PipelineDB 官方文档(0.9.6)

伏念译 (hello-funian@foxmail.com)

译者注：这篇官方文档是基于最新的 PipelineDB 0.9.6 版本翻译的。如有雷同，纯属巧合。

欢迎使用 PipelineDB's 文档！希望在这里你可以找到关于如何使用 Pipelinedb 以及它的工作原理的一切。如果你觉得这里缺少了一些东西，或者感觉到困惑，或者你发现了一个 bug 或者文档错误，亦或你觉得我们说的不对，别犹豫，赶快发邮件告知我们吧，邮件地址是： docs@pipelinedb.com.

简介

如果你想直接进入快速上手环节，请直接阅读“快速入门”章节。

概述

PipelineDB 用于在流数据上做 SQL 查询。流查询的结果存储在常规的表或者视图中。这样流查询可以被看做成拥有很高的吞吐量，可以随着数据的流入而逐渐

变化的物化视图。和其他数据处理系统类似，PipelineDB 是用来处理特定的任务的，而不是其他系统的扩充。

您可以立即查看客户端编程 和 快速入门 章节的样例来快速入门 PipelineDB。

PipelineDB 是什么

PipelineDB 被设计用于做 SQL 查询缩减后的流数据集。例如：求和和聚合；滑动时间窗口的计算；文本搜索过滤；空间搜索过滤等等。通过减少输入数据流的量， PipelineDB 能够戏剧性地大幅度减少需要存储到磁盘中的消息量，因为只有流查询的结果需要存储到磁盘中。裸数据一旦被流查询读取玩了就丢弃掉。

大量被传输到 PipelineDB 的数据因此可以被看做**虚拟数据。** 数据的虚拟化是 PipelineDB 所有秘密的核心，这也是它为什么能够使用相对较小的硬件设备高效处理大量数据的原因。

PipelineDB 的目标是消除掉很多常规数据传输时候的 ETL 过程。裸数据可以直接流入到 PipelineDB 中，并被你声明的流查询不断的实时转换和提取。这让它不必把加工好的数据加载到数据库之前就可以周期性地处理细粒度数据——当然只要这个处理过程可以用 SQL 查询来定义。

PipelineDB 的设计理念是实用主义第一。这也是我们为什么完全兼容 PostgreSQL 9.5 的原因。我们没有发明我们专有的语法，我们甚至都没有 PipelineDB 客户端，因为任何可用在 PostgreSQL 数据库工作的库都可以在 PipelineDB 上很好的工作。

PipelineDB 不是什么

鉴于流查询必须是优先级更高的工作，PipelineDB 并非一个即席查询的数据仓库。而流查询的结果可能会在即席查询中展示，所有传输到 PipelineDB 中的原始数据都可能不会存储因为数据记录在被读取后就会被抛掉。除此之外，如果流计算不能或不能用 SQL 来表示，那么 PipelineDB 可能不适合这类工作。

快速入门

初始化 pipelinedb 的数据目录，启动 pipelinedb 数据库服务实例：:

```
pipeline-init -D <data directory>
pipelinedb -D <data directory>
```

现在开启流式查询。这条命令只需要执行一次，重启后自动激活流。

```
psql -h localhost -p 5432 -d pipeline -c "ACTIVATE"
```

维基百科的页面流量统计

这个样例中我们会计算一些基础的关于 Wikipedia 页面浏览数的基础统计量。数据集的每个记录都包含了每个维基百科页面的小时级页面统计量。记录格式如下：

```
hour | project | page title | view count | bytes served
```

第一，让我们使用 psql 来创建我们的流查询：

```
psql -h localhost -p 5432 -d pipeline -c "
CREATE STREAM wiki_stream (hour timestamp, project text, title text, view_count bigint,
size bigint);
CREATE CONTINUOUS VIEW wiki_stats AS
SELECT hour, project,
       count(*) AS total_pages,
       sum(view_count) AS total_views,
       min(view_count) AS min_views,
       max(view_count) AS max_views,
       avg(view_count) AS avg_views,
       percentile_cont(0.99) WITHIN GROUP (ORDER BY view_count) AS p99_views,
       sum(size) AS total_bytes_served
FROM wiki_stream
GROUP BY hour, project;"
```

现在我们将解压数据集后作为一个数据流，然后写入 `stdin`，这样就可以用来当做 copy 命令的输入：

```
curl -sL http://pipelinedb.com/data/wiki-pagecounts | gunzip | \
       psql -h localhost -p 5432 -d pipeline -c "
       COPY wiki_stream (hour, project, title, view_count, size) FROM STDIN"
```

注意这个数据集非常大，因此上面的命令会运行相当长的时间（你可以在运行时任何事情取消运行）。当上面的程序运行时候，流视图就可以从输入流中查询出数据：

```
psql -h localhost -p 5432 -d pipeline -c "
SELECT * FROM wiki_stats ORDER BY total_views DESC";
```

安装

下载匹配你操作系统的 PipelineDB 的二进制安装包，下载地址如下： 点我下载

RPM 包

安装 PipelineDB 的 RPM 包，运行:

```
sudo rpm -ivh pipelinedb-<version>.rpm
```

执行后会把 PipelineDB 安装在 `/usr/lib/pipelinedb` 目录下。如果想安装在指定目录，需要使用参数`--prefix`：

```
sudo rpm -ivh --prefix=/path/to/pipelinedb pipelinedb-<version>.rpm
```

Debian 包

执行下面命令安装 PipelineDB 的 Debian 包:

```
sudo dpkg -i pipelinedb-<version>.deb
```

执行后会把 PipelineDB 安装在`/usr/lib/pipelinedb`目录下

OS X

OS X 系统下，只需要双击文件 `pipelinedb-<version>.pkg` 来启动 OS X 的安装向导。对于老版本的 OS X，你可能需要安装一些 PipelineDB 所依赖的包：:

```
brew install json-c freexl
```

初始化 PipelineDB

一旦安装完毕 PipelineDB ，你就可以初始化数据目录。这个目录是是 PipelineDB 存储文件以及数据库相关数据的目录。初始化数据目录命令如下：:

```
pipeline-init -D <data directory>
```

注意参数 `<data directory>` 是一个执行命令前并不存在的目录。一旦成功创建这个目录，你可以开启 PipelineDB 服务。

运行 PipelineDB 数据库服务

想要在后台运行 PipelineDB 服务，可以使用命令 `pipeline-ctl`，并启动目录设置为你刚刚初始化的数据目录。:

```
pipeline-ctl -D <data directory> -l pipelinedb.log start
```

参数 `-l` 指定启动生成的日志文件。命令 `pipeline-ctl` 也可以用于停止数据库服务:

```
pipeline-ctl -D <data directory> stop
```

执行命令 `pipeline-ctl --help` 来查看其它可用的功能。最后，PipelineDB 数据库服务也可以直接用下面的命令执行并以前端方式运行（译者注：意思是不在后台服务运行，按 ctl+c 关闭这个命令，数据库也直接关闭掉）:

```
pipelinedb -D <data directory>
```

如果想连接一个正在运行的数据库服务，连接的数据库是默认的 pipeline，可以使用命令 `pipeline`:

```
pipeline pipeline
```

PostgreSQL's 表中的客户端 `psql` 同样可以用于连接 PipelineDB。注意 PipelineDB 默认的端口是 `5432`:

```
psql -p 5432 -h localhost pipeline
```

现在你可以查看章节快速入门来把流数据插入到 PipelineDB 数据库中。

Debug 模式

在 PipelineDB 的 0.9.1 中开始引入

PipelineDB 数据库服务可以允许在 debug 模式下，当数据库出现问题，比如崩溃的情况下，可以进行 debug 以便进行断言或者获取额外的诊断信息。Debug 模式的设计，方便用户数据库出现问题时候，我们可以更好的支持。Debug 模式可以用下面两种方法运行:

首先在执行命令 `pipeline-ctl` 时候带上参数 `-d`/`--debug`:

```
pipeline-ctl -d -D ... start
pipeline-ctl --debug -D ... start
```

或者直接执行 debug 模式的命令:

```
pipelinedb-debug -D <data directory>
```

注意

Debug 模式使用非优化的代码，包含了很多断言 assertion 代码以及 debug 标识，因此性能上没有优化，debug 模式只能在数据库发生错误的时候进行使用。

配置

PipelineDB 数据库的配置一般是和 PostgreSQL 的配置是一致的，因此是一个很好的地方来查看 `pipelinedb.conf` 配置所起的作用。

默认地，PipelineDB 是不能接收本机以外的外部连接的。要想接收本机以外的连接，首先在文件 `pipelinedb.conf` 设置参数：

```
listen_addresses = '*'
```

然后在配置文件 `pg_hba.conf` 中添加类似下面的一行来允许外部连接:

```
host    all             all             <ip address>/<subnet>           md5
```

例如，想接收来自任意主机的连接:

```
host    all             all             0.0.0.0/0           md5
```

现在你已经把 PipelineDB 运行起来了。查看章节流视图或快速入门章节开始使用。

Docker 镜像

感谢 Josh Berkus，他提供了 PipelineDB 的 Docker 镜像。使用如下命令运行：

```
docker run -v /dev/shm:/dev/shm pipelinedb/pipelinedb
```

这个镜像对外开放 5432 端口用于和 PipelineDB 交互。认证账户和密码都是 pipeline。

Docker 镜像下数据库会被安装在挂载的目录 `/mnt/pipelinedb` 中。因此如果你想把数据库安装实际的存储地方或者你想修改配置文件，那么在第一次运行镜像前，你需要先挂载那个目录到实际的磁盘中即可。

注意

镜像方式的安装一般只适用于在你的笔记本电脑上测试运行，如果你想把它部署在生产环境，你需要编辑 pipelinedb.conf，并且需要大幅度提高大多数配置所使用的资源（译者注：比如内存，连接池数）

流视图

PipelineDB 的一个基本的概念抽象叫作流视图。一个流视图与数据库常规的视图非常类似，区别是它是从流（或表）的组合做为输入抽取数据，并且实时增加更新作为新的数据用于写入成其他的输入数据。

一旦流视图读取了一行流记录，读取完毕后就丢弃这条记录。流记录不会存取在任何位置。对流视图来说，唯一持久化的数据就是 `SELECT * FROM that_view` 返回的任何数据。因此你可以把流视图看成一个非常高吞吐量、实时的物化视图。

创建流视图

下面是创建流视图的语法：

```
CREATE CONTINUOUS VIEW name AS query
```

query 是一个 PostgreSQL 的 `SELECT` 查询语句返回的结果集:

```
SELECT [ DISTINCT [ ON ( expression [, ...] ) ] ]
    expression [ [ AS ] output_name ] [, ...]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]

 from_item 可以是如下之一:

    stream_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    table_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    from_item [ NATURAL ] join_type from_item [ ON join_condition ]
```

注意

本节提到了流的概念，流与数据表很类似，也是流视图在 FROM 子句中读取数据的数据源。在流这一章节中会解释的更深入，但是现在你可以把流看做一个只增加数据的表。

expression

一个 PostgreSQL expression 表达式

output_name

一个可选择的用于命名表达式的标识符。

condition

任何表达式计算结果是 `boolean` 类型的值都可以。任何不满足这个条件的记录都会在输出结果中过滤掉。如果实际的行记录值在替换引用变量后计算结果是 true，那么这行数据就满足了这个 condition 条件。

window_name

可以引用 OVER 子句的或后面定义的窗口中的名称

window_definition

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression ] [ NULLS { FIRST | LAST } ] [, ...] ]
[ frame_clause ]
```

注意

PipelineDB 的 **window_definitions** 如果输入的记录是来自于流数据，那么不支持 `ORDER BY` 子句。在这种情况下，流记录的 `arrival_timestamp` 字段可以隐式用于 `ORDER BY` 子句。

frame_clause

定义依赖窗口函数的窗口框。窗口框是每个查询的记录（被称作当前记录）对应的一些相关记录的记录集。**frame_clause** 子句可以是如下的：

