Streaming Databases & PipelineDB

REPORT STUDY ON STREAMING DATABASES
AND
USE CASE IMPLEMENATION WITH PIPELINEDB

Batuhan Tüter | ULB Student Id: 000460176
Marc Garnica | ULB Student Id: 000456513

Advanced Databases
Prof. Esteban Zimanyi

Université Libre de Bruxelles
Winter semestre 2017 - 2018

Table of Contents

Introduction

Computer science industry has on its own nature to be driven by strong and frequent revolutions. The emergence of the computers using microprocessors meant the spread of the computer science to all the sectors in human life. Relational databases and the RDBMS have been used around all the sectors since its first appearance in 1979. For all the experts, one of the most revolutionary and with highest impact topics currently is the emerging of Big Data and the current link with business insights and analytics.

Although being used in many companies for the past years, the words 'Big data' are still undefined in so many terms. Data volumes are growing rapidly, and this implies a huge challenge for all the businesses. It is more than probably that while reading this introduction article more than 29 Million WhatsApp messages have been sent, 500 hours of video uploaded to YouTube, over 150,000 emails, 3.3 million posts on Facebook and over 3.8 google searches have been done in the internet [1].

Beyond this example, real world gathers a lot of use cases where high volumes of data are collected and used, from big e-commerces like Amazon or governments data warehouses to Smarts Cities. All of them share the principle of "process data and extract actionable insights from it". Most Big Data applications are currently storing all the incoming data and processing them later usually by means of batch processing and analytical queries.

But a new paradigm is emerging from the needs of all this use cases where storing huge volumes of data is not enough. Some of the process may take quite a lot of time to generate the corresponding output. In some use cases **is much useful to know the results faster.** Even more, once we know the output, data is just redundant.

Mainly often, on these use cases the output is the priority of all the end-to-end process, and being able to act to fix problems or situations is the key to success. Think about Trading algorithms, smart patient care, monitoring tools, fraud detection, smart applications (smart car, smart homes...), vehicles tracking, sport monitoring and analytics. Being able to generate **real-time analytics** to drive instant decision can make the difference in all these sectors and more.

Stream processing is a computer programming paradigm, equivalent to data-flow programming, event stream processing and reactive programming that allows some applications to more easily exploit the data. Data stream management systems (DSMS) also called Streaming Databases are software systems handling data streams by offering a flexible processing of them concurrently. This kind of systems prioritizes data processing and computation over data storage.

This study reviews the topic of the Streaming databases, evaluating the initial state of art and the emerge of this paradigm as well as the different schemas and technologies applied to the systems.

The study drives this database system analysis from the general concepts presented in Streaming databases to the specific details of PipelineDB.

PipelineDB is an open-source, PostgreSQL based, project and it applies the concept of streaming processing to the relational databases. The study concludes with a simple introduction to the PipelineDB implementation. The main objectives of this implementation are to benchmark the performance and characteristics of this type of database systems, how PostgreSQL is adapted in PipelineDB to serve the high output requirements of the stream processing system [2].

Streaming databases

DATA-AT-REST VS DATA-IN-MOTION

The introduction of streaming databases or Data Stream Management systems (DBMS) can be summarized as processing data without storage to maximise the throughput and speed the system up.

As presented in the introduction of this study, since the emerge of Relational Databases and even with the new era of Big Data, most systems focus its efforts in store high volumes of data in different schemas: databases, file systems or other forms of massive storage. Applications then, would query the data or compute directly over the data as needed with ad-hoc query systems.

This approach involves so many technologies and is generally called as the **batch processing** or **data-at-rest infrastructure** approach (Figure 1), where by means of a query processing system the data is analysed. In the data-at-rest infrastructure approach, a process is needed to compute and query the data that will be used for further analytics.

