InfluxDB 1.3.x 中文文档

InfluxDB是一个用于存储和分析时间序列数据的开源数据库。主要特性有：内置HTTP接口，使用方便；数据可以打标记，这样查询可以很灵活；类SQL的查询语句；安装管理很简单，并且读写数据很高效；能够实时查...

目　录

致谢

当前文档 《InfluxDB 1.3.x 中文文档》 由 进击的皇虫 使用 书栈网(BookStack.CN) 进行构建，生成于 2020-02-20。

书栈网仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈网难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到书栈网，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到书栈网获取最新的文档，以跟上知识更新换代的步伐。

内容来源：jasper-zhang https://github.com/jasper-zhang/influxdb-document-cn

文档地址：http://www.bookstack.cn/books/jasper-zhang-influxdb-document-cn

书栈官网：https://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

InfluxDB中文文档

InfluxDB是一个用于存储和分析时间序列数据的开源数据库。

主要特性有:

- 内置HTTP接口，使用方便
- 数据可以打标记，这样查询可以很灵活
- 类SQL的查询语句
- 安装管理很简单，并且读写数据很高效
- 能够实时查询，数据在写入时被索引后就能够被立即查出
- ……

在最新的DB-ENGINES给出的时间序列数据库的排名中，InfluxDB高居第一位，可以预见，InfluxDB会越来越得到广泛的使用。

看了下国内鲜有这方面的教程，所以我就抽空翻译了一下官方的手册，官方文档的开源地址请见https://github.com/influxdata/docs.influxdata.com中的InfluxDB部分。

现在还在不断完善中，欢迎加入来贡献你的一份力量。

如果感觉对你有所帮助，请给个**star** ！

文档项目地址(Github): https://github.com/jasper-zhang/influxdb-document-cn

文档在线地址(Gitbook): https://jasper-zhang1.gitbooks.io/influxdb/content/

作者博客: http://www.opscoder.info

另外，文档基于*InfluxDB*版本*1.3.x*，如果后续版本有较大的改动，此文档也会跟进。

最后，希望您阅读愉快，并有所收获，O(∩_∩)O谢谢！

介绍

这篇介绍文档包括获取、运行InfluxDB的信息。

下载

提供了最新稳定版和其他版本的InfluxDB下载地址。

安装

介绍在Ubuntu、Debian、Redhat、Centos和OS X上如何安装InfluxDB。

入门指南

介绍利用InfluxDB怎样读写时序数据。

安装

这篇将会介绍怎么安装、运行和配置InfluxDB。

准备

安装InfluxDB包需要 `root` 或是有管理员权限才可以。

网络

InfluxDB默认使用下面的网络端口：

- TCP端口 `8086` 用作InfluxDB的客户端和服务端的http api通信
- TCP端口 `8088` 给备份和恢复数据的RPC服务使用

另外，InfluxDB也提供了多个可能需要自定义端口的插件，所以的端口映射都可以通过配置文件修改，对于默认安装的InfluxDB，这个配置文件位于 `/etc/influxdb/influxdb.conf` 。

NTP

InfluxDB使用服务器本地时间给数据加时间戳，而且是UTC时区的。并使用NTP来同步服务器之间的时间，如果服务器的时钟没有通过NTP同步，那么写入InfluxDB的数据的时间戳就可能不准确。

安装

对于不想安装的用户，可以使用inluxdata公司提供的云产品（帮忙给他们打个广告吧，毕竟开源不易）。

Debain & Ubuntu

Debian和Ubuntu用户可以直接用 `apt-get` 包管理来安装最新版本的InfluxDB。

对于Ubuntu用户，可以用下面的命令添加InfluxDB的仓库

```
curl -sL https://repos.influxdata.com/influxdb.key | sudo apt-key add -
source /etc/lsb-release
echo "deb https://repos.influxdata.com/${DISTRIB_ID,,} ${DISTRIB_CODENAME}
stable" | sudo tee /etc/apt/sources.list.d/influxdb.list
```

Debian用户用下面的命令：

```
curl -sL https://repos.influxdata.com/influxdb.key | sudo apt-key add -
source /etc/os-release
test $VERSION_ID = "7" && echo "deb https://repos.influxdata.com/debian wheezy stable" | sudo tee /etc/apt/sources.list.d/influxdb.list
test $VERSION_ID = "8" && echo "deb https://repos.influxdata.com/debian jessie stable" | sudo tee /etc/apt/sources.list.d/influxdb.list
```

然后安装、运行InfluxDB服务：

```
sudo apt-get update && sudo apt-get install influxdb
sudo service influxdb start
```

如果你的系统可以使用Systemd(比如Ubuntu 15.04+, Debian 8+)，也可以这样启动：

```
sudo apt-get update && sudo apt-get install influxdb
sudo systemctl start influxdb
```

RedHat & CentOS

RedHat和CentOS用户可以直接用 `yum` 包管理来安装最新版本的InfluxDB。

```
cat <<EOF | sudo tee /etc/yum.repos.d/influxdb.repo
[influxdb]
name = InfluxDB Repository - RHEL \$releasever
baseurl = https://repos.influxdata.com/rhel/\$releasever/\$basearch/stable
enabled = 1
gpgcheck = 1
gpgkey = https://repos.influxdata.com/influxdb.key
EOF
```

一旦加到了 `yum` 源里面，就可以运行下面的命令来安装和启动InfluxDB服务：

```
sudo yum install influxdb
sudo service influxdb start
```

如果你的系统可以使用Systemd(比如CentOS 7+, RHEL 7+)，也可以这样启动：

```
sudo yum install influxdb
sudo systemctl start influxdb
```

MAC OS X

OS X 10.8或者更高版本的用户，可以使用Homebrew来安装InfluxDB；一旦 `brew` 安装了，可以用下面的命令来安装InfluxDB:

```
1. brew update
2. brew install influxdb
```

登陆后在用 `launchd` 开始运行InfluxDB之前，先跑:

```
1. ln -sfv /usr/local/opt/influxdb/*.plist ~/Library/LaunchAgents
```

然后运行InfluxDB:

```
1. launchctl load ~/Library/LaunchAgents/homebrew.mxcl.influxdb.plist
```

如果你不想用或是不需要launchctl，你可以直接在terminal里运行下面命令来启动InfluxDB:

```
1. influxd -config /usr/local/etc/influxdb.conf
```

配置

安装好之后，每个配置文件都有了默认的配置，你可以通过命令 `influxd config` 来查看这些默认配置。

在配置文件 `/etc/influxdb/influxdb.conf` 之中的大部分配置都被注释掉了，所有这些被注释掉的配置都是由内部默认值决定的。配置文件里任意没有注释的配置都可以用来覆盖内部默认值，需要注意的是，本地配置文件不需要包括每一项配置。

有两种方法可以用自定义的配置文件来运行InfluxDB:

- 运行的时候通过可选参数 `-config` 来指定:

```
1. influxd -config /etc/influxdb/influxdb.conf
```

- 设置环境变量 `INFLUXDB_CONFIG_PATH` 来指定，例如:

```
1. echo $INFLUXDB_CONFIG_PATH
2. /etc/influxdb/influxdb.conf
3.
4.
```

```
influxd
```

其中 `-config` 的优先级高于环境变量。

想看更详细的配置解释，请移步配置文档。

托管在AWS上

硬件

我们建议使用两块SSD卷，一个是为了 `influxdb/wal` ，一个是为了 `influxdb/data` ；根据您的负载量，每个卷应具有大约1k-3k的IOPS。 `influxdb/data` 卷应该有更多的磁盘空间和较低的IOPS，而 `influxdb/wal` 卷则相反有较少的磁盘空间但是较高的IOPS。

每台机器应该有不少于8G的内存。

在AWS上的R4型号的机器上的我们看到了最好的性能，因为这种型号的机器提供的内存比C3/C4和M4型号机器大得多。

配置aws实例

这个例子假定你正在使用两个SSD卷，并且已正确安装它们。此示例还假定每个卷都安装在 `/mnt/influx` 和 `/mnt/db` 上。如何安装请看亚马逊提供的文档给你的aws实例添加卷。

配置文件

你需要修改每一个InfluxDB实例的配置文件：

```
...

[meta]
  dir = "/mnt/db/meta"
  ...

...

[data]
  dir = "/mnt/db/data"
  ...
wal-dir = "/mnt/influx/wal"
  ...

```

```
...

[hinted-handoff]
    ...
dir = "/mnt/db/hh"
    ...
```

权限

如果InfluxDB没有使用标准的数据和配置文件的文件夹的话，你需要确定文件系统的权限是正确的：

```
chown influxdb:influxdb /mnt/influx
chown influxdb:influxdb /mnt/db
```

入门指南

InfluxDB安装完成之后，我们开始来做一些有意思的事。在这一章里面我们将会用到 `influx` 这个命令行工具，这个工具包含在InfluxDB的安装包里，是一个操作数据库的轻量级工具。它直接通过InfluxDB的HTTP接口(如果没有修改，默认是8086)来和InfluxDB通信。

> 说明：也可以直接发送裸的HTTP请求来操作数据库，例如 `curl`

创建数据库

如果你已经在本地安装运行了InfluxDB，你就可以直接使用 `influx` 命令行，执行 `influx` 连接到本地的InfluxDB实例上。输出就像下面这样：

```
1. $ influx -precision rfc3339
2. Connected to http://localhost:8086 version 1.2.x
3. InfluxDB shell 1.2.x
4. >
```

> 说明：
>
> - InfluxDB的HTTP接口默认起在 `8086` 上，所以 `influx` 默认也是连的本地的 `8086` 端口，你可以通过 `influx --help` 来看怎么修改默认值。
> - `-precision` 参数表明了任何返回的时间戳的格式和精度，在上面的例子里， `rfc3339` 是让InfluxDB返回RFC339格式(YYYY-MM-DDTHH:MM:SS.nnnnnnnnnZ)的时间戳。

这样这个命令行已经准备好接收influx的查询语句了(简称InfluxQL)，用 `exit` 可以退出命令行。

第一次安装好InfluxDB之后是没有数据库的(除了系统自带的 `_internal`)，因此创建一个数据库是我们首先要做的事，通过 `CREATE DATABASE <db-name>` 这样的InfluxQL语句来创建，其中 `<db-name>` 就是数据库的名字。数据库的名字可以是被双引号引起来的任意Unicode字符。 如果名称只包含ASCII字母，数字或下划线，并且不以数字开头，那么也可以不用引起来。

我们来创建一个 `mydb` 数据库：

```
1. > CREATE DATABASE mydb
2. >
```

> 说明：在输入上面的语句之后，并没有看到任何信息，这在CLI里，表示语句被执行并且没有错误，如果有错误信息展示，那一定是哪里出问题了，这就是所谓的 没有消息就是好消息 。

现在数据库 `mydb` 已经创建好了，我们可以用 `SHOW DATABASES` 语句来看看已存在的数据库：

```
> SHOW DATABASES
name: databases
---------------
name
_internal
mydb

>
```

> 说明： `_internal` 数据库是用来存储InfluxDB内部的实时监控数据的。

不像 `SHOW DATABASES` ，大部分InfluxQL需要作用在一个特定的数据库上。你当然可以在每一个查询语句上带上你想查的数据库的名字，但是CLI提供了一个更为方便的方式 `USE <db-name>` ，这会为你后面的所以的请求设置到这个数据库上。例如：

```
> USE mydb
Using database mydb
>
```

以下的操作都作用于 `mydb` 这个数据库之上。

读写数据

现在我们已经有了一个数据库，那么InfluxDB就可以开始接收读写了。

首先对数据存储的格式来个入门介绍。InfluxDB里存储的数据被称为 `时间序列数据` ，其包含一个数值，就像CPU的load值或是温度值类似的。时序数据有零个或多个数据点，每一个都是一个指标值。数据点包括 `time` (一个时间戳)， `measurement` (例如cpu_load)，至少一个k-v格式的 `field` (也即指标的数值例如 “value=0.64”或者“temperature=21.2”)，零个或多个 `tag` ，其一般是对于这个指标值的元数据(例如“host=server01”, “region=EMEA”, “dc=Frankfurt)。

在概念上，你可以将 `measurement` 类比于SQL里面的table，其主键索引总是时间戳。 `tag` 和 `field` 是在table里的其他列， `tag` 是被索引起来的， `field` 没有。不同之处在于，在InfluxDB里，你可以有几百万的measurements，你不用事先定义数据的scheme，并且null值不会被存储。

将数据点写入InfluxDB，只需要遵守如下的行协议：

```
<measurement>[,<tag-key>=<tag-value>...] <field-key>=<field-value>[,<field2-key>=<field2-value>...] [unix-nano-timestamp]
```

下面是数据写入InfluxDB的格式示例：

```
cpu,host=serverA,region=us_west value=0.64
payment,device=mobile,product=Notepad,method=credit billed=33,licenses=3i 1434067467100293230
stock,symbol=AAPL bid=127.46,ask=127.48
temperature,machine=unit42,type=assembly external=25,internal=37 1434067467000000000
```

> 说明：关于写入格式的更多语法，请参考写入语法这一章。

使用CLI插入单条的时间序列数据到InfluxDB中，用 `INSERT` 后跟数据点：

```
> INSERT cpu,host=serverA,region=us_west value=0.64
>
```

这样一个measurement为 `cpu` ，tag是 `host` 和 `region` ， `value` 值为 `0.64` 的数据点被写入了InfluxDB中。

现在我们查出写入的这笔数据：

```
> SELECT "host", "region", "value" FROM "cpu"
name: cpu
---------
time                                host         region   value
2015-10-21T19:28:07.580664347Z  serverA      us_west     0.64

>
```

> 说明：我们在写入的时候没有包含时间戳，当没有带时间戳的时候，InfluxDB会自动添加本地的当前时间作为它的时间戳。

让我们来写入另一笔数据，它包含有两个字段：

```
> INSERT temperature,machine=unit42,type=assembly external=25,internal=37
>
```

查询的时候想要返回所有的字段和tag，可以用 `*` ：

```
1.
2.  > SELECT * FROM "temperature"
3.  name: temperature
4.  -----------------
    time                                    external      internal      machine
5.  type
    2015-10-21T19:28:08.385013942Z  25                37             unit42
6.  assembly
7.
8.  >
```

InfluxQL还有很多特性和用法没有被提及，包括支持golang样式的正则，例如：

```
1.  > SELECT * FROM /.*/ LIMIT 1
2.  --
3.  > SELECT * FROM "cpu_load_short"
4.  --
5.  > SELECT * FROM "cpu_load_short" WHERE "value" > 0.9
```

这就是你需要知道的读写InfluxDB的方法，想要学习更多的写的知识请移步写数据章节，读的知识请到读数据章节。要获取更多InfluxDB的概念的信息，请查看关键概念章节。

使用指南

写入数据

有很多可以向InfluxDB写数据的方式，包括命令行、客户端还有一些像 `Graphite` 有一样数据格式的插件。这篇文章将会展示怎样创建数据库，并使用内建的HTTP接口写入数据。

使用HTTP接口创建数据库

使用 `POST` 方式发送到URL的 `/query` 路径，参数 `q` 为 `CREATE DATABASE <new_database_name>` ，下面的例子发送一个请求到本地运行的InfluxDB创建数据库 `mydb` ：

```
curl -i -XPOST http://localhost:8086/query --data-urlencode "q=CREATE DATABASE
mydb"
```

使用HTTP接口写数据

通过HTTP接口 `POST` 数据到 `/write` 路径是我们往InfluxDB写数据的主要方式。下面的例子写了一条数据到 `mydb` 数据库。这条数据的组成部分是measurement为 `cpu_load_short` ，tag的key为host和region，对应tag的value是 `server01` 和 `us-west` ，field的key是 `value` ，对应的数值为 `0.64` ，而时间戳是 `1434055562000000000` 。

```
curl -i -XPOST 'http://localhost:8086/write?db=mydb' --data-binary
'cpu_load_short,host=server01,region=us-west value=0.64 1434055562000000000'
```

当写入这条数据点的时候，你必须明确存在一个数据库对应名字是 `db` 参数的值。如果你没有通过 `rp` 参数设置retention policy的话，那么这个数据会写到 `db` 默认的retention policy中。想要获取更多参数的完整信息，请移步到 `API参考` 章节。

POST的请求体我们称之为 `Line Protocol` ，包含了你希望存储的时间序列数据。它的组成部分有measurement，tags，fields和timestamp。measurement是InfluxDB必须的，严格地说，tags是可选的，但是对于大部分数据都会包含tags用来区分数据的来源，让查询变得容易和高效。tag的key和value都必须是字符串。fields的key也是必须的，而且是字符串，默认情况下field的value是float类型的。timestamp在这个请求行的最后，是一个从1/1/1970 UTC开始到现在的一个纳秒级的Unix time，它是可选的，如果不传，InfluxDB会使用服务器的本地的纳米级的timestamp来作为数据的时间戳，注意无论哪种方式，在InfluxDB中的timestamp只能是UTC时间。

同时写入多个点

要想同时发送多个数据点到多个series(在InfluxDB中measurement加tags组成了一个series)，

可以用新的行来分开这些数据点。这种批量发送的方式可以获得更高的性能。

下面的例子就是写了三个数据点到 `mydb` 数据库中。第一个点所属series的measurement为 `cpu_load_short` ，tag是 `host=server02` ，timestamp是server本地的时间戳；第二个点同样是measurement为 `cpu_load_short` ，但是tag为 `host=server02,region=us-west` ，且有明确timestamp为 `1422568543702900257` 的series；第三个数据点和第二个的timestamp是一样的，但是series不一样，其measurement为 `cpu_load_short` ，tag为 `direction=in,host=server01,region=us-west` 。

```
curl -i -XPOST 'http://localhost:8086/write?db=mydb' --data-binary
'cpu_load_short,host=server02 value=0.67
cpu_load_short,host=server02,region=us-west value=0.55 1422568543702900257
cpu_load_short,direction=in,host=server01,region=us-west value=2.0
1422568543702900257'
```

写入文件中的数据

可以通过 `curl` 的 `@filename` 来写入文件中的数据，且这个文件里的数据的格式需要满足InfluxDB那种行的语法。

给一个正确的文件(cpu_data.txt)的例子：

```
cpu_load_short,host=server02 value=0.67
cpu_load_short,host=server02,region=us-west value=0.55 1422568543702900257
cpu_load_short,direction=in,host=server01,region=us-west value=2.0
1422568543702900257
```

看我们如何把 `cpu_data.txt` 里的数据写入 `mydb` 数据库：

```
curl -i -XPOST 'http://localhost:8086/write?db=mydb' --data-binary
@cpu_data.txt
```

> 说明：如果你的数据文件的数据点大于5000时，你必须把他们拆分到多个文件再写入InfluxDB。因为默认的HTTP的timeout的值为5秒，虽然5秒之后，InfluxDB仍然会试图把这批数据写进去，但是会有数据丢失的风险。

无模式设计

InfluxDB是一个无模式(schemaless)的数据库，你可以在任意时间添加measurement，tags和fields。注意：如果你试图写入一个和之前的类型不一样的数据(例如，filed字段之前接收的是数字类型，现在写了个字符串进去)，那么InfluxDB会拒绝这个数据。

对于REST的一个说明

InfluxDB使用HTTP作为方便和广泛支持的数据传输协议。

现代web的APIs都基于REST的设计，因为这样解决了一个共同的需求。因为随着终端数量的增长，组织系统的需求变得越来越迫切。REST是为了组织大量终端的一个业内认可的标准。这种一致性对于开发者和API的消费者都是一件好事：所有的参与者都知道期望的是什么。

REST的确是很方便的，而InfluxDB也只提供了三个API，这使得InfluxQL在翻译为HTTP请求的时候很简单便捷。 所以InfluxDB API并不是RESTful的。

HTTP返回值概要

- 2xx：如果你写了数据后收到 `HTTP 204 No Content` ，说明写入成功了！
- 4xx：表示InfluxDB不知道你发的是什么鬼。
- 5xx：系统过载或是应用受损。

举几个返回错误的例子：

- 之前接收的是布尔值，现在你写入一个浮点值：

```
curl -i -XPOST 'http://localhost:8086/write?db=hamlet' --data-binary
'tobeornottobe booleanonly=true'

curl -i -XPOST 'http://localhost:8086/write?db=hamlet' --data-binary
'tobeornottobe booleanonly=5'
```

这时系统返回：

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Request-Id: [...]
X-Influxdb-Version: 1.2.x
Date: Wed, 01 Mar 2017 19:38:01 GMT
Content-Length: 150

{"error":"field type conflict: input field \"booleanonly\" on measurement
\"tobeornottobe\" is type float, already exists as type boolean dropped=1"}
```

- 写入数据到一个不存在的数据库中：

```
curl -i -XPOST 'http://localhost:8086/write?db=atlantis' --data-binary 'liters
value=10'
```

返回值：

```
HTTP/1.1 404 Not Found
Content-Type: application/json
Request-Id: [...]
X-Influxdb-Version: 1.2.x
Date: Wed, 01 Mar 2017 19:38:35 GMT
Content-Length: 45

{"error":"database not found: \"atlantis\""}
```

下一步

现在你已经知道了如何通过内置HTTP API写入数据了，下面我们来看看怎么通过读数据指南来读出它们。想要了解更多关于如何使用HTTP API写数据，请参考API参考文档。

查询数据

使用HTTP接口查询数据

HTTP接口是InfluxDB查询数据的主要方式。通过发送一个 `GET` 请求到 `/query` 路径，并设置URL的 `db` 参数为目标数据库，设置URL参数 `q` 为查询语句。下面的例子是查询在写数据里写入的数据点。

```
curl -G 'http://localhost:8086/query?pretty=true' --data-urlencode "db=mydb" --data-urlencode "q=SELECT \"value\" FROM \"cpu_load_short\" WHERE \"region\"='us-west'"
```

InfluxDB返回一个json值，你查询的结果在 `result` 列表中，如果有错误发送，InfluxDB会在 `error` 这个key里解释错误发生的原因。

```
{
    "results": [
        {
            "statement_id": 0,
            "series": [
                {
                    "name": "cpu_load_short",
                    "columns": [
                        "time",
                        "value"
                    ],
                    "values": [
                        [
                            "2015-01-29T21:55:43.702900257Z",
                            2
                        ],
                        [
                            "2015-01-29T21:55:43.702900257Z",
                            0.55
                        ],
                        [
                            "2015-06-11T20:46:02Z",
                            0.64
                        ]
                    ]
```

```
                }
            ]
        }
    ]
}
```

> 说明：添加 `pretty=ture` 参数在URL里面，是为了让返回的json格式化。这在调试或者是直接用 `curl` 的时候很有用，但在生产上不建议使用，因为这样会消耗不必要的网络带宽。

多个查询

在一次API调用中发送多个InfluxDB的查询语句，可以简单地使用分号分隔每个查询，例如：

```
curl -G 'http://localhost:8086/query?pretty=true' --data-urlencode "db=mydb" --data-urlencode "q=SELECT \"value\" FROM \"cpu_load_short\" WHERE \"region\"='us-west';SELECT count(\"value\") FROM \"cpu_load_short\" WHERE \"region\"='us-west'"
```

返回：

```
{
    "results": [
        {
            "statement_id": 0,
            "series": [
                {
                    "name": "cpu_load_short",
                    "columns": [
                        "time",
                        "value"
                    ],
                    "values": [
                        [
                            "2015-01-29T21:55:43.702900257Z",
                            2
                        ],
                        [
                            "2015-01-29T21:55:43.702900257Z",
                            0.55
                        ],
                        [
                            "2015-06-11T20:46:02Z",
```

```
                            0.64
                        ]
                    ]
                }
            ]
        },
        {
            "statement_id": 1,
            "series": [
                {
                    "name": "cpu_load_short",
                    "columns": [
                        "time",
                        "count"
                    ],
                    "values": [
                        [
                            "1970-01-01T00:00:00Z",
                            3
                        ]
                    ]
                }
            ]
        }
    ]
}
```

查询数据时的其他可选参数

时间戳格式

在InfluxDB中的所有数据都是存的UTC时间，时间戳默认返回RFC3339格式的纳米级的UTC时间，例如 `2015-08-04T19:05:14.318570484Z` ，如果你想要返回Unix格式的时间，可以在请求参数里设置 `epoch` 参数，其中epoch可以是 `[h,m,s,ms,u,ns]` 之一。例如返回一个秒级的epoch：

```
curl -G 'http://localhost:8086/query' --data-urlencode "db=mydb" --data-urlencode "epoch=s" --data-urlencode "q=SELECT \"value\" FROM \"cpu_load_short\" WHERE \"region\"='us-west'"
```

认证

InfluxDB里面的认证默认是关闭的，查看认证和鉴权章节了解如何开启认证。

最大行限制

可选参数 `max-row-limit` 允许使用者限制返回结果的数目，以保护InfluxDB不会在聚合结果的时候导致的内存耗尽。

在1.2.0和1.2.1版本中，InfluxDB默认会把返回的数目截断为10000条，如果有超过10000条返回，那么返回体里面会包含一个 `"partial":true` 的标记。该默认设置可能会导致Grafana面板出现意外行为，如果返回值大于10000时，这个面板就会看到截断/部分数据。

在1.2.2版本中， `max-row-limit` 参数默认被设置为了0，这表示说对于返回值没有限制。

这个最大行的限制仅仅作用于非分块(non-chunked)的请求中，分块(chunked)的请求还是返回无限制的数据。

分块(chunking)

可以设置参数 `chunked=true` 开启分块，使返回的数据是流式的batch，而不是单个的返回。返回结果可以按10000数据点被分块，为了改变这个返回最大的分块的大小，可以在查询的时候加上 `chunk_size` 参数，例如返回数据点是每20000为一个批次。

```
    curl -G 'http://localhost:8086/query' --data-urlencode "db=deluge" --data-
    urlencode "chunked=true" --data-urlencode "chunk_size=20000" --data-urlencode
1.  "q=SELECT * FROM liters"
```

InfluxQL

现在你已经知道了如何查询数据，查看数据探索页面可以熟悉InfluxQL的用法。想要获取更多关于数据查询的HTTP接口的用法，情况API参考文档。

采样和数据保留

InfluxDB每秒可以处理数十万的数据点。如果要长时间地存储大量的数据，对于存储会是很大的压力。一个很自然的方式就是对数据进行采样，对于高精度的裸数据存储较短的时间，而对于低精度的的数据可以保存得久一些甚至永久保存。

InfluxDB提供了两个特性—连续查询(Continuous Queries简称CQ)和保留策略(Retention Policies简称RP)，分别用来处理数据采样和管理老数据的。这一章将会展示CQs和RPs的例子，看下在InfluxDB中怎么使用这两个特性。

定义

Continuous Query (CQ)是在数据库内部自动周期性跑着的一个InfluxQL的查询，CQs需要在 `SELECT` 语句中使用一个函数，并且一定包括一个 `GROUP BY time()` 语句。

Retention Policy (RP)是InfluxDB数据架构的一部分，它描述了InfluxDB保存数据的时间。InfluxDB会比较服务器本地的时间戳和请求数据里的时间戳，并删除比你在RPs里面用 `DURATION` 设置的更老的数据。一个数据库中可以有多个RPs但是每个数据库的RPs是唯一的。

这一章不会详细地介绍创建和管理CQs和RPs的语法，如果你对这两个概念还是很陌生的话，建议查看CQ文档和RP文档。

数据采样

本节使用虚构的实时数据，以10秒的间隔，来追踪餐厅通过电话和网站订购食品的订单数量。我们会把这些数据存在 `food_data` 数据库里，其measurement为 `orders` ，fields分别为 `phone` 和 `website` 。

就像这样:

```
1. name: orders
2. ------------
3. time                             phone     website
4. 2016-05-10T23:18:00Z     10        30
5. 2016-05-10T23:18:10Z     12        39
6. 2016-05-10T23:18:20Z     11        56
```

目标

假定在长时间的运行中，我们只关心每三十分钟通过手机和网站订购的平均数量，我们希望用RPs和CQs实现下面的需求：

- 自动将十秒间隔数据聚合到30分钟的间隔数据
- 自动删除两个小时以上的原始10秒间隔数据
- 自动删除超过52周的30分钟间隔数据

数据库准备

在写入数据到数据库 `food_data` 之前，我们先做如下的准备工作，在写入之前设置CQs是因为CQ只对最近的数据有效；即数据的时间戳不会比 `now()` 减去CQ的 `FOR` 子句的时间早，或是如果没有 `FOR` 子句的话比 `now()` 减去 `GROUP BY time()` 间隔早。

1. 创建数据库

```
1.  > CREATE DATABASE "food_data"
```

2. 创建一个两个小时的默认RP

如果我们写数据的时候没有指定RP的话，InfluxDB会使用默认的RP，我们设置默认的RP是两个小时。使用 `CREATE RETENTION POLICY` 语句来创建一个默认RP：

```
    > CREATE RETENTION POLICY "two_hours" ON "food_data" DURATION 2h REPLICATION 1
1.  DEFAULT
```

这个RP的名字叫 `two_hours` 作用于 `food_data` 数据库上， `two_hours` 保存数据的周期是两个小时，并作为 `food_data` 的默认RP。

复制片参数*(REPLICATION 1)*是必须的，但是对于单个节点的*InfluxDB*实例，复制片只能设为*1*

> 说明：在步骤1里面创建数据库时，InfluxDB会自动生成一个叫做 `autogen` 的RP，并作为数据库的默认RP， `autogen` 这个RP会永远保留数据。在输入上面的命令之后， `two_hours` 会取代 `autogen` 作为 `food_data` 的默认RP。

3. 创建一个保留52周数据的RP

接下来我们创建另一个RP保留数据52周，但不是数据库的默认RP。最终30分钟间隔的数据会保存在这个RP里面。

使用 `CREATE RETENTION POLICY` 语句来创建一个非默认的RP：

```
> CREATE RETENTION POLICY "a_year" ON "food_data" DURATION 52w REPLICATION 1
```

这个语句对数据库 `food_data` 创建了一个叫做 `a_year` 的RP， `a_year` 保存数据的周期是52周。去掉 `DEFAULT` 参数可以保证 `a_year` 不是数据库 `food_data` 的默认RP。这样在读写的时候如果没有指定，仍然是使用 `two_hours` 这个默认RP。

4. 创建CQ

现在我们已经创建了RPs，现在我们要创建一个CQ，去将10秒间隔的数据采样到30分钟的间隔，并把它们安装不同存储策略把它们存在不同的measurement里。

使用 `CREATE CONTINUOUS QUERY` 来生成一个CQ:

```
> CREATE CONTINUOUS QUERY "cq_30m" ON "food_data" BEGIN
  SELECT mean("website") AS "mean_website",mean("phone") AS "mean_phone"
  INTO "a_year"."downsampled_orders"
  FROM "orders"
  GROUP BY time(30m)
END
```

上面创建了一个叫做 `cq_30m` 的CQ作用于 `food_data` 数据库上。 `cq_30m` 告诉InfluxDB每30分钟计算一次measurement为 `orders` 并使用默认RP `tow_hours` 的字段 `website` 和 `phone` 的平均值，然后把结果写入到RP为 `a_year` ，两个字段分别是 `mean_website` 和 `mean_phone` 的measurement名为 `downsampled_orders` 的数据中。InfluxDB会每隔30分钟跑对之前30分钟的数据跑一次这个查询。

> 说明：注意到我们在 `INTO` 语句中使用了 `"<retention_policy>"."<measurement>"` 这样的语句，当要写入到非默认的RP时，就需要这样的写法。

结果

使用新的CQ和两个新的RPs， `food_data` 已经开始接收数据了。之后我们向数据库里写数据，并且持续一段时间之后，我们可以看到两个measurement分别是 `orders` 和 `downsampled_orders` 。

```
> SELECT * FROM "orders" LIMIT 5
name: orders
---------
time                           phone  website
2016-05-13T23:00:00Z      10     30
2016-05-13T23:00:10Z      12     39
```

```
2016-05-13T23:00:20Z      11      56
2016-05-13T23:00:30Z      8       34
2016-05-13T23:00:40Z      17      32

> SELECT * FROM "a_year"."downsampled_orders" LIMIT 5
name: downsampled_orders
---------------------
time                           mean_phone  mean_website
2016-05-13T15:00:00Z      12          23
2016-05-13T15:30:00Z      13          32
2016-05-13T16:00:00Z      19          21
2016-05-13T16:30:00Z      3           26
2016-05-13T17:00:00Z      4           23
```

在 `orders` 里面是10秒钟间隔的裸数据，保存时间为2小时。在 `downsampled_orders` 里面是30分钟的聚合数据，保存时间为52周。

注意到 `downsampled_orders` 返回的第一个时间戳比 `orders` 返回的第一个时间戳要早，这是因为InfluxDB已经删除了 `orders` 中时间比本地早两个小时的数据。InfluxDB会在52周之后开始删除 `downsampled_orders` 中的数据。

> 说明：注意这里我们在第二个语句中使用了 `"<retention_policy>"."<measurement>"` 来查询 `downsampled_orders` ，因为只要不是使用默认的RP我们就需要指定RP。
>
> 默认InfluxDB是每隔三十分钟check一次RP，在两次check之间， `orders` 中可能有超过两个小时的数据，这个check的间隔可以在InfluxDB的配置文件中更改。

使用RPs和CQs的组合，我们已经成功地创建的数据库并保存高精度的裸数据较短的时间，而保存高精度的数据更长时间。现在我们对这些特性的工作有了大概的了解，我们推荐到CQs和RPs去看更详细的文档。

硬件指南

这一章会提供一些InfluxDB的硬件推荐，并会回答一些问的最多的关于硬件的问题。下面的推荐都是基于InfluxDB 1.2中的 `TSM` 存储引擎。

将要探讨的问题：

- 是用单节点还是集群？
- 对于单节点的一般硬件指南
- 对于集群的硬件指南
- 什么时候需要更多的内存？
- 需要那种类型的磁盘？
- 需要多大的存储空间？
- 该怎么配置硬件？

好现在我们来一一看看这些问题：

是用单节点还是集群？

InfluxDB的单节点是完全开源的，InfluxDB的集群版本是闭源的商业版。单节点的实例没有冗余，如果服务不可用，写入和读取数据都会马上失败。集群提供了高可用和冗余，多个数据副本分布在多台服务器上，任何一台服务器的丢失都不会对集群造成重大的影响。

如果您的性能要求仅仅是中等或低负载范围，那么使用单节点的InfluxDB实例就够了。如果你对性能要求相当高，那么你需要集群将负载分担到多台机器上。

对于单节点的一般硬件指南

我们这里定义的InfluxDB的负载是基于每秒的写入的数据量、每秒查询的次数以及唯一series的数目。基于你的负载，我们给出CPU、内存以及IOPS的推荐。

InfluxDB应该跑在SSD上，任何其他存储配置将具有较低的性能特征，并且在正常处理中可能甚至无法从小的中断中恢复。

负载	每秒写入的字段数	每秒中等查询数	series数量
低	< 5千	<5	<10万
中等	<25万	<25	<1百万
高	>25万	>25	>1百万
相当高	>75万	>100	>1千万

> 说明：查询对于系统性能的影响很大简单查询：
>
> - 几乎没有函数和正则表达式
> - 时间限制在几分钟，或是几个小时，又或者到一天
> - 通常在几毫秒到几十毫秒内执行
>
> 中等查询：
>
> - 有多个函数或者一两个正则表达式
> - 有复杂点的 `GROUP BY` 语句或是时间有几个星期的数据
> - 通常在几百毫秒到几千毫秒内执行
>
> 复杂查询：
>
> - 有多个聚合函数、转换函数或者多个正则表达式
> - 时间跨度很大有几个月或是几年
> - 通常执行时间需要几秒

低负载推荐

- CPU：2~4核
- 内存：2~4GB
- IOPS：500

中等负载推荐

- CPU：4~6核
- 内存：8~32GB
- IOPS：500~1000

高负载推荐

- CPU：8+核
- 内存：32+GB
- IOPS：1000+

超高负载

要达到这个范围挑战很大，甚至可能不能完成；联系我们的 `t sales@influxdb.com` 寻求专业帮助吧。

对于集群的硬件指南

元节点

一个集群至少要有三个独立的元节点才能允许一个节点的丢失，如果要容忍 `n` 个节点的丢失则需要 `2n+1` 个元节点。集群的元节点的数目应该为奇数。不要是偶数元节点，因为这样在特定的配置下会导致故障。

元节点不需要多大的计算压力，忽略掉集群的负载，我们建议元节点的配置：

- CPU:1~2核
- 内存：512MB~1GB
- IOPS：50

数据节点

一个集群运行只有一个数据节点，但这样数据就没有冗余了。这里的冗余通过写数据的RP中的 `副本个数` 来设置。一个集群在丢失 `n-1` 个数据节点后仍然能返回完整的数据，其中 `n` 是副本个数。为了在集群内实现最佳数据分配，我们建议数据节点的个数为偶数。

对于集群的数据节点硬件的推荐和单节点的类似，数据节点应该至少有两个核的CPU，因为必须处理正常的读取和写入压力，以及集群内的数据的读写。由于集群通信开销，集群中的数据节点处理的吞吐量比同一硬件配置上的单实例的要少。

负载	每秒写入的字段数	每秒中等查询数	series数量
低	< 5千	<5	<10万
中等	<10万	<25	<1百万
高	>10万	>25	>1百万
相当高	>50万	>100	>1千万

> 说明：查询对于系统性能的影响很大简单查询：
>
> - 几乎没有函数和正则表达式
> - 时间限制在几分钟，或是几个小时，又或者到一天
> - 通常在几毫秒到几十毫秒内执行
>
> 中等查询：
>
> - 有多个函数或者一两个正则表达式
> - 有复杂点的 `GROUP BY` 语句或是时间有几个星期的数据
> - 通常在几百毫秒到几千毫秒内执行
>
> 复杂查询：
>
> - 有多个聚合函数、转换函数或者多个正则表达式
> - 时间跨度很大有几个月或是几年
> - 通常执行时间需要几秒

低负载推荐

- CPU: 2~4核
- 内存: 2~4GB
- IOPS: 1000

中等负载推荐

- CPU: 4~6核
- 内存: 8~32GB
- IOPS: 1000+

高负载推荐

- CPU: 8+核
- 内存: 32+GB
- IOPS: 1000+

企业Web节点

企业Web服务器主要充当具有类似负载要求的HTTP服务器。 对于大多数应用程序，它不需要性能很强。 一般集群将仅使用一个Web服务器，但是考虑到冗余，可以将多个Web服务器连接到单个后端Postgres数据库。

> 注意: 生产集群不应该使用SQLite数据库，因为它不被冗余的Web服务器允许，也不能像Postgres一样处理高负载。

推荐配置:

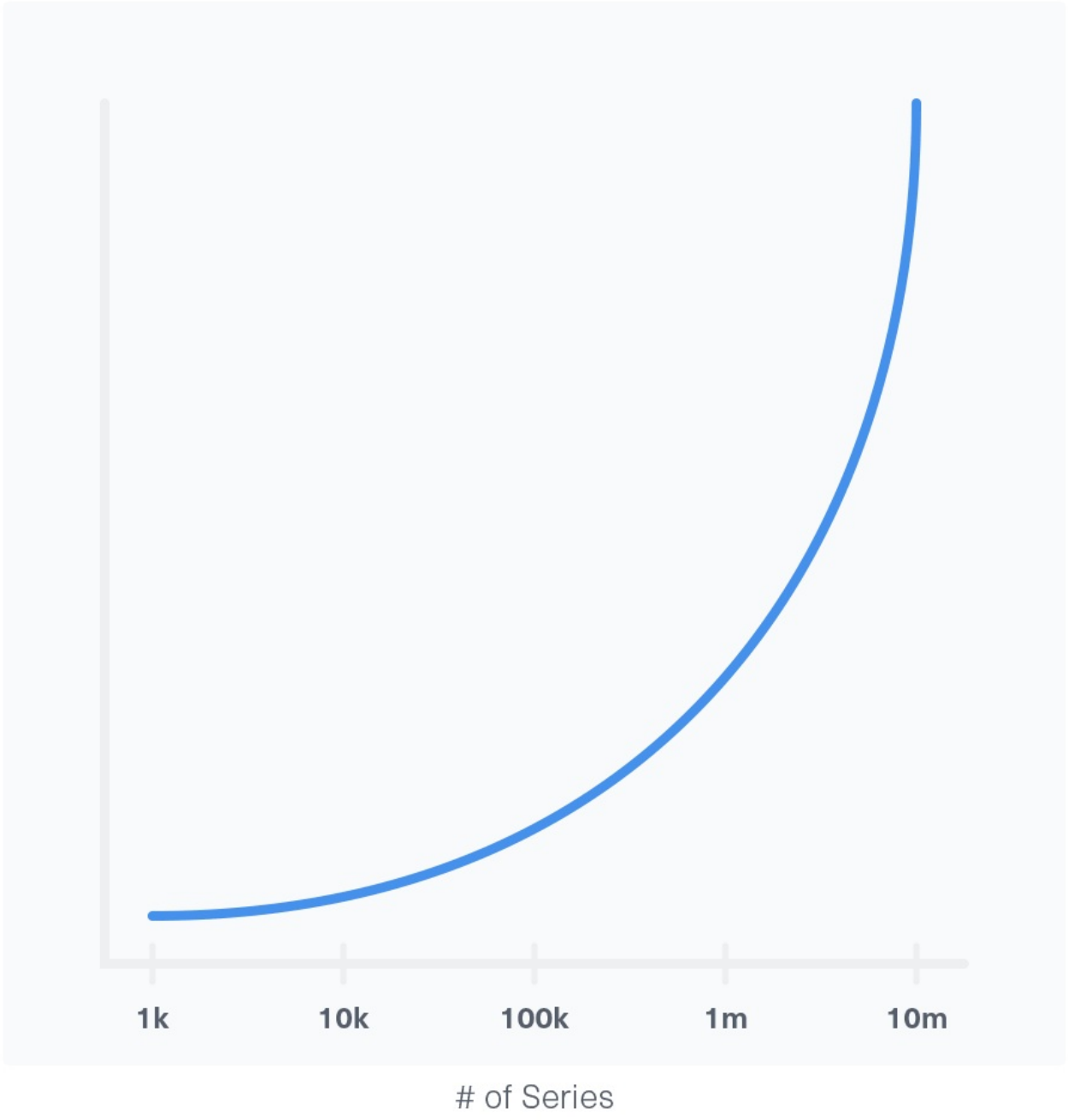
- CPU: 1~4核
- 内存: 1~2GB
- IOPS: 50

什么时候需要更多的内存？

一般来讲，内存越多，查询的速度越快，增加更多的内存总没有坏处。

影响内存的最主要的因素是series基数，series的基数大约或是超过千万时，就算有更多的内存也可能导致OOM，所以在设计数据的格式的时候需要考虑到这一点。

内存的增长和series的基数存在一个指数级的关系:



需要哪种类型的磁盘？

InfluxDB被设计运行在SSD上，InfluxData团队不会在HDD和网络存储上测试InfluxDB，所以不太建议在生产上这么去使用。在机械磁盘上性能会下降一个数量级甚至在中等负载下系统都可能死掉。为了最好的结果，InfluxDB至少需要磁盘提供1000 IOPS的性能。

注意集群的数据节点在做故障恢复的时候需要更高的IOPS，所以考虑到可能的数据恢复，我们建议磁盘至少有2000的IOPS，低于1000的IOPS，集群可能无法即时从短暂的中断中恢复。

需要多大的存储空间？

数据库的名字、measurement、tag keys、field keys和tag values只被存储一次且只能是字符串。只有field values和timestamp在每个数据点上都有存储。

非字符串类型的值大约需要3字节，字符串类型的值需要的空间由字符串的压缩来决定。

该怎么配置硬件？

当在生产上运行InfluxDB时， `wal` 和 `data` 文件夹需要在存储设备上分开。当系统处于大量写入负载下时，此优化可显着减少磁盘争用。 如果写入负载高度变化，这是一个重要的考虑因素。 如果写入负载不超过15%，则可能不需要优化。

HTTPS设置

这篇描述了怎样在InfluxDB中开启HTTPS。设置HTTPS可以保护客户端和InfluxDB服务器之间的通信，在某些情况下，HTTPS可以用来验证InfluxDB服务器对客户端的真实性。

如果你计划通过网络来发送请求到InfluxDB，我们强烈建议你开启HTTPS。

准备

为了给InfluxDB上设置HTTPS，你需要一个已有的或是新建的InfluxDB实例，还需要TLS证书也可以说SSL证书。InfluxDB支持三种类型的TLS/SSL证书：

- 由证书颁发机构签名的单域证书这些证书为HTTPS请求提供加密安全性，并允许客户端验证InfluxDB服务器的身份。 如果使用此证书，每个InfluxDB实例都需要一个唯一的单域证书。
- 证书颁发机构签发的通配证书这些证书为HTTPS请求提供加密安全性，并允许客户端验证InfluxDB服务器的身份。 可以在不同服务器上的多个InfluxDB实例中使用通配符证书。
- 自签证书自签名证书不由CA签名，您可以在自己的机器上生成。 与CA签署的证书不同，自签名证书仅为HTTPS请求提供加密安全性。 他们不允许客户端验证InfluxDB服务器的身份。 如果您无法获得CA签发的证书，我们建议使用自签名证书。 如果使用此证书，每个InfluxDB实例都需要一个唯一的自签名证书。

无论您的证书类型如何，InfluxDB都支持由私钥文件（.key）和签名证书文件（.crt）文件对组成的证书，以及将私钥文件和签名的证书文件组合成一个捆绑的证书文件（.pem）。

以下两部分将介绍在Ubuntu 16.04上如何使用CA签发的证书和自签名证书给InfluxDB设置HTTPS。其他操作系统的具体步骤可能不同。

使用CA签发的证书设置HTTPS

第一步：安装SSL / TLS证书

将私钥文件（.key）和签名的证书文件（.crt）或单个捆绑文件（.pem）放在 `/etc/ssl` 目录中。

第二步：确保文件权限

证书文件需要root用户的读写权限。通过运行以下命令确保您具有正确的文件权限：

```
1.  sudo chown root:root /etc/ssl/<CA-certificate-file>
```

```
sudo chmod 644 /etc/ssl/<CA-certificate-file>
sudo chmod 600 /etc/ssl/<private-key-file>
```

第三步：在InfluxDB的配置文件中开启HTTPS

默认HTTPS是关闭的，在InfluxDB的配置文件 `/etc/influxdb/influxdb.conf` 的 `[http]` 部分通过如下设置开启HTTPS：

- `https-enabled` 设为 `true`
- `http-certificate` 设为 `/etc/ssl/<signed-certificate-file>.crt`（或者 `/etc/ssl/<bundled-certificate-file>.pem`）
- `http-private-key` 设为 `/etc/ssl/<private-key-file>.key`（或者 `/etc/ssl/<bundled-certificate-file>.pem`）

```
[http]

  [...]

  # Determines whether HTTPS is enabled.
  https-enabled = true

  [...]