```
[ RANGE | ROWS ] frame_start
[ RANGE | ROWS ] BETWEEN frame_start AND frame_end
```

frame_start, **frame_end**

上面可以用如下的值：

```
UNBOUNDED PRECEDING
```

```
value PRECEDING
CURRENT ROW
value FOLLOWING
UNBOUNDED FOLLOWING
```

value

一个整型值。

注意

这里只介绍了 `CREATE CONTINUOUS VIEW` 语法，要想学习更多查询方面的知识，请参考。 PostgreSQL SELECT documentation

删除一个流视图

要想删除一个流视图，使用命令 `DROP CONTINUOUS VIEW` 。它的语法很简单：

```
DROP CONTINUOUS VIEW name
```

这条命令会移除流视图和相关的资源

清空流视图

如果想清空流视图的数据，并且不删除流视图，那么可以使用命令：`TRUNCATE CONTINUOUS VIEW`:

```
TRUNCATE CONTINUOUS VIEW name
```

这个命令会高效地移除所有的流视图的行，与 PostgreSQL 的 TRUNCATE 命令很类似。

查看流视图列表

如果想查看当前数据库系统中的所有流视图，你可以执行下面的查询：

```
SELECT * FROM pipeline_views();
```

别担心看不懂返回的所有列——他们大部分只是内部使用的。重要的列是 `name`,，这个列字段的值是你创建流视图的命名。字段 `query` 的值是这个流视图的查询定义。

数据查询

由于流视图与常规的视图非常类似，因此从流视图中查询数据也就是执行在这些视图上执行 select 语句的问题。

```
SELECT * FROM some_continuous_view
```

user	event_count
a	10
b	20
c	30

任意的作用于流视图上的有效的 `SELECT` 语句都可以让你对一直更新的内容进行分析。

```
SELECT t.name, sum(v.value) + sum(t.table_value) AS total
FROM some_continuous_view v JOIN some_table t ON v.id = t.id GROUP BY t.name
```

name	total
usman	10
jeff	20
derek	30

Time-to-Live (TTL) 过期

一种常见的使用 PipelineDB 模式是在聚合列中包含一个时间字段。另一种使用模式是移除不再需要的、基于时间字段的旧的数据（比如过期数据）。PipelineDB 内置了针对每行数据的 TTL 支持用于定位相关的模式。（译者注：就是对这些流数据打上时间戳，用于确定是否需要丢弃）

TTL 过期特征可以通过使用存储参数 ttl 和 ttl_column 来指定给流视图。自动运行的 vacuum 进程会删除任何值超过指定的 ttl 值的 `ttl_column` 值。下面是一个之前的样例，样例中会告知 autovacuumer 进程删除字段 minute 值超过一个月的记录：

```
CREATE CONTINUOUS VIEW v_ttl WITH (ttl = '1 month', ttl_column = 'minute') AS
  SELECT minute(arrival_timestamp), COUNT(*) FROM some_stream GROUP BY minute;
```

注意 TTL 行为对自动的 vacuum 进程优化器来说是一个提示（hint），因此不能保证记录过期后一定会被删除。

如果你想确保 TTL 过期的记录一定不能被读取，你应该在流视图基础上再创建一个视图，并且使用 where 语句来过滤掉过期的数据。

修改 TTL 参数值

通过 **set_ttl** 函数，可以增加、更新或者删除 TTL。

set_ttl (cv_name, ttl, ttl_column)

根据给定的参数更新指定的流视图的 TTL。 **ttl** 是一个间隔值，用字符串表示 (e.g. `'1 day'`)， **ttl_column** 是时间类型的列名。

如果 **ttl** 和 **ttl_column** 参数的值都设定为 NULL，那么会高效地从一个给定的流视图中移除 TTL 约束。注意 TTL 不能从一个滑动窗口的流视图中修改或删除。

流视图的激活和去激活

因为流视图会持续处理输入的流数据，如果能对进行流的启动和终止而不是通过关闭 PipelineDB 数据库的方式，那么是很有用的。例如，如果一个流视图遇到了超出预期的系统负载或者开始抛出大量异常，那么暂时性关闭流处理会很有帮助，直到问题得到解决。

流的视图控制是通过两个命令： `ACTIVATE` 和 `DEACTIVATE` 来完成的，含义等同于启动和停止流视图。当流视图处于激活状态时候，他们会从相关联的输入流中读取数据，并且会渐进地更新。相反地，非激活的流不能从相关联的输入流中读取数据，也不会更新流视图。. PipelineDB 在流视图不激活的情况下依然可以读取数据，只是这些数据不更新。

`ACTIVATE` 和 `DEACTIVATE` 语法命令很简单，不需要参数：

```
ACTIVATE | DEACTIVATE
```

重要

当流视图被去激活后，任何写入流中的事件都不会被读取，即便被再次激活，那么激活前的事件也不会被读取到了。

样例

让我们一起再看看一些流视图的使用样例，并看看这些流视图解决了什么问题。

重要

有一点很重要，那就是 PipelineDB 只保存流视图上执行 SELECT * 返回的数据(加上少量的元数据信息)。这是一个相对新的概念，但这也是让流视图如此强大的原因。强调上面的一点是，流视图只会永久存储一行数据在 PipelineDB 中(只有几个字节),即便它读取了万亿的事件数据。

```
CREATE CONTINUOUS VIEW avg_of_forever AS SELECT AVG(x) FROM one_trillion_events_stream
```

计算每天每个 url 链接被访问的独立用户数

这个查询每天只会花费常熟值的空间占用:

```
CREATE CONTINUOUS VIEW uniques AS
SELECT date_trunc('day', arrival_timestamp) AS day,
  referrer, COUNT(DISTINCT user_id)
FROM users_stream GROUP BY day, referrer;
```

计算每分钟每个流的数据点的线性分布:

```
CREATE CONTINUOUS VIEW lreg AS
SELECT date_trunc('minute', arrival_timestamp) AS minute,
  regr_slope(y, x) AS mx,
  regr_intercept(y, x) AS b
FROM datapoints_stream GROUP BY minute;
```

在过去 5 分钟里，我们服务的广告曝光了多少次？

```
CREATE CONTINUOUS VIEW imps AS
SELECT COUNT(*) FROM imps_stream
WHERE (arrival_timestamp > clock_timestamp() - interval '5 minutes');
```

我服务器的请求延迟在 90%，95%，99%有哪些？

```
CREATE CONTINUOUS VIEW latency AS
SELECT percentile_cont(array[90, 95, 99]) WITHIN GROUP (ORDER BY latency)
FROM latency_stream;
```

我有多少传感器位于 San Francisco 的 1000 米范围内？

```
-- PipelineDB ships natively with geospatial support
CREATE CONTINUOUS VIEW sf_proximity_count AS
SELECT COUNT(DISTINCT sensor_id)
```

```
FROM geo_stream WHERE ST_DWithin(

  -- Approximate SF coordinates
  ST_GeographyFromText('SRID=4326;POINT(37 -122)'), sensor_coords, 1000);
```

我们希望你能享受所有关于流视图的知识学习过程。下面，你应该学习流是怎样工作的内容了。

流转换

在 09.0 版本中新增该特性

流转换可以用于持续地把输入数据进行转换，并且不必存储数据。由于没有存储数据，流转换不支持聚合功能。转换的结果可以传输到其他流中或者输出到数据存储中。

创建流转换

下面是创建流转换的语法:

```
CREATE CONTINUOUS TRANSFORM name AS query THEN EXECUTE PROCEDURE function_name
( arguments )
```

query 是 PostgreSQL `SELECT` 查询语句的得到的数据集：

```
SELECT expression [ [ AS ] output_name ] [, ...]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
```

注意 `select` 语句里面的 `expression` 不能包含聚合函数，`from_item` 可以是如下语句：

```
    stream_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    table_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    from_item [ NATURAL ] join_type from_item [ ON join_condition ]
```

function_name：是一个用户提供的返回值类型为 trigger 的函数，流转换会对每一行的输出都执行这个函数。

arguments 是触发器被执行时提供给函数的用逗号分隔的值。参数只能是字符串常量。

注意

你可以把流转换看成触发器对每个输出的流数据进行函数处理。内部机制上，每个函数会以 `AFTER INSERT FOR EACH ROW`（插入数据后执行触发器）触发器形式执行，因此流转换输出的数据中不含旧的行和新的行。

删除流转换

要想从系统中删除流转换，使用命令 `DROP CONTINUOUS TRANSFORM` ，语法很简单：

```
DROP CONTINUOUS TRANSFORM name
```

这条命令会移除从系统中所有的流转换以及所有的相关资源。

查看流转换

想要查看系统中所有的流转换，执行下面的查询：

```
SELECT * FROM pipeline_transforms();
```

内置的转换触发器

当前 PipelineDB 只提供了一个内置的触发器函数 `pipeline_stream_insert` 可以用在流转换中。它会把流转换的输出插入到该函数参数为字符串的流中。例如：

```
CREATE CONTINUOUS TRANSFORM t AS
  SELECT x::int, y::int FROM stream WHERE mod(x, 2) = 0
  THEN EXECUTE PROCEDURE pipeline_stream_insert('even_stream');
```

这个流转换回插入所有的 x 值为奇数的 `(x, y)` 记录到流 `even_stream` 中。

重要

所有 `pipeline_stream_insert` 的参数必须是系统中已存在的流的有效名称，否则会抛出错误

创建自定义的触发器

你可以创建自定义的触发器函数，用于流转换。例如，如果你想把输出插入的一个表中，你可以这样做：

```
CREATE TABLE t (user text, value int);

CREATE OR REPLACE FUNCTION insert_into_t()
  RETURNS trigger AS
  $$
  BEGIN
    INSERT INTO t (user, value) VALUES (NEW.user, NEW.value);
    RETURN NEW;
  END;
  $$
  LANGUAGE plpgsql;

CREATE CONTINUOUS TRANSFORM ct AS
  SELECT user::text, value::int FROM stream WHERE value > 100
  THEN EXECUTE PROCEDURE insert_into_t();
```

流

流是允许客户端把数据通过流视图推送以便用户可以查看的一种通道的抽象。一行流数据或者简单地称之为 event 事件，非常像正常表的一行数据，写数据到流的接口也非常类似写数据到表中。然而，流在语义上和表是完全不同的。

就是说那些事件只“存在”于流中直到他们被所有的流视图消费掉。即便这样，用户也不能直接从流中查询数据，流数据只提供给流视图作为数据源输入。

流的创建语法和表的创建是很类似的：

```
CREATE STREAM stream_name ( [
       { column_name data_type [ COLLATE collation ] | LIKE parent_stream } [, ... ]
] )
```

stream_name

stream_name 是被创建的流的名字

column_name

在新的表结构中的列名（译者注：就是流的列名）

data_type

列的数据类型。这个列类型可以包含数组类型。关于 PipelineDB 所支持更多的数据类型，

参考内置函数 和 PostgreSQL 支持的数据类型

COLLATE collation

COLLATE 子句会指定一个排序规则给一个列（必须是一个可排序的数据类型）。如果没有指定列的数据排序规则是默认的。

LIKE parent_table [like_option ...]

`LIKE` 子句指定可以用于在创建一个新的流时候复制旧的流的所有字段名和数据类型。
可以通过 `ALTER STREAM` 命令给流添加字段

```
pipeline=# ALTER STREAM stream ADD COLUMN x integer;
ALTER STREAM
```

注意

不能 drop 掉流的列
流可以使用命令 `DROP STREAM` 进行删除。下面是一个读取流数据创建一个简单流视图的样例：

```
pipeline=# CREATE STREAM stream (x integer, y integer);
CREATE STREAM
pipeline=# CREATE CONTINUOUS VIEW v AS SELECT sum(x + y) FROM stream;
CREATE CONTINUOUS VIEW
```

往流里面写数据

插入数据

流写入使用 PostgreSQL 的 `INSERT` 语句。下面是使用语法：

让我们看一些样例

流写入是一个瞬时间写入的简单事件：

```
INSERT INTO stream (x, y, z) VALUES (0, 1, 2);

