>  
[[((KEY) (, <suffixPath>)+)] | [SET KEY "colName"]]  
[AT <long>]
```

- <filepath>：读取文件的路径，可以自动识别文件的编码格式（无法正确识别编码格式的文件，可能会插入乱码），路径必须以单引号或双引号括住
  - 可以不指定列名，仅指定前缀
  - 可以没有 key 列，IGinX 自动从 0 开始生成 key 列——如果同样的列名，导入到同样的前缀中，则会导致覆盖；想要不覆盖，可以通过给定 at <初始 key 值> 来避免覆盖
- FIELDS
  - TERMINATED BY：字段之间的分隔符，默认为","，还支持使用"\t"；
  - [OPTIONALLY] ENCLOSED BY：将字段用某个字符围起来，若指定了关键字 OPTIONALLY，则只包围非数字类型的字段，默认为""；
  - ESCAPED BY：字段数据存在特殊符号时使用的转移符，默认为"\"；
- LINES TERMINATED BY：每条记录之间的分隔符，默认为"\n"，还支持使用"\r"、"\r\n"等；
- SKIPPING HEADER：是否跳过解析第一行数据，默认不跳过。
- SET KEY "colName"：从 csv 中选出列 colName 复制为 key 列，即 key 列与 colName 相同
- AT <long>：key 值是否加上<long>，<long>不可以为负；若不声明，则 key 值加 0，因此，不指定 key 则从 0 开始

使用例子：

```
IGinX> LOAD DATA FROM INFILE "test.txt" AS CSV SKIPPING HEADER INTO
```



```
tchOpt.jets(key, name, loc, release, duration, due);

IGinX> LOAD DATA FROM INFILE "test.txt" AS CSV INTO tchOpttxt.jets;

IGinX> LOAD DATA FROM INFILE "test.csv" AS CSV INTO tchOpttag.jets
[tagM=jetOpt,tagT=engineX];
IGinX> LOAD DATA FROM INFILE "test.csv" AS CSV SKIPPING HEADER INTO
tchOpt1.jets[tagM=jetOpt,tagT=engineX](key, name, loc, release, duration, due);

IGinX> LOAD DATA FROM INFILE "test" AS CSV SKIPPING HEADER INTO
tchOpt2.jets(key, name [tagM=jetOpt,tagT=engineX], loc, release, duration, due);

IGinX> LOAD DATA FROM INFILE "test.txt" AS CSV FIELDS TERMINATED BY "\\t"
INTO t6 set key "开车时间" at 1000;
```

### 3.2.2 删除数据

```
DELETE FROM path (, path)* (WHERE <orExpression>)?;
```

使用删除语句可以删除指定的时间序列中符合时间删除条件的数据。在删除数据时，用户可以选择需要删除的一个或多个时间序列、时间序列的前缀、时间序列带\*路径对多个时间区间内的数据进行删除。需要注意的是，

1. 删除数据语句不能删除路径，只会删除对应路径的制定数据。
2. 删除语句不支持精确时间点保留，如 `delete from a.b.c where time != 2021-09-01 12:00:00`。
3. 删除语句不支持值过滤条件删除，如 `delete from a.b.c where a.b.c >= 100`。

以下是使用示例。我们可以用下面语句删除全部序列对应的数据：

```
IGinX> DELETE FROM *;
```

我们可以用下面语句删除 `<test.test_delete.*>` 序列在 2021-09-01 12:00:00 到 2021-10-01 12:00:00 对应的数据：

```
IGinX> INSERT INTO test.test_delete (key, status, version) VALUES (2021-09-01 12:22:01,
true, "v1"), (2021-09-01 12:36:03, false, "v2"), (2021-11-01 12:00:00, true, "v3");
IGinX> DELETE FROM test.test_delete WHERE key >= 2021-09-01 12:00:00 AND key <=
2021-10-01 12:00:00;
```

查询删除后的结果：

```
IGinX> SELECT * FROM test.test_delete;
```

### 3.2.3 查询数据

```

SELECT <expression> (, <expression>)* FROM prefixPath <whereClause>?
<downsamplingClause>?

<expression>
: <functionName>(<suffixPath>)
| <suffixPath>

<whereClause>
: WHERE KEY IN <timeInterval> (AND <orExpression>)?
| WHERE <orExpression>;

<orExpression>: <andExpression> (OR <andExpression>)*;

<andExpression>: <predicate> (AND <predicate>)*;

<downsamplingClause>: OVER WINDOW LEFT_BRACKET SIZE <duration> (IN
<timeInterval>)? (SLIDE DURATION)? RIGHT_BRACKET;

<timeInterval>
: (timeValue, timeValue)
| (timeValue, timeValue]
| [timeValue, timeValue)
| [timeValue, timeValue]

```

查询数据语句支持简单范围查询、值过滤查询、聚合查询、降采样查询。

1. 聚合查询函数，目前支持 FIRST\_VALUE、LAST\_VALUE、MIN、MAX、AVG、COUNT、SUM 七种。
2. 降采样查询函数，目前支持 FIRST\_VALUE、LAST\_VALUE、MAX、AVG、COUNT、SUM 六种。
3. timeRange 为一个连续的时间区间，支持左开右闭、左闭右开、左右同开、左右同闭区间。

以下是使用示例。先插入如下数据：

Time	test.test_select.boolean	test.test_select.string	test.test_select.double	test.test_select.long
0	true	fq4RUeRjS8	0.5	1
1	false	QKzYVQBquj	1.5	2
2	true	qdYVJZOYXI	2.5	3
3	false	CicZibVAAc	3.5	4
4	true	c8Dq337a7d	4.5	5
5	false	jjlohugmSi	5.5	6
6	true	inCgynQcwN	6.5	7
.....				
97	false	L5E42AIu4j	97.5	98
98	true	gzQQh2oBeg	98.5	99
99	false	9CR2VrtRrp	93.5	100

### 3.2.3.1 范围查询

查询序列 <test.test\_select.string> 和 <test.test\_select.double> 在 0-100ms 内的值:

```
IGinX> SELECT string, double FROM test.test_select WHERE key>= 0 AND key<= 100;
```

### 3.2.3.2 值过滤查询

查询序列 <test.test\_select.string> 和 <test.test\_select.long> 在 0-100ms 内, 且 <test.test\_select.long> 的值大于 20 的值:

```
IGinX> SELECT string, long FROM test.test_select WHERE key>= 0 AND key<= 100 AND long > 20;
```

#### 3.2.3.2.1 带 UDF 的值过滤查询

IGinX 同时还支持在值过滤查询中使用已经注册过的 UDF

例如, 我们希望查询序列<test.test\_select.string>的正弦值大于 0 的值:

```
IGinX> SELECT string FROM test.test_select WHERE sin(string) > 0;
```

### 3.2.3.3 聚合查询

聚合查询函数, 系统函数目前包括 FIRST、LAST、FIRST\_VALUE、LAST\_VALUE、MIN、MAX、AVG、COUNT、SUM 八种。

查询序列 <test.test\_select.long> 在 0-100ms 内的平均值:

```
IGinX> SELECT AVG(long) FROM test.test_select WHERE key>= 0 AND key<= 100;
```

### 3.2.3.4 降采样查询

降采样查询函数, 系统函数目前包括 FIRST\_VALUE、LAST\_VALUE、MIN、MAX、AVG、COUNT、SUM 六种。

目前, IGINX 支持在某个特定时间范围内进行降采样任务, 实现在滑动窗口上进行聚合运算。IGINX 支持设置降采样查询的时间间隔以及滑动步长, 其中时间间隔和滑动步长必须是正数, 滑动步长默认与时间间隔相等, 例如在 0-100 内:

1. 如果设置 Key 间隔为 10, 不指定滑动步长, 那么滑动步长默认为 10, 会生成 [0, 10), [10, 20), [20, 30)...等分组, 聚集函数在这些分组内分别运行。
2. 如果设置 Key 间隔为 10, 滑动步长为 5, 则会生成[0, 10), [5, 15), [10, 20)...分组, 聚集函数在这些分组内分别运行。
3. 如果设置 Key 间隔为 10, 滑动步长为 20, 则会生成[0, 10), [20, 30), [40, 50)...

分组，聚集函数在这些分组内分别运行。  
查询序列 <test.test\_select.double> 在 0-100 内，每 10 单位的最大值：

```
IGinX> SELECT MAX(double) FROM test.test_select OVER WINDOW (SIZE 10 IN [0, 100]);
```

查询序列 <test.test\_select.double> 在 0-100 内，每 20 的前 10 最大值：

```
IGinX> SELECT MAX(double) FROM test.test_select OVER WINDOW (SIZE 10 IN [0, 100] SLIDE 20);
```

如果不指定 IN <time\_interval>，IGinX 会将数据中最小 key 值作为左边界，最大 key 值作为右边界。例如：

```
IGinX> SELECT MAX(double) FROM test.test_select WHERE key > 300 and key <=500 OVER WINDOW (SIZE 100 SLIDE 50);
```

会划分为[301, 400], [351, 450], [401, 500], [451, 550] 四个窗口。

### 3.2.3.5 数量限制查询

查询序列 <test.test\_select.string> 的前 10 条记录：

```
IGinX> SELECT string FROM test.test_select LIMIT 10;
```

查询序列 <test.test\_select.string> 从第 5 条开始的 10 条记录：

```
IGinX> SELECT string FROM test.test_select LIMIT 10 OFFSET 5;
```

### 3.2.3.6 序列比较查询

序列比较查询允许用户在 value filter 中，添加序列比较条件。

查询 <test.test\_select.int> 在 0-100ms 内，且序列 <test.test\_select.int> 在同一时间戳大于序列 <test.test\_select.long> 的值：

```
IGinX> SELECT int FROM test.test_select WHERE key >= 0 AND key <= 100 AND int > long;
```

#### 3.2.3.6.1 带 UDF 的序列比较查询

IGinX 还支持带 UDF 的序列比较查询

例如，我们希望查询序列<test.a>的正弦值比序列<test.b>的余弦值更大的值：

```
IGinX> SELECT a, b FROM test WHERE sin(a) > cos(b);
```

### 3.2.3.7 模糊查询

模糊查询允许用户对字符值序列进行正则匹配查询。

查询序列 <test.test\_select.string> 以 a 开头的值：

```
IGinX> SELECT string FROM test.test_select WHERE string like "^a.*";
```

查询序列 <test.test\_select.string> 以 s 或者 f 开头的值：

```
IGinX> SELECT string FROM test.test_select WHERE string like "^[sf].*";
```

查询序列 <test.test\_select.string> 以 s 或者 d 结尾的值：

```
IGinX> SELECT string FROM test.test_select WHERE string like ".*[sd]";
```

### 3.2.3.8 序列重命名

IGinX-SQL 允许用户在 SELECT 子句中用 AS 对查询的结果集的列进行重命名。

对序列 <test.test\_select.int> 的降采样查询结果重命名，返回的结果集的序列名为 <max\_int\_per\_ten\_ms>：

```
IGinX> SELECT max(int) AS max_int_per_ten_ms FROM test.test_select OVER WINDOW  
(SIZE 10 IN [0, 100]);
```

IGinX-SQL 允许用户在 FROM 子句中用 AS 对查询序列的前缀进行重命名。

对序列 <test.test\_select> 进行自连接，并将前缀分别重命名为 a 和 b，返回的结果集的序列名为 <a.key>, <a.int>, <b.key>, <b.int> 等等。

```
IGinX> SELECT * FROM test.test_select AS a, test.test_select AS b;
```

AS 子句的执行时机在对应子句执行结束之后，例如下面这条语句，执行完 FROM 子句后，序列 <test.test\_select.int> 被重命名为序列 <a.int>，执行完 SELECT 子句后，序列 <a.int> 被重命名为序列 <result>。

```
IGinX> SELECT int AS result FROM test.test_select AS a WHERE int > 1;
```

IGinX-SQL 还允许用户对子查询的查询结果重命名，即给予查询返回的结果集增加一个前缀。

例如下面这条查询，执行子查询重命名之前，返回结果集的序列名为 <c> 和 <m>，执行子查询重命名后，返回结果集的序列名为 <group.c> 和 <group.m>。

```
IGinX> SELECT * FROM (SELECT char AS c, max(int) AS m FROM test.test_select GROUP BY char) AS group;
```

### 3.2.3.9 嵌套查询

IGinX-SQL 允许用户在 SELECT 子句、FROM 子句、WHERE 子句和 HAVING 子句中使用嵌套查询（暂时只支持非关联子查询）。

#### 3.2.3.9.1 SELECT 子查询

IGinX-SQL 允许用户用子查询代替 SELECT 子句中的表达式或者算术表达式中的某个值，例如：

查询序列<test.a.int>的值和序列<test.b.int>的平均值：

```
IGinX> SELECT int, (SELECT avg(int) FROM test.b) FROM test.a;
```

查询序列<test.a.int>的值与序列<test.b.int>的平均值的乘积：

```
IGinX> SELECT int * (SELECT avg(int) FROM test.b) FROM test.a;
```

在 SELECT 子句中使用子查询时，子查询如果有 WHERE 子句或 HAVING 子句，则 WHERE 子句或 HAVING 子句中的序列名应写出全名，例如：

```
IGinX> SELECT int * (SELECT avg(int) FROM test.b WHERE test.b.int > 1) FROM test.a;
```

在 SELECT 子句中使用子查询时，子查询的查询结果应最多返回一行，否则该查询语句执行时会报错，例如：

```
IGinX> SELECT int * (SELECT avg(int) FROM test.b GROUP BY string) FROM test.a;
```

对于上面的查询语句，该语句运行时是否报错取决于子查询的执行结果。如果序列<test.b.string>的值全部相同，则 GROUP BY 的结果只有一个分组，即子查询只返回一行，语句能正常执行；如果子查询返回多行，该语句则不能正常执行。

#### 3.2.3.9.2 FROM 子查询

IGinX-SQL 允许用户用子查询代替 FROM 子句中的序列名，例如：

查询序列 <test.test\_select.double> 的大于 6 的余弦值：

```
IGinX> SELECT cos_double FROM (SELECT COS(double) AS cos_double FROM test.test_select WHERE double < 6);
```

查询序列 <test.test\_select.double> 的余弦值大于 0 的最小值：

```
IGinX> SELECT MIN(cos_double) FROM (SELECT COS(double) AS cos_double FROM test.test_select) WHERE cos_double > 0;
```

查询序列 <test.test\_select.double> 的大于 6 的余弦值，且余弦值大于 0 的最小值

```
IGinX> SELECT cos_double FROM (SELECT COS(double) AS cos_double FROM test.test_select WHERE double < 6) WHERE cos_double > 0;
```

查询序列 <test.test\_select.int> 的平均值和序列 <test.test\_select.double> 的总和，且均值大于 1，总和小于 15 的值：

```
IGinX> SELECT avg_int, sum_double FROM (SELECT AVG(int) AS avg_int, SUM(double) AS sum_double FROM test.test_select OVER WINDOW (SIZE 2 IN [0, 10])) WHERE avg_int > 1 AND sum_double < 15;
```

查询序列 <test.test\_select.int> 的平均值和序列 <test.test\_select.double> 的总和，且均值大于 1，总和小于 15 的最大值：

```
IGinX> SELECT MAX(avg_int), MAX(sum_double) FROM (SELECT avg_int, sum_double FROM (SELECT AVG(int) AS avg_int, SUM(double) AS sum_double FROM test.test_select OVER WINDOW (SIZE 2 IN [0, 10])) WHERE avg_int > 1 AND sum_double < 15);
```

如果查询语句的 FROM 子句有多个部分，且子查询中的 FROM 子句也有多个部分，则该子查询的 AS 子句不能为空，例如：

```
IGinX> SELECT MIN(cos_double) FROM test.a, (SELECT * FROM test.b INNER JOIN test.c ON test.b.int = test.c.int AS subquery);
```

### 3.2.3.9.3 WHERE 子查询

IGinX-SQL 允许用户在 WHERE 子句中使用 EXIST 子查询、IN 子查询、量化比较子查询(ALL | SOME | ANY)以及用标量子查询进行比较。

在 WHERE 子句中使用子查询时，子查询如果有 WHERE 子句或 HAVING 子句，则 WHERE 子句或 HAVING 子句中的序列名应写出全名。

#### 3.2.3.9.3.1 EXIST 子查询

```
(NOT) EXIST <subquery>
```

如果<subquery>的返回结果不为空，则 EXIST 条件的值是 true；否则，如果<subquery>的返回结果为空，则 EXIST 条件的值是 false。NOT EXIST 条件的结果与 EXIST 相反，例如：

查询序列<test.a.int>在当序列<test.b.int>中存在大于 1 的值时的值：

```
IGinX> SELECT int FROM test.a WHERE EXIST (SELECT * FROM test.b WHERE int > 1);
```

查询序列<test.a.int>在同一时间戳时，序列<test.a.long>的值大于 2 或者序列<test.b.int>中不存在大于 1 的值时的值：

```
IGinX> SELECT int FROM test.a WHERE long > 2 OR NOT EXIST (SELECT * FROM test.b WHERE int > 1);
```

### 3.2.3.9.3.2 IN 子查询

```
<value> | <path> (NOT) IN <subquery>
```

将<subquery>的查询结果集记为 R，如果 R 中存在一行数据 r，使得<value> = r 或 <path> = r，则 IN 条件的结果为 true；如果不存在，则 IN 条件的结果为 false。NOT IN 条件的值与 IN 相反，例如：

查询序列<test.a.int>中所有在序列<test.b.int>中能找到的值：

```
IGinX> SELECT int FROM test.a WHERE int IN (SELECT int FROM test.b);
```

查询序列<test.a.int>中所有在序列<test.b.int>中不能找到的值：

```
IGinX> SELECT int FROM test.a WHERE int NOT IN (SELECT int FROM test.b);
```

IN 子查询的返回结果应最多返回一列。

### 3.2.3.9.3.3 ALL 子查询

```
<value> | <path> <op> ALL <subquery>
```

将<subquery>的查询结果集记为 R，如果对于 R 中的每一行数据 r，都满足<value> <op> r 或<path> <op> r，则 ALL 条件的结果为 true；否则，ALL 条件的结果为 false。例如：

查询序列<test.a.int>中所有大于序列<test.b.int>中的所有值的值：

```
IGinX> SELECT int FROM test.a WHERE int > ALL (SELECT int FROM test.b);
```

### 3.2.3.9.3.4 SOME(ANY)子查询

```
<value> | <path> <op> SOME | ANY <subquery>
```

将<subquery>的查询结果集记为 R，如果 R 中至少存在一行数据 r，满足<value> <op> r 或<path> <op> r，则 SOME(ANY)条件的结果为 true；否则，SOME(ANY)条件的结果为 false。例如：

查询序列<test.a.int>中所有小于序列<test.b.int>中的某个值的值：



```
IGinX> SELECT int FROM test.a WHERE int < SOME (SELECT int FROM test.b);
```

或者

```
IGinX> SELECT int FROM test.a WHERE int < ANY (SELECT int FROM test.b);
```

### 3.2.3.9.3.5 用标量子查询进行比较

IGinX-SQL 允许用户用标量子查询代替 WHERE 子句的值或者序列名，例如：

查询序列<test.a.int>和<test.a.string>，且序列<test.a.int>的值大于序列<test.b.int>的平均值的值：

```
IGinX> SELECT int, string FROM test.a WHERE int > (SELECT avg(int) FROM test.b);
```

### 3.2.3.9.4 HAVING 子查询

IGinX-SQL 允许用户在 HAVING 子句中使用标量子查询进行比较。在 HAVING 子句中使用子查询时，子查询如果有 WHERE 子句或 HAVING 子句，则 WHERE 子句或 HAVING 子句中的序列名应写出全名。WHERE 子句在数据分组前进行过滤，主要用于过滤数据行，而 HAVING 子句在数据分组后进行过滤，主要用于过滤分组。例如：

按序列<test.a.string> 的值分组，并计算每组序列<test.a.int>的平均值，并筛选出平均值大于序列<test.b.int>的平均值的结果：

```
IGinX> SELECT string, avg(int) FROM test.a GROUP BY string HAVING avg(int) > (SELECT avg(int) FROM test.b);
```

### 3.2.3.9.5 INSERT 子查询

IGinX-SQL 允许用户使用子查询来代替 INSERT 语句中的具体数据。

例如，我们需要将序列<us.d1.s1>中大于 1000 且小于 1010 的数据插入序列<us.d2.s1>中：

```
IGinX> INSERT INTO us.d2(key, s1) VALUES (SELECT s1 FROM us.d1 WHERE s1 >= 1000 AND s1 < 1010);
```

INSERT 语句中声明的插入序列数量必须和子查询结果集中的序列数量相等。在 INSERT 语句声明的插入序列和子查询的结果集序列均为多列时，子查询结果集中的序列将被依次插入到申明序列中。

例如，我们需要分别对序列<us.d1.s1>每 10 行求平均数，并将结果插入序列<us.d4.s1>中，对序列<us.d1.s2>每 10 行求和，并将结果插入序列<us.d4.s2>中：

```
IGinX> INSERT INTO us.d4(key, s1, s2) VALUES (SELECT AVG(s1) AS avg_s1, SUM(s2) AS sum_s2 FROM us.d1 OVER WINDOW (SIZE 10 IN [1000, 1100]));
```

### 3.2.3.10 查询序列