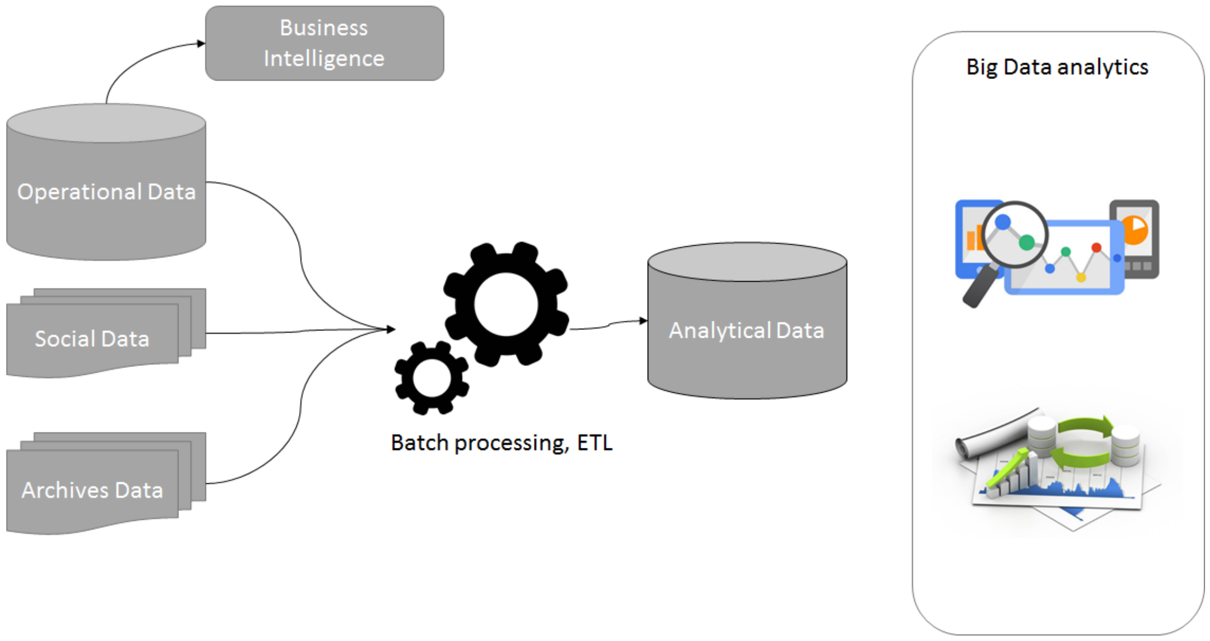


Figure 1: Data-at-rest infrastructure

Stream processing is the process of **data-in-motion** (Figure 2)**,** the data is computed directly as it is produced or received. This new paradigm differs from its root with the **data-at-rest infrastructure**, the application logic, analytics and queries exist continuously and data flows through them continuously.