INSERT INTO json_stream (payload) VALUES (
  '{"key": "value", "arr": [92, 12, 100, 200], "obj": { "nested": "value" } }'
);
```

或者可以批量插入以便获取更好的性能:

```
INSERT INTO stream (x, y, z) VALUES (0, 1, 2), (3, 4, 5), (6, 7, 8)
```

```
(9, 10, 11), (12, 13, 14), (15, 16, 17), (18, 19, 20), (21, 22, 23), (24, 25, 26);
```

流插入可以包含自定义的表达式:

```
INSERT INTO geo_stream (id, coords) VALUES (42, ST_MakePoint(-72.09, 41.40));

INSERT INTO udf_stream (result) VALUES (my_user_defined_function('foo'));

INSERT INTO str_stream (encoded, location) VALUES
  (encode('encode me', 'base64'), position('needle' in 'haystack'));

INSERT INTO rad_stream (circle, sphere) VALUES
  (pi() * pow(11.2, 2), 4 / 3 * pi() * pow(11.2, 3));

-- Subselects into streams are also supported
INSERT INTO ss_stream (x) SELECT generate_series(1, 10) AS x;

INSERT INTO tab_stream (x) SELECT x FROM some_table;
```

预插入语句

为了减少网络负载，可以使用预插入的方法来插入数据：

```
PREPARE write_to_stream AS INSERT INTO stream (x, y, z) VALUES ($1, $2, $3);

EXECUTE write_to_stream(0, 1, 2);
EXECUTE write_to_stream(3, 4, 5);
EXECUTE write_to_stream(6, 7, 8);
```

COPY 语句

最后，同样可以使用 COPY 语句来把文件的数据插入到流中：

```
COPY stream (data) FROM '/some/file.csv'
```

`COPY` 语句对于逆向地把归档数据输入到流视图中是很有用的。下面是一个把数据从亚马逊 S3 归档数据导入到 PipelineDB 的样例:

```
s3cmd get s3://bucket/logfile.gz - | gunzip | pipeline -c "COPY stream (data) FROM STDIN"
```

其他的客户端

由于 PipelineDB 是和 PostgreSQL 兼容的，任何可以在 PostgreSQL 上运行的客户端都可以把数据写入到 PipelineDB 的流中。(甚至大多数可以在任何 SQL

数据库工作的客户端都可以在这里支持),因此不必手动创建流的插入。想要了解这些客户端是什么样的，可以参考章节 客户端。

流输出

在 0.9.5 版本开始支持

输出流 Output streams make it possible to read from the stream of incremental changes made to any continuous view. 输出流和 PipelineDB 的正常的流是类似的，因此也可以用于其他流视图或者流转换。流输出可以通过在一个流视图上调用 `output_of` 函数获得。

每一个输出流的行都会包含一个旧的和新的元组，用于表示对这个流视图进行操作后的变化。如果这个变化对应的是流的插入，那么旧的元组的值是 NULL。如果对应的是删除（目前唯一出现这种情况是在一个滑动窗口元组超过了一个窗口的范围），那么新的元组值为 NULL。

让我们看一个简单例子来讲解这里面的一些概念。考虑一个简单的流视图，这个流视图只是对一个列进行求和:

```
CREATE CONTINUOUS VIEW v_sum AS SELECT sum(x) FROM stream;
```

现在假设有这样的场景：我们想计算每次和的变化超过 10 的记录。我们可以创建另一个流，这个流从 `v_sum` 的输出流读取数据，我们很容易得到如下结果：

```
CREATE CONTINUOUS VIEW v_deltas AS SELECT abs((new).sum - (old).sum) AS delta
  FROM output_of('v_sum')
  WHERE abs((new).sum - (old).sum) > 10;
```

注意

old 和 **new** 的元组必须用小括号包起来。

滑动窗口上的输出流

对于非滑动窗口的流视图，输出的流会在任何时候由于流的写入引起的变化下，简单地写入到流视图的结果中。然而由于滑动窗口的流视图的结果也依赖于时间，他们的流输出也自动写入并随着时间变化而变化。也就是说滑动窗口的流视图的流输出即便在没有流事件的输入情况下也会更新。

stream_targets 函数

有时候你可能想更新流视图相关联的流的查询数据，比如当你想把一些历史数据插入到新创建的流视图中。你可以使用 `stream_targets` 参数配置来指定要进行数据更新的流视图。你可以把设置参数 `stream_targets` 的值设置目标流，这样就可以可以插入数据到目标流中。

```
pipeline=# CREATE CONTINUOUS VIEW v0 AS SELECT COUNT(*) FROM stream;
CREATE CONTINUOUS VIEW
pipeline=# CREATE CONTINUOUS VIEW v1 AS SELECT COUNT(*) FROM stream;
CREATE CONTINUOUS VIEW
pipeline=# INSERT INTO stream (x) VALUES (1);
INSERT 0 1
pipeline=# SET stream_targets TO v0;
SET
pipeline=# INSERT INTO stream (x) VALUES (1);
INSERT 0 1
pipeline=# SET stream_targets TO DEFAULT;
SET
pipeline=# INSERT INTO stream (x) VALUES (1);
INSERT 0 1
pipeline=# SELECT count FROM v0;
 count
-------
     3
(1 row)

pipeline=# SELECT count FROM v1;
 count
-------
     2
(1 row)

pipeline=#
```

到达顺序

设计上 PipelineDB 使用事件的到达顺序来标识事件的顺序。这话的意思是事件在到达 PipelineDB 服务器时候会被打上时间戳，也就是会给出系统内置的额外属性称之为 `arrival_timestamp` 的时间戳。 `arrival_timestamp` 可以用于流视图的时间组件，比如滑动窗口。

事件过期

每个事件到达 PipelineDB 数据库服务后，时间会被附上一个占空间很小的位图数据结构，这个值表示所有的流视图需要读取该事件。当一个流视图读取了该事件后，当一个流视图读取了这个事件后，它就在这个位图上取反一个 bit 位即设置为 1。当 bitmap 的所有的 bit 位值都设置为 1 后，这个事件就被丢弃了，不会再被读取到。

现在你已经知道了流视图是什么以及如何写入数据到流中，现在是适合来学习 PipelineDB 大量的内置特性了。

内置的功能特性

普通函数

我们努力确保 PipelineDB 保持与 PostgreSQL 9.5 完全兼容。因此所有 PostgreSQL 的内置功能函数 对 PipelineDB 用户来说都是适用的。

除了对 PostgreSQL 9.5 兼容支持外，PipelineDB 也原生兼容支持 PostGIS 2.1，查看所有的 PostGIS 内置函数， 这些函数对 PipelineDB 用户也是可用的。

聚合函数

PipelineDB 的一个基本的设计目标是追求高性能的流聚合功能。PostgreSQL 和 PostGIS 的聚合函数完全适用于流视图 (可能会有极个别稀有的异常)。除了大范围兼容标准聚合函数外， PipelineDB 还增加了一些独有的聚合函数，以便用于流数据的处理。

参考流聚合获取更多 PipelineDB 最有用的特性。

PipelineDB 特有的数据类型

PipelineDB 为了在流上高效使用支持概率性的数据结构和算法，它使用了许多原生的数据类型。你可能在创建表的时候永远都不会使用这些类型，但是他们往往是 PipelineDB 专有的聚合算子得到的结果，因此他们往往由流视图自己自动创建。 这些类型如下：

bloom
: Bloom Filter

cmsketch
: Count-Min Sketch

fss
: Filtered-Space Saving Top-K

hll
: HyperLogLog

tdigest
: T-Digest

PipelineDB 独有的函数

PipelineDB 设计了一系列函数与上面独有的数据类型交互使用。这些函数如下：

bloom_add (bloom, expression)

把给定的表达式添加给数据结构 Bloom Filter.

bloom_cardinality (bloom)

返回 Bloom Filter 数据结构的基数（译者注：基数就是集合中去重的元素个数），这个就是我们添加到 Bloom filter 数据结构的去重元素个数，这个函数返回可能会有一点点误差。

bloom_contains (bloom, expression)

如果 Bloom filter 可能包含输入的参数值，那么就返回 true，会有一点点正负误差。译者注：Bloom Filter 之所以是概率性的是，是因为得到的结果并非完全确定，通过概率性算法在大量的数据处理中可以很快，但不能保证完全确定。

bloom_intersection (bloom, bloom, ...)

根据给定的 Bloom Filter 结构，返回其交集。

bloom_union (bloom, bloom, ...)

返回给定的 Bloom Filter 数据结构的并集。

date_round (timestamp, resolution)

获取离 resolution 格式最近的时间范围区间，这个对聚合运算很有用。例如想计算过去 10 分钟范围内的事件总数。