  # The SSL certificate to use when HTTPS is enabled.
  https-certificate = "<bundled-certificate-file>.pem"

  # Use a separate private key location.
  https-private-key = "<bundled-certificate-file>.pem"
```

第四步：重启InfluxDB

重启InfluxDB使配置生效：

```
sudo systemctl restart influxdb
```

第五步：验证HTTPS安装

可以通过InfluxDB的CLI来验证HTTPS是否工作：

```
influx -ssl -host <domain_name>.com
```

如果连接成功会返回：

```
1. Connected to https://<domain_name>.com:8086 version 1.x.x
2. InfluxDB shell version: 1.x.x
3. >
```

这样你就成功开启了InfluxDB的HTTPS了。

使用自签名证书设置HTTPS

第一步：生成自签名证书

以下命令生成私有密钥文件（.key）和自签名证书文件（.crt），该文件对于指定 `NUMBER_OF_DAYS` 情况下仍然有效。 它将这些文件输出到InfluxDB的默认证书文件路径，并向他们提供所需的权限。

```
    sudo openssl req -x509 -nodes -newkey rsa:2048 -keyout /etc/ssl/influxdb-
1. selfsigned.key -out /etc/ssl/influxdb-selfsigned.crt -days <NUMBER_OF_DAYS>
```

当执行该命令时，将提示您提供更多信息。 您可以选择填写该信息或将其留空；这两个操作都会生成有效的证书文件。

第二步：在InfluxDB的配置文件中开启HTTPS

默认HTTPS是关闭的，在InfluxDB的配置文件 `/etc/influxdb/influxdb.conf` 的 `[http]` 部分通过如下设置开启HTTPS：

- `https-enabled` 设为 `true`
- `http-certificate` 设为 `/etc/ssl/influxdb-selfsigned.crt`
- `http-private-key` 设为 `/etc/ssl/influxdb-selfsigned.key`

```
1. [http]
2.
3.   [...]
4.
5.   # Determines whether HTTPS is enabled.
6.   https-enabled = true
7.
8.   [...]
9.
10.  # The SSL certificate to use when HTTPS is enabled.
```

```
    https-certificate = "/etc/ssl/influxdb-selfsigned.crt"

    # Use a separate private key location.
    https-private-key = "/etc/ssl/influxdb-selfsigned.key"
```

第三步：重启InfluxDB

重启InfluxDB使配置生效：

```
sudo systemctl restart influxdb
```

第四步：验证HTTPS安装

可以通过InfluxDB的CLI来验证HTTPS是否工作：

```
influx -ssl -unsafeSsl -host <domain_name>.com
```

如果连接成功会返回：

```
Connected to https://<domain_name>.com:8086 version 1.x.x
InfluxDB shell version: 1.x.x
>
```

> 将Telegraf连接到一个安全的InfluxDB实例
>
> 将Telegraf连接到使用HTTPS的InfluxDB实例需要一些额外的步骤。
>
> 在Telegraf的配置文件（ `/etc/telegraf/telegraf.conf` ）中，编辑 `urls` 设置以指定 `https` 而不是 `http` ，并将 `localhost` 更改为相关域名。 如果您使用自签名证书，请取消 `insecure_skip_verify` 的注释设置并将其设置为true。

```
###############################################################################
#                            OUTPUT PLUGINS                                   #
###############################################################################

# Configuration for influxdb server to send metrics to
[[outputs.influxdb]]
  ## The full HTTP or UDP endpoint URL for your InfluxDB instance.
  ## Multiple urls can be specified as part of the same cluster,
  ## this means that only ONE of the urls will be written to each interval.
  # urls = ["udp://localhost:8089"] # UDP endpoint example
  urls = ["https://<domain_name>.com:8086"]
```

```
12.
13.  [...]
14.
15.    ## Optional SSL Config
16.    [...]
       insecure_skip_verify = true # <-- Update only if you're using a self-signed
17.  certificate
```

> 然后重启Telegraf就可以啦！

概念介绍

理解下面的概念，会让你更加充分利用InfluxDB。

关键概念

对InfluxDB核心架构的关键概念作简要说明，对于初学者来说很重要。

专业术语

列出InfluxDB的术语及其定义。

与SQL比较

InfluxDB的设计见解和权衡

简要介绍了在设计InfluxDB的时候对性能做的一些权衡。

schema设计

InfluxDB时间序列数据结构的概述及其如何影响性能。

存储引擎

概述下InfluxDB是如何将数据存储在磁盘上。

关键概念

在深入InfluxDB之前，最好是了解数据库的一些关键概念。 本文档简要介绍了这些概念和常用的InfluxDB术语。 我们在下面列出了所有涵盖的术语，但是我们建议您从头到尾阅读本文档，以获得对我们最喜爱的时间序列数据库的更全面了解。

database	field key	field set
field value	measurement	point
retention policy	series	tag key
tag set	tag value	timestamp

示例数据

下一节将参考下面列出的数据。 虽然数据是伪造的，但在InfluxDB中是一个很通用的场景。 数据展示了在2015年8月18日午夜至2015年8月18日上午6时12分在两个地点 `location` （地点 `1` 和地点 `2` ）显示两名科学家 `scientists` （ `langstroth` 和 `perpetua` ）计数的蝴蝶（ `butterflies`)和蜜蜂(`honeybees`)数量。 假设数据存在名为 `my_database` 的数据库中，而且存储策略是 `autogen` 。

```
1.  name: census
2.  -————————————
    time                                butterflies     honeybees
3.  location     scientist
    2015-08-18T00:00:00Z      12                23                   1
4.  langstroth
    2015-08-18T00:00:00Z      1                 30                   1
5.  perpetua
    2015-08-18T00:06:00Z      11                28                   1
6.  langstroth
    2015-08-18T00:06:00Z   3                    28                   1
7.  perpetua
    2015-08-18T05:54:00Z      2                 11                   2
8.  langstroth
    2015-08-18T06:00:00Z      1                 10                   2
9.  langstroth
    2015-08-18T06:06:00Z      8                 23                   2
10. perpetua
    2015-08-18T06:12:00Z      7                 22                   2
11. perpetua
```

其中census是 `measurement` ，butterflies和honeybees是 `field key` ，location和scientist是 `tag key` 。

讨论

现在您已经在InfluxDB中看到了一些示例数据，本节将详细分析这些数据。

InfluxDB是一个时间序列数据库，因此我们开始一切的根源就是—时间。在上面的数据中有一列是 `time` ，在InfluxDB中所有的数据都有这一列。 `time` 存着时间戳，这个时间戳以RFC3339格式展示了与特定数据相关联的UTC日期和时间。

接下来两个列叫作 `butterflies` 和 `honeybees` ，称为fields。fields由field key和field value组成。field key(`butterflies` 和 `honeybees`)都是字符串，他们存储元数据；field key `butterflies` 告诉我们蝴蝶的计数从12到7；field key `honeybees` 告诉我们蜜蜂的计数从23变到22。

field value就是你的数据，它们可以是字符串、浮点数、整数、布尔值，因为InfluxDB是时间序列数据库，所以field value总是和时间戳相关联。

在示例中，field value如下：

```
1.  12   23
2.  1    30
3.  11   28
4.  3    28
5.  2    11
6.  1    10
7.  8    23
8.  7    22
```

在上面的数据中，每组field key和field value的集合组成了 `field set` ，在示例数据中，有八个 `field set` ：

```
1.  butterflies = 12 honeybees = 23
2.  butterflies = 1 honeybees = 30
3.  butterflies = 11 honeybees = 28
4.  butterflies = 3 honeybees = 28
5.  butterflies = 2 honeybees = 11
6.  butterflies = 1 honeybees = 10
7.  butterflies = 8 honeybees = 23
8.  butterflies = 7 honeybees = 22
```

field是InfluxDB数据结构所必需的一部分——在InfluxDB中不能没有field。还要注意，field是没有索引的。如果使用field value作为过滤条件来查询，则必须扫描其他条件匹配后的所有值。因此，这些查询相对于tag上的查询（下文会介绍tag的查询）性能会低很多。 一般来说，字段不应包含常用来查询的元数据。

样本数据中的最后两列（ `location` 和 `scientist` ）就是tag。 tag由tag key和tag value组成。tag key和tag value都作为字符串存储，并记录在元数据中。示例数据中的tag key是 `location` 和 `scientist` 。 `location` 有两个tag value： `1` 和 `2` 。 `scientist` 还有两个tag value： `langstroth` 和 `perpetua` 。

在上面的数据中，tag set是不同的每组tag key和tag value的集合，示例数据里有四个tag set:

```
1. location = 1, scientist = langstroth
2. location = 2, scientist = langstroth
3. location = 1, scientist = perpetua
4. location = 2, scientist = perpetua
```

tag不是必需的字段，但是在你的数据中使用tag总是大有裨益，因为不同于field， tag是索引起来的。这意味着对tag的查询更快，tag是存储常用元数据的最佳选择。

> 不同场景下的数据结构设计
>
> 如果你说你的大部分的查询集中在字段 `honeybees` 和 `butterflies` 上:
>
> ```
> 1. SELECT * FROM "census" WHERE "butterflies" = 1
> 2. SELECT * FROM "census" WHERE "honeybees" = 23
> ```
>
> 因为field是没有索引的，在第一个查询里面InfluxDB会扫描所有的 `butterflies` 的值，第二个查询会扫描所有 `honeybees` 的值。这样会使请求时间很长，特别在规模很大时。为了优化你的查询，你应该重新设计你的数据结果，把field(`butterflies` 和 `honeybees`)改为tag，而将tag(`location` 和 `scientist`)改为field。
>
> ```
> 1. name: census
> 2. ------------
>    time                                 location     scientist
> 3. butterflies     honeybees
>    2015-08-18T00:00:00Z      1                 langstroth    12
> 4. 23
>    2015-08-18T00:00:00Z      1                 perpetua      1
> 5. 30
>    2015-08-18T00:06:00Z      1                 langstroth    11
> 6. 28
> ```

```
    2015-08-18T00:06:00Z   1                 perpetua      3
7.  28
    2015-08-18T05:54:00Z      2                 langstroth    2
8.  11
    2015-08-18T06:00:00Z      2                 langstroth    1
9.  10
    2015-08-18T06:06:00Z      2                 perpetua      8
10. 23
    2015-08-18T06:12:00Z      2                 perpetua      7
11. 22
```

现在 `butterflies` 和 `honeybees` 是tag了，当你再用上面的查询语句时，就不会扫描所有的值了，这也意味着查询更快了。

measurement作为tag，fields和time列的容器，measurement的名字是存储在相关fields数据的描述。 measurement的名字是字符串，对于一些SQL用户，measurement在概念上类似于表。样本数据中唯一的测量是 `census` 。 名称 `census` 告诉我们，fields值记录了 `butterflies` 和 `honeybees` 的数量，而不是不是它们的大小，方向或某种幸福指数。

单个measurement可以有不同的retention policy。 retention policy描述了InfluxDB保存数据的时间（DURATION）以及这些存储在集群中数据的副本数量（REPLICATION）。 如果您有兴趣阅读有关retention policy的更多信息，请查看数据库管理章节。

注意：在单节点的实例下，Replication系数不管用。

在样本数据中，measurement `census` 中的所有内容都属于 `autogen` 的retention policy。InfluxDB自动创建该存储策略； 它具有无限的持续时间和复制因子设置为1。

现在你已经熟悉了measurement，tag set和retention policy，那么现在是讨论series的时候了。 在InfluxDB中，series是共同retention policy，measurement和tag set的集合。 以上数据由四个series组成：

任意series编号	retention policy	measurement	tag set
series 1	`autogen`	`census`	`location = 1,scientist = langstroth`
series 2	`autogen`	`census`	`location = 2,scientist = langstroth`
series 3	`autogen`	`census`	`location = 1,scientist = perpetua`
series 4	`autogen`	`census`	`location = 2,scientist = perpetua`

理解series对于设计数据schema以及对于处理InfluxDB里面的数据都是很有必要的。

最后，point就是具有相同timestamp的相同series的field集合。例如，这就是一个point：

```
1. name: census
2. -----------------
```

```
    time                           butterflies     honeybees     location
3.  scientist
    2015-08-18T00:00:00Z     1                 30              1
4.  perpetua
```

例子里的series的retention policy为 `autogen` ，measurement为 `census` ，tag set为 `location = 1, scientist = perpetua` 。point的timestamp为 `2015-08-18T00:00:00Z` 。

我们刚刚涵盖的所有内容都存储在数据库（database）中——示例数据位于数据库 `my_database` 中。 InfluxDB数据库与传统的关系数据库类似，并作为users，retention policy，continuous以及point的逻辑上的容器。 有关这些主题的更多信息，请参阅身份验证和授权和连续查询(continuous query)。

数据库可以有多个users，retention policy，continuous和measurement。 InfluxDB是一个无模式数据库，意味着可以随时添加新的measurement，tag和field。 它旨在使时间序列数据的工作变得非常棒。

你做到了！你已经知道了InfluxDB中的基本概念和术语。如果你是初学者，我们建议您查看入门指南和写入数据和查询数据指南。 愿我们的时间序列数据库可以为您服务 。

专业术语

aggregation

这是一个InfluxQL的函数，可以返回一堆数据的聚合结果，可以看InfluxQL函数中现有的以及即将支持的聚合函数列表。

batch

用换行符分割的数据点的集合，这批数据可以使用HTTP请求写到数据库中。用这种HTTP接口的方式可以大幅降低HTTP的负载。尽管不同的场景下更小或更大的batch可能有更好地性能，InfluxData建议每个batch的大小在5000~10000个数据点。

continuous query（CQ）

这个一个在数据库中自动周期运行的InfluxQL的查询。Continuous query在 `select` 语句里需要一个函数，并且一定会包含一个 `GROUP BY time()` 的语法。

database

对于users，retention policy，continuous query以及时序数据的一个逻辑上的集合。

duration

retention policy中的一个属性，决定InfluxDB中数据保留多长时间。在duration之前的数据会自动从database中删除掉。

field

InfluxDB数据中记录metadata和真实数据的键值对。fields在InfluxDB的数据结构中是必须的且不会被索引。如果要用field做查询条件的话，那就必须遍历所选时间范围里面的所有数据点，这种方式对比与tag效率会差很多。

field key

组成field的键值对里面的键的部分。field key是字符串且保存在metadata中。

field set

数据点上field key和field value的集合。

field value

组成field的键值对里面的值的部分。field value才是真正的数据，可以是字符串，浮点数，整数，布尔型数据。一个field value总是和一个timestamp相关联。

field value不会被索引，如果要对field value做过滤话，那就必须遍历所选时间范围里面的所有数据点，这种方式对比与tag效率会差很多。

function

包括InfluxQL中的聚合，查询和转换，可以在InfluxQL函数中查看InfluxQL中的完整函数列表。

identifier

涉及continuous query的名字，database名字，field keys，measurement名字，retention policy名字，subscription 名字，tag keys以及user 名字的一个标记。

line protocol

写入InfluxDB时的数据点的文本格式。

measurement

InfluxDB数据结果中的一部分，描述了存在关联field中的数据的意义，measurement是字符串。

metastore

包含了系统状态的内部信息。metastore包含了用户信息，database，retention policy，shard metadata，continuous query以及subscription。

node

一个独立的 `influxd` 进程。

now()

本地服务器的当前纳秒级时间戳。

point

InfluxDB数据结构的一部分由series中的的一堆field组成。 每个点由其series和timestamp唯一标识。

你不能在同一series中存储多个具有相同timestamp的点。 相反，当你使用与该series中现有点相同的timestamp记将新point写入同一series时，该field set将成为旧field set和新field set的并集。

query

从InfluxDB里面获取数据的一个操作

replication factor

retention policy的一个参数，决定在集群模式下数据的副本的个数。InfluxDB在N个数据节点上复制数据，其中N就是replication factor。

> replication factor在单节点的实例上不起作用

retention policy(RP)

InfluxDB数据结构的一部分，描述了InfluxDB保存数据的长短(duration)，数据存在集群里面的副本数(replication factor)，以及shard group的时间范围(shard group duration)。RPs在每个database里面是唯一的，连同measurement和tag set定义一个series。

当你创建一个database的时候，InfluxDB会自动创建一个叫做 `autogen` 的retention policy，其duration为永远，replication factor为1，shard group的duration设为的七天。

schema

数据在InfluxDB里面怎么组织。InfluxDB的schema的基础是database，retention policy，series，measurement，tag key，tag value以及field keys。

selector

一个InfluxQL的函数，从特定范围的数据点中返回一个点。可以看InfluxQL函数中现有的以及即将支持的selector函数列表。

series

InfluxDB数据结构的集合，一个特定的series由measurement，tag set和retention policy组成。

> 注意：field set不是series的一部分

series cardinality

在InfluxDB实例上唯一database，measurement和tag set组合的数量。

例如，假设一个InfluxDB实例有一个单独的database，一个measurement。这个measurement有两个tag key: `email` 和 `status` 。如果有三个不同的email，并且每个email的地址关联两个不同的 `status` ，那么这个measurement的series cardinality就是6(3*2=6):

email	status
lorr@influxdata.com	start
lorr@influxdata.com	finish
marv@influxdata.com	start
marv@influxdata.com	finish
cliff@influxdata.com	start
cliff@influxdata.com	finish

注意到，因为所依赖的tag的存在，在某些情况下，简单地执行该乘法可能会高估series cardinality。 依赖的tag是由另一个tag限定的tag并不增加series cardinality。 如果我们将tag `firstname` 添加到上面的示例中，则系列基数不会是18（3 *2* 3 = 18）。 它将保持不变为6，因为 `firstname` 已经由 `email` 覆盖了:

email	status	firstname
lorr@influxdata.com	start	lorraine
lorr@influxdata.com	finish	lorraine
marv@influxdata.com	start	marvin
marv@influxdata.com	finish	marvin
cliff@influxdata.com	start	clifford

cliff@influxdata.com	finish	clifford

在常见问题中可以看到怎么根据series cardinality来查询InfluxDB。

server

一个运行InfluxDB的服务器，可以使虚拟机也可以是物理机。每个server上应该只有一个InfluxDB的进程。

shard

shard包含实际的编码和压缩数据，并由磁盘上的TSM文件表示。 每个shard都属于唯一的一个shard group。多个shard可能存在于单个shard group中。每个shard包含一组特定的series。给定shard group中的给定series上的所有点将存储在磁盘上的相同shard（TSM文件）中。

shard duration

shard duration决定了每个shard group跨越多少时间。具体间隔由retention policy中的 `SHARD DURATION` 决定。

例如，如果retention policy的 `SHARD DURATION` 设置为1w，则每个shard group将跨越一周，并包含时间戳在该周内的所有点。

shard group

shard group是shard的逻辑组合。shard group由时间和retention policy组织。包含数据的每个retention policy至少包含一个关联的shard group。给定的shard group包含shard group覆盖的间隔的数据的所有shard。每个shard group跨越的间隔是shard的持续时间。

subscription

subscription允许Kapacitor在push model中接收来自InfluxDB的数据，而不是基于查询数据的pull model。当Kapacitor配置为使用InfluxDB时，subscription将自动将订阅的数据库的每个写入从InfluxDB推送到Kapacitor。subscription可以使用TCP或UDP传输写入。

tag

InfluxDB数据结构中的键值对，tags在InfluxDB的数据中是可选的，但是它们可用于存储常用的metadata; tags会被索引，因此tag上的查询是很高效的。

tag key

组成tag的键值对中的键部分，tag key是字符串，存在metadata中。

tag set

数据点上tag key和tag value的集合。

tag value

组成tag的键值对中的值部分，tag value是字符串，存在metadata中。

timestamp

数据点关联的日期和时间，在InfluxDB里的所有时间都是UTC的。

transformation

一个InfluxQL的函数，返回一个值或是从特定数据点计算后的一组值。但是不是返回这些数据的聚合值。

tsm(Time Structured Merge tree)

InfluxDB的专用数据存储格式。 TSM可以比现有的B+或LSM树实现更大的压缩和更高的写入和读取吞吐量。

user

在InfluxDB里有两种类型的用户：

- admin用户对所有数据库都有读写权限，并且有管理查询和管理用户的全部权限。
- 非admin用户有针对database的可读，可写或者二者兼有的权限。

当认证开启之后，InfluxDB只执行使用有效的用户名和密码发送的HTTP请求。

values per second

对数据持续到InfluxDB的速率的度量，写入速度通常以values per second表示。

要计算每秒速率的值，将每秒写入的点数乘以每点存储的值数。 例如，如果这些点各有四个field，并且每秒写入batch是5000个点，那么values per second是每点4个fieldx每batch 5000个点x10个batch/秒=每秒200,000个值。

wal(Write Ahead Log)

最近写的点数的临时缓存。为了减少访问永久存储文件的频率，InfluxDB将最新的数据点缓冲进WAL中，直到其总大小或时间触发然后flush到长久的存储空间。这样可以有效地将写入batch处理到TSM中。

可以查询WAL中的点，并且系统重启后仍然保留。在进程开始时，在系统接受新的写入之前，WAL中的所有点都必须flushed。

与SQL比较

database里面是什么？

该章向SQL用户介绍了InfluxDB哪里像SQL数据库以及它哪里不像。将突出讲了两者之间的一些主要区别，并提供了不同的数据库术语和查询语言。

一般来说

InfluxDB是一个时间序列数据库，SQL数据库可以提供时序的功能，但严格说时序不是其目的。简而言之，InfluxDB用于存储大量的时间序列数据，并对这些数据进行快速的实时分析。

时间是一切

在InfluxDB中，timestamp标识了在任何给定数据series中的单个点。这就像一个SQL数据库表，其中主键是由系统预先设置的，并且始终是时间。

InfluxDB还会认识到您的schema可能随时间而改变。在InfluxDB中，您不需要在前面定义schema。数据点可以有一个measurement的field的一个，也可以有这个measurement的所有field，或其间的任何数字。 您可以在写数据的时候为该measurement添加一个新的field。

术语

下表是一个叫 `foodships` 的SQL数据库的例子，并有没有索引的 `#_foodships` 列和有索引的 `park_id` , `planet` 和 `time` 列。

park_id	planet	time	#_foodships
1	Earth	1429185600000000000	0
1	Earth	1429185601000000000	3
1	Earth	1429185602000000000	15
1	Earth	1429185603000000000	15
2	Saturn	1429185600000000000	5
2	Saturn	1429185601000000000	9
2	Saturn	1429185602000000000	10
2	Saturn	1429185603000000000	14
3	Jupiter	1429185600000000000	20
3	Jupiter	1429185601000000000	21
3	Jupiter	1429185602000000000	21

3	Jupiter	1429185603000000000	20
4	Saturn	1429185600000000000	5
4	Saturn	1429185601000000000	5
4	Saturn	1429185602000000000	6
4	Saturn	1429185603000000000	5

这些数据在InfluxDB看起来就像这样：

```
1. name: foodships
2. tags: park_id=1, planet=Earth
3. time                           #_foodships
4. ----                           ------------
5. 2015-04-16T12:00:00Z     0
6. 2015-04-16T12:00:01Z     3
7. 2015-04-16T12:00:02Z     15
8. 2015-04-16T12:00:03Z     15
9.
10. name: foodships
11. tags: park_id=2, planet=Saturn
12. time                           #_foodships
13. ----                           ------------
14. 2015-04-16T12:00:00Z     5
15. 2015-04-16T12:00:01Z     9
16. 2015-04-16T12:00:02Z     10
17. 2015-04-16T12:00:03Z     14
18.
19. name: foodships
20. tags: park_id=3, planet=Jupiter
21. time                           #_foodships
22. ----                           ------------
23. 2015-04-16T12:00:00Z     20
24. 2015-04-16T12:00:01Z     21
25. 2015-04-16T12:00:02Z     21
26. 2015-04-16T12:00:03Z     20
27.
28. name: foodships
29. tags: park_id=4, planet=Saturn
30. time                           #_foodships
31. ----                           ------------
32. 2015-04-16T12:00:00Z     5
33. 2015-04-16T12:00:01Z     5
34. 2015-04-16T12:00:02Z     6
```

```
35. 2015-04-16T12:00:03Z     5
```

参考上面的数据，一般可以这么说：

- InfluxDB的measurement(`foodships`)和SQL数据库里的table类似；
- InfluxDB的tag(`park_id` 和 `planet`)类似于SQL数据库里索引的列；
- InfluxDB中的field(`#_foodships`)类似于SQL数据库里没有索引的列；
- InfluxDB里面的数据点(例如 `2015-04-16T12:00:00Z 5`)类似于SQL数据库的行；

基于这些数据库术语的比较，InfluxDB的continuous query和retention policy与SQL数据库中的存储过程类似。 它们被指定一次，然后定期自动执行。

当然，SQL数据库和InfluxDB之间存在一些重大差异。SQL中的 `JOIN` 不适用于InfluxDB中的measurement。而且，正如我们上面提到的那样，一个measurement就像一个SQL的table，其中主索引总是被预设为时间。InfluxDB的时间戳记必须在UNIX epoch（GMT）或格式化为日期时间RFC3339格式的字符串才有效。

查看更多关于InfluxDB的术语的详细解释，请参考专业术语。

InfluxQL和SQL

在InfluxDB中InfluxQL是一种类SQL的语言。对于来自其他SQL或类SQL环境的用户来说，它已经被精心设计，而且还提供特定于存储和分析时间序列数据的功能。

InfluxQL的 `select` 语句来自于SQL中的 `select` 形式：

```
1. SELECT <stuff> FROM <measurement_name> WHERE <some_conditions>
```

`where` 是可选的，在InfluxDB里为了查询到上面数据，需要输入：

```
1. SELECT * FROM "foodships"
```

如果你仅仅想看planet为 `Saturn` 的数据：

```
1. SELECT * FROM "foodships" WHERE "planet" = 'Saturn'
```

如果你想看到planet为 `Saturn` ，并且在UTC时间为2015年4月16号12:00:01之后的数据：

```
    SELECT * FROM "foodships" WHERE "planet" = 'Saturn' AND time > '2015-04-16
1. 12:00:01'
```

如上例所示，InfluxQL允许您在 `WHERE` 子句中指定查询的时间范围。您可以使用包含单引号的日期

时间字符串，格式为YYYY-MM-DD HH: MM: SS.mmm（mmm为毫秒，为可选项，您还可以指定微秒或纳秒。您还可以使用相对时间与 `now()` 来指代服务器的当前时间戳：

```
1.  SELECT * FROM "foodships" WHERE time > now() - 1h
```

该查询输出measurement为 `foodships` 中的数据，其中时间戳比服务器当前时间减1小时。与now()做计算来决定时间范围的可选单位有：

字母	意思
u或μ	微秒
ms	毫秒
s	秒
m	分钟
h	小时
d	天
w	星期

InfluxQL还支持正则表达式，表达式中的运算符， `SHOW` 语句和 `GROUP BY` 语句。有关这些主题的深入讨论，请参阅我们的数据探索页面。 InfluxQL功能还包括 `COUNT` ， `MIN` ， `MAX` ， `MEDIAN` ， `DERIVATIVE` 等。 有关完整列表，请查看函数页面。

为什么InfluxDB不是CRUD的一个解释

InfluxDB是针对时间序列数据进行了优化的数据库。这些数据通常来自分布式传感器组，来自大型网站的点击数据或金融交易列表等。

这个数据有一个共同之处在于它只看一个点没什么用。一个读者说，在星期二UTC时间为12:38:35时根据他的电脑CPU利用率为12%，这个很难得出什么结论。只有跟其他的series结合并可视化时，它变得更加有用。随着时间的推移开始显现的趋势，是我们从这些数据里真正想要看到的。另外，时间序列数据通常是一次写入，很少更新。

结果是，由于优先考虑create和read数据的性能而不是update和delete，InfluxDB不是一个完整的CRUD数据库，更像是一个CR-ud。

InfluxDB的设计见解和权衡

InfluxDB是一个时间序列数据库。针对这种用例进行优化需要进行一些权衡，主要是以牺牲功能为代价来提高性能。以下列出了一些权衡过的设计见解：

1、对于时间序列用例，我们假设如果相同的数据被多次发送，那么认为客户端几次都是同一笔数据。

- 优势：通过简化的冲突解决增加了写入性能
- 劣势：不能存储重复数据;可能会在极少数情况下覆盖数据

2、删除是罕见的事情。当它们发生时，肯定是针对大量的旧数据，这些数据对于写入来说是冷数据。

- 优势：限制删除操作，从而增加查询和写入性能
- 劣势：删除功能受到很大限制

3、对现有数据的更新是罕见的事件，持续地更新永远不会发生。时间序列数据主要是永远不更新的新数据。

- 优势：限制更新操作，从而增加查询和写入性能
- 劣势：更新功能受到很大限制

4、绝大多数写入都是接近当前时间戳的数据，并且数据是按时间递增的顺序添加。

- 优势：按时间递增的顺序添加数据明显更高效些
- 劣势：随机时间或时间不按升序写入点的性能要低得多

5、 规模至关重要。数据库必须能够处理大量的读取和写入。

- 优势：数据库可以处理大量的读取和写入
- 劣势：InfluxDB开发团队被迫做出权衡来提高性能

6、能够写入和查询数据比具有强一致性更重要。

- 优势：多个客户端可以在高负载的情况下完成查询和写入数据库操作
- 劣势：如果数据库负载较重，查询返回结果可能不包括最近的点

7、许多时间序列都是短暂的。经常是时间序列，只出现了几个小时，然后消失，例如一个新的主机，开机并监控数据被写入一段时间，然后被关闭。

- 优势：InfluxDB善于管理不连续数据
- 劣势：无模式设计意味着不支持某些数据库功能，例如没有交叉表连接

8、没有数据点太重要了。

- 优势：InfluxDB具有非常强大的工具来处理聚合数据和大数据集
- 劣势：数据点没有传统意义上的ID，它们被时间戳和series区分开来

schema设计

每个InfluxDB用例都可能是不一样的，schema将反映出这种独特性。 但是，在设计schema时，有一些遵循的一般准和可以避免的陷阱。

一般建议

推崇的schema设计

没有特定的顺序，我们有如下建议：

哪些情况下使用tag

一般来说，你的查询可以指引你哪些数据放在tag中，哪些放在field中。

- 把你经常查询的字段作为tag
- 如果你要对其使用`GROUP BY()`，也要放在tag中
- 如果你要对其使用InfluxQL函数，则将其放到field中
- 如果你需要存储的值不是字符串，则需要放到field中，因为tag value只能是字符串

避免InfluxQL中关键字作为标识符名称

这不是必需的，但它简化了写查询；您不必将这些标识符包装在双引号中。 标识符有database名称，retention policy名称，user名，measurement名称，tag key和field key。请参阅InfluxQL关键词看下哪些单词需要被避免。

请注意，如果查询中包含除[A-z，_]以外的字符，则还需要将它们用双引号括起来。

避免的schema设计

没有特定的顺序，我们有如下建议：

不要有太多的series

tags包含高度可变的信息，如UUID，哈希值和随机字符串，这将导致数据库中的大量measurement，通俗地说是高series cardinality。series cardinality高是许多数据库高内存使用的主要原因。

请参阅硬件指南中基于你的硬件的series cardinality的建议。如果系统有内存限制，请考虑将高cardinality数据存储为field而不是tag。

如何设计measurement

一般来说，谈论这一步可以简化你的查询。InfluxDB的查询会合并属于同一measurement范围内的数据；用tag区分数据比使用详细的measurement名字更好。

例如：考虑如下的schema：

```
Schema 1 - Data encoded in the measurement name
-------------
blueberries.plot-1.north temp=50.1 1472515200000000000
blueberries.plot-2.midwest temp=49.8 1472515200000000000
```

这个没有tag的长长的measurement名字(`blueberries.plot-1.north`)有些类似于Graphite的metric。像 `plot``region` 这样的信息放在measurement名字里面将会使数据很难去查询。

例如，使用schema 1计算两个图1和2的平均 `temp` 是不可能的。将其与如下schema进行比较：

```
Schema 2 - Data encoded in tags
-------------
weather_sensor,crop=blueberries,plot=1,region=north temp=50.1
1472515200000000000
weather_sensor,crop=blueberries,plot=2,region=midwest temp=49.8
1472515200000000000
```

以下查询计算了落在北部地区的蓝莓的平均 `temp` 。 虽然这两个查询都比较简单，但使用正则表达式使得某些查询更加复杂或根本不可能实现。

```
# Schema 1 - Query for data encoded in the measurement name
> SELECT mean("temp") FROM /\.north$/

# Schema 2 - Query for data encoded in tags
> SELECT mean("temp") FROM "weather_sensor" WHERE "region" = 'north'
```

不要把多条信息放到一个tag里面

与上述相似，将具有多条信息的单个tag拆分为多个单独的tag将简化查询并减少对正则表达式的需求。

例如，考虑如下的schema：

```
Schema 1 - Multiple data encoded in a single tag
-------------
```

```
    weather_sensor,crop=blueberries,location=plot-1.north temp=50.1
3.  1472515200000000000
    weather_sensor,crop=blueberries,location=plot-2.midwest temp=49.8
4.  1472515200000000000
```

上述数据将多个单独的参数 `plot``region` 放到了一个长tag value里面（ `plot-1.north` ）。将其与如下schema进行比较：

```
1.  Schema 2 - Data encoded in multiple tags
2.  -------------
    weather_sensor,crop=blueberries,plot=1,region=north temp=50.1
3.  1472515200000000000
    weather_sensor,crop=blueberries,plot=2,region=midwest temp=49.8
4.  1472515200000000000
```

以下查询计算了落在 `north` 地区的蓝莓的平均 `temp` 。 虽然这两个查询都是相似的，但在Schema 2中使用多个tag避免了使用正则表达式。

```
1.  # Schema 1 - Query for multiple data encoded in a single tag
2.  > SELECT mean("temp") FROM "weather_sensor" WHERE location =~ /\.north$/
3.
4.  # Schema 2 - Query for data encoded in multiple tags
5.  > SELECT mean("temp") FROM "weather_sensor" WHERE region = 'north'
```

shard group的保留时间(duration)的管理

shard group的保留时间(duration)预览

InfluxDB将数据存储在shard group中。shard group由存储策略（RP）管理，并存储具有特定时间间隔内的时间戳的数据。 该时间间隔的长度称为shard group的duration。

默认情况下，shard group的duration是由RP的duration决定：

RP duration	shard group duration
< 2 days	a hour
>= 2 days and <= 6 months	1 day
> 6 months	7 days

在每个RP上shard group的duration也是可以配置的，可以看Retention Policy管理看如何配置shard group的duration。

shard group的duration的建议

通常，较短的shard group的duration允许系统有效地丢弃数据。 当InfluxDB强制执行RP时，它会丢弃整个shard group，而不是单个数据点。 例如，如果您的RP有一天的持续时间，一个shard group持续时间为一小时； 则InfluxDB每小时将丢弃一小时的数据。

如果您的RP的持续时间大于6个月，则不需要具有较短的shard group持续时间。 事实上，将shard group的持续时间提高到默认的七天值以上可以改善压缩，提高写入速度，并减少每个shard group的固定迭代器开销。 例如，50年及以上的shard group持续时间是可接受的配置。

> 说明：当配置shard group的duration的时候， `INF` (infinite)是一个不合理的配置。在实际工作中，特定的duration `1000w` 就足够达到非常长的shard group持续时间。

我们建议shard group的配置如下：

- 是你一般查询时间范围的两倍
- 每个shard group里至少有100000个数据点
- 每个shard group里面的每个series至少有1000个数据点

存储引擎

InfluxDB的存储引擎和TSM

新的InfluxDB的存储引擎看起来和LSM树很像。它具有wal和一组只读数据文件，它们在概念上与LSM树中的SSTables类似。 TSM文件包含排序，压缩的series数据。

InfluxDB将为每个时间段创建一个分片。例如，如果您有一个持续时间无限制的存储策略，则会为每7天的时间段创建一个分片。 这些每一个分片都映射到底层存储引擎数据库。 每一个这些数据库都有自己的WAL和TSM文件。

下面我们来深入存储引擎的这些部分。

存储引擎

存储引擎将多个组件结合在一起，并提供用于存储和查询series数据的外部接口。 它由许多组件组成，每个组件都起着特定的作用:

- In-Memory Index —— 内存中的索引是分片上的共享索引，可以快速访问measurement，tag和series。 引擎使用该索引，但不是特指存储引擎本身。
- WAL —— WAL是一种写优化的存储格式，允许写入持久化，但不容易查询。 对WAL的写入就是append到固定大小的段中。
- Cache —— Cache是存储在WAL中的数据的内存中的表示。 它在运行时可以被查询，并与TSM文件中存储的数据进行合并。
- TSM Files —— TSM Files中保存着柱状格式的压缩过的series数据。
- FileStore —— FileStore可以访问磁盘上的所有TSM文件。 它可以确保在现有的TSM文件被替换时以及删除不再使用的TSM文件时，创建TSM文件是原子性的。
- Compactor —— Compactor负责将不够优化的Cache和TSM数据转换为读取更为优化的格式。它通过压缩series，去除已经删除的数据，优化索引并将较小的文件组合成较大的文件来实现。
- Compaction Planner —— Compaction Planner决定哪个TSM文件已准备好进行压缩，并确保多个并发压缩不会彼此干扰。
- Compression —— Compression由各种编码器和解码器对特定数据类型作处理。一些编码器是静态的，总是以相同的方式编码相同的类型；还有一些可以根据数据的类型切换其压缩策略。
- Writers/Readers —— 每个文件类型（WAL段，TSM文件，tombstones等）都有相应格式的Writers和Readers。

Write Ahead Log(WAL)

WAL被组织成一堆看起来像 `_000001.wal` 这样的文件。 文件编号单调增，并称为WAL段。 当分段达到10MB的大小时，该段将被关闭并且打开一个新的分段。每个WAL段存储多个压缩过的写入和删除块。

当一个新写入的点被序列化时，使用Snappy进行压缩，并写入WAL文件。 该文件是 `fsync'd` ，并且在返回成功之前将数据添加到内存中的索引。 这意味着批量的数据点写入可以实现更高的性能。（在大多数情况下，最佳批量大小似乎是每批5,000-10,000点。）

WAL中的每个条目都遵循TLV标准，以一个单字节表示条目类型（写入或删除），然后压缩块长度的4字节 `uint32` ，最后是压缩块。

Cache

缓存是对存储在WAL中的所有数据点的内存拷贝。这些点由这些key组成，他们是measurement，tag set和唯一field组成。每个field都按照自己的有序时间范围保存。缓存数据在内存中不被压缩。

对存储引擎的查询将会把Cache中的数据与TSM文件中的数据进行合并。在查询运行时间内，对数据副本的查询都是从缓存中获取的。这样在查询进行时写入的数据不会被查询出来。

发送到缓存的删除指令，将清除给定键或给定键的特定时间范围的数据。

缓存提供了一些控制器用于快照。两个最重要的控制器是内存限制。 有一个下限， `cache-snapshot-memory-size` ，超出时会触发快照到TSM文件，并删除相应的WAL段。 还有一个上限， `cache-max-memory-size` ，当超出时会导致Cache拒绝新的写入。 这些配置有助于防止内存不足的情况，并让客户端写数据比实例可承受的更快。

内存阈值的检查发生在每次写入时。

还有快照控制器是基于时间的。 `cache-snapshot-write-cold-duration` ，如果在指定的时间间隔内没有收到写入，则强制缓存到TSM文件的快照。

通过重新读取磁盘上的WAL文件，可以重新创建内存中缓存。

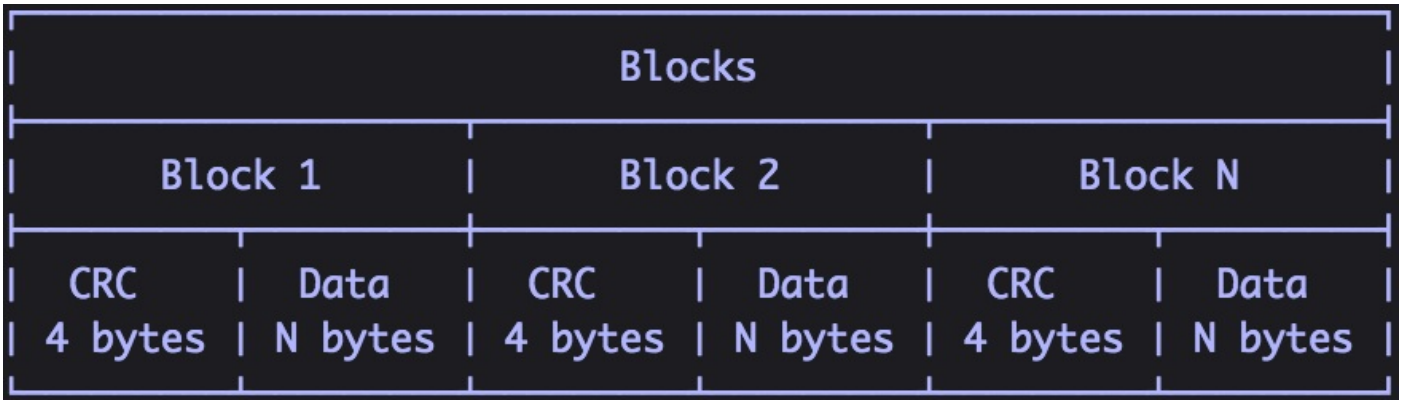
TSM Files

TSM files是内存映射的只读文件的集合。 这些文件的结构看起来与LevelDB中的SSTable或其他LSM Tree变体非常相似。

一个TSMfile由四部分组成：header，blocks，index和footer：

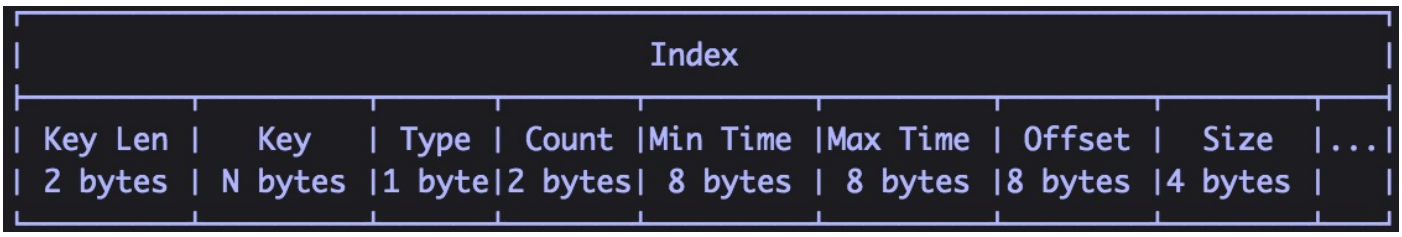
Header是识别文件类型和版本号的一个魔法数字：

blocks是一组CRC32校验和数据对的序列。block数据对文件是不透明的。CRC32用于块级错误检测。block的长度存储在索引中。



blocks之后是文件中blocks的索引。索引由先按key顺序，如果key相同则按时间顺序排列的索引条目序列组成。key包括measurement名称，tag set和一个field。如果一个点有多个field则在TSM文件中创建多个索引条目。每个索引条目以密钥长度和密钥开始，后跟block类型（float，int，bool，string）以及该密钥后面的索引block条目数的计数。 然后是每个索引block条目，其由block的最小和最大时间组成，之后是block所在的文件的偏移量以及block的大小。 包含该key的TSM文件中每个block都有一个索引block条目。

索引结构可以提供对所有block的有效访问，以及能够确定访问给定key相关联数据需要多大代价。给定一个key和时间戳，我们可以确定文件是否包含该时间戳的block。我们还可以确定该block所在的位置，以及取出该block必须读取多少数据。了解了block的大小，我们可以有效地提供IO语句。



最后一部分是footer，它存储了索引开头的offset。

Compression

每个block都被压缩，以便减少存储空间和查询时磁盘IO。block包含时间戳和给定seris和field的值。每个block都有一个字节的header，之后跟着压缩过的时间戳，然后是压缩后的值。

时间戳和值都会被压缩，并使用依赖于数据类型及其形状的编码分开存储。独立存储允许时间戳编码用于所有时间戳，同时允许不同字段类型的不同编码。例如，一些点可能能够使用游程长度编码，而其他点可能不能。

每个值类型还包含一个1byte的header，表示剩余字节的压缩类型。 四个高位存储压缩类型，如果需要，四个低位由编码器使用。

Timestamps

时间戳编码是自适应的，并且基于被编码的时间戳的结构。它使用delta编码，缩放和使用simple8b游程编码压缩的组合，当然如果需要，可以回退到无压缩。

时间戳分辨率是可变的，可以像纳秒一样粒度，最多需要8个字节来存储未压缩的时间戳。在编码期间，这些值首先进行delta编码。第一个值是起始时间戳，后续值是与先前值的差值。这通常将值转换成更小的整数，更容易压缩。许多时间戳也是单调增加，并且在每10秒的时间的均匀边界上落下。当时间戳具有这种结构时，它们由也是10的因子的最大公约数来缩放。这具有将非常大的整数增量转换成更小的压缩更好的效果。

使用这些调整值，如果所有delta都相同，则使用游程编码来存储时间范围。如果游程长度编码是不可能的，并且纳秒分辨率的所有值都小于（1 << 60）-1（～36.5年+ - + 1 +纳秒+至+年）），则使用simple8b编码对时间戳进行编码。 Simple8b编码是一个64位字对齐的整数编码，将多个整数打包成一个64位字。如果任何值超过最大值，则使用每个块的8个字节对未压缩的三进制进行存储。未来的编码可能使用修补方案，如“Patched Frame-Of-Reference (PFOR)”来更有效地处理异常值。

Floats

使用Facebook Gorilla paper实现对浮点数的编码。当值靠近在一起时，编码将连续值XORs连在一起让结果集变得更小。然后使用控制位存储增量，以指示XOR值中有多少前导零和尾随零。我们的实现会删除paper中描述的时间戳编码，并且仅对浮点值进行编码。

Integers

整数编码使用两种不同的策略，具体取决于未压缩数据中的值的范围。编码值首先使用ZigZag编码进行编码。这样在正整数范围内交错正整数和负整数。

例如，[-2，-1,0,1]变成[3,1,0,2]。 有关详细信息，请参阅Google的Protocol Buffers文档。

如果所有ZigZag编码值都小于（1 << 60）-1，则使用simple8b编码进行压缩。如果有值大于最大值，则所有值都将在未压缩的块中存储。如果所有值相同，则使用游程长度编码。这对于频繁不变的值非常有效。

Booleans

布尔值使用简单的位打包策略进行编码，其中每个布尔值使用1位。使用可变字节编码在块的开始处存储编码的布尔值的数量。

Strings

字符串使用Snappy压缩进行编码。每个字符串连续打包，然后被压缩为一个较大的块。

Compactions

Compactions是将以写优化格式存储的数据迁移到更加读取优化的格式的循环过程。在分片写入时，会发生Compactions的许多阶段：

- Snapshots —— Cache和WAL中的数据必须转换为TSM文件以释放WAL段使用的内存和磁盘空间。这些Compactions基于高速缓存和时间阈值进行。
- Level Compactions —— Level Compactions（分为1-4级）随TSM文件增长而发生。TSM文件从snapshot压缩到1级文件。多个1级文件被压缩以产生2级文件。该过程继续，直到文件达到级别4和TSM文件的最大大小。除非需要运行删除，index optimization compactions或者full compactions，否则它们会进一步压缩。较低级别的压缩使用避免CPU密集型活动（如解压缩和组合块）的策略。 较高的水平（因此较不频繁）的压缩将重新组合块来完全彻底压缩它们并增加压缩比。
- Index Optimization —— 当许多4级TSM文件累积时，内部索引变大，访问成本更高。Index Optimization compaction通过一组新的TSM文件分割series和index，将给定series的所有点排序到一个TSM文件中。 在Index Optimization之前，每个TSM文件包含大多数或全部series的点，因此每个TSM文件包含相同的series索引。Index Optimization后，每个TSM文件都包含从最小的series中得到的点，文件之间几乎没有series重叠。因此，每个TSM文件具有较小的唯一series索引，而不是完整series列表的副本。此外，特定series的所有点在

TSM文件中是连续的，而不是分布在多个TSM文件中。

- Full Compaction —— 当分片数据已经写入很长时间(也就是冷数据)，或者在分片上发生删除时，Full Compaction就会运行。Full Compaction产生最佳的TSM文件集，并包括来自Level和Index Optimization的所有优化。一旦一个shard上运行了Full Compaction，除非存储有新的写入或删除，否则不会在其上运行其他压缩。

Writes(写)

写入是同时写到WAL的segment和Cache中。每个WAL segment都有最大尺寸。一旦当前文件填满，将写入一个新文件。Cache也有大小限制，当Cache满了之后，会产生snapshot并且启动WAL的compaction。如果在一定时间内，写入速率超过了WAL的compaction速率，则Cache可能变得过满，在这种情况下，新的写入将失败直到snapshot进程追赶上。

当WAL segment填满并关闭时，Compactor会将Cache并将数据写入新的TSM文件。当TSM文件成功写入和 `fsync` ‘d时，它将会被FileStore加载和引用。

Updates(更新)

正常写入会发生更新（为已存在的点写入较新的值），由于缓存值覆盖现有值，因此较新的写入优先。如果写入将覆盖先前TSM文件中的一个点，则这些点在查询运行时会合并，较新的写入优先。

Deletes(删除)

通过向measurement或series的WAL写入删除条目然后更新Cache和FileStore来进行删除。这时Cache会去掉所有相关条目，FileStore为包含相关数据的每个TSM文件写入一个tombstone文件。这些tombstone文件被用于在启动时忽略相应的block，以及在compaction期间移除已删除的条目。

在compaction完全从TSM文件中删除数据之前，部分删除的series在查询时处理。

Queries(查询)

当存储引擎执行查询时，它本质上是寻找给定时间相关的特定series key和field。首先，我们对数据文件进行搜索，以查找包含与查询匹配的时间范围以及包含匹配series的文件。

一旦我们选择了数据文件，我们接下来需要找到series key索引条目的文件中的位置。我们针对每个TSM索引运行二进制搜索，以查找其索引块的位置。

在通常的情况下，这些块不会跨多个TSM文件重叠，我们可以线性搜索索引条目以找到要读取的起始块。如果存在重叠的时间块，则索引条目将被排序，以确保较新的写入将优先，并且可以在查询执行期间按顺序处理该块。

当迭代索引条目时，块将从其位置顺序地被读取。该块被解压缩，以便我们寻求具体的数据点。

新的InfluxDB存储引擎：从LSM树到B+树，然后重新创建TSM

写一个新的存储引擎应该是最后的手段。那么InfluxData最终如何写我们自己的引擎的呢？InfluxData已经尝试了许多存储格式，发现每个在某一方面都有一些缺点。InfluxDB的性能要求非常高，当然最终压倒了其他存储系统。InfluxDB的0.8版本允许多个存储引擎，包括LevelDB，RocksDB，HyperLevelDB和LMDB。 InfluxDB的0.9版本使用BoltDB作为底层存储引擎。下面要介绍的TSM，它在0.9.5中发布，是InfluxDB 0.11+中唯一支持的存储引擎，包括整个1.x系列。

时间序列数据用例的特性使许多现有存储引擎很有挑战性。在InfluxDB开发过程中，我们尝试了一些更受欢迎的选项。我们从LevelDB开始，这是一种基于LSM树的引擎，针对写入吞吐量进行了优化。之后，我们尝试了BoltDB，这是一个基于内存映射B+ Tree的引擎，它是针对读取进行了优化的。最后，我们最终建立了我们自己的存储引擎，它在许多方面与LSM树类似。

借助我们的新存储引擎，我们可以达到比B+ Tree实现高达45倍的磁盘空间使用量的减少，甚至比使用LevelDB及其变体有更高的写入吞吐量和压缩率。这篇文章将介绍整个演变的细节，并深入了解我们的新存储引擎及其内部工作。

时序数据的特性

时间序列数据与正常的数据库工作负载有很大的不同。有许多因素使得它难以提高和保持性能：

- 数十亿个数据点
- 高吞吐量的写入
- 高吞吐量的读取
- 大量删除（数据到期）
- 大部分数据是插入/追加，很少更新

第一个也是最明显的问题是规模。在DevOps，IoT或APM中，每天很容易收集数亿或数十亿的数据点。

例如，假设我们有200个VM或服务器运行，每个服务器平均有100个measurement每10秒收集一次。鉴于一天中有86,400秒，单个measurement每个服务器将在一天内产生8,640点。这样我们每天总共200 *100* 8,640 = 172,800,000个数据点。我们在传感器数据用例中还可以找到类似或更大的数字。

数据量大意味着写入吞吐量可能非常高。一些较大的公司一般需要每秒处理数百万次写入的系统。

同时，时间序列数据也可能是需要高吞吐量读取的。的确，如果您正在跟踪70万个metric或时间序列，那么你肯定不希望将其全部可视化。这导致许多人认为您实际上并没有读取大量数据的需求。然而，除了人们在屏幕上的仪表板之外，还有自动化系统用于监视或组合大量时间序列数据与其他类型的数据。

在InfluxDB内部，实时计算的聚合函数可将数万个不同时间series组合成单个视图。这些查询中的每一个都必须读取每个聚合数据点，因此对于InfluxDB，读取吞吐量通常比写入吞吐量高许多倍。

鉴于时间序列大多是顺序插入，你可能会认为可以在B+树上获得出色的性能，因为顺序插入是高效的，您可以达到每秒100,000以上。但是，我们有这些数据的写入发生在不同的时间序列。因此，插入最终看起来更像是随机插入，而不仅仅是顺序插入。

使用时间序列数据发现的最大问题之一是，在超过一定时间后需要删除所有数据。这里的常见模式是用户拥有高精度的数据，保存在短时间内，如几天或几个月。然后用户将数据采样并将其汇总到保存较长时间的较低精度数据。

最容易的实现将是简单地删除每个记录一旦超过其过期时间。然而，这意味着一旦写入的第一个点到达其到期日期，系统正在处理与写入一样多的删除，大多数存储引擎都不会这样去设计的。

我们来看看我们尝试过的两种存储引擎的细节，以及这些特性对我们性能的重大的影响。

LevelDB和LSM树

当InfluxDB项目开始时，我们选择了LevelDB作为存储引擎，因为我们将其用于作为InfluxDB前身的产品中的时间序列数据存储。我们知道它具有很好的写入吞吐量，但一切似乎都“just work”。

LevelDB是在Google构建为开源项目的Log Structured Merge Tree（或LSM树）的实现。 它暴露了键/值存储的API，其中key space是经过排序的。这最后一部分对于时间序列数据很重要，只要把时间戳放在key中，就允许我们快速扫描时间范围。

LSM树基于采用写入和两个称为Mem Tables和SSTables的结构的日志。 这些tables代表了排序的keyspace。SSTables是只读文件，只能被其插入和更新的其他SSTables所替换。

LevelDB为我们带来的两大优势是写入吞吐量高，内置压缩。 然而，当我们从时间序列数据中了解到人们需要什么时，我们遇到了一些不可逾越的挑战。

我们遇到的第一个问题是LevelDB不支持热备份。如果要对数据库进行安全备份，则必须将其关闭，然后将其复制。 LevelDB变体RocksDB和HyperLevelDB解决了这个问题，但还有另一个更紧迫的问题，我们认为他们解决不了。

我们的用户需要一种自动管理数据保留的方法。这意味着我们需要大量的删除。在LSM树中，删除与写入一样甚至更加昂贵。删除需要写入一个称为tombstone的新纪录。之后，查询会将结果集与任何tombstone合并，以从查询返回中清除已删除的数据。之后，将执行一个compaction操作，删除SSTable文件中的tombstone和底层删除的记录。

为了避免删除操作，我们将数据分割成我们称之为shard的数据，这些数据是连续的时间块。shard通常会持有一天或七天的数据。每个shard映射到底层的LevelDB。这意味着我们可以通过关闭数据库并删除底层文件来删除一整天的数据。

RocksDB的用户现在可以提出一个名为ColumnFamilies的功能。当将时间序列数据放入Rocks时，通常将时间块分成列族，然后在时间到达时删除它们。这是一个一般的想法：创建一个单独的区域，您可以在删除大量数据时只删除文件而不是更新索引。 删除列族是一个非常有效的操作。然而，列族是一个相当新的功能，我们还有另一个shard的用例。

将数据组织成shard意味着它可以在集群内移动，而不必检查数十亿个key。在撰写本文时，不可能将RocksDB中的列族移动到另一个。旧的碎片通常是冷写的，所以移动它们将会很便宜而且代价很小。我们将获得额外的好处是在keyspace中存在一个写冷的地方，所以之后进行的一致性检查会更容易。

将数据组织到shard中运行了一段时间，直到大量的数据进入InfluxDB。 LevelDB将数据分解成许多小文件。在单个进程中打开数十个数百个这些数据库，最终造成了一个大问题。有六个月或一年数据的用户将用尽文件句柄。这不是我们与大多数用户发现的，任何将数据库推到极限的人都会遇到这个问题，我们没有解决。打开的文件柄实在太多了。

BoltDB和mmap B+树

在与LevelDB及其变体一起挣扎了一年之后，我们决定转移到BoltDB，BoltDB是一个纯粹的Golang数据库，它受到LMDB的高度启发，这是一个用C编写的mmap B+ Tree数据库。它具有与LevelDB相同的API语义：keyspace有序地存储。我们的许多用户感到惊讶，我们自己发布的LevelDB变体与LMDB（mmap B+ Tree）的测试显示，RocksDB是表现最好的。

然而，在纯粹的写的表现之外，还有其他因素需要考虑进来。 在这一点上，我们最重要的目标是获得可以在生产和备份中运行的稳定的东西。BoltDB还具有以纯Go编写的优势，它极大地简化了我们的构建链，并使其易于构建在其他操作系统和平台上。

对我们来说，最大的好处是BoltDB使用单个文件作为数据库。在这一点上，我们之前最常见的bug报告来源是用户用尽了文件句柄。Bolt还同时解决了热备份问题和文件限制问题。

如果这意味着我们可以建立一个更可靠和稳定的系统，我们愿意对写入吞吐量上作出妥协。 我们的理由是，对于任何人想要真正大的写入负载，他们将会运行一个集群。我们根据BoltDB发布了0.9.0到0.9.2版本。从发展的角度来看，这是令人愉快的。 简洁的API，快速轻松地构建在我们的Go项目中，并且可靠。 然而，运行一段时间后，我们发现了写入吞吐量的一大问题。在数据库超过几GB之后，IOPS开始成为瓶颈。

有些用户可以通过将InfluxDB放在具有接近无限制IOPS的大硬件上，从而达到这个目标。但是，大多数用户都是云端资源有限的虚拟机。 我们必须找出一种方法来减少同时将一堆数据写入到成百上千个series的影响。

随着0.9.3和0.9.4版本的发布，我们的计划是在Bolt面前写一个WAL，这样我们可以减少随机插入到keyspace的数量。相反，我们会缓冲彼此相邻的多个写入，然后一次flush它们 但是，这仅仅是为了延缓了这个问题。高IOPS仍然成为一个问题，对于任何在适度工作负荷的场景下，它都会很快出现。

然而，我们在Bolt面前建立一个WAL实施的经验使我们有信心可以解决写入问题。WAL本身的表现太棒

了，索引根本无法跟上。在这一点上，我们再次开始思考如何创建类似于LSM Tree的东西，使之可以跟上我们的写入负载。

这就是TSM Tree的诞生过程。

写入协议

InfluxDB的行协议是一种写入数据点到InfluxDB的文本格式。

行协议

行协议的教程样式文档

行协议

InfluxDB的行协议是一种写入数据点到InfluxDB的文本格式。必须要是这样的格式的数据点才能被Influxdb解析和写入成功，当然除非你使用一些其他服务插件。

使用虚构的温度数据，本页面介绍了行协议。 它涵盖：

语法	数据类型	引号	特殊字符和关键字

最后一节，将数据写入InfluxDB，介绍如何将数据存入InfluxDB，以及InfluxDB如何处理行协议重复问题。

语法

一行Line Protocol表示InfluxDB中的一个数据点。它向InfluxDB通知点的measurement，tag set，field set和timestamp。以下代码块显示了行协议的示例，并将其分解为其各个组件：