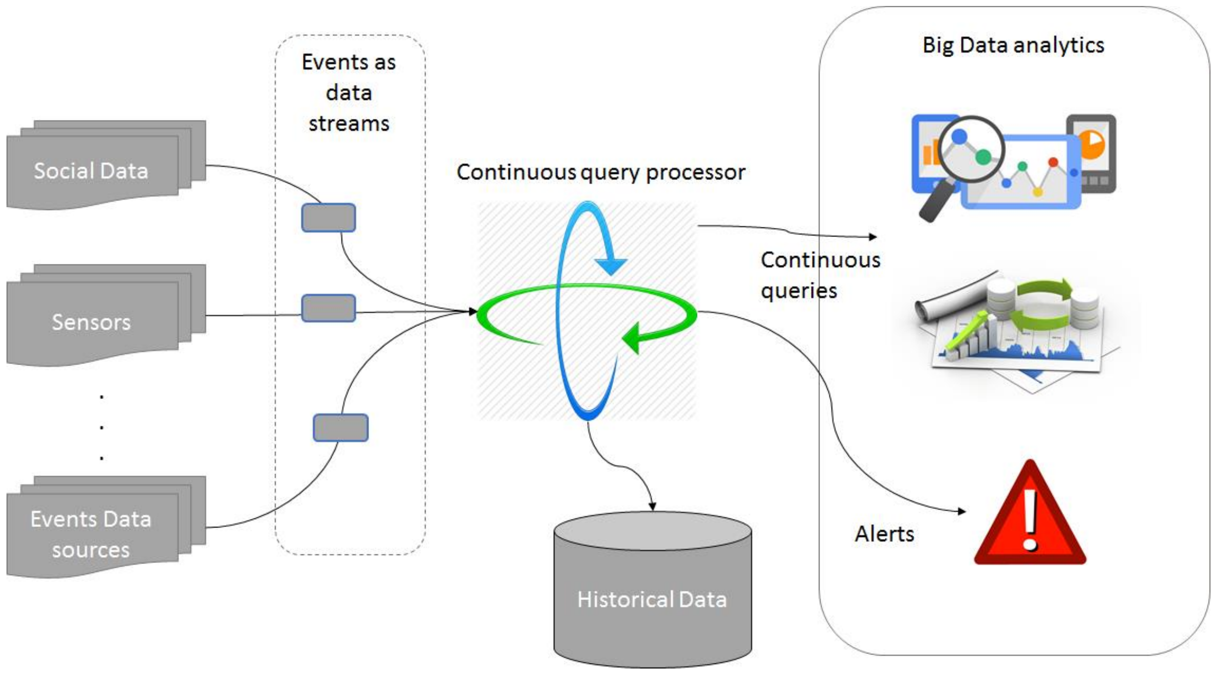


Figure 2: Data-in-motion infrastructure

The direct benefits of this new architecture and paradigm are present in day to day challenges of the real world:

- Analytics react to events instantly. No lag time between the occurrence of the event, the treatment of the event and the action taken.
- Stream databases can handle larger volumes than other processing systems.
- The nature of the data is indeed continuous in the real world.
- Decentralization of the infrastructure. No more large and expensive data centres, way towards microservices architecture.

As we can see in Figure 2, sometimes the computation of data needs to maintain a contextual state or historical data. This state can be used to store needed information from the events treated. This variant of the data-in-motion infrastructure is called **Stateful Stream Processing.**

For instance, a fraud detection system would keep the last made transactions for each credit card to be able to compare the new transactions with them. Or the system state can keep the previous added items to shopping cards for responding immediately to the user needs.

EVENTS, DATA STREAMS AND CONTINUOUS QUERIES

Most of data nowadays is based on continuous streams, think about sensor events informing about the weather state of a city or a monitoring tool for a complete manufacturing chain of cars. All this data is created as a series of events over time.

A **data stream** is defined as a none ending, real-time and continuous sequence of items, ordered (by arrival time or timestamp). Usually we can analyse two types of data streams:

- Transactional DS: Interactions between actors in a process: Credit card operations, web monitoring, etc.
- Measurement DS: State broadcast from entities to the central gateway of control: Sensors, climate, real-time patient care or IoT devices.

Continuous queries are the engine running behind Continuous processors and real-time analytics. The basic definition of a continuous query is a query support that updates previously emitted results. In other words, a continuous query outputs the results in a dynamic table that is continuously updated and can be queried like any regular static table. In contrast to a normal query which terminates and returns a static table, continuous queries run over-time and produce a dynamic table updated on the fly. This concept is a similar approach to what materialized view maintenance tried to implement in SQL.

A materialized view is defined as a regular query where all the results are stored in memory or disk so then the view does not need to be computed on-the-fly when it is needed. In traditional Database Systems the system is the responsible to update this view whenever the base relation changes. Following the change of paradigms to data-in-motion, the data streams are the equivalent entity to the changes in the base relations, with the major difference that this streams are directly treated to update the view.

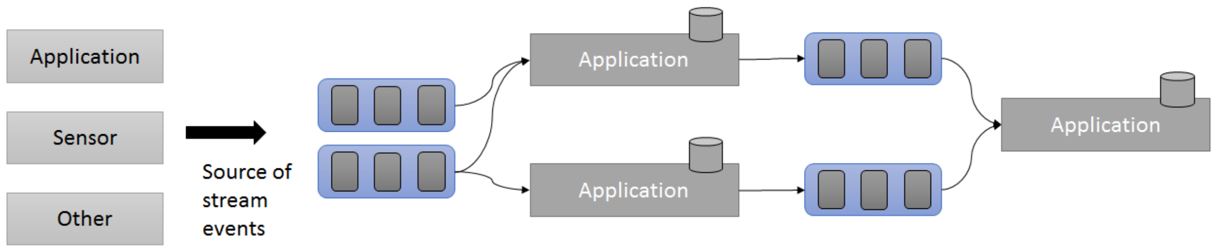


Figure 3: Data streams processing

By means of streams and continuous queries, several applications can consume the data for their own benefits and event produce streams for some other applications. With this approach data flows through the application logic of the system without an implicit store. See Figure 3.

QUICK VIEW INTO THE HISTORY: REAL-TIME PROCESSING SYSTEMS

Processing data on the fly has been a constant requirement in most of the systems. The earliest attempt to create an engine able to capture the changes in real time and update its data was at the end of 1980s with Active Databases.