```
CREATE CONTINUOUS VIEW v AS SELECT
  date_round(arrival_timestam, '10 minutes') AS bucket_10m, COUNT(*) FROM stream
  GROUP BY bucket_10m;
```

fss_increment (fss, expression)

根据给定的 fss 和 express 参数,提高至频率为 fss 值，返回 Filtered-Space Saving Top-K.

fss_increment_weighted (fss, expression, weight)

根据给定的 Filtered-Space Saving Top-K 按照给定的权重比例，提高 expression 对应值的比例，返回 Filtered-Space Saving Top-K 类型对象。

fss_topk (fss)

返回 k 个元组，这些元组表示 Filtered-Space Saving Top-K 结构里面 top-k 个元素值和他们出现的频率。

fss_topk_freqs (fss)

返回 Filtered-Space Saving Top-K 中有最多 k 个的值。

fss_topk_values (fss)

返回 Filtered-Space Saving Top-K 中 top k 个值。

cmsketch_add (cmsketch, expression, weight)

根据给定的 cmsketch，提高 expression 在 cmsketch 的权重为 weight。

cmsketch_frequency (cmsketch, expression)

根据给定的 cmsketch 数据结构变量，返回 expression 表达式添加到该数据结构的次数，会有一点点误差。

cmsketch_norm_frequency (cmsketch, expression)

根据给定的 Count-Min Sketch 数据结构，返回 expression 的正则频率值，会有一点点误差。

cmsketch_total (cmsketch)

根据给定 cmsketch，返回添加到数据结构 Count-Min Sketch 的总数。

hll_add (hll, expression)

把给定的 expression 添加到 HyperLogLog 数据结构中。

hll_cardinality (hll)

根据给定 HyperLogLog 对象，返回其基数，会有大约 0.2% 的误差。

hll_union (hll, hll, ...)

根据给定的 HyperLogLog 返回其并集。

set_cardinality (array)

根据给定的集合数组，返回其基数，集合可以用函数 **set_agg** 来创建。

tdigest_add (tdigest, expression, weight)

根据给定 expression，在 T-Digest 数据结构对象中按照 weight 值的步长增加频率。

tdigest_cdf (tdigest, expression)

根据给定的 tdigest 参数，返回累计分布函数在 expression 上的值，会有很小的误差。

tdigest_quantile (tdigest, float)

给定 tdigest,返回给定分位点的值。Float 值的范围在 `[0, 1]` 内。

注意

请查看 PipelineDB 专有的聚合算子,会说明这些类型是如何创建的。

其他函数

pipeline_version ( )

返回一个字符串，表示你安装的 PipelineDB 安装版本的信息。

pipeline_views ( )

返回所有流视图的集合

pipeline_transforms ( )

返回所有流转换的集合

流聚合

PipelineDB 的一个基本的目标是追求高性能的流聚合，因此不要对聚合功能是 PipelineDB 基础功能感到惊讶。流聚合在大多数情况下是非常高效的，它可以让推送给它的数据的量以常量方式保存在 PipelineDB 中。这可以在普通的硬件上进行持续高吞吐量的运算。

流聚合是实时渐进变化的，因为新的事件会不断地被读取。例如对于聚合算子 count 和 sum，我们很容易看到他们的查询结果是在渐进变化的——因为不断有新的值添加到已有的计算结果中。

但是对于更复杂的聚合比如 avg（求平均）, stddev（方差）, percentile_cont 等等。更多高级特性需要支持更高效的渐进变化， PipelineDB 可以帮你处理这些复杂的聚合.

下面你会找到所有 PipelineDB 所支持的聚合函数。其中一些函数可能和同类的函数有轻微的不同，这主要是为了高效的操作无限的数据流。这类的聚合函数会有注释有哪些轻微的不同。

注意

关于 PostgreSQL 和 PostGIS 等价的聚合算子，你可以查看 PostgreSQL aggregates 或 PostGIS aggregates 文档

PipelineDB 独有的聚合函数

bloom_agg (expression)

把所有的输入值添加到数据结构 Bloom Filter 中

bloom_agg (expression , p , n)

把所有的输入值添加的一个 Bloom 过滤器中，根据输入参数计算大小。 **p** 是正负率， **n** 是要添加的唯一元素的期望值。

bloom_union_agg (bloom filter)

把输入的 Bloom filter 合并成一个单个的 Bloom filter，新的 Bloom Filter 包含所有输入的

bloom filter 的信息。

bloom_intersection_agg (bloom filter)

输入是多个 Bloom filter 对象，返回的是这些 bloom filter 对象信息的交集。

cmsketch_agg (expression)

把所有的输入值添加到 Count-Min Sketch 中。

cmsketch_agg (expression, epsilon, p)

和上面的类似，但是接受 **epsilon** 和 **p** 作为底层的 cmsketch 参数，**epsilon** 决定了 cmsketch 可接受的错误率默认是 0.2%。**p** 决定了置信度，默认是 **0.995** (99.5%)。更低的 **epsilon** 和 **p** 会得到更小的 **cmsketch 数据结构**，反之亦然。

cmsketch_merge_agg (count-min sketch)

把所有的 Count-min sketche 数据结构对象合并到单个的包含所有信息的 Count-min sketch 结构中。

exact_count_distinct (expression)

根据输入的 expression 计算精确的去重个数。由于流视图为了高效计算会使用 HyperLogLog ， **exact_count_distinct** 可以在不允许使用 HyperLogLog 情况下，容忍很小误差下使用。

重要

exact_count_distinct 为了确保唯一性必须要存储所有观察到的唯一值，因此不建议在唯一值太多的情况下使用。

fss_agg (expression , k)

把所有的输入值添加到一个 Filtered-Space_Saving_Top-K 数据结构中，数据元素个数为 k 个，每次插入都增加 1。

fss_agg_weighted (expression, k, weight)

把所有的输入值都添加到 FSS 结构中，输入值个数为 k 个，每次按照权重 weight 值增加。

fss_merge_agg (fss)

把所有的 FSS 输入合并到单个 FSS 中

keyed_max (key, value)

返回与 value 最接近的最大的 key

keyed_min (key, value)

返回与 value 最接近的最小的 key

hll_agg (expression)

把所有的输入值添加到 HyperLogLog 中

hll_agg (expression, p)

根据置信度 p，添加所有的输入值到数据结构 HyperLogLog 中，更大的 p 值会减少 HyperLogLog 的错误率，但会消耗更多空间占用。

hll_union_agg (hyperloglog)

把输入的 HyperLogLog 合并成一个单个的 HyperLogLog，新的 HyperLogLog 包含所有输入的 HyperLogLog 的信息。

set_agg (expression)

把所有的输入值添加到一个 set 集合中

tdigest_agg (expression)

把所有的输入添加到 T-Digest 结构中

tidgest_merge_agg (tdigest)

把输入的**tdigest**合并成一个单个的**tdigest**，新的**tdigest**包含所有输入的**tdigest**的信息。

first_values (n) WITHIN GROUP (ORDER BY sort_expression)

有个有序集合的聚合函数，根据提供的排序表达式返回前 n 的值。

注意

参考 PipelineDB 专有的函数,详细说明了 PipelineDB 的用于操作 Bloom filters, Count-min sketches, HyperLogLogs 和 T-Digests 的非聚合函数。同时，请查看概率性的数据结构和算法 查看这些数据结构和函数是什么以及你怎样使用他们。

合并结果

由于 PipelineDB 能够不断更新聚合结果，它可以合并更多结果而不是他们当前的值。例如，合并多个平均值并非只是计算平均值的平均值。他们的权重也必须考虑进去。

对于这种类型的操作， PipelineDB 提供了专有的合并结果的函数，具体描述如下：

combine (aggregate column)

根据要聚合的列，把所有的值聚合成一个，就好像所有单次聚合的输入被一次聚合成功了。

注意

combine 只对流视图的被聚合的列其作用。
我们看个样例：

```
pipeline=# CREATE CONTINUOUS VIEW v AS
     SELECT g::integer, AVG(x::integer) FROM stream GROUP BY g;
CREATE CONTINUOUS VIEW
pipeline=# INSERT INTO stream (g, x) VALUES (0, 10), (0, 10), (0, 10), (0, 10), (0, 10);
INSERT 0 5
pipeline=# INSERT INTO stream (g, x) VALUES (1, 20);
INSERT 0 1
pipeline=# SELECT * FROM v;
 g |        avg
---+--------------------
 0 | 10.0000000000000000
 1 | 20.0000000000000000
(2 rows)

pipeline=# SELECT avg(avg) FROM v;
        avg
--------------------
 15.0000000000000000
(1 row)

pipeline=# -- But that didn't take into account that the value of 10 weighs much more,
pipeline=# -- because it was inserted 5 times, whereas 20 was only inserted once.
pipeline=# -- combine() will take this weight into account
pipeline=#
```

```
pipeline=# SELECT combine(avg) FROM v;
       combine
---------------------
 11.6666666666666667
(1 row)

pipeline=# -- There we go! This is the same average we would have gotten if we ran
pipeline=# -- a single average on all 6 of the above inserted values, yet we only
pipeline=# -- needed two rows to do it.
```

创建聚合函数

除了 PipelineDB 内置的聚合函数外，用户可以为流视图来自定义聚合函数。用户自定义的合并函数可以使用 PostgreSQL 的 CREATE AGGREGATE 命令。为了想创建一个可合并的聚合函数，必须要提供 **combinefunc** 参数。**combineinfunc** 和 **transoutfunc** 都是可选的：

```
CREATE AGGREGATE name ( [ argmode ] [ argname ] arg_data_type [ , ... ] ) (
        ...
        COMBINEFUNC = combinefunc,
        [ , COMBINEINFUNC = combineinfunc ]
        [ , TRANSOUTFUNC = transoutfunc ]
)
```

combinefunc (stype, stype)

这个函数需要两个中间状态参数并返回一个状态。例如，下面是一个返回值为 integer 类型的 avg 的合并函数实现。

```
CREATE FUNCTION avg_combine(state integer[], incoming integer[]) RETURNS integer[] AS
$$
BEGIN
        RETURN ARRAY[state[1] + incoming[1], state[2] + incoming[2]];
END;
$$
LANGUAGE plpgsql
```

中间状态参数表示 2 个元素的数组，这两个元素是元素的个数和他们的和，可以用来计算最终结果。

combineinfunc (any)

一个把中间状态从外部的反解析成内部格式表示的函数。反解析只在中间状态不是基础类型状态的情况下是必要的。

transoutfunc (stype)

一个把聚合函数的中间状态从内部序列化成外部表示的函数，以便这个外部状态可以存储在表中。序列化只有在红军状态不是原生基础类型下才是必要的。

普通聚合函数

array_agg (expression)

把输入值，包括 null 连接成一个数组

avg (expression)

计算输入值的平均值

bit_and (expression)

对非 null 的输入值进行按位做 AND 计算，如果没有输入值，则结果为 null。

bit_or (expression)

对非 null 的输入值进行按位做 OR 计算，如果没有输入值，则结果为 null。

bool_and (expression)

如果输入值是 true，那么返回结果 true，否则返回 false。

bool_or (expression)

如果输入的值至少有一个是 true，那么返回的就是 true，否则是 false。

count (*)

计算输入行的记录数

count (DISTINCT expression)

计算输入的去重记录数

注意

计算无限的流的去重个数需要无限的内存，因此流视图使用 HyperLogLog 数据结构在恒定的空间和时间范围完成这个事情，并且误差很小。经验表明的 HyperLogLog 实现大概有 0.81%的误差。例如 **count distinct** 计算结果是 1008，但实际上去重可能只有 1000.

count (expression)

计算 expression 非 null 的行数。

every (expression)

等价于 **bool_and 函数**

json_agg (expression)

聚合成一个 json 数组。

json_object_agg (key, value)

聚合一个 **key-value** 键值对为一个 JSON 数组。

jsonb_agg (expression)

聚合成一个 JSONB 数组

jsonb_object_agg (key, value)

聚合 **key-value** 键值对为一个 JSONB 数组

max (expression)

计算表达式的最大值

min (expression)

计算所有输入值的最小值。

string_agg (expression, delimiter)

把输入的字符串，按照 **delimiter 分隔符进行分隔。**

sum (expression)

计算 **expression** 表达式的和

统计聚合函数

corr (y, x)

相关系数
covar_pop (y, x)

总体协方差
covar_samp (y, x)

样本方差
regr_avgx (y, x)

所有自变量的平均值 `(sum(x)/N)`
regr_avgy (y, x)

所有自变量的平均值 `(sum(y)/N)`
regr_count (y, x)

计算所有输入行 y 和 x 都不为空的记录数。
regr_intercept (y, x)

根据(x,y)点按照最小二乘法计算得到的 y 轴截距。
regr_r2 (y, x)

相关系数的平方
regr_slope (y, x)

根据(x,y)点按照最小二乘法计算得到的斜率。

regr_sxx (y, x)

`sum(X^2) - sum(X)^2/N` — 自变量平方的和
regr_sxy (y, x)

`sum(X*Y) - sum(X) * sum(Y)/N` — 独立时间变量乘积的和
regr_syy (y, x)

`sum(Y^2) - sum(Y)^2/N` —独立自变量的和

stddev (expression)

输入值的标准差

stddev_pop (expression)

输入值的总体协方标准差

variance (expression)

输入值的样本方差（输入值的样本差的平方）

var_pop (expression)

输入值的总体协方差(总体标准差的平方)

有序集合的聚合

有序集合 的聚合为了获取结构，会把顺序加到输入数据上，他们使用 `WITHIN GROUP` 子句，语法如下：

```
aggregate_name ( [ expression [ , ... ] ] ) WITHIN GROUP ( order_by_clause )
```

让我们看看几个例子：

计算 99%占比的值：

```
SELECT percentile_cont(0.99) WITHIN GROUP (ORDER BY value) FROM some_table;
```

或者用流视图：

```
CREATE CONTINUOUS VIEW percentile AS
SELECT percentile_cont(0.99) WITHIN GROUP (ORDER BY value::float8)
FROM some_stream;
```

percentile_cont (fraction)

连续百分位数：返回对应于排序中指定分数的值，如果需要，在相邻输入项之间进行插值

percentile_cont (array of fractions)

多重连续百分位数：返回匹配分数参数形状的结果数组，每个非空元素被对应于该值的百分值替换。

注意

计算无限流的百分位数需要大量的内存，所以对于 percentile_cont 的两种形式，当通过流视图使用时，使用 t-digest 来估计分位点是一种具有很高精度的方法。一般来说，流视图的分位点约束靠近上限或是下限（上下限范围是[0，1))时候，就越精确

Hypothetical-set 聚合函数

hypothetical-set 聚合函数需要一个表达式作为参数，然后根据一些记录集计算出一些东西。例如 **rank(2)** 计算 2 的秩，不论输入的行记录是什么。

hypothetical-set 聚合算子使用 `WITHIN GROUP` 子句定义输入的行，语法如下：

```
aggregate_name ( [ expression [ , ... ] ] ) WITHIN GROUP ( order_by_clause )
```

下面是一个流视图使用的 hypothetical-set 聚合的样例