```
1. weather,location=us-midwest temperature=82 1465839830100400200
2.   |    -------------------- --------------  |
3.   |             |             |             |
4.   |             |             |             |
5. +-----------+--------+-+---------+-+---------+
6. |measurement|,tag_set| |field_set| |timestamp|
7. +-----------+--------+-+---------+-+---------+
```

measurement

你想要写入数据的measurement，这在行协议中是必需的，例如这里的measurement是 `weather` 。

Tag set

你想要数据点中包含的tag，tag在行协议里是可选的。注意measurement和tag set是用不带空格的逗号分开的。

用不带空格的 `=` 来分割一组tag的键值：

```
1. <tag_key>=<tag_value>
```

多组tag直接用不带空格的逗号分开：

```
<tag_key>=<tag_value>,<tag_key>=<tag_value>
```

例如上面的tag set由一个tag组成 `location=us-midwest` ，现在加另一个tag(`season=summer`)，就变成了这样：

```
weather,location=us-midwest,season=summer temperature=82 1465839830100400200
```

为了获得最佳性能，您应该在将它们发送到数据库之前按键进行排序。排序应该与Go bytes.Compare function的结果相匹配。

空格1

分离measurement和field set，或者如果您使用数据点包含tag set，则使用空格分隔tag set和field set。行协议中空格是必需的。

没有tag set的有效行协议：

```
weather temperature=82 1465839830100400200
```

Field set

每个数据点在行协议中至少需要一个field。使用无空格的 `=` 分隔field的键值对：

```
<field_key>=<field_value>
```

多组field直接用不带空格的逗号分开：

```
<field_key>=<field_value>,<field_key>=<field_value>
```

例如上面的field set由一个field组成 `temperature=82` ，现在加另一个field(`bug_concentration=98`)，就变成了这样：

```
    weather,location=us-midwest temperature=82,bug_concentration=98
1.  1465839830100400200
```

空格2

使用空格分隔field set和可选的时间戳。如果你包含时间戳，则行协议中需要空格。

Timestamp

数据点的时间戳记以纳秒精度Unix时间。行协议中的时间戳是可选的。 如果没有为数据点指定时间戳，InfluxDB会使用服务器的本地纳秒时间戳。

在这个例子中，时间戳记是 `1465839830100400200` （这就是RFC6393格式的 `2016-06-13T17：43：50.1004002Z` ）。下面的行协议是相同的数据点，但没有时间戳。当InfluxDB将其写入数据库时，它将使用您的服务器的本地时间戳而不是 `2016-06-13T17：43：50.1004002Z` 。

```
1.  weather,location=us-midwest temperature=82
```

使用HTTP API来指定精度超过纳秒的时间戳，例如微秒，毫秒或秒。我们建议使用最粗糙的精度，因为这样可以显着提高压缩率。 有关详细信息，请参阅[API参考]。

> 小贴士：使用网络时间协议（NTP）来同步主机之间的时间。InfluxDB使用主机在UTC的本地时间为数据分配时间戳；如果主机的时钟与NTP不同步，写入InfluxDB的数据的时间戳可能不正确。

数据类型

本节介绍行协议的主要组件的数据类型：measurement，tag keys，tag values，field keys，field values和timestamp。

其中measurement，tag keys，tag values，field keys始终是字符串。

> 注意：因为InfluxDB将tag value存储为字符串，所以InfluxDB无法对tag value进行数学运算。此外，InfluxQL函数不接受tag value作为主要参数。 在设计架构时要考虑到这些信息。

Timestamps是UNIX时间戳。 最小有效时间戳为 `-9223372036854775806` 或 `1677-09-21T00：12：43.145224194Z` 。最大有效时间戳为 `9223372036854775806` 或 `2262-04-11T23：47：16.854775806Z` 。 如上所述，默认情况下，InfluxDB假定时间戳具有纳秒精度。有关如何指定替代精度，请参阅API参考。

Field value可以是整数、浮点数、字符串和布尔值：

- 浮点数 —— 默认是浮点数，InfluxDB假定收到的所有field value都是浮点数。以浮点类型存储上面的 `82` ：

  ```
  1.  weather,location=us-midwest temperature=82 1465839830100400200
  ```

- 整数 —— 添加一个 `i` 在field之后，告诉InfluxDB以整数类型存储：以整数类型存储上面的 `82` ：

  ```
  1.  weather,location=us-midwest temperature=82i 1465839830100400200
  ```

- 字符串 —— 双引号把字段值引起来表示字符串：以字符串类型存储值 `too warm` ：

```
1. weather,location=us-midwest temperature="too warm" 1465839830100400200
```

- 布尔型 —— 表示TRUE可以用 `t` ， `T` ， `true` ， `True` ， `TRUE` ；表示FALSE可以用 `f` ， `F` ， `false` ， `False` 或者 `FALSE` ：以布尔类型存储值 `true` ：

```
1. weather,location=us-midwest too_hot=true 1465839830100400200
```

> 注意：数据写入和数据查询可接受的布尔语法不同。 有关详细信息，请参阅常见问题。

在measurement中，field value的类型在分片内不会有差异，但在分片之间可能会有所不同。例如，如果InfluxDB尝试将整数写入到与浮点数相同的分片中，则写入会失败：

```
1. > INSERT weather,location=us-midwest temperature=82 1465839830100400200
2. > INSERT weather,location=us-midwest temperature=81i 1465839830100400300
   ERR: {"error":"field type conflict: input field \"temperature\" on measurement
3. \"weather\" is type int64, already exists as type float"}
```

但是，如果InfluxDB将整数写入到一个新的shard中，虽然之前写的是浮点数，那依然可以写成功：

```
1. > INSERT weather,location=us-midwest temperature=82 1465839830100400200
2. > INSERT weather,location=us-midwest temperature=81i 1467154750000000000
3. >
```

有关字段值类型差异如何影响 `SELECT *` 查询的，请参阅常见问题。

引号

本节涵盖何时不要和何时不要在行协议中使用双（ `"` ）或单（ `'` ）引号。

- 时间戳不要双或单引号。下面这是无效的行协议。 例：

```
1. > INSERT weather,location=us-midwest temperature=82 "1465839830100400200"
   ERR: {"error":"unable to parse 'weather,location=us-midwest temperature=82
2. \"1465839830100400200\"': bad timestamp"}
```

- field value不要单引号，即时是字符串类型。下面这是无效的行协议。 例：

```
1. > INSERT weather,location=us-midwest temperature='too warm'
```

```
    ERR: {"error":"unable to parse 'weather,location=us-midwest temperature='too
2.  warm'': invalid boolean"}
```

- measurement名称，tag keys，tag value和field key不用单双引号。InfluxDB会假定引号是名称的一部分。例如：

```
1.  > INSERT weather,location=us-midwest temperature=82 1465839830100400200
2.  > INSERT "weather",location=us-midwest temperature=87 1465839830100400200
3.  > SHOW MEASUREMENTS
4.  name: measurements
5.  ------------------
6.  name
7.  "weather"
8.  weather
```

查询数据中的 `"weather"` ，你需要为measurement名称中的引号转义：

```
1.  > SELECT * FROM "\"weather\""
2.  name: "weather"
3.  ---------------
4.  time                             location     temperature
5.  2016-06-13T17:43:50.1004002Z     us-midwest   87
```

- 当field value是整数，浮点数或是布尔型时，不要使用双引号，不然InfluxDB会假定值是字符串类型：

```
1.  > INSERT weather,location=us-midwest temperature="82"
2.  > SELECT * FROM weather WHERE temperature >= 70
3.  >
```

- 当Field value是字符串时，使用双引号：

```
1.  > INSERT weather,location=us-midwest temperature="too warm"
2.  > SELECT * FROM weather
3.  name: weather
4.  -------------
5.  time                             location     temperature
6.  2016-06-13T19:10:09.995766248Z   us-midwest   too warm
```

特殊字符和关键字

特殊字符

对于tag key, tag value和field key, 始终使用反斜杠字符\来进行转义:

- 逗号 `,` :

```
weather,location=us\,midwest temperature=82 1465839830100400200
```

- 等号 `=` :

```
weather,location=us-midwest temp\=rature=82 1465839830100400200
```

- 空格:

```
weather,location\ place=us-midwest temperature=82 1465839830100400200
```

对于measurement, 也要反斜杠\来转义。

- 逗号 `,` :

```
wea\,ther,location=us-midwest temperature=82 1465839830100400200
```

- 空格:

```
wea\ ther,location=us-midwest temperature=82 1465839830100400200
```

字符串类型的field value, 也要反斜杠\来转义。

- 双引号 `"` :

```
weather,location=us-midwest temperature="too\"hot\"" 1465839830100400200
```

行协议不要求用户转义反斜杠字符\。所有其他特殊字符也不需要转义。例如, 行协议处理emojis没有问题:

```
> INSERT we▫ther,location=us-midwest temperture=82 1465839830100400200
> SELECT * FROM "we▫ther"
name: we▫ther
------------------
time                            location       temperture
1465839830100400200     us-midwest     82
```

关键字

行协议接受InfluxQL关键字作为标识符名称。一般来说，我们建议避免在schema中使用InfluxQL关键字，因为它可能会在查询数据时引起混淆。

关键字 `time` 是特殊情况。 `time` 可以是cq的名称，数据库名称，measurement名称，RP名称，subscription名称和用户名 在这种情况下，查询 `time` 不需要双引号。 `time` 不能是field key或tag key; 当把 `time` 作为field key或是tag key写入时，InfluxDB会拒绝并返回错误。有关详细信息，请参阅常见问题。

写数据到InfluxDB

写入数据的方法

现在你知道所有关于行协议的信息，你如何在使用中用行协议写入数据到InfluxDB呢？在这里，我们将给出两个快速示例，然后可以到工具部分以获取更多信息。

HTTP API

使用HTTP API将数据写入InfluxDB。 向 `/write` 端点发送 `POST` 请求，并在请求主体中提供您的行协议:

```
    curl -i -XPOST "http://localhost:8086/write?db=science_is_cool" --data-binary
1.  'weather,location=us-midwest temperature=82 1465839830100400200'
```

有关查询字符串参数，状态码，响应和更多示例的深入描述，请参阅API参考。

CLI

使用InfluxDB的命令行界面（CLI）将数据写入InfluxDB。启动CLI，使用相关数据库，并将 `INSERT` 放在行协议之前:

```
1.  INSERT weather,location=us-midwest temperature=82 1465839830100400200
```

你还可以使用CLI从文件导入行协议。

还有几种将数据写入InfluxDB的方式。有关HTTP API，CLI和可用的服务插件（UDP，Graphite，CollectD和OpenTSDB）的更多信息，请参阅工具部分。

重复数据

一个点由measurement名称，tag set和timestamp唯一标识。如果您提交具有相同

measurement，tag set和timestamp，但具有不同field set的行协议，则field set将变为旧field set与新field set的合并，并且如果有任何冲突以新field set为准。

有关此行为的完整示例以及如何避免此问题，请参阅常见问题。

查询语言

本部分介绍InfluxQL，InfluxDB的SQL类查询语言，用于与InfluxDB中的数据进行交互。

InfluxQL教程

本部分的前七个文档提供了InfluxQL的教程式介绍。你可以随时下载文档相关的示例数据。

数据查询语法

涵盖InfluxQL的查询语言基础知识，包括 `SELECT` 语句， `GROUP BY` 子句， `INTO` 子句等。参阅数据查询还可以了解查询中的时间语法和正则表达式。

schema查询语法

涵盖schema相关的查询语法。有关InfluxQL的 `SHOW` 查询的语法说明和示例。

数据库管理

涵盖InfluxQL用于管理InfluxDB中的数据库和存储策略，具体有创建删除数据库和存储策略，以及删除数据。

函数

涵盖InfluxQL的函数。

Continuous Queries

涵盖Continuous Queries的基本语法，高级语法和常见用例。 此页面还介绍了如何进行 `SHOW` 和 `DROP` Continuous Queries。

数学计算

涵盖InfluxQL中的数学计算。

认证和授权

介绍如何设置身份验证和如何验证InfluxDB中的请求。此页面还描述了用于管理数据库用户的不同用户类型和InfluxQL。

InfluxQL参考

InfluxQL参考文档

数据查询语法

InfluxQL是一种类似SQL的查询语言，用于与InfluxDB中的数据进行交互。 以下部分详细介绍了InfluxQL的 `SELECT` 语句有关查询语法。

示例数据

本文使用国家海洋和大气管理局（NOAA）海洋作业和服务中心的公开数据。请参阅示例数据页面下载数据，并按照以下部分中的示例查询进行跟踪。开始之后，请随时了解 `h2o_feet` 这个measurement中的数据样本：

```
1.  name: h2o_feet
2.  --------------
3.  time                   level description      location      water_level
4.  2015-08-18T00:00:00Z    between 6 and 9 feet       coyote_creek    8.12
5.  2015-08-18T00:00:00Z    below 3 feet        santa_monica       2.064
6.  2015-08-18T00:06:00Z    between 6 and 9 feet      coyote_creek      8.005
7.  2015-08-18T00:06:00Z    below 3 feet        santa_monica       2.116
8.  2015-08-18T00:12:00Z    between 6 and 9 feet       coyote_creek    7.887
9.  2015-08-18T00:12:00Z    below 3 feet        santa_monica       2.028
```

`h2o_feet` 这个measurement中的数据以六分钟的时间间隔进行。measurement具有一个tag key(`location`)，它具有两个tag value: `coyote_creek` 和 `santa_monica` 。measurement还有两个field: `level description` 用字符串类型和 `water_level` 浮点型。所有这些数据都在 `NOAA_water_database` 数据库中。

> 声明: `level description` 字段不是原始NOAA数据的一部分——我们将其存储在那里，以便拥有一个带有特殊字符和字符串field value的field key。

基本的SELECT语句

`SELECT` 语句从特定的measurement中查询数据。 如果厌倦阅读，查看这个InfluxQL短片(注意：可能看不到，请到原文处查看*https://docs.influxdata.com/influxdb/v1.3/query_language/data_exploration/#syntax*):

语法

```
SELECT <field_key>[,<field_key>,<tag_key>] FROM <measurement_name>[,
<measurement_name>]
```

语法描述

`SELECT` 语句需要一个 `SELECT` 和 `FROM` 子句。

`SELECT` 子句

`SELECT` 支持指定数据的几种格式:

`SLECT *`

返回所有的field和tag。

`SELECT "<field_key>"`

返回特定的field。

`SELECT "<field_key>","<field_key>"`

返回多个field。

`SELECT "<field_key>","<tag_key>"`

返回特定的field和tag， `SELECT` 在包括一个tag时，必须至少指定一个field。

`SELECT "<field_key>"::field,"<tag_key>"::tag`

返回特定的field和tag， `::[field | tag]` 语法指定标识符的类型。 使用此语法来区分具有相同名称的field key和tag key。

`FROM` 子句

`FROM` 子句支持几种用于指定measurement的格式:

`FROM <measurement_name>`

从单个measurement返回数据。如果使用CLI需要先用 `USE` 指定数据库，并且使用的 `DEFAULT` 存储策略。如果您使用HTTP API,需要用 `db` 参数来指定数据库，也是使用 `DEFAULT` 存储策略。

`FROM <measurement_name>,<measurement_name>`

从多个measurement中返回数据。

`FROM <database_name>.<retention_policy_name>.<measurement_name>`

从一个完全指定的measurement中返回数据，这个完全指定是指指定了数据库和存储策略。

`FROM <database_name>..<measurement_name>`

从一个用户指定的数据库中返回存储策略为 `DEFAULT` 的数据。

引号

如果标识符包含除[A-z，0-9，_]之外的字符，如果它们以数字开头，或者如果它们是InfluxQL关键字，那么它们必须用双引号。虽然并不总是需要，我们建议您双引号标识符。

> 注意：查询的语法与行协议是不同的。

例子

例一：从单个measurement查询所有的field和tag

```
> SELECT * FROM "h2o_feet"

name: h2o_feet
--------------
time                   level description      location       water_level
2015-08-18T00:00:00Z   below 3 feet           santa_monica   2.064
2015-08-18T00:00:00Z   between 6 and 9 feet   coyote_creek   8.12
[...]
2015-09-18T21:36:00Z   between 3 and 6 feet   santa_monica   5.066
2015-09-18T21:42:00Z   between 3 and 6 feet   santa_monica   4.938
```

该查询从 `h2o_feet` measurement中选择所有field和tag。

如果您使用CLI，请确保在运行查询之前输入 `USE NOAA_water_database` 。CLI查询 `USE` 的数据库并且存储策略是 `DEFAULT` 的数据。如果使用HTTP API，请确保将 `db` 查询参数设置为 `NOAA_water_database` 。如果没有设置rp参数，则HTTP API会自动选择数据库的 `DEFAULT` 存储策略。

例二：从单个measurement中查询特定tag和field

```
 > SELECT "level description","location","water_level" FROM "h2o_feet"

name: h2o_feet
--------------
time                   level description      location       water_level
2015-08-18T00:00:00Z   below 3 feet           santa_monica   2.064
2015-08-18T00:00:00Z   between 6 and 9 feet   coyote_creek   8.12
[...]
2015-09-18T21:36:00Z   between 3 and 6 feet   santa_monica   5.066
2015-09-18T21:42:00Z   between 3 and 6 feet   santa_monica   4.938
```

该查询field `level descriptio` , tag `location` 和field `water_leval` 。 请注意， `SELECT` 子句在包含tag时必须至少指定一个field。

例三：从单个measurement中选择特定的tag和field，并提供其标识符类型

```
> SELECT "level description"::field,"location"::tag,"water_level"::field FROM "h2o_feet"

name: h2o_feet
--------------
time                   level description      location       water_level
2015-08-18T00:00:00Z   below 3 feet           santa_monica   2.064
2015-08-18T00:00:00Z   between 6 and 9 feet   coyote_creek   8.12
[...]
2015-09-18T21:36:00Z   between 3 and 6 feet   santa_monica   5.066
2015-09-18T21:42:00Z   between 3 and 6 feet   santa_monica   4.938
```

查询从measurement `h2o_feet` 中选择field `level description` , tag `location` 和field `water_leval` 。 `:: [field | tag]` 语法指定标识符是field还是tag。使用 `:: [field | tag]` 以区分相同的field key和tag key。大多数用例并不需要该语法。

例四：从单个measurement查询所有field

```
> SELECT *::field FROM "h2o_feet"

name: h2o_feet
--------------
time                   level description      water_level
2015-08-18T00:00:00Z   below 3 feet           2.064
2015-08-18T00:00:00Z   between 6 and 9 feet   8.12
[...]
2015-09-18T21:36:00Z   between 3 and 6 feet   5.066
2015-09-18T21:42:00Z   between 3 and 6 feet   4.938
```

该查询从measurement `h2o_feet` 中选择所有field。 `SELECT` 子句支持将 `*` 语法与 `::` 语法相结合。

例五：从measurement中选择一个特定的field并执行基本计算

```
> SELECT ("water_level" * 2) + 4 from "h2o_feet"

name: h2o_feet
--------------
```

```
time                   water_level
2015-08-18T00:00:00Z   20.24
2015-08-18T00:00:00Z   8.128
[...]
2015-09-18T21:36:00Z   14.132
2015-09-18T21:42:00Z   13.876
```

该查询将 `water_level` 字段值乘以2，并加上4。请注意，InfluxDB遵循标准操作顺序。

例六：从多个measurement中查询数据

```
> SELECT * FROM "h2o_feet","h2o_pH"

name: h2o_feet
--------------
time                   level description      location       pH   water_level
2015-08-18T00:00:00Z   below 3 feet           santa_monica        2.064
2015-08-18T00:00:00Z   between 6 and 9 feet   coyote_creek        8.12
[...]
2015-09-18T21:36:00Z   between 3 and 6 feet   santa_monica        5.066
2015-09-18T21:42:00Z   between 3 and 6 feet   santa_monica        4.938

name: h2o_pH
------------
time                   level description   location       pH   water_level
2015-08-18T00:00:00Z                       santa_monica   6
2015-08-18T00:00:00Z                       coyote_creek   7
[...]
2015-09-18T21:36:00Z                       santa_monica   8
2015-09-18T21:42:00Z                       santa_monica   7
```

该查询从两个measurement `h2o_feet` 和 `h2o_pH` 中查询所有的field和tag，多个measurement之间用逗号 `,` 分割。

例七：从完全限定的measurement中选择所有数据

```
> SELECT * FROM "NOAA_water_database"."autogen"."h2o_feet"

name: h2o_feet
--------------
time                   level description      location       water_level
2015-08-18T00:00:00Z   below 3 feet           santa_monica   2.064
2015-08-18T00:00:00Z   between 6 and 9 feet   coyote_creek   8.12
```

```
[...]
2015-09-18T21:36:00Z   between 3 and 6 feet   santa_monica   5.066
2015-09-18T21:42:00Z   between 3 and 6 feet   santa_monica   4.938
```

该查询选择数据库 `NOAA_water_database` 中的数据， `autogen` 为存储策略， `h2o_feet` 为measurement。

在CLI中，可以直接这样来代替 `USE` 指定数据库，以及指定 `DEFAULT` 之外的存储策略。 在HTTP API中，如果需要完全限定使用 `db` 和 `rp` 参数来指定。

例八：从特定数据库中查询measurement的所有数据

```
> SELECT * FROM "NOAA_water_database".."h2o_feet"

name: h2o_feet
--------------
time                   level description      location       water_level
2015-08-18T00:00:00Z   below 3 feet           santa_monica   2.064
2015-08-18T00:00:00Z   between 6 and 9 feet   coyote_creek   8.12
[...]
2015-09-18T21:36:00Z   between 3 and 6 feet   santa_monica   5.066
2015-09-18T21:42:00Z   between 3 and 6 feet   santa_monica   4.938
```

该查询选择数据库 `NOAA_water_database` 中的数据， `DEFAULT` 为存储策略和 `h2o_feet` 为measurement。 `..`表示指定数据库的 `DEFAULT` 存储策略。

SELECT语句中常见的问题

问题一：在SELECT语句中查询tag key

一个查询在 `SELECT` 子句中至少需要一个field key来返回数据。如果 `SELECT` 子句仅包含单个tag key或多个tag key，则查询返回一个空的结果。这是系统如何存储数据的结果。

例如：

下面的查询不会返回结果，因为在 `SELECT` 子句中只指定了一个tag key(`location`)：

```
> SELECT "location" FROM "h2o_feet"
>
```

要想有任何有关tag key为 `location` 的数据， `SELECT` 子句中必须至少有一个field(`water_level`)：

```
> SELECT "water_level","location" FROM "h2o_feet" LIMIT 3
name: h2o_feet
time                   water_level  location
----                   -----------  --------
2015-08-18T00:00:00Z   8.12         coyote_creek
2015-08-18T00:00:00Z   2.064        santa_monica
[...]
2015-09-18T21:36:00Z   5.066        santa_monica
2015-09-18T21:42:00Z   4.938        santa_monica
```

WHERE子句

`WHERE` 子句用作field，tag和timestamp的过滤。如果厌倦阅读，查看这个InfluxQL短片(注意：可能看不到，请到原文处查看*https://docs.influxdata.com/influxdb/v1.3/query_language/data_exploration/#the-where-clause*)：

语法

```
SELECT_clause FROM_clause WHERE <conditional_expression> [(AND|OR)
<conditional_expression> [...]]
```

语法描述

`WHERE` 子句在field，tag和timestamp上支持 `conditional_expressions` .

fields

```
field_key <operator> ['string' | boolean | float | integer]
```

`WHERE` 子句支持field value是字符串，布尔型，浮点数和整数这些类型。

在 `WHERE` 子句中单引号来表示字符串字段值。具有无引号字符串字段值或双引号字符串字段值的查询将不会返回任何数据，并且在大多数情况下也不会返回错误。

支持的操作符：

`=` 等于 `<>` 不等于 `!=` 不等于 `>` 大于 `>=` 大于等于 `<` 小于 `<=` 小于等于

tags

```
tag_key <operator> ['tag_value']
```

`WHERE` 子句中的用单引号来把tag value引起来。具有未用单引号的tag或双引号的tag查询将不会返回任何数据，并且在大多数情况下不会返回错误。

支持的操作符：

`=` 等于 `<>` 不等于 `!=` 不等于

timestamps

对于大多数 `SELECT` 语句，默认时间范围为UTC的 `1677-09-21 00：12：43.145224194` 到 `2262-04-11T23：47：16.854775806Z` 。 对于只有 `GROUP BY time()` 子句的 `SELECT` 语句，默认时间范围在UTC的 `1677-09-21 00：12：43.145224194` 和 `now()` 之间。

例子

例一：查询有特定field的key value的数据

```
1.  > SELECT * FROM "h2o_feet" WHERE "water_level" > 8
2.
3.  name: h2o_feet
4.  --------------
5.  time                   level description      location       water_level
6.  2015-08-18T00:00:00Z   between 6 and 9 feet   coyote_creek   8.12
7.  2015-08-18T00:06:00Z   between 6 and 9 feet   coyote_creek   8.005
8.  [...]
9.  2015-09-18T00:12:00Z   between 6 and 9 feet   coyote_creek   8.189
10. 2015-09-18T00:18:00Z   between 6 and 9 feet   coyote_creek   8.084
```

这个查询将会返回measurement为 `h2o_feet` ，字段 `water_level` 的值大于8的数据。

例二：查询有特定field的key value为字符串的数据

```
1.  > SELECT * FROM "h2o_feet" WHERE "level description" = 'below 3 feet'
2.
3.  name: h2o_feet
4.  --------------
5.  time                   level description   location       water_level
6.  2015-08-18T00:00:00Z   below 3 feet        santa_monica   2.064
7.  2015-08-18T00:06:00Z   below 3 feet        santa_monica   2.116
8.  [...]
9.  2015-09-18T14:06:00Z   below 3 feet        santa_monica   2.999
10. 2015-09-18T14:36:00Z   below 3 feet        santa_monica   2.907
```

该查询从 `h2o_feet` 返回数据，其中 `level description` 等于 `below 3 feet` 。InfluxQL

在 `WHERE` 子句中需要单引号来将字符串field value引起来。

例三：查询有特定field的key value并且带计算的数据

```
> SELECT * FROM "h2o_feet" WHERE "water_level" + 2 > 11.9

name: h2o_feet
--------------
time                   level description           location       water_level
2015-08-29T07:06:00Z   at or greater than 9 feet   coyote_creek   9.902
2015-08-29T07:12:00Z   at or greater than 9 feet   coyote_creek   9.938
2015-08-29T07:18:00Z   at or greater than 9 feet   coyote_creek   9.957
2015-08-29T07:24:00Z   at or greater than 9 feet   coyote_creek   9.964
2015-08-29T07:30:00Z   at or greater than 9 feet   coyote_creek   9.954
2015-08-29T07:36:00Z   at or greater than 9 feet   coyote_creek   9.941
2015-08-29T07:42:00Z   at or greater than 9 feet   coyote_creek   9.925
2015-08-29T07:48:00Z   at or greater than 9 feet   coyote_creek   9.902
2015-09-02T23:30:00Z   at or greater than 9 feet   coyote_creek   9.902
```

该查询从 `h2o_feet` 返回数据，其字段值为 `water_level` 加上2大于11.9。

例四：查询有特定tag的key value的数据

```
> SELECT "water_level" FROM "h2o_feet" WHERE "location" = 'santa_monica'

name: h2o_feet
--------------
time                   water_level
2015-08-18T00:00:00Z   2.064
2015-08-18T00:06:00Z   2.116
[...]
2015-09-18T21:36:00Z   5.066
2015-09-18T21:42:00Z   4.938
```

该查询从 `h2o_feet` 返回数据，其中tag `location` 为 `santa_monica` 。InfluxQL需要 `WHERE` 子句中tag的过滤带单引号。

例五：查询有特定tag的key value以及特定field的key value的数据

```
> SELECT "water_level" FROM "h2o_feet" WHERE "location" <> 'santa_monica' AND
(water_level < -0.59 OR water_level > 9.95)

name: h2o_feet
```

```
4.  --------------
5.  time                   water_level
6.  2015-08-29T07:18:00Z   9.957
7.  2015-08-29T07:24:00Z   9.964
8.  2015-08-29T07:30:00Z   9.954
9.  2015-08-29T14:30:00Z   -0.61
10. 2015-08-29T14:36:00Z   -0.591
11. 2015-08-30T15:18:00Z   -0.594
```

该查询从 `h2o_feet` 中返回数据，其中tag `location` 设置为 `santa_monica` ，并且field `water_level` 的值小于-0.59或大于9.95。 `WHERE` 子句支持运算符 `AND` 和 `O` R，并支持用括号分隔逻辑。

例六：根据时间戳来过滤数据

```
1.  > SELECT * FROM "h2o_feet" WHERE time > now() - 7d
```

该查询返回来自 `h2o_feet` ，该measurement在过去七天内的数据。

WHERE子句常见的问题

问题一：WHERE子句返回结果为空

在大多数情况下，这个问题是tag value或field value缺少单引号的结果。具有无引号或双引号tag value或field value的查询将不会返回任何数据，并且在大多数情况下不会返回错误。

下面的代码块中的前两个查询尝试指定tag value为 `santa_monica` ，没有任何引号和双引号。那些查询不会返回结果。 第三个查询单引号 `santa_monica` （这是支持的语法），并返回预期的结果。

```
1.  > SELECT "water_level" FROM "h2o_feet" WHERE "location" = santa_monica
2.
3.  > SELECT "water_level" FROM "h2o_feet" WHERE "location" = "santa_monica"
4.
5.  > SELECT "water_level" FROM "h2o_feet" WHERE "location" = 'santa_monica'
6.
7.  name: h2o_feet
8.  --------------
9.  time                   water_level
10. 2015-08-18T00:00:00Z   2.064
11. [...]
12. 2015-09-18T21:42:00Z   4.938
```

下面的代码块中的前两个查询尝试指定field字符串为 `at or greater than 9 feet` ，没有任何引号和双引号。第一个查询返回错误，因为field字符串包含空格。 第二个查询不返回结果。 第三个查询单引号 `at or greater than 9 feet` （这是支持的语法），并返回预期结果。

```
> SELECT "level description" FROM "h2o_feet" WHERE "level description" = at or
greater than 9 feet

ERR: error parsing query: found than, expected ; at line 1, char 86

> SELECT "level description" FROM "h2o_feet" WHERE "level description" = "at or
greater than 9 feet"

> SELECT "level description" FROM "h2o_feet" WHERE "level description" = 'at or
greater than 9 feet'

name: h2o_feet
--------------
time                   level description
2015-08-26T04:00:00Z   at or greater than 9 feet
[...]
2015-09-15T22:42:00Z   at or greater than 9 feet
```

GROUP BY子句

`GROUP BY` 子句后面可以跟用户指定的tags或者是一个时间间隔。

GROUP BY tags

`GROUP BY <tag>` 后面跟用户指定的tags。如果厌倦阅读，查看这个InfluxQL短片(注意：可能看不到，请到原文处查看*https://docs.influxdata.com/influxdb/v1.3/query_language/data_exploration/#group-by-tags*):

语法

```
SELECT_clause FROM_clause [WHERE_clause] GROUP BY [* | <tag_key>[,<tag_key]]
```

语法描述

`GROUP BY *` 对结果中的所有tag作group by。

`GROUP BY <tag_key>` 对结果按指定的tag作group by。

`GROUP BY <tag_key>,<tag_key>` 对结果数据按多个tag作group by，其中tag key的顺序没所谓。

例子

例一：对单个tag作group by

```
> SELECT MEAN("water_level") FROM "h2o_feet" GROUP BY "location"

name: h2o_feet
tags: location=coyote_creek
time                           mean
----                           ----
1970-01-01T00:00:00Z     5.359342451341401


name: h2o_feet
tags: location=santa_monica
time                           mean
----                           ----
1970-01-01T00:00:00Z     3.530863470081006
```

上面的查询中用到了InfluxQL中的函数来计算measurement `h2o_feet` 的每 `location` 的 `water_level` 的平均值。InfluxDB返回了两个series：分别是 `location` 的两个值。

> 说明：在InfluxDB中，epoch 0(`1970-01-01T00:00:00Z`)通常用作等效的空时间戳。如果要求查询不返回时间戳，例如无限时间范围的聚合函数，InfluxDB将返回epoch 0作为时间戳。

例二：对多个tag作group by

```
> SELECT MEAN("index") FROM "h2o_quality" GROUP BY location,randtag

name: h2o_quality
tags: location=coyote_creek, randtag=1
time                   mean
----                   ----
1970-01-01T00:00:00Z  50.69033760186263

name: h2o_quality
tags: location=coyote_creek, randtag=2
```

```
time                    mean
----                    ----
1970-01-01T00:00:00Z    49.661867544220485

name: h2o_quality
tags: location=coyote_creek, randtag=3
time                    mean
----                    ----
1970-01-01T00:00:00Z    49.360939907550076

name: h2o_quality
tags: location=santa_monica, randtag=1
time                    mean
----                    ----
1970-01-01T00:00:00Z    49.132712456344585

name: h2o_quality
tags: location=santa_monica, randtag=2
time                    mean
----                    ----
1970-01-01T00:00:00Z    50.2937984496124

name: h2o_quality
tags: location=santa_monica, randtag=3
time                    mean
----                    ----
1970-01-01T00:00:00Z    49.99919903884662
```

上面的查询中用到了InfluxQL中的函数来计算measurement `h2o_quality` 的每个 `location` 和 `randtag` 的 `Index` 的平均值。在 `GROUP BY` 子句中用逗号来分割多个tag。

例三：对所有tag作group by

```
> SELECT MEAN("index") FROM "h2o_quality" GROUP BY *

name: h2o_quality
tags: location=coyote_creek, randtag=1
time                        mean
----                        ----
1970-01-01T00:00:00Z    50.55405446521169


name: h2o_quality
```

```
tags: location=coyote_creek, randtag=2
time                           mean
----                           ----
1970-01-01T00:00:00Z     50.49958856271162


name: h2o_quality
tags: location=coyote_creek, randtag=3
time                           mean
----                           ----
1970-01-01T00:00:00Z     49.5164137518956


name: h2o_quality
tags: location=santa_monica, randtag=1
time                           mean
----                           ----
1970-01-01T00:00:00Z     50.43829082296367


name: h2o_quality
tags: location=santa_monica, randtag=2
time                           mean
----                           ----
1970-01-01T00:00:00Z     52.0688508894012


name: h2o_quality
tags: location=santa_monica, randtag=3
time                           mean
----                           ----
1970-01-01T00:00:00Z     49.29386362086556
```

上面的查询中用到了InfluxQL中的函数来计算measurement `h2o_quality` 的每个tag 的 `Index` 的平均值。

请注意，查询结果与例二中的查询结果相同，其中我们明确指定了tag key `location` 和 `randtag` 。 这是因为measurement `h2o_quality` 中只有这两个tag key。

GROUP BY时间间隔

`GROUP BY time()` 返回结果按指定的时间间隔group by。

基本的GROUP BY time()语法

语法

```
SELECT <function>(<field_key>) FROM_clause WHERE <time_range> GROUP BY
time(<time_interval>),[tag_key] [fill(<fill_option>)]
```

基本语法描述

基本 `GROUP BY time()` 查询需要 `SELECT` 子句中的InfluxQL函数和 `WHERE` 子句中的时间范围。请注意， `GROUP BY` 子句必须在 `WHERE` 子句之后。

`time(time_interval)` `GROUP BY time()` 语句中的 `time_interval` 是一个时间duration。决定了InfluxDB按什么时间间隔group by。例如： `time_interval` 为 `5m` 则在 `WHERE` 子句中指定的时间范围内将查询结果分到五分钟时间组里。

`fill(<fill_option>)` `fill（<fill_option>）` 是可选的。它会更改不含数据的时间间隔的返回值。

覆盖范围：基本 `GROUP BY time()` 查询依赖于 `time_interval` 和InfluxDB的预设时间边界来确定每个时间间隔中包含的原始数据以及查询返回的时间戳。

基本语法示例

下面的例子用到的示例数据如下：

```
> SELECT "water_level","location" FROM "h2o_feet" WHERE time >= '2015-08-
18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z'

name: h2o_feet
--------------
time                   water_level   location
2015-08-18T00:00:00Z   8.12          coyote_creek
2015-08-18T00:00:00Z   2.064         santa_monica
2015-08-18T00:06:00Z   8.005         coyote_creek
2015-08-18T00:06:00Z   2.116         santa_monica
2015-08-18T00:12:00Z   7.887         coyote_creek
2015-08-18T00:12:00Z   2.028         santa_monica
2015-08-18T00:18:00Z   7.762         coyote_creek
2015-08-18T00:18:00Z   2.126         santa_monica
2015-08-18T00:24:00Z   7.635         coyote_creek
2015-08-18T00:24:00Z   2.041         santa_monica
2015-08-18T00:30:00Z   7.5           coyote_creek
2015-08-18T00:30:00Z   2.051         santa_monica
```

例一：时间间隔为12分钟的group by

```
> SELECT COUNT("water_level") FROM "h2o_feet" WHERE "location"='coyote_creek' AND time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z' GROUP BY time(12m)

name: h2o_feet
--------------
time                   count
2015-08-18T00:00:00Z   2
2015-08-18T00:12:00Z   2
2015-08-18T00:24:00Z   2
```

该查询使用InfluxQL函数来计算 `location=coyote_creek` 的 `water_level` 数，并将其分组结果分为12分钟间隔。每个时间戳的结果代表一个12分钟的间隔。 第一个时间戳记的计数涵盖大于 `2015-08-18T00：00：00Z` 的原始数据，但小于且不包括 `2015-08-18T00：12：00Z` 。第二时间戳的计数涵盖大于 `2015-08-18T00：12：00Z` 的原始数据，但小于且不包括 `2015-08-18T00：24：00Z` 。

例二：时间间隔为12分钟并且还对tag key作group by

```
> SELECT COUNT("water_level") FROM "h2o_feet" WHERE time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z' GROUP BY time(12m),"location"

name: h2o_feet
tags: location=coyote_creek
time                   count
----                   -----
2015-08-18T00:00:00Z   2
2015-08-18T00:12:00Z   2
2015-08-18T00:24:00Z   2

name: h2o_feet
tags: location=santa_monica
time                   count
----                   -----
2015-08-18T00:00:00Z   2
2015-08-18T00:12:00Z   2
2015-08-18T00:24:00Z   2
```

该查询使用InfluxQL函数来计算 `water_leval` 的数量。它将结果按 `location` 分组并分隔12分钟。请注意，时间间隔和tag key在 `GROUP BY` 子句中以逗号分隔。查询返回两个measurement的

结果：针对tag `location` 的每个值。每个时间戳的结果代表一个12分钟的间隔。第一个时间戳记的计数涵盖大于 `2015-08-18T00：00：00Z` 的原始数据，但小于且不包括 `2015-08-18T00：12：00Z` 。第二时间戳的计数涵盖大于 `2015-08-18T00：12：00Z` 原始数据，但小于且不包括 `2015-08-18T00：24：00Z` 。

基本语法的共同问题

在查询结果中出现意想不到的时间戳和值

使用基本语法，InfluxDB依赖于 `GROUP BY time()` 间隔和系统预设时间边界来确定每个时间间隔中包含的原始数据以及查询返回的时间戳。 在某些情况下，这可能会导致意想不到的结果。

原始值：

```
> SELECT "water_level" FROM "h2o_feet" WHERE "location"='coyote_creek' AND time
>= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:18:00Z'
name: h2o_feet
--------------
time                   water_level
2015-08-18T00:00:00Z   8.12
2015-08-18T00:06:00Z   8.005
2015-08-18T00:12:00Z   7.887
2015-08-18T00:18:00Z   7.762
```

查询和结果：

以下查询涵盖12分钟的时间范围，并将结果分组为12分钟的时间间隔，但返回两个结果：