```
SHOW COLUMNS;
```

查询序列语句可以查询存储的全部序列名和对应的类型。

### 3.2.3.11 统计数据总量

```
COUNT POINTS;
```

统计数据总量语句用于统计 IGINX 中数据总量。

### 3.2.3.12 清除数据

```
CLEAR DATA;
```

清除数据语句用于删除 IGINX 中全部数据和路径。

以下是应用示例。清除数据，并查询清除之后的数据：

```
IGINX> CLEAR DATA;  
IGINX> SELECT * FROM *;
```

### 3.2.3.13 Explain 查询

```
EXPLAIN <queryClause>;
```

Explain 查询语句用于查询 IGINX 中生成的逻辑查询计划

以下是应用示例。

```
IGINX> EXPLAIN SELECT MAX(s2), MIN(s1) FROM us.d1
```

```
IGINX> explain select max(s2),min(s1) from us.d1  
ResultSets:  
+-----+-----+-----+-----+  
| Logical Tree|Operator Type|Operator Info|  
+-----+-----+-----+-----+  
|Reorder      |Reorder      |Order: max(us.d1.s2),min(us.d1.s1)|  
|  +--Join    |Join         |JoinBy: ordinal|  
|  +--SetTransform|SetTransform|Func: {Name: min, FuncType: System, MappingType: SetMapping}|  
|  +--Project  |Project      |Patterns: us.d1.s1,us.d1.s2, Target DU: unit0000000000|  
|  +--SetTransform|SetTransform|Func: {Name: max, FuncType: System, MappingType: SetMapping}|  
|  +--Project  |Project      |Patterns: us.d1.s1,us.d1.s2, Target DU: unit0000000000|  
+-----+-----+-----+-----+  
Total line number = 6
```

### 3.2.3.14 分组查询

注意：分组查询中选取的（未被聚集函数修饰的）列必须出现在 group by 子句中

以下是应用示例。

数据准备

```
IGINX> insert into test(key, a, b, c, d) values (1, 3, 2, 3.1, \"val1\"), (2, 1, 3, 2.1, \"val2\"), (3,  
2, 2, 1.1, \"val5\"), (4, 3, 2, 2.1, \"val2\"), (5, 1, 2, 3.1, \"val1\"), (6, 2, 2, 5.1, \"val3\");
```

```
IGinX> insert into test(key, a, b, c, d) values (1, 3, 2, 3.1, "val1"), (2, 1, 3, 2.1, "val2"), (3, 2, 2, 1.1, "val5"), (4, 3, 2, 2.1, "val2"), (5, 1, 2, 3.1, "val1"), (6, 2, 2, 5.1, "val3");
success
IGinX> select * from test;
ResultSet:
+-----+-----+-----+-----+
|key|test.a|test.b|test.c|test.d|
+-----+-----+-----+-----+
| 1|    3|    2|   3.1|  val1|
| 2|    1|    3|   2.1|  val2|
| 3|    2|    2|   1.1|  val5|
| 4|    3|    2|   2.1|  val2|
| 5|    1|    2|   3.1|  val1|
| 6|    2|    2|   5.1|  val3|
+-----+-----+-----+-----+
Total line number = 6
IGinX>
```

按序列 test.b 的值分组，并查询 test.a 序列每组的平均值

```
IGinX> select avg(a), b from test group by b;
```

```
IGinX> select avg(a), b from test group by b;
ResultSet:
+-----+-----+
|avg(test.a)|test.b|
+-----+-----+
|          2.2|    2|
|          1.0|    3|
+-----+-----+
Total line number = 2
```

按照序列 test.b 和 test.d 的值进行分组，并求 test.a 序列每组的平均值，并筛选出均值大于 2 的结果

```
IGinX> select avg(a), b, d from test group by b, d having avg(a) > 2;
IGinX> select avg(a), b, d from test group by b, d having avg(a) > 2;
ResultSet:
+-----+-----+-----+
|avg(test.a)|test.b|test.d|
+-----+-----+-----+
|          3.0|    2|  val2|
+-----+-----+-----+
Total line number = 1
```

按照序列 test.b 、test.c 和 test.d 的值进行分组，并求 test.a 序列每组的平均值，并将结果按照 test.c 的值进行排序

```
IGinX> select avg(a), c, b, d from test group by c, b, d order by c;
```

```
IGinX> select avg(a), c, b, d from test group by c, b, d order by c
ResultSets:
+-----+-----+-----+-----+
|avg(test.a)|test.c|test.b|test.d|
+-----+-----+-----+-----+
|          2.0|    1.1|    2| val5|
|          3.0|    2.1|    2| val2|
|          1.0|    2.1|    3| val2|
|          2.0|    3.1|    2| val1|
|          2.0|    5.1|    2| val3|
+-----+-----+-----+-----+
Total line number = 5
```

### 3.2.3.15 查询排序

查询排序按照指定序列的值排序，支持多值和升降序（默认为按值升序排列）

数据准备

```
IGinX> INSERT INTO us.d2 (key, s1, s2, s3) values (1, \"apple\", 871, 232.1), (2, \"peach\",
123, 132.5), (3, \"banana\", 356, 317.8), (4, \"cherry\", 621, 456.1), (5, \"grape\", 336, 132.5),
(6, \"dates\", 119, 232.1), (7, \"melon\", 516, 113.6), (8, \"mango\", 458, 232.1), (9, \"pear\",
336, 613.1);
```

```
IGinX> SELECT * FROM us.d2
ResultSet:
+---+-----+-----+-----+
|key|us.d2.s1|us.d2.s2|us.d2.s3|
+---+-----+-----+-----+
| 1|apple|871|232.1|
| 2|peach|123|132.5|
| 3|banana|356|317.8|
| 4|cherry|621|456.1|
| 5|grape|336|132.5|
| 6|dates|119|232.1|
| 7|melon|516|113.6|
| 8|mango|458|232.1|
| 9|pear|336|613.1|
+---+-----+-----+-----+
Total line number = 9
IGinX> 
```

查询匹配 `us.d2.*` 的序列，并将结果按 `us.d2.s3` 排序，如果 `us.d2.s3` 的值相等，则按 `us.s2.s2` 排序

```
IGinX> SELECT * FROM us.d2 ORDER BY s3, s2;
```

```
IGinX> SELECT * FROM us.d2 ORDER BY s3, s2  
ResultSets:
```

key	us.d2.s1	us.d2.s2	us.d2.s3
7	melon	516	113.6
2	peach	123	132.5
5	grape	336	132.5
6	dates	119	232.1
8	mango	458	232.1
1	apple	871	232.1
3	banana	356	317.8
4	cherry	621	456.1
9	pear	336	613.1

```
Total line number = 9
```

查询匹配 us.d2.\* 的序列，并将结果按 us.d2.s1 降序排列

```
IGinX> SELECT * FROM us.d2 ORDER BY s1 DESC;
```

```
IGinX> SELECT * FROM us.d2 ORDER BY s1 DESC
ResultSets:
```

key	us.d2.s1	us.d2.s2	us.d2.s3
9	pear	336	613.1
2	peach	123	132.5
7	melon	516	113.6
8	mango	458	232.1
5	grape	336	132.5
6	dates	119	232.1
4	cherry	621	456.1
3	banana	356	317.8
1	apple	871	232.1

Total line number = 9

```
IGinX>
```

### 3.2.3.16 集合操作

IGinX-SQL 允许用户使用 UNION(并)、EXCEPT(差)和 INTERSECT(交)三种集合操作符。使用集合操作符时，需要注意以下几点，

1. 两条查询语句应同时返回 key 或不返回 key。
2. 两条查询语句返回的列的数量应该相等。
3. 两条查询语句对应的列应该可比较。
4. 最后返回的列名是第一条查询语句的列名。
5. 默认会对结果集去重，使用 ALL 不进行去重。

查询序列 <a.int>、<a.boolean>、<a.string> 以及序列 <b.double>、<b.boolean>、<b.string> 的并集，保留所有结果不去重。

```
IGinX> SELECT int, boolean, string FROM a UNION ALL SELECT double, boolean, string FROM b;
```

查询序列 <a.int>、<a.boolean>、<a.string> 以及序列 <b.double>、<b.boolean>、<b.string> 的交集，并去重。

```
IGinX> SELECT int, boolean, string FROM a INTERSECT SELECT double, boolean, string FROM b;
```

此外，若要对左右查询使用 ORDER BY 或 LIMIT 子句，需要在查询语句外使用括号，例如：

```
IGinX> (SELECT int, boolean, string FROM a ORDER BY int LIMIT 100) UNION ALL  
SELECT double, boolean, string FROM b;
```

或

```
IGinX> SELECT int, boolean, string FROM a UNION ALL (SELECT double, boolean,  
string FROM b ORDER BY int LIMIT 100);
```

```
IGinX> SELECT int, boolean, string FROM a UNION ALL SELECT double, boolean,  
string FROM b ORDER BY int LIMIT 100;
```

等价于

```
IGinX> (SELECT int, boolean, string FROM a UNION ALL SELECT double, boolean,  
string FROM b) ORDER BY int LIMIT 100;
```

### 3.2.3.17 DISTINCT 关键字

IGinX-SQL 允许用户在 SELECT 子句的表达式中使用 DISTINCT，例如：  
查询序列<test.a.int>中出现过的值的集合。

```
IGinX> SELECT DISTINCT int FROM test.a;
```

需要注意的是，使用 DISTINCT 后，原来的 key 列将会消失，不会再返回。

IGinX-SQL 还允许用户在一些聚合函数中使用 DISTINCT，支持使用 DISTINCT 的聚合函数有 COUNT，AVG，SUM，MIN，MAX。

查询序列<test.a.int>中出现过的值的种类，以及这些不重复值的平均值。

```
IGinX> SELECT COUNT(DISTINCT int), AVG(DISTINCT int) FROM test.a;
```

### 3.2.3.18 SQL 查询结果导出到文件

#### 3.2.3.18.1 导出为 csv

```
<queryclause>  
INTO OUTFILE <filepath> AS CSV  
[ FIELDS TERMINATED BY ","  
  [OPTIONALLY] ENCLOSED BY ""  
  ESCAPED BY "\\"]  
[ LINES TERMINATED BY "\\n" ]  
[ WITH HEADER ]
```

- <queryclause>: 查询语句
- <filepath>: 保存文件路径，需以.csv 结尾
- FIELDS



- TERMINATED BY: 字段之间的分隔符，默认为","，还支持使用"\t"；
- [OPTIONALLY] ENCLOSED BY: 将字段用某个字符围起来，若指定了关键字 OPTIONALLY，则只包围非数字类型的字段，默认为""；
- ESCAPED BY: 字段数据存在特殊符号时使用的转移符，默认为"\\"；
- LINES TERMINATED BY: 每条记录之间的分隔符，默认为"\n"，还支持使用"\r"、"\r\n"等；
- WITH HEADER: 是否输出表头，默认不输出表头。

### 3.2.3.18.1 以字节流形式导出

```
<query> INTO OUTFILE <dirpath> AS STREAM
```

- <queryclause>: 查询语句
- <dirpath>: 导出文件所在的目录，查询结果的每一列会分别保存到该目录下的一个文件中，文件名为列名。若有重复列名，依次在文件名末尾加上(1), (2).....

### 3.2.3.19 公用表表达式（WITH 子句）

IGinX-SQL 允许用户在查询语句中使用 CTE，例如：

```
WITH a(c1, c2) AS (SELECT column1, column2 FROM test) SELECT * FROM a;
```

多个 CTE 之间用逗号连接，例如：

```
WITH a(c1, c2) AS (SELECT column1, column2 FROM test WHERE key < 100),  
b(c1, c2, c3) AS (SELECT column1, column2, column3 FROM test WHERE key > 100)  
SELECT * FROM a, b;
```

写在后面的 CTE 可以引用前面的 CTE，例如：

```
WITH a(c1, c2) AS (SELECT * FROM test),  
b AS (SELECT c1 FROM a GROUP BY c1)  
SELECT * FROM a, b;
```

写在前面的 CTE 无法引用后面的 CTE，例如：

```
WITH a(c1, c2) AS (SELECT * FROM b),  
b AS (SELECT c1 FROM a GROUP BY c1)  
SELECT * FROM a, b;
```

这时，第一条 CTE 中的 FROM b 并不是后面的 CTE，而是会从底层数据库中取出路径 b.\* 中的数据。

### 3.2.3.20 VALUE2META

当 SELECT 子句中要选取的路径未知时，IGinX-SQL 允许用户在查询语句中使用

VALUE2META 来解决该问题，例如：

```
SELECT VALUE2META(SELECT suffix FROM prefix) FROM test;
```

若语句 `SELECT suffix FROM prefix` 返回的结果为单列，且该列内容有三个元素，依次为 `a.a`, `a.b`, `b.a`, 则上述语句等价于

```
SELECT a.a, a.b, b.a FROM test;
```

当 VALUE2META 中的语句返回多列时，会“逐行逐列”，即“按行序，行内按列序”地读取其中的数据，转为字符串，遇到空值时跳过。

### 3.2.3.21 关联（Join）查询

IGinX 支持通过 join 操作连接数据并返回符合连接条件和查询条件的数据。下文为您介绍左连接、右连接、全连接、内连接、自然连接、隐式连接和多路连接的使用方法。

#### 3.2.3.21.1 IGINX 内关联（Join）概念的说明

由于 IGINX 使用的序列模型可能区别于一般关系数据库中使用的关系模型，也没有对应的“表”这一概念，因此特在此做一些说明。

假设 IGINX 系统内现有六条序列

1. test1.a
2. test1.b
3. test1.c.cc
4. test2.a
5. test2.b
6. test2.c.cc

IGinX 在关联查询的过程中，会将所有满足指定前缀，且后缀只有一级路径的序列认定为同一张表，并基于此进行关联查询。举例说明

对于查询

```
SELECT * FROM test1 JOIN test2 on test1.a = test2.a;
```

来说，IGinX 会认为 `test1.a`、`test2.b` 按 key 对齐的数据为表 1，`test2.a`、`test2.b` 按 key 对齐的数据为表 2，表 1 和表 2 按照关联条件 `test1.a == test2.a` 计算关联查询结果（注意：`test1.c.cc` 和 `test2.c.cc` 两个序列既不属于表 1 也不属于表 2，因为其后缀有两级路径）

#### 3.2.3.21.2 支持的关联类型

IGinX 支持如下关联操作

#### 3.2.3.21.2.1 左连接（left outer join）

可简写为 left join。返回左表中的所有记录，即使右表中没有与之匹配的记录。注意：Join 条件要写全序列名，例如

```
IGinX> select * from test1 left join test2 on test1.a = test2.a;  
IGinX> select * from test1 left outer join test2 on test1.a = test2.a;
```

#### 3.2.3.21.2.2 右连接（right outer join）

可简写为 right join。返回右表中的所有记录，即使左表中没有与之匹配的记录。注意：Join 条件要写全序列名，例如

```
IGinX> select * from test1 right join test2 on test1.a = test2.a;  
IGinX> select * from test1 right outer join test2 on test1.a = test2.a;
```

#### 3.2.3.21.2.3 全连接（full outer join）

可简写为 full join。返回左右表中的所有记录。注意：Join 条件要写全序列名，例如

```
IGinX> select * from test1 full join test2 on test1.a = test2.a;  
IGinX> select * from test1 full outer join test2 on test1.a = test2.a;
```

#### 3.2.3.21.2.4 内连接（inner join）

关键字 inner 可以省略。左右表中至少存在一个匹配行时，inner join 返回数据行。注意：Join 条件要写全序列名，例如

```
IGinX> select * from test1 join test2 on test1.a = test2.a;  
IGinX> select * from test1 inner join test2 on test1.a = test2.a;
```

#### 3.2.3.21.2.5 自然连接（natural join）

参与 join 的两张表根据字段名称自动决定连接字段。支持 outer natural join，支持使用 using 子句执行 join，输出字段中公共字段只出现一次。注意：使用 using 字句时，应提供关联查询两边的公共后缀，例如 test1.a 和 test2.a 的公共后缀 a，示例如下

```
IGinX> select * from test1 natural join test2;
IGinX> select * from test1 natural join test2 using a;
IGinX> select * from test1 natural left join test2 using a;
IGinX> select * from test1 natural right outer join test2 using a;
```

### 3.2.3.21.2.6 笛卡尔积（cross join）

cross join 可省略，写成逗号分隔的形式。注意：

1. where 条件中涉及到的序列需要写全列名。
2. IGINX 优化器目前还不支持满足条件的 cross join 自动改写为其他 join 的形式，因此执行时间相比于直接写成其他 join 的形式会慢一些。

```
IGinX> select * from test1 cross join test2;
IGinX> select * from test1, test2;
IGinX> select * from test1, test2 where test1.a = test2.a and test2.a > 10;
```

### 3.2.3.21.3 在关联查询中使用嵌套查询

IGinX 还支持某一部分 joinpart 为嵌套查询语句。注意：Join 条件要写全序列名，例如

```
IGinX> select * from test1 join (
    select a, b from test2
) on test1.a = test2.a;
```

## 3.3 系统相关操作

### 3.3.1 增加存储引擎

```
ADD STORAGEENGINE (ip, port, engineType, extra)+;
```

1. 添加引擎之前先保证示例存在且正确运行。
2. 为了方便拓展，extra 目前是 string 类型，具体为一个 string 类型的 map。

以下是应用示例。(1)添加一个 Parquet 实例—127.0.0.1:6667 (2) 添加一个文件系统实例——127.0.0.1:6668，数据写出目录为 hello，并将当前在 python\_scripts 目录下的文件融入系统（因而设置 has\_data 为 true），同时设置 2 个相关参数

```
ADD STORAGEENGINE ("127.0.0.1", 6667, "parquet", "dir:/usr/home/data/parquetData,
iginx_port=6888");
add storageengine ("127.0.0.1", 6668, "filesystem", "dir:hello, has_data:true,
```

```
is_read_only:false, dummy_dir:python_scripts, chunk_size_mb:1, memory_pool_size:100")
```

如果需要查询数据库中原有数据，需在额外参数中配置 `has_data` 参数（以 Parquet 为例，其他数据库格式相同）：

```
# 读取/usr/home/parquetFiles 中的 parquet 文件，向/usr/home/data/parquetData 中写入数据
ADD STORAGEENGINE ("127.0.0.1", 6667, "parquet", "dir:/usr/home/data/parquetData,
iginx_port=6888, dummy_dir = /usr/home/parquetFiles, has_data=true");
```

如果该数据库仅读，需要配置 `is_read_only` 参数：

```
# 读取/usr/home/parquetFiles 中的 parquet 文件，不写入
ADD STORAGEENGINE ("127.0.0.1", 6667, "parquet", "iginx_port=6888, dummy_dir =
/usr/home/parquetFiles, has_data=true, is_read_only=true");
```

`has_data` 和 `is_read_only` 参数可在所有数据库类型的注册语句中自由结合使用，但 `has_data=false, is_read_only=true` 的数据库没有意义，视作非法。在下面分别介绍的数据库注册样例中，不再介绍这两个参数。

目前，IGinX 支持的底层存储引擎包括：

1. IoTDB12
2. InfluxDB
3. Parquet
4. MongoDB
5. Redis
6. 提供 jdbc 接口的关系型数据库
  - (1) MySql
  - (2) PostgreSQL
  - (3) ...
7. FileSystem

各数据库需要配置的连接参数如下：

## 1. IoTDB

```
ADD STORAGEENGINE ("<ip>", <port>, "iotdb12", "username:<username>,
password:<password>, sessionPoolSize:<sessionPoolSize>");

# example:
ADD STORAGEENGINE ("127.0.0.1", 6667, "iotdb12", "username:root, password:root,
sessionPoolSize:130");
```

## 2. Influxdb

```
ADD STORAGEENGINE ("<ip>", <port>, "influxdb", "url:http://localhost:<port>/,
username:<username>, password:<password>, token:<influxdbToken>,
organization:<influxdbOrg>");
```

# example:

```
ADD STORAGEENGINE ("127.0.0.1", 8086, "influxdb", "url:http://localhost:8086/,
username:user, password:12345678, token:testToken, organization:testOrg");
```

### 3. Parquet

在 IGINX 系统中，Parquet 数据文件夹可以被视为底层数据库进行连接和查询。用户需要在 Parquet 数据文件夹本地的 IGINX 服务节点中，注册 Parquet 数据库。

因此，用户在注册 Parquet 数据库时，需要提供数据文件路径 `dir` 和 `iginx_port`，即注册时连接的 IGINX 节点端口，`port` 需要是未被占用的端口。`iginx_port` 与 `ip` 结合，能唯一标识某个 IGINX 服务节点。`ip` 是本机 `ip`。`dir` 路径是 IGINX 写入的数据位置，需要是空文件夹。

Windows 中，文件分隔符需要写为\\，具体可参照示例。

语句如下：

```
ADD STORAGEENGINE ("<ip>", <port>, "parquet", "dir:<data_dir>,
iginx_port:<iginx_port>");
```

如果需要读取已有的 parquet 文件，需要设置 `dummy_dir` 参数为文件夹路径，并设置 `has_data=true`。`dummy_dir` 不可为空。

```
ADD STORAGEENGINE ("<ip>", <port>, "parquet", "dir:<data_dir>,
iginx_port:<iginx_port>, dummy_dir=<dummy_dir>, has_data=true");
```

如果不需要写入 parquet 文件，只需要读取已有文件，可以不设置 `dir` 参数，同时需要设置 `is_read_only=true`。

```
ADD STORAGEENGINE ("<ip>", <port>, "parquet", "iginx_port:<iginx_port>,
dummy_dir=<dummy_dir>, has_data=true, is_read_only=true");
```

用例：

```
# 本机 IGINX 监听 6888 端口，写入数据时存至/usr/home/data/parquetData
ADD STORAGEENGINE ("127.0.0.1", 6667, "parquet", "dir:/usr/home/data/parquetData,
iginx_port=6888");

# 读取 E:/data/parquetFiles 中的 parquet 文件，向 E:/data/parquetData 中写入数据
ADD STORAGEENGINE ("127.0.0.1", 6667, "parquet", "dir:E:\\data\\parquetFiles,
```

```
iginx_port=6888, dummy_dir = E:\\data\\parquetData, has_data=true");  
  
# 读取/usr/home/parquetFiles 中的 parquet 文件，不写入  
ADD STORAGEENGINE ("127.0.0.1", 6667, "parquet", "iginx_port=6888, dummy_dir =  
/usr/home/parquetFiles, has_data=true, is_read_only=true");
```

更多详细参数见 11.3.1

#### 4. MongoDB

```
ADD STORAGEENGINE ("127.0.0.1", 27017, "mongodb", "");
```

更多详细参数见 11.3.3

#### 5. Redis

```
ADD STORAGEENGINE ("127.0.0.1", 6379, "redis", "");
```

更多详细参数见 11.3.4

#### 6. 关系型数据库

IGinX 可以通过 jdbc 连接传统关系数据库，包括 MySQL、PostgreSQL 等。

##### (1). PostgreSQL

```
ADD STORAGEENGINE ("<ip>", <port>, "relational", "engine:postgresql,  
username:<postgres_username>, password:<postgres_password>");  
  