Triggers and other mechanisms implemented in Active Databases enable the system to respond automatically to the events from the real world and from the database itself. The evolution from passive databases to active databases was the first step towards the design of database systems responding to the income of events.

Although being used around all relational databases the active model was mainly derived to rules processing and maintenance actions. Several times, the syntax was too complex and drove the systems to actions with high computational cost.

Between the 1990s and early 2000s various systems for managing data streams emerged. Most of them differed from the traditional database management systems in several key points:

Table 1: DBMS and DSMS comparison

Traditional DBMS	Data Stream Management systems
Persistent data in relations	Volatile data streams
Random access	Sequential access in time
One-time queries	Continuous queries
Unlimited (or limited by hardware) secondary storage	Limited to main memory storage
Focus on the current state	More and more focus on the sequentially of the data events
Relatively low update rate	Real-time requirements in updates and analytics
Query processing predictable by physical plan optimization	Data arrival and metadata variable

The first tools came from the relational databases and tried to manage streams as tables to be able to use SQL over it. These systems were facing the challenge to handle data streams using a limited amount of memory and no random access to the data. To achieve this, two main strategies were applied initially:

- Synopses: this strategy is based in the idea to summarizes the incoming data by compression techniques. Maintaining only a synopsis of the data obtained by statistical methods as sketching. Even though reducing the amount of data, this strategy may lead to wrong results and interpretations due to accuracy lose in the data.

- Windows: cut the incoming streams into parts and manage each part as a static table. Instead of compressing the data, window operators only look to a portion of it. The nature of the cutting windows may differ depending on the objectives of the analysis, element based windows, time based windows...etc.

Several tools appeared following these approaches, TelegraphCQ (a PostgreSQL fork), StreamBase or StreamSQL. TelegraphCQ uses a continuous query processor based in timestamped windows cutting the incoming data streams. On the other hand, StreamSQL has been used in some sectors of the industry. This software extends the SQL system to support streams addition to tables, streams then can be treated as tables which means that projections, joins, unions and aggregations can be executed on them.

These strategies meant a remarkable improvement, but they were not able to provide a high and efficient throughput when working with large streams of data. With the new wave of NoSQL database technologies, a lot of stream processing systems have emerged to face the data-in-motion paradigm. Well-known examples are Apache Storm, Heron or recently launched Google Dataflow.

- Apache Storm is distributed computational system to support for real-time data processing to Hadoop System. It can process large volumes of high-velocity data (more than 100 bytes messages per second per node). Storms uses **spouts** to read tuples form an external source and emit them as streams to be treated. **Bolts** are the responsible to filter, operate, aggregate, join or interact with databases during the stream process.

- Heron (from Twitter) is a real-time and distributed stream processing engine.

- Google Dataflow is an easy to use and simplified stream data processing engine. As many other products of Google Cloud, it enables the user to develop and built a operative stream processing engine in a short time and without many expertise. The user can stream data or batch some historical repository to build real-time analytics on them. It is automatically integrated with Google Cloud services as Cloud Pub/Sub, Big Query or Cloud machine learning or even with Apache Kafka.

All this solutions and even more emerging solutions in the NoSQL world can provide a truly efficient system to manage data in motion and real-time analytics. Their main drawback is that they are extremely developed for this concept, new conceptual design, new frameworks which sometimes lead companies not to use these technologies.

For the purpose of this study, PipelineDB was selected as the Stream processing tool to analyse. All its characteristics will be explained with details in the following sections of the study but PipelineDB has one main advantage: **PostgreSQL code-based,** which means that everything is inside the PostgreSQL engine. SQL is one of the most used and spread technologies around the Database systems sector, so companies have years of experience using it.

Moreover, setting up PipelineDB and PostgreSQL as the stream processing system can be combined in the same instance with having a standard PostgreSQL database design. All the advantages of PostgreSQL as a database management system are included inside PipelineDB. This can include for instance, complex operators, joins, query plan optimization or geographical and temporal database easy integration.

PipelineDB

WHY STREAM IN SQL?

Over the past 5-6 years, more and more new technologies have emerged to cope with the new and useful data-in-motion management. Apache Flume, Kafka, Spark, Storm, Google Data flow. Smart tools included in the wave of NoSQL were the relational model is replaced by more efficient, flexible and ad-hoc models.