```
> SELECT COUNT("water_level") FROM "h2o_feet" WHERE "location"='coyote_creek'
AND time >= '2015-08-18T00:06:00Z' AND time < '2015-08-18T00:18:00Z' GROUP BY
time(12m)

name: h2o_feet
time                   count
----                   -----
2015-08-18T00:00:00Z   1        <----- 请注意，此时间戳记的发生在查询时间范围最小值之前
2015-08-18T00:12:00Z   1
```

解释：

InfluxDB使用独立于 `WHERE` 子句中任何时间条件的 `GROUP BY` 间隔的预设的四舍五入时间边界。当计算结果时，所有返回的数据必须在查询的显式时间范围内发生，但 `GROUP BY` 间隔将基于预设的时间边界。

下表显示了结果中预设时间边界，相关 `GROUP BY time()` 间隔，包含的点以及每个 `GROUP BY time()` 间隔的返回时间戳。

时间间隔序号	预设的时间边界	`GROUP BY time()` 间隔	包含的数据点	返回的时间戳
1	`time >= 2015-08-18T00:00:00Z AND time < 2015-08-18T00:12:00Z`	`time >= 2015-08-18T00:06:00Z AND time < 2015-08-18T00:12:00Z`	`8.005`	`2015-08-18T00:00:00Z`
2	`time >= 2015-08-18T00:12:00Z AND time < 2015-08-18T00:24:00Z`	`time >= 2015-08-18T00:12:00Z AND time < 2015-08-18T00:18:00Z`	`7.887`	`2015-08-18T00:12:00Z`

第一个预设的12分钟时间边界从0 `0:00` 开始，在 `00:12` 之前结束。只有一个数据点（ `8.005` ）落在查询的第一个 `GROUP BY time()` 间隔内，并且在第一个时间边界。请注意，虽然返回的时间戳在查询的时间范围开始之前发生，但查询结果排除了查询时间范围之前发生的数据。

第二个预设的12分钟时间边界从 `00:12` 开始，在 `00:24` 之前结束。 只有一个原点（ `7.887` ）都在查询的第二个 `GROUP BY time()` 间隔内，在该第二个时间边界内。

高级 `GROUP BY time()` 语法允许用户移动InfluxDB预设时间边界的开始时间。高级语法部分中的例三将继续显示此处的查询； 它将预设时间边界向前移动六分钟，以便InfluxDB返回：

```
1. name: h2o_feet
2. time                  count
3. ----                  -----
4. 2015-08-18T00:06:00Z  2
```

高级 `GROUP BY time()` 语法

语法

```
    SELECT <function>(<field_key>) FROM_clause WHERE <time_range> GROUP BY
1.  time(<time_interval>,<offset_interval>),[tag_key] [fill(<fill_option>)]
```

高级语法描述

高级 `GROUP BY time()` 查询需要 `SELECT` 子句中的InfluxQL函数和 `WHERE` 子句中的时间范围。 请注意， `GROUP BY` 子句必须在 `WHERE` 子句之后。

`time(time_interval,offset_interval)`

`offset_interval` 是一个持续时间。它向前或向后移动InfluxDB的预设时间界限。 `offset_interval` 可以为正或负。

`fill(<fill_option>)`

`fill(<fill_option>)` 是可选的。它会更改不含数据的时间间隔的返回值。

范围

高级 `GROUP BY time()` 查询依赖于 `time_interval` ， `offset_interval` 和InfluxDB的预设时间边界，以确定每个时间间隔中包含的原始数据以及查询返回的时间戳。

高级语法的例子

下面例子都使用这份示例数据：

```
> SELECT "water_level" FROM "h2o_feet" WHERE "location"='coyote_creek' AND time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:54:00Z'

name: h2o_feet
--------------
time                   water_level
2015-08-18T00:00:00Z   8.12
2015-08-18T00:06:00Z   8.005
2015-08-18T00:12:00Z   7.887
2015-08-18T00:18:00Z   7.762
2015-08-18T00:24:00Z   7.635
2015-08-18T00:30:00Z   7.5
2015-08-18T00:36:00Z   7.372
2015-08-18T00:42:00Z   7.234
2015-08-18T00:48:00Z   7.11
2015-08-18T00:54:00Z   6.982
```

例一：查询结果间隔按18分钟group by，并将预设时间边界向前移动

```
> SELECT MEAN("water_level") FROM "h2o_feet" WHERE "location"='coyote_creek' AND time >= '2015-08-18T00:06:00Z' AND time <= '2015-08-18T00:54:00Z' GROUP BY time(18m,6m)

name: h2o_feet
time                   mean
----                   ----
2015-08-18T00:06:00Z   7.884666666666667
2015-08-18T00:24:00Z   7.502333333333333
2015-08-18T00:42:00Z   7.108666666666667
```

该查询使用InfluxQL函数来计算平均 `water_level` ，将结果分组为18分钟的时间间隔，并将预设时间边界偏移六分钟。

没有 `offset_interval` 的查询的时间边界和返回的时间戳符合InfluxDB的预设时间边界。我们先来看看没有 `offset_interval` 的结果：

```
> SELECT MEAN("water_level") FROM "h2o_feet" WHERE "location"='coyote_creek'
AND time >= '2015-08-18T00:06:00Z' AND time <= '2015-08-18T00:54:00Z' GROUP BY
time(18m)

name: h2o_feet
time                   mean
----                   ----
2015-08-18T00:00:00Z   7.946
2015-08-18T00:18:00Z   7.6323333333333325
2015-08-18T00:36:00Z   7.238666666666667
2015-08-18T00:54:00Z   6.982
```

没有 `offset_interval` 的查询的时间边界和返回的时间戳符合InfluxDB的预设时间界限：

时间间隔序号	预设的时间边界	`GROUP BY time()` 间隔	包含的数据点	返回的时间戳
1	`time >= 2015-08-18T00:00:00Z AND time < 2015-08-18T00:18:00Z`	`time >= 2015-08-18T00:06:00Z AND time < 2015-08-18T00:18:00Z`	`8.005,7.887`	`2015-08-18T00:00:00Z`
2	`time >= 2015-08-18T00:18:00Z AND time < 2015-08-18T00:36:00Z`	同坐	`7.762,7.635,7.5`	`2015-08-18T00:18:00Z`
3	`time >= 2015-08-18T00:36:00Z AND time < 2015-08-18T00:54:00Z`	同左	`7.372,7.234,7.11`	`2015-08-18T00:36:00Z`
4	`time >= 2015-08-18T00:54:00Z AND time < 2015-08-18T01:12:00Z`	`time = 2015-08-18T00:54:00Z`	`6.982`	`2015-08-18T00:54:00Z`

第一个预设的18分钟时间边界从 `00:00` 开始，在 `00:18` 之前结束。 两个点（ `8.005和7.887` ）都落在第一个 `GROUP BY time()` 间隔内，并且在第一个时间边界。请注意，虽然返回的时间戳在查询的时间范围开始之前发生，但查询结果排除了查询时间范围之前发生的数据。

第二个预设的18分钟时间边界从 `00:18` 开始，在 `00:36` 之前结束。 三个点（ `7.762和7.635和7.5` ）都落在第二个 `GROUP BY time()` 间隔内，在第二个时间边界。 在这种情况下，边界时间范围和间隔时间范围是相同的。

第四个预设的18分钟时间边界从 `00:54` 开始，在 `1:12:00` 之前结束。 一个点（ `6.982` ）落在第四个 `GROUP BY time()` 间隔内，在第四个时间边界。

具有 `offset_interval` 的查询的时间边界和返回的时间戳符合偏移时间边界：

时间间隔序号	预设的时间边界	GROUP BY time() 间隔	包含的数据点	返回的时间戳
1	time >= 2015-08-18T00:06:00Z AND time < 2015-08-18T00:24:00Z	同左	8.005,7.887,7.762	2015-08-18T00:06:00Z
2	time >= 2015-08-18T00:24:00Z AND time < 2015-08-18T00:42:00Z	同坐	7.635,7.5,7.372	2015-08-18T00:24:00Z
3	time >= 2015-08-18T00:42:00Z AND time < 2015-08-18T01:00:00Z	同左	7.234,7.11,6.982	2015-08-18T00:42:00Z
4	time >= 2015-08-18T01:00:00Z AND time < 2015-08-18T01:18:00Z	无	无	无

六分钟偏移间隔向前移动预设边界的时间范围，使得边界时间范围和相关 `GROUP BY time()` 间隔时间范围始终相同。使用偏移量，每个间隔对三个点执行计算，返回的时间戳与边界时间范围的开始和 `GROUP BY time()` 间隔时间范围的开始匹配。

请注意， `offset_interval` 强制第四个时间边界超出查询的时间范围，因此查询不会返回该最后一个间隔的结果。

例二：查询结果按12分钟间隔group by，并将预设时间界限向后移动

```
> SELECT MEAN("water_level") FROM "h2o_feet" WHERE "location"='coyote_creek'
AND time >= '2015-08-18T00:06:00Z' AND time <= '2015-08-18T00:54:00Z' GROUP BY
time(18m,-12m)

name: h2o_feet
time                   mean
----                   ----
2015-08-18T00:06:00Z   7.884666666666667
2015-08-18T00:24:00Z   7.502333333333333
2015-08-18T00:42:00Z   7.108666666666667
```

该查询使用InfluxQL函数来计算平均 `water_level` ，将结果分组为18分钟的时间间隔，并将预设时间边界偏移-12分钟。

> 注意：例二中的查询返回与例一中的查询相同的结果，但例二中的查询使用负的 `offset_interval` 而不是正的 `offset_interval` 。 两个查询之间没有性能差异；在确定正负 `offset_intervel` 之间时，请任意选择最直观的选项。

没有 `offset_interval` 的查询的时间边界和返回的时间戳符合InfluxDB的预设时间边界。 我们首先检查没有偏移量的结果：

```
> SELECT MEAN("water_level") FROM "h2o_feet" WHERE "location"='coyote_creek'
AND time >= '2015-08-18T00:06:00Z' AND time <= '2015-08-18T00:54:00Z' GROUP BY
time(18m)

```

```
name: h2o_feet
time                   mean
----                   ----
2015-08-18T00:00:00Z   7.946
2015-08-18T00:18:00Z   7.6323333333333325
2015-08-18T00:36:00Z   7.238666666666667
2015-08-18T00:54:00Z   6.982
```

没有 `offset_interval` 的查询的时间边界和返回的时间戳符合InfluxDB的预设时间界限：

时间间隔序号	预设的时间边界	`GROUP BY time()` 间隔	包含的数据点	返回的时间戳
1	`time >= 2015-08-18T00:00:00Z AND time < 2015-08-18T00:18:00Z`	`time >= 2015-08-18T00:06:00Z AND time < 2015-08-18T00:18:00Z`	`8.005,7.887`	`2015-08-18T00:00:00Z`
2	`time >= 2015-08-18T00:18:00Z AND time < 2015-08-18T00:36:00Z`	同坐	`7.762,7.635,7.5`	`2015-08-18T00:18:00Z`
3	`time >= 2015-08-18T00:36:00Z AND time < 2015-08-18T00:54:00Z`	同左	`7.372,7.234,7.11`	`2015-08-18T00:36:00Z`
4	`time >= 2015-08-18T00:54:00Z AND time < 2015-08-18T01:12:00Z`	`time = 2015-08-18T00:54:00Z`	`6.982`	`2015-08-18T00:54:00Z`

第一个预设的18分钟时间边界从 `00:00` 开始，在 `00:18` 之前结束。 两个点（ `8.005和7.887` ）都落在第一个 `GROUP BY time()` 间隔内，并且在第一个时间边界。请注意，虽然返回的时间戳在查询的时间范围开始之前发生，但查询结果排除了查询时间范围之前发生的数据。

第二个预设的18分钟时间边界从 `00:18` 开始，在 `00:36` 之前结束。 三个点（ `7.762和7.635和7.5` ）都落在第二个 `GROUP BY time()` 间隔内，在第二个时间边界。 在这种情况下，边界时间范围和间隔时间范围是相同的。

第四个预设的18分钟时间边界从 `00:54` 开始，在 `1:12:00` 之前结束。 一个点（ `6.982` ）落在第四个 `GROUP BY time()` 间隔内，在第四个时间边界。

具有 `offset_interval` 的查询的时间边界和返回的时间戳符合偏移时间边界：

时间间隔序号	预设的时间边界	`GROUP BY time()` 间隔	包含的数据点	返回的时间戳
1	`time >= 2015-08-17T23:48:00Z AND time < 2015-08-18T00:06:00Z`	无	无	无
2	`time >= 2015-08-18T00:06:00Z AND time < 2015-08-18T00:24:00Z`	同左	`8.005,7.887,7.762`	`2015-08-18T00:06:00Z`
	`time >= 2015-08-18T00:24:00Z`	同坐	`7.635,7.5,7.372`	`2015-08-`

3	`AND time < 2015-08-18T00:42:00Z`	同坐	`7.635,7.5,7.372`	`18T00:24:00Z`
4	`time >= 2015-08-18T00:42:00Z AND time < 2015-08-18T01:00:00Z`	同左	`7.234,7.11,6.982`	`2015-08-18T00:42:00Z`

负十二分钟偏移间隔向后移动预设边界的时间范围，使得边界时间范围和相关 `GROUP BY time()` 间隔时间范围始终相同。使用偏移量，每个间隔对三个点执行计算，返回的时间戳与边界时间范围的开始和 `GROUP BY time()` 间隔时间范围的开始匹配。

请注意， `offset_interval` 强制第一个时间边界超出查询的时间范围，因此查询不会返回该最后一个间隔的结果。

例三：查询结果按12分钟间隔group by，并将预设时间边界向前移动

这个例子是上面基本语法的问题的继续

```
> SELECT COUNT("water_level") FROM "h2o_feet" WHERE "location"='coyote_creek'
AND time >= '2015-08-18T00:06:00Z' AND time < '2015-08-18T00:18:00Z' GROUP BY
time(12m,6m)

name: h2o_feet
time                   count
----                   -----
2015-08-18T00:06:00Z   2
```

该查询使用InfluxQL函数来计算平均 `water_level` ，将结果分组为12分钟的时间间隔，并将预设时间边界偏移六分钟。

没有 `offset_interval` 的查询的时间边界和返回的时间戳符合InfluxDB的预设时间边界。我们先来看看没有 `offset_interval` 的结果：

```
> SELECT COUNT("water_level") FROM "h2o_feet" WHERE "location"='coyote_creek'
AND time >= '2015-08-18T00:06:00Z' AND time < '2015-08-18T00:18:00Z' GROUP BY
time(12m)

name: h2o_feet
time                   count
----                   -----
2015-08-18T00:00:00Z   1
2015-08-18T00:12:00Z   1
```

没有 `offset_interval` 的查询的时间边界和返回的时间戳符合InfluxDB的预设时间界限：

时间间隔	预设的时间边界	`GROUP BY time()` 间隔	包含的	返回的时间戳

1	`time >= 2015-08-18T00:00:00Z AND time < 2015-08-18T00:12:00Z`	`time >= 2015-08-18T00:06:00Z AND time < 2015-08-18T00:12:00Z`	`8.005`	`2015-08-18T00:00:00Z`
2	`time >= 2015-08-12T00:12:00Z AND time < 2015-08-18T00:24:00Z`	`time >= 2015-08-12T00:12:00Z AND time < 2015-08-18T00:18:00Z`	`7.887`	`2015-08-18T00:12:00Z`

第一个预设的12分钟时间边界从0 `0:00` 开始，在 `00:12` 之前结束。只有一个数据点（ `8.005` ）落在查询的第一个 `GROUP BY time()` 间隔内，并且在第一个时间边界。请注意，虽然返回的时间戳在查询的时间范围开始之前发生，但查询结果排除了查询时间范围之前发生的数据。

第二个预设的12分钟时间边界从 `00:12` 开始，在 `00:24` 之前结束。 只有一个原点（ `7.887` ）都在查询的第二个 `GROUP BY time()` 间隔内，在该第二个时间边界内。

具有 `offset_interval` 的查询的时间边界和返回的时间戳符合偏移时间边界：

时间间隔序号	预设的时间边界	`GROUP BY time()` 间隔	包含的数据点	返回的时间戳
1	`time >= 2015-08-18T00:06:00Z AND time < 2015-08-18T00:18:00Z`	同左	`8.005, 7.887`	`2015-08-18T00:06:00Z`
2	`time >= 2015-08-18T00:18:00Z AND time < 2015-08-18T00:30:00Z`	无	无	无

六分钟偏移间隔向前移动预设边界的时间范围，使得边界时间范围和相关 `GROUP BY time()` 间隔时间范围始终相同。使用偏移量，每个间隔对三个点执行计算，返回的时间戳与边界时间范围的开始和 `GROUP BY time()` 间隔时间范围的开始匹配。

请注意， `offset_interval` 强制第二个时间边界超出查询的时间范围，因此查询不会返回该最后一个间隔的结果。

GROUP BY time()加fill()

`fill()` 更改不含数据的时间间隔的返回值。

语法

```
SELECT <function>(<field_key>) FROM_clause WHERE <time_range> GROUP BY
time(time_interval,[<offset_interval])[,tag_key] [fill(<fill_option>)]
```

语法描述

默认情况下，没有数据的 `GROUP BY time()` 间隔返回为null作为输出列中的值。 `fill()` 更改不含数据的时间间隔返回的值。请注意，如果 `GROUP(ing)BY` 多个对象（例如，tag和时间间隔），那么 `fill()` 必须位于 `GROUP BY` 子句的末尾。

fill的参数

- 任一数值：用这个数字返回没有数据点的时间间隔

- 任一数值：用这个数字返回没有数据点的时间间隔
- linear：返回没有数据的时间间隔的线性插值结果。
- none：不返回在时间间隔里没有点的数据
- previous：返回时间隔间的前一个间隔的数据

例子：

例一：fill(100)

不带 `fill(100)` ：

```
> SELECT MAX("water_level") FROM "h2o_feet" WHERE "location"='coyote_creek' AND time >= '2015-09-18T16:00:00Z' AND time <= '2015-09-18T16:42:00Z' GROUP BY time(12m)

name: h2o_feet
--------------
time                   max
2015-09-18T16:00:00Z   3.599
2015-09-18T16:12:00Z   3.402
2015-09-18T16:24:00Z   3.235
2015-09-18T16:36:00Z
```

带 `fill(100)` ：

```
> SELECT MAX("water_level") FROM "h2o_feet" WHERE "location"='coyote_creek' AND time >= '2015-09-18T16:00:00Z' AND time <= '2015-09-18T16:42:00Z' GROUP BY time(12m) fill(100)

name: h2o_feet
--------------
time                   max
2015-09-18T16:00:00Z   3.599
2015-09-18T16:12:00Z   3.402
2015-09-18T16:24:00Z   3.235
2015-09-18T16:36:00Z   100
```

例二：fill(linear)

不带 `fill(linear)` ：

```
> SELECT MEAN("tadpoles") FROM "pond" WHERE time >= '2016-11-11T21:00:00Z' AND time <= '2016-11-11T22:06:00Z' GROUP BY time(12m)

```

```
name: pond
time                   mean
----                   ----
2016-11-11T21:00:00Z   1
2016-11-11T21:12:00Z
2016-11-11T21:24:00Z   3
2016-11-11T21:36:00Z
2016-11-11T21:48:00Z
2016-11-11T22:00:00Z   6
```

带 `fill(linear)` :

```
> SELECT MEAN("tadpoles") FROM "pond" WHERE time >= '2016-11-11T21:00:00Z' AND time <= '2016-11-11T22:06:00Z' GROUP BY time(12m) fill(linear)

name: pond
time                   mean
----                   ----
2016-11-11T21:00:00Z   1
2016-11-11T21:12:00Z   2
2016-11-11T21:24:00Z   3
2016-11-11T21:36:00Z   4
2016-11-11T21:48:00Z   5
2016-11-11T22:00:00Z   6
```

例三: fill(none)

不带 `fill(none)` :

```
> SELECT MAX("water_level") FROM "h2o_feet" WHERE "location"='coyote_creek' AND time >= '2015-09-18T16:00:00Z' AND time <= '2015-09-18T16:42:00Z' GROUP BY time(12m)

name: h2o_feet
--------------
time                   max
2015-09-18T16:00:00Z   3.599
2015-09-18T16:12:00Z   3.402
2015-09-18T16:24:00Z   3.235
2015-09-18T16:36:00Z
```

带 `fill(none)` :

```
> SELECT MAX("water_level") FROM "h2o_feet" WHERE "location"='coyote_creek' AND time >= '2015-09-18T16:00:00Z' AND time <= '2015-09-18T16:42:00Z' GROUP BY time(12m) fill(none)

name: h2o_feet
--------------
time                   max
2015-09-18T16:00:00Z   3.599
2015-09-18T16:12:00Z   3.402
2015-09-18T16:24:00Z   3.235
```

例四：fill(null)

不带 `fill(null)` ：

```
> SELECT MAX("water_level") FROM "h2o_feet" WHERE "location"='coyote_creek' AND time >= '2015-09-18T16:00:00Z' AND time <= '2015-09-18T16:42:00Z' GROUP BY time(12m)

name: h2o_feet
--------------
time                   max
2015-09-18T16:00:00Z   3.599
2015-09-18T16:12:00Z   3.402
2015-09-18T16:24:00Z   3.235
2015-09-18T16:36:00Z
```

带 `fill(null)` ：

```
> SELECT MAX("water_level") FROM "h2o_feet" WHERE "location"='coyote_creek' AND time >= '2015-09-18T16:00:00Z' AND time <= '2015-09-18T16:42:00Z' GROUP BY time(12m) fill(null)

name: h2o_feet
--------------
time                   max
2015-09-18T16:00:00Z   3.599
2015-09-18T16:12:00Z   3.402
2015-09-18T16:24:00Z   3.235
2015-09-18T16:36:00Z
```

例五：fill(previous)

不带 `fill(previous)` ：

```
> SELECT MAX("water_level") FROM "h2o_feet" WHERE "location"='coyote_creek' AND time >= '2015-09-18T16:00:00Z' AND time <= '2015-09-18T16:42:00Z' GROUP BY time(12m)

name: h2o_feet
--------------
time                   max
2015-09-18T16:00:00Z   3.599
2015-09-18T16:12:00Z   3.402
2015-09-18T16:24:00Z   3.235
2015-09-18T16:36:00Z
```

带 `fill(previous)` ：

```
> SELECT MAX("water_level") FROM "h2o_feet" WHERE "location"='coyote_creek' AND time >= '2015-09-18T16:00:00Z' AND time <= '2015-09-18T16:42:00Z' GROUP BY time(12m) fill(previous)

name: h2o_feet
--------------
time                   max
2015-09-18T16:00:00Z   3.599
2015-09-18T16:12:00Z   3.402
2015-09-18T16:24:00Z   3.235
2015-09-18T16:36:00Z   3.235
```

`fill()` 的问题

问题一： `fill()` 当没有数据在查询时间范围内时

目前，如果查询的时间范围内没有任何数据，查询会忽略 `fill()` 。 这是预期的行为。GitHub上的一个开放feature request建议，即使查询的时间范围不包含数据， `fill()` 也会强制返回值。

例子：

以下查询不返回数据，因为 `water_level` 在查询的时间范围内没有任何点。 请注意， `fill(800)` 对查询结果没有影响。

```
> SELECT MEAN("water_level") FROM "h2o_feet" WHERE "location" = 'coyote_creek' AND time >= '2015-09-18T22:00:00Z' AND time <= '2015-09-18T22:18:00Z' GROUP BY time(12m) fill(800)
>
```

问题二：`fill(previous)` 当前一个结果超出查询时间范围

当前一个结果超出查询时间范围，`fill(previous)` 不会填充这个时间间隔。

例子：

以下查询涵盖 `2015-09-18T16：24：00Z` 和 `2015-09-18T16：54：00Z` 之间的时间范围。 请注意，`fill(previos)` 用 `2015-09-18T16：24：00Z` 的结果填写到了 `2015-09-18T16：36：00Z` 中。

```
> SELECT MAX("water_level") FROM "h2o_feet" WHERE location = 'coyote_creek' AND time >= '2015-09-18T16:24:00Z' AND time <= '2015-09-18T16:54:00Z' GROUP BY time(12m) fill(previous)

name: h2o_feet
--------------
time                   max
2015-09-18T16:24:00Z   3.235
2015-09-18T16:36:00Z   3.235
2015-09-18T16:48:00Z   4
```

下一个查询会缩短上一个查询的时间范围。 它现在涵盖 `2015-09-18T16：36：00Z` 和 `2015-09-18T16：54：00Z` 之间的时间。请注意，`fill(previos)` 不会用 `2015-09-18T16：24：00Z` 的结果填写到 `2015-09-18T16：36：00Z` 中。因为 `2015-09-18T16：24：00Z` 的结果在查询的较短时间范围之外。

```
> SELECT MAX("water_level") FROM "h2o_feet" WHERE location = 'coyote_creek' AND time >= '2015-09-18T16:36:00Z' AND time <= '2015-09-18T16:54:00Z' GROUP BY time(12m) fill(previous)

name: h2o_feet
--------------
time                   max
2015-09-18T16:36:00Z
2015-09-18T16:48:00Z   4
```

问题三：`fill(linear)` 当前一个结果超出查询时间范围

当前一个结果超出查询时间范围，`fill(linear)` 不会填充这个时间间隔。

例子：

以下查询涵盖 `2016-11-11T21:24:00Z` 和 `2016-11-11T22:06:00Z` 之间的时间范围。请注

意， `fill(linear)` 使用 `2016-11-11T21：24：00Z` 到 `2016-11-11T22：00：00Z` 时间间隔的值，填充到 `2016-11-11T21：36：00Z` 到 `2016-11-11T21：48：00Z` 时间间隔中。

```
> SELECT MEAN("tadpoles") FROM "pond" WHERE time > '2016-11-11T21:24:00Z' AND
time <= '2016-11-11T22:06:00Z' GROUP BY time(12m) fill(linear)

name: pond
time                  mean
----                  ----
2016-11-11T21:24:00Z  3
2016-11-11T21:36:00Z  4
2016-11-11T21:48:00Z  5
2016-11-11T22:00:00Z  6
```

下一个查询会缩短上一个查询的时间范围。 它现在涵盖 `2016-11-11T21:36:00Z` 和 `2016-11-11T22:06:00Z` 之间的时间。请注意， `fill()` 不会使用 `2016-11-11T21：24：00Z` 到 `2016-11-11T22：00：00Z` 时间间隔的值，填充到 `2016-11-11T21：36：00Z` 到 `2016-11-11T21：48：00Z` 时间间隔中。因为 `2015-09-18T16：24：00Z` 的结果在查询的较短时间范围之外。

```
> SELECT MEAN("tadpoles") FROM "pond" WHERE time >= '2016-11-11T21:36:00Z' AND
time <= '2016-11-11T22:06:00Z' GROUP BY time(12m) fill(linear)
name: pond
time                  mean
----                  ----
2016-11-11T21:36:00Z
2016-11-11T21:48:00Z
2016-11-11T22:00:00Z  6
```

INTO子句

`INTO` 子句将查询的结果写入到用户自定义的measurement中。

语法

```
SELECT_clause INTO <measurement_name> FROM_clause [WHERE_clause]
[GROUP_BY_clause]
```

语法描述

`INTO` 支持多种格式的measurement。

`INTO <measurement_name>`

写入到特定measurement中，用CLI时，写入到用 `USE` 指定的数据库，保留策略为 `DEFAULT` ，用HTTP API时，写入到 `db` 参数指定的数据库，保留策略为 `DEFAULT` 。

`INTO <database_name>.<retention_policy_name>.<measurement_name>`

写入到完整指定的measurement中。

`INTO <database_name>..<measurement_name>`

写入到指定数据库保留策略为 `DEFAULT` 。

`INTO <database_name>.<retention_policy_name>.:MEASUREMENT FROM /<regular_expression>/`

将数据写入与 `FROM` 子句中正则表达式匹配的用户指定数据库和保留策略的所有measurement。 `:MEASUREMENT` 是对 `FROM` 子句中匹配的每个measurement的反向引用。

例子

例一：重命名数据库

```
> SELECT * INTO "copy_NOAA_water_database"."autogen".:MEASUREMENT FROM
"NOAA_water_database"."autogen"./.*/ GROUP BY *

name: result
time written
---- -------
0    76290
```

在InfluxDB中直接重命名数据库是不可能的，因此 `INTO` 子句的常见用途是将数据从一个数据库移动到另一个数据库。 上述查询将 `NOAA_water_database` 和 `autogen` 保留策略中的所有数据写入 `copy_NOAA_water_database` 数据库和 `autogen` 保留策略中。

反向引用语法（ `:MEASUREMENT` ）维护目标数据库中的源measurement名称。 请注意，在运行 `INTO` 查询之前， `copy_NOAA_water_database` 数据库及其 `autogen` 保留策略都必须存在。

`GROUP BY *` 子句将源数据库中的tag留在目标数据库中的tag中。以下查询不为tag维护series的上下文;tag将作为field存储在目标数据库（ `copy_NOAA_water_database` ）中:

```
SELECT * INTO "copy_NOAA_water_database"."autogen".:MEASUREMENT FROM
"NOAA_water_database"."autogen"./.*/
```

当移动大量数据时，我们建议在 `WHERE` 子句中顺序运行不同measurement的 `INTO` 查询并使用时

间边界。这样可以防止系统内存不足。下面的代码块提供了这些查询的示例语法：

```
SELECT *
INTO <destination_database>.<retention_policy_name>.<measurement_name>
FROM <source_database>.<retention_policy_name>.<measurement_name>
WHERE time > now() - 100w and time < now() - 90w GROUP BY *

SELECT *
INTO <destination_database>.<retention_policy_name>.<measurement_name>
FROM <source_database>.<retention_policy_name>.<measurement_name>}
WHERE time > now() - 90w  and time < now() - 80w GROUP BY *

SELECT *
INTO <destination_database>.<retention_policy_name>.<measurement_name>
FROM <source_database>.<retention_policy_name>.<measurement_name>
WHERE time > now() - 80w  and time < now() - 70w GROUP BY *
```

例二：将查询结果写入到一个measurement

```
> SELECT "water_level" INTO "h2o_feet_copy_1" FROM "h2o_feet" WHERE "location"
= 'coyote_creek'

name: result
------------
time                   written
1970-01-01T00:00:00Z   7604

> SELECT * FROM "h2o_feet_copy_1"

name: h2o_feet_copy_1
---------------------
time                   water_level
2015-08-18T00:00:00Z   8.12
[...]
2015-09-18T16:48:00Z   4
```

该查询将其结果写入新的measurement： `h2o_feet_copy_1` 。如果使用CLI，InfluxDB会将数据写入 `USE` d数据库和 `DEFAULT` 保留策略。 如果您使用HTTP API，InfluxDB会将数据写入参数 `db` 指定的数据库和 `rp` 指定的保留策略。如果您没有设置 `rp` 参数，HTTP API将自动将数据写入数据库的 `DEFAULT` 保留策略。

响应显示InfluxDB写入 `h2o_feet_copy_1` 的点数（7605）。 响应中的时间戳是无意义的；

InfluxDB使用epoch 0（ `1970-01-01T00：00：00Z` ）作为空时间戳等价物。

例三：将查询结果写入到一个完全指定的measurement中

```
> SELECT "water_level" INTO "where_else"."autogen"."h2o_feet_copy_2" FROM
"h2o_feet" WHERE "location" = 'coyote_creek'

name: result
------------
time                    written
1970-01-01T00:00:00Z    7604

> SELECT * FROM "where_else"."autogen"."h2o_feet_copy_2"

name: h2o_feet_copy_2
---------------------
time                    water_level
2015-08-18T00:00:00Z    8.12
[...]
2015-09-18T16:48:00Z    4
```

例四：将聚合结果写入到一个measurement中(采样)

```
> SELECT MEAN("water_level") INTO "all_my_averages" FROM "h2o_feet" WHERE
"location" = 'coyote_creek' AND time >= '2015-08-18T00:00:00Z' AND time <=
'2015-08-18T00:30:00Z' GROUP BY time(12m)

name: result
------------
time                    written
1970-01-01T00:00:00Z    3

> SELECT * FROM "all_my_averages"

name: all_my_averages
---------------------
time                    mean
2015-08-18T00:00:00Z    8.0625
2015-08-18T00:12:00Z    7.8245
2015-08-18T00:24:00Z    7.5675
```

查询使用InfluxQL函数和 `GROUP BY time()` 子句聚合数据。它也将其结果写

入 `all_my_averages` measurement。

该查询是采样的示例：采用更高精度的数据，将这些数据聚合到较低的精度，并将较低精度数据存储在数据库中。 采样是 `INTO` 子句的常见用例。

例五：将多个measurement的聚合结果写入到一个不同的数据库中(逆向引用采样)

```
> SELECT MEAN(*) INTO "where_else"."autogen".:MEASUREMENT FROM /.*/ WHERE time
>= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:06:00Z' GROUP BY time(12m)

name: result
time                  written
----                  -------
1970-01-01T00:00:00Z  5

> SELECT * FROM "where_else"."autogen"./.*/

name: average_temperature
time                  mean_degrees  mean_index  mean_pH  mean_water_level
----                  ------------  ----------  -------  ----------------
2015-08-18T00:00:00Z  78.5

name: h2o_feet
time                  mean_degrees  mean_index  mean_pH  mean_water_level
----                  ------------  ----------  -------  ----------------
2015-08-18T00:00:00Z                                     5.07625

name: h2o_pH
time                  mean_degrees  mean_index  mean_pH  mean_water_level
----                  ------------  ----------  -------  ----------------
2015-08-18T00:00:00Z                            6.75

name: h2o_quality
time                  mean_degrees  mean_index  mean_pH  mean_water_level
----                  ------------  ----------  -------  ----------------
2015-08-18T00:00:00Z                51.75

name: h2o_temperature
time                  mean_degrees  mean_index  mean_pH  mean_water_level
----                  ------------  ----------  -------  ----------------
2015-08-18T00:00:00Z  63.75
```

查询使用InfluxQL函数和 `GROUP BY time()` 子句聚合数据。它会在与 `FROM` 子句中的正则表达式匹配的每个measurement中聚合数据，并将结果写入 `where_else` 数据库和 `autogen` 保留策略中具有相同名称的measurement中。请注意，在运行 `INTO` 查询之前， `where_else` 和 `autogen` 都必须存在。

该查询是使用反向引用进行下采样的示例。它从多个measurement中获取更高精度的数据，将这些数据聚合到较低的精度，并将较低精度数据存储在数据库中。使用反向引用进行下采样是 `INTO` 子句的常见用例。

INTO子句的共同问题

问题一：丢数据

如果 `INTO` 查询在 `SELECT` 子句中包含tag key，则查询将当前measurement中的tag转换为目标measurement中的字段。这可能会导致InfluxDB覆盖以前由tag value区分的点。请注意，此行为不适用于使用 `TOP()` 或 `BOTTOM()` 函数的查询。

要将当前measurement的tag保留在目标measurement中的tag中， `GROUP BY` 相关tag key或 `INTO` 查询中的 `GROUP BY *` 。

问题二：使用INTO子句自动查询

本文档中的 `INTO` 子句部分显示了如何使用 `INTO` 子句手动实现查询。 有关如何自动执行 `INTO` 子句查询实时数据，请参阅Continous Queries文档。除了其他用途之外，Continous Queries使采样过程自动化。

ORDER BY TIME DESC

默认情况下，InfluxDB以升序的顺序返回结果； 返回的第一个点具有最早的时间戳，返回的最后一个点具有最新的时间戳。 `ORDER BY time DESC` 反转该顺序，使得InfluxDB首先返回具有最新时间戳的点。

语法

```
SELECT_clause [INTO_clause] FROM_clause [WHERE_clause] [GROUP_BY_clause] ORDER BY time DESC
```

语法描述

如果查询包含 `GROUP BY` 子句， `ORDER by time DESC` 必须出现在 `GROUP BY` 子句之后。如果查询包含一个 `WHERE` 子句并没有 `GROUP BY` 子句， `ORDER by time DESC` 必须出现在 `WHERE` 子句之后。

例子

例一：首先返回最新的点

```
> SELECT "water_level" FROM "h2o_feet" WHERE "location" = 'santa_monica' ORDER BY time DESC

name: h2o_feet
time                   water_level
----                   -----------
2015-09-18T21:42:00Z   4.938
2015-09-18T21:36:00Z   5.066
[...]
2015-08-18T00:06:00Z   2.116
2015-08-18T00:00:00Z   2.064
```

该查询首先从 `h2o_feet` measurement返回具有最新时间戳的点。没有 `ORDER by time DESC` ，查询将首先返回 `2015-08-18T00：00：00Z` 最后返回 `2015-09-18T21：42：00Z` 。

例二：首先返回最新的点并包括GROUP BY time()子句

```
> SELECT MEAN("water_level") FROM "h2o_feet" WHERE time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:42:00Z' GROUP BY time(12m) ORDER BY time DESC

name: h2o_feet
time                   mean
----                   ----
2015-08-18T00:36:00Z   4.6825
2015-08-18T00:24:00Z   4.80675
2015-08-18T00:12:00Z   4.950749999999999
2015-08-18T00:00:00Z   5.07625
```

该查询在 `GROUP BY` 子句中使用InfluxQL函数和时间间隔来计算查询时间范围内每十二分钟间隔的平均 `water_level` 。 `ORDER BY time DESC` 返回最近12分钟的时间间隔。

LIMIT和SLIMIT子句

`LIMIT <N>` 从指定的measurement中返回前 `N` 个数据点。

语法

```
SELECT_clause [INTO_clause] FROM_clause [WHERE_clause] [GROUP_BY_clause]
[ORDER_BY_clause] LIMIT <N>
```

语法描述

`N` 指定从指定measurement返回的点数。如果 `N` 大于measurement的点总数，InfluxDB返回该measurement中的所有点。请注意， `LIMIT` 子句必须以上述语法中列出的顺序显示。

例子

例一：限制返回的点数

```
> SELECT "water_level","location" FROM "h2o_feet" LIMIT 3

name: h2o_feet
time                   water_level   location
----                   -----------   --------
2015-08-18T00:00:00Z   8.12          coyote_creek
2015-08-18T00:00:00Z   2.064         santa_monica
2015-08-18T00:06:00Z   8.005         coyote_creek
```

这个查询从measurement `h2o_feet` 中返回最旧的三个点。

例二：限制返回的点数并包含一个GROUP BY子句

```
> SELECT MEAN("water_level") FROM "h2o_feet" WHERE time >= '2015-08-
18T00:00:00Z' AND time <= '2015-08-18T00:42:00Z' GROUP BY *,time(12m) LIMIT 2

name: h2o_feet
tags: location=coyote_creek
time                   mean
----                   ----
2015-08-18T00:00:00Z   8.0625
2015-08-18T00:12:00Z   7.8245

name: h2o_feet
tags: location=santa_monica
time                   mean
----                   ----
2015-08-18T00:00:00Z   2.09
2015-08-18T00:12:00Z   2.077
```

该查询使用InfluxQL函数和GROUP BY子句来计算每个tag以及查询时间内每隔十二分钟的间隔的平均 `water_level` 。 `LIMIT 2` 请求两个最旧的十二分钟平均值。

请注意，没有 `LIMIT 2` ，查询将返回每个series四个点；在查询的时间范围内每隔十二分钟的时间间隔一个点。

SLIMIT子句

`SLIMIT <N>` 返回指定measurement的前个series中的每一个点。

语法

```
SELECT_clause [INTO_clause] FROM_clause [WHERE_clause] GROUP BY *
[,time(<time_interval>)] [ORDER_BY_clause] SLIMIT <N>
```

语法描述

`N` 表示从指定measurement返回的序列数。如果 `N` 大于measurement中的series数，InfluxDB将从该measurement中返回所有series。

有一个issue，要求使用 `SLIMIT` 来查询 `GROUP BY *` 。 请注意， `SLIMIT` 子句必须按照上述语法中的顺序显示。

例子

例一：限制返回的series的数目

```
> SELECT "water_level" FROM "h2o_feet" GROUP BY * SLIMIT 1

name: h2o_feet
tags: location=coyote_creek
time                   water_level
----                   -----
2015-08-18T00:00:00Z   8.12
2015-08-18T00:06:00Z   8.005
2015-08-18T00:12:00Z   7.887
[...]
2015-09-18T16:12:00Z   3.402
2015-09-18T16:18:00Z   3.314
2015-09-18T16:24:00Z   3.235
```

该查询从measurement `h2o_feet` 中返回一个series的所有点。

例二：限制返回的series的数目并且包括一个GROUP BY time()子句

```
> SELECT MEAN("water_level") FROM "h2o_feet" WHERE time >= '2015-08-
18T00:00:00Z' AND time <= '2015-08-18T00:42:00Z' GROUP BY *,time(12m) SLIMIT 1

name: h2o_feet
tags: location=coyote_creek
time                   mean
----                   ----
2015-08-18T00:00:00Z   8.0625
2015-08-18T00:12:00Z   7.8245
2015-08-18T00:24:00Z   7.5675
2015-08-18T00:36:00Z   7.303
```

该查询在GROUP BY子句中使用InfluxQL函数和时间间隔来计算查询时间范围内每十二分钟间隔的平均 `water_level` 。 `SLIMIT 1` 要求返回与measurement `h2o_feet` 相关联的一个series。

请注意，如果没有 `SLIMIT 1` ，查询将返回与 `h2o_feet` 相关联的两个series的结果： `location = coyote_creek` 和 `location = santa_monica` 。

LIMIT和SLIMIT一起使用

`SLIMIT <N>` 后面跟着 `LIMIT <N>` 返回指定measurement的个series中的个数据点。

语法

```
SELECT_clause [INTO_clause] FROM_clause [WHERE_clause] GROUP BY *
[,time(<time_interval>)] [ORDER_BY_clause] LIMIT <N1> SLIMIT <N2>
```

语法描述

`N1` 指定每次measurement返回的点数。如果 `N1` 大于measurement的点数，InfluxDB将从该测量中返回所有点。

`N2` 指定从指定measurement返回的series数。如果 `N2` 大于measurement中series联数，InfluxDB将从该measurement中返回所有series。

有一个issue，要求需要 `LIMIT` 和 `SLIMIT` 的查询才能包含 `GROUP BY *` 。

例子

例一：限制数据点数和series数的返回

```
> SELECT "water_level" FROM "h2o_feet" GROUP BY * LIMIT 3 SLIMIT 1

name: h2o_feet
tags: location=coyote_creek
time                   water_level
----                   -----------
2015-08-18T00:00:00Z   8.12
2015-08-18T00:06:00Z   8.005
2015-08-18T00:12:00Z   7.887
```

该查询从measurement `h2o_feet` 中的一个series钟返回最老的三个点。

例二：限制数据点数和series数并且包括一个GROUP BY time()子句

```
> SELECT MEAN("water_level") FROM "h2o_feet" WHERE time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:42:00Z' GROUP BY *,time(12m) LIMIT 2 SLIMIT 1

name: h2o_feet
tags: location=coyote_creek
time                   mean
----                   ----
2015-08-18T00:00:00Z   8.0625
2015-08-18T00:12:00Z   7.8245
```

该查询在 `GROUP BY` 子句中使用InfluxQL函数和时间间隔来计算查询时间范围内每十二分钟间隔的平均 `water_level` 。 `LIMIT 2` 请求两个最早的十二分钟平均值， `SLIMIT 1` 请求与measurement `h2o_feet` 相关联的一个series。

请注意，如果没有 `LIMIT 2` `SLIMIT 1` ，查询将返回与 `h2o_feet` 相关联的两个series中的每一个的四个点。

OFFSET和SOFFSET子句

`OFFSET` 和 `SOFFSET` 分页和series返回。

OFFSET子句

`OFFSET <N>` 从查询结果中返回分页的N个数据点

语法

```
SELECT_clause [INTO_clause] FROM_clause [WHERE_clause] [GROUP_BY_clause]
[ORDER_BY_clause] LIMIT_clause OFFSET <N> [SLIMIT_clause]
```

语法描述

`N` 指定分页数。 `OFFSET` 子句需要一个 `LIMIT` 子句。使用没有 `LIMIT` 子句的 `OFFSET` 子句可能会导致不一致的查询结果。

例子

例一：分页数据点

```
> SELECT "water_level","location" FROM "h2o_feet" LIMIT 3 OFFSET 3

name: h2o_feet
time                   water_level   location
----                   -----------   --------
2015-08-18T00:06:00Z   2.116         santa_monica
2015-08-18T00:12:00Z   7.887         coyote_creek
2015-08-18T00:12:00Z   2.028         santa_monica
```

该查询从measurement `h2o_feet` 中返回第4，5，6个数据点，如果查询语句中不包括 `OFFSET 3` ，则会返回measurement中的第1，2，3个数据点。

例二：分页数据点并包括多个子句

```
> SELECT MEAN("water_level") FROM "h2o_feet" WHERE time >= '2015-08-
18T00:00:00Z' AND time <= '2015-08-18T00:42:00Z' GROUP BY *,time(12m) ORDER BY
time DESC LIMIT 2 OFFSET 2 SLIMIT 1

name: h2o_feet
tags: location=coyote_creek
time                   mean
----                   ----
2015-08-18T00:12:00Z   7.8245
2015-08-18T00:00:00Z   8.0625
```

这个例子包含的东西很多，我们一个一个来看：

- `SELECT` 指明InfluxQL的函数；
- `FROM` 指明单个measurement；
- `WHERE` 指明查询的时间范围；

- `GROUP BY` 将结果对所有tag作group by;
- `GROUP BY time DESC` 按照时间戳的降序返回结果;
- `LIMIT 2` 限制返回的点数为2;
- `OFFSET 2` 查询结果中不包括最开始的两个值;
- `SLIMIT 1` 限制返回的series数目为1;

如果没有 `OFFSET 2` ，查询将会返回最先的两个点:

```
name: h2o_feet
tags: location=coyote_creek
time                   mean
----                   ----
2015-08-18T00:36:00Z   7.303
2015-08-18T00:24:00Z   7.5675
```

SOFFSET子句

`SOFFSET <N>` 从查询结果中返回分页的N个series

语法

```
    SELECT_clause [INTO_clause] FROM_clause [WHERE_clause] GROUP BY *
    [,time(time_interval)] [ORDER_BY_clause] [LIMIT_clause] [OFFSET_clause]
1.  SLIMIT_clause SOFFSET <N>
```

语法描述

`N` 指定series的分页数。 `SOFFSET` 子句需要一个 `SLIMIT` 子句。使用没有 `SLIMIT` 子句的 `SOFFSET` 子句可能会导致不一致的查询结果。

> 注意: 如果 `SOFFSET` 指定的大于series的数目，则InfluxDB返回空值。

例子

例一: 分页series

```
1.  > SELECT "water_level" FROM "h2o_feet" GROUP BY * SLIMIT 1 SOFFSET 1
2.
3.  name: h2o_feet
4.  tags: location=santa_monica
5.  time                   water_level
```

```
----                    -----------
2015-08-18T00:00:00Z    2.064
2015-08-18T00:06:00Z    2.116
[...]
2015-09-18T21:36:00Z    5.066
2015-09-18T21:42:00Z    4.938
```

查询返回与 `h2o_feet` 相关的series数据，并返回tag `location = santa_monica` 。没有 `SOFFSET 1` ，查询返回与 `h2o_feet` 和 `location = coyote_creek` 相关的series的所有数据。

例二：分页series并包括多个子句

```
> SELECT MEAN("water_level") FROM "h2o_feet" WHERE time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:42:00Z' GROUP BY *,time(12m) ORDER BY time DESC LIMIT 2 OFFSET 2 SLIMIT 1 SOFFSET 1

name: h2o_feet
tags: location=santa_monica
time                    mean
----                    ----
2015-08-18T00:12:00Z    2.077
2015-08-18T00:00:00Z    2.09
```

这个例子包含的东西很多，我们一个一个来看：

- `SELECT` 指明InfluxQL的函数；
- `FROM` 指明单个measurement；
- `WHERE` 指明查询的时间范围；
- `GROUP BY` 将结果对所有tag作group by；
- `GROUP BY time DESC` 按照时间戳的降序返回结果；
- `LIMIT 2` 限制返回的点数为2；
- `OFFSET 2` 查询结果中不包括最开始的两个值；
- `SLIMIT 1` 限制返回的series数目为1；
- `SOFFSET 1` 分页返回的series；

如果没有 `SOFFSET 2` ，查询将会返回不同的series：

```
name: h2o_feet
tags: location=coyote_creek
time                    mean
----                    ----
```

```
2015-08-18T00:12:00Z   7.8245
2015-08-18T00:00:00Z   8.0625
```

Time Zone子句

`tz()` 子句返回指定时区的UTC偏移量。

语法

```
SELECT_clause [INTO_clause] FROM_clause [WHERE_clause] [GROUP_BY_clause]
[ORDER_BY_clause] [LIMIT_clause] [OFFSET_clause] [SLIMIT_clause]
[SOFFSET_clause] tz('<time_zone>')
```

语法描述

默认情况下，InfluxDB以UTC为单位存储并返回时间戳。 `tz()` 子句包含UTC偏移量，或UTC夏令时（DST）偏移量到查询返回的时间戳中。 返回的时间戳必须是RFC3339格式，用于UTC偏移量或UTC DST才能显示。 `time_zone` 参数遵循Internet Assigned Numbers Authority时区数据库中的TZ语法，它需要单引号。

例子

例一：返回从UTC偏移到芝加哥时区的数据