# example:  
ADD STORAGEENGINE ("127.0.0.1", 5432, "relational", "engine:postgresql,  
username:postgres, password:postgres");
```

##### (2) . MySQL

```
ADD STORAGEENGINE ("<ip>", <port>, "relational", "engine:mysql,  
username:<mysql_username>, meta_properties_path:<mysql_props_path>");  
  
# example:  
ADD STORAGEENGINE ("127.0.0.1", 3306, "relational", "engine:mysql, username:root,  
meta_properties_path:resources/mysql-meta-template.properties");
```

更多详细参数见 11.3.5

## 7. 本地文件系统 filesystem

本地文件夹可以被视为 IGINX 的底层数据库，它与 Parquet 相似，都通过本地 IGINX 节点进行注册。

用户在注册 filesystem 数据库时，同样需要提供数据文件路径 `dir` 和 `iginx_port`，即注册时连接的 IGINX 节点端口。`port` 需要是未被其他程序使用的端口。`iginx_port` 与 `ip` 结合，能唯一标识某个 IGINX 服务节点。`dir` 路径是 IGINX 写入的数据位置，需要是空文件夹。

Windows 中，文件分隔符需要写为\\，具体可参照示例。

语句如下：

```
ADD STORAGEENGINE ("<ip>", <port>, "filesystem", "dir:<data_dir>,  
iginx_port:<iginx_port>");
```

如果需要管理已有的本地文件，需要设置 `dummy_dir` 参数为文件夹路径，并设置 `has_data=true`。IGINX 对已有的本地文件作为字节流处理。

```
ADD STORAGEENGINE ("<ip>", <port>, "filesystem", "dir:<data_dir>,  
iginx_port:<iginx_port>, dummy_dir=<dummy_dir>, has_data=true");
```

如果不需要写入数据，只需要管理已有文件，可以不设置 `dir` 参数，同时需要设置 `is_read_only=true`。

```
ADD STORAGEENGINE ("<ip>", <port>, "filesystem", "iginx_port:<iginx_port>,  
dummy_dir=<dummy_dir>, has_data=true, is_read_only=true");
```

用例：

```
# 本机 IGINX 监听 6888 端口，写入数据时存至/usr/home/data/fsData  
ADD STORAGEENGINE ("127.0.0.1", 6667, "filesystem", "dir:/usr/home/data/fsData,  
iginx_port=6888");  
  
# 读取/usr/home/files 中的文件，向/usr/home/data/fsData 中写入数据  
ADD STORAGEENGINE ("127.0.0.1", 6667, "filesystem", "dir:/usr/home/data/fsData,  
iginx_port=6888, dummy_dir = /usr/home/files, has_data=true");  
  
# 读取 E:/data/files 中的文件，向 E:/data/fsData 中写入数据  
ADD STORAGEENGINE ("127.0.0.1", 6667, "filesystem", "dir:E:\\data\\files,  
iginx_port=6888, dummy_dir = E:\\data\\fsData, has_data=true");  
  
# 读取/usr/home/files 中的文件，不写入  
ADD STORAGEENGINE ("127.0.0.1", 6667, "filesystem", "iginx_port=6888, dummy_dir =  
/usr/home/files, has_data=true, is_read_only=true");
```



在 Parquet 和 Filesystem 中，均不允许将根目录作为数据文件地址。

### 3.3.2 查询副本数量

```
SHOW REPLICA NUMBER;
```

查询副本数量语句用于查询现在 IGinX 存储的副本数量。

### 3.3.3 查询集群信息

```
SHOW CLUSTER INFO;
```

查询集群信息语句用于查询 IGinX 集群信息，包括 IGinX 节点、存储引擎节点、元数据节点信息等。应用示例如图 3.1 所示。

```
IginX> show cluster info
IginX infos:
+-----+
| ID |      IP | PORT |
+-----+
| 0 | 0.0.0.0 | 6888 |
+-----+
Storage engine infos:
+-----+
| ID |      IP | PORT | TYPE |
+-----+
| 0 | 127.0.0.1 | 6667 | iotdb11 |
| 1 | 127.0.0.1 | 6668 | iotdb12 |
+-----+
Meta Storage infos:
+-----+
|      IP | PORT | TYPE |
+-----+
| 127.0.0.1 | 2181 | zookeeper |
+-----+
IginX>
```

图 3.1

### 3.3.4 用户权限管理

```
CREATE USER <username> IDENTIFIED BY <password>;
GRANT <permissionSpec> TO USER <username>;
```

```
SET PASSWORD FOR <username> = PASSWORD(<password>);  
DROP USER <username>  
SHOW USER (<username>)*  
  
<permissionSpec>: (<permission>,+  
  
<permission>: READ | WRITE | ADMIN | CLUSTER
```

用户权限管理语句用于 IGINX 新增用户，更新用户密码、权限，删除用户等。  
以下是操作示例。添加一个无任何权限的用户 root1，密码为 root1：

```
CREATE USER root1 IDENTIFIED BY root1;
```

授予用户 root1 以读写权限：

```
GRANT WRITE, READ TO USER root1;
```

将用户 root1 的权限变为只读：

```
GRANT READ TO USER root1;
```

将用户 root1 的密码改为 root2：

```
SET PASSWORD FOR root1 = PASSWORD(root2);
```

删除用户 root1：

```
DROP USER root1
```

展示用户 root2, root3 信息

```
SHOW USER root2, root3;
```

展示所有用户信息

```
SHOW USER;
```

### 3.3.5 移除堆叠分片

```
REMOVE HISTORYDATASOURCE removedStorageEngine (,removedStorageEngine)*  
  
removedStorageEngine : (ip=stringLiteral, port=INT, schemaPrefix=stringLiteral,  
dataPrefix=stringLiteral )
```

在 has\_data=true 的情况下，添加了一个节点的部分或全部数据（带 prefix 或不带 prefix），允许用户删除单次添加形成的单堆叠分片，或多次添加形成的多个堆叠分片。通过四元组（host,port,schemaPrefix,dataPrefix）唯一确定一个堆叠分片，移除该堆叠分片。

以下是操作示例。移除在本地，6668 端口，schemaPrefix 为 p1，dataPrefix 为 test 的所有历史数据源。

```
REMOVE HISTORYDATASOURCE ("127.0.0.1", 6668, "p1", "test");
```

// 若 schemaPrefix 或者 dataPrefix 为空，使用空字符串：

```
REMOVE HISTORYDATASOURCE ("127.0.0.1", 6668, "", "");
```

### 3.3.6 配置管理

```
SET CONFIG configName configValue;
```

```
SHOW CONFIG configName;
```

该功能可以用来调整 IGINX 配置参数和查询当前的配置参数，注意：某些和初始化相关的配置参数通过命令动态调整可能不生效

使用示例

将当前配置项 enableInstantCompaction 设置为 true

```
SET CONFIG "enableInstantCompaction" "true";
```

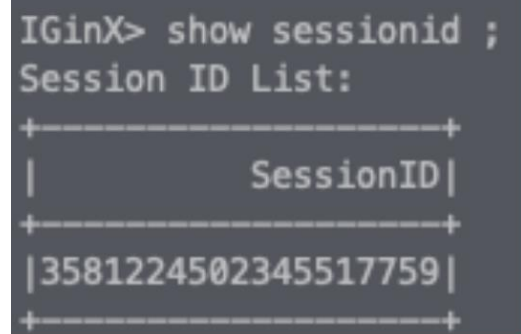
查询当前配置项 enableInstantCompaction 的值

```
SHOW CONFIG "enableInstantCompaction";
```

### 3.3.7 查询当前的 session id

```
SHOW SESSIONID;
```

该语句用于查询当前的连接当前 IGINX 的 session id 列表，例如



```
IGinx> show sessionid ;
Session ID List:
+-----+
| SessionID |
+-----+
| 3581224502345517759 |
+-----+
```

### 3.3.8 优化规则管理

```
SET RULES ruleName1=on, ruleName2=off;
```

```
SHOW RULES;
```

IGinX 会按照配置项中的 `ruleBasedOptimizer` 初始化优化器规则的开关情况，配置中未出现的规则默认为关。

该功能可以动态开关查询优化器的优化规则，以及查看当前所有优化规则的开关情况。

使用示例

将当前的优化规则 `FilterFragmentRule` 关闭

```
SET RULES FilterFragmentRule=off;
```

查询当前所有优化规则的开关情况。

```
SHOW RULES;
```

展示结果

## 四、基于 SQL 支持 TagKV 定义

在 0.5.0 版本的 I GinX 中，已经新增了对 TagKV 语法的支持。

### 4.1 概述

TagKV 指的是你在对向某一个时间序列中写入数据的时候，可以为每一个数据点提供一组字符串键值对，用于存储相关的信息，这一组键值对就被称之为 TagKV。

在进行查询的时候，也可以在提供部分/全部 TagKV 的基础上，对查询结果进行过滤。

### 4.2 语法

#### 4.2.1 插入数据

```
INSERT INTO <prefixPath> ([<key=value>(, <key=value>)+])? ((KEY) (, <suffixPath>([<key=value>(, <key=value>)+])?)?) VALUES (KeyID (, constant)+);
```

#### 注意：

(1) 如果 prefix path 之后直接跟了 tagkv，那么后面的 suffix path 中就不能再出现，并且 prefix path 中的 tagkv 被所有的 suffix path 共享。

(2) 可以单独为 suffix path 中的某些提供各自的 tagkv。

(3) tagkv 是可选的，也可以不提供 tagkv 信息。

以下是使用示例。

```
insert into ln.wf02 (key, s, v) values (100, true, "v1"); # 不含有 tagkv
insert into ln.wf02[t1=v1] (key, s, v) values (400, false, "v4"); # 提供一个 tagkv，并且被多个序列所共享
insert into ln.wf02[t1=v1,t2=v2] (key, v) values (800, "v8"); # 提供多个 tagkv，并且被多个序列所共享
insert into ln.wf03 (key, s[t1=vv1,t2=v2], v[t1=vv11]) values (1600, true, "v16"); # 针对不同的序列提供不同的 tagkv
```

#### 4.2.2 查询数据

```
SELECT <expression> (, <expression>)* FROM prefixPath <whereClause>? <withClause>? <downsamplingClause>?
```

```
<expression>
: <functionName>(<suffixPath>)
| <suffixPath>
```

```
<whereClause>
```

```

: WHERE KEY IN <timeInterval> (AND <orExpression>)?
| WHERE <orExpression>;

<orExpression>: <andExpression> (OR <andExpression>)*;

<andExpression>: <predicate> (AND <predicate>)*;

withClause
: WITH orTagExpression
;

orTagExpression
: andTagExpression (OPERATOR_OR andTagExpression)*
;

andTagExpression
: tagExpression (OPERATOR_AND tagExpression)*
;

tagExpression
: tagKey OPERATOR_EQ tagValue
| LR_BRACKET orTagExpression RR_BRACKET
;

<downsamplingClause>: OVER WINDOW LEFT_BRACKET SIZE DURATION IN
<timeInterval> (SLIDE DURATION)? RIGHT_BRACKET;

<timeInterval>
: (timeValue, timeValue)
| (timeValue, timeValue]
| [timeValue, timeValue)
| [timeValue, timeValue]

```

以下是使用示例。查询 `data_center.memory` 序列过去一小时的数据，并且要求包含有 `tagk = rack & tagv = A` 以及 `tagk = room & tagv = ROOMA` 这两对组合。

```

select memory from data_center where key >= now() - 1h and key < now() with rack=A and
room=ROOMA;

```

## 4.3 例子

首先利用如下的语句向集群中写入若干数据点：

```

insert into ln.wf02 (key, s, v) values (100, true, "v1");
insert into ln.wf02[t1=v1] (key, s, v) values (400, false, "v4");
insert into ln.wf02[t1=v1,t2=v2] (key, v) values (800, "v8");
insert into ln.wf03 (key, s[t1=vv1,t2=v2], v[t1=vv1]) values (1600, true, "v16");

```

```
insert into ln.wf03 (key, s[t1=v1,t2=vv2], v[t1=v1]) values (3200, true, "v32");
```

可以利用前缀语法查询系统中以 `ln` 开头的数据，所有的序列以及其对应的所有的 `TagKV` 的组合都会被查询出来，结果如图 4.1 所示。

```
select * from ln;
```

```
IginX> select * from ln;
ResultSets:
```

	Time	ln.wf02.s	ln.wf02.s{t1=v1}	ln.wf02.v	ln.wf02.v{t1=v1,t2=vv2}	ln.wf02.v{t1=v1}	ln.wf03.s{t1=v1,t2=vv2}	ln.wf03.s{t1=vv1,t2=v2}	ln.wf03.v{t1=v1}	ln.wf03.v{t1=vv1}
	1970-01-01T08:00:00.100	true	null	v1	null	null	null	null	null	null
	1970-01-01T08:00:00.400	null	false	null	null	v4	null	null	null	null
	1970-01-01T08:00:00.800	null	null	null	v8	null	null	null	null	null
	1970-01-01T08:00:01.600	null	null	null	null	null	null	true	null	v16
	1970-01-01T08:00:03.200	null	null	null	null	null	true	null	v32	null

Total line number = 5

图 4.1

随后不指定 `tagKV`，对 `ln.*.s` 模式的数据进行查询，可以看到所有的符合条件的蓄力的及其所有的 `tagKV` 组合均被正确查出，结果如图 4.2 所示。

```
select s from ln.*;
```

```
IginX> select s from ln.*;
ResultSets:
```

	Time	ln.wf02.s	ln.wf02.s{t1=v1}	ln.wf03.s{t1=v1,t2=vv2}	ln.wf03.s{t1=vv1,t2=v2}
	1970-01-01T08:00:00.100	true	null	null	null
	1970-01-01T08:00:00.400	null	false	null	null
	1970-01-01T08:00:01.600	null	null	null	true
	1970-01-01T08:00:03.200	null	null	true	null

Total line number = 4

图 4.2

在此基础上，用户可以指定某个 `tagkv` 对，查询的序列下只要符合该 `tagkv` 组合的数据点都能被正确查出，结果如图 4.3 所示。

```
select s from ln.* with t1=v1;
```

```
IginX> select s from ln.* with t1=v1;
ResultSets:
```

	Time	ln.wf02.s{t1=v1}	ln.wf03.s{t1=v1,t2=vv2}
	1970-01-01T08:00:00.400	false	null
	1970-01-01T08:00:03.200	null	true

Total line number = 2

图 4.3

也可以指定多组不同的 `tagkv` 对并使用逻辑谓词相连，结果如图 4.4 和图 4.5 所

示。

```
select s from ln.* with t1=v1 or t2=v2;
IginX> select s from ln.* with t1=v1 or t2=v2;
ResultSets:
+-----+-----+-----+-----+
|          Time|ln.wf02.s{t1=v1}|ln.wf03.s{t1=v1,t2=vv2}|ln.wf03.s{t1=vv1,t2=v2}|
+-----+-----+-----+-----+
|1970-01-01T08:00:00.400|          false|                null|                null|
|1970-01-01T08:00:01.600|                null|                null|                true|
|1970-01-01T08:00:03.200|                null|                true|                null|
+-----+-----+-----+-----+
Total line number = 3
```

图 4.4

```
select s from ln.* with t1=v1 and t2=vv2;
IginX> select s from ln.* with t1=v1 and t2=vv2;
ResultSets:
+-----+-----+-----+-----+
|          Time|ln.wf03.s{t1=v1,t2=vv2}|
+-----+-----+-----+-----+
|1970-01-01T08:00:03.200|                true|
+-----+-----+-----+-----+
Total line number = 1
```

图 4.5

也可以不指定 tagv 的具体值，而采用通配符的形式，只要包含 t2 这个 tagk 即可，结果如图 4.6 所示。

```
select s from ln.* with t2=*
IginX> select s from ln.* with t2=*
ResultSets:
+-----+-----+-----+-----+
|          Time|ln.wf03.s{t1=v1,t2=vv2}|ln.wf03.s{t1=vv1,t2=v2}|
+-----+-----+-----+-----+
|1970-01-01T08:00:01.600|                null|                true|
|1970-01-01T08:00:03.200|                true|                null|
+-----+-----+-----+-----+
Total line number = 2
```

图 4.6



## 五、基于 Python 的数据处理流程：Transform

IGinX-Transform 允许用户通过 Python 编写自定义数据处理逻辑，并按照一定的顺序编排提交给 IGINX 执行，并将处理结果输出到标准流、指定文件或写回 IGINX。

### 5.1 环境准备

首先确保本地有 python3 环境。IGinX 的 Python UDF 依赖 Pemja 实现，需要在本地安装 PemjaX 依赖：

```
> pip install pemjax
```

我们要在配置文件里面设置本地的 python 执行路径：

```
# python 脚本启动命令，windows 设置为 pythonCMD=python  
pythonCMD=python3
```

建议设置为"which python"查询出的绝对路径，否则可能会找不到某些第三方依赖包，如下所示：

```
> which python  
> python: aliased to /Library/Frameworks/Python.framework/Versions/3.7/bin/python3  
  
# python 脚本启动命令  
pythonCMD=/Library/Frameworks/Python.framework/Versions/3.7/bin/python3
```

### 5.2 脚本编写与注册

注意：

1. 由于当前 transform 注册不支持文件在客户端与 IGINX 服务器之间传输，因此，要求被注册的 Python 脚本必须已经上传到 IGINX 服务器的相关路径下。
2. 同时，如果要求在多个 IGINX 上都可运行该 transform，则需要分别在多个 IGINX 节点上进行注册。

合法的 Python Transform 脚本至少要包括一个自定义类（名称不做要求）、和一个类成员函数 transform(self, rows)。

Transform 方法的输入输出均为一个二维列表，且必须是基础类型，用户可以在 transform 方法里面实现自己的数据处理逻辑。

一个合法的 Python 脚本 row\_sum.py 如下所示：

```
import pandas as pd  
import numpy as np
```

```
class RowSumTransformer:
    def __init__(self):
        pass

    def transform(self, rows):
        df = pd.DataFrame(rows)
        ret = np.zeros((df.shape[0], 2), dtype=np.integer)
        for index, row in df.iterrows():
            row_sum = 0
            for num in row[1:]:
                row_sum += num
            ret[index][0] = row[0]
            ret[index][1] = row_sum
        return pd.DataFrame(ret, columns=['key', 'sum']).values.tolist()
```

这个 Python 脚本实现了按行求和的功能。

实现完 Python 脚本之后，我们要将其注册到 IGINX 实例中：

```
CREATE FUNCTION transform <name> FROM <className> IN <filePath>;
```

其中，<className>为自定义脚本类名，<filePath>为 Python 脚本文件所在的绝对路径，<name>为 transform 脚本别名。

比如，假设我们将上面编写的 row\_sum.py 在 data/script 目录下注册到 IGINX：

```
CREATE FUNCTION transform "row_sum" "RowSumTransformer" IN
"data/script/row_sum.py";
```

注册完成后我们还能查询已注册的 Python 脚本：

```
SHOW FUNCTIONS;
```

或者删除已注册的 Python 脚本：

```
DROP FUNCTION <name>;
```

## 5.3 运行 Transform 作业

当做完上述的准备工作之后，我们就可以开始编排 Transform 作业，并提交给 IGINX 执行了。通过任务编排以进行使用，是 Transform 与下一章节所介绍的 UDF 之间的主要差别。相比之下 UDF 主要用于在 SQL 中进行使用。

IGINX 收到 Transform 作业，并在检查过作业合法性之后，会返回一个 JobId，并开始异步执行作业，用户可以通过 JobId 来查询作业的执行状态，任务总共有 8 种状态，如表 5.1 所示。

表 5.1

名称	描述
<i>JOB_UNKNOWN</i> (0)	作业状态未知
<i>JOB_FINISHED</i> (1)	作业完成（有设置重复调度时，表示所有调度完成）
<i>JOB_CREATED</i> (2)	作业创建
<i>JOB_IDLE</i> (3)	作业等待运行（有调度时）
<i>JOB_RUNNING</i> (4)	作业运行中
<i>JOB_FAILING</i> (5)	作业失败中（正在释放资源）
<i>JOB_FAILED</i> (6)	作业失败
<i>JOB_CLOSING</i> (7)	作业取消中（正在释放资源）
<i>JOB_CLOSED</i> (8)	作业取消

作业中的任务，根据其计算所依赖数据的局部性情况，会形成不同的数据流动方式，并形成 2 种不同类型的任务，如表 5.2 所示，即（1）对于给定输入数据集，如果任务可以仅使用部分数据，就输出这部分数据的正确计算结果，且部分结果直接合并即为全局结果，即为流式任务（2）对于给定输入数据集，如果任务需要获得所有输入数据后，才能输出全局正确结果，即为批式任务。

注意：任务类型的正确判断与定义，会直接影响作业的调度与性能。

表 5.2

dataFlowType	描述
stream	流式任务，数据会以流式的方式一批一批的回调 python 脚本中定义的 transform 方法，每批数据量的大小可以通过修改配置文件的 batchSize 项来定义
batch	批式任务，等上游全部数据都被读取进内存后才会回调 python 脚本中定义的 transform 方法

每一个合法的 Transform Job 都必须包含至少一个 task，且第一个 task 必须为一个 IGINX 的查询任务，以保证我们有数据源提供数据给后续的 Transform 脚本执行。

我们有两种方式可以提交执行，一种是编写 yaml 文件，并通过 SQL 提交执行作业；另一种是通过 IGINX-Session 提交作业，并通过 Session 查询作业状态。

### 5.3.1 方式 1: IGINX-SQL

一个合法的任务配置文件 job.yaml 如下所示:

```
taskList:
  - taskType: I GinX
    dataFlowType: stream
    timeout: 10000000
    sqllist:
      - select value1, value2, value3, value4 from transform;
  - taskType: python
    dataFlowType: stream
    timeout: 10000000
    pyTaskName: row_sum

#exportType: none
#exportType: I GinX
exportType: file
#exportFile: /path/to/your/output/dir
exportFile: /Users/cauchy-ny/Downloads/export_file_sum_sql.txt
exportNameList:
  - col1
  - col2
schedule: "after 10 second"
```

提交文件时, 应提供其绝对路径, 例如: job.yaml 位于 xxx/xxx/job.yaml, 则可以使用以下语句提交:

```
COMMIT TRANSFORM JOB "xxx/xxx/job.yaml";
```

通过返回的 JobId 查询 transform 作业状态

```
SHOW TRANSFORM JOB STATUS 12323232;
```

## 参数说明:

1. taskList: 定义 transform 任务列表, transform 任务可被看作是 SQL 语句和 python 脚本任务的序列。dataFlowType 指定数据流动方式, 可以是 stream 或者 batch。timeout 指定超时时间。可定义的任务一共有两种, 通过 taskType 指定:

- a. I GinX: 在 I GinX 中顺序执行多个 SQL 语句, SQL 语句列表通过“sqllist”指定
- b. python: 任务的上一步应该是以查询语句结尾的 I GinX 类型任务或者 python 任务, 即应是有数据输出的任务。这里 pyTaskName 指定的 python 函数会接收上一步输出的数据, 并进行脚本中定义的处理过程。

2. exportType: 定义 transform 任务结果的输出目标。一共有三种类型:

- a. none: 在服务端日志中输出

- b. I GinX: 将数据插入到 I GinX 系统中, 列名默认加上 transform. 前缀
- c. file: 将输出数据写入文件, 文件路径通过 exportFile 参数指定

3. schedule(optional): 控制 transform 任务的时间调度, 可以定时或者重复执行。schedule 字符串一共支持四种格式:

a. "every"- 重复执行

- every 3 second/minute/hour/day/month/year: 每 3 秒/分钟/小时/天/月/年 执行一次
- every 3 minute starts '2024-07-19 12:00:00' ends '2024-07-20 12:00:00': 从 2024-07-19 12:00:00 到 2024-07-20 12:00:00, 每三分钟执行一次。
- every 3 minute ends '23:59:59': 从现在到同一天的 23:59:59 , 每 3 分钟执行一次
- every 3 minute starts '13:00:00': 从今天的 13:00:00 开始, 每 3 分钟执行一次, 永不停止
- every mon,wed: 每周一、周三各执行一次

b. "after"- 延后执行

- after 3 second/minute/hour/day/month/year: 3 秒/分钟/小时/天/月/年后执行一次

c. "at"- 定时执行

- at '2024-07-19 12:00:00': 在 2024-07-19 12:00:00 执行一次
- at '12:00:00': 在今天的 12:00:00 执行一次

d. Cron 格式

如果需要更高级的时间调度, 可以使用 cron 字符串。cron 格式与 Quartz 调度器中的 cron 格式相同, 具体编写方法可以参考链接: <https://www.quartz-scheduler.org/documentation/quartz-2.3.0/tutorials/crontrigger.html>

- (0 0/1 \* 1/1 \* ? \*): 从一分钟之后开始, 每分钟执行一次

### 5.3.2 方式 2: I GinX-Session

```
// 构造任务
```

```
List<TaskInfo> taskInfoList = new ArrayList<>();
```

```
TaskInfo I GinXTask = new TaskInfo(TaskType.I GinX, DataFlowType.Stream);
I GinXTask.setSql("select value1, value2, value3, value4 from transform;");
taskInfoList.add(I GinXTask);
```

```
TaskInfo pyTask = new TaskInfo(TaskType.Python, DataFlowType.Stream);
pyTask.setPyTaskName("RowSumTransformer");
taskInfoList.add(pyTask);
```

```
String schedule = "after 10 second";
```

```
// 提交任务
```

```
long jobId = session.commitTransformJob(taskInfoList, ExportType.File, "data" +
File.separator + "export_file.txt");
System.out.println("job id is " + jobId);

// 轮询查看任务情况
JobState jobState = JobState.JOB_CREATED;
while (!jobState.equals(JobState.JOB_CLOSED)
&& !jobState.equals(JobState.JOB_FAILED)
&& !jobState.equals(JobState.JOB_FINISHED)) {
    Thread.sleep(500);
    jobState = session.queryTransformJobStatus(jobId);
}
System.out.println("job state is " + jobState.toString());

// 如果需要调度，使用：
long jobId = session.commitTransformJob(taskInfoList, ExportType.File, "data" +
File.separator + "export_file.txt", schedule);
```

## 六、基于 Python 的用户定义函数：UDF

### 6.1 概述

IGinX 提供了多种系统内置函数，包括 min、max、sum、avg、count、first、last 等，但某些时候在实际的运用过程中仍然不能满足我们所有的需求，这时我们提供了 IGINX UDF（User-Defined Function）。

IGinX UDF 使用户能通过编写 Python 脚本来定义自己所需要的函数，注册到 IGINX 元数据中，并在 IGINX-SQL 中使用，UDF 可以分为三类 UDTF、UDAF、UDSF。

#### 6.1.1 UDTF (user-defined time series function)

接受一行数据输入，并产生一行数据输出，如计算正弦值的 UDF 函数：

```
import math

class UDFCos:
    def __init__(self):
        pass

    def transform(self, data):
        res = self.buildHeader(data)
        cosRow = []
        for num in data[2]:
            cosRow.append(math.sin(num))
        res.append(cosRow)
        return res

    def buildHeader(self, data):
        colNames = []
        colTypes = []
        for name in data[0]:
            colNames.append("sin(" + name + ")")
            colTypes.append("DOUBLE")
        return [colNames, colTypes]
```

#### 6.1.2 UDAF (user-defined aggregate function)

接受多行数据输入，并产生一行数据输出，如计算平均值的 UDF 函数：

```
class UDFAvg:
    def __init__(self):
        pass
```

```

def transform(self, data):
    res = self.buildHeader(data)

    avgRow = []
    rows = data[2:]
    for row in zip(*rows):
        sum, count = 0, 0
        for num in row:
            if num is not None:
                sum += num
                count += 1
        avgRow.append(sum / count)
    res.append(avgRow)
    return res

def buildHeader(self, data):
    colNames = []
    colTypes = []
    for name in data[0]:
        colNames.append("udf_avg(" + name + ")")
        colTypes.append("DOUBLE")
    return [colNames, colTypes]

```

### 6.1.3 UDSF (user-defined set transform function)

接受多行数据输入，并产生多行数据输出，如按行逆转输出的 UDF 函数：

```

class UDFReverseRows:
    def __init__(self):
        pass

    def transform(self, data):
        res = self.buildHeader(data)
        res.extend(list(reversed(data[2:])))
        return res

    def buildHeader(self, data):
        colNames = []
        for name in data[0]:
            colNames.append("reverse_rows(" + name + ")")
        return [colNames, data[1]]

```

## 6.2 环境准备

首先确保本地有 python3 环境。IGinX 的 Python UDF 依赖 Pemja 实现，需要在本



地安装 PemjaX 依赖:

```
> pip install pemjax
```

我们要在配置文件里面设置本地的 python 执行路径:

```
# python 脚本启动命令, windows 设置为 pythonCMD=python  
pythonCMD=python3
```

建议设置为"which python"查询出的绝对路径, 否则可能会找不到某些第三方依赖包, 如下所示:

```
> which python  
> python: aliased to /Library/Frameworks/Python.framework/Versions/3.7/bin/python3  
  
# python 脚本启动命令  
pythonCMD=/Library/Frameworks/Python.framework/Versions/3.7/bin/python3
```

## 6.3 UDF 编写

合法的 Python UDF 脚本至少要包括一个自定义类 (名称不做要求)、和一个类成员函数 transform(self, data)。

```
class UDFFunction:  
    def __init__(self):  
        pass  
  
    """  
    data: 函数输入  
        对于 UDTF 来说是一个 3*n 的一维列表, 前两行分别为列名和数据类型,  
        第三行为数据行  
        对于 UDAF 和 UDSF 来说输入为一个 m*n 的二维列表, 前两行分别为列名  
        和数据类型, 第 3-n 行为数据行  
  
    return: 返回值  
        对于 UDTF 和 UDAF 来说, 返回的是一个 3*n 的一位列表, 前两行分  
        别为列名和数据类型, 第三行为数据行  
        对于 UDSF 来说输入为一个 m*n 的二维列表, 前两行分别为列名和数据  
        类型, 第 3-n 行为数据行  
    """  
    def transform(self, data):  
        // 实现自定义逻辑  
        return ret
```

注意：列名和数据类型均为 string 类型，合法的数据类型有以下几类：

1. boolean: 布尔类型，取值范围 true/false
  2. integer: 整数类型，取值范围 -21,4748,3648 ~ 21,4748,3647（需要注意的是，所有从 python 脚本返回 IGINX 的整数类型都会被转换为 java 的 Long 类型，因此在 transform 函数返回的二维列表中的第二行：数据类型里，不可使用 INTEGER，应当使用 LONG）
  3. long: 长整数类型，取值范围  $-2^{63} \sim 2^{63}-1$ （-922,3372,0368,5477,5808 ~ 922,3372,0368,5477,5807）
  4. float: 浮点数类型，取值范围 -21,4748,3648 ~ 21,4748,3647，小数点后可以保留 7 位有效数字
  5. double: 双精度浮点数类型，取值范围  $-2^{63} \sim 2^{63}-1$ （-922,3372,0368,5477,5808 ~ 922,3372,0368,5477,5807），小数点后可以保留 15 位有效数字
  6. binary: 二进制或字符串类型
- 三种合法的 UDF 示例可以参见概述部分。

## 6.4 UDF 注册

注意：

1. 由于当前 UDF 注册不支持文件在客户端与 IGINX 服务器之间传输，因此，要求被注册的 Python 脚本必须已经上传到 IGINX 服务器的相关路径下。
2. 同时，如果要求在多个 IGINX 上都可运行该 UDF，则需要分别在多个 IGINX 节点上进行注册。

对于 IGINX 来说，UDF 编写完成后还需要注册到 IGINX 中才能被使用，IGINX 主要需要知道以下信息：

1. UDF 名（可以与自定义类名不一致）
2. 自定义脚本类名
3. Python 脚本的文件名
4. 函数类型

对于 UDF 注册来说主要有以下两种方式，一是通过 SQL 注册，二是通过修改配置文件在启动时加载。

注：目前三类 UDF 仅支持单一入参，入参可以是单列名或者是通配符中的“\*”（表示同一行的所有列，以 Key 对齐）

### 6.4.1 方式 1：注册

```
CREATE FUNCTION <udfType> <udfName> FROM <className> (, (<udfType>)?  
<udfName> FROM <className>)* IN <filePath>;
```

```
udfType  
: UDAF  
| UDTF  
| UDSF
```

```
;
```

udfType 为函数类型，className 为自定义脚本类名，filePath 为 Python 脚本文件或模块文件夹所在的绝对路径，udfName 为 UDF 名。

以下是应用示例。我们编写了一个计算正弦值的 UDTF，文件名为 udtf\_sin.py，位于文件夹 aa/bb 下。

```
import math

class UDFCos:
    def __init__(self):
        pass

    def transform(self, data):
        res = self.buildHeader(data)
        cosRow = []
        for num in data[2]:
            cosRow.append(math.sin(num))
        res.append(cosRow)
        return res

    def buildHeader(self, data):
        colNames = []
        colTypes = []
        for name in data[0]:
            colNames.append("sin(" + name + ")")
            colTypes.append("DOUBLE")
        return [colNames, colTypes]
    return res
```

我们可以通过下面的语句，将其注册到 IGINX 中：

```
CREATE FUNCTION UDTF "sin" FROM "UDFSin" IN "aa/bb/udtf_sin.py";
```

## 6.4.2 方式 2：配置文件

IGINX 启动时，如果 needInitBasicUDFFunctions 选项被设置为 true，则会根据 udfList 中配置的 UDF 信息加载 python\_scripts 文件夹下的 Python 脚本，并注册到元信息中。

1. 将配置文件的 needInitBasicUDFFunctions 项设置为 true:

```
# 是否初始化配置文件内指定的 UDF
needInitBasicUDFFunctions=true
```

2. 将编写好的 Python 文件放在 python\_scripts 文件夹中。

3. 修改配置文件中的 udfList 配置项，不同的 udf 之间用 "," 隔开。

```
# 初始化 UDF 列表，用","分隔 UDF 实例
# 函数别名(用于 SQL)#类名#文件名#函数类型
udfList=udf_min#UDFMin#udf_min.py#UDAF,udf_max#UDFMax#udf_max.py#UDAF,udf_sum#UDFSum#udf_sum.py#UDAF,udf_avg#UDFAvg#udf_avg.py#UDAF,udf_count#UDFCount#udf_count.py#UDAF
```

以下是应用示例。我们编写了一个计算正弦值的 UDTF，文件名为 udtf\_sin.py，并将其拷贝到 python\_scripts 文件夹下。

```
import math

class UDFCos:
    def __init__(self):
        pass

    def transform(self, data):
        res = self.buildHeader(data)
        cosRow = []
        for num in data[2]:
            cosRow.append(math.sin(num))
        res.append(cosRow)
        return res

    def buildHeader(self, data):
        colNames = []
        colTypes = []
        for name in data[0]:
            colNames.append("sin(" + name + ")")
            colTypes.append("DOUBLE")
        return [colNames, colTypes]
```

我们可以修改配置文件，让 IGINX 在启动时自动将其注册到元信息中。

```
# 是否初始化配置文件内指定的 UDF
needInitBasicUDFFunctions=true

# 初始化 UDF 列表，用","分隔 UDF 实例
# 函数别名(用于 SQL)#类名#文件名#函数类型
udfList=sin#UDFSin#udf_sin.py#UDTF
```

## 6.5 使用示例

现在我们希望实现一个计算正弦值的自定义 Python 函数，并注册到 IGINX 中，并

用它来计算序列 root.a 的正弦值。

首先，我们编写这个脚本：

```
import math

class UDFCos:
    def __init__(self):
        pass

    def transform(self, data):
        res = self.buildHeader(data)
        cosRow = []
        for num in data[2]:
            cosRow.append(math.sin(num))
        res.append(cosRow)
        return res

    def buildHeader(self, data):
        colNames = []
        colTypes = []
        for name in data[0]:
            colNames.append("sin(" + name + ")")
            colTypes.append("DOUBLE")
        return [colNames, colTypes]
```

其次，我们将这个脚本注册到 IGINX 中，假设它在路径 aa/bb 下：

```
CREATE FUNCTION UDTF "sin" FROM "UDFSin" IN "aa/bb/udtf_sin.py";
```

最后，我们在 SQL 查询中使用这个函数：

```
SELECT sin(a) FROM root;
```

### 6.5.1 UDTF

对序列 test.a 序列中 key 小于 3 的每个值求余弦值，一行输入对应一行输出

```
IginX> select a from test where a <= 3
ResultSets:
+---+-----+
|key|test.a|
+---+-----+
| 1|    1|
| 2|    2|
| 3|    3|
+---+-----+
Total line number = 3
IginX> select cos(a) from test where a <= 3
ResultSets:
+---+-----+
|key|      cos(test.a)|
+---+-----+
| 1| 0.5403023058681398|
| 2|-0.4161468365471424|
| 3|-0.9899924966004454|
+---+-----+
Total line number = 3
```

## 6.5.2 UDAF

对序列 test.a 序列的 key 在 [0, 10) 范围内，每 3ns 聚合一次，滑窗步长为 2ns，多行输入对应一行输出

```
IginX> select a from test
ResultSets:
+---+-----+
|key|test.a|
+---+-----+
| 1|    1|
| 2|    2|
| 3|    3|
| 4|    4|
| 5|    5|
| 6|    6|
| 7|    7|
| 8|    8|
| 9|    9|
|10|   10|
+---+-----+
Total line number = 10
IginX> select udf_avg(a) from test GROUP [0, 10) BY 3ns slide 2ns;
ResultSets:
+---+-----+
|key|udf_avg(test.a)|
+---+-----+
| 0|          1.5|
| 2|          3.0|
| 4|          5.0|
| 6|          7.0|
| 8|          8.5|
+---+-----+
Total line number = 5
```

## 6.5.3 UDSF

对序列 test.a 序列的 key 在值小于 3 的范围内的数据的倒序，多行输入对应多行输出

```
IginX> select a from test where a <= 3
ResultSets:
+---+-----+
|key|test.a|
+---+-----+
| 1|    1|
| 2|    2|
| 3|    3|
+---+-----+
Total line number = 3
IginX> select reverse_rows(a) from test where a <= 3
ResultSets:
+-----+
|reverse_rows(test.a)|
+-----+
|                    3|
|                    2|
|                    1|
+-----+
Total line number = 3
```

## 6.5.4 例：基于 UDTF 实现不定长参数的函数进行查询

下面给出一个简单的 UDTF 不定长参数的例子，对于输入的参数按行求其乘积

### 6.5.4.1 Multiply 实现

```
class UDFMultiply:
    def __init__(self):
        pass

    def transform(self, data):
        res = self.buildHeader(data)
        multiplyRet = 1.0
        for num in data[2]:
            multiplyRet *= num
        res.append([multiplyRet])
        return res

    def buildHeader(self, data):
        retName = "multiply("
        for name in data[0]:
            retName += name + ", "
        retName = retName[:-2] + ")"
        return [[retName], ["DOUBLE"]]
```

### 6.5.4.2 使用上述函数定义进行查询的例子

```

IGinx> select * from test
ResultSets:
+-----+
|key|test.s1|test.s2|test.s3|
+-----+
| 1|    2|    3|    2|
| 2|    3|    1|    3|
| 3|    4|    3|    1|
| 4|    9|    7|    5|
| 5|    3|    6|    2|
| 6|    6|    4|    2|
+-----+
Total line number = 6
IGinx> SELECT multiply(s1, s2) FROM test;
ResultSets:
+-----+
|key|multiply(test.s1, test.s2)|
+-----+
| 1|                        6.0|
| 2|                        3.0|
| 3|                       12.0|
| 4|                       63.0|
| 5|                       18.0|
| 6|                       24.0|
+-----+
Total line number = 6
IGinx> SELECT multiply(s1, s2, s3) FROM test;
ResultSets:
+-----+
|key|multiply(test.s1, test.s2, test.s3)|
+-----+
| 1|                                12.0|
| 2|                                9.0|
| 3|                                12.0|
| 4|                               315.0|
| 5|                                36.0|
| 6|                                48.0|
+-----+
Total line number = 6
IGinx>

```



## 七、RESTful 访问接口

### 7.1 定义

依照 KairosDB，支持基于典型的时序数据访问 RESTful API，列出如表 7.1 所示：

表 7.1

IGinX 相关 RESTful 接口
http://[host]:[port]/api/v1/datapoints
http://[host]:[port]/api/v1/datapoints/query
http://[host]:[port]/api/v1/datapoints/delete
http://[host]:[port]/api/v1/metric

上述接口中，第一个主要涉及写入操作；第二个主要涉及查询相关操作；另外主要涉及删除操作。

其返回结果都为 RESTful 服务的状态码和提示信息，用于标识执行结果，另外查询请求会返回对应的 json 格式查询结果。状态码及其含义如表 7.2 所示：

表 7.2

RESTful 接口服务状态码
200 OK - [GET]：服务器成功返回用户请求的数据，该操作是幂等的（Idempotent）。
201 CREATED - [POST/PUT/PATCH]：用户新建或修改数据成功。
202 Accepted - [*]：表示一个请求已经进入后台排队（异步任务）
204 NO CONTENT - [DELETE]：用户删除数据成功。
400 INVALID REQUEST -[POST/PUT/PATCH]：用户发出的请求有错误，服务器没有进行新建或修改数据的操作，该操作是幂等的。
401 Unauthorized - [*]：表示用户没有权限（令牌、用户名、密码错误）。
403 Forbidden - [*] 表示用户得到授权（与 401 错误相对），但是访问是被禁止的。
404 NOT FOUND - [*]：用户发出的请求针对的是不存在的记录，服务器没有进

行操作，该操作是幂等的。
406 Not Acceptable - [GET]: 用户请求的格式不可得（比如用户请求 JSON 格式，但是只有 XML 格式）。
410 Gone -[GET]: 用户请求的资源被永久删除，且不会再得到的。
422 Unprocesable entity - [POST/PUT/PATCH] 当创建一个对象时，发生一个验证错误。
500 INTERNAL SERVER ERROR - [*]: 服务器发生错误，用户将无法判断发出的请求是否成功。

如果写入、查询和删除请求失败会返回状态码 **INVALID REQUEST**，在返回的正文会提供错误原因。

如果请求的路径错误会返回状态码 **NOT FOUND**，正文为空。

如果请求成功执行会返回状态码 **OK**，除查询请求会返回结果外剩余的请求正文均为空。

## 7.2 描述

目前支持的操作分写入、查询、删除三部分，部分操作需要在请求中附加一个 json 文件说明操作涉及的数据或数据范围。相关操作及接口访问具体定义描述如下：

### 7.2.1 写入操作

该操作实现插入一个或多个 **metric** 的方法。每个 **metric** 包含一个或多个数据点，并可通过 **tag** 添加该 **metric** 的分类信息便于查询。

#### 7.2.1.1 插入数据点

- 命令：  
`curl -XPOST -H'Content-Type: application/json' -d @insert.json http://\[host\]:\[port\]/api/v1/datapoints`
- insert.json 示例内容

```
[{
  "name": "archive_file.tracked",
  "datapoints": [
    [1359788400000, 123.3],
    [1359788300000, 13.2],
    [1359788410000, 23.1]
  ],
  "tags": {
    "host": "server1",
    "data_center": "DC1"
  },
}
```

```

        "annotation": {
            "category": ["cat1"],
            "title": "text",
            "description": "desp"
        }
    },
    {
        "name": "archive_file.search",
        "timestamp": 1359786400000,
        "value": 321,
        "tags": {
            "host": "server2"
        }
    }
}

```

- **insert.json 参数描述:**

该文件包含一个 metric 构成的数组，数组每一个值为一个表示一个 metric 的 json 字符串，字符串支持的 key 和对应的 value 含义如下：

- **name:** 必须包含该参数。表示需要插入的 metric 的名称，一个 metric 名称唯一对应一个 metric。
- **timestamp:** 插入单个数据点时使用，表示该数据点的时间戳，数值为从 UTC 时间 1970 年 1 月 1 日到该对应时间的毫秒数。
- **value:** 插入单个数据点时使用，表示该数据点的值，可以是数值或字符串。
- **datapoints:** 插入多个数据点时使用，内容为一个数组，每一个值表示一个数据点，为[timestamp, value]的形式（定义参照上述 timestamp 和 value）。
- **tags:** 必须包含该参数。表示为该 metric 添加的标签，可用于定向查找相应的内容。内容为一个字典，可自由添加键和值。可为空值。
- **annotation:** 可选项，用于标记该 metric 的所有数据点，须提供 category、title 和 description 三个域，其中 category 为字符串数组，title 和 description 为字符串。

## 7.2.2 查询操作

### 7.2.2.1 查询时间范围内的数据

该查询支持对一定时间范围内的数据执行查询，并返回查询结果。

- **命令:**  