```
CREATE CONTINUOUS VIEW continuous_rank AS
SELECT rank(42) WITHIN GROUP (ORDER BY value::float8)
FROM some_stream;
```

这流视图会根据读取的事件，不断更新 42 的秩。

rank (arguments)

假定行的秩，对于重复记录有 gap 间隙。

dense_rank (arguments)

假设行的秩，没有统计间隙

注意

根据一个不断的流，计算一个值的假设 dense_rank 值，这需要无限大的内存，所以流视图使用 HyperLogLog 在恒定的时间和空间来解决，误差会很小。根据经验，PipelineDB 的 Hyperloglog 实现的误差率为 0.2%。换句话说，dense_rank（1000）在一个连续的视图中可能为 998，而实际的值为 1000

percent_rank (arguments)

假定行的相对秩，值范围从 0 到 1（译者注：秩是线性代数中的矩阵概念）

cume_dist (arguments)

假定行的相对秩，秩的值范围是 1/N 到 1

暂不支持的聚合

jsonb_agg (any)

很快就会支持，现在请暂时使用函数 `json_agg` 吧

jsonb_object_agg (any, any)

很快就支持，现在请使用函数 `json_object_agg`.

mode ( )

未来版本的 PipelineDB 会包含一种在线模式的估计算法，现在暂时不支持

percentile_disc (arguments)

给定一个输入的百分数(比如 0.99)，**percentile_disc** 返回的是两个离得最近值的除法值。（译者注：如果读者不明白这两个函数，可以搜索一下，目前 oracle 支持这个函数）。这实际上需要有序的输入数据集，对于无限的流来说显然是不合实际的，甚至对于高精度的估算函数 **percentile_cont** 也不允许。

xmlagg (xml)

:(（译者注：官方文档即是这个标签，就是目前占时不支持的意思）

aggregate_name (DISTINCT expression)

只有 `count` 聚合函数支持 `DISTINCT` 关键字，参见一般聚合章节。在未来的版本规划中，我们可能让 Bloom Filter 支持所有的聚合函数。

客户端编程语言支持

由于 PipelineDB 完全兼容 PostgreSQL 9.5,所以也没有自己的客户端库。相反，任何可以在 PostgreSQL (或者就任何 SQL 数据库而言)上运行的，都适用于 PipelineDB。

下面你可以看到几种不同编程语言编写的应用样例，应用样例中只会创建流视图：

```
CREATE CONTINUOUS VIEW continuous view AS
     SELECT x::integer, COUNT(*) FROM stream GROUP BY x;
```

这个应用会发送 `100,000` 个事件也就是 10 个按 x 聚合后的组记录到流视图中，并打印输出结果。

Python

下面的 Python 样例中，你需要安装 psycopg2 。

```
import psycopg2

conn = psycopg2.connect('dbname=test user=user host=localhost port=5432')
pipeline = conn.cursor()

create_cv = """
CREATE CONTINUOUS VIEW continuous_view AS SELECT x::integer, COUNT(*) FROM stream GROUP
BY x
"""
pipeline.execute(create_cv)
conn.commit()

rows = []

for n in range(100000):
    # 10 unique groupings
    x = n % 10
    rows.append({'x': x})

# Now write the rows to the stream
pipeline.executemany('INSERT INTO stream (x) VALUES (%(x)s)', rows)

# Now read the results
pipeline.execute('SELECT * FROM continuous_view')
rows = pipeline.fetchall()

for row in rows:
    x, count = row

    print x, count

pipeline.execute('DROP CONTINUOUS VIEW continuous_view')
pipeline.close()
```

Ruby

这个 Ruby 案例使用 pg gem.

```ruby
require 'pg'
pipeline = PGconn.connect("dbname='test' user='user' host='localhost' port=5432")

# This continuous view will perform 3 aggregations on page view traffic, grouped by url:
#
# total_count - count the number of total page views for each url
# uniques     - count the number of unique users for each url
# p99_latency - determine the 99th-percentile latency for each url

q = "
CREATE CONTINUOUS VIEW v AS
SELECT
  url::text,
  count(*) AS total_count,
  count(DISTINCT cookie::text) AS uniques,
  percentile_cont(0.99) WITHIN GROUP (ORDER BY latency::integer) AS p99_latency
FROM page_views GROUP BY url"

pipeline.exec(q)

for n in 1..10000 do
  # 10 unique urls
  url = '/some/url/%d' % (n % 10)

  # 1000 unique cookies
  cookie = '%032d' % (n % 1000)

  # latency uniformly distributed between 1 and 100
  latency = rand(101)

  # NOTE: it would be much faster to batch these into a single INSERT
  # statement, but for simplicity's sake let's do one at a time
  pipeline.exec(
  "INSERT INTO page_views (url, cookie, latency) VALUES ('%s', '%s', %d)"
       % [url, cookie, latency])
end
```

```
# The output of a continuous view can be queried like any other table or view
rows = pipeline.exec('SELECT * FROM v ORDER BY url')

rows.each do |row|
  puts row
end

# Clean up
pipeline.exec('DROP CONTINUOUS VIEW v')
```

Java

这个样例中，你需要安装 JDBC 支持，并且配置 `CLASSPATH`.

```
import java.util.Properties;
import java.sql.*;

public class Example {

  static final String HOST = "localhost";
  static final String DATABASE = "test";
  static final String USER = "user";

  public static void main(String[] args) throws SQLException {

    // Connect to "test" database on port 5432
    String url = "jdbc:postgresql://" + HOST + ":5432/" + DATABASE;
    ResultSet  rs;
    Properties props = new Properties();

    props.setProperty("user", USER);
    Connection conn = DriverManager.getConnection(url, props);

    Statement stmt = conn.createStatement();
    stmt.executeUpdate(
      "CREATE CONTINUOUS VIEW v AS SELECT x::integer, COUNT(*) FROM stream GROUP BY x");

    for (int i=0; i<100000; i++)
    {
      // 10 unique groupings
      int x = i % 10;
```

```
      // INSERT INTO stream (x) VALUES (x)
      stmt.addBatch("INSERT INTO stream (x) VALUES (" + Integer.toString(x) + ")");
    }

    stmt.executeBatch();

    rs = stmt.executeQuery("SELECT * FROM v");
    while (rs.next())
    {
      int id = rs.getInt("x");
      int count = rs.getInt("count");

      System.out.println(id + " = " + count);
    }

    // Clean up
    stmt.executeUpdate("DROP CONTINUOUS VIEW v");
    conn.close();
  }
}
```

概率数据结构和算法

译者注：这样的数据结构和概率性算法在很多应用中都有，比如 elastic search，这种算法的好处是快速计算，不足之处是在一定数据量时候计算结果是估算的，并不精确。

PipelineDB 打包了一些大部分数据库系统用户没有见过的数据类型和聚合函数，但是这些却对流数据处理非常有用。在这里你会了解到这些类型和函数的具体情况，以及你可以用它们来做什么。

Bloom Filter 数据结构

Bloom filter 过滤器是一种空间优化的数据结构，设计用来估算集合的基数，以及是否包含一个元素，具有很高的可信度。Bloom Filter 过滤器的工作原理是通过把一个被添加的元素映射到位图中的一个或多个位上。当一个元素被添加到 Bloom Filter 过滤器后，这些位（理想情况下只是一个 bit 位）设置为 1。

直观地说,这意味着一个 n 位的输入可以被压缩到一个 bit 位上。虽然使用 Bloom 过滤器可以节省大量的空间，代价是当确定是否有一个给定的元素已被添加到 Bloom 过滤器中时候会有较小的误差，因为单个输入可能映射到多个位上面

怎样在 PipelineDB 使用 Bloom filters？

流视图包含了 `SELECT DISTINCT (...)` 语句来使用 Bloom filters 来决定一个给定的 expression 表达式是否是唯一的，以便确定是否将其计算在流式计算的结果内。这让流视图可以用一个固定的空间内来决定一个无限不断的流中的记录的 expression 是否唯一，并且精度非常高。

用户同样可以自己构建他们的 Bloom filter 算法，并用 PipelineDB 自定义函数类似的方式来使用。

Count-Min Sketch 数据结构

Count-min sketch 数据结构类似于 Bloom filter 过滤器,主要区别是 Count-min sketch 估算的是每个被添加进去的元素的个数，而 Bloom filter 只记录一个给定的元素是否可能已经被添加进去了。目前 PipelineDB 没有功能内部使用 Count-min sketch 这种结构，虽然用户可以自定义他们的 Count-min sketch 数据结构，并使用。

Filtered-Space Saving Top-K

Filtered-Space Saving (FSS)是一种把数据结构和算法结合起来很有用的工具，它可以精确估计出流里面出现次数最多的 top k 个值，并且会使用一个很小的常量内存。一个简单的计算办法是 维护数据流里面的值的一个表以及他们出现的次数，这种想法对于流来说并不实际。相反，FSS 把输入的值做 hash 运算到 bucket 桶里面，每个桶里面包含了一个已经添加值的集合。如果输入的元素已经存在于一个给定的桶里面，那么它的频率就增加一次。如果这个元素不存在，那么只要满足一些确定的条件，就可以添加到桶里面。目前没有 PipelineDB 功能隐式使用 FSS。FSS 类型和它相关的函数可以通过 PipelineDB 专有的函数和聚合算子来访问。

HyperLogLog

HyperLogLog 是数据结构和算法的结合，类似于 Bloom 过滤器，是用来估计集合的基数的，具有很高的精度。在功能方面，HyperLogLog 只支持添加元素和估计已添加元素的基数。它们不支持像 Bloom 过滤器这样可以检查指定元素的是否存在。然而，他们的空间效率比 Bloom 过滤器更强。

HyperLogLog 通过对其输入流的元素做进一步划分并存储每个子部分的前导零的最大数量。由于元素在检查前导零的个数前是均匀散列的，一般的想法是：前导零的个数越多，许多唯一的元素被添加的可能性就越高。根据经验，这个估计是非常准确的 - PipelineDB 实现的 HyperLogLog 结构和算法误差只有约 0.81%（约 8 /1000）。

怎样在 PipelineDB 中使用 HyperLogLog ？

流视图包含了 `COUNT DISTINCT (...)` 语句来使用 **HyperLogLog** 来来估计去重后的 expression 的总数，这让流视图使用的是一个固定空间。hypothetical-set 聚合函数, **dense_rank** 也使用 HyperLogLog 来精确估算唯一的低秩表达式的个数。

用户同样可以自己构建他们的 HyperLogLog 算法，并用 PipelineDB 自定义函数类似的方式来使用。

T-Digest

T - Digest 是一种数据结构，支持基于排序统计精确估计，如百分位数（分位点）和中位数，并且只有使用固定数量的空间。以一个小的误差代价换得高效的空间效率，使得 t-digest 适合排序统计为基础的流的计算，这在很多情况下需要他们的输入是有限的并且有序数据以便得到完美的精度。t-digest 本质上是一个自适应直方图，智能调整桶和频率以便让更多的元素添加到它。

怎样在 PipelineDB 中使用 T–Digest ？

percentile_cont 算子内部使用 t-digest 操作一个流。

用户同样可以自己构建他们的 T-Digest 算法，并用 PipelineDB 自定义函数类似的方式来使用。

滑动窗口函数

由于流视图是一直不断的更新的（因为流视图关联的流数据源在不断流入新数据），PipelineDB 在更新流视图的结果时候能够考虑到当前的时间。SQL 查询中带有与当前时间相关的 where 子句称作滑动窗口查询。Where 子句过滤或接受的事件集合是不断随着时间变化的。

滑动关键字 where 子句包含了两个重要部分：

clock_timestamp ( )

一个内置的函数，返回当前的时间戳

arrival_timestamp

所有输入的事件都包含了一个特殊的属性：就是这些事件的到达时间，在事件的到达顺序那一节有描述。然而，不必显式地在 where 子句中引用这些值。PipelineDB 内部已经做了，只需要在流视图定义时候指定一个 sw 存储参数即可。

这些概念可能最好通过例子说明。

样例

虽然滑动窗口对于 SQL 数据库来说是一个新概念，但 PipelineDB 不使用任何新的窗口语法，相反 PipelineDB 使用标准的 PostgreSQL 9.5 语法。下面是一个简单的例子：

过去一分钟我看了那些用户？

```
CREATE CONTINUOUS VIEW recent_users WITH (sw = '1 minute') AS
   SELECT user_id::integer FROM stream;
```

内部上 PipelineDB 会重写这个查询如下：