```
> SELECT "water_level" FROM "h2o_feet" WHERE "location" = 'santa_monica' AND
time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:18:00Z'
tz('America/Chicago')

name: h2o_feet
time                        water_level
----                        -----------
2015-08-17T19:00:00-05:00  2.064
2015-08-17T19:06:00-05:00  2.116
2015-08-17T19:12:00-05:00  2.028
2015-08-17T19:18:00-05:00  2.126
```

查询的结果包括UTC偏移-5个小时的美国芝加哥时区的时间戳。

时间语法

对于大多数 `SELECT` 语句，默认时间范围为UTC的 `1677-09-21 00：12：43.145224194` 到 `2262-04-11T23：47：16.854775806Z` 。 对于具有 `GROUP BY time()` 子句的 `SELECT` 语句，默认时间范围在UTC的 `1677-09-21 00：12：43.145224194` 和now()之间。以下部分详细说明了如何在 `SELECT` 语句的 `WHERE` 子句中指定替代时间范围。

绝对时间

用时间字符串或是epoch时间来指定绝对时间

语法

```
    SELECT_clause FROM_clause WHERE time <operator> ['<rfc3339_date_time_string>' |
    '<rfc3339_like_date_time_string>' | <epoch_time>] [AND
    ['<rfc3339_date_time_string>' | '<rfc3339_like_date_time_string>' |
1.  <epoch_time>] [...]]
```

语法描述

支持的操作符

`=` 等于 `<>` 不等于 `!=` 不等于 `>` 大于 `>=` 大于等于 `<` 小于 `<=` 小于等于

最近，InfluxDB不再支持在 `WHERE` 的绝对时间里面使用 `OR` 了。

rfc3399时间字符串

```
1.  'YYYY-MM-DDTHH:MM:SS.nnnnnnnnnZ'
```

`.nnnnnnnnn` 是可选的，如果没有的话，默认是 `.00000000` ,rfc3399格式的时间字符串要用单引号引起来。

epoch_time

Epoch时间是1970年1月1日星期四00:00:00（UTC）以来所经过的时间。默认情况下，InfluxDB假定所有epoch时间戳都是纳秒。也可以在epoch时间戳的末尾包括一个表示时间精度的字符，以表示除纳秒以外的精度。

基本算术

所有时间戳格式都支持基本算术。用表示时间精度的字符添加（+）或减去（-）一个时间。请注意，InfluxQL需要+或-和表示时间精度的字符之间用空格隔开。

例子

例一：指定一个RFC3339格式的时间间隔

```
> SELECT "water_level" FROM "h2o_feet" WHERE "location" = 'santa_monica' AND
time >= '2015-08-18T00:00:00.000000000Z' AND time <= '2015-08-18T00:12:00Z'

name: h2o_feet
time                   water_level
----                   -----------
2015-08-18T00:00:00Z   2.064
2015-08-18T00:06:00Z   2.116
2015-08-18T00:12:00Z   2.028
```

该查询会返回时间戳在2015年8月18日00: 00: 00.000000000和2015年8月18日00:12:00之间的数据。 第一个时间戳（.000000000）中的纳秒是可选的。

请注意，RFC3339日期时间字符串必须用单引号引起来。

例二：指定一个类似于RFC3339格式的时间间隔

```
> SELECT "water_level" FROM "h2o_feet" WHERE "location" = 'santa_monica' AND
time >= '2015-08-18' AND time <= '2015-08-18 00:12:00'

name: h2o_feet
time                   water_level
----                   -----------
2015-08-18T00:00:00Z   2.064
2015-08-18T00:06:00Z   2.116
2015-08-18T00:12:00Z   2.028
```

该查询会返回时间戳在2015年8月18日00:00:00和2015年8月18日00:12:00之间的数据。 第一个日期时间字符串不包含时间； InfluxDB会假设时间是00:00:00。

例三：指定epoch格式的时间间隔

```
> SELECT "water_level" FROM "h2o_feet" WHERE "location" = 'santa_monica' AND
time >= 1439856000000000000 AND time <= 1439856720000000000

name: h2o_feet
time                   water_level
----                   -----------
2015-08-18T00:00:00Z   2.064
2015-08-18T00:06:00Z   2.116
2015-08-18T00:12:00Z   2.028
```

该查询返回的数据的时间戳为2015年8月18日00:00:00和2015年8月18日00:12:00之间。默认情况

下，InfluxDB处理epoch格式下时间戳为纳秒。

例四：指定epoch以秒为精度的时间间隔

```
> SELECT "water_level" FROM "h2o_feet" WHERE "location" = 'santa_monica' AND
time >= 1439856000s AND time <= 1439856720s

name: h2o_feet
time                   water_level
----                   -----------
2015-08-18T00:00:00Z   2.064
2015-08-18T00:06:00Z   2.116
2015-08-18T00:12:00Z   2.028
```

该查询返回的数据的时间戳为2015年8月18日00:00:00和2015年8月18日00:12:00之间。在epoch时间戳结尾处的 `s` 表示时间戳以秒为单位。

例五：对RFC3339格式的时间戳的基本计算

```
> SELECT "water_level" FROM "h2o_feet" WHERE time > '2015-09-18T21:24:00Z' + 6m

name: h2o_feet
time                   water_level
----                   -----------
2015-09-18T21:36:00Z   5.066
2015-09-18T21:42:00Z   4.938
```

该查询返回数据，其时间戳在2015年9月18日21时24分后六分钟。请注意， `+` 和 `6m` 之间的空格是必需的。

例六：对epoch时间戳的基本计算

```
> SELECT "water_level" FROM "h2o_feet" WHERE time > 24043524m - 6m

name: h2o_feet
time                   water_level
----                   -----------
2015-09-18T21:24:00Z   5.013
2015-09-18T21:30:00Z   5.01
2015-09-18T21:36:00Z   5.066
2015-09-18T21:42:00Z   4.938
```

该查询返回数据，其时间戳在2015年9月18日21:24:00之前六分钟。请注意， `-` 和 `6m` 之间的空

格是必需的。

相对时间

使用 `now()` 查询时间戳相对于服务器当前时间戳的的数据。

语法

```
    SELECT_clause FROM_clause WHERE time <operator> now() [[ - | + ]
1.  <duration_literal>] [(AND|OR) now() [...]]
```

语法描述

`now()` 是在该服务器上执行查询时服务器的Unix时间。 `-` 或 `+` 和时间字符串之间需要空格。

支持的操作符

`=` 等于 `<>` 不等于 `!=` 不等于 `>` 大于 `>=` 大于等于 `<` 小于 `<=` 小于等于

时间字符串

`u` 或 `µ` 微秒 `ms` 毫秒 `s` 秒 `m` 分钟 `h` 小时 `d` 天 `w` 星期

例子

例一：用相对时间指定时间间隔

```
1.  > SELECT "water_level" FROM "h2o_feet" WHERE time > now() - 1h
```

该查询返回过去一个小时的数据。

例二：用绝对和相对时间指定时间间隔

```
    > SELECT "level description" FROM "h2o_feet" WHERE time > '2015-09-
1.  18T21:18:00Z' AND time < now() + 1000d
2.
3.  name: h2o_feet
4.  time                   level description
5.  ----                   -----------------
6.  2015-09-18T21:24:00Z   between 3 and 6 feet
7.  2015-09-18T21:30:00Z   between 3 and 6 feet
8.  2015-09-18T21:36:00Z   between 3 and 6 feet
9.  2015-09-18T21:42:00Z   between 3 and 6 feet
```

该查询返回的数据的时间戳在2015年9月18日的21:18:00到从现在之后1000天之间。

时间语法的一些常见问题

问题一：在绝对时间中使用OR

当前，InfluxDB不支持在绝对时间的 `WHERE` 子句中使用 `OR` 。

问题二：在有GROUP BY time()中查询发生在now()之后的数据

大多数 `SELECT` 语句的默认时间范围为UTC的 `1677-09-21 00：12：43.145224194` 到 `2262-04-11T23：47：16.854775806Z` 。对于具有 `GROUP BY time()` 子句的 `SELECT` 语句，默认时间范围在UTC的 `1677-09-21 00：12：43.145224194` 和 `now()` 之间。

要查询 `now()` 之后发生的时间戳的数据，具有 `GROUP BY time()` 子句的 `SELECT` 语句必须在 `WHERE` 子句中提供一个时间的上限。

例子

使用CLI写入数据库 `NOAA_water_database` ,且发生在 `now()` 之后的数据点。

```
1. > INSERT h2o_feet,location=santa_monica water_level=3.1 1587074400000000000
```

运行 `GROUP BY time()` 查询，涵盖 `2015-09-18T21：30：00Z` 和 `now()` 之间的时间戳的数据：

```
    > SELECT MEAN("water_level") FROM "h2o_feet" WHERE "location"='santa_monica'
1.  AND time >= '2015-09-18T21:30:00Z' GROUP BY time(12m) fill(none)
2.
3.  name: h2o_feet
4.  time                   mean
5.  ----                   ----
6.  2015-09-18T21:24:00Z   5.01
7.  2015-09-18T21:36:00Z   5.002
```

运行 `GROUP BY time()` 查询，涵盖 `2015-09-18T21：30：00Z` 和 `now()` 之后180星期之间的时间戳的数据：

```
     > SELECT MEAN("water_level") FROM "h2o_feet" WHERE "location"='santa_monica'
    AND time >= '2015-09-18T21:30:00Z' AND time <= now() + 180w GROUP BY time(12m)
1.  fill(none)
2.
3.  name: h2o_feet
4.  time                   mean
5.  ----                   ----
6.  2015-09-18T21:24:00Z   5.01
```

```
7.  2015-09-18T21:36:00Z   5.002
8.  2020-04-16T22:00:00Z   3.1
```

请注意， `WHERE` 子句必须提供替代上限来覆盖默认的 `now()` 上限。 以下查询仅将下限重置为 `now()` ，这样查询的时间范围在 `now()` 和 `now()` 之间：

```
    > SELECT MEAN("water_level") FROM "h2o_feet" WHERE "location"='santa_monica'
1.  AND time >= now() GROUP BY time(12m) fill(none)
2.  >
```

问题三：配置返回的时间戳

默认情况下，CLI以纳秒时间格式返回时间戳。使用 `precision <format>` 命令指定替代格式。默认情况下，HTTP API返回RFC3339格式的时间戳。使用 `epoch` 查询参数指定替代格式。

正则表达式

InluxDB支持在以下场景使用正则表达式：

- 在 `SELECT` 中的field key和tag key;
- 在 `FROM` 中的measurement
- 在 `WHERE` 中的tag value和字符串类型的field value
- 在 `GROUP BY` 中的tag key

目前，InfluxQL不支持在 `WHERE` 中使用正则表达式去匹配不是字符串的field value，以及数据库名和retention policy。

> 注意：正则表达式比精确的字符串更加耗费计算资源；具有正则表达式的查询比那些没有的性能要低一些。

语法

```
    SELECT /<regular_expression_field_key>/ FROM /<regular_expression_measurement>/
    WHERE [<tag_key> <operator> /<regular_expression_tag_value>/ | <field_key>
    <operator> /<regular_expression_field_value>/] GROUP BY
1.  /<regular_expression_tag_key>/
```

语法描述

正则表达式前后使用斜杠 `/` ，并且使用Golang的正则表达式语法。

支持的操作符：

`=~` 匹配 `!~` 不匹配

例子：

例一：在SELECT中使用正则表达式指定field key和tag key

```
1. > SELECT /l/ FROM "h2o_feet" LIMIT 1
2.
3. name: h2o_feet
4. time                   level description      location       water_level
5. ----                   -----------------      --------       -----------
6. 2015-08-18T00:00:00Z   between 6 and 9 feet   coyote_creek   8.12
```

查询选择所有包含 `l` 的tag key和field key。请注意， `SELECT` 子句中的正则表达式必须至少匹配一个field key，以便返回与正则表达式匹配的tag key。

目前，没有语法来区分 `SELECT` 子句中field key的正则表达式和tag key的正则表达式。不支持语法 `/<regular_expression>/::[field | tag]` 。

例二：在SELECT中使用正则表达式指定函数里面的field key

```
    > SELECT DISTINCT(/level/) FROM "h2o_feet" WHERE "location" = 'santa_monica'
1. AND time >= '2015-08-18T00:00:00.000000000Z' AND time <= '2015-08-18T00:12:00Z'
2.
3. name: h2o_feet
4. time                   distinct_level description   distinct_water_level
5. ----                   --------------------------   --------------------
6. 2015-08-18T00:00:00Z   below 3 feet                 2.064
7. 2015-08-18T00:00:00Z                                2.116
8. 2015-08-18T00:00:00Z                                2.028
```

该查询使用InfluxQL函数返回每个包含 `level` 的field key的去重后的field value。

例三：在FROM中使用正则表达式指定measurement

```
1. > SELECT MEAN("degrees") FROM /temperature/
2.
3. name: average_temperature
4. time                   mean
5. ----                   ----
6. 1970-01-01T00:00:00Z   79.98472932232272
7.
```

```
name: h2o_temperature
time                mean
----                ----
1970-01-01T00:00:00Z   64.98872722506226
```

该查询使用InfluxQL函数计算在数据库 `NOAA_water_database` 中包含 `temperature` 的每个measurement的平均 `degrees` 。

例四：在WHERE中使用正则表达式指定tag value

```
> SELECT MEAN(water_level) FROM "h2o_feet" WHERE "location" =~ /[m]/ AND
"water_level" > 3

name: h2o_feet
time                   mean
----                   ----
1970-01-01T00:00:00Z   4.47155532049926
```

该查询使用InfluxQL函数来计算平均水位，其中 `location` 的tag value包括 `m` 并且 `water_level` 大于3。

例五：在WHERE中使用正则表达式指定无值的tag

```
> SELECT * FROM "h2o_feet" WHERE "location" !~ /./
>
```

该查询从measurement `h2o_feet` 中选择所有数据，其中tag `location` 没有值。 `NOAA_water_database` 中的每个数据点都具有 `location` 这个tag。

例六：在WHERE中使用正则表达式指定有值的tag

```
> SELECT MEAN("water_level") FROM "h2o_feet" WHERE "location" =~ /./

name: h2o_feet
time                   mean
----                   ----
1970-01-01T00:00:00Z   4.442107025822523
```

该查询使用InfluxQL函数计算所有 `location` 这个tag的数据点的平均 `water_level` 。

例七：在WHERE中使用正则表达式指定一个field value

```
> SELECT MEAN("water_level") FROM "h2o_feet" WHERE "location" = 'santa_monica'
AND "level description" =~ /between/

name: h2o_feet
time                   mean
----                   ----
1970-01-01T00:00:00Z   4.47155532049926
```

该查询使用InfluxQL函数计算所有字段 `level description` 的值含有 `between` 的数据点的平均 `water_level` 。

例八：在GROUP BY中使用正则表达式指定tag key

```
> SELECT FIRST("index") FROM "h2o_quality" GROUP BY /l/

name: h2o_quality
tags: location=coyote_creek
time                   first
----                   -----
2015-08-18T00:00:00Z   41

name: h2o_quality
tags: location=santa_monica
time                   first
----                   -----
2015-08-18T00:00:00Z   99
```

该查询使用InfluxQL函数查询每个tag key包含字母 `l` 的tag的第一个 `index` 值。

数据类型和转换

在 `SELECT` 中支持指定field的类型，以及使用 `::` 完成基本的类型转换。

数据类型

field的value支持浮点，整数，字符串和布尔型。 `::` 语法允许用户在查询中指定field的类型。

> 注意：一般来说，没有必要在SELECT子句中指定字段值类型。 在大多数情况下，InfluxDB拒绝尝试将字段值写入以前接受的不同类型的字段值的字段的任何数据。字段值类型可能在分片组之间不同。在这些情况下，可能需要在SELECT子句中指定字段值类型。

语法

```
SELECT_clause <field_key>::<type> FROM_clause
```

语法描述

`type` 可以是 `float` ， `integer` ， `string` 和 `boolean` 。在大多数情况下，如果 `field_key` 没有存储指定 `type` 的数据，那么InfluxDB将不会返回数据。

例子

```
> SELECT "water_level"::float FROM "h2o_feet" LIMIT 4

name: h2o_feet
--------------
time                   water_level
2015-08-18T00:00:00Z   8.12
2015-08-18T00:00:00Z   2.064
2015-08-18T00:06:00Z   8.005
2015-08-18T00:06:00Z   2.116
```

该查询返回field key `water_level` 为浮点型的数据。

类型转换

`::` 语法允许用户在查询中做基本的数据类型转换。目前，InfluxDB支持冲整数转到浮点，或者从浮点转到整数。

语法

```
SELECT_clause <field_key>::<type> FROM_clause
```

语法描述

`type` 可以是 `float` 或者 `integer` 。

如果查询试图把整数或者浮点数转换成字符串或者布尔型，InfluxDB将不会返回数据。

例子

例一：浮点数转换成整型

```
> SELECT "water_level"::integer FROM "h2o_feet" LIMIT 4
```

```
2.
3.  name: h2o_feet
4.  --------------
5.  time                   water_level
6.  2015-08-18T00:00:00Z   8
7.  2015-08-18T00:00:00Z   2
8.  2015-08-18T00:06:00Z   8
9.  2015-08-18T00:06:00Z   2
```

例一：浮点数转换成字符串(目前不支持)

```
1.  > SELECT "water_level"::string FROM "h2o_feet" LIMIT 4
2.  >
```

所有返回为空。

多语句

用分号 `;` 分割多个 `SELECT` 语句。

例子

CLI:

```
    > SELECT MEAN("water_level") FROM "h2o_feet"; SELECT "water_level" FROM
1.  "h2o_feet" LIMIT 2
2.
3.  name: h2o_feet
4.  time                   mean
5.  ----                   ----
6.  1970-01-01T00:00:00Z   4.442107025822522
7.
8.  name: h2o_feet
9.  time                   water_level
10. ----                   -----------
11. 2015-08-18T00:00:00Z   8.12
12. 2015-08-18T00:00:00Z   2.064
```

HTTP API

```
1.  {
```

```
    "results": [
        {
            "statement_id": 0,
            "series": [
                {
                    "name": "h2o_feet",
                    "columns": [
                        "time",
                        "mean"
                    ],
                    "values": [
                        [
                            "1970-01-01T00:00:00Z",
                            4.442107025822522
                        ]
                    ]
                }
            ]
        },
        {
            "statement_id": 1,
            "series": [
                {
                    "name": "h2o_feet",
                    "columns": [
                        "time",
                        "water_level"
                    ],
                    "values": [
                        [
                            "2015-08-18T00:00:00Z",
                            8.12
                        ],
                        [
                            "2015-08-18T00:00:00Z",
                            2.064
                        ]
                    ]
                }
            ]
        }
    ]
```

```
44. }
```

子查询

子查询是嵌套在另一个查询的 `FROM` 子句中的查询。使用子查询将查询作为条件应用于其他查询。子查询提供与嵌套函数和SQL `HAVING` 子句类似的功能。

语法

```
1. SELECT_clause FROM ( SELECT_statement ) [...]
```

语法描述

InfluxDB首先执行子查询，再次执行主查询。

主查询围绕子查询，至少需要 `SELECT` 和 `FROM` 子句。主查询支持本文档中列出的所有子句。

子查询显示在主查询的 `FROM` 子句中，它需要附加的括号。 子查询支持本文档中列出的所有子句。

InfluxQL每个主要查询支持多个嵌套子查询。 多个子查询的示例语法：

```
1. SELECT_clause FROM ( SELECT_clause FROM ( SELECT_statement ) [...] ) [...]
```

例子

例一：计算多个 `MAX()` 值的 `SUM()`

```
    > SELECT SUM("max") FROM (SELECT MAX("water_level") FROM "h2o_feet" GROUP BY
1. "location")
2.
3. name: h2o_feet
4. time                   sum
5. ----                   ---
6. 1970-01-01T00:00:00Z   17.169
```

该查询返回 `location` 的每个tag值之间的最大 `water_level` 的总和。

InfluxDB首先执行子查询；它计算每个tag值的 `water_level` 的最大值：

```
1. > SELECT MAX("water_level") FROM "h2o_feet" GROUP BY "location"
2. name: h2o_feet
```

```
3.
4.  tags: location=coyote_creek
5.  time                   max
6.  ----                   ---
7.  2015-08-29T07:24:00Z   9.964
8.
9.  name: h2o_feet
10. tags: location=santa_monica
11. time                   max
12. ----                   ---
13. 2015-08-29T03:54:00Z   7.205
```

接下来，InfluxDB执行主查询并计算这些最大值的总和：9.964 + 7.205 = 17.169。 请注意，主查询将 `max` (而不是 `water_level`)指定为 `SUM()` 函数中的字段键。

例二：计算两个field的差值的 `MEAN()`

```
    > SELECT MEAN("difference") FROM (SELECT "cats" - "dogs" AS "difference" FROM
1.  "pet_daycare")
2.
3.  name: pet_daycare
4.  time                   mean
5.  ----                   ----
6.  1970-01-01T00:00:00Z   1.75
```

查询返回measurement `pet_daycare``cats` 和 `dogs` 数量之间的差异的平均值。

InfluxDB首先执行子查询。 子查询计算 `cats` 字段中的值和 `dogs` 字段中的值之间的差值，并命名输出列 `difference` ：

```
1.  > SELECT "cats" - "dogs" AS "difference" FROM "pet_daycare"
2.
3.  name: pet_daycare
4.  time                   difference
5.  ----                   ----------
6.  2017-01-20T00:55:56Z   -1
7.  2017-01-21T00:55:56Z   -49
8.  2017-01-22T00:55:56Z   66
9.  2017-01-23T00:55:56Z   -9
```

接下来，InfluxDB执行主要查询并计算这些差的平均值。请注意，主查询指定 `difference` 作为 `MEAN()` 函数中的字段键。

例三：计算 `MEAN()` 然后将这些平均值作为条件

```
> SELECT "all_the_means" FROM (SELECT MEAN("water_level") AS "all_the_means"
FROM "h2o_feet" WHERE time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-
18T00:30:00Z' GROUP BY time(12m) ) WHERE "all_the_means" > 5

name: h2o_feet
time                   all_the_means
----                   -------------
2015-08-18T00:00:00Z   5.07625
```

该查询返回 `water_level` 的平均值大于5的所有平均值。

InfluxDB首先执行子查询。子查询从2015-08-18T00: 00: 00Z到2015-08-18T00: 30: 00Z计算 `water_level` 的 `MEAN()` 值，并将结果分组为12分钟。它也命名输出列 `all_the_means` ：

```
> SELECT MEAN("water_level") AS "all_the_means" FROM "h2o_feet" WHERE time >=
'2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z' GROUP BY time(12m)

name: h2o_feet
time                   all_the_means
----                   -------------
2015-08-18T00:00:00Z   5.07625
2015-08-18T00:12:00Z   4.950749999999999
2015-08-18T00:24:00Z   4.80675
```

接下来，InfluxDB执行主查询，只返回大于5的平均值。请注意，主查询将 `all_the_means` 指定为 `SELECT` 子句中的字段键。

例四：计算多个 `DERIVATIVE()` 值得 `SUM()`

```
> SELECT SUM("water_level_derivative") AS "sum_derivative" FROM (SELECT
DERIVATIVE(MEAN("water_level")) AS "water_level_derivative" FROM "h2o_feet"
WHERE time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z' GROUP
BY time(12m),"location") GROUP BY "location"

name: h2o_feet
tags: location=coyote_creek
time                   sum_derivative
----                   --------------
1970-01-01T00:00:00Z   -0.4950000000000001

```

```
name: h2o_feet
tags: location=santa_monica
time                   sum_derivative
----                   --------------
1970-01-01T00:00:00Z   -0.043999999999999595
```

查询返回每个tag `location` 的平均 `water_level` 的导数之和。

InfluxDB首先执行子查询。子查询计算以12分钟间隔获取的平均 `water_level` 的导数。它对 `location` 的每个tag value进行计算，并将输出列命名为 `water_level_derivative` ：

```
> SELECT DERIVATIVE(MEAN("water_level")) AS "water_level_derivative" FROM "h2o_feet" WHERE time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z' GROUP BY time(12m),"location"

name: h2o_feet
tags: location=coyote_creek
time                   water_level_derivative
----                   ----------------------
2015-08-18T00:12:00Z   -0.23800000000000043
2015-08-18T00:24:00Z   -0.2569999999999997

name: h2o_feet
tags: location=santa_monica
time                   water_level_derivative
----                   ----------------------
2015-08-18T00:12:00Z   -0.0129999999999999
2015-08-18T00:24:00Z   -0.030999999999999694
```

接下来，InfluxDB执行主查询，并计算 `location` 的 `water_level_derivative` 值的总和。请注意，主要查询指定了 `water_level_derivative` ，而不是 `water_level` 或者 `derivative` ，作为 `SUM()` 函数中的字段键。

子查询的常见问题

子查询中多个 `SELECT` 语句

InfluxQL支持在每个主查询中嵌套多个子查询：

```
SELECT_clause FROM ( SELECT_clause FROM ( SELECT_statement ) [...] ) [...]
                     ------------------   ----------------
                        Subquery 1           Subquery 2
```

InfluxQL不支持每个子查询中多个 `SELECT` 语句:

```
1.  SELECT_clause FROM (SELECT_statement; SELECT_statement) [...]
```

如果一个子查询中多个 `SELECT` 语句，系统会返回一个解析错误。

schema查询语法

InfluxQL是一种类似SQL的查询语言，用于与InfluxDB中的数据进行交互。下面我们要介绍一些有用的查询schema的语法：

- SHOW DATABASES
- SHOW RETENTION POLICIES
- SHOW SERIES
- SHOW MEASUREMENTS
- SHOW TAG KEYS
- SHOW TAG VALUES
- SHOW FIELD KEYS

在开始之前，默认已经登入了CLI：

```
$ influx -precision rfc3339
Connected to http://localhost:8086 version 1.3.x
InfluxDB shell 1.3.x
>
```

SHOW DATABASES

返回当前实例上的所有的数据库。

语法

```
SHOW DATABASES
```

例子

例一：运行 `SHOW DATABASES` 查询

```
> SHOW DATABASES

name: databases
name
----
NOAA_water_database
_internal
```

SHOW RETENTION POLICIES

返回指定数据库的保留策略的列表。

语法

```
SHOW RETENTION POLICIES [ON <database_name>]
```

语法描述

`ON <database_name>` 是可选的。如果查询不包括 `ON <database_name>` ，则必须在CLI中使用 `USE <database_name>` 指定数据库，或者在HTTP API请求中指定 `db` 查询字符串参数。

例子

例一：运行带有 `ON` 子句的 `SHOW RETENTION POLICIES`

```
> SHOW RETENTION POLICIES ON NOAA_water_database

name      duration   shardGroupDuration   replicaN   default
----      --------   ------------------   --------   -------
autogen   0s         168h0m0s             1          true
```

该查询以表格格式返回数据库 `NOAA_water_database` 中的保留策略列表。 该数据库有一个名为 `autogen` 的保留策略。该保留策略具有无限持续时间，持续时间七天的shard group，副本数为1，并且是数据库的 `DEFAULT` 保留策略。

例二：运行不带 `ON` 子句的 `SHOW RETENTION POLICIES`

CLI

使用 `USE <database_name>` 指定数据库：

```
> USE NOAA_water_database
Using database NOAA_water_database

> SHOW RETENTION POLICIES

name      duration   shardGroupDuration   replicaN   default
----      --------   ------------------   --------   -------
```

```
8.  autogen   0s         168h0m0s             1          true
```

HTTP API

用 `db` 参数指定数据库:

```
    ~# curl -G "http://localhost:8086/query?db=NOAA_water_database&pretty=true" --
1.  data-urlencode "q=SHOW RETENTION POLICIES"
2.
3.  {
4.      "results": [
5.          {
6.              "statement_id": 0,
7.              "series": [
8.                  {
9.                      "columns": [
10.                         "name",
11.                         "duration",
12.                         "shardGroupDuration",
13.                         "replicaN",
14.                         "default"
15.                     ],
16.                     "values": [
17.                         [
18.                             "autogen",
19.                             "0s",
20.                             "168h0m0s",
21.                             1,
22.                             true
23.                         ]
24.                     ]
25.                 }
26.             ]
27.         }
28.     ]
29. }
```

SHOW SERIES

返回指定数据库的series列表。

语法

```
SHOW SERIES [ON <database_name>] [FROM_clause] [WHERE <tag_key> <operator> ['<tag_value>' | <regular_expression>]] [LIMIT_clause] [OFFSET_clause]
```

语法描述

`ON <database_name>` 是可选的。如果查询不包括 `ON <database_name>` ，则必须在CLI中使用 `USE <database_name>` 指定数据库，或者在HTTP API请求中指定 `db` 查询字符串参数。

`FROM` ， `WHERE` ， `LIMIT` 和 `OFFSET` 子句是可选的。 `WHERE` 子句支持tag比较; field比较对 `SHOW SERIES` 查询无效。

`WHERE` 子句中支持的运算符:

`=` 等于 `<>` 不等于 `!=` 不等于 `=~` 匹配 `!~` 不匹配

例子

例一: 运行带 `ON` 子句的 `SHOW SERIES`

```
> SHOW SERIES ON NOAA_water_database

key
---
average_temperature,location=coyote_creek
average_temperature,location=santa_monica
h2o_feet,location=coyote_creek
h2o_feet,location=santa_monica
h2o_pH,location=coyote_creek
h2o_pH,location=santa_monica
h2o_quality,location=coyote_creek,randtag=1
h2o_quality,location=coyote_creek,randtag=2
h2o_quality,location=coyote_creek,randtag=3
h2o_quality,location=santa_monica,randtag=1
h2o_quality,location=santa_monica,randtag=2
h2o_quality,location=santa_monica,randtag=3
h2o_temperature,location=coyote_creek
h2o_temperature,location=santa_monica
```

查询的输出类似于行协议格式。第一个逗号之前的所有内容都是measurement名称。第一个逗号后的所有内容都是tag key或tag value。 `NOAA_water_database` 有五个不同的measurement和14个不同的series。

例二：运行不带 `ON` 子句的 `SHOW SERIES`

CLI

用 `USE <database_name>` 指定数据库：

```
> USE NOAA_water_database
Using database NOAA_water_database

> SHOW SERIES

key
---
average_temperature,location=coyote_creek
average_temperature,location=santa_monica
h2o_feet,location=coyote_creek
h2o_feet,location=santa_monica
h2o_pH,location=coyote_creek
h2o_pH,location=santa_monica
h2o_quality,location=coyote_creek,randtag=1
h2o_quality,location=coyote_creek,randtag=2
h2o_quality,location=coyote_creek,randtag=3
h2o_quality,location=santa_monica,randtag=1
h2o_quality,location=santa_monica,randtag=2
h2o_quality,location=santa_monica,randtag=3
h2o_temperature,location=coyote_creek
h2o_temperature,location=santa_monica
```

HTTP API

用 `db` 参数指定数据库：

```
~# curl -G "http://localhost:8086/query?db=NOAA_water_database&pretty=true" --
data-urlencode "q=SHOW SERIES"

{
    "results": [
        {
            "statement_id": 0,
            "series": [
                {
                    "columns": [
                        "key"
```

```
                ],
                "values": [
                    [
                        "average_temperature,location=coyote_creek"
                    ],
                    [
                        "average_temperature,location=santa_monica"
                    ],
                    [
                        "h2o_feet,location=coyote_creek"
                    ],
                    [
                        "h2o_feet,location=santa_monica"
                    ],
                    [
                        "h2o_pH,location=coyote_creek"
                    ],
                    [
                        "h2o_pH,location=santa_monica"
                    ],
                    [
                        "h2o_quality,location=coyote_creek,randtag=1"
                    ],
                    [
                        "h2o_quality,location=coyote_creek,randtag=2"
                    ],
                    [
                        "h2o_quality,location=coyote_creek,randtag=3"
                    ],
                    [
                        "h2o_quality,location=santa_monica,randtag=1"
                    ],
                    [
                        "h2o_quality,location=santa_monica,randtag=2"
                    ],
                    [
                        "h2o_quality,location=santa_monica,randtag=3"
                    ],
                    [
                        "h2o_temperature,location=coyote_creek"
                    ],
                    [
```

```
                            "h2o_temperature,location=santa_monica"
                        ]
                    ]
                }
            ]
        }
    ]
}
```

例三：运行带有多个子句的 `SHOW SERIES`

```
> SHOW SERIES ON NOAA_water_database FROM "h2o_quality" WHERE "location" =
'coyote_creek' LIMIT 2

key
---
h2o_quality,location=coyote_creek,randtag=1
h2o_quality,location=coyote_creek,randtag=2
```

查询返回数据库 `NOAA_water_database` 中与measurement `h2o_quality` 相关联的并且tag为 `location = coyote_creek` 的两个series。

SHOW MEASUREMENTS

返回指定数据库的measurement列表。

语法

```
SHOW MEASUREMENTS [ON <database_name>] [WITH MEASUREMENT <regular_expression>]
[WHERE <tag_key> <operator> ['<tag_value>' | <regular_expression>]]
[LIMIT_clause] [OFFSET_clause]
```

语法描述

`ON <database_name>` 是可选的。如果查询不包括 `ON <database_name>` ，则必须在CLI中使用 `USE <database_name>` 指定数据库，或者在HTTP API请求中指定 `db` 查询字符串参数。

`WITH` ， `WHERE` ， `LIMIT` 和 `OFFSET` 子句是可选的。 `WHERE` 子句支持tag比较; field比较对 `SHOW MEASUREMENTS` 查询无效。

`WHERE` 子句中支持的运算符：

`=` 等于 `<>` 不等于 `!=` 不等于 `=~` 匹配 `!~` 不匹配

例子

例一：运行带 `ON` 子句的 `SHOW MEASUREMENTS`

```
> SHOW MEASUREMENTS ON NOAA_water_database

name: measurements
name
----
average_temperature
h2o_feet
h2o_pH
h2o_quality
h2o_temperature
```

查询返回数据库 `NOAA_water_database` 中的measurement列表。数据库有五个measurement: `average_temperature` , `h2o_feet` , `h2o_pH` , `h2o_quality` 和 `h2o_temperature` 。

例二：运行不带 `ON` 子句的 `SHOW MEASUREMENTS`

CLI

用 `USE <database_name>` 指定数据库。

```
> USE NOAA_water_database
Using database NOAA_water_database

> SHOW MEASUREMENTS
name: measurements
name
----
average_temperature
h2o_feet
h2o_pH
h2o_quality
h2o_temperature
```

HTTP API

使用参数 `db` 指定数据库：

```
~# curl -G "http://localhost:8086/query?db=NOAA_water_database&pretty=true" --data-urlencode "q=SHOW MEASUREMENTS"

{
  {
      "results": [
          {
              "statement_id": 0,
              "series": [
                  {
                      "name": "measurements",
                      "columns": [
                          "name"
                      ],
                      "values": [
                          [
                              "average_temperature"
                          ],
                          [
                              "h2o_feet"
                          ],
                          [
                              "h2o_pH"
                          ],
                          [
                              "h2o_quality"
                          ],
                          [
                              "h2o_temperature"
                          ]
                      ]
                  }
              ]
          }
      ]
  }
```

例三：运行有多个子句的 `SHOW MEASUREMENTS` (1)

```
> SHOW MEASUREMENTS ON NOAA_water_database WITH MEASUREMENT =~ /h2o.*/ LIMIT 2
OFFSET 1

name: measurements
name
----
h2o_pH
h2o_quality
```

该查询返回以以 `h2o` 开头的 `NOAA_water_database` 数据库中的measurement。 `LIMIT` 和 `OFFSET` 子句将返回的measurement名称,并且数量限制为两个，再将结果偏移一个，所以跳过了measurement `h2o_feet` 。

例四：运行有多个子句的 `SHOW MEASUREMENTS` (2)

```
> SHOW MEASUREMENTS ON NOAA_water_database WITH MEASUREMENT =~ /h2o.*/ WHERE
"randtag"  =~ /\d/

name: measurements
name
----
h2o_quality
```

该查询返回 `NOAA_water_database` 中以 `h2o` 开头，并且tag `randtag` 包含一个整数的所有measurement。

SHOW TAG KEYS

返回指定数据库的tag key列表。

语法

```
SHOW TAG KEYS [ON <database_name>] [FROM_clause] [WHERE <tag_key> <operator>
['<tag_value>' | <regular_expression>]] [LIMIT_clause] [OFFSET_clause]
```

语法描述

`ON <database_name>` 是可选的。如果查询不包括 `ON <database_name>` ，则必须在CLI中使用 `USE <database_name>` 指定数据库，或者在HTTP API请求中指定 `db` 查询字符串参数。

`FROM` 和 `WHERE` 子句是可选的。 `WHERE` 子句支持tag比较; field比较对 `SHOW TAG`

`KEYS` 查询无效。

`WHERE` 子句中支持的运算符：

`=` 等于 `<>` 不等于 `!=` 不等于 `=~` 匹配 `!~` 不匹配

例子

例一：运行带有 `ON` 子句的 `SHOW TAG KEYS`

```
1.  > SHOW TAG KEYS ON "NOAA_water_database"
2.
3.  name: average_temperature
4.  tagKey
5.  ------
6.  location
7.
8.  name: h2o_feet
9.  tagKey
10. ------
11. location
12.
13. name: h2o_pH
14. tagKey
15. ------
16. location
17.
18. name: h2o_quality
19. tagKey
20. ------
21. location
22. randtag
23.
24. name: h2o_temperature
25. tagKey
26. ------
27. location
```

查询返回数据库 `NOAA_water_database` 中的tag key列表。输出按measurement名称给tag key分组；它显示每个measurement都具有tag key `location` ，并且measurement `h2o_quality` 具有额外的tag key `randtag` 。

例二：运行不带 `ON` 子句的 `SHOW TAG KEYS`

CLI

用 `USE <database_name>` 指定数据库：

```
> USE NOAA_water_database
Using database NOAA_water_database

> SHOW TAG KEYS

name: average_temperature
tagKey
------
location

name: h2o_feet
tagKey
------
location

name: h2o_pH
tagKey
------
location

name: h2o_quality
tagKey
------
location
randtag

name: h2o_temperature
tagKey
------
location
```

HTTP API

用参数 `db` 指定数据库：

```
~# curl -G "http://localhost:8086/query?db=NOAA_water_database&pretty=true" --data-urlencode "q=SHOW TAG KEYS"
```

```
 2.
 3. {
 4.     "results": [
 5.         {
 6.             "statement_id": 0,
 7.             "series": [
 8.                 {
 9.                     "name": "average_temperature",
10.                     "columns": [
11.                         "tagKey"
12.                     ],
13.                     "values": [
14.                         [
15.                             "location"
16.                         ]
17.                     ]
18.                 },
19.                 {
20.                     "name": "h2o_feet",
21.                     "columns": [
22.                         "tagKey"
23.                     ],
24.                     "values": [
25.                         [
26.                             "location"
27.                         ]
28.                     ]
29.                 },
30.                 {
31.                     "name": "h2o_pH",
32.                     "columns": [
33.                         "tagKey"
34.                     ],
35.                     "values": [
36.                         [
37.                             "location"
38.                         ]
39.                     ]
40.                 },
41.                 {
42.                     "name": "h2o_quality",
43.                     "columns": [
```

```
                        "tagKey"
                    ],
                    "values": [
                        [
                            "location"
                        ],
                        [
                            "randtag"
                        ]
                    ]
                },
                {
                    "name": "h2o_temperature",
                    "columns": [
                        "tagKey"
                    ],
                    "values": [
                        [
                            "location"
                        ]
                    ]
                }
            ]
        }
    ]
}
```

例三：运行带有多个子句的 `SHOW TAG KEYS`

```
> SHOW TAG KEYS ON "NOAA_water_database" FROM "h2o_quality" LIMIT 1 OFFSET 1

name: h2o_quality
tagKey
------
randtag
```

该查询从数据库 `NOAA_water_database` 的measurement `h2o_quality` 中返回tag key。 `LIMIT` 和 `OFFSET` 子句限制返回到一个tag key，再将结果偏移一个。

SHOW TAG VALUES

返回数据库中指定tag key的tag value列表。

语法

```
SHOW TAG VALUES [ON <database_name>][FROM_clause] WITH KEY [ [<operator> "
<tag_key>" | <regular_expression>] | [IN ("<tag_key1>","<tag_key2")]] [WHERE
<tag_key> <operator> ['<tag_value>' | <regular_expression>]] [LIMIT_clause]
[OFFSET_clause]
```

语法描述

`ON <database_name>` 是可选的。如果查询不包括 `ON <database_name>` ，则必须在CLI中使用 `USE <database_name>` 指定数据库，或者在HTTP API请求中指定 `db` 查询字符串参数。

`WITH` 子句是必须的，它支持指定一个单独的tag key、一个表达式或是多个tag key。

`FROM` 、 `WHERE` 、 `LIMIT` 和 `OFFSET` 子句是可选的。 `WHERE` 子句支持tag比较; field比较对 `SHOW TAG KEYS` 查询无效。

`WHERE` 子句中支持的运算符:

`=` 等于 `<>` 不等于 `!=` 不等于 `=~` 匹配 `!~` 不匹配

例子

例一: 运行带有 `ON` 子句的 `SHOW TAG VALUES`

```
> SHOW TAG VALUES ON "NOAA_water_database" WITH KEY = "randtag"

name: h2o_quality
key       value
---       -----
randtag   1
randtag   2
randtag   3
```

该查询返回数据库 `NOAA_water_database` ，tag key为 `randtag` 的所有tag value。 `SHOW TAG VALUES` 将结果按measurement名字分组。

例二: 运行不带 `ON` 子句的 `SHOW TAG VALUES`

CLI

用 `USE <database_name>` 指定数据库：

```
> USE NOAA_water_database
Using database NOAA_water_database

> SHOW TAG VALUES WITH KEY = "randtag"

name: h2o_quality
key       value
---       -----
randtag   1
randtag   2
randtag   3
```

HTTP API

用参数 `db` 指定数据库：

```
~# curl -G "http://localhost:8086/query?db=NOAA_water_database&pretty=true" --data-urlencode 'q=SHOW TAG VALUES WITH KEY = "randtag"'

{
    "results": [
        {
            "statement_id": 0,
            "series": [
                {
                    "name": "h2o_quality",
                    "columns": [
                        "key",
                        "value"
                    ],
                    "values": [
                        [
                            "randtag",
                            "1"
                        ],
                        [
                            "randtag",
                            "2"
                        ],
                        [
                            "randtag",
```

```
                            "3"
                        ]
                    ]
                }
            ]
        }
    ]
}
```

例三：运行带有多个子句的 `SHOW TAG VALUES`

```
> SHOW TAG VALUES ON "NOAA_water_database" WITH KEY IN ("location","randtag")
WHERE "randtag" =~ /./ LIMIT 3

name: h2o_quality
key         value
---         -----
location    coyote_creek
location    santa_monica
randtag     1
```

该查询从数据库 `NOAA_water_database` 的所有measurement中返回tag key为 `location` 或者 `randtag` ，并且 `randtag` 的tag value不为空的tag value。 `LIMIT` 子句限制返回三个tag value。

SHOW FIELD KEYS

返回field key以及其field value的数据类型。

语法

```
SHOW FIELD KEYS [ON <database_name>] [FROM <measurement_name>]
```

语法描述

`ON <database_name>` 是可选的。如果查询不包括 `ON <database_name>` ，则必须在CLI中使用 `USE <database_name>` 指定数据库，或者在HTTP API请求中指定 `db` 查询字符串参数。

`FROM` 子句也是可选的。

例子

例一：运行一个带 `ON` 子句的 `SHOW FIELD KEYS`

```
> SHOW FIELD KEYS ON "NOAA_water_database"

name: average_temperature
fieldKey            fieldType
--------            ---------
degrees             float

name: h2o_feet
fieldKey            fieldType
--------            ---------
level description   string
water_level         float

name: h2o_pH
fieldKey            fieldType
--------            ---------
pH                  float

name: h2o_quality
fieldKey            fieldType
--------            ---------
index               float

name: h2o_temperature
fieldKey            fieldType
--------            ---------
degrees             float
```

该查询返回数据库 `NOAA_water_database` 中的每个measurement对应的field key以及其数据类型。

例二：运行一个不带 `ON` 子句的 `SHOW FIELD KEYS`

CLI

用 `USE <database_name>` 指定数据库：

```
> USE NOAA_water_database
Using database NOAA_water_database
```

```
> SHOW FIELD KEYS

name: average_temperature
fieldKey            fieldType
--------            ---------
degrees             float

name: h2o_feet
fieldKey            fieldType
--------            ---------
level description   string
water_level         float

name: h2o_pH
fieldKey            fieldType
--------            ---------
pH                  float

name: h2o_quality
fieldKey            fieldType
--------            ---------
index               float

name: h2o_temperature
fieldKey            fieldType
--------            ---------
degrees             float
```

HTTP API

用参数 `db` 指定数据库:

```
~# curl -G "http://localhost:8086/query?db=NOAA_water_database&pretty=true" --data-urlencode 'q=SHOW FIELD KEYS'

{
    "results": [
        {
            "statement_id": 0,
            "series": [
                {
                    "name": "average_temperature",
```

```
                    "columns": [
                        "fieldKey",
                        "fieldType"
                    ],
                    "values": [
                        [
                            "degrees",
                            "float"
                        ]
                    ]
                },
                {
                    "name": "h2o_feet",
                    "columns": [
                        "fieldKey",
                        "fieldType"
                    ],
                    "values": [
                        [
                            "level description",
                            "string"
                        ],
                        [
                            "water_level",
                            "float"
                        ]
                    ]
                },
                {
                    "name": "h2o_pH",
                    "columns": [
                        "fieldKey",
                        "fieldType"
                    ],
                    "values": [
                        [
                            "pH",
                            "float"
                        ]
                    ]
                },
                {
```

```
                    "name": "h2o_quality",
                    "columns": [
                        "fieldKey",
                        "fieldType"
                    ],
                    "values": [
                        [
                            "index",
                            "float"
                        ]
                    ]
                },
                {
                    "name": "h2o_temperature",
                    "columns": [
                        "fieldKey",
                        "fieldType"
                    ],
                    "values": [
                        [
                            "degrees",
                            "float"
                        ]
                    ]
                }
            ]
        }
    ]
}
```

例三：运行带有 `FROM` 子句的 `SHOW FIELD KEYS`

```
> SHOW FIELD KEYS ON "NOAA_water_database" FROM "h2o_feet"

name: h2o_feet
fieldKey            fieldType
--------            ---------
level description   string
water_level         float
```

该查询返回数据库 `NOAA_water_database` 中measurement为 `h2o_feet` 的对应的field key以及其数据类型。

`SHOW FIELD KEYS`的常见问题

问题一：`SHOW FIELD KEYS`和field 类型的差异

field value的数据类型在同一个shard里面一样但是在多个shard里面可以不同，`SHOW FIELD KEYS`遍历每个shard返回与field key相关的每种数据类型。

例子

field `all_the_types`中存储了四个不同的数据类型

```
1. > SHOW FIELD KEYS
2.
3. name: mymeas
4. fieldKey        fieldType
5. --------        ---------
6. all_the_types   integer
7. all_the_types   float
8. all_the_types   string
9. all_the_types   boolean
```

注意`SHOW FIELD KEYS`处理field的类型差异和`SELECT`语句不一样。

数据库管理

InfluxQL提供了一整套管理命令，包括数据管理和保留策略的管理。

下面的示例使用InfluxDB的命令行界面（CLI）。 您还可以使用HTTP API执行命令；只需向 `/query` 发送GET请求，并将该命令包含在URL参数 `q` 中。

> 注意：如果启用了身份验证，只有管理员可以执行本页上列出的大部分命令。

数据管理

创建数据库

语法

```
    CREATE DATABASE <database_name> [WITH [DURATION <duration>] [REPLICATION <n>]
1.  [SHARD DURATION <duration>] [NAME <retention-policy-name>]]
```

语法描述

CREATE DATABASE需要一个数据库名称。

`WITH` ， `DURATION` ， `REPLICATION` ， `SHARD DURATION` 和 `NAME` 子句是可选的用来创建与数据库相关联的单个保留策略。如果您没有在 `WITH` 之后指定其中一个子句，将默认为 `autogen` 保留策略。创建的保留策略将自动用作数据库的默认保留策略。

一个成功的 `CREATE DATABASE` 查询返回一个空的结果。如果您尝试创建已存在的数据库，InfluxDB什么都不做，也不会返回错误。

例子

例一：创建数据库

```
1.  > CREATE DATABASE "NOAA_water_database"
2.  >
```

该语句创建了一个叫做 `NOAA_water_database` 的数据库，默认InfluxDB也会创建 `autogen` 保留策略，并和数据库 `NOAA_water_database` 关联起来。

例二：创建一个有特定保留策略的数据库

```
> CREATE DATABASE "NOAA_water_database" WITH DURATION 3d REPLICATION 1 SHARD
DURATION 1h NAME "liquid"
>
```

该语句创建了一个叫做 `NOAA_water_database` 的数据库，并且创建了 `liquid` 作为数据库的默认保留策略，其持续时间为3天，副本数是1，shard group的持续时间为一个小时。

删除数据库

`DROP DATABASE` 从指定数据库删除所有的数据，以及measurement，series，continuous queries，和retention policies。语法为：

```
DROP DATABASE <database_name>
```

一个成功的 `DROP DATABASE` 查询返回一个空的结果。如果您尝试删除不存在的数据库，InfluxDB什么都不做，也不会返回错误。

用DROP从索引中删除series

`DROP SERIES` 删除一个数据库里的一个series的所有数据，并且从索引中删除series。

> `DROP SERIES` 不支持 `WHERE` 中带时间间隔。

该查询采用以下形式，您必须指定 `FROM` 子句或 `WHERE` 子句：