```
curl -XPOST -H'Content-Type: application/json' -d @query.json http://[host]:[port]/api/v1/datapoints/query
```
- **query.json 示例内容**

```

{
    "start_absolute": 1,
    "end_relative": {
        "value": "5",
    }
}

```

```

        "unit": "days"
    },
    "metrics": [{
        "name": "archive_file_tracked",
        "tags": {
            "host": ["server1", "server2"],
            "data_center": ["DC1"]
        }
    },
    {
        "name": "archive_file_search"
    }
    ]
}

```

● query.json 参数描述:

- **start\_absolute/end\_absolute**: 表示查询对应的起始/终止的绝对时间, 数值为从 UTC 时间 1970 年 1 月 1 日到该对应时间的毫秒数。
- **start\_relative/end\_relative**: 表示查询对应的起始/终止的相对当前系统的时间, 值为包含两个键 **value** 和 **unit** 的字典。其中 **value** 对应采样间隔时间值, **unit** 对应单位, 值表示查询 "milliseconds", "seconds", "minutes", "hours", "days", "weeks", "months" 中的一个。
- (上述两组参数中, **start\_absolute** 和 **start\_relative** 必须包含且仅包含其中一个, **end\_absolute** 和 **end\_relative** 必须包含且仅包含其中一个)。
- **metrics**: 表示对应需要查询的不同 **metric**。值为一个数组, 数组中每一个值为一个字典, 表示一个 **metric**。字典的参数如下:
  - ◆ **name**: 必要参数, 表示需要查询的 **metric** 的名字。
  - ◆ **tags**: 可选参数, 表示需要查询的 **metrics** 中对应的 **tag** 信息需要符合的范围。**tags** 对应的值为一个字典, 其中每一个键表示查询结果必须包含该 **tag**, 该键对应的值为一个数组, 表示查询结果中这个 **tag** 的值必须是该数组中所有值中的一个。例子中 **archive\_file\_tracked** 的 **tags** 参数表示查询结果中 **data\_center** 标签的值必须为 **DC1**, **host** 标签的值必须为 **server1** 或 **server2**。

### 7.2.2.2 包含聚合的查询

该查询支持对相应的查询结果进行均值 (**avg**)、方差 (**dev**)、计数 (**count**)、首值 (**first**)、尾值 (**last**)、最大值 (**max**)、最小值 (**min**)、求和值 (**sum**)、一阶差分 (**diff**)、除法 (**div**)、值过滤 (**filter**)、另存为 (**save\_as**)、变化率 (**rate**)、采样率 (**sampler**)、百分数 (**percentile**) 这些聚合查询。

若需要执行包含聚合的查询, 相应查询 **json** 文件结构基本与章节 7.2.2.1 中一致, 在需要执行聚合查询的 **metric** 项中添加 **aggregators** 参数, 表示这一聚合器的具体信息。**aggregators** 对应的值为一个数组, 数组中每一个值表示一个特定的 **aggregator** (上述 15 种功能之一), 查询结果按照 **aggregator** 出现的顺序按照顺序输出。

同时, 部分聚合查询需要采样操作, 即从查询的起始时间每隔一定的时间得到聚合结果。采样操作需要在 **aggregators** 对应的某一个 **aggregator** 值中添加 **sampling** 的字

典，包含两个键 value 和 unit。其中 value 对应采样间隔时间值，unit 对应单位，可为 "milliseconds", "seconds", "minutes", "hours", "days", "weeks", "months", "years" 中的一个。

每一种 aggregator 的具体功能的实例和详细参数如下所示：

#### 1. 均值聚合查询 (avg)

■ 含义：查询对应范围内数据的平均值，仅对数值类型数据有效。

■ 命令：

```
curl -XPOST -H'Content-Type: application/json' -d @avg_query.json  
http://[host]:[port]/api/v1/datapoints/query
```

■ avg\_query.json 示例内容

```
{  
  "start_absolute": 1,  
  "end_relative": {  
    "value": "5",  
    "unit": "days"  
  },  
  "metrics": [{  
    "name": "test_query",  
    "tags": {  
      "host": [  
        "server2"  
      ]  
    },  
    "aggregators": [{  
      "name": "avg",  
      "sampling": {  
        "value": 2,  
        "unit": "seconds"  
      }  
    }  
  ]  
}]  
}
```

■ avg\_query.json 参数描述：

◆ name：表示该聚合查询的类型，必须为"avg"。

◆ sampling：必要参数，value 对应采样间隔时间值，unit 对应单位，如章节 7.2.2.1 所述。

#### 2. 方差聚合查询 (dev)

■ 含义：查询对应范围内数据的方差，仅对数值类型数据有效。

■ 命令：

```
curl -XPOST -H'Content-Type: application/json' -d @dev_query.json  
http://[host]:[port]/api/v1/datapoints/query
```

■ dev\_query.json 示例内容

```
{
```

```

    "start_absolute": 1,
    "end_relative": {
      "value": "5",
      "unit": "days"
    },
    "metrics": [{
      "name": "test_query",
      "aggregators": [{
        "name": "dev",
        "sampling": {
          "value": 2,
          "unit": "seconds"
        },
        "return_type": "value"
      }]
    }]
  }
}

```

■ dev\_query.json 参数描述:

- ◆ name: 表示该聚合查询的类型, 必须为"dev".
- ◆ sampling: 必要参数, value 对应采样间隔时间值, unit 对应单位, 如章节 7.2.2.1 所述。

3. 计数聚合查询 (count)

- 含义: 查询对应范围内数据点的个数。

■ 命令:

```
curl -XPOST -H'Content-Type: application/json' -d @count_query.json
http://[host]:[port]/api/v1/datapoints/query
```

■ count\_query.json 示例内容

```

{
  "start_absolute": 1,
  "end_relative": {
    "value": "5",
    "unit": "days"
  },
  "metrics": [{
    "name": "test_query",
    "aggregators": [{
      "name": "count",
      "sampling": {
        "value": 2,
        "unit": "seconds"
      }
    }]
  }]
}

```

■ count\_query.json 参数描述:

- ◆ name: 表示该聚合查询的类型, 必须为"count".
- ◆ sampling: 必要参数, value 对应采样间隔时间值, unit 对应单位, 如章节 7.2.2.1 所述。

4. 首值聚合查询 (first)

■ 含义: 查询对应范围内数据的时间戳最早的数据点的值。

■ 命令:

```
curl -XPOST -H'Content-Type: application/json' -d @first_query.json
http://[host]:[port]/api/v1/datapoints/query
```

■ first\_query.json 示例内容

```
{
  "start_absolute": 1,
  "end_relative": {
    "value": "5",
    "unit": "days"
  },
  "metrics": [{
    "name": "test_query",
    "aggregators": [{
      "name": "first",
      "sampling": {
        "value": 2,
        "unit": "seconds"
      }
    }]
  }]
}
```

■ first\_query.json 参数描述:

- ◆ name: 表示该聚合查询的类型, 必须为"first".
- ◆ sampling: 必要参数, value 对应采样间隔时间值, unit 对应单位, 如章节 7.2.2.1 所述。

5. 尾值聚合查询 (last)

■ 含义: 查询对应范围内数据的时间戳最晚的数据点的值。

■ 命令:

```
curl -XPOST -H'Content-Type: application/json' -d @last_query.json
http://[host]:[port]/api/v1/datapoints/query
```

■ last\_query.json 示例内容

```
{
  "start_absolute": 1,
  "end_relative": {
    "value": "5",
```

```

        "unit": "days"
    },
    "metrics": [{
        "name": "test_query",
        "aggregators": [{
            "name": "last",
            "sampling": {
                "value": 2,
                "unit": "seconds"
            }
        }]
    }]
}

```

■ last\_query.json 参数描述:

- ◆ name: 表示该聚合查询的类型, 必须为"last"。
- ◆ sampling: 必要参数, value 对应采样间隔时间值, unit 对应单位, 如章节 7.2.2.1 所述。

6. 最大值聚合查询 (max)

- 含义: 查询对应范围内数据的最大值, 仅对数值类型数据有效。

■ 命令:

```
curl -XPOST -H'Content-Type: application/json' -d @max_query.json
http://[host]:[port]/api/v1/datapoints/query
```

■ max\_query.json 示例内容

```

{
    "start_absolute": 1,
    "end_relative": {
        "value": "5",
        "unit": "days"
    },
    "metrics": [{
        "name": "test_query",
        "aggregators": [{
            "name": "max",
            "sampling": {
                "value": 2,
                "unit": "seconds"
            }
        }]
    }]
}

```

■ max\_query.json 参数描述:

- ◆ name: 表示该聚合查询的类型, 必须为"max"。
- ◆ sampling: 必要参数, value 对应采样间隔时间值, unit 对应单位, 如章节



7.2.2.1 所述。

#### 7. 最小值聚合查询 (min)

■ 含义：查询对应范围内数据的最小值，仅对数值类型数据有效。

■ 命令：

```
curl -XPOST -H'Content-Type: application/json' -d @min_query.json
http://[host]:[port]/api/v1/datapoints/query
```

■ min\_query.json 示例内容

```
{
  "start_absolute": 1,
  "end_relative": {
    "value": "5",
    "unit": "days"
  },
  "metrics": [{
    "name": "test_query",
    "aggregators": [{
      "name": "min",
      "sampling": {
        "value": 2,
        "unit": "seconds"
      }
    }]
  }]
}
```

■ min\_query.json 参数描述：

◆ name：表示该聚合查询的类型，必须为"min"。

◆ sampling：必要参数，value 对应采样间隔时间值，unit 对应单位，如章节 7.2.2.1 所述。

#### 8. 求和值聚合查询 (sum)

■ 含义：查询对应范围内数据的所有数值的和，仅对数值类型数据有效。

■ 命令：

```
curl -XPOST -H'Content-Type: application/json' -d @sum_query.json
http://[host]:[port]/api/v1/datapoints/query
```

■ sum\_query.json 示例内容

```
{
  "start_absolute": 1,
  "end_relative": {
    "value": "5",
    "unit": "days"
  },
  "metrics": [{
    "name": "test_query",
    "aggregators": [{
```

```

        "name": "sum",
        "sampling": {
            "value": 2,
            "unit": "seconds"
        }
    }
}

```

■ **sum\_query.json 参数描述:**

- ◆ **name:** 表示该聚合查询的类型，必须为"sum"。
- ◆ **sampling:** 必要参数，value 对应采样间隔时间值，unit 对应单位，如章节 7.2.2.1 所述。

9. 一阶差分聚合查询 (diff)

- **含义:** 查询对应范围内数据的一阶差分，仅对数值类型数据有效。

■ **命令:**

```
curl -XPOST -H'Content-Type: application/json' -d @diff_query.json
http://[host]:[port]/api/v1/datapoints/query
```

■ **diff\_query.json 示例内容**

```

{
    "start_absolute": 1,
    "end_relative": {
        "value": "5",
        "unit": "days"
    },
    "metrics": [{
        "name": "test_query",
        "aggregators": [{
            "name": "diff"
        }]
    }]
}

```

■ **diff\_query.json 参数描述:**

- ◆ **name:** 表示该聚合查询的类型，必须为"diff"。

10. 除法聚合查询 (div)

- **含义:** 返回对应时间范围内每一个数据除以一个定值后的值，仅对数值类型数据有效。

■ **命令:**

```
curl -XPOST -H'Content-Type: application/json' -d @div_query.json
http://[host]:[port]/api/v1/datapoints/query
```

■ **div\_query.json 示例内容**

```

{

```

```

    "start_absolute": 1,
    "end_relative": {
      "value": "5",
      "unit": "days"
    },
    "metrics": [{
      "name": "test_query",
      "aggregators": [{
        "name": "div",
        "divisor": "2"
      }]
    }]
  }

```

- **div\_query.json 参数描述:**
  - ◆ **name:** 表示该聚合查询的类型，必须为"div"。
  - ◆ **divisor:** 表示对应除法的除数。

#### 11. 值过滤聚合查询 (filter)

- **含义:** 查询对应范围内数据进行简单值过滤后的结果，仅对数值类型数据有效。
- **命令:**

```
curl -XPOST -H'Content-Type: application/json' -d @filter_query.json http://[host]:[port]/api/v1/datapoints/query
```
- **filter\_query.json 示例内容**

```

{
  "start_absolute": 1,
  "end_relative": {
    "value": "5",
    "unit": "days"
  },
  "metrics": [{
    "name": "test_query",
    "aggregators": [{
      "name": "filter",
      "filter_op": "lt",
      "threshold": "25"
    }]
  }]
}

```

- **filter\_query.json 参数描述:**
  - ◆ **name:** 表示该聚合查询的类型，必须为"filter"。
  - ◆ **filter\_op:** 表示执行值过滤对应的符号，值可为"lte", "lt", "gte", "gt", "equal", 分别代表“大于等于”、“大于”、“小于等于”、“小于”、“等于”。
  - ◆ **threshold:** 表示值过滤对应的阈值，为数值。

#### 12. 另存为聚合查询 (save\_as)

- 含义：该操作将查询得到的结果保存到一个新的 metric 中。
- 命令：  
curl -XPOST -H'Content-Type: application/json' -d @save\_as\_query.json  
http://[host]:[port]/api/v1/datapoints/query
- save\_as\_query.json 示例内容

```
{
  "start_absolute": 1,
  "end_relative": {
    "value": "5",
    "unit": "days"
  },
  "metrics": [{
    "name": "test_query",
    "aggregators": [{
      "name": "save_as",
      "metric_name": "test_save_as"
    }]
  }]
}
```

- save\_as\_query.json 参数描述：
  - ◆ name：表示该聚合查询的类型，必须为"save\_as"。
  - ◆ metric\_name：表示将查询结果保存到新的 metric 的名称。

### 13. 变化率聚合查询（rate）

- 含义：查询对应范围内数据在每两个相邻区间的变化率，仅对数值类型数据有效。
- 命令：  
curl -XPOST -H'Content-Type: application/json' -d @rate\_query.json  
http://[host]:[port]/api/v1/datapoints/query
- rate\_query.json 示例内容

```
{
  "start_absolute": 1,
  "end_relative": {
    "value": "5",
    "unit": "days"
  },
  "metrics": [{
    "name": "test_query",
    "aggregators": [{
      "name": "rate",
      "sampling": {
        "value": 1,
        "unit": "seconds"
      }
    }]
  }]
}
```

```
    }
  }
}
```

■ **rate\_query.json 参数描述:**

- ◆ **name:** 表示该聚合查询的类型，必须为"rate"。
- ◆ **sampling:** 必要参数，value 对应采样间隔时间值，unit 对应单位，如章节 7.2.2.1 所述。

14. 采样率聚合查询 (sampler)

- **含义:** 查询对应范围内数据的采样率。

■ **命令:**

```
curl -XPOST -H'Content-Type: application/json' -d @sampler_query.json
http://[host]:[port]/api/v1/datapoints/query
```

■ **sampler\_query.json 示例内容**

```
{
  "start_absolute": 1,
  "end_relative": {
    "value": "5",
    "unit": "days"
  },
  "metrics": [{
    "name": "test_query",
    "aggregators": [{
      "name": "sampler",
      "unit": "minutes"
    }]
  }]
}
```

■ **sampler\_query.json 参数描述:**

- ◆ **name:** 表示该聚合查询的类型，必须为"sampler"。
- ◆ **unit:** 必要参数，对应单位，如章节 7.2.2.1 所述。

15. 百分位数聚合查询 (percentile)

- **含义:** 计算数据在目标区间的概率分布，并返回该分布的指定百分位数。这一查询仅对数值类型数据有效。

■ **命令:**

```
curl -XPOST -H'Content-Type: application/json' -d @percentile_query.json
http://[host]:[port]/api/v1/datapoints/query
```

■ **percentile\_query.json 示例内容**

```
{
  "start_absolute": 1,
  "end_relative": {
    "value": "5",
```

```

        "unit": "days"
    },
    "metrics": [{
        "name": "test_query",
        "aggregators": [{
            "name": "percentile",
            "sampling": {
                "value": "5",
                "unit": "seconds"
            },
            "percentile": "0.75"
        }]
    }]
}

```

■ percentile\_query.json 参数描述:

- ◆ name: 表示该聚合查询的类型, 必须为"percentile"。
- ◆ sampling: 必要参数, value 对应采样间隔时间值, unit 对应单位, 如章节 7.2.2.1 所述。
- ◆ percentile: 为需要查询的概率分布中的百分比值, 定义为  $0 < \text{percentile} \leq 1$ , 其中 0.75 为第 75% 大的值, 1 为 100% 即最大值。

## 7.2.3 删除操作

该操作实现两个删除方法: 包括通过时间范围、metric 信息查询并删除符合条件的所有数据点的方法, 和直接根据 metric 名称删除某一个 metric 与其中所有数据点的方法。

### 7.2.3.1 删除数据点

- 命令:
 

```
curl -XPOST -H'Content-Type: application/json' -d @delete.json http://[host]:[port]/api/v1/datapoints/delete
```
- delete.json 示例内容

```

{
    "start_absolute": 1,
    "end_relative": {
        "value": "5",
        "unit": "days"
    },
    "metrics": [{
        "name": "test_query",
        "tags": {
            "host": ["server2"]
        }
    }]
}

```

```
}
```

- delete.json 参数描述：
  - 参数含义参见章节 7.2.2.1“查询时间范围内的数据”。

### 7.2.3.2 删除 metric

- 命令：  
`curl -XDELETE http://[host]:[port]/api/v1/metric/[metric_name]`  
其中，metric\_name 表示对应需要删除的 metric 的名称。

## 7.3 特性

IGinX 的 RESTful 接口具备以下访问特性：

- 接口定义与实现符合 RESTful 通用规范。
- RESTful 接口与 API 接口同时提供服务，互不干扰。

## 7.4 性能

- 数据精确度：与底层时序数据库相同。
- 吞吐性能特性：IGinX RESTful 服务也是无状态的，因此，当应用连接增加的时候，可以实时进行任意规模的扩展，从而确保底层单实例数据库的性能可得到全面体现，即 IGINX RESTful 服务不会成为系统瓶颈。

## 八、基于 RESTful 实现序列段打标记

### 8.1 添加标记

添加标记功能支持对于满足条件的数据增加标记。即，给出时间范围、要求添加标签的时序列信息，以及要添加的标签信息，向这段时序列添加给定标签，具体说明如下：

- 命令：  
`curl -XPOST -H'Content-Type: application/json' -d @add.json http://[host]:[port]/api/v1/datapoints/annotations/add`
- add.json 示例内容

```
{
  "start_absolute": 1359788400000,
  "end_absolute": 1359788400001,
  "metrics": [{
    "name": "archive_file_tracked.ann",
    "tags": {
      "host": ["server1"],
      "data_center": ["DC1"]
    },
    "annotation": {
      "category": ["cat3", "cat4"],
      "title": "titleNewUp",
      "description": "dspNewUp"
    }
  }], {
    "name": "archive_file_tracked.bcc",
    "tags": {
      "host": ["server1"],
      "data_center": ["DC1"]
    },
    "annotation": {
      "category": ["cat3", "cat4"],
      "title": "titleNewUpbcc",
      "description": "dspNewUpbcc"
    }
  }
}]
}
```

- add.json 参数描述：
  - start\_absolute/end\_absolute：必要参数，表示查询对应的起始/终止的绝对时间，数值为从 UTC 时间 1970 年 1 月 1 日到该对应时间的毫秒数。
  - start\_relative/end\_relative：必要参数，表示查询对应的起始/终止的相对当前系统的时间，值为包含两个键 value 和 unit 的字典。其中 value 对应采样间隔时



间值, unit 对应单位, 值表示查询"milliseconds","seconds", "minutes", "hours", "days", "weeks", "months"中的一个。

- 该文件包含一个 metric 构成的数组, 数组每一个值为一个表示一个 metric 的 json 字符串, 字符串支持的 key 和对应的 value 含义如下:

- ◆ name: 必要参数, 表示需要查询的 metric 的名字。
- ◆ tags: 必要参数, 表示需要添加标签的 metrics 中对应的 tag 信息需要符合的范围。tags 对应的值为一个字典, 其中每一个键表示查询结果必须包含该 tag, 该键对应的值为一个数组, 表示要添加标签的对象中这个 tag 的值必须是该数组中所有值中的一个。
- ◆ annotation: 必须包含该参数, 用于标记该 metric 的所有数据点, 包含 category、title 和 description 三个域, 其中 category 为字符串数组, 且为必要参数, title 和 description 为字符串, 为非必要参数。

## 8.2 更新标记

对于已经添加的标记, 还可以通过更新标记的功能对其更新。即, 给出要更新标签的时序列信息 (包括其已有的标签信息), 以及要最终要更新为的标签信息, 更新此段时序列的标签信息, 具体说明如下:

- 命令:  
`curl -XPOST -H'Content-Type: application/json' -d @update.json http://[host]:[port]/api/v1/datapoints/annotations/update`
- update.json 示例内容

```
{
  "metrics": [{
    "name": "archive_file_tracked.ann",
    "tags": {
      "host": ["server2"],
      "data_center": ["DC2"]
    },
    "annotation": {
      "category": ["cat3"]
    },
    "annotation-new": {
      "category": ["cat6"],
      "title": "titleNewUp111",
      "description": "dspNewUp111"
    }
  }], {
    "name": "archive_file_tracked.bcc",
    "tags": {
      "host": ["server1"],
      "data_center": ["DC1"]
    },
    "annotation": {
```