```
CREATE CONTINUOUS VIEW recent_users AS
   SELECT user_id::integer FROM stream
WHERE (arrival_timestamp > clock_timestamp() - interval '1 minute');
```

注意

PipelineDB 允许用户在创建滑动窗口流视图的时候手动构建一个WHERE 子句，虽然这是不推荐的，因为参数 `sw` 就是为了避免用户再编写冗长的语句。
上面的这个流视图的 SELECT 语句只返回包含在过去一分钟内的用户。因此重复查询 SELECT 语句会返回不同的行，即便流视图没有显式地更新。

让我们一起来看看上面 SQL 代码的分解：
the`(arrival_timestamp > clock_timestamp() - interval '1 minute')`

每次都会执行 `clock_timestamp() - interval '1 minute'` ，它会返回一个在过去 1 分钟内的时间戳。添加 `arrival_timestamp` 和 `>` 意味着这个语句
在 `arrival_timestamp` 的值如果在过去一分钟内，那么将返回 true。由于这个 SQL 谓词会在每个新时间到达读取后都执行一次，这就会非常高效地给我们过去一分钟范围内的滑动窗口。

注意

PipelineDB 给出了可以在查询中使用的 `current_date`, `current_time`,
和 `current_timestamp` 三个内置的值，但是设计上这些并不能在滑动窗口中使用，因为在一次事务查询中他们是一个不变的常量，因此不能及时反映每条事件所在的当前时刻。

Sliding 聚合

滑动窗口查询同样也支持聚合函数。滑动聚合函数能够尽可能聚合更多的输入，但是为了保证处理间隔，还需要知道怎样在时间推移过程中，从窗口中移除已经处理过的。这样的部分聚合对用户来说是透明的（隐式处理）——只有滑动窗口完全聚合后的结果才对用户可见。

我们看一些样例：

过去一分钟我看了多少用户？

```
CREATE CONTINUOUS VIEW count_recent_users WITH (sw = '1 minute') AS
   SELECT COUNT(*) FROM stream;
```

每次在流视图上执行 SELECT 语句时候，就会返回过去一分钟内有多少个事件。例如，如果事件停止进入，上面的 count 结果每次查询后都不断减少。这样的现象对于 PipelineDB 所支持的所有流视图聚合函数都是一样的。

过去 5 分钟内，我的传感器的移动平均温度是多少？

```
CREATE CONTINUOUS VIEW sensor_temps WITH (sw = '5 minutes') AS
   SELECT sensor::integer, AVG(temp::numeric) FROM sensor_stream
GROUP BY sensor;
```

过去 30 天我们有多少独立用户数？

```
CREATE CONTINUOUS VIEW uniques WITH (sw = '30 days') AS
   SELECT COUNT(DISTINCT user::integer) FROM user_stream;
```

过去 5 分钟内，我的服务器 99%的响应时间延迟是多少？

```
CREATE CONTINUOUS VIEW latency WITH (sw = '5 minutes') AS
   SELECT server_id::integer, percentile_cont(0.99)
   WITHIN GROUP (ORDER BY latency::numeric) FROM server_stream
GROUP BY server_id;
```

临时失效

显然，流视图里的滑动窗口行记录在一定时间后就会失效了。这样的记录因此需要做垃圾回收。垃圾回收有两种方式：

通过后台进程失效

PipelineDB 有一个和 PostgreSQL 类似的自动 vacuum 进程在周期性地运行，并且物理删除掉流视图中的滑动窗口里的过期记录。

事件实时失效

当用 select 语句查询流视图后，任何被读取后的数据在计算生成结果的时候立即被抛弃掉，这确保了即便数据依然存在也是无效的，他们不会包含在任何查询结果内。

多窗口使用

对于相同查询但时间窗口范围不同是很常见的场景。例如，我们想跟踪过去 5 分钟，10 分钟，1 天等等的用户事件总数。对于这样的需求，PipelineDB 支持可以在单个时间滑动窗口的流视图上再次创建常规的流视图，并保存最终的结果，而单个时间窗口的流视图内部的计算结果会改变。

例如，要维护计算三个不同范围的时间窗口计算：

```
CREATE CONTINUOUS VIEW sw0 WITH (sw = '1 hour') AS SELECT COUNT(*) FROM event_stream;
CREATE VIEW sw1 WITH (sw = '5 minutes') AS SELECT * FROM sw0;
CREATE VIEW sw2 WITH (sw = '10 minutes') AS SELECT * FROM sw0;
```

注意 `sw1` 和 `sw2` 并没有使用 `CONTINUOUS` 关键字，然而，查询他们只会返回他们各自窗口的计算结果。

step_factor

在内部，支持滑动窗口查询物化表尽可能支持更多的聚合。然而，行不能聚合到也是行粒度级别并作为查询的最终输出，因为行数据在脱离窗口范围后，从聚合结果中删除

例如，一个按小时汇总的滑动窗口查询实际上可能有磁盘上的分钟级聚合数据，因此只返回最后的 60 分钟内的数据给使用者。这些在内部的里的更细粒度的滑动窗口查询的聚合级别被称为“步骤”。“重叠”视图被放置在这些步骤的聚合中，以便在读取时候给出最终聚合结果。

您可能已经注意到在这一点上，步骤聚合是影响滑动窗口查询读取性能的一个重要因素，因为每个最终的滑动窗口聚合都是内部里的若干步骤组成。每个滑动窗口的步骤数将有多少是通过 step_factor 参数可调的

step_factor

这个参数是一个 integer 类型的值，范围在 1 到 50 之间，可以用于指定滑动窗口的 step 的计算时间长度占窗口大小（由 sw 确定）的百分比。一个小的 step_factor 值在数据离开窗口后提供更细粒度的数据，代价是会占用磁盘上的物化表的更多空间。一个更大的 step_factor 将减少对磁盘的物化表的大小，代价是离开窗口后数据更粗的粒度。
下面是一个用 step_factor 结合 sw 来聚合过去 1 个小时，step 长度为 30 分钟的例子（译者注：step_factor=50,即占比 50%，因此为 30 分钟）：

```
CREATE CONTINUOUS VIEW hourly (WITH sw = '1 hour', step_factor = 50)
  AS SELECT COUNT(*) FROM stream;
```

现在你已经知道滑动窗口查询是怎样工作的，那么是时候需要学习关于流的 join 关联了。

流的 join 关联

流视图并不限于只在流本身上做 select 查询。很多时候它还可以用于流数据与存储在数据库中的静态数据表进行关联查询。这很容易用我们称之为流-表关联的方式进行完成。

流-表关联查询

流-表关联通过将传入的 event 事件记录与被 join 表中存在的行进行匹配，来做关联。也就是说，如果表中插入的行记录能与先前读取到的事件关联上，流视图包含的流-表关联是不会更新来反映的。新的关联记录只在事件被读取时候生成。即便被 join 的表的所有数据都删除了，流视图的结果也不会改变。

支持的 join 类型

在 0.9.2 版本中开始支持

流的 join 目前只支持一部分 join 类型。`CROSS JOIN` 和 `FULL JOIN` 目前都不支持。`LEFT JOIN` 和 `RIGHT JOIN` 只在返回不匹配记录的是流这一侧的情况下才支持。 `ANTI JOIN` 和 `SEMI JOIN` 需要在进行 join 的列上创建一个索引。

样例

计算 event 的 id 在表 whitelist 中的实时事件个数:

```
CREATE CONTINUOUS VIEW count_whitelisted AS SELECT COUNT(*) FROM
stream JOIN whitelist ON stream.id = whitelist.id;
```

使用 user 表来增加关于 user 用户的更多信息。:

```
CREATE CONTINUOUS VIEW augmented AS SELECT user_data.full_name, COUNT(*)
FROM stream JOIN user_data on stream.id::integer = user_data.id
GROUP BY user_data.full_name;
```

空间上进行关联最近的城市，按 city 的 name 进行聚合:

```
CREATE CONTINUOUS VIEW spatial AS SELECT cities.name, COUNT(*) FROM
geo_stream, cities WHERE st_within(geo_stream.coords::geometry, cities.borders)
GROUP BY cities.name;
```

注意

正如你可能猜测的那样，流与表进行关联时候，如果表是非常大的表，那么可能会遇到严重的性能问题。为了最好的性能，要使用的表必须相对较小，小道足以装在内存中。另外建议在要关联的表上面创建索引。

流与流之间的关联

目前还不支持一个流和另一流的关联，但在未来版本的 PipelineDB 可能会支持。

数据备份

PipelineDB 可以使用 `pipeline-dump` 或 `pipeline-dumpall` 工具来备份，他们分别等价于 PostgreSQL 的工具 pg_dump 和 pg_dumpall 工具，并且可以到处流视图。

它的用法如下：

```
Usage:
  pipeline-dump [OPTION]... [DBNAME]

一般参数:
  -f, --file=FILENAME         output file or directory name
  -F, --format=c|d|t|p        output file format (custom, directory, tar,
                              plain text (default))
  -j, --jobs=NUM              use this many parallel jobs to dump
  -v, --verbose               verbose mode
  -V, --version               output version information, then exit
  -Z, --compress=0-9          compression level for compressed formats
  --lock-wait-timeout=TIMEOUT  fail after waiting TIMEOUT for a table lock
  -?, --help                  show this help, then exit

控制输出内容的参数:
  -a, --data-only             dump only the data, not the schema
  -b, --blobs                 include large objects in dump
  -c, --clean                 clean (drop) database objects before recreating
  -C, --create                include commands to create database in dump
  -E, --encoding=ENCODING      dump the data in encoding ENCODING
  -n, --schema=SCHEMA         dump the named schema(s) only
  -N, --exclude-schema=SCHEMA  do NOT dump the named schema(s)
  -o, --oids                  include OIDs in dump
  -O, --no-owner              skip restoration of object ownership in
                              plain-text format
  -s, --schema-only           dump only the schema, no data
```

```
  -S, --superuser=NAME        superuser user name to use in plain-text format
  -t, --table=TABLE           dump the named table(s) only
  -T, --exclude-table=TABLE    do NOT dump the named table(s)
  -x, --no-privileges         do not dump privileges (grant/revoke)
  --binary-upgrade            for use by upgrade utilities only
  --column-inserts            dump data as INSERT commands with column names
  --disable-dollar-quoting     disable dollar quoting, use SQL standard quoting
  --disable-triggers          disable triggers during data-only restore
  --exclude-table-data=TABLE   do NOT dump data for the named table(s)
  --if-exists                 use IF EXISTS when dropping objects
  --inserts                   dump data as INSERT commands, rather than COPY
  --no-security-labels        do not dump security label assignments
  --no-synchronized-snapshots  do not use synchronized snapshots in parallel jobs
  --no-tablespaces            do not dump tablespace assignments
  --no-unlogged-table-data     do not dump unlogged table data
  --quote-all-identifiers      quote all identifiers, even if not key words
  --section=SECTION           dump named section (pre-data, data, or post-data)
  --serializable-deferrable    wait until the dump can run without anomalies
  --use-set-session-authorization
                              use SET SESSION AUTHORIZATION commands instead of
                              ALTER OWNER commands to set ownership
```

连接参数：

```
  -d, --dbname=DBNAME      database to dump
  -h, --host=HOSTNAME      database server host or socket directory
  -p, --port=PORT          database server port number
  -U, --username=NAME      connect as specified database user
  -w, --no-password        never prompt for password
  -W, --password           force password prompt (should happen automatically)
  --role=ROLENAME          do SET ROLE before dump

If no database name is supplied, then the PGDATABASE environment
variable value is used.