```
DROP SERIES FROM <measurement_name[,measurement_name]> WHERE
<tag_key>='<tag_value>'
```

从单个measurement删除所有series：

```
> DROP SERIES FROM "h2o_feet"
```

从单个measurement删除指定tag的series：

```
> DROP SERIES FROM "h2o_feet" WHERE "location" = 'santa_monica'
```

从数据库删除有指定tag的所有measurement中的所有数据：

```
> DROP SERIES WHERE "location" = 'santa_monica'
```

用DELETE删除series

`DELETE` 删除数据库中的measurement中的所有点。与 `DROP SERIES` 不同，它不会从索引中删除series，并且它支持 `WHERE` 子句中的时间间隔。

该查询采用以下格式，必须包含 `FROM` 子句或 `WHERE` 子句，或两者都有：

```
    DELETE FROM <measurement_name> WHERE [<tag_key>='<tag_value>'] | [<time
1.  interval>]
```

删除measurement `h2o_feet` 的所有相关数据：

```
1.  > DELETE FROM "h2o_feet"
```

删除measurement `h2o_quality` 并且tag `randtag` 等于3的所有数据：

```
1.  > DELETE FROM "h2o_quality" WHERE "randtag" = '3'
```

删除数据库中2016年一月一号之前的所有数据：

```
1.  > DELETE WHERE time < '2016-01-01'
```

一个成功的 `DELETE` 返回一个空的结果。 关于DELETE的注意事项：

- 当指定measurement名称时， `DELETE` 在 `FROM子` 句中支持正则表达式，并在指定tag时支持 `WHERE` 子句中的正则表达式。
- `DELETE` 不支持 `WHERE` 子句中的field。
- 如果你需要删除之后的数据点，则必须指定 `DELETE SERIES` 的时间间隔，因为其默认运行的时间为 `time <now()` 。

删除measurement

`DROP MEASUREMENT` 删除指定measurement的所有数据和series，并且从索引中删除measurement。

该语法格式为：

```
1.  DROP MEASUREMENT <measurement_name>
```

> 注意： `DROP MEASUREMENT` 删除measurement中的所有数据和series，但是不会删除相关的continuous queries。

目前，*InfluxDB*不支持在 *`DROP MEASUREMENT`* 中使用正则表达式，具体在*#4275*中查看详情。

删除shard

`DORP SHARD` 删除一个shard，也会从metastore中删除shard。格式如下：

```
1.  DROP SHARD <shard_id_number>
```

保留策略管理

以下部分介绍如何创建，更改和删除保留策略。 请注意，创建数据库时，InfluxDB会自动创建一个名为 `autogen` 的保留策略，该保留策略保留时间为无限。您可以重命名该保留策略或在配置文件中禁用其自动创建。

创建保留策略

语法

```
    CREATE RETENTION POLICY <retention_policy_name> ON <database_name> DURATION
1.  <duration> REPLICATION <n> [SHARD DURATION <duration>] [DEFAULT]
```

语法描述

`DURATION`

`DURATION` 子句确定InfluxDB保留数据的时间。 `<duration>` 是持续时间字符串或INF（无限）。 保留策略的最短持续时间为1小时，最大持续时间为INF。

`REPLICATION`

`REPLICATION` 子句确定每个点的多少独立副本存储在集群中，其中 `n` 是数据节点的数量。该子句不能用于单节点实例。

`SHARD DURATION`

`SHARD DURATION` 子句确定shard group覆盖的时间范围。 `<duration>` 是一个持续时间字符串，不支持INF（无限）持续时间。此设置是可选的。 默认情况下，shard group持续时间由保留策略的 `DURATION` 决定：

保留策略的持续时间	shard group的持续时间
< 2天	1小时
>= 2天并<=6个月	1天
> 6个月	7天

最小允许 `SHARD GROUP DURATION` 为1小时。 如果 `CREATE RETENTION POLICY` 查询尝试将 `SHARD GROUP DURATION` 设置为小于1小时且大于0，则InfluxDB会自动将 `SHARD GROUP DURATION` 设置为1h。 如果 `CREATE RETENTION POLICY` 查询尝试将 `SHARD GROUP DURATION` 设置为0，InfluxDB会根据上面列出的默认设置自动设置 `SHARD GROUP DURATION` 。

`DEFAULT`

将新的保留策略设置为数据库的默认保留策略。此设置是可选的。

例子

创建一个保留策略

```
> CREATE RETENTION POLICY "one_day_only" ON "NOAA_water_database" DURATION 1d
REPLICATION 1
>
```

该语句给数据库 `NOAA_water_database` 创建一个保留策略 `one_day_only` ，持续时间为1天，副本数为1。

创建一个默认的保留策略

```
> CREATE RETENTION POLICY "one_day_only" ON "NOAA_water_database" DURATION
23h60m REPLICATION 1 DEFAULT
>
```

该查询创建与上述示例中相同的保留策略，但将其设置为数据库的默认保留策略。

成功的 `CREATE RETENTION POLICY` 执行返回为空。如果您尝试创建与已存在的保留策略相同的保留策略，InfluxDB不会返回错误。 如果您尝试创建与现有保留策略名称相同但具有不同属性的保留策略，InfluxDB会返回错误。

> 注意：也可以在 `CREATE DATABASE` 时指定一个新的保留策略。

修改保留策略

`ALTER RETENTION POLICY` 形式如下，你必须至少指定一个属性： `DURATION` ， `REPLICATION` ， `SHARD DURATION` ,或者 `DEFAULT` ：

```
ALTER RETENTION POLICY <retention_policy_name> ON <database_name> DURATION
<duration> REPLICATION <n> SHARD DURATION <duration> DEFAULT
```

现在我们来创建一个保留策略 `what_is_time` 其持续时间为两天：

```
> CREATE RETENTION POLICY "what_is_time" ON "NOAA_water_database" DURATION 2d
REPLICATION 1
>
```

修改 `what_is_time` 的持续时间为3个星期，shard group的持续时间为30分钟，并将其作为数据库 `NOAA_water_database` 的默认保留策略：

```
> ALTER RETENTION POLICY "what_is_time" ON "NOAA_water_database" DURATION 3w
SHARD DURATION 30m DEFAULT
>
```

在这个例子中， `what_is_time` 将保留其原始副本数为1。

删除保留策略

删除指定保留策略的所有measurement和数据：

```
DROP RETENTION POLICY <retention_policy_name> ON <database_name>
```

成功的 `DROP RETENTION POLICY` 返回一个空的结果。如果您尝试删除不存在的保留策略，InfluxDB不会返回错误。

连续查询

介绍

连续查询(Continuous Queries下文统一简称CQ)是InfluxQL对实时数据自动周期运行的查询，然后把查询结果写入到指定的measurement中。

语法

基本语法

```
CREATE CONTINUOUS QUERY <cq_name> ON <database_name>
BEGIN
  <cq_query>
END
```

语法描述

cq_query

`cq_query` 需要一个函数，一个 `INTO` 子句和一个 `GROUP BY time()` 子句:

```
SELECT <function[s]> INTO <destination_measurement> FROM <measurement> [WHERE
<stuff>] GROUP BY time(<interval>)[,<tag_key[s]>]
```

> 注意: 请注意，在 `WHERE` 子句中， `cq_query` 不需要时间范围。 InfluxDB在执行CQ时自动生成 `cq_query` 的时间范围。 `cq_query` 的 `WHERE` 子句中的任何用户指定的时间范围将被系统忽略。

运行时间点以及覆盖的时间范围

CQ对实时数据进行操作。他们使用本地服务器的时间戳， `GROUP BY time()` 间隔和InfluxDB的预设时间边界来确定何时执行以及查询中涵盖的时间范围。

CQs以与 `cq_query` 的 `GROUP BY time()` 间隔相同的间隔执行，并且它们在InfluxDB的预设时间边界开始时运行。如果 `GROUP BY time()` 间隔为1小时，则CQ每小时开始执行一次。

当CQ执行时，它对于 `now()` 和 `now()` 减去 `GROUP BY time()` 间隔的时间范围运行单个查询。如果 `GROUP BY time()` 间隔为1小时，当前时间为17:00，查询的时间范围为16:00至16:59999999999。

基本语法的例子

以下例子使用数据库 `transportation` 中的示例数据。measurement `bus_data` 数据存储有关公共汽车乘客数量和投诉数量的15分钟数据：

```
name: bus_data
--------------
time                  passengers   complaints
2016-08-28T07:00:00Z  5            9
2016-08-28T07:15:00Z  8            9
2016-08-28T07:30:00Z  8            9
2016-08-28T07:45:00Z  7            9
2016-08-28T08:00:00Z  8            9
2016-08-28T08:15:00Z  15           7
2016-08-28T08:30:00Z  15           7
2016-08-28T08:45:00Z  17           7
2016-08-28T09:00:00Z  20           7
```

例一：自动采样数据

使用简单的CQ自动从单个字段中下采样数据，并将结果写入同一数据库中的另一个measurement。

```
CREATE CONTINUOUS QUERY "cq_basic" ON "transportation"
BEGIN
  SELECT mean("passengers") INTO "average_passengers" FROM "bus_data" GROUP BY time(1h)
END
```

`cq_basic` 从 `bus_data` 中计算乘客的平均小时数，并将结果存储在数据库 `transportation` 中的 `average_passengers` 中。

`cq_basic` 以一小时的间隔执行，与 `GROUP BY time()` 间隔相同的间隔。 每个小时， `cq_basic` 运行一个单一的查询，覆盖了 `now()` 和 `now()` 减去 `GROUP BY time()` 间隔之间的时间范围，即 `now()` 和 `now()` 之前的一个小时之间的时间范围。

下面是2016年8月28日上午的日志输出：

> 在8点时， `cq_basic` 执行时间范围为 `time => '7:00' AND time <'08：00'` 的查询。 `cq_basic` 向 `average_passengers` 写入一个点：
>
> ```
> name: average_passengers
> ------------------------
> time                  mean
> ```

```
4. 2016-08-28T07:00:00Z   7
```

在9点时，`cq_basic` 执行时间范围为 `time => '8:00' AND time <'09：00'` 的查询。`cq_basic` 向 `average_passengers` 写入一个点：

```
1. name: average_passengers
2. ------------------------
3. time                    mean
4. 2016-08-28T08:00:00Z   13.75
```

结果：

```
1. > SELECT * FROM "average_passengers"
2. name: average_passengers
3. ------------------------
4. time                    mean
5. 2016-08-28T07:00:00Z   7
6. 2016-08-28T08:00:00Z   13.75
```

例二：自动采样数据到另一个保留策略里

从默认的的保留策略里面采样数据到完全指定的目标measurement中：

```
1. CREATE CONTINUOUS QUERY "cq_basic_rp" ON "transportation"
2. BEGIN
     SELECT mean("passengers") INTO
   "transportation"."three_weeks"."average_passengers" FROM "bus_data" GROUP BY
3. time(1h)
4. END
```

`cq_basic_rp` 从 `bus_data` 中计算乘客的平均小时数，并将结果存储在数据库 `tansportation` 的RP为 `three_weeks` 的measurement `average_passengers` 中。

`cq_basic_rp` 以一小时的间隔执行，与 `GROUP BY time()` 间隔相同的间隔。每个小时，`cq_basic_rp` 运行一个单一的查询，覆盖了 `now()` 和 `now()` 减去 `GROUP BY time()` 间隔之间的时间段，即 `now()` 和 `now()` 之前的一个小时之间的时间范围。

下面是2016年8月28日上午的日志输出：

在8:00 `cq_basic_rp` 执行时间范围为 `time >='7:00' AND time <'8:00'` 的查询。`cq_basic_rp` 向RP为 `three_weeks` 的measurement `average_passengers` 写入一个点：

```
1. name: average_passengers
```

```
2. ------------------------
3. time                    mean
4. 2016-08-28T07:00:00Z    7
```

在9:00 `cq_basic_rp` 执行时间范围为 `time >='8:00' AND time <'9:00'` 的查询。 `cq_basic_rp` 向RP为 `three_weeks` 的measurement `average_passengers` 写入一个点：

```
1. name: average_passengers
2. ------------------------
3. time                    mean
4. 2016-08-28T08:00:00Z    13.75
```

结果：

```
1. > SELECT * FROM "transportation"."three_weeks"."average_passengers"
2. name: average_passengers
3. ------------------------
4. time                    mean
5. 2016-08-28T07:00:00Z    7
6. 2016-08-28T08:00:00Z    13.75
```

`cq_basic_rp` 使用CQ和保留策略自动降低样本数据，并将这些采样数据保留在不同的时间长度上。

例三：使用逆向引用自动采样数据

使用带有通配符（ `*` ）和 `INTO` 查询的反向引用语法的函数可自动对数据库中所有measurement和数值字段中的数据进行采样。

```
1. CREATE CONTINUOUS QUERY "cq_basic_br" ON "transportation"
2. BEGIN
     SELECT mean(*) INTO "downsampled_transportation"."autogen".:MEASUREMENT FROM
3. /.*/ GROUP BY time(30m),*
4. END
```

`cq_basic_br` 计算数据库 `transportation` 中每个measurement的30分钟平均乘客和投诉。它将结果存储在数据库 `downsampled_transportation` 中。

`cq_basic_br` 以30分钟的间隔执行，与 `GROUP BY time()` 间隔相同的间隔。每30分钟一次， `cq_basic_br` 运行一个查询，覆盖了 `now()` 和 `now()` 减去 `GROUP BY time()` 间隔之间的时间段，即 `now()` 到 `now()` 之前的30分钟之间的时间范围。

下面是2016年8月28日上午的日志输出：

```
在7:30， cq_basic_br 执行查询，时间间隔 time >='7:00' AND time
```

`<'7:30'` 。 `cq_basic_br` 向 `downsampled_transportation` 数据库中的measurement为 `bus_data` 写入两个点：

```
name: bus_data
--------------
time                   mean_complaints   mean_passengers
2016-08-28T07:00:00Z   9                 6.5
```

8点时， `cq_basic_br` 执行时间范围为 `time >='7:30' AND time <'8:00'` 的查询。 `cq_basic_br` 向 `downsampled_transportation` 数据库中measurement为 `bus_data` 写入两个点：

```
name: bus_data
--------------
time                   mean_complaints   mean_passengers
2016-08-28T07:30:00Z   9                 7.5
```

[…]

9点时， `cq_basic_br` 执行时间范围为 `time >='8:30' AND time <'9:00'` 的查询。 `cq_basic_br` 向 `downsampled_transportation` 数据库中measurement为 `bus_data` 写入两个点：

```
name: bus_data
--------------
time                   mean_complaints   mean_passengers
2016-08-28T08:30:00Z   7                 16
```

结果为：

```
> SELECT * FROM "downsampled_transportation."autogen"."bus_data"
name: bus_data
--------------
time                   mean_complaints   mean_passengers
2016-08-28T07:00:00Z   9                 6.5
2016-08-28T07:30:00Z   9                 7.5
2016-08-28T08:00:00Z   8                 11.5
2016-08-28T08:30:00Z   7                 16
```

例四：自动采样数据并配置CQ的时间边界

使用 `GROUP BY time()` 子句的偏移间隔来改变CQ的默认执行时间和呈现的时间边界：

```
CREATE CONTINUOUS QUERY "cq_basic_offset" ON "transportation"
```

```
BEGIN
  SELECT mean("passengers") INTO "average_passengers" FROM "bus_data" GROUP BY
time(1h,15m)
END
```

`cq_basic_offset` 从 `bus_data` 中计算乘客的平均小时数，并将结果存储在 `average_passengers` 中。

`cq_basic_offset` 以一小时的间隔执行，与 `GROUP BY time()` 间隔相同的间隔。15分钟偏移间隔迫使CQ在默认执行时间后15分钟执行； `cq_basic_offset` 在8:15而不是8:00执行。

每个小时， `cq_basic_offset` 运行一个单一的查询，覆盖了 `now()` 和 `now()` 减去 `GROUP BY time()` 间隔之间的时间段，即 `now()` 和 `now()` 之前的一个小时之间的时间范围。 15分钟偏移间隔在CQ的 `WHERE` 子句中向前移动生成的预设时间边界； `cq_basic_offset` 在7:15和8:14.999999999而不是7:00和7: 59.999999999之间进行查询。

下面是2016年8月28日上午的日志输出：

在8:15 `cq_basic_offset` 执行时间范围 `time> ='7:15'AND time <'8:15'` 的查询。 `cq_basic_offset` 向 `average_passengers` 写入一个点：

```
name: average_passengers
------------------------
time                    mean
2016-08-28T07:15:00Z    7.75
```

在9:15 `cq_basic_offset` 执行时间范围 `time> ='8:15'AND time <'9:15'` 的查询。 `cq_basic_offset` 向 `average_passengers` 写入一个点：

```
name: average_passengers
------------------------
time                    mean
2016-08-28T08:15:00Z    16.75
```

结果为：

```
> SELECT * FROM "average_passengers"
name: average_passengers
------------------------
time                    mean
2016-08-28T07:15:00Z    7.75
2016-08-28T08:15:00Z    16.75
```

请注意，时间戳为7:15和8:15而不是7:00和8:00。

基本语法的常见问题

问题一：无数据处理时间间隔

如果没有数据落在该时间范围内，则CQ不会在时间间隔内写入任何结果。请注意，基本语法不支持使用 `fill()` 更改不含数据的间隔报告的值。如果基本语法CQs包括了 `fill()` ，则会忽略 `fill()` 。一个解决办法是使用下面的高级语法。

问题二：重新采样以前的时间间隔

基本的CQ运行一个查询，覆盖了 `now()` 和 `now()` 减去 `GROUP BY time()` 间隔之间的时间段。有关如何配置查询的时间范围，请参阅高级语法。

问题三：旧数据的回填结果

CQ对实时数据进行操作，即具有相对于 `now()` 发生的时间戳的数据。使用基本的 `INTO` 查询来回填具有较旧时间戳的数据的结果。

问题四：CQ结果中缺少tag

默认情况下，所有 `INTO` 查询将源measurement中的任何tag转换为目标measurement中的field。

在CQ中包含 `GROUP BY *` ，以保留目的measurement中的tag。

高级语法

```
CREATE CONTINUOUS QUERY <cq_name> ON <database_name>
RESAMPLE EVERY <interval> FOR <interval>
BEGIN
  <cq_query>
END
```

高级语法描述

cq_query

同上面基本语法里面的 `cq_query` 。

运行时间点以及覆盖的时间范围

CQs对实时数据进行操作。使用高级语法，CQ使用本地服务器的时间戳以及 `RESAMPLE` 子句中的信息和InfluxDB的预设时间边界来确定执行时间和查询中涵盖的时间范围。

CQs以与 `RESAMPLE` 子句中的 `EVERY` 间隔相同的间隔执行，并且它们在InfluxDB的预设时间边界开始时运行。如果 `EVERY` 间隔是两个小时，InfluxDB将在每两小时的开始执行CQ。

当CQ执行时，它运行一个单一的查询，在 `now()` 和 `now()` 减去 `RESAMPLE` 子句中的 `FOR` 间隔之间的时间范围。如果 `FOR` 间隔为两个小时，当前时间为17:00，查询的时间间隔为15:00至16:59999999999。

`EVERY` 间隔和 `FOR` 间隔都接受时间字符串。 `RESAMPLE` 子句适用于同时配置 `EVERY` 和 `FOR` ，或者是其中之一。如果没有提供 `EVERY` 间隔或 `FOR` 间隔，则CQ默认为相关为基本语法。

高级语法例子

示例数据如下：

```
1.  name: bus_data
2.  --------------
3.  time                    passengers
4.  2016-08-28T06:30:00Z    2
5.  2016-08-28T06:45:00Z    4
6.  2016-08-28T07:00:00Z    5
7.  2016-08-28T07:15:00Z    8
8.  2016-08-28T07:30:00Z    8
9.  2016-08-28T07:45:00Z    7
10. 2016-08-28T08:00:00Z    8
11. 2016-08-28T08:15:00Z    15
12. 2016-08-28T08:30:00Z    15
13. 2016-08-28T08:45:00Z    17
14. 2016-08-28T09:00:00Z    20
```

例一：配置执行间隔

在 `RESAMPLE` 中使用 `EVERY` 来指明CQ的执行间隔。

```
1.  CREATE CONTINUOUS QUERY "cq_advanced_every" ON "transportation"
2.  RESAMPLE EVERY 30m
3.  BEGIN
      SELECT mean("passengers") INTO "average_passengers" FROM "bus_data" GROUP BY
4.  time(1h)
5.  END
```

`cq_advanced_every` 从 `bus_data` 中计算 `passengers` 的一小时平均值，并将结果存储在数据库 `transportation` 中的 `average_passengers` 中。

`cq_advanced_every` 以30分钟的间隔执行，间隔与 `EVERY` 间隔相同。每30分钟， `cq_advanced_every` 运行一个查询，覆盖当前时间段的时间范围，即与 `now()` 交叉的一小时时间段。

下面是2016年8月28日上午的日志输出：

> 在8:00 `cq_basic_every` 执行时间范围 `time> ='7:00'AND time <'8:00'` 的查询。 `cq_basic_every` 向 `average_passengers` 写入一个点：
>
> ```
> name: average_passengers
> ------------------------
> time                  mean
> 2016-08-28T07:00:00Z  7
> ```
>
> 在8:30 `cq_basic_every` 执行时间范围 `time> ='8:00'AND time <'9:00'` 的查询。 `cq_basic_every` 向 `average_passengers` 写入一个点：
>
> ```
> name: average_passengers
> ------------------------
> time                  mean
> 2016-08-28T08:00:00Z  12.6667
> ```
>
> 在9:00 `cq_basic_every` 执行时间范围 `time> ='8:00'AND time <'9:00'` 的查询。 `cq_basic_every` 向 `average_passengers` 写入一个点：
>
> ```
> name: average_passengers
> ------------------------
> time                  mean
> 2016-08-28T08:00:00Z  13.75
> ```

结果为：

```
> SELECT * FROM "average_passengers"
name: average_passengers
------------------------
time                  mean
2016-08-28T07:00:00Z  7
2016-08-28T08:00:00Z  13.75
```

请注意， `cq_advanced_every` 计算8:00时间间隔的结果两次。第一次，它运行在8:30，计算每个可用数据点在8:00和9:00（8,15和15）之间的平均值。 第二次，它运行在9:00，计算每个可用数据点在8:00和9:00（8,15,15和17）之间的平均值。由于InfluxDB处理重复点的方式，所以第二个结果只是覆盖第一个结果。

例二：配置CQ的重采样时间范围

在 `RESAMPLE` 中使用 `FOR` 来指明CQ的时间间隔的长度。

```
CREATE CONTINUOUS QUERY "cq_advanced_for" ON "transportation"
RESAMPLE FOR 1h
BEGIN
  SELECT mean("passengers") INTO "average_passengers" FROM "bus_data" GROUP BY time(30m)
END
```

`cq_advanced_for` 从 `bus_data` 中计算 `passengers` 的30分钟平均值，并将结果存储在数据库 `transportation` 中的 `average_passengers` 中。

`cq_advanced_for` 以30分钟的间隔执行，间隔与 `GROUP BY time()` 间隔相同。每30分钟， `cq_advanced_for` 运行一个查询，覆盖时间段为 `now()` 和 `now()` 减去 `FOR` 中的间隔，即是 `now()` 和 `now()` 之前的一个小时之间的时间范围。

下面是2016年8月28日上午的日志输出：

在8:00 `cq_advanced_for` 执行时间范围 `time> ='7:00'AND time <'8:00'` 的查询。 `cq_advanced_for` 向 `average_passengers` 写入两个点：

```
name: average_passengers
------------------------
time                  mean
2016-08-28T07:00:00Z  6.5
2016-08-28T07:30:00Z  7.5
```

在8:30 `cq_advanced_for` 执行时间范围 `time> ='7:30'AND time <'8:30'` 的查询。 `cq_advanced_for` 向 `average_passengers` 写入两个点：

```
name: average_passengers
------------------------
time                  mean
2016-08-28T07:30:00Z  7.5
2016-08-28T08:00:00Z  11.5
```

在9:00 `cq_advanced_for` 执行时间范围 `time> ='8:00'AND time <'9:00'` 的查询。 `cq_advanced_for` 向 `average_passengers` 写入两个点：

```
name: average_passengers
------------------------
time                  mean
```

```
4. 2016-08-28T08:00:00Z   11.5
5. 2016-08-28T08:30:00Z   16
```

请注意， `cq_advanced_for` 会计算每次间隔两次的结果。CQ在8:00和8:30计算7:30的平均值，在8:30和9:00计算8:00的平均值。

结果为：

```
1. > SELECT * FROM "average_passengers"
2. name: average_passengers
3. ------------------------
4. time                    mean
5. 2016-08-28T07:00:00Z   6.5
6. 2016-08-28T07:30:00Z   7.5
7. 2016-08-28T08:00:00Z   11.5
8. 2016-08-28T08:30:00Z   16
```

例三：配置执行间隔和CQ时间范围

在 `RESAMPLE` 子句中使用 `EVERY` 和 `FOR` 来指定CQ的执行间隔和CQ的时间范围长度。

```
1. CREATE CONTINUOUS QUERY "cq_advanced_every_for" ON "transportation"
2. RESAMPLE EVERY 1h FOR 90m
3. BEGIN
     SELECT mean("passengers") INTO "average_passengers" FROM "bus_data" GROUP BY
4. time(30m)
5. END
```

`cq_advanced_every_for` 从 `bus_data` 中计算 `passengers` 的30分钟平均值，并将结果存储在数据库 `transportation` 中的 `average_passengers` 中。

`cq_advanced_every_for` 以1小时的间隔执行，间隔与 `EVERY` 间隔相同。每1小时， `cq_advanced_every_for` 运行一个查询，覆盖时间段为 `now()` 和 `now()` 减去 `FOR` 中的间隔，即是 `now()` 和 `now()` 之前的90分钟之间的时间范围。

下面是2016年8月28日上午的日志输出：

```
在8:00 cq_advanced_every_for 执行时间范围 time>='6:30'AND time <'8:00' 的查
询。 cq_advanced_every_for 向 average_passengers 写三个个点:

1. name: average_passengers
2. ------------------------
3. time                    mean
```

```
2016-08-28T06:30:00Z   3
2016-08-28T07:00:00Z   6.5
2016-08-28T07:30:00Z   7.5
```

在9:00 `cq_advanced_every_for` 执行时间范围 `time> ='7:30'AND time <'9:00'` 的查询。 `cq_advanced_every_for` 向 `average_passengers` 写入三个点：

```
name: average_passengers
------------------------
time                    mean
2016-08-28T07:30:00Z   7.5
2016-08-28T08:00:00Z   11.5
2016-08-28T08:30:00Z   16
```

请注意， `cq_advanced_every_for` 会计算每次间隔两次的结果。CQ在8:00和9:00计算7:30的平均值。

结果为：

```
> SELECT * FROM "average_passengers"
name: average_passengers
------------------------
time                    mean
2016-08-28T06:30:00Z   3
2016-08-28T07:00:00Z   6.5
2016-08-28T07:30:00Z   7.5
2016-08-28T08:00:00Z   11.5
2016-08-28T08:30:00Z   16
```

例四：配置CQ的时间范围并填充空值

使用 `FOR` 间隔和 `fill()` 来更改不含数据的时间间隔值。请注意，至少有一个数据点必须在 `fill()` 运行的 `FOR` 间隔内。 如果没有数据落在 `FOR` 间隔内，则CQ不会将任何点写入目标measurement。

```
CREATE CONTINUOUS QUERY "cq_advanced_for_fill" ON "transportation"
RESAMPLE FOR 2h
BEGIN
  SELECT mean("passengers") INTO "average_passengers" FROM "bus_data" GROUP BY
time(1h) fill(1000)
END
```

`cq_advanced_for_fill` 从 `bus_data` 中计算 `passengers` 的1小时的平均值，并将结果存储在

数据库 `transportation` 中的 `average_passengers` 中。并会在没有结果的时间间隔里写入值 `1000` 。

`cq_advanced_for_fill` 以1小时的间隔执行，间隔与 `GROUP BY time()` 间隔相同。每1小时， `cq_advanced_for_fill` 运行一个查询，覆盖时间段为 `now()` 和 `now()` 减去 `FOR` 中的间隔，即是 `now()` 和 `now()` 之前的两小时之间的时间范围。

下面是2016年8月28日上午的日志输出：

> 在6:00 `cq_advanced_for_fill` 执行时间范围 `time>='4:00'AND time <'6:00'` 的查询。 `cq_advanced_for_fill` 向 `average_passengers` 不写入任何点，因为在那个时间范围 `bus_data` 没有数据：
>
> 在7:00 `cq_advanced_for_fill` 执行时间范围 `time>='5:00'AND time <'7:00'` 的查询。 `cq_advanced_for_fill` 向 `average_passengers` 写入两个点：
>
> ```
> 1. name: average_passengers
> 2. ------------------------
> 3. time                    mean
> 4. 2016-08-28T05:00:00Z    1000          <------ fill(1000)
> 5. 2016-08-28T06:00:00Z    3             <------ 2和4的平均值
> ```
>
> […]
>
> 在11:00 `cq_advanced_for_fill` 执行时间范围 `time> ='9:00'AND time <'11:00'` 的查询。 `cq_advanced_for_fill` 向 `average_passengers` 写入两个点：
>
> ```
> 1. name: average_passengers
> 2. ------------------------
> 3. 2016-08-28T09:00:00Z    20            <------ 20的平均
> 4. 2016-08-28T10:00:00Z    1000          <------ fill(1000)
> ```
>
> 在12:00 `cq_advanced_for_fill` 执行时间范围 `time>='10:00'AND time <'12:00'` 的查询。 `cq_advanced_for_fill` 向 `average_passengers` 不写入任何点，因为在那个时间范围 `bus_data` 没有数据.

结果：

```
1. > SELECT * FROM "average_passengers"
2. name: average_passengers
3. ------------------------
4. time                    mean
5. 2016-08-28T05:00:00Z    1000
6. 2016-08-28T06:00:00Z    3
7. 2016-08-28T07:00:00Z    7
```

```
2016-08-28T08:00:00Z   13.75
2016-08-28T09:00:00Z   20
2016-08-28T10:00:00Z   1000
```

> 注意：如果前一个值在查询时间之外，则 `fill(previous)` 不会在时间间隔里填充数据。

高级语法的常见问题

问题一：如果 `EVERY` 间隔大于 `GROUP BY time()` 的间隔

如果 `EVERY` 间隔大于 `GROUP BY time()` 间隔，则CQ以与 `EVERY` 间隔相同的间隔执行，并运行一个单个查询，该查询涵盖 `now()` 和 `now()` 减去 `EVERY` 间隔之间的时间范围(不是在 `now()` 和 `now()` 减去 `GROUP BY time()` 间隔之间)。

例如，如果 `GROUP BY time()` 间隔为5m，并且 `EVERY` 间隔为10m，则CQ每10分钟执行一次。每10分钟，CQ运行一个查询，覆盖 `now()` 和 `now()` 减去 `EVERY` 间隔之间的时间段，即 `now()` 到 `now()` 之前十分钟之间的时间范围。

此行为是故意的，并防止CQ在执行时间之间丢失数据。

问题二：如果 `FOR` 间隔比执行的间隔少

如果 `FOR` 间隔比 `GROUP BY time()` 或者 `EVERY` 的间隔少，InfluxDB返回如下错误：

```
error parsing query: FOR duration must be >= GROUP BY time duration: must be a
minimum of <minimum-allowable-interval> got <user-specified-interval>
```

为了避免在执行时间之间丢失数据， `FOR` 间隔必须等于或大于 `GROUP BY time()` 或者 `EVERY` 间隔。

目前，这是预期的行为。GitHub上Issue＃6963要求CQ支持数据覆盖的差距。

CQ的管理

只有admin用户允许管理CQ。

列出CQ

列出InfluxDB实例上的所有CQ：

```
SHOW CONTINUOUS QUERIES
```

`SHOW CONTINUOUS QUERIES` 按照database作分组。

例子

下面展示了 `telegraf` 和 `mydb` 的CQ:

```
> SHOW CONTINUOUS QUERIES
name: _internal
---------------
name   query


name: telegraf
--------------
name           query
idle_hands     CREATE CONTINUOUS QUERY idle_hands ON telegraf BEGIN SELECT min(usage_idle) INTO telegraf.autogen.min_hourly_cpu FROM telegraf.autogen.cpu GROUP BY time(1h) END
feeling_used   CREATE CONTINUOUS QUERY feeling_used ON telegraf BEGIN SELECT mean(used) INTO downsampled_telegraf.autogen.:MEASUREMENT FROM telegraf.autogen./.*/ GROUP BY time(1h) END


name: downsampled_telegraf
--------------------------
name   query


name: mydb
----------
name      query
vampire   CREATE CONTINUOUS QUERY vampire ON mydb BEGIN SELECT count(dracula) INTO mydb.autogen.all_of_them FROM mydb.autogen.one GROUP BY time(5m) END
```

删除CQ

从一个指定的database删除CQ:

```
DROP CONTINUOUS QUERY <cq_name> ON <database_name>
```

`DROP CONTINUOUS QUERY` 返回一个空的结果。

例子

从数据库 `telegraf` 中删除 `idle_hands` 这个CQ:

```
1.  > DROP CONTINUOUS QUERY "idle_hands" ON "telegraf"`
2.  >
```

修改CQ

CQ一旦创建就不能修改了，你必须 `DROP` 再 `CREATE` 才行。

CQ的使用场景

采样和数据保留

使用CQ与InfluxDB的保留策略（RP）来减轻存储问题。结合CQ和RP自动将高精度数据降低到较低的精度，并从数据库中移除可分配的高精度数据。

预先计算昂贵的查询

通过使用CQ预先计算昂贵的查询来缩短查询运行时间。使用CQ自动将普通查询的高精度数据下采样到较低的精度。较低精度数据的查询需要更少的资源并且返回更快。

提示：预先计算首选图形工具的查询，以加速图形和仪表板的展示。

替换HAVING子句

InfluxQL不支持 `HAVING` 子句。通过创建CQ来聚合数据并查询CQ结果以达到应用 `HAVING` 子句相同的功能。

> 注意：InfluxDB提供了子查询也可以达到类似于 `HAVING` 相同的功能。

例子

InfluxDB不接受使用 `HAVING` 子句的以下查询。该查询以30分钟间隔计算平均 `bees` 数，并请求大于20的平均值。

```
1.  SELECT mean("bees") FROM "farm" GROUP BY time(30m) HAVING mean("bees") > 20
```

要达到相同的结果：

1. 创建一个CQ

此步骤执行以上查询的 `mean("bees")` 部分。因为这个步骤创建了CQ，所以只需要执行一次。

以下CQ自动以30分钟间隔计算 `bees` 的平均数，并将这些平均值写入

measurement `aggregate_bees` 中的 `mean_bees` 字段。

```
    CREATE CONTINUOUS QUERY "bee_cq" ON "mydb" BEGIN SELECT mean("bees") AS
1.  "mean_bees" INTO "aggregate_bees" FROM "farm" GROUP BY time(30m) END
```

2. 查询CQ的结果

这一步要实现 `HAVING mean("bees") > 20` 部分的查询。

在 `WHERE` 子句中查询measurement `aggregate_bees` 中的数据和大于20的 `mean_bees` 字段的请求值：

```
1.  SELECT "mean_bees" FROM "aggregate_bees" WHERE "mean_bees" > 20
```

替换嵌套函数

一些InfluxQL函数支持嵌套其他函数，大多数是不行的。如果函数不支持嵌套，可以使用CQ获得相同的功能来计算最内部的函数。然后简单地查询CQ结果来计算最外层的函数。

> 注意：InfluxQL支持也提供与嵌套函数相同功能的子查询。

例子

InfluxDB不接受使用嵌套函数的以下查询。 该查询以30分钟间隔计算 `bees` 的非空值数量，并计算这些计数的平均值：

```
1.  SELECT mean(count("bees")) FROM "farm" GROUP BY time(30m)
```

为了得到结果：

1. 创建一个CQ

此步骤执行上面的嵌套函数的 `count(“bees”)` 部分 因为这个步骤创建了一个CQ，所以只需要执行一次。 以下CQ自动以30分钟间隔计算 `bees` 的非空值数，并将这些计数写入 `aggregate_bees` 中的 `count_bees` 字段。

```
    CREATE CONTINUOUS QUERY "bee_cq" ON "mydb" BEGIN SELECT count("bees") AS
1.  "count_bees" INTO "aggregate_bees" FROM "farm" GROUP BY time(30m) END
```

2. 查询CQ的结果

此步骤执行上面的嵌套函数的 `mean([...])` 部分。 在 `aggregate_bees` 中查询数据，以计算 `count_bees` 字段的平均值：

```
SELECT mean("count_bees") FROM "aggregate_bees" WHERE time >= <start_time> AND
time <= <end_time>
```

函数

InfluxDB的函数可以分成Aggregate，select和predict类型。

Aggregations

COUNT()

返回非空字段值得数目

语法

```
SELECT COUNT( [ * | <field_key> | /<regular_expression>/ ] ) [INTO_clause]
FROM_clause [WHERE_clause] [GROUP_BY_clause] [ORDER_BY_clause] [LIMIT_clause]
[OFFSET_clause] [SLIMIT_clause] [SOFFSET_clause]
```

嵌套语法

```
SELECT COUNT(DISTINCT( [ * | <field_key> | /<regular_expression>/ ] )) [...]
```

语法描述

`COUNT(field_key)`

返回field key对应的field values的数目。

`COUNT(/regular_expression/)`

返回匹配正则表达式的field key对应的field values的数目。

`COUNT(*)`

返回measurement中的每个field key对应的field value的数目。

`COUNT()` 支持所有数据类型的field value，InfluxQL支持 `COUNT()` 嵌套 `DISTINCT()` 。

例子

例一：计数指定field key的field value的数目

```
> SELECT COUNT("water_level") FROM "h2o_feet"

```

```
name: h2o_feet
time                   count
----                   -----
1970-01-01T00:00:00Z   15258
```

该查询返回measurement `h2o_feet` 中的 `water_level` 的非空字段值的数量。

例二：计数measurement中每个field key关联的field value的数量

```
> SELECT COUNT(*) FROM "h2o_feet"

name: h2o_feet
time                   count_level description   count_water_level
----                   -----------------------   -----------------
1970-01-01T00:00:00Z   15258                     15258
```

该查询返回与measurement `h2o_feet` 相关联的每个字段键的非空字段值的数量。 `h2o_feet` 有两个字段键： `level_description` 和 `water_level` 。

例三：计数匹配一个正则表达式的每个field key关联的field value的数目

```
> SELECT COUNT(/water/) FROM "h2o_feet"

name: h2o_feet
time                   count_water_level
----                   -----------------
1970-01-01T00:00:00Z   15258
```

该查询返回measurement `h2o_feet` 中包含 `water` 单词的每个field key的非空字段值的数量。

例四：计数包括多个子句的field key的field value的数目

```
> SELECT COUNT("water_level") FROM "h2o_feet" WHERE time >= '2015-08-17T23:48:00Z' AND time <= '2015-08-18T00:54:00Z' GROUP BY time(12m),* fill(200) LIMIT 7 SLIMIT 1

name: h2o_feet
tags: location=coyote_creek
time                   count
----                   -----
2015-08-17T23:48:00Z   200
2015-08-18T00:00:00Z   2
2015-08-18T00:12:00Z   2
```

```
10. 2015-08-18T00:24:00Z   2
11. 2015-08-18T00:36:00Z   2
12. 2015-08-18T00:48:00Z   2
```

该查询返回 `water_level` 字段键中的非空字段值的数量。它涵盖 `2015-08-17T23：48：00Z` 和 `2015-08-18T00：54：00Z` 之间的时间段，并将结果分组为12分钟的时间间隔和每个tag。并用 `200` 填充空的时间间隔，并将点数返回7measurement返回1。

例五：计数一个field key的distinct的field value的数量

```
1. > SELECT COUNT(DISTINCT("level description")) FROM "h2o_feet"
2.
3. name: h2o_feet
4. time                   count
5. ----                   -----
6. 1970-01-01T00:00:00Z   4
```

查询返回measurement为 `h2o_feet` field key为 `level description` 的唯一field value的数量。

COUNT()的常见问题

问题一：COUNT()和fill()

大多数InfluxQL函数对于没有数据的时间间隔返回 `null` 值， `fill(<fill_option>)` 将该 `null` 值替换为 `fill_option` 。 `COUNT()` 针对没有数据的时间间隔返回 `0` ， `fill(<fill_option>)` 用 `fill_option` 替换0值。

例如

下面的代码块中的第一个查询不包括 `fill()` 。最后一个时间间隔没有数据，因此该时间间隔的值返回为零。第二个查询包括 `fill(800000)` ；它将最后一个间隔中的零替换为800000。

```
    > SELECT COUNT("water_level") FROM "h2o_feet" WHERE time >= '2015-09-
1. 18T21:24:00Z' AND time <= '2015-09-18T21:54:00Z' GROUP BY time(12m)
2.
3. name: h2o_feet
4. time                   count
5. ----                   -----
6. 2015-09-18T21:24:00Z   2
7. 2015-09-18T21:36:00Z   2
8. 2015-09-18T21:48:00Z   0
9.
```

```
    > SELECT COUNT("water_level") FROM "h2o_feet" WHERE time >= '2015-09-
    18T21:24:00Z' AND time <= '2015-09-18T21:54:00Z' GROUP BY time(12m)
10. fill(800000)
11.
12. name: h2o_feet
13. time                   count
14. ----                   -----
15. 2015-09-18T21:24:00Z   2
16. 2015-09-18T21:36:00Z   2
17. 2015-09-18T21:48:00Z   800000
```

DISTINCT()

返回field value的不同值列表。

语法

```
    SELECT DISTINCT( [ * | <field_key> | /<regular_expression>/ ] ) FROM_clause
    [WHERE_clause] [GROUP_BY_clause] [ORDER_BY_clause] [LIMIT_clause]
1.  [OFFSET_clause] [SLIMIT_clause] [SOFFSET_clause]
```

嵌套语法

```
1.  SELECT COUNT(DISTINCT( [ * | <field_key> | /<regular_expression>/ ] )) [...]
```

语法描述

`DISTINCT(field_key)`

返回field key对应的不同field values。

`DISTINCT(/regular_expression/)`

返回匹配正则表达式的field key对应的不同field values。

`DISTINCT(*)`

返回measurement中的每个field key对应的不同field value。

`DISTINCT()` 支持所有数据类型的field value，InfluxQL支持 `COUNT()` 嵌套 `DISTINCT()` 。

例子

例一：列出一个field key的不同的field value

```
> SELECT DISTINCT("level description") FROM "h2o_feet"

name: h2o_feet
time                   distinct
----                   --------
1970-01-01T00:00:00Z   between 6 and 9 feet
1970-01-01T00:00:00Z   below 3 feet
1970-01-01T00:00:00Z   between 3 and 6 feet
1970-01-01T00:00:00Z   at or greater than 9 feet
```

查询返回 `level description` 的所有的不同的值。

例二：列出一个measurement中每个field key的不同值

```
> SELECT DISTINCT(*) FROM "h2o_feet"

name: h2o_feet
time                   distinct_level description   distinct_water_level
----                   --------------------------   --------------------
1970-01-01T00:00:00Z   between 6 and 9 feet         8.12
1970-01-01T00:00:00Z   between 3 and 6 feet         8.005
1970-01-01T00:00:00Z   at or greater than 9 feet    7.887
1970-01-01T00:00:00Z   below 3 feet                 7.762
[...]
```

查询返回 `h2o_feet` 中每个字段的唯一字段值的列表。 `h2o_feet` 有两个字段： `description` 和 `water_level` 。

例三：列出匹配正则表达式的field的不同field value

```
> SELECT DISTINCT(/description/) FROM "h2o_feet"

name: h2o_feet
time                   distinct_level description
----                   --------------------------
1970-01-01T00:00:00Z   below 3 feet
1970-01-01T00:00:00Z   between 6 and 9 feet
1970-01-01T00:00:00Z   between 3 and 6 feet
1970-01-01T00:00:00Z   at or greater than 9 feet
```

查询返回 `h2o_feet` 中含有 `description` 的字段的唯一字段值的列表。

例四：列出包含多个子句的field key关联的不同值得列表

```
> SELECT DISTINCT("level description") FROM "h2o_feet" WHERE time >= '2015-08-17T23:48:00Z' AND time <= '2015-08-18T00:54:00Z' GROUP BY time(12m),* SLIMIT 1

name: h2o_feet
tags: location=coyote_creek
time                   distinct
----                   --------
2015-08-18T00:00:00Z   between 6 and 9 feet
2015-08-18T00:12:00Z   between 6 and 9 feet
2015-08-18T00:24:00Z   between 6 and 9 feet
2015-08-18T00:36:00Z   between 6 and 9 feet
2015-08-18T00:48:00Z   between 6 and 9 feet
```

该查询返回 `level description` 字段键中不同字段值的列表。它涵盖 `2015-08-17T23：48：00Z` 和 `2015-08-18T00：54：00Z` 之间的时间段，并将结果按12分钟的时间间隔和每个tag分组。查询限制返回一个series。

例五：对一个字段的不同值作计数

```
> SELECT COUNT(DISTINCT("level description")) FROM "h2o_feet"

name: h2o_feet
time                   count
----                   -----
1970-01-01T00:00:00Z   4
```

查询返回 `h2o_feet` 这个measurement中字段 `level description` 的不同值的数目。

DISTINCT()的常见问题

问题一：DISTINCT()和INTO子句

使用 `DISTINCT()` 与 `INTO` 子句可能导致InfluxDB覆盖目标measurement中的点。 `DISTINCT()` 通常返回多个具有相同时间戳的结果；InfluxDB假设具有相同series的点，时间戳是重复的点，并且仅覆盖目的measurement中最近一个点的任何重复点。

例如

下面的代码中的第一个查询使用 `DISTINCT()` 函数，返回四个结果。请注意，每个结果具有相同的时间戳。第二个查询将 `INTO` 子句添加到初始查询中，并将查询结果写入measurement `distincts` 中。代码中的最后一个查询选择 `distincts` 中的所有数据。最后一个查询返回一个点，因为四个初始结果是重复点；它们属于同一series，具有相同的时间戳。 当系统遇到重复点时，它会用最近一个点覆盖上一个点。

```
> SELECT DISTINCT("level description") FROM "h2o_feet"

name: h2o_feet
time                   distinct
----                   --------
1970-01-01T00:00:00Z   below 3 feet
1970-01-01T00:00:00Z   between 6 and 9 feet
1970-01-01T00:00:00Z   between 3 and 6 feet
1970-01-01T00:00:00Z   at or greater than 9 feet

> SELECT DISTINCT("level description") INTO "distincts" FROM "h2o_feet"

name: result
time                   written
----                   -------
1970-01-01T00:00:00Z   4

> SELECT * FROM "distincts"

name: distincts
time                   distinct
----                   --------
1970-01-01T00:00:00Z   at or greater than 9 feet
```

INTEGRAL()

返回字段曲线下的面积，即是积分。

语法

```
SELECT INTEGRAL( [ * | <field_key> | /<regular_expression>/ ] [ , <unit> ]  )
[INTO_clause] FROM_clause [WHERE_clause] [GROUP_BY_clause] [ORDER_BY_clause]
[LIMIT_clause] [OFFSET_clause] [SLIMIT_clause] [SOFFSET_clause]
```

语法描述

InfluxDB计算字段曲线下的面积，并将这些结果转换为每 `unit` 的总和面积。 `unit` 参数是一个整数，后跟一个时间字符串，它是可选的。如果查询未指定单位，则单位默认为1秒（ `1s` ）。

`INTEGRAL(field_key)`

返回field key关联的值之下的面积。

`INTEGRAL(/regular_expression/)`

返回满足正则表达式的每个field key关联的值之下的面积。

`INTEGRAL(*)`

返回measurement中每个field key关联的值之下的面积。

`INTEGRAL()` 不支持 `fill()` ， `INTEGRAL()` 支持int64和float64两个数据类型。

例子

下面的五个例子，使用数据库 `NOAA_water_database` 中的数据：

```
> SELECT "water_level" FROM "h2o_feet" WHERE "location" = 'santa_monica' AND
time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z'

name: h2o_feet
time                   water_level
----                   -----------
2015-08-18T00:00:00Z   2.064
2015-08-18T00:06:00Z   2.116
2015-08-18T00:12:00Z   2.028
2015-08-18T00:18:00Z   2.126
2015-08-18T00:24:00Z   2.041
2015-08-18T00:30:00Z   2.051
```

例一：计算指定的field key的值得积分

```
> SELECT INTEGRAL("water_level") FROM "h2o_feet" WHERE "location" =
'santa_monica' AND time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-
18T00:30:00Z'

name: h2o_feet
time                   integral
----                   --------
1970-01-01T00:00:00Z 3732.66
```

该查询返回 `h2o_feet` 中的字段 `water_level` 的曲线下的面积（以秒为单位）。

例二：计算指定的field key和时间单位的值得积分