```

        "category": ["cat2"]
    },
    "annotation-new": {
        "category": ["cat6"],
        "title": "titleNewUp111bcc",
        "description": "dspNewUp111bcc"
    }
}
}

```

- **update.json 参数描述:**
  - 该文件包含一个 **metric** 构成的数组，数组每一个值为一个表示一个 **metric** 的 json 字符串，字符串支持的 **key** 和对应的 **value** 含义如下：
    - ◆ **name:** 必要参数，表示需要更新的 **metric** 的名字。
    - ◆ **tags:** 必要参数，表示需要更新标签的 **metrics** 中对应的 **tag** 信息需要符合的范围。**tags** 对应的值为一个字典，其中每一个键表示查询结果必须包含该 **tag**，该键对应的值为一个数组，表示要添加标签的对象中这个 **tag** 的值必须是该数组中所有值中的一个。
    - ◆ **annotation:** 必须包含该参数，表示需要更新标签的 **metrics** 中对应的 **annotation** 信息需要符合的范围，包含 **category** 一个域，其中 **category** 为字符串数组，且为必要参数。**category** 该键对应的值为一个数组，表示要添加标签的对象中这个 **tag** 的值必须包含该数组中所有的值。
    - ◆ **annotation-new:** 必须包含该参数，表示需要最终 **annotation** 的更新结果，包含 **category**、**title** 和 **description** 三个域，其中 **category** 为字符串数组，且为必要参数，**title** 和 **description** 为字符串，为非必要参数。

## 8.3 查询标记

### 1. 给定时间序列查询标记

给定单条或多条时间序列，查询标记功能可以返回其包含的标记集合，具体如下：

- **命令:**

```
curl -XPOST -H'Content-Type: application/json' -d @query.json http://[host]:[port]/api/v1/datapoints/query/annotations
```
- **query.json 示例内容**

```

{
  "metrics": [{
    "name": "archive_file_tracked.ann",
    "tags": {
      "host": ["server1"],
      "data_center": ["DC1"]
    }
  }, {
    "name": "archive_file_tracked.bcc",
    "tags": {

```

```

        "host": ["server1"],
        "data_center": ["DC1"]
    }
}

```

- query.json 参数描述:

- 该文件包含一个 metric 构成的数组，数组每一个值为一个表示一个 metric 的 json 字符串，字符串支持的 key 和对应的 value 含义如下:

- ◆ name: 必要参数，表示需要查询的 metric 的名字。
- ◆ tags: 必要参数，表示要查询的 metrics 中对应的 tag 信息需要符合的范围。tags 对应的值为一个字典，其中每一个键表示查询结果必须包含该 tag，该键对应的值为一个数组，表示要添加标签的对象中这个 tag 的值必须是该数组中所有值中的一个。

2. 给定标记查询时间序列及数据点

相应地，除了给定时间序列查询标记，查询标记功能还支持给定标记查询时间序列及数据点，具体说明如下:

- 命令:

```
curl -XPOST -H'Content-Type: application/json' -d @ query.json
http://[host]:[port]/api/v1/datapoints/query/annotations/data
```

- query.json 示例内容如下:

```

{
  "metrics": [{
    "annotation": {
      "category": ["cat4"]
    }
  }, {
    "annotation": {
      "category": ["cat3"]
    }
  }]
}

```

- query.json 参数描述:

- 该文件包含一个 metric 构成的数组，数组每一个值为一个表示一个 metric 的 json 字符串，字符串支持的 key 和对应的 value 含义如下:

- ◆ annotation: 必须包含该参数，为要查询的 metric 的需要包含的标签信息，包含 category、title 两个域，其中 category 为字符串数组，且为必要参数，title 为字符串，为非必要参数。category 该键对应的值为一个数组，表示要添加标签的对象中这个 tag 的值必须包含该数组中所有的值。

## 8.4 删除标记

对于已经添加的标记，如果发现添加错误，可以通过删除标记的功能将其删除，具体说明如下：

- 命令：  
curl -XPOST -H'Content-Type: application/json' -d @delete.json http://[host]:[port]/api/v1/datapoints/annotations/delete
- delete.json 示例内容

```
{
  "metrics": [{
    "name": "archive_file_tracked.ann",
    "tags": {
      "host": ["server2"],
      "data_center": ["DC2"]
    },
    "annotation": {
      "category": ["cat3", "cat4"]
    }
  }]
}
```

- delete.json 参数描述：
  - 该文件包含一个 metric 构成的数组，数组每一个值为一个表示一个 metric 的 json 字符串，字符串支持的 key 和对应的 value 含义如下：
    - ◆ name: 必要参数，表示需要查询的 metric 的名字。
    - ◆ tags: 必要参数，表示需要添加标签的 metrics 中对应的 tag 信息需要符合的范围。tags 对应的值为一个字典，其中每一个键表示查询结果必须包含该 tag，该键对应的值为一个数组，表示要添加标签的对象中这个 tag 的值必须是该数组中所有值中的一个。
    - ◆ annotation: 必须包含该参数，为要查询的 metric 的需要包含的标签信息，包含 category、title 两个域，其中 category 为字符串数组，且为必要参数，title 为字符串，为非必要参数。category 该键对应的值为一个数组，表示要添加标签的对象中这个 tag 的值必须包含该数组中所有的值。

## 九、 数据访问接口

### 9.1 定义

如表 9.1 所示，IGinX 支持基于典型的数据访问 API：

表 9.1

IGinX 接口
OpenSessionResp openSession(1:OpenSessionReq req);
Status closeSession(1:CloseSessionReq req);
Status deleteColumns(1:DeleteColumnsReq req);
Status insertRowRecords(1:InsertRowRecordsReq req);
Status insertNonAlignedRowRecords(1:InsertNonAlignedRowRecordsReq req);
Status insertColumnRecords(1:InsertColumnRecordsReq req);
Status insertNonAlignedColumnRecords(1:InsertNonAlignedColumnRecordsReq req);
Status deleteDataInColumns(1:DeleteDataInColumnsReq req);
QueryDataResp queryData(1:QueryDataReq req);
Status addStorageEngines(1: AddStorageEnginesReq req);
Status removeHistoryDataSource(1: RemoveHistoryDataSourceReq req);
AggregateQueryResp aggregateQuery(1:AggregateQueryReq req);
LastQueryResp lastQuery(1: LastQueryReq req);
DownsampleQueryResp downsampleQuery(DownsampleQueryReq req);
ShowColumnsResp showColumns(1: ShowColumnsReq req);
GetReplicaNumResp getReplicaNum(1: GetReplicaNumReq req);
ExecuteSqlResp executeSql(1: ExecuteSqlReq req);

Status updateUser(1: UpdateUserReq req);
Status addUser(1: AddUserReq req);
Status deleteUser(1: DeleteUserReq req);
GetUserResp getUser(1: GetUserReq req);
GetClusterInfoResp getClusterInfo(1: GetClusterInfoReq req);
ExecuteStatementResp executeStatement(1: ExecuteStatementReq req);
FetchResultsResp fetchResults(1: FetchResultsReq req);
Status closeStatement(1: CloseStatementReq req);
CommitTransformJobResp commitTransformJob(1: CommitTransformJobReq req);
QueryTransformJobStatusResp queryTransformJobStatus(1: QueryTransformJobStatusReq req);
ShowEligibleJobResp showEligibleJob(1: ShowEligibleJobReq req);
Status cancelTransformJob (1: CancelTransformJobReq req);
Status registerTask(1: RegisterTaskReq req);
Status dropTask(1: DropTaskReq req);
GetRegisterTaskInfoResp getRegisterTaskInfo(1: GetRegisterTaskInfoReq req);
CurveMatchResp curveMatch(1: CurveMatchReq req);
DebugInfoResp debugInfo(1: DebugInfoReq req);
ShowSessionIDResp showSessionID(1: ShowSessionIDReq req);
LoadCSVResp loadCSV(1: LoadCSVReq req);
ShowRulesResp showRules(1: ShowRulesReq req);
Status setRules(1: SetRulesReq req);

## 9.2 描述

各接口具体含义如下：

- **openSession**: 创建 Session
  - 输入参数: IP, 端口号, 用户名和密码
  - 返回结果: 是否创建成功, 如果成功, 返回分配的 Session ID
- **closeSession**: 关闭 Session
  - 输入参数: 待关闭的 Session 的 ID
  - 返回结果: 是否关闭成功
- **deleteColumns**: 删除列
  - 输入参数: 待删除的列名称列表
  - 返回结果: 是否删除成功
- **insertRowRecords**: 行式插入数据
  - 输入参数: 列名称列表、时间戳列表、数据列表、数据类型列表和额外参数
  - 返回结果: 是否插入成功
  - 说明: 数据列表是二维的, 内层以列组织, 外层以行组织; 数据不强制要求对齐
- **insertNonAlignedRowRecords**: 行式插入非对齐数据
  - 输入参数: 列名称列表、时间戳列表、数据列表、数据类型列表和额外参数
  - 返回结果: 是否插入成功
  - 说明: 数据列表是二维的, 内层以列组织, 外层以行组织; 数据不强制要求对齐
- **insertColumnRecords**: 列式插入数据
  - 输入参数: 列名称列表、时间戳列表、数据列表、数据类型列表和额外参数
  - 返回结果: 是否插入成功
  - 说明: 数据列表是二维的, 内层以行组织, 外层以列组织; 数据要求对齐
- **insertNonAlignedColumnRecords**: 列式插入非对齐的数据
  - 输入参数: 列名称列表、时间戳列表、数据列表、数据类型列表和额外参数
  - 返回结果: 是否插入成功
  - 说明: 数据列表是二维的, 内层以行组织, 外层以列组织; 数据要求对齐
- **deleteDataInColumns**: 删除数据
  - 输入参数: 列名称列表、开始时间戳和结束时间戳
  - 返回结果: 是否删除成功
- **queryData**: 原始数据查询
  - 输入参数: 列名称列表、开始时间戳和结束时间戳
  - 返回结果: 查询结果集, 可以提供列名称列表、时间戳列表、数据列表和数据类型列表等信息
- **addStorageEngines**: 增加存储引擎
  - 输入参数: 待添加的存储引擎列表
  - 返回结果: 是否添加成功
  - 说明: 存储引擎类必须包含存储引擎 ip 地址、端口、存储引擎类型和其他参数 extraParams (字典类型)。目前存储引擎类型包括: iotdb12、influxdb、

parquet、postgresql、mongodb、redis、filesystem、opentsdb、timescaledb。

- **removeHistoryDataSource**: 删除存储引擎
  - 输入参数: 待移除的存储引擎信息列表
  - 返回结果: 是否移除成功
  - 说明: 待移除的存储引擎信息类必须包含存储引擎 ip 地址、端口、模式前缀 schemaPrefix 和 dataPrefix。
    - **aggregateQuery**: 聚合查询
      - 输入参数: 列名称列表、开始时间戳、结束时间戳和聚合查询类型
      - 返回结果: 查询结果集, 可以提供列名称列表、时间戳列表、数据列表和数据类型列表等信息
      - 说明: 目前聚合查询支持最大值(MAX)、最小值(MIN)、求和(SUM)、计数(COUNT)、平均值(AVG)、第一个非空值(FIRST)和最后一个非空值(LAST)七种
- **lastQuery**: Last 查询 (最新值查询)
  - 输入参数: 列名称列表、开始时间戳
  - 返回结果: 查询结果集, 可以提供列名称列表和数据类型列表等信息
    - **downsampleQuery**: 降采样查询
      - 输入参数: 列名称列表、开始时间戳、结束时间戳、聚合类型以及聚合查询的精度
      - 返回结果: 查询结果集, 可以提供列名称列表、时间戳列表、数据列表和数据类型列表等信息
      - 说明: 目前降采样查询中的聚合, 支持最大值(MAX)、最小值(MIN)、求和(SUM)、计数(COUNT)、平均值(AVG)、第一个非空值(FIRST)和最后一个非空值(LAST)七种
- **showColumns**: 查询序列
  - 输入参数: 无
  - 返回结果: 查询结果集, 可以提供列名称列表和数据类型列表等信息。
- **getReplicaNum**: 查询副本数量
  - 输入参数: 无
  - 返回结果: 目前 IGINX 存储的副本数量。
    - **executeSql**: sql 查询
      - 输入参数: IGINX-SQL 语句
      - 返回结果: 查询结果集, 可以提供列名称列表、时间戳列表、数据列表和数据类型列表等信息
      - 说明: 具体使用方式见章节 3
- **updateUser**: 更新用户权限
  - 输入参数: 用户名、密码和权限
  - 返回结果: 是否更新成功
- **addUser**: 添加用户
  - 输入参数: 用户名、密码和权限
  - 返回结果: 是否添加成功
- **deleteUser**: 删除用户



- 输入参数: 用户名
  - 返回结果: 是否删除成功
- **getUser:** 获取用户
  - 输入参数: 用户名列表
  - 返回结果: 用户名对应的用户信息, 包括名称、类型、权限
- **getClusterInfo:** 获取集群信息
  - 输入参数: 无
  - 返回结果: 获取当前集群信息
- **executeStatement:** 执行流式查询
  - 输入参数: 查询语句、获取一批数据的行数、超时时间
  - 返回结果: 返回第一批数据, 其中包括这次流式查询的 `queryId`
- **fetchResults:** 获取流式查询后续结果数据
  - 输入参数: `queryId`、获取一批数据的行数、超时时间
  - 返回结果: 返回下一批数据
  - 说明: 如果剩下的行数小于制定行数, 则返回全部数据
- **closeStatement:** 关闭流式查询
  - 输入参数: `queryId`
  - 返回结果: 是否关闭成功
- **commitTransformJob:** 提交 Transform 任务
  - 输入参数: 提交的 task 信息, 和导出地址和导出文件名
  - 返回结果: 任务 id
- **queryTransformJobStatus:** 查询 Transform 任务状态
  - 输入参数: 任务 id
  - 返回结果: 当前任务的运行状态
- **showEligibleJob:** 查询某一状态的 Transform 任务
  - 输入参数: 任务状态
  - 返回结果: 任务 id 列表
- **cancelTransformJob:** 取消 Transform 任务
  - 输入参数: 任务 id
  - 返回结果: 是否取消成功
- **registerTask:** 注册 python udf
  - 输入参数: udf 名、文件路径、类名和 udf 类型
  - 返回结果: 是否注册成功
- **dropTask:** 删除 python udf
  - 输入参数: udf 名
  - 返回结果: 是否删除成功
- **getRegisterTaskInfo:** 获取已注册的 udf
  - 输入参数: 无
  - 返回结果: 已注册的 udf 列表
- **curveMatch:** 曲线匹配
  - 输入参数: 路径列表、开始和结束 key、匹配的数据点、单位
  - 返回结果: 匹配的路径和开始 key

- debugInfo: 获得调试信息
  - 输入参数: 负载类型和负载
  - 返回结果: 对应的调试信息 (目前只支持获得元数据调试信息)
  - 说明: 目前负载类型只支持 GET\_META
    - showSessionID: 查询 session id
  - 输入参数: 无
  - 返回结果: 返回当前 session id
- showRules: 查询优化规则
  - 输入参数: 无
  - 返回结果: 当前正在生效的优化规则
- executeLoadCSV
  - 输入参数: 3.2.1.3 章节所涉及将 CSV 数据插入 IGINX 的相关语句, 以及相关 CSV 文件
  - 返回结果: 插入的行数以及列头信息

## 9.3 特性

- 连接池: 将前端应用程序查询复用到底层数据库连接池中以优化性能。
- IGINX 可进行面向单机版数据库的并行查询, 从而提高数据库查询相关性能。

## 9.4 性能

- 数据精度: 与底层数据库相同。
- 吞吐性能特性: IGINX 是无状态的, 因此, 当应用连接增加的时候, 可以实时进行任意规模的扩展, 从而确保底层单实例数据库的性能可得到全面体现, 即 IGINX 不会成为系统瓶颈。

# 十、新版数据访问接口

## 10.1 安装方法

- 安装命令

```
> mvn clean install -pl session -am -Dmaven.test.skip=true
```

- 依赖

- JDK >= 1.8

- Maven >= 3.6

- 在 Maven 中添加依赖包

```
<dependency>
  <groupId>cn.edu.tsinghua</groupId>
  <artifactId>IGinX-session</artifactId>
  <version>0.7.1</version>
</dependency>
```

## 10.2 接口说明

IGinX 提供 IGINXClient 接口，来提供查询、写入、删除、用户管理、集群操作的能力。

```
public interface IGINXClient extends AutoCloseable {

    WriteClient getWriteClient();

    AsyncWriteClient getAsyncWriteClient();

    QueryClient getQueryClient();

    DeleteClient getDeleteClient();

    UsersClient getUserClient();

    ClusterClient getClusterClient();

    void close();
}
```

## 10.3 初始化

IGinXClient 由 IGINXClientFactory 提供的工厂方法进行创建。该工厂允许用户传入 url 或者 host + port 以及用户名密码，来创建 Client:

```
public static IGINXClient create(); // 使用默认用户名密码，连接到本地的 6888 端口部
```

署的 IginX

```
public static IginXClient create(String url);

public static IginXClient create(String host, int port);

public static IginXClient create(String url, String username, String password);

public static IginXClient create(String host, int port, String username, String password);
```

还可以传入封装各种参数的 IginXClientOptions 实例，来创建 Client:

```
public static IginXClient create(IginXClientOptions options);
```

若干创建 Client 的样例:

```
IginXClient client;

client = IginXClientFactory.create();
client = IginXClientFactory.create("127.0.0.1:6324");

client = IginXClientFactory.create("127.0.0.1", 2333, "root", "root");

client = IginXClientFactory.create(
    IginXClientOptions.builder()
        .host("192.172.1.102").port(2123).username("user").password("password")
        .build()
);
```

## 10.4 数据写入

IginX 支持数据的同步写入与异步写入。其中，同步的写入接口包括:

```
public interface WriteClient {

    void writePoint(final Point point) throws IginXException;

    void writePoints(final List<Point> points) throws IginXException;

    void writeRecord(final Record record) throws IginXException;

    void writeRecords(final List<Record> records) throws IginXException;

    <M> void writeMeasurement(final M measurement) throws IginXException;
```

```

    <M> void writeMeasurements(final List<M> measurements) throws I GinXException;

    void writeTable(final Table table) throws I GinXException;

}

```

异步的写入接口除了不会抛出 `I GinXException`（`RuntimeException`）外，与同步的写入接口完全相同。

```

public interface AsyncWriteClient extends AutoCloseable {

    void writePoint(final Point point);

    void writePoints(final List<Point> points);

    void writeRecord(final Record record);

    void writeRecords(final List<Record> records);

    <M> void writeMeasurement(final M measurement);

    <M> void writeMeasurements(final List<M> measurements);

    void writeTable(final Table table) throws InterruptedException;

    @Override
    void close() throws Exception;

}

```

数据可以以数据点、数据行、数据表以及数据对象的形式，以同步或异步的方式写入到 `I GinX` 集群中。

### 10.4.1 数据点

数据点为某个时间序列在给定时间点的数据采样。`I GinX` 支持写入单个数据点，也一次写入多个数据点（不一定属于同一个时间序列）。

```

WriteClient writeClient = client.getWriteClient();

// 写入一个数据点： cpu.usage.host1 1640918819147 87.4
writeClient.writePoint(
    Point.builder()
        .timestamp(1640918819147L)
        .measurement("cpu.usage.host1")
        .doubleValue(87.4)

```

```

        .build()
    );

    // 写入两个数据点:
    // cpu.usage.host2 1640918819147 66.3
    // user.login.host2 1640918819147 admin
    writeClient.writePoints(
        Arrays.asList(
            Point.builder()
                .now()
                .measurement("cpu.usage.host2")
                .doubleValue(66.3)
                .build(),
            Point.builder()
                .now()
                .measurement("user.login.host2")
                .binaryValue("admin".getBytes(StandardCharsets.UTF_8))
                .build()
        )
    );

```

### 10.4.2 数据行

数据行是多个时间序列在同一时刻的数据采样。IGinX 支持写入单行，也一次写入多行（多个行的时间序列之间可以相同也可以不同）。

```

AsyncWriteClient asyncWriteClient = client.getAsyncWriteClient();

// 异步写入一行数据，包含了 4 个时间序列:
// memory.usage.host1 now() 33.4
// memory.usage.host2 now() 24.1
// memory.usage.host3 now() 76.4
// memory.usage.host4 now() 99.8
asyncWriteClient.writeRecord(
    Record.builder()
        .measurement("memory.usage")
        .now()
        .addDoubleField("host1", 33.4)
        .addDoubleField("host2", 24.1)
        .addDoubleField("host3", 76.4)
        .addDoubleField("host4", 99.8)
        .build()
);

```

### 10.4.3 数据表

数据表是多个时间序列在多个时间戳的数据采样。

```

WriteClient writeClient = client.getWriteClient();

// 写入如下的二维表:
//      Time          cpu.usage.host1 cpu.usage.host2 cpu.usage.host3
//      1640918819147    23.1          22.1          null
//      1640918820147    null          77.1          86.1

long timestamp = System.currentTimeMillis() - 1000;
writeClient.writeTable(
    Table.builder()
        .measurement("cpu.usage")
        .addField("host1", DataType.DOUBLE)
        .addField("host2", DataType.DOUBLE)
        .addField("host3", DataType.DOUBLE)
        .timestamp(timestamp)
        .doubleValue("host1", 23.1)
        .doubleValue("host2", 22.1)
        .next()
        .timestamp(timestamp + 1000)
        .doubleValue("host2", 77.1)
        .doubleValue("host3", 86.1)
        .build()
);

```

#### 10.4.4 数据对象

IGinX 也支持直接写入数据对象，不过数据对象的域必须均为基本类型以及字节数组和字符串。

```

// 待写入的数据对象
@Measurement(name = "demo.pojo")
static class POJO {

    @Field(timestamp = true)
    long time;

    @Field(name = "field_int")
    int a;

    @Field(name = "field_double")
    double b;

    POJO(long time, int a, double b) {
        this.time = time;
        this.a = a;
        this.b = b;
    }
}

```

```

    }
}

WriteClient writeClient = client.getWriteClient();

writeClient.writeMeasurement(
    new POJO(1640918819147L, 10, 45.1)
);

// POJO 会转化为 2 个序列进行存储：
// demo.pojo.field_int, 类型是 integer
// demo.pojo.field_double, 类型是 double

```

## 10.5 数据查询

集群操作通过 QueryClient 接口来提供。具体接口如下：