These systems have not failed in stand in the industry and represent a huge block in the Streaming databases race. Then, why can be useful to implement streaming concepts on SQL? Which is the main benefit the companies can get through?

All these systems have great performance rates and scalability. But in contrast, the installation and setup require high level of expertise and includes a very difficult learning curve. Having systems implementing streaming concepts in SQL boost up all the advantages of SQL with the addition of the streaming features.

Companies feel more comfortable with SQL and the resources are more available. This is the reason why some other companies working with streaming databases are extending their products to handle streams on SQL models: StreamBase, StreamSQL, Truviso (Also a fork of PostgreSQL) and Apache SparkSQL.

PipelineDB provides an easy setup and use as it is based (originally forked) from PostgreSQL. PipelineDB can do everything PostgreSQL can do, but with some powerful addition to handle high throughput and streaming workloads.

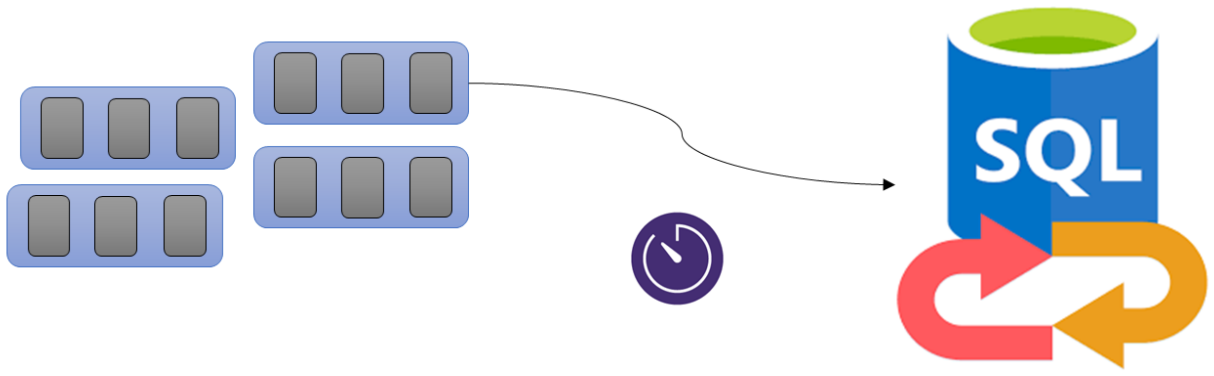


Figure 4: Streaming and SQL, a trustful environment

PIPELINEDB: STREAMING POSTGRESQL

PipelineDB was founded on 1st of December in 2013 by Derek Nelson and Jeff Fergusson, and the first enterprise version was launched on 14th of January in 2016. The last version of this software, PipelineDB 0.9.8, was released on August 2017.

PipelineDB is a open-source SQL analytics databases that runs SQL queries continuously on streaming data. Results for the queries specified are updated incrementally as new data arrives. This enforces low latency rates, no need to deploy real time application code managing the database and no external data marts or data warehouse to manage analytical data from operational data. It is fully compatible with PostgreSQL.

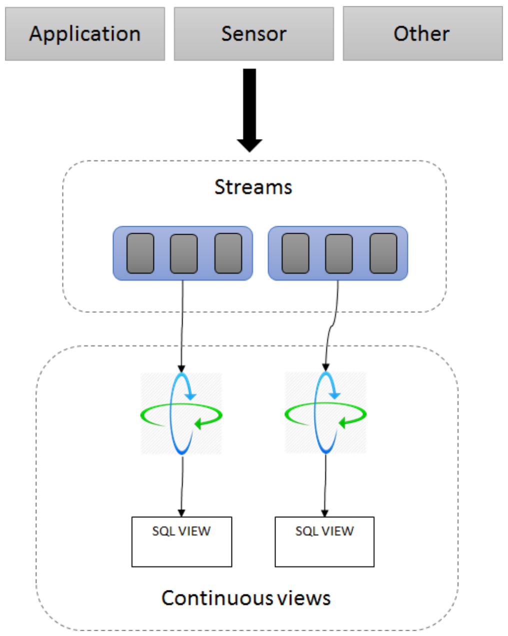
Figure 5: PipelineDB and PostgreSQL logos

This software differentiates form PostgreSQL by handling streaming data and using SQL queries instead of some expert and special programming languages. In other words, anyone familiar with SQL queries should be able to setup and deploy the software quickly.