```
> SELECT INTEGRAL("water_level",1m) FROM "h2o_feet" WHERE "location" =
'santa_monica' AND time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-
18T00:30:00Z'
```

```
2.
3.  name: h2o_feet
4.  time                   integral
5.  ----                   --------
6.  1970-01-01T00:00:00Z 62.211
```

该查询返回 `h2o_feet` 中的字段 `water_level` 的曲线下的面积（以分钟为单位）。

例三：计算measurement中每个field key在指定时间单位的值得积分

```
    > SELECT INTEGRAL(*,1m) FROM "h2o_feet" WHERE "location" = 'santa_monica' AND
1.  time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z'
2.
3.  name: h2o_feet
4.  time                   integral_water_level
5.  ----                   --------------------
6.  1970-01-01T00:00:00Z 62.211
```

查询返回measurement `h2o_feet` 中存储的每个数值字段相关的字段值的曲线下面积（以分钟为单位）。 `h2o_feet` 的数值字段为 `water_level` 。

例四：计算measurement中匹配正则表达式的field key在指定时间单位的值得积分

```
    > SELECT INTEGRAL(/water/,1m) FROM "h2o_feet" WHERE "location" = 'santa_monica'
1.  AND time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z'
2.
3.  name: h2o_feet
4.  time                   integral_water_level
5.  ----                   --------------------
6.  1970-01-0
```

查询返回field key包括单词 `water` 的每个数值类型的字段相关联的字段值的曲线下的区域（以分钟为单位）。

例五：在含有多个子句中计算指定字段的积分

```
    > SELECT INTEGRAL("water_level",1m) FROM "h2o_feet" WHERE "location" =
    'santa_monica' AND time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-
1.  18T00:30:00Z' GROUP BY time(12m) LIMIT 1
2.
3.  name: h2o_feet
4.  time                   integral
5.  ----                   --------
```

```
6. 2015-08-18T00:00:00Z 24.972
```

查询返回与字段 `water_level` 相关联的字段值的曲线下面积（以分钟为单位）。 它涵盖 `2015-08-18T00：00：00Z` 和 `2015-08-18T00：30：00Z` 之间的时间段，分组结果间隔12分钟，并将结果数量限制为1。

MEAN()

返回字段的平均值

语法

```
    SELECT MEAN( [ * | <field_key> | /<regular_expression>/ ] ) [INTO_clause]
    FROM_clause [WHERE_clause] [GROUP_BY_clause] [ORDER_BY_clause] [LIMIT_clause]
1.  [OFFSET_clause] [SLIMIT_clause] [SOFFSET_clause]
```

语法描述

`MEAN(field_key)`

返回field key关联的值的平均值。

`MEAN(/regular_expression/)`

返回满足正则表达式的每个field key关联的值的平均值。

`MEAN(*)`

返回measurement中每个field key关联的值的平均值。

`MEAN()` 支持int64和float64两个数据类型。

例子

例一：计算指定字段的平均值

```
1. > SELECT MEAN("water_level") FROM "h2o_feet"
2.
3. name: h2o_feet
4. time                   mean
5. ----                   ----
6. 1970-01-01T00:00:00Z   4.442107025822522
```

该查询返回measurement `h2o_feet` 的字段 `water_level` 的平均值。

例二：计算measurement中每个字段的平均值

```
> SELECT MEAN(*) FROM "h2o_feet"

name: h2o_feet
time                   mean_water_level
----                   ----------------
1970-01-01T00:00:00Z   4.442107025822522
```

查询返回在 `h2o_feet` 中数值类型的每个字段的平均值。 `h2o_feet` 有一个数值字段： `water_level` 。

例三：计算满足正则表达式的字段的平均值

```
> SELECT MEAN(/water/) FROM "h2o_feet"

name: h2o_feet
time                   mean_water_level
----                   ----------------
1970-01-01T00:00:00Z   4.442107025822523
```

查询返回在 `h2o_feet` 中字段中含有 `water` 的数值类型字段的平均值。

例四：计算含有多个子句字段的平均值

```
> SELECT MEAN("water_level") FROM "h2o_feet" WHERE time >= '2015-08-17T23:48:00Z' AND time <= '2015-08-18T00:54:00Z' GROUP BY time(12m),*fill(9.01) LIMIT 7 SLIMIT 1

name: h2o_feet
tags: location=coyote_creek
time                   mean
----                   ----
2015-08-17T23:48:00Z   9.01
2015-08-18T00:00:00Z   8.0625
2015-08-18T00:12:00Z   7.8245
2015-08-18T00:24:00Z   7.5675
2015-08-18T00:36:00Z   7.303
2015-08-18T00:48:00Z   7.046
```

查询返回字段 `water_level` 中的值的平均值。它涵盖 `2015-08-17T23：48：00Z` 和 `2015-08-18T00：54：00Z` 之间的时间段，并将结果按12分钟的时间间隔和每个tag分组。该查询用 `9.01` 填充空时间间隔，并将点数和series分别限制到7和1。

MEDIAN()

返回排好序的字段的中位数。

语法

```
SELECT MEDIAN( [ * | <field_key> | /<regular_expression>/ ] ) [INTO_clause]
FROM_clause [WHERE_clause] [GROUP_BY_clause] [ORDER_BY_clause] [LIMIT_clause]
[OFFSET_clause] [SLIMIT_clause] [SOFFSET_clause]
```

语法描述

`MEDIAN(field_key)`

返回field key关联的值的中位数。

`MEDIAN(/regular_expression/)`

返回满足正则表达式的每个field key关联的值的中位数。

`MEDIAN(*)`

返回measurement中每个field key关联的值的中位数。

`MEDIAN()` 支持int64和float64两个数据类型。

> 注意: `MEDIAN()` 近似于 `PERCENTILE(field_key, 50)` ，除了如果该字段包含偶数个值， `MEDIAN()` 返回两个中间字段值的平均值之外。

例子

例一：计算指定字段的中位数

```
> SELECT MEDIAN("water_level") FROM "h2o_feet"

name: h2o_feet
time                   median
----                   ------
1970-01-01T00:00:00Z   4.124
```

该查询返回measurement `h2o_feet` 的字段 `water_level` 的中位数。

例二：计算measurement中每个字段的中位数

```
> SELECT MEDIAN(*) FROM "h2o_feet"
```

```
2.
3.  name: h2o_feet
4.  time                   median_water_level
5.  ----                   ------------------
6.  1970-01-01T00:00:00Z   4.124
```

查询返回在 `h2o_feet` 中数值类型的每个字段的中位数。 `h2o_feet` 有一个数值字段： `water_level` 。

例三：计算满足正则表达式的字段的中位数

```
1.  > SELECT MEDIAN(/water/) FROM "h2o_feet"
2.
3.  name: h2o_feet
4.  time                   median_water_level
5.  ----                   ------------------
6.  1970-01-01T00:00:00Z   4.124
```

查询返回在 `h2o_feet` 中字段中含有 `water` 的数值类型字段的中位数。

例四：计算含有多个子句字段的中位数

```
    > SELECT MEDIAN("water_level") FROM "h2o_feet" WHERE time >= '2015-08-
    17T23:48:00Z' AND time <= '2015-08-18T00:54:00Z' GROUP BY time(12m),* fill(700)
1.  LIMIT 7 SLIMIT 1 SOFFSET 1
2.
3.  name: h2o_feet
4.  tags: location=santa_monica
5.  time                   median
6.  ----                   ------
7.  2015-08-17T23:48:00Z   700
8.  2015-08-18T00:00:00Z   2.09
9.  2015-08-18T00:12:00Z   2.077
10. 2015-08-18T00:24:00Z   2.0460000000000003
11. 2015-08-18T00:36:00Z   2.0620000000000003
12. 2015-08-18T00:48:00Z   700
```

查询返回字段 `water_level` 中的值的中位数。它涵盖 `2015-08-17T23：48：00Z` 和 `2015-08-18T00：54：00Z` 之间的时间段，并将结果按12分钟的时间间隔和每个tag分组。该查询用 `700` 填充空时间间隔，并将点数和series分别限制到7和1，并将series的返回偏移1。

MODE()

返回字段中出现频率最高的值。

语法

```
SELECT MODE( [ * | <field_key> | /<regular_expression>/ ] ) [INTO_clause]
FROM_clause [WHERE_clause] [GROUP_BY_clause] [ORDER_BY_clause] [LIMIT_clause]
[OFFSET_clause] [SLIMIT_clause] [SOFFSET_clause]
```

语法描述

`MODE(field_key)`

返回field key关联的值的出现频率最高的值。

`MODE(/regular_expression/)`

返回满足正则表达式的每个field key关联的值的出现频率最高的值。

`MODE(*)`

返回measurement中每个field key关联的值的出现频率最高的值。

`MODE()` 支持所有数据类型。

> 注意: `MODE()` 如果最多出现次数有两个或多个值，则返回具有最早时间戳的字段值。

例子

例一：计算指定字段的最常出现的值

```
> SELECT MODE("level description") FROM "h2o_feet"

name: h2o_feet
time                   mode
----                   ----
1970-01-01T00:00:00Z   between 3 and 6 feet
```

该查询返回measurement `h2o_feet` 的字段 `level description` 的最常出现的值。

例二：计算measurement中每个字段最常出现的值

```
> SELECT MODE(*) FROM "h2o_feet"

name: h2o_feet
time                   mode_level description   mode_water_level
```

```
5. ----                   ----------------------   ---------------
6. 1970-01-01T00:00:00Z   between 3 and 6 feet     2.69
```

查询返回在 `h2o_feet` 中数值类型的每个字段的最常出现的值。 `h2o_feet` 有两个字段： `water_level` 和 `level description` 。

例三：计算满足正则表达式的字段的最常出现的值

```
1. > SELECT MODE(/water/) FROM "h2o_feet"
2.
3. name: h2o_feet
4. time                   mode_water_level
5. ----                   ----------------
6. 1970-01-01T00:00:00Z   2.69
```

查询返回在 `h2o_feet` 中字段中含有 `water` 的字段的最常出现的值。

例四：计算含有多个子句字段的最常出现的值

```
    > SELECT MODE("level description") FROM "h2o_feet" WHERE time >= '2015-08-
    17T23:48:00Z' AND time <= '2015-08-18T00:54:00Z' GROUP BY time(12m),* LIMIT 3
1. SLIMIT 1 SOFFSET 1
2.
3. name: h2o_feet
4. tags: location=santa_monica
5. time                   mode
6. ----                   ----
7. 2015-08-17T23:48:00Z
8. 2015-08-18T00:00:00Z   below 3 feet
9. 2015-08-18T00:12:00Z   below 3 feet
```

查询返回字段 `water_level` 中的值的最常出现的值。它涵盖 `2015-08-17T23：48：00Z` 和 `2015-08-18T00：54：00Z` 之间的时间段，并将结果按12分钟的时间间隔和每个tag分组。，并将点数和series分别限制到3和1，并将series的返回偏移1。

SPREAD( )

返回字段中最大和最小值的差值。

语法

```
SELECT SPREAD( [ * | <field_key> | /<regular_expression>/ ] ) [INTO_clause]
FROM_clause [WHERE_clause] [GROUP_BY_clause] [ORDER_BY_clause] [LIMIT_clause]
[OFFSET_clause] [SLIMIT_clause] [SOFFSET_clause]
```

语法描述

`SPREAD(field_key)`

返回field key最大和最小值的差值。

`SPREAD(/regular_expression/)`

返回满足正则表达式的每个field key最大和最小值的差值。

`SPREAD(*)`

返回measurement中每个field key最大和最小值的差值。

`SPREAD()` 支持所有的数值类型的field。

例子

例一：计算指定字段最大和最小值的差值

```
> SELECT SPREAD("water_level") FROM "h2o_feet"

name: h2o_feet
time                   spread
----                   ------
1970-01-01T00:00:00Z   10.574
```

该查询返回measurement `h2o_feet` 的字段 `water_level` 的最大和最小值的差值。

例二：计算measurement中每个字段最大和最小值的差值

```
> SELECT SPREAD(*) FROM "h2o_feet"

name: h2o_feet
time                   spread_water_level
----                   ------------------
1970-01-01T00:00:00Z   10.574
```

查询返回在 `h2o_feet` 中数值类型的每个数值字段的最大和最小值的差值。 `h2o_feet` 有一个数值字段： `water_level` 。

例三：计算满足正则表达式的字段最大和最小值的差值

```
> SELECT SPREAD(/water/) FROM "h2o_feet"

name: h2o_feet
time                   spread_water_level
----                   ------------------
1970-01-01T00:00:00Z   10.574
```

查询返回在 `h2o_feet` 中字段中含有 `water` 的所有数值字段的最大和最小值的差值。

例四：计算含有多个子句字段最大和最小值的差值

```
> SELECT SPREAD("water_level") FROM "h2o_feet" WHERE time >= '2015-08-17T23:48:00Z' AND time <= '2015-08-18T00:54:00Z' GROUP BY time(12m),* fill(18) LIMIT 3 SLIMIT 1 SOFFSET 1

name: h2o_feet
tags: location=santa_monica
time                   spread
----                   ------
2015-08-17T23:48:00Z   18
2015-08-18T00:00:00Z   0.052000000000000046
2015-08-18T00:12:00Z   0.09799999999999986
```

查询返回字段 `water_level` 中的最大和最小值的差值。它涵盖 `2015-08-17T23：48：00Z` 和 `2015-08-18T00：54：00Z` 之间的时间段，并将结果按12分钟的时间间隔和每个tag分组，空值用18来填充，并将点数和series分别限制到3和1，并将series的返回偏移1。

STDDEV( )

返回字段的标准差。

语法

```
SELECT STDDEV( [ * | <field_key> | /<regular_expression>/ ] ) [INTO_clause] FROM_clause [WHERE_clause] [GROUP_BY_clause] [ORDER_BY_clause] [LIMIT_clause] [OFFSET_clause] [SLIMIT_clause] [SOFFSET_clause]
```

语法描述

`STDDEV(field_key)`

返回field key的标准差。

`STDDEV(/regular_expression/)`

返回满足正则表达式的每个field key的标准差。

`STDDEV(*)`

返回measurement中每个field key的标准差。

`STDDEV()` 支持所有的数值类型的field。

例子

例一：计算指定字段的标准差

```
> SELECT STDDEV("water_level") FROM "h2o_feet"

name: h2o_feet
time                   stddev
----                   ------
1970-01-01T00:00:00Z   2.279144584196141
```

该查询返回measurement `h2o_feet` 的字段 `water_level` 的标准差。

例二：计算measurement中每个字段的标准差

```
> SELECT STDDEV(*) FROM "h2o_feet"

name: h2o_feet
time                   stddev_water_level
----                   ------------------
1970-01-01T00:00:00Z   2.279144584196141
```

查询返回在 `h2o_feet` 中数值类型的每个数值字段的标准差。 `h2o_feet` 有一个数值字段： `water_level` 。

例三：计算满足正则表达式的字段的标准差

```
> SELECT STDDEV(/water/) FROM "h2o_feet"

name: h2o_feet
time                   stddev_water_level
----                   ------------------
1970-01-01T00:00:00Z   2.279144584196141
```

查询返回在 `h2o_feet` 中字段中含有 `water` 的所有数值字段的标准差。

例四：计算含有多个子句字段的标准差

```
> SELECT STDDEV("water_level") FROM "h2o_feet" WHERE time >= '2015-08-17T23:48:00Z' AND time <= '2015-08-18T00:54:00Z' GROUP BY time(12m),*fill(18000) LIMIT 2 SLIMIT 1 SOFFSET 1

name: h2o_feet
tags: location=santa_monica
time                   stddev
----                   ------
2015-08-17T23:48:00Z   18000
2015-08-18T00:00:00Z   0.03676955262170051
```

查询返回字段 `water_level` 的标准差。它涵盖 `2015-08-17T23：48：00Z` 和 `2015-08-18T00：54：00Z` 之间的时间段，并将结果按12分钟的时间间隔和每个tag分组，空值用18000来填充，并将点数和series分别限制到2和1，并将series的返回偏移1。

SUM( )

返回字段值的和。

语法

```
SELECT SUM( [ * | <field_key> | /<regular_expression>/ ] ) [INTO_clause] FROM_clause [WHERE_clause] [GROUP_BY_clause] [ORDER_BY_clause] [LIMIT_clause] [OFFSET_clause] [SLIMIT_clause] [SOFFSET_clause]
```

语法描述

`SUM(field_key)`

返回field key的值的和。

`SUM(/regular_expression/)`

返回满足正则表达式的每个field key的值的和。

`SUM(*)`

返回measurement中每个field key的值的和。

`SUM()` 支持所有的数值类型的field。

例子

例一：计算指定字段的值的和

```
> SELECT SUM("water_level") FROM "h2o_feet"

name: h2o_feet
time                   sum
----                   ---
1970-01-01T00:00:00Z   67777.66900000004
```

该查询返回measurement `h2o_feet` 的字段 `water_level` 的值的和。

例二：计算measurement中每个字段的值的和

```
> SELECT SUM(*) FROM "h2o_feet"

name: h2o_feet
time                   sum_water_level
----                   ---------------
1970-01-01T00:00:00Z   67777.66900000004
```

查询返回在 `h2o_feet` 中数值类型的每个数值字段的值的和。 `h2o_feet` 有一个数值字段： `water_level` 。

例三：计算满足正则表达式的字段的值的和

```
> SELECT SUM(/water/) FROM "h2o_feet"

name: h2o_feet
time                   sum_water_level
----                   ---------------
1970-01-01T00:00:00Z   67777.66900000004
```

查询返回在 `h2o_feet` 中字段中含有 `water` 的所有数值字段的值的和。

例四：计算含有多个子句字段的值的和

```
    > SELECT SUM("water_level") FROM "h2o_feet" WHERE time >= '2015-08-
    17T23:48:00Z' AND time <= '2015-08-18T00:54:00Z' GROUP BY time(12m),*
1.  fill(18000) LIMIT 4 SLIMIT 1
2.
3.  name: h2o_feet
```

```
tags: location=coyote_creek
time                    sum
----                    ---
2015-08-17T23:48:00Z    18000
2015-08-18T00:00:00Z    16.125
2015-08-18T00:12:00Z    15.649
2015-08-18T00:24:00Z    15.135
```

查询返回字段 `water_level` 的值的和。它涵盖 `2015-08-17T23：48：00Z` 和 `2015-08-18T00：54：00Z` 之间的时间段，并将结果按12分钟的时间间隔和每个tag分组，空值用18000来填充，并将点数和series分别限制到2和1，并将series的返回偏移1。

Selectors

BOTTOM()

返回最小的N个field值。

语法

```
SELECT BOTTOM(<field_key>[,<tag_key(s)>],<N> )[,<tag_key(s)>|<field_key(s)>]
[INTO_clause] FROM_clause [WHERE_clause] [GROUP_BY_clause] [ORDER_BY_clause]
[LIMIT_clause] [OFFSET_clause] [SLIMIT_clause] [SOFFSET_clause]
```

语法描述

`BOTTOM(field_key,N)`

返回field key的最小的N个field value。

`BOTTOM(field_key,tag_key(s),N)`

返回某个tag key的N个tag value的最小的field value。

`BOTTOM(field_key,N),tag_key(s),field_key(s)`

返回括号里的字段的最小N个field value，以及相关的tag或field，或者两者都有。

`BOTTOM()` 支持所有的数值类型的field。

> 说明：
>
> - 如果一个field有两个或多个相等的field value， `BOTTOM()` 返回时间戳最早的那个。
> - `BOTTOM()` 和 `INTO` 子句一起使用的时候，和其他的函数有些不一样。

例子

例一：选择一个field的最小的三个值

```
> SELECT BOTTOM("water_level",3) FROM "h2o_feet"

name: h2o_feet
time                   bottom
----                   ------
2015-08-29T14:30:00Z   -0.61
2015-08-29T14:36:00Z   -0.591
2015-08-30T15:18:00Z   -0.594
```

该查询返回measurement `h2o_feet` 的字段 `water_level` 的最小的三个值。

例二：选择一个field的两个tag的分别最小的值

```
> SELECT BOTTOM("water_level","location",2) FROM "h2o_feet"

name: h2o_feet
time                   bottom   location
----                   ------   --------
2015-08-29T10:36:00Z   -0.243   santa_monica
2015-08-29T14:30:00Z   -0.61    coyote_creek
```

该查询返回和tag `location` 相关的两个tag值的字段 `water_level` 的分别最小值。

例三：选择一个field的最小的四个值，以及其关联的tag和field

```
> SELECT BOTTOM("water_level",4),"location","level description" FROM "h2o_feet"

name: h2o_feet
time                   bottom  location      level description
----                   ------  --------      -----------------
2015-08-29T14:24:00Z   -0.587  coyote_creek  below 3 feet
2015-08-29T14:30:00Z   -0.61   coyote_creek  below 3 feet
2015-08-29T14:36:00Z   -0.591  coyote_creek  below 3 feet
2015-08-30T15:18:00Z   -0.594  coyote_creek  below 3 feet
```

查询返回 `water_level` 中最小的四个字段值以及tag `location` 和field `level description` 的相关值。

例四：选择一个field的最小的三个值，并且包括了多个子句

```
> SELECT BOTTOM("water_level",3),"location" FROM "h2o_feet" WHERE time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:54:00Z' GROUP BY time(24m) ORDER BY time DESC

name: h2o_feet
time                  bottom  location
----                  ------  --------
2015-08-18T00:48:00Z  1.991   santa_monica
2015-08-18T00:54:00Z  2.054   santa_monica
2015-08-18T00:54:00Z  6.982   coyote_creek
2015-08-18T00:24:00Z  2.041   santa_monica
2015-08-18T00:30:00Z  2.051   santa_monica
2015-08-18T00:42:00Z  2.057   santa_monica
2015-08-18T00:00:00Z  2.064   santa_monica
2015-08-18T00:06:00Z  2.116   santa_monica
2015-08-18T00:12:00Z  2.028   santa_monica
```

查询将返回在 `2015-08-18T00：00：00Z` 和 `2015-08-18T00：54：00Z` 之间的每24分钟间隔内， `water_level` 最小的三个值。它还以降序的时间戳顺序返回结果。

请注意， `GROUP BY time()` 子句不会覆盖点的原始时间戳。有关该行为的更详细解释，请参阅下面的问题一。

`BOTTOM()` 的常见问题

问题一： `BOTTOM()` 和 `GROUP BY time()` 子句

`BOTTOM()` 和 `GROUP BY time()` 子句的查询返回每个 `GROUP BY time()` 间隔指定的点数。对于大多数 `GROUP BY time()` 查询，返回的时间戳记标记 `GROUP BY time()` 间隔的开始。 `GROUP BY time()` 查询与 `BOTTOM()` 函数的行为不同； 它们保留原始数据点的时间戳。

例如

下面的查询返回每18分钟· `GROUP BY time()` 间隔的两点。请注意，返回的时间戳是点的原始时间戳； 它们不会被强制匹配 `GROUP BY time()` 间隔的开始。

```
> SELECT BOTTOM("water_level",2) FROM "h2o_feet" WHERE time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z' AND "location" = 'santa_monica' GROUP BY time(18m)

name: h2o_feet
time                  bottom
----                  ------
                        __
```

```
2015-08-18T00:00:00Z  2.064 |
2015-08-18T00:12:00Z  2.028 | <------- Smallest points for the first time
interval
                          --
                          __
2015-08-18T00:24:00Z  2.041 |
2015-08-18T00:30:00Z  2.051 | <------- Smallest points for the second time
interval
                          --
```

问题二：`BOTTOM()`和一个少于N个值得tag key

使用语法 `SELECT BOTTOM (<field_key>, <tag_key>, <N>)` 的查询可以返回比预期少的点。如果tag具有X标签值，则查询指定N个值，当X小于N，则查询返回X点。

例如

下面的查询将要求tag `location` 的三个值的 `water_level` 的最小字段值。由于 `location` 具有两个值（ `santa_monica` 和 `coyote_creek` ），所以查询返回两点而不是三个。

```
> SELECT BOTTOM("water_level","location",3) FROM "h2o_feet"

name: h2o_feet
time                   bottom   location
----                   ------   --------
2015-08-29T10:36:00Z   -0.243   santa_monica
2015-08-29T14:30:00Z   -0.61    coyote_creek
```

问题三：`BOTTOM()`，tags和 `INTO` 子句

当与 `INTO` 子句和 `GROUP BY tag` 子句结合使用时，大多数InfluxQL函数将初始数据中的任何tag转换为新写入的数据中的field。此行为也适用于 `BOTTOM()` 函数，除非 `BOTTOM()` 包含一个tag key作为参数： `BOTTOM(field_key, tag_key(s), N)` 。在这些情况下，系统将指定的tag作为新写入的数据中的tag。

例如

下面的代码块中的第一个查询返回与tag `location` 相关联的两个tag value的field `water_level` 中最小的字段值。它也将这些结果写入measurement `bottom_water_levels` 。 第二个查询显示InfluxDB在 `bottom_water_levels` 中将 `location` 保存为tag。

```
> SELECT BOTTOM("water_level","location",2) INTO "bottom_water_levels" FROM
"h2o_feet"
```

```
2.
3.  name: result
4.  time                  written
5.  ----                  -------
6.  1970-01-01T00:00:00Z 2
7.
8.  > SHOW TAG KEYS FROM "bottom_water_levels"
9.
10. name: bottom_water_levels
11. tagKey
12. ------
13. location
```

FIRST()

返回时间戳最早的值

语法

```
    SELECT FIRST(<field_key>)[,<tag_key(s)>|<field_key(s)>] [INTO_clause]
    FROM_clause [WHERE_clause] [GROUP_BY_clause] [ORDER_BY_clause] [LIMIT_clause]
1.  [OFFSET_clause] [SLIMIT_clause] [SOFFSET_clause]
```

语法描述

`FIRST(field_key)`

返回field key时间戳最早的值。

`FIRST(/regular_expression/)`

返回满足正则表达式的每个field key的时间戳最早的值。

`FIRST(*)`

返回measurement中每个field key的时间戳最早的值。

`FIRST(field_key),tag_key(s),field_key(s)`

返回括号里的字段的时间戳最早的值，以及相关联的tag或field，或者两者都有。

`FIRST()` 支持所有类型的field。

例子

例一：返回field key时间戳最早的值

```
1.  > SELECT FIRST("level description") FROM "h2o_feet"
2.
3.  name: h2o_feet
4.  time                   first
5.  ----                   -----
6.  2015-08-18T00:00:00Z   between 6 and 9 feet
```

查询返回 `level description` 的时间戳最早的值。

例二：列出一个measurement中每个field key的时间戳最早的值

```
1.  > SELECT FIRST(*) FROM "h2o_feet"
2.
3.  name: h2o_feet
4.  time                   first_level description   first_water_level
```

```
5. ----                    ----------------------    ----------------
6. 1970-01-01T00:00:00Z    between 6 and 9 feet      8.12
```

查询返回 `h2o_feet` 中每个字段的时间戳最早的值。 `h2o_feet` 有两个字段： `level description` 和 `water_level` 。

例三：列出匹配正则表达式的field的时间戳最早的值

```
1. > SELECT FIRST(/level/) FROM "h2o_feet"
2.
3. name: h2o_feet
4. time                    first_level description   first_water_level
5. ----                    ----------------------    ----------------
6. 1970-01-01T00:00:00Z    between 6 and 9 feet      8.12
```

查询返回 `h2o_feet` 中含有 `level` 的字段的时间戳最早的值。

例四：返回field的最早的值，以及其相关的tag和field

```
1. > SELECT FIRST("level description"),"location","water_level" FROM "h2o_feet"
2.
3. name: h2o_feet
4. time                    first                  location      water_level
5. ----                    -----                  --------      -----------
6. 2015-08-18T00:00:00Z    between 6 and 9 feet   coyote_creek  8.12
```

查询返回 `level description` 的时间戳最早的值，以及其相关的tag `location` 和field `water_level` 。

例五：列出包含多个子句的field key的时间戳最早的值

```
    > SELECT FIRST("water_level") FROM "h2o_feet" WHERE time >= '2015-08-
    17T23:48:00Z' AND time <= '2015-08-18T00:54:00Z' GROUP BY time(12m),*
1.  fill(9.01) LIMIT 4 SLIMIT 1
2.
3.  name: h2o_feet
4.  tags: location=coyote_creek
5.  time                    first
6.  ----                    -----
7.  2015-08-17T23:48:00Z    9.01
8.  2015-08-18T00:00:00Z    8.12
9.  2015-08-18T00:12:00Z    7.887
10. 2015-08-18T00:24:00Z    7.635
```

查询返回字段 `water_level` 中最早的字段值。它涵盖 `2015-08-17T23：48：00Z` 和 `2015-08-18T00：54：00Z` 之间的时间段，并将结果按12分钟的时间间隔和每个tag分组。查询用 `9.01` 填充空时间间隔，并将点数和measurement限制到4和1。

请注意， `GROUP BY time()` 子句覆盖点的原始时间戳。结果中的时间戳表示每12分钟时间间隔的开始；结果的第一点涵盖 `2015-08-17T23：48：00Z` 和 `2015-08-18T00：00：00Z` 之间的时间间隔，结果的最后一点涵盖 `2015-08-18T00:24:00Z` 和 `2015-08-18T00：36：00Z` 之间的间隔。

LAST( )

返回时间戳最近的值

语法

```
    SELECT LAST(<field_key>)[,<tag_key(s)>|<field_keys(s)>] [INTO_clause]
    FROM_clause [WHERE_clause] [GROUP_BY_clause] [ORDER_BY_clause] [LIMIT_clause]
1.  [OFFSET_clause] [SLIMIT_clause] [SOFFSET_clause]
```

语法描述

`LAST(field_key)`

返回field key时间戳最近的值。

`LAST(/regular_expression/)`

返回满足正则表达式的每个field key的时间戳最近的值。

`LAST(*)`

返回measurement中每个field key的时间戳最近的值。

`LAST(field_key),tag_key(s),field_key(s)`

返回括号里的字段的时间戳最近的值，以及相关联的tag或field，或者两者都有。

`LAST()` 支持所有类型的field。

例子

例一：返回field key时间戳最近的值

```
1.  > SELECT LAST("level description") FROM "h2o_feet"
2.
3.  name: h2o_feet
4.  time                   last
```

```
5. ----                    ----
6. 2015-09-18T21:42:00Z    between 3 and 6 feet
```

查询返回 `level description` 的时间戳最近的值。

例二：列出一个measurement中每个field key的时间戳最近的值

```
1. > SELECT LAST(*) FROM "h2o_feet"
2.
3. name: h2o_feet
4. time                   first_level description   first_water_level
5. ----                   -----------------------   -----------------
6. 1970-01-01T00:00:00Z   between 3 and 6 feet      4.938
```

查询返回 `h2o_feet` 中每个字段的时间戳最近的值。 `h2o_feet` 有两个字段： `level description` 和 `water_level` 。

例三：列出匹配正则表达式的field的时间戳最近的值

```
1. > SELECT LAST(/level/) FROM "h2o_feet"
2.
3. name: h2o_feet
4. time                   first_level description   first_water_level
5. ----                   -----------------------   -----------------
6. 1970-01-01T00:00:00Z   between 3 and 6 feet      4.938
```

查询返回 `h2o_feet` 中含有 `level` 的字段的时间戳最近的值。

例四：返回field的最近的值，以及其相关的tag和field

```
1. > SELECT LAST("level description"),"location","water_level" FROM "h2o_feet"
2.
3. name: h2o_feet
4. time                   last                    location      water_level
5. ----                   ----                    --------      -----------
6. 2015-09-18T21:42:00Z   between 3 and 6 feet    santa_monica  4.938
```

查询返回 `level description` 的时间戳最近的值，以及其相关的tag `location` 和field `water_level` 。

例五：列出包含多个子句的field key的时间戳最近的值

```
> SELECT LAST("water_level") FROM "h2o_feet" WHERE time >= '2015-08-17T23:48:00Z' AND time <= '2015-08-18T00:54:00Z' GROUP BY time(12m),*fill(9.01) LIMIT 4 SLIMIT 1

name: h2o_feet
tags: location=coyote_creek
time                   last
----                   ----
2015-08-17T23:48:00Z   9.01
2015-08-18T00:00:00Z   8.005
2015-08-18T00:12:00Z   7.762
2015-08-18T00:24:00Z   7.5
```

查询返回字段 `water_level` 中最近的字段值。它涵盖 `2015-08-17T23：48：00Z` 和 `2015-08-18T00：54：00Z` 之间的时间段，并将结果按12分钟的时间间隔和每个tag分组。查询用 `9.01` 填充空时间间隔，并将点数和measurement限制到4和1。

请注意， `GROUP BY time()` 子句覆盖点的原始时间戳。结果中的时间戳表示每12分钟时间间隔的开始；结果的第一点涵盖 `2015-08-17T23：48：00Z` 和 `2015-08-18T00：00：00Z` 之间的时间间隔，结果的最后一点涵盖 `2015-08-18T00:24:00Z` 和 `2015-08-18T00：36：00Z` 之间的间隔。

MAX( )

返回最大的字段值

语法

```
SELECT MAX(<field_key>)[,<tag_key(s)>|<field__key(s)>] [INTO_clause] FROM_clause [WHERE_clause] [GROUP_BY_clause] [ORDER_BY_clause] [LIMIT_clause] [OFFSET_clause] [SLIMIT_clause] [SOFFSET_clause]
```

语法描述

`MAX(field_key)`

返回field key的最大值。

`MAX(/regular_expression/)`

返回满足正则表达式的每个field key的最大值。

`MAX(*)`

返回measurement中每个field key的最大值。

`MAX(field_key),tag_key(s),field_key(s)`

返回括号里的字段的最大值，以及相关联的tag或field，或者两者都有。

`MAX()` 支持所有数值类型的field。

例子

例一：返回field key的最大值

```
1. > SELECT MAX("water_level") FROM "h2o_feet"
2.
3. name: h2o_feet
4. time                   max
5. ----                   ---
6. 2015-08-29T07:24:00Z   9.964
```

查询返回 `water_level` 的最大值。

例二：列出一个measurement中每个field key的最大值

```
1. > SELECT MAX(*) FROM "h2o_feet"
2.
3. name: h2o_feet
4. time                   max_water_level
5. ----                   ---------------
6. 2015-08-29T07:24:00Z   9.964
```

查询返回 `h2o_feet` 中每个字段的最大值。 `h2o_feet` 有一个数值类型的字段： `water_level` 。

例三：列出匹配正则表达式的field的最大值

```
1. > SELECT MAX(/level/) FROM "h2o_feet"
2.
3. name: h2o_feet
4. time                   max_water_level
5. ----                   ---------------
6. 2015-08-29T07:24:00Z   9.964
```

查询返回 `h2o_feet` 中含有 `level` 的数值字段的最大值。

例四：返回field的最大值，以及其相关的tag和field

```
> SELECT MAX("water_level"),"location","level description" FROM "h2o_feet"

name: h2o_feet
time                   max    location      level description
----                   ---    --------      -----------------
2015-08-29T07:24:00Z   9.964  coyote_creek  at or greater than 9 feet
```

查询返回 `water_level` 的最大值，以及其相关的tag `location` 和field `level description` 。

例五：列出包含多个子句的field key的最大值

```
> SELECT MAX("water_level") FROM "h2o_feet" WHERE time >= '2015-08-17T23:48:00Z' AND time <= '2015-08-18T00:54:00Z' GROUP BY time(12m),*fill(9.01) LIMIT 4 SLIMIT 1

name: h2o_feet
tags: location=coyote_creek
time                   max
----                   ---
2015-08-17T23:48:00Z   9.01
2015-08-18T00:00:00Z   8.12
2015-08-18T00:12:00Z   7.887
2015-08-18T00:24:00Z   7.635
```

查询返回字段 `water_level` 的最大值。它涵盖 `2015-08-17T23：48：00Z` 和 `2015-08-18T00：54：00Z` 之间的时间段，并将结果按12分钟的时间间隔和每个tag分组。查询用 `9.01` 填充空时间间隔，并将点数和measurement限制到4和1。

请注意， `GROUP BY time()` 子句覆盖点的原始时间戳。结果中的时间戳表示每12分钟时间间隔的开始；结果的第一点涵盖 `2015-08-17T23：48：00Z` 和 `2015-08-18T00：00：00Z` 之间的时间间隔，结果的最后一点涵盖 `2015-08-18T00:24:00Z` 和 `2015-08-18T00：36：00Z` 之间的间隔。

MIN()

返回最小的字段值

语法

```
SELECT MIN(<field_key>)[,<tag_key(s)>|<field__key(s)>] [INTO_clause] FROM_clause [WHERE_clause] [GROUP_BY_clause] [ORDER_BY_clause] [LIMIT_clause] [OFFSET_clause] [SLIMIT_clause] [SOFFSET_clause]
```

语法描述

`MIN(field_key)`

返回field key的最小值。

`MIN(/regular_expression/)`

返回满足正则表达式的每个field key的最小值。

`MIN(*)`

返回measurement中每个field key的最小值。

`MIN(field_key),tag_key(s),field_key(s)`

返回括号里的字段的最小值，以及相关联的tag或field，或者两者都有。

`MIN()` 支持所有数值类型的field。

例子

例一：返回field key的最小值

```
1. > SELECT MIN("water_level") FROM "h2o_feet"
2.
3. name: h2o_feet
4. time                   min
5. ----                   ---
6. 2015-08-29T14:30:00Z   -0.61
```

查询返回 `water_level` 的最小值。

例二：列出一个measurement中每个field key的最小值

```
1. > SELECT MIN(*) FROM "h2o_feet"
2.
3. name: h2o_feet
4. time                   min_water_level
5. ----                   ---------------
6. 2015-08-29T14:30:00Z   -0.61
```

查询返回 `h2o_feet` 中每个字段的最小值。 `h2o_feet` 有一个数值类型的字段： `water_level` 。

例三：列出匹配正则表达式的field的最小值

```
> SELECT MIN(/level/) FROM "h2o_feet"

name: h2o_feet
time                   min_water_level
----                   ---------------
2015-08-29T14:30:00Z   -0.61
```

查询返回 `h2o_feet` 中含有 `level` 的数值字段的最小值。

例四：返回field的最小值，以及其相关的tag和field

```
> SELECT MIN("water_level"),"location","level description" FROM "h2o_feet"

name: h2o_feet
time                   min    location      level description
----                   ---    --------      -----------------
2015-08-29T14:30:00Z   -0.61  coyote_creek  below 3 feet
```

查询返回 `water_level` 的最小值，以及其相关的tag `location` 和field `level description` 。

例五：列出包含多个子句的field key的最小值

```
> SELECT MIN("water_level") FROM "h2o_feet" WHERE time >= '2015-08-17T23:48:00Z' AND time <= '2015-08-18T00:54:00Z' GROUP BY time(12m),*fill(9.01) LIMIT 4 SLIMIT 1

name: h2o_feet
tags: location=coyote_creek
time                   min
----                   ---
2015-08-17T23:48:00Z   9.01
2015-08-18T00:00:00Z   8.005
2015-08-18T00:12:00Z   7.762
2015-08-18T00:24:00Z   7.5
```

查询返回字段 `water_level` 的最小值。它涵盖 `2015-08-17T23：48：00Z` 和 `2015-08-18T00：54：00Z` 之间的时间段，并将结果按12分钟的时间间隔和每个tag分组。查询用 `9.01` 填充空时间间隔，并将点数和measurement限制到4和1。

请注意， `GROUP BY time()` 子句覆盖点的原始时间戳。结果中的时间戳表示每12分钟时间间隔的开始；结果的第一点涵盖 `2015-08-17T23：48：00Z` 和 `2015-08-18T00：00：00Z` 之间的时间间隔，结果的最后一点涵盖 `2015-08-18T00:24:00Z` 和 `2015-08-18T00：36：00Z` 之间的间隔。

PERCENTILE()

返回较大百分之N的字段值

语法

```
SELECT PERCENTILE(<field_key>, <N>)[,<tag_key(s)>|<field_key(s)>] [INTO_clause]
FROM_clause [WHERE_clause] [GROUP_BY_clause] [ORDER_BY_clause] [LIMIT_clause]
[OFFSET_clause] [SLIMIT_clause] [SOFFSET_clause]
```

语法描述

`PERCENTILE(field_key,N)`

返回field key较大的百分之N的值。

`PERCENTILE(/regular_expression/,N)`

返回满足正则表达式的每个field key较大的百分之N的值。

`PERCENTILE(*,N)`

返回measurement中每个field key较大的百分之N的值。

`PERCENTILE(field_key,N),tag_key(s),field_key(s)`

返回括号里的字段较大的百分之N的值，以及相关联的tag或field，或者两者都有。

`N` 必须是0到100的整数或者浮点数。

`PERCENTILE()` 支持所有数值类型的field。

例子

例一：返回field key较大的百分之5的值

```
> SELECT PERCENTILE("water_level",5) FROM "h2o_feet"

name: h2o_feet
time                   percentile
----                   ----------
2015-08-31T03:42:00Z   1.122
```

查询返回 `water_level` 中值在总的field value中比较大的百分之五。

例二：列出一个measurement中每个field key较大的百分之5的值

```
> SELECT PERCENTILE(*,5) FROM "h2o_feet"

name: h2o_feet
time                   percentile_water_level
----                   ----------------------
2015-08-31T03:42:00Z   1.122
```

查询返回 `h2o_feet` 中每个字段中值在总的field value中比较大的百分之五。 `h2o_feet` 有一个数值类型的字段： `water_level` 。

例三：列出匹配正则表达式的field较大的百分之5的值

```
> SELECT PERCENTILE(/level/,5) FROM "h2o_feet"

name: h2o_feet
time                   percentile_water_level
----                   ----------------------
2015-08-31T03:42:00Z   1.122
```

查询返回 `h2o_feet` 中含有 `water` 的数值字段的较大的百分之5的值。

例四：返回field较大的百分之5的值，以及其相关的tag和field

```
> SELECT PERCENTILE("water_level",5),"location","level description" FROM "h2o_feet"

name: h2o_feet
time                   percentile  location      level description
----                   ----------  --------      -----------------
2015-08-31T03:42:00Z   1.122       coyote_creek  below 3 feet
```

查询返回 `water_level` 的较大的百分之5的值，以及其相关的tag `location` 和field `level description` 。

例五：列出包含多个子句的field key的较大的百分之20的值

```
> SELECT PERCENTILE("water_level",20) FROM "h2o_feet" WHERE time >= '2015-08-17T23:48:00Z' AND time <= '2015-08-18T00:54:00Z' GROUP BY time(24m) fill(15) LIMIT 2

name: h2o_feet
time                   percentile
----                   ----------
```

```
6.  2015-08-17T23:36:00Z   15
7.  2015-08-18T00:00:00Z   2.064
```

查询返回字段 `water_level` 较大的百分之20的值。它涵盖 `2015-08-17T23：48：00Z` 和 `2015-08-18T00：54：00Z` 之间的时间段，并将结果按24分钟的时间间隔分组。查询用 `15` 填充空时间间隔，并将点数限制到2。

请注意， `GROUP BY time()` 子句覆盖点的原始时间戳。结果中的时间戳表示每24分钟时间间隔的开始；结果的第一点涵盖 `2015-08-17T23：36：00Z` 和 `2015-08-18T00：00：00Z` 之间的时间间隔，结果的最后一点涵盖 `2015-08-18T00:00:00Z` 和 `2015-08-18T00：24：00Z` 之间的间隔。

PERCENTILE()的常见问题

问题一：PERCENTILE()和其他函数的比较

- `PERCENTILE(<field_key>,100)` 相当于 `MAX(<field_key>)` 。
- `PERCENTILE(<field_key>, 50)` 几乎等于 `MEDIAN(<field_key>)` ，除了如果字段键包含偶数个字段值，`MEDIAN()` 函数返回两个中间值的平均值.
- `PERCENTILE(<field_key>,0)` 相当于 `MIN(<field_key>)`

SAMPLE()

返回 `N` 个随机抽样的字段值。 `SAMPLE()` 使用reservoir sampling来生成随机点。

语法

```
    SELECT SAMPLE(<field_key>, <N>)[,<tag_key(s)>|<field_key(s)>] [INTO_clause]
    FROM_clause [WHERE_clause] [GROUP_BY_clause] [ORDER_BY_clause] [LIMIT_clause]
1.  [OFFSET_clause] [SLIMIT_clause] [SOFFSET_clause]
```

`SAMPLE(field_key,N)`

返回field key的N个随机抽样的字段值。

`SAMPLE(/regular_expression/,N)`

返回满足正则表达式的每个field key的N个随机抽样的字段值。

`SAMPLE(*,N)`

返回measurement中每个field key的N个随机抽样的字段值。

`SAMPLE(field_key,N),tag_key(s),field_key(s)`

返回括号里的字段的N个随机抽样的字段值，以及相关联的tag或field，或者两者都有。

`N` 必须是整数。

`SAMPLE()` 支持所有类型的field。

例子

例一：返回field key的两个随机抽样的字段值

```
> SELECT SAMPLE("water_level",2) FROM "h2o_feet"

name: h2o_feet
time                   sample
----                   ------
2015-09-09T21:48:00Z   5.659
2015-09-18T10:00:00Z   6.939
```

查询返回 `water_level` 的两个随机抽样的字段值。

例二：列出一个measurement中每个field key的两个随机抽样的字段值

```
> SELECT SAMPLE(*,2) FROM "h2o_feet"

name: h2o_feet
time                   sample_level description   sample_water_level
----                   ------------------------   ------------------
2015-08-25T17:06:00Z                              3.284
2015-09-03T04:30:00Z   below 3 feet
2015-09-03T20:06:00Z   between 3 and 6 feet
2015-09-08T21:54:00Z                              3.412
```

查询返回 `h2o_feet` 中每个字段的两个随机抽样的字段值。 `h2o_feet` 有两个字段： `water_level` 和 `level description` 。

例三：列出匹配正则表达式的field的两个随机抽样的字段值

```
> SELECT SAMPLE(/level/,2) FROM "h2o_feet"

name: h2o_feet
time                   sample_level description   sample_water_level
----                   ------------------------   ------------------
2015-08-30T05:54:00Z   between 6 and 9 feet
2015-09-07T01:18:00Z                              7.854
2015-09-09T20:30:00Z                              7.32
```

```
9.  2015-09-13T19:18:00Z   between 3 and 6 feet
```

查询返回 `h2o_feet` 中含有 `level` 的字段的两个随机抽样的字段值。

例四：返回field两个随机抽样的字段值，以及其相关的tag和field

```
1.  > SELECT SAMPLE("water_level",2),"location","level description" FROM "h2o_feet"
2.
3.  name: h2o_feet
4.  time                  sample  location      level description
5.  ----                  ------  --------      -----------------
6.  2015-08-29T10:54:00Z  5.689   coyote_creek  between 3 and 6 feet
7.  2015-09-08T15:48:00Z  6.391   coyote_creek  between 6 and 9 feet
```

查询返回 `water_level` 的两个随机抽样的字段值，以及其相关的tag `location` 和field `level description` 。

例五：列出包含多个子句的field key的一个随机抽样的字段值

```
    > SELECT SAMPLE("water_level",1) FROM "h2o_feet" WHERE time >= '2015-08-
    18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z' AND "location" =
1.  'santa_monica' GROUP BY time(18m)
2.
3.  name: h2o_feet
4.  time                  sample
5.  ----                  ------
6.  2015-08-18T00:12:00Z  2.028
7.  2015-08-18T00:30:00Z  2.051
```

查询返回字段 `water_level` 的一个随机抽样的字段值。它涵盖 `2015-08-18T00：00：00Z` 和 `2015-08-18T00：30：00Z` 之间的时间段，并将结果按18分钟的时间间隔分组。

请注意， `GROUP BY time()` 子句没有覆盖点的原始时间戳。有关该行为的更详细解释，请参阅下面的问题一。

SAMPLE( )的常见问题

问题一： `SAMPLE()` 和 `GROUP BY time()`

使用 `SAMPLE()` 和 `GROUP BY time()` 子句的查询返回每个 `GROUP BY time()` 间隔的指定点数（ `N` ）。对于大多数 `GROUP BY time()` 查询，返回的时间戳是每个 `GROUP BY time()` 间隔的开始。 `GROUP BY time()` 查询与 `SAMPLE()` 函数的行为不同；它们保留原始数据点的时间戳。

例如

下面的查询每18分钟 `GROUP BY time()` 间隔返回两个随机的点。请注意，返回的时间戳是点的原始时间戳；它们不会被强制置为 `GROUP BY time()` 间隔的开始。

```
> SELECT SAMPLE("water_level",2) FROM "h2o_feet" WHERE time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z' AND "location" = 'santa_monica' GROUP BY time(18m)

name: h2o_feet
time                   sample
----                   ------
                           __
2015-08-18T00:06:00Z   2.116 |
2015-08-18T00:12:00Z   2.028 | <------- Randomly-selected points for the first time interval
                           --
                           __
2015-08-18T00:18:00Z   2.126 |
2015-08-18T00:30:00Z   2.051 | <------- Randomly-selected points for the second time interval
```

TOP()

返回最大的N个field值。

语法