```

public interface QueryClient {

    // 数据查询，返回一个二维数据表
    IGINXTable query(final Query query) throws IGINXException;

    // 数据查询，并将结果集映射成对象列表（每行映射为一个对象）
    <M> List<M> query(final Query query, final Class<M> measurementType) throws
    IGINXException;

    // 传入 SQL 语句进行查询，返回一个二维数据表
    IGINXTable query(final String query) throws IGINXException;

    // 数据查询，并通过传入的 consumer 流式消费查询结果的每一行
    void query(final Query query, final Consumer<IGINXRecord> onNext) throws
    IGINXException;

    // 传入 SQL 语句进行查询，通过传入的 consumer 流式消费查询结果的每一行
    void query(final String query, final Consumer<IGINXRecord> onNext) throws
    IGINXException;

    // 传入 SQL 语句进行查询，并将结果集映射成对象列表（每行映射为一个对
    象）
    <M> List<M> query(final String query, final Class<M> measurementType) throws
    IGINXException;

    // 传入 SQL 语句进行查询，将结果集映射成对象列表（每行映射为一个对象），
    并通过传入的 consumer 流式消费查询结果

```



```

    <M> void query(final String query, final Class<M> measurementType, final
Consumer<M> onNext) throws I GinXException;

    // 数据查询，将结果集映射成对象列表（每行映射为一个对象），并通过传入的
consumer 流式消费查询结果
    <M> void query(final Query query, final Class<M> measurementType, final
Consumer<M> onNext) throws I GinXException;
}

```

### 10.5.1 一般查询

通常查询可以传入一个 Query 对象，也可以直接传入一条 SQL 语句。现有的 Query 包括 SimpleQuery、DownsampleQuery、AggregateQuery、LastQuery 等等。

```

QueryClient queryClient = client.getQueryClient();

IGinXTable table = queryClient.query( // 查询 a.a.a 序列最近一秒内的数据
    SimpleQuery.builder()
        .addMeasurement("a.a.a")
        .startTime(System.currentTimeMillis() - 1000L)
        .endTime(System.currentTimeMillis())
        .build()
);

table = queryClient.query( // 查询 a.a.a 序列最近 60 秒内的数据，并以 1 秒为单位进
行聚合
    DownsampleQuery.builder()
        .addMeasurement("a.a.a")
        .startTime(System.currentTimeMillis() - 60 * 1000L)
        .endTime(System.currentTimeMillis())
        .precision(1000)
        .build()
);

```

查询出的结果为一张二维表，由表头和若干行组成，下述代码会遍历整张数据表并打印：

```

IGinXHeader header = table.getHeader();
if (header.hasTimestamp()) {
    System.out.print("Time\t");
}
for (IGinXColumn column: header.getColumns()) {
    System.out.print(column.getName() + "\t");
}
System.out.println();

```

```
List<IGinXRecord> records = table.getRecords();
for (IGinXRecord record: records) {
    if (header.hasTimestamp()) {
        System.out.print(record.getTimestamp());
    }
    for (IGinXColumn column: header.getColumns()) {
        System.out.print(record.getValue(column.getName()));
        System.out.print("\t");
    }
    System.out.println();
}
```

### 10.5.2 对象映射

IGinX 支持将查询到的二维表 IGINXTable 的每一行映射成对象，来进行处理。整个对象映射可以视为写入数据对象的逆过程。

```
@Measurement(name = "demo.pojo")
static class POJO {

    @Field(timestamp = true)
    long timestamp;

    @Field(name = "field_int")
    int a;

    @Field(name = "field_double")
    double b;

    POJO(long timestamp, int a, double b) {
        this.timestamp = timestamp;
        this.a = a;
        this.b = b;
    }
}

QueryClient queryClient = client.getQueryClient();
List<POJO> pojoList = queryClient.query("select * from demo.pojo where time < now() and time > now() - 1000", POJO.class); // 查询最近一秒内的 pojo 对象
```

### 10.5.3 流式读取

无论是一般查询还是对象查询，IGinX 都支持流式的消费查询结果。只需要在执行查询时候，额外传入一个 Consumer，用于顺序消费每一行数据即可。

```
QueryClient queryClient = client.getQueryClient();
```

```
// 查询 a.a 开头的序列在最近一小时内的数据
queryClient.query("select * from a.a where time < now() and time > now() - 1h", new
Consumer<IGinXRecord>() {
    @Override
    public void accept(IGinXRecord record) {
        // 打印该行的数据
        IGINXHeader header = record.getHeader();
        if (header.hasTimestamp()) {
            System.out.print(record.getTimestamp());
        }
        for (IGINXColumn column: header.getColumns()) {
            System.out.print(record.getValue(column.getName()));
            System.out.print("\t");
        }
        System.out.println();
    }
});
```

## 10.6 数据删除

删除操作通过 DeleteClient 接口来提供。具体接口如下：

```
public interface DeleteClient {

    // 删除某个时间序列
    void deleteMeasurement(final String measurement) throws IGINXException;

    // 删除多个时间序列
    void deleteMeasurements(final Collection<String> measurements) throws
IGINXException;

    // 删除某个类对应的时间序列
    void deleteMeasurement(final Class<?> measurementType) throws IGINXException;

    // 删除某个时间序列在 [startTime, endTime) 这段时间上的数据
    void deleteMeasurementData(final String measurement, long startTime, long endTime)
throws IGINXException;

    // 删除多个时间序列在 [startTime, endTime) 这段时间上的数据
    void deleteMeasurementsData(final Collection<String> measurements, long startTime,
long endTime) throws IGINXException;

    // 删除某个类对应的时间序列在 [startTime, endTime) 这段时间上的数据
    void deleteMeasurementData(final Class<?> measurementType, long startTime, long
endTime) throws IGINXException;
```

```
}
```

## 10.7 用户管理

用户管理通过 `UsersClient` 接口来提供。具体接口如下：

```
public interface UsersClient {  
  
    // 增加新用户  
    void addUser(final User user) throws IGinxException;  
  
    // 更新用户权限  
    void updateUser(final User user) throws IGinxException;  
  
    // 更新用户密码  
    void updateUserPassword(final String username, final String newPassword) throws  
IGinxException;  
  
    // 根据用户名删除用户  
    void removeUser(final String username) throws IGinxException;  
  
    // 根据用户名查询用户  
    User findUserByName(final String username) throws IGinxException;  
  
    // 获取系统所有的用户信息  
    List<User> findUsers() throws IGinxException;  
  
}
```

## 10.8 集群操作

集群操作通过 `ClusterClient` 接口来提供。具体接口如下：

```
public interface ClusterClient {  
  
    // 获取集群的拓扑结构  
    ClusterInfo getClusterInfo() throws IGinxException;  
  
    // 集群扩容：增加一个底层存储节点  
    void scaleOutStorage(final Storage storage) throws IGinxException;  
  
    // 集群扩容：增加多个底层存储节点  
    void scaleOutStorages(final List<Storage> storages) throws IGinxException;  
  
}
```

```
// 获取集群副本数  
int getReplicaNum() throws I GinXException;  
}
```

其中，扩容逻辑相对复杂，样例代码如下：

```
ClusterClient clusterClient = client.getClusterClient();  
  
// 向集群中新增一个 iotdb 节点和 influxdb 节点：  
// iotdb: 10.10.1.122:6667  
// influxdb: 10.10.1.123:6668  
clusterClient.scaleOutStorages(  
    Arrays.asList(  
        Storage.builder()  
            .ip("10.10.1.122")  
            .port(6667)  
            .type("iotdb11")  
            .build(),  
        Storage.builder()  
            .ip("10.10.1.123")  
            .port(6668)  
            .type("influxdb")  
            .build()  
    )  
);
```

# 十一、存储扩容功能

IGinX 可进行 2 个层次上的扩容操作，即 IGINX 层和时序数据库层。

## 11.1 IGINX 扩容操作

为 IGINX 设置待扩容集群的 ZooKeeper 相关 IP 及端口后，启动 IGINX 实例即可：

```
zookeeperConnectionString=127.0.0.1:2181
```

## 11.2 底层数据库扩容操作

在已有集群基础上，要增加底层数据库节点，我们需要执行以下 3 个步骤：

1. 启动客户端，给定一个 IGINX 所在的 IP 及其端口（详见章节 2.5.3）。

```
/sbin/start_cli.sh -h 192.168.10.43 -p 6324
```

(Windows 环境中应当使用 start\_cli.bat 脚本)

2. 进行命令行交互，输入以下命令，可以增加一个 IP 为 127.0.0.1，端口为 6667，数据目录为/usr/home/data/parquetData 的 Parquet 数据引擎：

```
ADD STORAGEENGINE ("127.0.0.1", 6667, "parquet", "dir:/usr/home/data/parquetData, iginx_port=6888");
```

3. 客户端回复“success”，即扩容成功。此时，可输入“quit”退出客户端。

注：在扩容的时候，一般假设原数据库无数据，即使已经存在数据，在 IGINX 的整个视图中，原本的数据不可见，仅后续经 IGINX 写入的数据才可见

## 11.3 数据库扩容操作参数说明

### 11.3.1 Parquet 存储

Parquet 存储引擎以对接层的方式添加到 IGINX 中，支持在添加时配置额外参数，支持的参数为：

参数名	描述	类型	默认值
dir	IGINX 数据存储目录	String	无
dummy_dir	历史数据文件读取目录	String	无
embedded_prefix	历史数据的数据路径前缀	String	dummy_dir 指定的目录的名称
thrift_timeout	发起 Thrift 连接的超时时间，单位为毫秒	Integer	30000
close.flush	是否在关闭时写出缓冲区内容	Boolean	true
write.buffer.size	写入缓冲区大小，单位为字节	Long	100*1024*1024
write.buffer.timeout	缓冲区超时时间，缓冲区创	Duration	0 秒，即立即写

	建后达到超时时间会将内容强制写出到文件。配置格式遵循 ISO-8601 持续时间格式，详见： <a href="#">Duration</a>		出
write.batch.size	单次最大写入大小，单位为字节	Long	1024*1024
compact.permits	写入缓冲区刷写为文件的最大并发数	Integer	16
cache.capacity	读缓存的最大容量，单位为字节	Long	1024*1024*1024
cache.timeout	读缓存中数据的超时淘汰时间，格式为 ISO-8601 规定的持续时间格式。例如，“PT1H”表示一小时，“PT20.345S”--解析为“20.345 秒”，“PT15M”--解析为“15 分钟”，“PT10H”——解析为“10 小时”，“P2D”--解析为“2 天”，“P2DT3H4M”--解析为“2 天 3 小时 4 分钟”	String	无超时时间
cache.value.soft	是否以 soft value 的形式保存读缓冲区中的数据。若为 true，则在 JVM 因缺少内存而 GC 时，会回收读缓存中数据占用的内存。	Boolean	false
parquet.block.size	写 Parquet 文件时的最大行组大小，单位为字节	Long	128*1024*1024
parquet.page.size	写 Parquet 文件时的最大页大小，单位为字节	Long	8*1024

### 11.3.2 文件目录存储

Filesystem 存储引擎以对接层的方式添加到 IGINX 中，支持在添加时配置额外参数，支持的参数为：

参数名	描述	类型	默认值
dir	IGINX 数据存储目录	String	无
dummy_dir	历史数据文件读取目录	String	无
chunk_size_in_bytes	以字节流形式读取历史文件，每次读取的大小，以 byte 为单位	Long	1024*1024
memory_pool_size	内存池中 chunk 的数量，即	Long	100

	提前申请的 <b>chunk</b> 的数量。 文件系统读取字节流时会将数据保存在内存中，所以申请内存池以减少 GC 压力		
thrift_pool_max_size	IGinX 上的文件系统之间连接时，能创建的连接池的最大容量	Long	100
thrift_pool_min_evictable_idle_time_millis	thrift 连接池中，idle 连接超时关闭的时间阈值	Long	600000
thrift_timeout	发起 Thrift 连接的超时时间，单位为毫秒	Integer	30000

### 11.3.3 MongoDB 对接层

Mongodb 对接层支持在添加时配置额外参数，支持的参数为：

参数名	描述	类型	默认值
uri	Mongodb 连接字符串，用于指定服务器地址和详细连接参数，具体格式详见 MongoDB 官方文档	String	根据 IP 和 Port 生成
schema.sample.size	获取 Dummy 数据当前存在的列和类型时，对接层需要从 MongoDB 每个集合采样若干文档分析当前存在的列和类型，该配置用于设置每次采样的文档数量。若该配置小于等于 0，则只在列名中以前缀的方式提供数据库名称和集合名称。	Integer	1000
dummy.sample.size	读取 Dummy 数据时，首先采样一定数量的文档分析数据模式（所有列和类型），然后根据模式从 MongoDB 中逐个读取文档，该配置用于设置每次采样的文档；若该配置小于等于 0，则将需要读取的所有文档一次性加载到 IGINX 分析以分析数据模式。前者占用内存较少，但是由于数据模式是通过采样分析得到，如果 MongoDB 中的文档没有固定模式，则有可能遗漏某些数据。	Integer	0

### 11.3.4 Redis 对接层

Redis 对接层支持在添加时配置额外参数，支持的参数为：

参数名	描述	类型	默认值
-----	----	----	-----



username	用户名，默认不指定用户名	String	null
password	密码，默认不使用密码鉴权	String	null
timeout	Redis 客户端请求超时时间，单位毫秒	Integer	5000

### 11.3.5 Relational 对接层

连接 PostgreSQL、MySQL 等关系数据库时，可在 `storageEngineList` 处配置 HikariDataSource 连接池的参数

配置项	描述	默认值
connection_timeout	连接超时时间（单位：毫秒）	30000
idle_timeout	空闲连接超时时间（单位：毫秒）	10000
maximum_pool_size	连接池中的最大连接数	20
minimum_idle	连接池中的最小空闲连接数	1
leak_detection_threshold	检测连接泄漏的阈值（单位：毫秒）	2500
prep_stmt_cache_size	SQL 预编译对象缓存个数	250
prep_stmt_cache_sql_limit	SQL 预编译对象缓存个数上限	2048

## 十二、数据源接入功能

### 12.1 功能简介

IGinX 支持数据源接入，即带数据的节点的扩容功能，具体包括：

1. 用户可以使用带有数据的节点创建集群。
2. 扩容时，用户可以选择添加带有数据的节点，并标识新加入的节点为只读/读写状态。
3. 扩容完毕后，用户可以对新加入的读写状态的节点进行写入，并且保证后续的写入不会覆盖原有的数据。
4. 扩容完毕后，用户可以对新加入的读写状态的节点上的数据进行查询。
5. 扩容完毕后，后续的数据不会写入到新加入的只读状态的节点上。
6. 扩容完毕后，用户可以对新加入的只读状态的节点上的数据进行查询。
7. 用户可以使用命令行、JDBC、各种语言版本的原生接口进行数据扩容。

### 12.2 配置参数

在创建初始集群时，用户如果选择使用带有数据的节点，那么除了原有的各项系统参数外，每个节点可以额外指定表 12.1 中给出的三个参数，且均为可选项：

表 12.1

参数名	描述	类型	默认值
has_data	用来标识该节点是否带有数据，配置为 <code>true</code> 表示该节点带有数据，配置为 <code>false</code> 表示该节点不带有数据	Boolean	false
data_prefix	数据前缀，特指，原数据库中带该前缀的数据才能加入 IGINX 中被查询到	String	null
is_read_only	用来标识该节点是否为只读状态，配置为 <code>true</code> 表示该节点为只读状态，即该节点在加入 IGINX 后不再接收写入数据；配置为 <code>false</code> 表示该节点为读写状态，即该节点在加入 IGINX 后接受读写请求，但是，加入前已经存在的数据是只读的（不可更新、不可删除）	Boolean	false
schema_prefix	模式前缀，给整个数据节点中的所有时序列都加上一个虚拟前缀。例如，一个节点中有时序列	String	null

	a.b、a.c、a.d，通过该选项为该数据节点上加上前缀"prefix"，则在查询时，所有的时序列对应变为 prefix.a.b 、 prefix.a.c 、 prefix.a.d。		
--	---	--	--

## 12.3 运行说明

IGinX 执行带数据的节点扩容功能的逻辑与创建初始集群类似，也是通过表 1 中三个可选的参数来进行控制。具体 SQL 语句如下：

```
add storageengine (ip, port, engineType, extra)
```

ip 和 port 分别为节点的 IP 地址和端口号，engineType 是节点的数据库类型，extra 用于填写上述的三个参数。具体样例如下：

```
add storageengine ("127.0.0.1", 6667, "iotdb12", "username:root, password:root, sessionPoolSize:130, read_only:true, data_prefix:demo, has_data:true, schema_prefix:prefix")
```

含义为添加一个 IP 地址为 127.0.0.1，端口号为 6667，数据库类型为 0.11.x 版本的 iotdb，用户名为 root，密码为 root，Session 池大小为 130，状态为只读，数据前缀为 demo，并且所有加入的所有时序列都获取了一个虚拟前缀 prefix，且带有数据的新节点。

## 十三、多数据库/存储扩展实现

IGinX 目前支持的底层存储包括 InfluxDB、IoTDB、PostgreSQL、MongoDB、Redis、Parquet 文件目录、本地文件系统等七种，用户可根据需要自行扩展其他类型的数据库或者存储。异构部署的 IGINX 架构如图 13.1 所示。

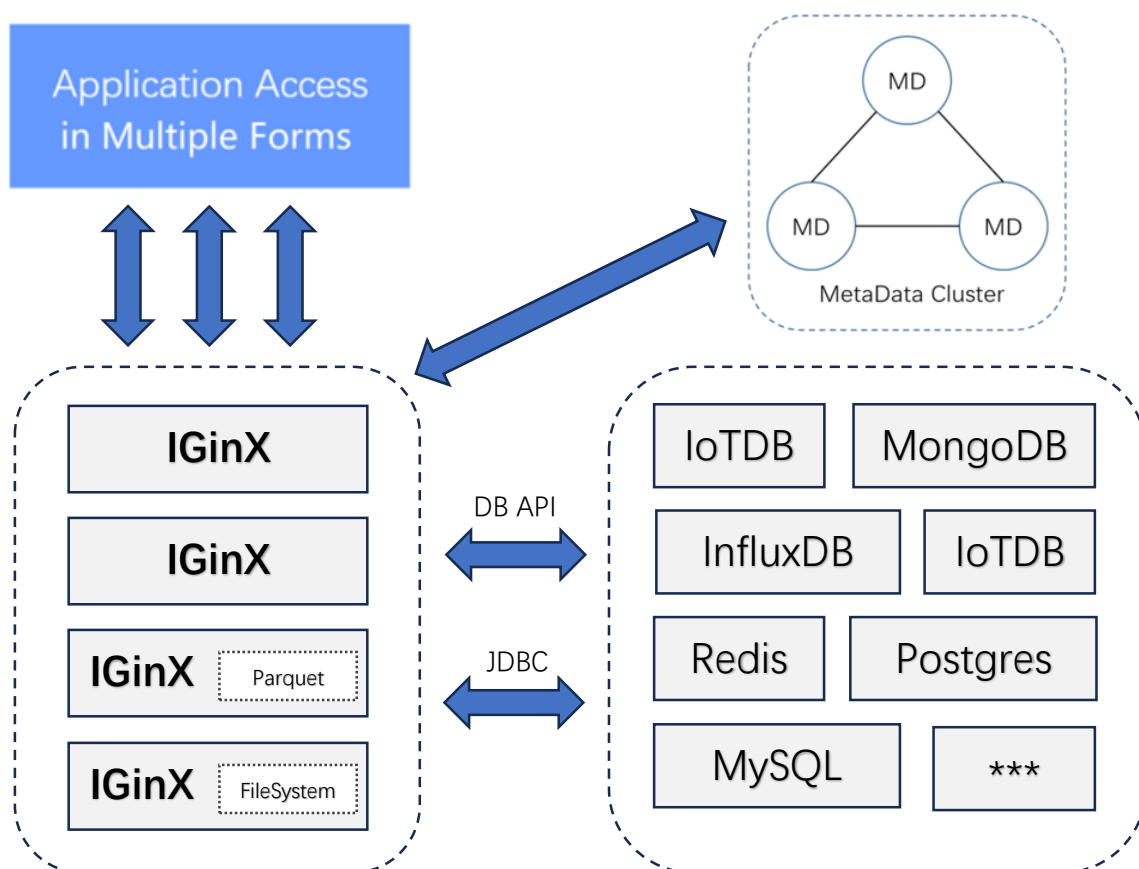


图 13.1 IGINX 部署架构图

### 13.1 需支持的功能

其他类型的时序数据库如果想要成为 IGINX 的数据后端，必须支持以下功能：

- 对某个分片（叠加/非叠加）查询指定的数据。
  - 对某个非叠加分片删除数据。
  - 对某个非叠加分片插入数据。
  - 获取所有列信息
  - 获取制定前缀的数据边界信息
  - 释放底层链接
- IGinX 的其他功能是可选的，如果有相关需求的话需要支持，否则无需支持：
- 询问底层是否支持带下推的查询。

- 对分片（叠加/非叠加）带谓词下推的查询。

## 13.2 接口

扩展数据库需要实现以下两类接口：

- IStorageEngine：接口名称与含义的对应关系如表 13.1 所示、

表 13.1

名称	含义
executeProject	对非叠加分片查询数据
executeProjectDummy	对叠加分片查询数据
isSupportProjectWithSelect	询问底层是否支持带谓词下推的查询
executeProjectWithSelect	对非叠加分片带谓词下推的查询
executeProjectDummyWithSelect	对叠加分片带谓词下推的查询
executeDelete	对非叠加分片删除数据
executeInsert	对非叠加分片插入数据
getColumns	获取所有列信息
getBoundaryOfStorage	获取指定前缀的数据边界
release	释放底层连接

每个接口的输入参数为对应类型的计划，输出参数为相应的执行结果。其功能是将计划转换为待扩展数据库可用的数据结构，包装后将请求发送到给定的数据后端，再解析得到的结果，按照不同类型的执行结果进行封装。这样一来便可完成 IGINX 与底层数据库功能的对接。

## 13.3 异构数据库部署操作

### 1. 启动 InfluxDB2.0

在 influxdbd 同一目录（如/tpc/influxdb2.0.4）下，创建配置文件 config.yaml，内容如下，则可使数据放在以下目录中：

- engine-path: /tpc/influxdb2.0.4/data/engine  
其余配置在用户目录下的.influxdbv2 目录中，其它相关配置项见：  
<https://docs.influxdata.com/influxdb/v2.0/reference/config-options>  
启动 InfluxDB 命令：