It is important to understand that PipelineDB excels SQL management in Postgres reducing the amount of information that needs to be persisted in the database. Raw data is no longer a priority. Moreover, PipelineDB paradigm is only efficient when the queries are known **a priori.** In other words, the data warehouse extracted from the continuous views is not an easy-changeable schema. Continuous views can be explored and analysed in ad-hoc way, but the data that flowed into the system cannot be analysed in ad-hoc fashion, the exploit process this data receives is fixed in advance.

Practically speaking, PipelineDB is a stream processor, that enables the DBMS the possibility to avoid the ETL stage between Operational and Analytical data. With an a priori definition of the required queries, PipelineDB continuously build your analytical data.

STREAMS AND CONTINUOUS VIEWS



PipelineDB is based around the abstraction of continuous views. As presented in previous sections of this study, the concept of continuous views extends the tradition SQL views but with the main difference that these views define SELECT clauses including streams as a source to read FROM.

Data streams are defined as events containing a subset of data referent to the event. The main feature of continuous views is that they store uniquely their output in the database. Furthermore, the output is continuously updated incrementally as new data is ingested as streams by the system. The stream is no longer used neither stored once all views that require have use it.

An event contained in a data stream only exists until it is consumed by all the continuous views reading from it. Different continuous views can share as a source the same stream channel. This enables the possibility to analyse the data flowing into the system in different ways.

Data streams syntax

As mentioned, PipelineDB is strictly using the syntax of PostgreSQL. Creating a stream is similar to create a table:

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

The most interesting thing of this syntax is the COLLATE collation. Specifying a collation means to assign to each column the methods to compare and sort. The LIKE clause enables the stream to extend the column names and data types of an existing entity.

Streams can be modified by simply using ALTER STREAM close:

```
ALTER STREAM stream_name ADD COLUMN new_column new_data_type;
```

To push data to the streams in order to let the continuous views exploit them, the same syntax as PostgreSQL INSERT statement is used. The insert into streams can be performed at once, batched, using some system and user-specific functions or even with subselects inside.

```
INSERT INTO stream_name(column_1_name, …, column_N_name) VALUES(column_1_value, …, column_N_va
lue);

INSERT INTO stream_name(column_1_name, …, column_N_value) VALUES(value_set_1), …, (value_set_M
);
```

Where each value set is of type (column_1_value, … , column_N_value)

```
INSERT INTO stream_name(column_1_name, …, column_N_name) VALUES(system_function(), user_specif
ic_function());

INSERT INTO stream_name(column_1_name, …, column_N_name) SELECT(column_1_name, …, column_N_nam
e) FROM table_name;
```

Insert into streams can also be used by SQL prepared inserts syntax:

```
PREPARE insert_to_stream_name AS INSERT INTO stream_name(column_1_name, …, column_N_name) VALU
ES(column_1_value, …, column_N_value);

EXECUTE insert_to_stream_name
```

Copy into streams is also available in PipelineDB:

```
COPY stream_name(column_1_value, …, column_N_value) FROM‘ path / to / file’
```

This statement can be extremely useful to get data from a bash process copying from the standard input channel:

```
$> ./executeBashProcess |

> pipeline -c “COPY stream_name ((column_1_value, … , column_N_value) FROM STDIN”
```

Due to the direct compatibility with PostgreSQL, all clients working with PostgreSQL can insert data into streams.

Continuous views syntax

Once understood the creation and population of data streams. The syntax is the following:

```
CREATE CONTINUOUS VIEW view_name AS select_query
```

Select_query is a subset of PostgreSQL SELECT statement but referencing streams but using STREAMS in the FROM clause:

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

The *from_item* is the item from where the continuous view will populate and can be one of the followings:

A data stream: `stream_name[[AS]] alias[(column_alias[, ...])]]`
A table name: `table_name[[AS]] alias[(column_alias[, …])]]`

Or a join of different from_item entities:

```
from_item[NATURAL] join_type from_item[] ON join_condition]
```

The expression on the create statement is purely a PostgreSQL expression and can be given an alias. An important concept that will be explained in detail in the following sections of this document is the WINDOW entity, used to reference from OVER clauses or subsequent window definitions.