```
SELECT TOP(<field_key>[,<tag_key(s)>],<N> )[,<tag_key(s)>|<field_key(s)>] [INTO_clause] FROM_clause [WHERE_clause] [GROUP_BY_clause] [ORDER_BY_clause] [LIMIT_clause] [OFFSET_clause] [SLIMIT_clause] [SOFFSET_clause]
```

语法描述

`TOP(field_key,N)`

返回field key的最大的N个field value。

`TOP(field_key,tag_key(s),N)`

返回某个tag key的N个tag value的最大的field value。

`TOP(field_key,N),tag_key(s),field_key(s)`

返回括号里的字段的最大N个field value，以及相关的tag或field，或者两者都有。

`TOP()` 支持所有的数值类型的field。

> 说明：
>
> - 如果一个field有两个或多个相等的field value， `TOP()` 返回时间戳最早的那个。
> - `TOP()` 和 `INTO` 子句一起使用的时候，和其他的函数有些不一样。

例子

例一：选择一个field的最大的三个值

```
> SELECT TOP("water_level",3) FROM "h2o_feet"

name: h2o_feet
time                   top
----                   ---
2015-08-29T07:18:00Z   9.957
2015-08-29T07:24:00Z   9.964
2015-08-29T07:30:00Z   9.954
```

该查询返回measurement `h2o_feet` 的字段 `water_level` 的最大的三个值。

例二：选择一个field的两个tag的分别最大的值

```
> SELECT TOP("water_level","location",2) FROM "h2o_feet"

name: h2o_feet
time                   top     location
----                   ---     --------
2015-08-29T03:54:00Z   7.205   santa_monica
2015-08-29T07:24:00Z   9.964   coyote_creek
```

该查询返回和tag `location` 相关的两个tag值的字段 `water_level` 的分别最大值。

例三：选择一个field的最大的四个值，以及其关联的tag和field

```
> SELECT TOP("water_level",4),"location","level description" FROM "h2o_feet"

name: h2o_feet
time                   top    location      level description
----                   ---    --------      -----------------
2015-08-29T07:18:00Z   9.957  coyote_creek  at or greater than 9 feet
2015-08-29T07:24:00Z   9.964  coyote_creek  at or greater than 9 feet
2015-08-29T07:30:00Z   9.954  coyote_creek  at or greater than 9 feet
```

```
9.  2015-08-29T07:36:00Z  9.941  coyote_creek  at or greater than 9 feet
```

查询返回 `water_level` 中最大的四个字段值以及tag `location` 和field `level description` 的相关值。

例四：选择一个field的最大的三个值，并且包括了多个子句

```
    > SELECT TOP("water_level",3),"location" FROM "h2o_feet" WHERE time >= '2015-
    08-18T00:00:00Z' AND time <= '2015-08-18T00:54:00Z' GROUP BY time(24m) ORDER BY
1.  time DESC
2.
3.  name: h2o_feet
4.  time                   top    location
5.  ----                   ---    --------
6.  2015-08-18T00:48:00Z  7.11   coyote_creek
7.  2015-08-18T00:54:00Z  6.982  coyote_creek
8.  2015-08-18T00:54:00Z  2.054  santa_monica
9.  2015-08-18T00:24:00Z  7.635  coyote_creek
10. 2015-08-18T00:30:00Z  7.5    coyote_creek
11. 2015-08-18T00:36:00Z  7.372  coyote_creek
12. 2015-08-18T00:00:00Z  8.12   coyote_creek
13. 2015-08-18T00:06:00Z  8.005  coyote_creek
14. 2015-08-18T00:12:00Z  7.887  coyote_creek
```

查询将返回在 `2015-08-18T00：00：00Z` 和 `2015-08-18T00：54：00Z` 之间的每24分钟间隔内， `water_level` 最大的三个值。它还以降序的时间戳顺序返回结果。

请注意， `GROUP BY time()` 子句不会覆盖点的原始时间戳。有关该行为的更详细解释，请参阅下面的问题一。

`TOP()` 的常见问题

问题一： `TOP()` 和 `GROUP BY time()` 子句

`TOP()` 和 `GROUP BY time()` 子句的查询返回每个 `GROUP BY time()` 间隔指定的点数。对于大多数 `GROUP BY time()` 查询，返回的时间戳被置为 `GROUP BY time()` 间隔的开始。 `GROUP BY time()` 查询与 `TOP()` 函数的行为不同； 它们保留原始数据点的时间戳。

例如

下面的查询返回每18分钟· `GROUP BY time()` 间隔的两点。请注意，返回的时间戳是点的原始时间戳； 它们不会被强制匹配 `GROUP BY time()` 间隔的开始。

```
> SELECT TOP("water_level",2) FROM "h2o_feet" WHERE time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z' AND "location" = 'santa_monica' GROUP BY time(18m)

name: h2o_feet
time                   top
----                   ------
                           __
2015-08-18T00:00:00Z  2.064 |
2015-08-18T00:06:00Z  2.116 | <------- Greatest points for the first time interval
                           --
                           __
2015-08-18T00:18:00Z  2.126 |
2015-08-18T00:30:00Z  2.051 | <------- Greatest points for the second time interval
                           --
```

问题二：`TOP()`和一个少于N个值得tag key

使用语法 `SELECT TOP (<field_key>, <tag_key>, <N>)` 的查询可以返回比预期少的点。如果tag具有X标签值，则查询指定N个值，当X小于N，则查询返回X点。

例如

下面的查询将要求tag `location` 的三个值的 `water_level` 的最大字段值。由于 `location` 具有两个值（ `santa_monica` 和 `coyote_creek` ），所以查询返回两点而不是三个。

```
> SELECT TOP("water_level","location",3) FROM "h2o_feet"

name: h2o_feet
time                   top    location
----                   ---    --------
2015-08-29T03:54:00Z  7.205  santa_monica
2015-08-29T07:24:00Z  9.964  coyote_creek
```

问题三：`TOP()`，tags和`INTO`子句

当与 `INTO` 子句和 `GROUP BY tag` 子句结合使用时，大多数InfluxQL函数将初始数据中的任何tag转换为新写入的数据中的field。此行为也适用于 `TOP()` 函数，除非 `TOP()` 包含一个tag key作为参数： `TOP(field_key, tag_key(s), N)` 。在这些情况下，系统将指定的tag作为新写入的数据中的tag。

例如

下面的代码块中的第一个查询返回与tag `location` 相关联的两个tag value的field `water_level` 中最大的字段值。它也将这些结果写入measurement `top_water_levels` 。第二个查询显示InfluxDB在 `top_water_levels` 中将 `location` 保存为tag。

```
> SELECT TOP("water_level","location",2) INTO "top_water_levels" FROM "h2o_feet"

name: result
time                written
----                -------
1970-01-01T00:00:00Z 2

> SHOW TAG KEYS FROM "top_water_levels"

name: top_water_levels
tagKey
------
location
```

Transformations

CEILING()

`CEILING()` 已经不再是一个函数了，具体请查看Issue #5930。

CUMULATIVE_SUM()

返回字段实时前序字段值的和。

基本语法

```
SELECT CUMULATIVE_SUM( [ * | <field_key> | /<regular_expression>/ ] ) [INTO_clause] FROM_clause [WHERE_clause] [GROUP_BY_clause] [ORDER_BY_clause] [LIMIT_clause] [OFFSET_clause] [SLIMIT_clause] [SOFFSET_clause]
```

基本语法描述

`CUMULATIVE_SUM(field_key)`

返回field key实时前序字段值的和。

`CUMULATIVE_SUM(/regular_expression/)`

返回满足正则表达式的所有字段的实时前序字段值的和。

`CUMULATIVE_SUM(*)`

返回measurement的所有字段的实时前序字段值的和。

`CUMULATIVE_SUM()` 支持所有的数值类型的field。

基本语法支持 `GROUP BY` tags子句，但是不支持 `GROUP BY` 时间。在高级语法中， `CUMULATIVE_SUM` 支持 `GROUP BY time()` 子句。

基本语法的例子

下面的1~4例子使用如下的数据：

```
> SELECT "water_level" FROM "h2o_feet" WHERE time >= '2015-08-18T00:00:00Z' AND
time <= '2015-08-18T00:30:00Z' AND "location" = 'santa_monica'

name: h2o_feet
time                   water_level
----                   -----------
2015-08-18T00:00:00Z   2.064
2015-08-18T00:06:00Z   2.116
2015-08-18T00:12:00Z   2.028
2015-08-18T00:18:00Z   2.126
2015-08-18T00:24:00Z   2.041
2015-08-18T00:30:00Z   2.051
```

例一：计算一个字段的实时前序字段值的和。

```
> SELECT CUMULATIVE_SUM("water_level") FROM "h2o_feet" WHERE time >= '2015-08-
18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z' AND "location" =
'santa_monica'

name: h2o_feet
time                   cumulative_sum
----                   --------------
2015-08-18T00:00:00Z   2.064
2015-08-18T00:06:00Z   4.18
2015-08-18T00:12:00Z   6.208
2015-08-18T00:18:00Z   8.334
2015-08-18T00:24:00Z   10.375
```

```
11. 2015-08-18T00:30:00Z   12.426
```

该查询返回measurement `h2o_feet` 的字段 `water_level` 的实时前序字段值的和。

例二：计算measurement中每个字段的实时前序字段值的和

```
    > SELECT CUMULATIVE_SUM(*) FROM "h2o_feet" WHERE time >= '2015-08-18T00:00:00Z'
 1. AND time <= '2015-08-18T00:30:00Z' AND "location" = 'santa_monica'
 2.
 3. name: h2o_feet
 4. time                   cumulative_sum_water_level
 5. ----                   --------------------------
 6. 2015-08-18T00:00:00Z   2.064
 7. 2015-08-18T00:06:00Z   4.18
 8. 2015-08-18T00:12:00Z   6.208
 9. 2015-08-18T00:18:00Z   8.334
10. 2015-08-18T00:24:00Z   10.375
11. 2015-08-18T00:30:00Z   12.426
```

该查询返回 `h2o_feet` 中每个数值类型的字段的实时前序字段值的和。 `h2o_feet` 只有一个数值类型的字段 `water_level` 。

例三：计算measurement中满足正则表达式的每个字段的实时前序字段值的和。

```
    > SELECT CUMULATIVE_SUM(/water/) FROM "h2o_feet" WHERE time >= '2015-08-
    18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z' AND "location" =
 1. 'santa_monica'
 2.
 3. name: h2o_feet
 4. time                   cumulative_sum_water_level
 5. ----                   --------------------------
 6. 2015-08-18T00:00:00Z   2.064
 7. 2015-08-18T00:06:00Z   4.18
 8. 2015-08-18T00:12:00Z   6.208
 9. 2015-08-18T00:18:00Z   8.334
10. 2015-08-18T00:24:00Z   10.375
11. 2015-08-18T00:30:00Z   12.426
```

查询返回measurement中含有单词 `word` 的每个数值字段的实时前序字段值的和。

例四：计算一个字段的实时前序字段值的和，并且包括了多个子句

```
> SELECT CUMULATIVE_SUM("water_level") FROM "h2o_feet" WHERE time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z' AND "location" = 'santa_monica' ORDER BY time DESC LIMIT 4 OFFSET 2

name: h2o_feet
time                   cumulative_sum
----                   --------------
2015-08-18T00:18:00Z   6.218
2015-08-18T00:12:00Z   8.246
2015-08-18T00:06:00Z   10.362
2015-08-18T00:00:00Z   12.426
```

查询将返回在 `2015-08-18T00：00：00Z` 和 `2015-08-18T00：30：00Z` 之间的实时前序字段值的和，以降序的时间戳顺序返回结果。并且限制返回的数据点为4，偏移数据点2个。

高级语法

```
SELECT CUMULATIVE_SUM(<function>( [ * | <field_key> | /<regular_expression>/ ] )) [INTO_clause] FROM_clause [WHERE_clause] GROUP_BY_clause [ORDER_BY_clause] [LIMIT_clause] [OFFSET_clause] [SLIMIT_clause] [SOFFSET_clause]
```

高级语法描述

高级语法要求一个 `GROUP BY time()` 子句和一个嵌套的InfluxQL函数。查询首先计算在指定时间区间嵌套函数的结果，然后应用 `CUMULATIVE_SUM()` 函数的结果。

`CUMULATIVE_SUM()` 支持以下嵌套函数： `COUNT(), MEAN(), MEDIAN(), MODE(), SUM(), FIRST(), LAST(), MIN(), MAX(), PERCENTILE()` 。

高级语法的例子

例一：计算平均值的cumulative和

```
> SELECT CUMULATIVE_SUM(MEAN("water_level")) FROM "h2o_feet" WHERE time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z' AND "location" = 'santa_monica' GROUP BY time(12m)

name: h2o_feet
time                   cumulative_sum
----                   --------------
2015-08-18T00:00:00Z   2.09
2015-08-18T00:12:00Z   4.167
2015-08-18T00:24:00Z   6.213
```

该查询返回每隔12分钟的 `water_level` 的平均值的实时和。

为得到这个结果，InfluxDB首先计算每隔12分钟的平均 `water_level` 值：

```
> SELECT MEAN("water_level") FROM "h2o_feet" WHERE time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z' AND "location" = 'santa_monica' GROUP BY time(12m)

name: h2o_feet
time                   mean
----                   ----
2015-08-18T00:00:00Z   2.09
2015-08-18T00:12:00Z   2.077
2015-08-18T00:24:00Z   2.0460000000000003
```

下一步，InfluxDB计算这些平均值的实时和。第二个点 `4.167` 是 `2.09` 和 `2.077` 的和，第三个点 `6.213` 是 `2.09` ， `2.077` 和 `2.04600000000003` 的和。

DERIVATIVE

返回字段的相邻两个点的变化率。

基本语法

```
SELECT DERIVATIVE( [ * | <field_key> | /<regular_expression>/ ] [ , <unit> ] ) [INTO_clause] FROM_clause [WHERE_clause] [GROUP_BY_clause] [ORDER_BY_clause] [LIMIT_clause] [OFFSET_clause] [SLIMIT_clause] [SOFFSET_clause]
```

基本语法描述

InfluxDB计算字段值之间的差并将结果转换为每 `unit` 变化率。 `unit` 参数是一个表示时间单位的字符，它是可选的。如果查询没有指定，则该 `unit` 默认为1秒（1s）。

`DERIVATIVE(field_key)`

返回field key的字段值的变化率。

`DERIVATIVE(/regular_expression/)`

返回满足正则表达式的所有字段的字段值的变化率。

`DERIVATIVE(*)`

返回measurement的所有字段的字段值的变化率。

`DERIVATIVE()` 支持所有的数值类型的field。

基本语法支持 `GROUP BY` tags子句，但是不支持 `GROUP BY` 时间。在高级语法中， `DERIVATIVE` 支持 `GROUP BY time()` 子句。

基本语法的例子

下面的1~5例子使用如下的数据：

```
> SELECT "water_level" FROM "h2o_feet" WHERE time >= '2015-08-18T00:00:00Z' AND
time <= '2015-08-18T00:30:00Z' AND "location" = 'santa_monica'

name: h2o_feet
time                   water_level
----                   -----------
2015-08-18T00:00:00Z   2.064
2015-08-18T00:06:00Z   2.116
2015-08-18T00:12:00Z   2.028
2015-08-18T00:18:00Z   2.126
2015-08-18T00:24:00Z   2.041
2015-08-18T00:30:00Z   2.051
```

例一：计算一个字段的变化率

```
> SELECT DERIVATIVE("water_level") FROM "h2o_feet" WHERE "location" =
'santa_monica' AND time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-
18T00:30:00Z'

name: h2o_feet
time                   derivative
----                   ----------
2015-08-18T00:06:00Z   0.00014444444444444457
2015-08-18T00:12:00Z   -0.00024444444444444465
2015-08-18T00:18:00Z   0.0002722222222222218
2015-08-18T00:24:00Z   -0.00023611111111111
2015-08-18T00:30:00Z   2.777777777777842e-05
```

该查询返回measurement `h2o_feet` 的字段 `water_level` 的每秒变化率。

第一个结果 `0.00014444444444444457` 是原始数据两个相邻字段值到每秒的变化率。InfluxDB计算字段值的变化，并且转化到每秒：

```
(2.116 - 2.064) / (360s / 1s)
```

```
--------------    ----------
       |               |
       |          the difference between the field values' timestamps / the
default unit
second field value - first field value
```

例二：计算一个字段的变化率并指定时间单位

```
> SELECT DERIVATIVE("water_level",6m) FROM "h2o_feet" WHERE "location" =
'santa_monica' AND time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-
18T00:30:00Z'

name: h2o_feet
time            derivative
----            ----------
2015-08-18T00:06:00Z    0.052000000000000046
2015-08-18T00:12:00Z    -0.08800000000000008
2015-08-18T00:18:00Z    0.09799999999999986
2015-08-18T00:24:00Z    -0.08499999999999996
2015-08-18T00:30:00Z    0.010000000000000231
```

该查询返回measurement `h2o_feet` 的字段 `water_level` 的每6分钟的变化率。

第一个结果 `0.052000000000000046` 是原始数据两个相邻字段值到每6分钟的变化率。InfluxDB计算字段值的变化，并且转化到每6分钟：

```
(2.116 - 2.064) / (6m / 6m)
--------------    ----------
       |               |
       |          the difference between the field values' timestamps / the
specified unit
second field value - first field value
```

例三：计算measurement中每个一个字段的变化率并指定时间单位

```
> SELECT DERIVATIVE(*,3m) FROM "h2o_feet" WHERE "location" = 'santa_monica' AND
time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z'


name: h2o_feet
time                    derivative_water_level
----                    ----------------------
2015-08-18T00:06:00Z    0.026000000000000023
```

```
2015-08-18T00:12:00Z   -0.04400000000000004
2015-08-18T00:18:00Z   0.04899999999999993
2015-08-18T00:24:00Z   -0.04249999999999998
2015-08-18T00:30:00Z   0.005000000000001155
```

该查询返回measurement `h2o_feet` 中每个数值字段的每3分钟的变化率。该measurement有一个数值字段: `water_level` 。

第一个结果 `0.026000000000000023` 是原始数据两个相邻字段值到每3分钟的变化率。InfluxDB计算字段值的变化，并且转化到每3分钟:

```
(2.116 - 2.064) / (6m / 3m)
--------------    ----------
       |              |
       |          the difference between the field values' timestamps / the
specified unit
second field value - first field value
```

例四: 计算measurement中满足正则表达式每个一个字段的变化率并指定时间单位

```
> SELECT DERIVATIVE(/water/,2m) FROM "h2o_feet" WHERE "location" =
'santa_monica' AND time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-
18T00:30:00Z'

name: h2o_feet
time                   derivative_water_level
----                   ----------------------
2015-08-18T00:06:00Z   0.01733333333333335
2015-08-18T00:12:00Z   -0.02933333333333336
2015-08-18T00:18:00Z   0.03266666666666662
2015-08-18T00:24:00Z   -0.02833333333333332
2015-08-18T00:30:00Z   0.0033333333333334103
```

该查询返回measurement `h2o_feet` 中满足正则表达式的每个数值字段的每2分钟的变化率。该measurement有一个数值字段: `water_level` 。

第一个结果 `0.01733333333333335` 是原始数据两个相邻字段值到每3分钟的变化率。InfluxDB计算字段值的变化，并且转化到每2分钟:

```
(2.116 - 2.064) / (6m / 2m)
--------------    ----------
       |              |
```

```
            |          the difference between the field values' timestamps / the
4.  specified unit
5.  second field value - first field value
```

例五：计算个一个字段的变化率并包括多个子句

```
    > SELECT DERIVATIVE("water_level") FROM "h2o_feet" WHERE "location" =
    'santa_monica' AND time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-
1.  18T00:30:00Z' ORDER BY time DESC LIMIT 1 OFFSET 2
2.
3.  name: h2o_feet
4.  time                   derivative
5.  ----                   ----------
6.  2015-08-18T00:12:00Z   -0.0002722222222222218
```

查询将返回在 `2015-08-18T00：00：00Z` 和 `2015-08-18T00：30：00Z` 之间， `water_level` 的每秒的变化率。它还以降序的时间戳顺序返回结果。 并且限制返回的数据点为1，偏移两个数据点

第一个结果 `0.0002722222222222218` 是原始数据两个相邻字段值到每秒的变化率。InfluxDB计算字段值的变化，并且转化到每秒：

```
1.  (2.126 - 2.028) / (360s / 1s)
2.  --------------    ----------
3.         |              |
           |          the difference between the field values' timestamps / the
4.  default unit
5.  second field value - first field value
```

高级语法

```
    SELECT DERIVATIVE(<function> ([ * | <field_key> | /<regular_expression>/ ]) [ ,
    <unit> ] ) [INTO_clause] FROM_clause [WHERE_clause] GROUP_BY_clause
    [ORDER_BY_clause] [LIMIT_clause] [OFFSET_clause] [SLIMIT_clause]
1.  [SOFFSET_clause]
```

高级语法的描述

高级语法要求一个 `GROUP BY time()` 子句和一个嵌套的InfluxQL函数。查询首先计算在指定时间区间嵌套函数的结果，然后应用 `DERIVATIVE()` 函数的结果。

`unit` 参数是一个整数后面跟时间字符，该参数是可选的。如果没有指定 `unit` ，那么 `unit` 默认就是 `GROUP BY time()` 的间隔。

`DERIVATIVE()` 支持以下嵌套函数：`COUNT(), MEAN(), MEDIAN(), MODE(), SUM(), FIRST(), LAST(), MIN(), MAX(), PERCENTILE()` 。

高级语法的例子

例一：计算一个字段平均值的变化率

```
> SELECT DERIVATIVE(MEAN("water_level")) FROM "h2o_feet" WHERE "location" = 'santa_monica' AND time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z' GROUP BY time(12m)

name: h2o_feet
time                   derivative
----                   ----------
2015-08-18T00:12:00Z   -0.0129999999999999
2015-08-18T00:24:00Z   -0.030999999999999694
```

该查询返回measurement `h2o_feet` 的字段 `water_level` 的每12分钟的平均值得每12分钟的变化率。

为了得到这个结果，InfluxDB首先计算 `water_level` 每12分钟的间隔的平均值，这一步就是使用带 `GROUP BY time()` 的 `MEAN()` 函数：

```
> SELECT MEAN("water_level") FROM "h2o_feet" WHERE "location" = 'santa_monica' AND time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z' GROUP BY time(12m)

name: h2o_feet
time                   mean
----                   ----
2015-08-18T00:00:00Z   2.09
2015-08-18T00:12:00Z   2.077
2015-08-18T00:24:00Z   2.0460000000000003
```

接下来，InfluxDB计算这些平均值每12分钟的变化率，第一个结果 `0.0129999999999999` 是两个相邻平均字段值到每12分钟的变化率。InfluxDB计算字段值的变化，并且转化到12分钟：

```
(2.077 - 2.09) / (12m / 12m)
-------------    ----------
      |               |
      |          the difference between the field values' timestamps / the default unit
second field value - first field value
```

例二：计算一个字段平均值的变化率，并指明时间单位

```
> SELECT DERIVATIVE(MEAN("water_level"),6m) FROM "h2o_feet" WHERE "location" = 'santa_monica' AND time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z' GROUP BY time(12m)

name: h2o_feet
time                   derivative
----                   ----------
2015-08-18T00:12:00Z   -0.00649999999999995
2015-08-18T00:24:00Z   -0.015499999999999847
```

该查询返回measurement `h2o_feet` 的字段 `water_level` 的每12分钟的平均值得每6分钟的变化率。

为了得到这个结果，InfluxDB首先计算 `water_level` 每12分钟的间隔的平均值，这一步就是使用带 `GROUP BY time()` 的 `MEAN()` 函数：

```
> SELECT MEAN("water_level") FROM "h2o_feet" WHERE "location" = 'santa_monica' AND time >= '2015-08-18T00:00:00Z' AND time <= '2015-08-18T00:30:00Z' GROUP BY time(12m)

name: h2o_feet
time                   mean
----                   ----
2015-08-18T00:00:00Z   2.09
2015-08-18T00:12:00Z   2.077
2015-08-18T00:24:00Z   2.0460000000000003
```

接下来，InfluxDB计算这些平均值每6分钟的变化率，第一个结果 `0.00649999999999995` 是两个相邻平均字段值到每6分钟的变化率。InfluxDB计算字段值的变化，并且转化到6分钟：

```
(2.077 - 2.09) / (12m / 6m)
-------------    ----------
       |               |
       |          the difference between the field values' timestamps / the specified unit
second field value - first field value
```

DIFFERENCE

数学运算符

数学运算符遵循标准的操作顺序。也就是说，圆括号优先于除法和乘法，而乘除法优先于加法和减法。例如 `5 / 2 + 3 * 2 = (5 / 2) + (3 * 2)` 和 `5 + 2 * 3 - 2 = 5 + (2 * 3) - 2` 。

数学运算符

加法

加一个常数。

```
SELECT "A" + 5 FROM "add"
SELECT * FROM "add" WHERE "A" + 5 > 10
```

两个字段相加。

```
SELECT "A" + "B" FROM "add"
SELECT * FROM "add" WHERE "A" + "B" >= 10
```

减法

减法里带常数。

```
SELECT 1 - "A" FROM "sub"
SELECT * FROM "sub" WHERE 1 - "A" <= 3
```

两个字段做减法。

```
SELECT "A" - "B" FROM "sub"
SELECT * FROM "sub" WHERE "A" - "B" <= 1
```

乘法

乘以一个常数。

```
SELECT 10 * "A" FROM "mult"
SELECT * FROM "mult" WHERE "A" * 10 >= 20
```

两个字段相乘。

```
1.  SELECT "A" * "B" * "C" FROM "mult"
2.  SELECT * FROM "mult" WHERE "A" * "B" <= 80
```

乘法和其他运算符混用。

```
1.  SELECT 10 * ("A" + "B" + "C") FROM "mult"
2.  SELECT 10 * ("A" - "B" - "C") FROM "mult"
3.  SELECT 10 * ("A" + "B" - "C") FROM "mult"
```

除法

除法里带常数。

```
1.  SELECT 10 / "A" FROM "div"
2.  SELECT * FROM "div" WHERE "A" / 10 <= 2
```

两个字段相除。

```
1.  SELECT "A" / "B" FROM "div"
2.  SELECT * FROM "div" WHERE "A" / "B" >= 10
```

除法和其他运算符混用。

```
1.  SELECT 10 / ("A" + "B" + "C") FROM "mult"
```

求模

模一个常数。

```
1.  SELECT "B" % 2 FROM "modulo"
2.  SELECT "B" FROM "modulo" WHERE "B" % 2 = 0
```

两个字段求模。

```
1.  SELECT "A" % "B" FROM "modulo"
2.  SELECT "A" FROM "modulo" WHERE "A" % "B" = 0
```

按位与

你可以在任何整数和布尔值中使用这个操作符，无论是字段或常数。该操作符不支持浮点数或字符串数据类型。并且不能混合使用整数和布尔值。

```
SELECT "A" & 255 FROM "bitfields"
SELECT "A" & "B" FROM "bitfields"
SELECT * FROM "data" WHERE "bitfield" & 15 > 0
SELECT "A" & "B" FROM "booleans"
SELECT ("A" ^ true) & "B" FROM "booleans"
```

按位或

你可以在任何整数和布尔值中使用这个操作符，无论是字段或常数。该操作符不支持浮点数或字符串数据类型。并且不能混合使用整数和布尔值。

```
SELECT "A" | 5 FROM "bitfields"
SELECT "A" | "B" FROM "bitfields"
SELECT * FROM "data" WHERE "bitfield" | 12 = 12
```

按位异或

你可以在任何整数和布尔值中使用这个操作符，无论是字段或常数。该操作符不支持浮点数或字符串数据类型。并且不能混合使用整数和布尔值。

```
SELECT "A" ^ 255 FROM "bitfields"
SELECT "A" ^ "B" FROM "bitfields"
SELECT * FROM "data" WHERE "bitfield" ^ 6 > 0
```

数学运算符的常见问题

问题一：带有通配和正则的数学运算符

InfluxDB在 `SELECT` 语句中不支持正则表达式或通配符。下面的查询是不合法的，系统会返回一个错误。

数学运算符和通配符一起使用。

```
> SELECT * + 2 FROM "nope"
ERR: unsupported expression with wildcard: * + 2
```

数学运算符和带函数的通配符一起使用。

```
> SELECT COUNT(*) / 2 FROM "nope"
ERR: unsupported expression with wildcard: count(*) / 2
```

数学运算符和正则表达式一起使用。

```
> SELECT /A/ + 2 FROM "nope"
ERR: error parsing query: found +, expected FROM at line 1, char 12
```

数学运算符和带函数的正则表达式一起使用。

```
> SELECT COUNT(/A/) + 2 FROM "nope"
ERR: unsupported expression with regex field: count(/A/) + 2
```

问题二：数学运算符和函数

在函数里面使用数学运算符现在不支持。

例如：

```
SELECT 10 * mean("value") FROM "cpu"
```

是允许的，但是

```
SELECT mean(10 * "value") FROM "cpu"
```

将会返回一个错误。

> InfluxQL支持子查询，这样可以达到函数里面使用数学运算符相同的功能。

不支持的运算符

比较

在 `SELECT` 中使用任意的 `=,!=,<,>,<=,>=,<>` ，都会返回空。

逻辑运算符

使用 `!|,NAND,XOR,NOR` 都会返回解析错误。

此外，在一个查询的 `SELECT` 子句中使用 `AND` 和 `OR` 不会像数学运算符只产生空的结果，因为他

们是InfluxQL的关键字。然而，你可以对布尔字段使用按位运算 `&` ， `|` 和 `^` 。

按位非

没有按位NOT运算符，因为你期望的结果取决于位的宽度。InfluxQL不知道位的宽度，所以无法实现适当的按位NOT运算符。

例如，如果位是8位的，那么你来说，1代表 `0000 0001` 。

按位非本应该返回 `1111 1110` ，即整数254。

然而，如果位是16位的，那么1代表 `0000 0000 0000 0001` 。按位非本应该返回 `1111 1111 1111 1110` ，即整数65534。

解决方案

你可以使用 `^` 加位数的数值来实现按位非的效果：

对于8位的数据：

```
1.  SELECT "A" ^ 255 FROM "data"
```

对于16位数据：

```
1.  SELECT "A" ^ 65535 FROM "data"
```

对于32位数据：

```
1.  SELECT "A" ^ 4294967295 FROM "data"
```

里面的每一个常数的计算方式为 `(2 ** width) - 1` 。

认证和授权

认证

InfluxDB的HTTP API和命令行界面（CLI），包括简单的基于用户凭据的内置认证。当开启认证时，InfluxDB只会执行发送中带有有效证书的HTTP请求。

> 注意：认证只发生在HTTP请求范围内。插件目前不具备认证请求的能力，（例如Graphite、collectd等）是没有认证的。

创建认证

1. 至少创建一个admin用户

> 如果你开启了认证但是没有用户，那么InfluxDB将不会开启认证，而且只有在创建了一个admin用户之后才会接受外部请求。

当创建一个admin用户后，InfluxDB才能开启认证。

2. 在配置文件中，认证默认是不开启的

将 `[http]` 区域的配置 `auth-enabled` 设为 `true` ，可以开启认证：

```
1.  [http]
2.    enabled = true
3.    bind-address = ":8086"
      auth-enabled = true #
4.
5.    log-enabled = true
6.    write-tracing = false
7.    pprof-enabled = false
8.    https-enabled = false
9.    https-certificate = "/etc/ssl/influxdb.pem"
```

3. 重启进程

现在InfluxDB会核对每个请求中的用户信息，只会处理已有用户而认证通过的请求。

认证请求

HTTP API中的认证

有两个HTTP API进行验证的方式。

如果使用基本身份验证和URL查询参数进行身份验证，则以查询参数中指定的用户优先。下面的示例中的查询假定用户是admin用户。

用RFC 2617中所描述的基本身份验证

这是提供用户的首选方法。例：

```
curl -G http://localhost:8086/query -u todd:influxdb4ever --data-urlencode
"q=SHOW DATABASES"
```

在URL的参数或是请求体里面提供认证

设置 `u` 和 `p` 参数， 例：

```
curl -G "http://localhost:8086/query?u=todd&p=influxdb4ever" --data-urlencode
"q=SHOW DATABASES"
```

认证在请求体中的例子：

```
curl -G http://localhost:8086/query --data-urlencode "u=todd" --data-urlencode
"p=influxdb4ever" --data-urlencode "q=SHOW DATABASES"
```

CLI中的认证

在CLI中有三种认证方式。

设置`INFLUX_USERNAME`和`INFLUX_PASSWORD`环境变量

例如：

```
export INFLUX_USERNAME todd
export INFLUX_PASSWORD influxdb4ever
echo $INFLUX_USERNAME $INFLUX_PASSWORD
todd influxdb4ever

influx
Connected to http://localhost:8086 version 1.3.x
InfluxDB shell 1.3.x
```

开启CLI时设置`username`和`password`

例如：

```
influx -username todd -password influxdb4ever
Connected to http://localhost:8086 version 1.3.x
InfluxDB shell 1.3.x
```

开启CLI后使用 `auth <username> <password>`

例如：

```
influx
Connected to http://localhost:8086 version 1.3.x
InfluxDB shell 1.3.x
> auth
username: todd
password:
>
```

授权

当开启认证之后，授权也就开启了。默认情况下，所有的用户都有所有的权限。

用户类型和权限

admin用户

admin用户有所有数据库的读写权限，这些所有的权限包括如下：

数据库管理：

- `CREATE DATABASE` ， `DROP DATABASE`
- `DROP SERIES` ， `DROP MEASUREMENT`
- `CREATE RETENTION POLICY` ， `ALTER RETENTION POLICY` ， 和 `DROP RETENTION POLICY`
- `CREATE CONTINUOUS QUERY` 和 `DROP CONTINUOUS QUERY`

用户管理：

- admin用户管理： `CREATE USER` ， `GRANT ALL PRIVILEGES` ， `REVOKE ALL PRIVILEGES` ，和 `SHOW USERS`
- 非admin用户管理： `CREATE USER` ， `GRANT [READ,WRITE,ALL]` ， `REVOKE [READ,WRITE,ALL]` ，和 `SHOW GRANTS`

- 一般admin用户管理： `SET PASSWORD` 和 `DROP USER`

非admin用户

非admin用户对于每个数据库，有如下三个权限：

- `READ`
- `WRITE`
- `ALL` （包括 `READ` 和 `WRITE` ）

`READ` ， `WRITE` 和 `ALL` 控制到每个数据库每个用户上。一个新的非admin用户对任何数据库都没有权限，除非被admin用户指定一个数据库权限。非admin用户可以在他们有 `READ` 或/和 `WRITE` 权限的机器上运行 `SHOW` 命令。

用户管理命令

admin用户管理

当开启HTTP认证之后，在你操作系统之前，InfluxDB要求你至少创建一个admin用户。

```
1.  CREATE USER admin WITH PASSWORD '<password>' WITH ALL PRIVILEGES
```

`CREATE` 另一个admin用户

```
1.  CREATE USER <username> WITH PASSWORD '<password>' WITH ALL PRIVILEGES
```

CLI例子：

```
1.  > CREATE USER paul WITH PASSWORD 'timeseries4days' WITH ALL PRIVILEGES
2.  >
```

> 注意：重复创建确切的用户语句是幂等的。如果有值发生变化，数据库将返回一个重复的用户错误。CLI例子：
>
> ```
> 1.  CREATE USER todd WITH PASSWORD '123456' WITH ALL PRIVILEGES
> 2.  CREATE USER todd WITH PASSWORD '123456' WITH ALL PRIVILEGES
> 3.  CREATE USER todd WITH PASSWORD '123' WITH ALL PRIVILEGES
> 4.  ERR: user already exists
> 5.  CREATE USER todd WITH PASSWORD '123456'
> 6.  ERR: user already exists
> 7.  CREATE USER todd WITH PASSWORD '123456' WITH ALL PRIVILEGES
> ```

给一个存在的用户 `GRANT` 权限

```
GRANT ALL PRIVILEGES TO <username>
```

CLI例子:

```
> GRANT ALL PRIVILEGES TO "todd"
>
```

给一个admin用户 `REVOKE` 权限

```
REVOKE ALL PRIVILEGES FROM <username>
```

CLI例子:

```
> REVOKE ALL PRIVILEGES FROM "todd"
>
```

`SHOW` 所有存在的用户以及其admin状态

```
SHOW USERS
```

CLI例子:

```
> SHOW USERS
user      admin
todd     false
paul     true
hermione false
dobby    false
```

非admin用户管理

`CREATE` 一个非admin用户:

```
CREATE USER <username> WITH PASSWORD '<password>'
```

CLI例子:

```
> CREATE USER todd WITH PASSWORD 'influxdb41yf3'
> CREATE USER alice WITH PASSWORD 'wonder\'land'
> CREATE USER "rachel_smith" WITH PASSWORD 'asdf1234!'
> CREATE USER "monitoring-robot" WITH PASSWORD 'XXXXX'
```

```
> CREATE USER "$savyadmin" WITH PASSWORD 'm3tr1cL0v3r'
>
```

> 注意：
>
> - 用户名如果用一个数字的开始，或者是一个influxql关键字，又或者包含任何特殊字符，例如： `!@#$%^&*()-` ，则必须用双引号引起来
> - 密码字符串必须用单引号引起来。
> - 在验证请求时不包含单引号。
>
> 如果密码包含单引号或换行符，当提交认证请求时，应该用反斜杠转义。

`GRANT READ, WRITE` 或者 `ALL` 数据库权限给一个存在的用户

```
GRANT [READ,WRITE,ALL] ON <database_name> TO <username>
```

CLI例子： 给用户 `todd``GRANT` 数据库 `NOAA_water_database` 的 `READ` 的权限：

```
> GRANT READ ON "NOAA_water_database" TO "todd"
>
```

给用户 `todd``GRANT` 数据库 `NOAA_water_database` 的 `ALL` 的权限：

```
> GRANT ALL ON "NOAA_water_database" TO "todd"
>
```

`REVOKE READ, WRITE` 或者 `ALL` 数据库权限给一个存在的用户

```
REVOKE [READ,WRITE,ALL] ON <database_name> FROM <username>
```

CLI例子： 给用户 `todd``REVOKE` 数据库 `NOAA_water_database` 的 `ALL` 的权限：

```
> REVOKE ALL ON "NOAA_water_database" FROM "todd"
>
```

给用户 `todd``REVOKE` 数据库 `NOAA_water_database` 的 `WRITE` 的权限：

```
> REVOKE WRITE ON "NOAA_water_database" FROM "todd"
>
```

`SHOW` 一个用户的数据库权限

```
SHOW GRANTS FOR <user_name>
```

CLI例子：

```
> SHOW GRANTS FOR "todd"
database                    privilege
NOAA_water_database            WRITE
another_database_name         READ
yet_another_database_name   ALL PRIVILEGES
```

普通admin和非admin用户管理

重新设置一个用户的密码

```
SET PASSWORD FOR <username> = '<password>'
```

CLI例子：

```
> SET PASSWORD FOR "todd" = 'influxdb4ever'
>
```

```
> **Note:** The password [string]($Query_language-influxdb-v1.3-query_language-
spec-#strings) must be wrapped in single quotes.
```

`DROP` 一个用户

```
DROP USER <username>
```

CLI例子：

```
> DROP USER "todd"
>
```

认证和授权的HTTP错误

没有认证或者是带有不正确认证信息的请求发到InfluxDB，将会返回一个 `HTTP 401 Unauthorized` 的错误。

没有授权的用户的请求发到InfluxDB，将会返回一个 `HTTP 403 Forbidden` 的错误。

故障排除

FAQ

本页面讨论了被频繁问及的易混淆的InfluxDB相对于其他数据库系统以意想不到的方式行事的地方。

系统监控

系统监控意味着InfluxDB系统用户可以获得有关系统本身的所有统计和诊断信息。其目的是协助对数据库本身进行故障排除和性能分析。

查询管理

借助InfluxDB的查询管理功能，用户可以识别当前正在运行的查询，并能够终止超载系统的查询。 另外，用户可以通过几个配置设置来阻止和停止执行低效查询。

FAQ

管理

如何在密码中包含单引号？

在创建密码和发送身份验证请求时使用反斜杠（\）来转义单引号。

如何查看InfluxDB的版本？

有几种方法可以查看InfluxDB的版本：

在终端运行 `influxd version`：

```
$ influxd version

InfluxDB  v1.3.0  (git: master b7bb7e8359642b6e071735b50ae41f5eb343fd42)
```

`curl` 路径 `/ping`：

```
$ curl -i 'http://localhost:8086/ping'

HTTP/1.1 204 No Content
Content-Type: application/json
Request-Id: 1e08aeb6-fec0-11e6-8486-000000000000
 X-Influxdb-Version: 1.3.x
Date: Wed, 01 Mar 2017 20:46:17 GMT
```

运行InfluxDB的命令行：

```
$ influx

Connected to http://localhost:8086 version 1.3.x
InfluxDB shell version: 1.3.x
```

在日志里查看HTTP返回结果：

```
$ journald-ctl -u influxdb.service

```

```
    Mar 01 20:49:45 rk-api influxd[29560]: [httpd] 127.0.0.1 - -
    [01/Mar/2017:20:49:45 +0000] "POST /query?db=&epoch=ns&q=SHOW+DATABASES
    HTTP/1.1" 200 151 "-"  "InfluxDBShell/1.3.x"  9a4371a1-fec0-11e6-84b6-
3.  000000000000 1709
```

怎样查看InfluxDB的日志？

在System V操作系统上，日志存储在/var/log/influxdb/下。在systemd操作系统上，可以使用 `journalctl` 访问日志。 使用 `journalctl -u influxdb` 查看日志或 `journalctl -u influxdb> influxd.log` 将日志打印到文本文件。使用systemd时，日志保留取决于系统的日记设置。

分片组的保留时间和保留策略之间的关系？

nfluxDB将数据存储在分片组中。一个分片组覆盖特定的时间间隔； InfluxDB通过查看相关保留策略（RP）的 `DURATION` 来确定时间间隔。下表列出了RP的 `DURATION` 和分片组的时间间隔之间的默认关系：

RP持续时间	分片组间隔
<2天	1小时
>= 2天，<= 6个月	1天
> 6个月	7天

用户还可以使用 `CREATE RETENTION POLICY` 和 `ALTER RETENTION POLICY` 语句配置分片组持续时间。使用 `SHOW RETENTION POLICY` 语句检查保留策略的分片组持续时间。

为什么在修改了RP之后数据没有被删除？

有几个因素可以解释为什么保留策略（RP）更改后数据可能不会立即丢失。

第一个也是最可能的原因是，默认情况下，InfluxDB每30分钟检查一次强制RP。可能需要等待下一次RP检查InfluxDB才能删除RP的新 `DURATION` 设置之外的数据。30分钟的间隔是可配置的。

其次，改变RP的 `DURATION` 和 `SHARD DURATION` 可能会导致意外的数据保留。InfluxDB将数据存储在包含特定RP和时间间隔的分片组中。当InfluxDB执行一个RP时，它会删除整个分片组，而不是单个数据点。InfluxDB不能拆分分片组。

如果RP的新 `DURATION` 小于旧的 `SHARD DURATION` ，并且InfluxDB正在将数据写入其中一个较旧的分片组，则系统将被迫将所有数据保留在该分片组中。即使该分片组中的某些数据不在新的 `DURATION` 中，也会发生这种情况。一旦所有的数据都在新的 `DURATION` 之外，InfluxDB将删除该分片组。系统将开始将数据写入新的分片组，这些分片组具有新的更短的 `SHARD DURATION` ，以防止写入不被期望的数据。

为什么InfluxDB无法在配置文件中解析微秒单位？

用于指定微秒持续时间单位的语法在配置设置，和写入，查询以及在InfluxDB命令行界面（CLI）中设置精度方面有所不同。 下表显示了每个类别支持的语法：

单位	配置文件	HTTP API写入	所有的查询	CLI精度命令行
u				
us				
µ				
µs				

如果配置选项指定u或µ语法，InfluxDB将无法启动并在日志中报告以下错误：

```
1.  run: parse config: time: unknown unit [µ|u] in duration [<integer>µ|<integer>u]
```

命令行

怎么让InfluxDB的CLI返回用户可读的时间戳?

当你第一次连CLI，可以指定rfc3339精度：

```
1.  $ influx -precision rfc3339
```

此外，如果你已经连接了CLI，则可以通过指令来指定：

```
1.  $ influx
2.  Connected to http://localhost:8086 version 0.xx.x
3.  InfluxDB shell 0.xx.x
4.  > precision rfc3339
5.  >
```

一个非admin用户怎么在InfluxDB的CLI下USE一个数据库？

在v1.3之前的版本中，非admin用户是不能在CLI执行 `USE <database_name>` 的查询的，即使拥有这个数据库的 `READ` 或/和 `Write` 权限。

从1.3开始，非admin用户也可以执行 `USE <database_name>` 的查询只要拥有这个数据库的 `READ` 或/和 `Write` 权限。如果非admin使用 `USE` 一个数据库，但是没有权限时，系统会返回一个错误：

```
ERR: Database <database_name> doesn't exist. Run SHOW DATABASES for a list of existing databases.
```

> 注意：非admin用户执行 `SHOW DATABASES` ，只返回有权限的数据库。

在InfluxDB的CLI下，如何写入一个非默认的retention policy？

使用语法 `INSERT INTO [<database>.]<retention_policy> <line_protocol>` 可以在CLI下把数据写入非 `DEFAULT` 的retention policy里面。（如果使用HTTP来写入数据的话，可以通过参数 `db` 和 `rp` 来指定数据库和retention policy）。

例如：

```
> INSERT INTO one_day mortality bool=true
Using retention policy one_day
> SELECT * FROM "mydb"."one_day"."mortality"
name: mortality
---------------
time                            bool
2016-09-13T22:29:43.229530864Z  true
```

注意如果要查询出非 `DEFAULT` 的retention policy，需要指定完整的measurement路径：

```
"<database>"."<retention_policy>"."<measurement>"
```

怎么取消一个长期运行的查询？

在CLI下，你可以用 `Ctrl+C` 来取消执行的查询。对于其他的长时间运行的查询，你可以先使用 `SHOW QUERIES` 列出来，然后使用 `KILL QUERY` 来停止对应的查询。

为什么不能查询布尔型的field value？

对于写和读，接受布尔型的数据的语法不一样。

布尔语法	写	读
t,f		
T,F		
true,false		
True,False		
TRUE,FALSE		

例如， `SELECT * FROM "hamlet" WHERE "bool"=True` 返回所有 `bool` 的值为 `TRUE` 的，但是 `SELECT * FROM "hamlet" WHERE "bool"=T` 什么都不会返回。Github Issue #3939

InfluxDB怎么处理不同shard里面的字段类型冲突？

字段的值类型可以是浮点，整数，字符串和布尔型。字段的类型在同一个shard里面是一致的，但是在不同的shard里面可以不一样。

SELECT语句

如果所有值都具有相同的类型， `SELECT` 语句会返回所有字段值。如果字段的值类型在不同的shard里面不一样，InfluxDB首先执行正常操作并将所有值返回为出现在下面的列表的第一个类型：浮点，整数，字符串和布尔型。

如果你的数据字段的值类型不符，使用语法 `<field_key>::<type>` 查询不同的数据类型。

例子

measurement `just_my_type` 有一个字段叫作 `my_field` 。 `my_field` 在四个不同shard里面有四个不同的类型分别为（浮点，整数，字符串和布尔型）。

`SELECT*` 只返回浮点数和整数字段值。注意InfluxDB在返回时把整数转化为了浮点类型。

```
SELECT * FROM just_my_type

name: just_my_type
------------------
time                            my_field
2016-06-03T15:45:00Z      9.87034
2016-06-03T16:45:00Z      7
```

`SELECT<field_key>::<type>[…]` 返回所有值类型。我们可以在InfluxDB列名里增加自己的列输出的值类型。在可能的情况下，InfluxDB字段值会转到另一个类型；它把整数7在第一列中转化为了浮点数，而且把9.879034在第二列中转化为了整数。InfluxDB不能把浮点或整数转化为字符串或布尔值。

```
    SELECT
    "my_field"::float,"my_field"::integer,"my_field"::string,"my_field"::boolean
FROM just_my_type

name: just_my_type
------------------
```

```
    time                           my_field     my_field_1     my_field_2
5.  my_field_3
6.  2016-06-03T15:45:00Z     9.87034      9
7.  2016-06-03T16:45:00Z     7            7
8.  2016-06-03T17:45:00Z                                 a string
9.  2016-06-03T18:45:00Z                                                 true
```

SHOW FIELD KEYS查询

`SHOW FIELD KEYS` 会返回相关field在不同的shard里面的每种类型。

measurement `just_my_type` 有一个字段叫作 `my_field` 。 `my_field` 在四个不同shard里面有四个不同的类型分别为（浮点，整数，字符串和布尔型）。 `SHOW FIELD KEYS` 返回所有四种数据类型：

```
1.  > SHOW FIELD KEYS
2.
3.  name: just_my_type
4.  fieldKey   fieldType
5.  --------   ---------
6.  my_field   float
7.  my_field   string
8.  my_field   integer
9.  my_field   boolean
```

InfluxDB可以存储的最大最小整数是什么？

系统监控

查询管理

错误信息