```
./influxd
```

启动后，操作获得用于 I GinX 配置的 token 和 organization，命令如下：

```
./influx setup --username root --password root1234 --org THUIGinX --bucket I GinX --token I GinX-token-for-you-best-way-ever --force
```

## 2. 配置项

下面以配置 2 个数据库，1 个为 IoTDB，1 个为 InfluxDB 为例，说明主要相关配置项：

```
storageEngineList=192.168.10.44#6667#iotdb#username=root#password=root#sessionPoolSize=100,192.168.10.45#8086#influxdb#url=http://192.168.10.45:8086/#token=your-token#organization=your-organization
```

## 3. 启动

按正常过程启动系统即可。

# 13.4 同数据库多版本实现

- 不能重名，可采用以下命名方式：如 IoTDB0.11.4 与 0.12.6 版本，可分别实现接口并命名为 IoTDB11 和 IoTDB12。
- 构建单独的模块，撰写单独的 POM，进行实现，具体可参考已有模块。

## 十四、导入导出 CSV 工具

可从下面这个仓库获取这些工具

<https://github.com/IGinX-THU/MiscTools>

### 14.1 导出 CSV 工具

#### 14.1.1 运行方法

- Unix / OS X 系统

```
> tools/export_csv.sh -h <host> -p <port> -u <username> -pw <password> -d <directory> -q <query statement> -s <sql file> -tf <time format> -tp <time precision>
```

- Windows 系统

```
> tools\export_csv.bat -h <host> -p <port> -u <username> -pw <password> -d <directory> -q <query statement> -s <sql file> -tf <time format> -tp <time precision>
```

参数说明:

- -h <host>
  - ◆ 可选项。表示 IP 地址。默认为 127.0.0.1。
- -p <port>
  - ◆ 可选项。表示端口号。默认为 6888。
- -u <username>
  - ◆ 可选项。表示用户名。默认为 root。
- -pw <password>
  - ◆ 可选项。表示密码。默认为 root。
- -d <directory>
  - ◆ 可选项。表示存储导出 CSV 文件的目录。默认为当前工作目录。
  - ◆ 建议手动设置该参数，且如果导出文件过多，推荐将该参数设置为一个干净的空目录。
- -q <query statement>
  - ◆ 可选项。表示需要导出的数据所对应的查询语句。
  - ◆ 支持同时指定多条查询语句，以“;”进行分割。
  - ◆ 例如: select \* from \*; select \* from sg.d1; select \* from sg.d2;
- -s <sql file>
  - ◆ 可选项。指定一个 SQL 文件，里面包含一条或多条 SQL 语句。如果一个 SQL 文件中包含多条 SQL 语句，SQL 语句之间应该用换行符进行分割。每一条 SQL 语句对应一个输出的 CSV 文件。
  - ◆ 虽然-q 和-s 均为可选项，但这二者必须指定且只能指定一个，即同时指定-q 和-s 或者同时不指定-q 和-s 都是不合法的输入。
- -tf <time format>
  - ◆ 可选项。表示导出 CSV 文件的时间格式。默认为数值型的时间戳。
- -tp <time precision>
  - ◆ 可选项。表示导出 CSV 文件的时间精度。默认为毫秒，还可设置为秒、微

秒和纳秒。

## 14.1.2 运行示例

### ● Unix / OS X 系统

```
> tools/export_csv.sh -h 127.0.0.1 -p 6667 -u root -pw root -q "select * from *;"

> tools/export_csv.sh -h 127.0.0.1 -p 6667 -u root -pw root -d /Users/XXX/export/ -q "select
* from *;"

> tools/export_csv.sh -h 127.0.0.1 -p 6667 -u root -pw root -d /Users/XXX/export/ -q "select
* from sg.d1; select * from sg.d2;"

> tools/export_csv.sh -h 127.0.0.1 -p 6667 -u root -pw root -s /Users/XXX/sql.txt

> tools/export_csv.sh -h 127.0.0.1 -p 6667 -u root -pw root -d /Users/XXX/export/ -s
/Users/XXX/sql.txt

> tools/export_csv.sh -h 127.0.0.1 -p 6667 -u root -pw root -d /Users/XXX/export/ -s
/Users/XXX/sql.txt -tf "yyyy-MM-dd'T'HH:mm:ss.SSS"

> tools/export_csv.sh -h 127.0.0.1 -p 6667 -u root -pw root -d /Users/XXX/export/ -s
/Users/XXX/sql.txt -tf "yyyy-MM-dd'T'HH:mm:ss.SSS" -tp ms
```

### ● Windows 系统

```
> tools\export_csv.bat -h 127.0.0.1 -p 6667 -u root -pw root -q "select * from *;"

> tools\export_csv.bat -h 127.0.0.1 -p 6667 -u root -pw root -d C:\Users\XXX\export\ -q
"select * from *;"

> tools\export_csv.bat -h 127.0.0.1 -p 6667 -u root -pw root -d C:\Users\XXX\export\ -q
"select * from sg.d1; select * from sg.d2;"

> tools\export_csv.bat -h 127.0.0.1 -p 6667 -u root -pw root -s C:\Users\XXX\sql.txt

> tools\export_csv.bat -h 127.0.0.1 -p 6667 -u root -pw root -d C:\Users\XXX\export\ -s
C:\Users\XXX\sql.txt

> tools\export_csv.bat -h 127.0.0.1 -p 6667 -u root -pw root -d C:\Users\XXX\export\ -s
C:\Users\XXX\sql.txt -tf "yyyy-MM-dd'T'HH:mm:ss.SSS"

> tools\export_csv.bat -h 127.0.0.1 -p 6667 -u root -pw root -d C:\Users\XXX\export\ -s
C:\Users\XXX\sql.txt -tf "yyyy-MM-dd'T'HH:mm:ss.SSS" -tp ms
```

## 14.1.3 注意事项



1. 如果在-q 参数中指定多条查询语句，或者在-s 指定的 SQL 文件中写入多条查询语句，那么每一条查询语句都会相应地导出一个 CSV 文件，记作 export{0..n}.csv。例如：如果指定三条查询语句，那么系统会生成 export0.csv，export1.csv 和 export2.csv 三个 CSV 文件。
2. 导出的 CSV 文件的 header 的第一个字段名称为“Key”。
3. -tf 和-tp 须保持一致，且-tp 须与想要导出数据的精度保持一致。例如：如果想要导出数据的精度为秒，那么-tf 只能设置为"yyyy-MM-dd'T'HH:mm:ss"等精确到秒的时间格式，且-tp 必须设置为 s。
4. 空值统一记作“null”。

## 14.2 导入 CSV 工具

### 14.2.1 运行方法

#### ● Unix / OS X 系统

```
> tools/import_csv.sh -h <ip> -p <port> -u <username> -pw <password> -d <directory> -f <file> -tf <time format>
```

#### ● Windows 系统

```
> tools\import_csv.bat -h <ip> -p <port> -u <username> -pw <password> -d <directory> -f <file> -tf <time format>
```

参数说明：

- -h <host>
  - ◆ 可选项。表示 IP 地址。默认为 127.0.0.1。
- -p <port>
  - ◆ 可选项。表示端口号。默认为 6888。
- -u <username>
  - ◆ 可选项。表示用户名。默认为 root。
- -pw <password>
  - ◆ 可选项。表示密码。默认为 root。
- -f
  - ◆ 可选项。表示需要导入的 CSV 文件。
  - ◆ 例如：-f import.csv
- -d
  - ◆ 可选项。表示需要导入的目录。
  - ◆ 例如：-d /Users/XXX/import/
  - ◆ 虽然-f 和-d 均为可选项，但这二者必须指定且只能指定一个，即同时指定-f 和-d 或者同时不指定-f 和-d 都是不合法的输入。
- -tf <time format>
  - ◆ 可选项。表示导入 CSV 文件的时间格式。默认为数值型的时间戳。

### 14.2.2 运行示例

#### ● Unix / OS X 系统

```
> tools/import_csv.sh -h 127.0.0.1 -p 6667 -u root -pw root -f import.csv  
> tools/import_csv.sh -h 127.0.0.1 -p 6667 -u root -pw root -d /Users/XXX/import/  
> tools/import_csv.sh -h 127.0.0.1 -p 6667 -u root -pw root -d /Users/XXX/import/ -tf "yyyy-MM-dd'T'HH:mm:ss.SSS"
```

● Windows 系统

```
> tools\import_csv.bat -h 127.0.0.1 -p 6667 -u root -pw root -f import.csv  
> tools\import_csv.bat -h 127.0.0.1 -p 6667 -u root -pw root -d C:\Users\XXX\import\  
> tools\import_csv.bat -h 127.0.0.1 -p 6667 -u root -pw root -d C:\Users\XXX\import\ -tf "yyyy-MM-dd'T'HH:mm:ss.SSS"
```

### 14.2.3 注意事项

1. 对于-d 参数，系统只会导入指定目录下以“.csv”为后缀的文件。因此，如果需要导入多个 CSV 文件，推荐将所有文件放到一个干净的空目录下。
2. 导入的 CSV 文件的 header 的第一个字段必须为“Key”。
3. 系统会根据导入的每一列的第一个不为“null”的值自动推断该列的数据类型。如果某列所有值均为“null”，那么会将其推断为字符串类型。

## 十五、曲线匹配功能

### 15.1 原理简介

IGinX 的曲线匹配功能基于 DTW (Dynamic Time Warping, 动态时间规整) 算法实现, 支持基本的 DTW 算法以及以形状距离代替欧式距离的 DTW 算法。输入为固定长度的序列 Q 以及该序列数据点之间的时间间隔, 支持查询给定序列集和时间范围内所有符合条件的子序列中和 Q 最相似的数据段。

DTW 算法是时间序列相似性度量领域的最经典算法, 它解决了欧式距离作为度量指标时难以处理形状相似性的问题。欧式距离的计算方法是将两条时间序列的所有点一一对应; 而 DTW 距离的计算方法是找到两条时间序列之间对应的点之后再计算它们的距离。因此, DTW 的根本任务就是正确地匹配两条时间序列, 而且如果两条时间序列的点正确匹配了, 那么它们之间的距离达到最小。

DTW 算法的时间复杂度比简单的欧式距离算法高。为了优化算法, 降低复杂度, IGINX 内部实现了几种常见的 DTW 优化加速策略, 包括增加匹配阈值、替换距离计算方法、通过早弃策略剪枝等。

### 15.2 使用说明

曲线匹配算法目前支持使用原生 Session 接口进行调用, 具体介绍如下:

#### 15.2.1 接口名称

```
Pair<String matchedPath, Long matchedTimestamp>curveMatch(List<String> paths, long startTime, long endTime, List<Double> curveQuery, long curveUnit)
```

#### 15.2.2 参数介绍

##### 1. 输入参数介绍

- **paths:** 类型为字符串列表。含义为需匹配的时间序列集合。
- **startTime:** 类型为长整型。含义为需匹配的时间范围的起始时间戳。
- **endTime:** 类型为长整型。含义为需匹配的时间范围的结束时间戳。
- **curveQuery:** 类型为浮点数列表。含义为待匹配的数据段。
- **curveUnit:** 类型为长整型。含义为待匹配的数据段的时间间隔。

##### 2. 输出参数介绍

- **matchedPath:** 类型为字符串。含义为匹配到的最接近的时间序列。
- **matchedTimestamp:** 类型为长整型。含义为匹配到的最接近的时间范围的起始时间戳。

#### 15.2.3 应用示例

给定需匹配数据段 1,3,5,2,4,6 以及时间间隔 5ms, 即该时间段内任意两个相邻数据点的时间间隔均为 5ms。在时间区间为[1000ms,5000ms], 时间序列集合为{s1,s2,s3,s4}

的待匹配范围内，匹配与给定数据段最相似的数据段。假设返回结果为时间序列  $s_2$ ，时间戳  $2000\text{ms}$ ，则与给定数据段最匹配的数据段出现在时间序列  $s_2$  中，且起始时间戳为  $2000\text{ms}$ ，长度与给定数据段相同。

在上例中， $paths$  为  $\{s_1, s_2, s_3, s_4\}$ ， $startTime$  为  $1000\text{ms}$ ， $endTime$  为  $5000\text{ms}$ ， $curveQuery$  为  $1, 3, 5, 2, 4, 6$ ， $curveUnit$  为  $5\text{ms}$ ， $matchedPath$  为  $s_2$ ， $matchedTimestamp$  为  $2000\text{ms}$ 。

## 十六、部署指导原则

以下不同的部署模式，应当搭配不同的数据分布策略。目前，默认可选的策略包括以下三种：

- [1] `policyClassName=cn.edu.tsinghua.IGinX.policy.naive.NaivePolicy`
- [2] `policyClassName=cn.edu.tsinghua.IGinX.policy.simple.SimplePolicy`
- [3] `policyClassName=cn.edu.tsinghua.IGinX.policy.historical.HistoricalPolicy`

### 16.1 边缘端部署原则

一般的边缘端数据管理场景，要确保数据可靠性，可以通过 2 副本 2 节点的时序数据库实例部署来实现。

如果应用连接数较高，可以启动多个 IGINX；否则，可以仅启动 1 个 IGINX。

### 16.2 云端部署原则

在云端单数据中心场景，可通过 3 副本多节点的时序数据库实例部署来实现。如果应用连接数较高，可以启动多个 IGINX；否则，可以仅启动 1 个 IGINX。

在云端多数据中心场景，可通过跨数据中心 3 副本多节点的时序数据库实例部署来实现。如果应用连接数较高，可以在每个数据中心启动多个 IGINX；否则，可以在每个数据中心仅启动 1 个 IGINX。

## 十七、常见问题

### 17.1 如何知道当前有哪些 IGINX 节点？

在相应的 ZooKeeper 客户端中直接执行查询。我们需要执行以下步骤：

1. 进入 ZooKeeper 客户端：首先进入 `apache-zookeeper-x.x.x/bin` 文件夹，之后执行命令启动客户端 `./zkCli.sh`。
2. 执行查询命令查看包含哪些 IGINX 节点：`ls /IGINX`，返回结果为形如 `[node0000000000,node0000000001]` 的节点列表。
3. 查看某一个 IGINX 节点的具体信息：如需要查询（2）中对应 `node0000000000` 节点具体信息，则需要执行命令：`get /IGINX/node0000000000` 得到 `node0000000000` 节点具体信息，返回结果为形如 `{"id":0,"ip":"0.0.0.0","port":6324}` 的字典形式，参数分别代表节点在 ZooKeeper 中对应的 ID，IGINX 节点自身的 IP 和端口号。

### 17.2 如何知道当前有哪些时序数据库节点？

在相应的 ZooKeeper 客户端中直接执行查询。我们需要执行以下步骤：

1. 进入 ZooKeeper 客户端：首先进入 `apache-zookeeper-x.x.x/bin` 文件夹，之后执行命令启动客户端 `./zkCli.sh`。
2. 执行查询命令查看包含哪些时序数据库节点：`ls /storage`，返回结果为形如 `[node0000000000]` 的节点列表。
3. 查看某一个时序数据库节点的具体信息：如需要查询（2）中对应 `node0000000000` 节点具体信息，则需要执行命令：`get /storage/node0000000000` 得到 `node0000000000` 节点的信息，返回结果为形如 `{"id":0,"ip":"127.0.0.1","port":6667,"extraParams":{"password":"root","sessionPoolSize":100,"username":"root"},"storageEngine":"IoTDB"}` 的字典形式，参数按照顺序分别代表节点在 ZooKeeper 中对应的 ID，时序数据库节点自身的 IP 和端口号，以及额外的启动参数，与该数据库节点的数据库类型。

### 17.3 如何知道数据分片当前有几个副本？

在相应的 ZooKeeper 客户端中执行查询。

需要了解的是，分片在 IGINX 存储的格式如图 17.1 所示：

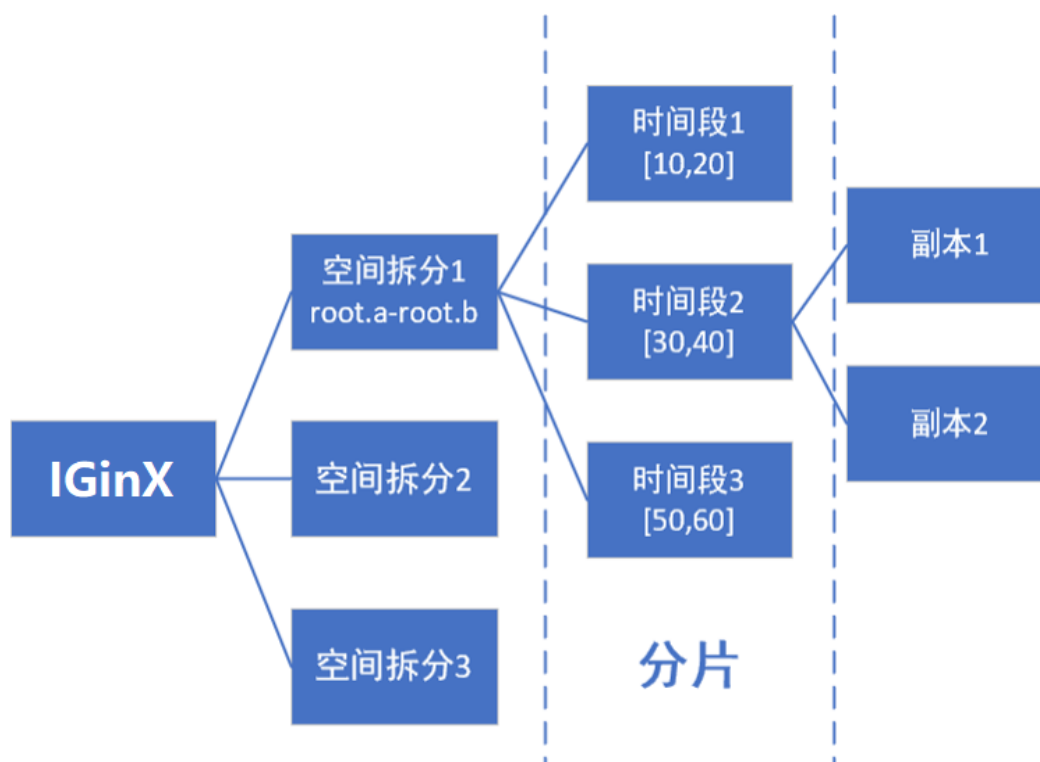


图 17.1 数据分片空间

需要先通过对应的空间拆分和时间拆分确定需求的分片，然后即可查询该分片的具体信息。

我们需要执行以下步骤：

1. 进入 ZooKeeper 客户端：首先进入 `apache-zookeeper-x.x.x/bin` 文件夹，之后执行命令启动客户端 `./zkCli.sh`。
2. 执行查询命令查看 IGINX 目前包含哪些空间拆分：`ls /fragment`，得到的结果为形如 `[null-root.sg1.d1.s1, root.sg1.d3.s1-null, root.sg1.d1.s1-null, null-root.sg1.d3.s1]` 的空间拆分列表。
3. 查看该空间拆分下的分片情况：如需要查询（2）中对应 `null-root.sg1.d1.s1` 这一空间拆分具体信息，则需要执行命令：`ls /fragment/null-root.sg1.d1.s1`，返回结果为形如 `[0, 100, 1000]` 的列表形式，其中每一个参数表示有一个分片以该时间作为起始时间。如 `[0, 100, 1000]` 的列表表示该空间拆分下包含 3 个分片，其分片的最小时间戳分别为 0，100 和 1000。
4. 查询某一个分片的具体信息。在前三步中我们确定了该分片在结构中的对应位置，如需要查询 `null-root.sg1.d1.s1` 空间拆分下，起始时间为 0 的分片的具体信息，则需要执行命令：`get /fragment/null-root.sg1.d1.s1/0`，返回结果为形如 `{"timeInterval":{"startTime":0,"endTime":99},"tsInterval":{"endTimeSeries":"root.sg1.d1.s1"},"replicaMetas":{"0":{"timeInterval":{"startTime":0,"endTime":9223372036854775807},"tsInterval":{"endTimeSeries":"root.sg1.d1.s1"},"replicaIndex":0,"storageEngineId":0},"createdBy":0,"updatedBy":0}` 的字典形式。其中，`replicaMetas` 参数表示存储该分片上所有副本元信息的字典，其元素个数即为该分片的副本个数。其他参数含

义如下：

- (1) **timeInterval**：表示这一分片的起始和终止时间。
- (2) **tsInterval**：表示这一分片的起始和终止时间序列（可为空）。
- (3) **replicaMetas** 中每一个元素的键表示副本的序号，值包含该副本数据起始终止时间、起始终止时间序列、对应时序数据库节点等信息
- (4) **createdBy**：表示创建该分片的 IGINX 编号。
- (5) **updatedBy**：表示最近更新该分片的 IGINX 编号。

## 17.4 如何加入 IGINX 的开发，成为 IGINX 代码贡献者？

在 IGINX 的开源项目地址上 <https://github.com/thulab/IGINX> 提 Issue、提 PR，IGINX 项目核心成员将对代码进行审核后，合并进代码主分支中。

IGINX 当前版本在写入和查询功能方面，支持还不丰富，只有写入、范围查询和整体聚合查询。因此，非常欢迎喜欢 IGINX 的开发者为 IGINX 完成相关功能的开发。

## 17.5 IGINX 集群版管理时序数据与 IoTDB-Raft 版相比，各自特色在何处？

当前的 IGINX 集群版管理时序数据与 IoTDB-Raft 版特性对比如表 17.1 所示：

表 17.1 在管理时序数据方面，2 者的差异

特性\系统	IoTDB-Raft 版	IGINX 集群版
平滑可扩展性	无	有
异构数据库支持	无	有
存算分层扩展性	无	有
分布式一致性维护代价	有	无
灵活分片	无	有
灵活副本策略	无	有
对等强一致性	有	无
部署组件	同构	异构