Report bugs to <eng@pipelinedb.org>.
```

导出指定的流视图

要想导出单个流视图，必须把流视图和它相关联的物化表也同时导出，例如：

```
pipeline-dump -t <CV name> -t <CV name>_mrel # <-- Note the "_mrel" suffix
```

恢复流视图

可用使用命令 `pipeline-dump` 进行备份后，然后使用 pipeline 命令恢复备份:

```
pipeline-dump > backup.sql
pipeline -f backup.sql
```

版本升级

在 0.9.5 版本中引入

大多数 PipelineDB 的版本升级不需要任何条件，除了安装 PipelineDB 以及在你先前数据库版本的数据目录上运行数据库服务。

然而，有一些版本会改变内部系统的 catalog，这需要迁移旧的 catalog schema 到新的 schema。一种办法来完成是简单的 dump（备份）出你旧版本的数据库，然后在新版本的数据库中 恢复数据。

对于小版本的数据库升级，这是最直接的办法。对于大版本的数据库升级，这可能慢了，升级重大版本最快的方法是使用 `pipeline-upgrade` 工具. `pipeline-upgrade` 会拷贝所有的旧版本的 catalog 数据到新版本的 PipelineDB 中，然后在操作系统层面简单拷贝所有其他的数据库数据。

使用方法如下：

```
Usage:
       pipeline-upgrade [OPTION]...

参数:
       -b, --old-bindir=BINDIR      old cluster executable directory
       -B, --new-bindir=BINDIR      new cluster executable directory
       -c, --check                  check clusters only, don't change any data
       -d, --old-datadir=DATADIR    old cluster data directory
       -D, --new-datadir=DATADIR    new cluster data directory
       -j, --jobs                   number of simultaneous processes or threads to use
       -k, --link                   link instead of copying files to new cluster
       -o, --old-options=OPTIONS    old cluster options to pass to the server
       -O, --new-options=OPTIONS    new cluster options to pass to the server
       -p, --old-port=PORT          old cluster port number (default 50432)
```

```
  -P, --new-port=PORT           new cluster port number (default 50432)
  -r, --retain                  retain SQL and log files after success
  -U, --username=NAME           cluster superuser (default "derek")
  -v, --verbose                 enable verbose internal logging
  -V, --version                 display version information, then exit
  -?, --help                    show this help, then exit
```

在运行 pipeline-upgrade 前，你必须做如下的事情：

创建数据库 cluster 集群 (使用新版本的 pipeline-init 命令)
关闭旧集群的 postmaster 服务
关闭新集群的 postmaster 服务

当你运行 pipeline-upgrade 命令时候，你必须提供如下信息：

旧集群的数据目录(-d DATADIR)
新集群的数据目录(-D DATADIR)
旧版本的”bin”目录(-b BINDIR)
新版本的”bin”目录(-B BINDIR)

例如：

```
pipeline-upgrade -d oldCluster/data -D newCluster/data \
  -b oldCluster/bin -B newCluster/bin
```

或

```
$ export PGDATAOLD=oldCluster/data
$ export PGDATANEW=newCluster/data
$ export PGBINOLD=oldCluster/bin
$ export PGBINNEW=newCluster/bin
$ pipeline-upgrade
```

要把 `pipeline-upgrade` 指向 data 数据目录和 bin 程序目录和由新版本的 pipeline-init 创建的新的数据目录及新版本的 bin 目录。

注意

你应该备份（只需要 cp 到其他目录就好了）旧版本的安装目录（通常是 `/usr/lib/pipelinedb`）（只需要 cp 到某个目录），以便在升级时候依然可以用它的 bin 目录：

```
# After installing the newset PipelineDB version...
$ pipeline-init -D new_data_dir

# new_data_dir is initialized...

# Migrate old catalog data to new schema, and copy old database
# files to the new data directory
```

```
$ pipeline-upgrade -b old_version/bin -d old_data_dir \
  -B new_version/bin -D new_data_dir
```

主从复制

配置 PipelineDB 复制和在 PostgreSQL 上面配置复制是一样的。如果你已经做过这样的事情，那么对于 PipelineDB 应该也非常熟悉。如果没有这里可能值得一读，因为配置 PostgresSQL 的复制设置有一些很奇特的配置，这些配置大多数是随着复制技术演变形成的。Peter Eisentraut 写的 PostgreSQL 复制技术历史是一篇有趣的值得一读文章，如果你想知道关于复制技术更多的话。

我们不打算去研究老的复制技术比如 Log-Shipping Standby Servers ，因为这样的技术非常复杂而且不强大。唯一使用的理由估计是你有一个很老版本的 PostgreSQL 系统在运行。但由于我们是基于 PostgreSQL 9.5 内核构建的 PipelineDB 版本，我们可以充分发挥 PostgreSQL 提供的最新和重要的特性。

流复制（译者注：这里的流和流视图里面的流是不一样的，这流复制是 PostgreSQL 支持的一种主从复制机制）

PipelineDB 非常好的支持 PostgreSQL 的 streaming replication（流复制技术，包括同步和异步两种方式）使用 streaming replication 流复制技术，我们可以创建一个 hot-standby 节点，并通过 tail write-ahead 预写式日志的方式来保持与 primary 节点的日志同步，以便来服务 read-only 只读请求。在主节点失效的情况下， hot-standby 节点可以成为新的主节点。

比方说已经有一个 PipelineDB 实例运行在 `localhost:5432` 端口下。我们需要做的第一件事是在主节点上创建一个有 REPLICATION 复制权限的角色。这个角色只会用于让 standby 节点连接 primary 节点并且复制 WAL 日志流。

```
$ psql -h localhost -p 5432 -d pipeline -c \
"CREATE ROLE replicator WITH LOGIN REPLICATION;"

CREATE ROLE
```

在你的 primary 节点处于开放网络的情况下（免费建议：千万不要这么做），你也可以创建一个带有密码的角色。

下一步我们需要在 pg_hba.conf 中为 standby 节点创建一个 entry 入口，你可以在 primary 节点的数据目录中找到这个文件。`pg_hba.conf` 文件处理 PipelineDB 所有的客户端认证细节，对于我们这个样例，我们把下面的入口提供给它，但是对于真实环境，配置和这里是完全不同的。

```
host replication replicator 0.0.0.0/0 trust
```

下一步，我们需要在 primary 节点上设置一些参数，你可以通过更新参数文件 `pipelinedb.conf` 或者使用命令 pipelinedb 时候设置参数-c。你需要设置参数 wal_level 为 `hot_standby`, 设置 hot_standby 为 `on`,设置 max_replication_slots 为 1，设置 max_wal_senders 为 2。即便一个 standby 节点只需要一个 sender 连接，但在启动节点需要 2 个连接（虽然不必要，但下面的文档中的方法是需要的）。更新这些参数后，你需要重启 primary 节点。

最后我们需要为 standby 节点创建一个 replication slot (复制槽)。复制槽是一种 standby 向 primary 节点注册信息的一种方法，以便 primary 节点能够感知到那一段 WAL 日志段需要保留。一旦 standby 节点已经获取到了一个 WAL 日志段，它就会更新 pg_replication_slots 的 `restart_lsn` 列的值，以便让 primary 节点知道它现在可以回收这个 WAL 日志段。

```
$ psql -h localhost -p 5432 -d pipeline -c \
"SELECT * FROM pg_create_physical_replication_slot('replicator_slot');"

    slot_name    | xlog_position
-----------------+---------------
 replicator_slot |
(1 row)
```

这就是我们需要在 primary 节点上要做的配置。现在让我们把目光移到 standby 节点上。我们需要做的第一件事就是对 primary 节点做一个拷贝。同步 primary 的数据目录到 standby 节点是因为 standby 要从数据库的这个备份点开始启动（译者注：如果不拷贝就会导致数据不一致）。为此，我们使用 `pipeline-basebackup` 工具 (等价于 pg_basebackup)。你也可以使用 `rsync` 命令，它应该会快一点，但是会增加你自己处理鉴权的复杂性。.`pipeline-basebackup` 工具使用正常的 PostgreSQL 连接来打包所有的基础文件，你不必担心鉴权细节。

```
$ pipeline-basebackup -X stream -D /path/to/standby_datadir -h localhost -p 5432 -U
replicator
```

这里的 `-X stream` 参数就是为什么上面配置需要配置两个 slot 原因，因为需要做一个基础备份。基本上这个参数的目的是为了解决在备份过程中同步数据库的

WAL 日志的变化到 standby 节点。因此我们不必手工配置 wal_keep_segments 参数。

我们最后一步需要做的是在 standby 节点上编写一个 recovery.conf 文件，用于告诉 PipelineDB 实例：它需要在 standby 模式下操作以及如何 连接 primary 节点。例如这个文件可能是这样的：

```
standby_mode = 'on'
primary_slot_name = 'replicator_slot'
primary_conninfo = 'user=replicator host=localhost port=5432'
recovery_target_timeline = 'latest'
```

我们已经设置完毕了，让我们把 hot standby 节点的端口配置在 6544 上

```
pipeline-ctl start -D /path/to/standby_datadir -o "-p 6544"
```

你应该在 standby 的日志文件上看到如下的东西：

```
LOG:  entering standby mode
LOG:  redo starts at 0/5000028
LOG:  consistent recovery state reached at 0/50000F0
LOG:  database system is ready to accept read only connections
LOG:  started streaming WAL from primary at 0/6000000 on timeline 1
```

只需要确保连接到 standby 节点，然后确认它处于 recovery 恢复模式：

```
$ psql -h localhost -p 6544 -d pipeline -c \
"SELECT pg_is_in_recovery();"

 pg_is_in_recovery
-------------------
 t
(1 row)
```

高可用性

PostgreSQL 并没有非常好的支持高可用。大多数的应用部署都需要在主节点故障时候手动切换到备节点。故障切换可以通过命令 `pipeline-ctl promote` 或创建触发器文件 `trigger_file` （配置在 `recovery.conf` 文件）来实现转移. Compose.io 有一个博客 blog post 讲述了他们如何设计 HA 高可用方案。你可以使用他们的 Governor 系统; 要确保修改代码中的 PostgreSQL 的二进制库依赖到他们的 PipelineDB 等效的库

如果这些还不满足你的要求，请联系我们，我们帮助你找到一些办法。

逻辑解码

然而我们还没有开始研究逻辑解码（译者注：这个是逻辑复制中的技术），同样没有理由相信逻辑解码在这里可以没有任何小问题地工作。如果你是逻辑解码的使用者，并且发现 PipelineDB 与之不兼容，请联系我们。

集成其他应用

Apache Kafka

PipelineDB 支持从 Kafka 的 topic 中获取数据到流中。所有的这些功能都包含在 **pipeline_kafka** 这个扩展中。

实际上 **pipeline_kafka** 使用 PostgreSQL 的 COPY 命令来把 Kafka 消息转换成 PipelineDB 所使用的行数据。

注意

pipeline_kafka 扩展目前是官方支持的，但是没有和 PipelineDB 打包在一起，因此必须单独安装。该扩展的 github 目录在这里。编译和安装这个扩展可用在 `README.md` 中找到。

pipeline_kafka 内部使用共享内存来同步 worker 之间的状态，因此它必须从共享库目录中加载。你可以在 `pipelinedb.conf` 中配置。如果你已经加载了一些共享库，那么 `pipeline_kafka` 写在已有的共享库后面，用逗号隔开。