Data retrieval from a continuous view is as simple as PostgreSQL SELECT clause.

```
SELECT * FROM continuous_view_name;
```

As the continuous views may grow exponentially, PipelineDB provides native support for row expiration via time-to-live TTL specified at continuous view level. The complete syntax is available online .

```
CREATE CONTINUOUS VIEW continous_view_name WITH(ttl = ‘1 week’, ttl_column = ‘minute’)

AS SELECT minute(arrival_timestamp), COUNT( * ) FROM stream_name GROUP BY minute;
```

Continuous view can be easily deleted from the database with the statement *DROP CONTINUOUS VIEW*, and PipelineDB provides a useful way to delete the rows of a continuous view without having to delete the view itself by

```
SELECT truncate_continuous_view(‘continuous_view_name’)
```

The stream processing implicitly attached to the continuous view can be activated or deactivated by the user at any time required by this both functions:

```
SELECT activate(‘continuous_view’)

SELECT deactivate(‘continuous_view’)
```

Continuous aggregates, transforms and joins

As presented in the definition of continuous views, one of the main features of PipelineDB is the easy and light computation of **continuous aggregations**. Continuous aggregates are updated in real time as new streams flow into the system and are consumed by the continuous views. More complex aggregate functions are managed transparently in PipelineDB as Avg, stddev, percentile etc.

Continuous Joins

Although Continuous Views are selecting tuples from stream, sometimes it is necessary to combine incoming streaming data with static data stored in PipelineDB tables, thus streams can be joined with existing tables to retrieve information, that can be achieved with stream-table joins. An example query which augments incoming user data with richer user information stored in the "users" table is as follows;

```
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;
```

Whereas, joining a stream with another stream is not supported by PipelineDB yet.

Continuous transforms

Continuous transforms are defined as streaming operations performed to the data flowing into the system without storing it. No data is stored explicitly in the system, which means that no aggregations are possible for continuous transforms.

What PipelineDB manage to perform by continuous transformations is to mimic the functionality of trigger AFTER INSERT FOR EACH ROW. The output of the continuous transforms is by default inserted in their output streams. The syntax for creating Continuous Transforms is the following:

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

As presented before from_item can be either a stream, a table in the database or a join of from_items.

Sliding windows

PipelineDB is also capable of using the current time when a Continuous View is being updated. Queries that are relating to this current time within the "Where" clause are called sliding-window queries. In other words, sliding windows only consider data within a certain time window at read time. Data outside of the specified window will never be visible at read time, and consequently the aggregations are only using the data within the window boundaries.

To get the users that are seen in the last minute (from a stream) we can create a Continuous View by write the following query;

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

Where clock_timestamp() is a built-in function that returns the current timestamp and arrival_timestamp is a special attribute of the all incoming events to the PipelineDB.

“clock_timestamp() - interval '1 minute' returns the timestamp corresponding to 1 minute previously and we the continuous query will compare this with the arrival_timestamp every time a new event is read. This gives us a sliding window with 1-minute width.

Although this is a standard PostgreSQL syntax, it is not necessary to add this “Where” clause explicitly; the query can also be written in the following way;

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

“With” clause, having the “sw” (sliding window) storage parameter is internally translated into the previous “Where” clause, by the PipelineDB.

Sliding-window queries also work with aggregate functions and it is also possible to create regular views over a sliding window continuous view to have multiple sliding windows for the same query, which can be done as follows:

```
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;
```

This query will be keeping track of user event counts for the last 5 minutes, 10 minutes on different views by using the 1 hour continuous view.

Continuous views can also be declared using a window definition:

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

In this case, window_name should reference OVER clauses or window definitions. A window definition is set up as follows:

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

The *frame_clause* is a set of related rows for each row consumed by the continuous query (current row). The *frame_clause* has the following syntax:

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

Where *frame_start* and *frame_end* can be:

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

Built-in Integrations

One of the greatest features of PipelineDB is its full compatibility with PostgreSQL 9.5, which means that all PostgreSQL built-in functionality is available to PipelineDB users

PipelineDB supports ingestion of data from Kafka topics, which consist of stream of records holding different information, into streams with **pipeline_kafka** extension and from Amazon Kinesis streams with **pipeline_kinesis** extension.