```
shared_preload_libraries = pipeline_kafka
```

现在你可以在数据库中创建扩展了：

```
# CREATE EXTENSION pipeline_kafka;
CREATE EXTENSION
```

在你使用 **pipeline_kafka 之前**,你必须给你的 Kafka 添加 broker 节点。

pipeline_kafka.add_broker (hostname text)

hostname 的格式是`<host>[:<port>]`。多个 broker 可以通过每次调用 **pipeline_kafka.add_broker** 来添加。

消费消息

pipeline_kafka.consume_begin (topic text, stream text, format := 'text', delimiter := E'\t', quote := NULL, escape := NULL, batchsize := 1000, maxbytes := 32000000, parallelism := 1, start_offset := NULL)

parallelism 是表示启动后台的 worker 工作进程，这些进程会读取给定的 kafka topic 的消息到给定的 stream 流中。目标流必须事先使用 `CREATE STREAM` 创建好。给定的 topic 的所有 partition 的消息会均匀地分配到每个 worker 进程读取。

可选参数 **format**, **delimiter**, **escape** 和 **quote** 和 PostgreSQL COPY 命令的参数 `FORMAT`, `DELIMITERESCAPE` 及 `QUOTE` 是类似的。区别是 **pipeline_kafka** 支持一种额外的格式: **json**。**json** 参数会让该函数把 kafka 的消息格式化成 json 对象。

batchsize 参数是用于传递给 kafka 的 `batch_size`（译者注：batch_sizes 是 kafka 侧的参数名）参数值。当消息缓冲量达到 batchsize 的值后，消费者进程就会强制 copy 并提交，然后进行下一轮数据获取。

maxbytes 控制传递给 kafka 客户端的对应参 `fetch.message.max.bytes` 。当消息缓冲量达到 maxbytes 的值后，消费者进程就会强制 copy 并提交，然后进行下一轮数据获取

start_offset ：指定消费者进程从指定 kafka 的 topic 的 partition 的 指定 offset 开始读取。

pipeline_kafka 会一直保存它所读取的 offset 值到数据库中。start_offset 值为 `NULL`,消费者进程会从数据库中已保存的 offset 开始读取消息，如果没有保存的 offset，就从 partition 最新的位置开始读取。start_offset 如果取 -1 会从每个 partition 的末尾开始（即最新）的位置开始读取，如果 start_offset 取-2，那么会从每个 partition 的起始位置开始读取消息。start_offset 取其他值会是很奇怪的事情，因为 partition 之间的 offset 值是没有关联的。

pipeline_kafka.consume_begin ( )

和上面的类似，但是会启动所有先前创建的消费者进程，而不是只启动消费者指定的 stream-topic 进程。

pipeline_kafka.consume_end (topic text, stream text)

根据给定 stream-topic 参数对，终结指定的 consumer 进程。

pipeline_kafka.consume_end ( )

和上面的一样，但是会终止所有的 consumer 进程。

生产消息

在版本 0.9.1 中添加

pipeline_kafka.produce_message (topic text, message bytea, partition := NULL, key := NULL)

产生消息 **message** 发送到目标 **topic** 中。**partition** 和 **key** 都是可选的。默认情况下，topic 的 partition 分区位置不指定，因此 broker 会根据 topic 的 partition 函数来决定哪个 partition 来产生 message 消息。如果你想把消息发送到指定的 partition 中，需要把 partition 设定为一个 integer 类型的值。参数 key 是一个 `bytea` 类型参数，这个参数会在 partition 函数中用到。

pipeline_kafka.emit_tuple (topic, partition, key)

这个触发器函数可以用来吧元组数据以 json 格式发送到 kafka 的 stream 流中。这个函数只能用在 `AFTER INSERT OR UPDATE` 和 `FOR EACH ROW` 触发器中。在 `UPDATE` 语句中，更新后的元组会被发送过去。必须要提供 **topic** 参数，而 **partition** 和 **key** 都是可选的。因为这是一个触发器函数，所有的参数必须以字符串形式传递，并且不可以以关键字参数的形式指定参数（译者注：这种参数形式是 partition=1，这种写法）如果你只想指定 **topic** 和 **key**，把 partition 的值设置为`'-1'`，这样会让 partition 不指定。**key** 是被发送到 kafka 的元组里的列名，这个 key 会被当做分区的 key。

元数据

pipeline_kafka 使用若干张表用于跟踪系统启动过程中的持续状态跟踪。

pipeline_kafka.consumers

存储每个流-topic 的消费者信心，消费者是 **pipeline_kafka.consume_begin** 创建的。

pipeline_kafka.brokers

保存所有的 kafka 的 broker 服务节点，消费者可以连接 broker 服务。

pipeline_kafka.offsets

存储 kafka 的 topic 的 offset，以便消费者可以在系统中断或重启的情况下，可以从 offset 值的地方再次读取数据。

注意

参考 SQL on Kafka 获得使用 PipelineDB 与 Kafka 的深度教程。

亚马逊 Kinesis 流计算应用

PipelineDB 也支持集成来自 Amazon Kinesis 的流数据。这个功能是由插件 **pipeline_kinesis** 来提高的。内部上，这个插件管理了 bgworker 进程，这个进程是使用 AWS SDK 来消费数据的，然后把它拷贝到 pipeline 的流中。

这个插件的 github 目录在 这里. 你可以在 README.md 文件中找到打包和安装这个插件的方法。

想要开启这个插件的话，你需要显式地创建这个插件：

```
# CREATE EXTENSION pipeline_kinesis;
CREATE EXTENSION
```

想要开始抽取数据的话，你必须首先告诉 pipeline 在哪里以及怎样获取 kinesis 的数据，配置在这里：

pipeline_kinesis.add_endpoint(name text, region text, credfile text := NULL, url text := NULL)

name 是一个 kinesis 端点的唯一标识符。Region 是标识 AWS 区域的一个标识符，比如 `us-east-1` or `us-west-2`.

credfile 是一个可选参数，它允许重写 AWS 认证的默认文件位置。

url 是一个可选参数。它使用一个不同的（非 AWS）kinesis 服务器。这在测试本地 kinesis 服务时候比如 kinesalite 非常有用。

消费消息

pipeline_kinesis.consume_begin (endpoint text, stream text, relation text, format text := 'text', delimiter text := E'\t', quote text := NULL, escape text := NULL, batchsize int := 1000, parallelism int := 1, start_offset int := NULL)

启动一个逻辑消费进程组来，在 endpoint 服务地址上消费 kinesis 流的数据，然后把数据拷贝到 pipeline 流中。

parallelism 是表示启动后台的 worker 工作进程，每个进程会用于进行负载均衡。注意——这个不是用于设置分片的个数，因为这个扩展是内部进程。默认在 1 是足够的，除非消费者消费能力不足。

可选参数 **format**, **delimiter**, **escape** 和 **quote** 和 PostgreSQL COPY 命令的参数 `FORMAT`, `DELIMITERESCAPE` 及 `QUOTE` 是类似的。

batchsize 参数是用于传递给 AWS SDK 的 `limit` 参数，这个参数在 Kinesis GetRecords 中使用

start_offset 是用于这个扩展从流的那个位置开始读取数据。start_offset 如果取 -1 会从每个 partition 的末尾开始(即最新)的位置开始读取，如果 start_offset 取-2，那么会从每个 partition 的起始位置开始读取消息。内部机制上，这两个值映射 Kinesis GetShardIterator 的 `TRIM_HORIZON` 和 `LATEST` 上面，参考 Kinesis GetShardIterator 获取更多细节。

pipeline_kinesis.consume_end (endpoint text, stream text, relation text)

终止一个指定的消费者的后台工作进程。

pipeline_kinesis.consume_begin()

启动所有的先前创建的消费者进程。

pipeline_kinesis.consume_end()

关闭所有的后台先前启动的消费者工作进程。

元数据

pipeline_kinesis 使用若干个表持续性跟踪在系统重启过程中自身的状态。

pipeline_kinesis.endpoints

存储由 **kinesis_add_endpoint** 创建的每个服务地址的元数据信息。

pipeline_kinsesis.consumers

存储每个消费者的元数据信息，消费者 consumer 由 **kinesis_consume_begin** 创建。

pipeline_kinsesis.seqnums

存储每个 consumer 的每个分片元数据信息，也就是序列号。

统计性视图

PipelineDB 包含了一些统计收集视图，这些视图用于窥探数据库系统的运行状况。统计进程可能会被进程，流视图，流或安装过程打断。

每个统计视图描述如下：

pipeline_proc_stats

统计工作进程和合并进程的信息。这些统计信息只会统计维护基础进程的延续信息。

```
                          View "pg_catalog.pipeline_proc_stats"
              Column     |           Type           | Modifiers
---------------+--------------------------+-----------
 type          | text                     |
 pid           | integer                  |
 start_time    | timestamp with time zone |
 input_rows    | bigint                   |
 output_rows   | bigint                   |
 updated_rows  | bigint                   |
 input_bytes   | bigint                   |
 output_bytes  | bigint                   |
 updated_bytes | bigint                   |
```

```
 executions    | bigint                 |
 tuples_ps     | bigint                 |
 bytes_ps      | bigint                 |
 time_pb       | bigint                 |
 tuples_pb     | bigint                 |
 memory        | bigint                 |
 errors        | bigint                 |
 exec_ms       | bigint                 |
```

pipeline_query_stats

关于流视图的统计函数

```
                            View "pg_catalog.pipeline_query_stats"
              Column     |  Type  | Modifiers
---------------+--------+-----------
 name          | text   |
 type          | text   |
 input_rows    | bigint |
 output_rows   | bigint |
 updated_rows  | bigint |
 input_bytes   | bigint |
 output_bytes  | bigint |
 updated_bytes | bigint |
 tuples_ps     | bigint |
 bytes_ps      | bigint |
 time_pb       | bigint |
 tuples_pb     | bigint |
 errors        | bigint |
 exec_ms       | bigint |
```

pipeline_stream_stats

针对流的统计函数

```
                            View "pg_catalog.pipeline_stream_stats"
              Column     |  Type  | Modifiers
---------------+--------+-----------
 schema        | text   |
 name          | text   |
 input_rows    | bigint |
 input_batches | bigint |
 input_bytes   | bigint |
```

pipeline_stats

数据库安装层面的统计函数

```
                          View "pg_catalog.pipeline_stats"
              Column     |          Type          | Modifiers
---------------+-------------------------+-----------
 type          | text                    |
 start_time    | timestamp with time zone |
 input_rows    | bigint                  |
 output_rows   | bigint                  |
 updated_rows  | bigint                  |
 input_bytes   | bigint                  |
 output_bytes  | bigint                  |
 updated_bytes | bigint                  |
 executions    | bigint                  |
 errors        | bigint                  |
 cv_create     | bigint                  |
 cv_drop       | bigint                  |
```

配置参数

synchronous_stream_insert

用于设置插入流的数据是否需要是同步的或者非同步的，默认是 false，即异步插入。

continuous_queries_enabled

用于设置是否应该开启针对新创建数据库的流视图查询，默认值是 false。

continuous_query_crash_recovery

用于设置是否流查询经常是否需要从错误中恢复。默认是 true。

anonymous_update_checks

设置 PipelineDB 是否需要异步检测有新版本。默认是 true。

continuous_query_materialization_table_updatable

设置流视图的变化是否可以直接作用于物化表。默认是 false。

continuous_queries_adhoc_enabled

用于设置是否允许即席流查询，默认是 false。

continuous_query_ipc_shared_mem

用于设置对于 IPC 的共享内存队列大小。默认是 32MB。

continuous_query_combiner_work_mem

设置用于合并流查询部分计算结果的最大内存。在把数据存储到临时的磁盘文件前，每个合并进程内部的排序计算和哈希表会使用大量内存。默认是 256mb。

continuous_query_max_wait

设置一个流查询进程聚集批量数据的等待时间。更高的值通常意味着更少的流视图变化频率，但是会导致计算延迟，并且在进程崩溃的情况下导致更多的数据丢失。默认是 10ms（毫秒）。

continuous_query_batch_size

用于设置在执行一个流查询计划前，累积的最大事件数。更大的值通常意味着更少的流视图更新频率，但是会导致计算延迟，并且在进程崩溃的情况下导致更多的数据丢失。默认值是 10000。

continuous_query_num_combiners

设置对于每个数据库并行的流视图查询合并进程数。更高的值会使用多核，并且会提高吞吐量直到 I/O 耗尽。默认值为 1。

continuous_query_num_workers

设置对于每个数据库并行的流视图查询工作进程数。更高的值会使用多核，并且会提高吞吐量直到 CPU 耗尽。默认值为 1。

continuous_view_fillfactor

设置使用物化数据表的默认的填充因子，默认是 50

sliding_window_step_factor

设置默认的滑动窗口查询步长占窗口大小的百分比。较高的数字将提高性能，但延迟刷新间隔。默认值：5。

continuous_query_commit_interval

设置在提交结果之前，数据保存在内存中的毫秒数。更长的 commit 提交时间间隔会在更少频率的流视图更新成本下获取更高的性能，当然也会有更多丢失数据的风险。默认是 50 毫秒。