As a generic JVM-based service wide used for stream messaging and real-time infrastructures, Kafka is present in many streaming databases projects. Pipeline_kafka is an extension to produce streaming data from Kafka ingestible at PipelineDB side.

Use case study

In this section of the project, an end-to-end use case has been implemented using PipelineDB with two main objectives:

- Test and interact with the most important features of the tool
- Analyse and compare the performance of PipelineDB against PostgreSQL in some specific queries.

The complete repository with all the source code for this project can be found in https://github.com/marcgarnica13/Pipeline-DB-project.git .

DEFINITION

The topic selected for the implementation is 'Events and social monitoring in real time'. From an event organizer point of view, it is important to keep a real tie analysis on the social media opinions regarding the event, for example, the hashtags used, or the likes given to the different speakers. For that reason, a set of data visualizations tools have been implemented to improve their tracking of the event, and the reactions in the social media as well.

In this virtual social media from where the real-time dashboard is consuming data from have these main features:

- Organizers post photos and they tag other organizers.
- Users can like the photos.
- Users can post comments and use hashtags.

The real-time monitoring dashboard is extremely useful for the event organizer team to track how the event is behaving in the social media by means of hashtags analysis and the performance of the organizers. In order to do that, the dashboard contains:

- List of all hashtags used.
- Search for hashtags.
- Top hashtags used
- Likes per organizers.
- Evolution of hashtags in a time frame.

As the list of queries and visualizations is fixed a priori from the requirements of the clients, there is no need for a further analysis of the data, this is an exemplar case for the streaming process implemented by PipelineDB.

It is important to mention that the social media source has been virtualized to minimize the scope of the use case to only PipelineDB related implementation. In order to do that, several bash scripts have been implemented simulating the generation of hashtags and likes. Refer to the main page of the GitHub repository to know the details on how to insert data in the streaming system, as well as, how to install PipelineDB and the use case implementation.

DATA MODEL AND USE CASES

For the test of the main features of PipelineDB, two different use cases have been implemented. On each use cases different functionalities of PipelineDB have been introduced in order to analyse its capabilities. The first use case includes the treatment of the hashtags flowing form the social medias to the system. In this case, the analyses focused on the implementation of Continuous Aggregates and Sliding Windows. The second use case analyses the Continuous Transforms and Joins by means of continuous streaming of likes into the system.

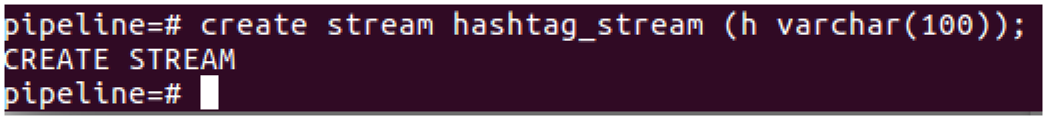
Hashtags Use Case

In this use case two different analysis will be taken into account: A simple streaming analysis implementing the concept of continuous views of PipelineDB and a streaming analysis with sliding windows.

Simple Streaming Analysis: Continuous aggregates

The main concept of this use case is having a total count of hashtag appearances. For that reason, two main entities need to be created. First, the data stream flowing into the system as a hashtag **hashtag_stream.**

```
create stream hashtag_stream (h varchar(100))
```

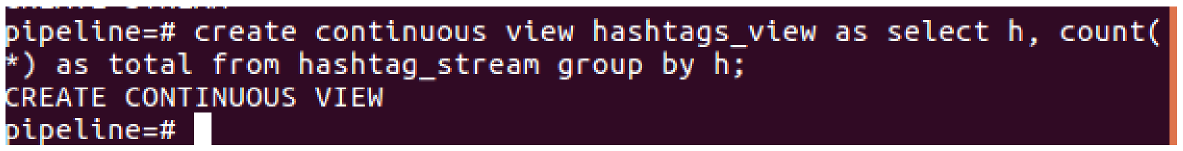


```
pipeline=# create stream hashtag_stream (h varchar(100));
CREATE STREAM
pipeline=#
```

Figure 6: Stream creation in PipelineDB (psql) command line

This data stream will be consumed then by a simple continuous view counting each hashtag appearance as follows:

```
create continuous view hashtags_view as
select h, count( * ) as total from hashtag_stream group by h;
```



```
pipeline=# create continuous view hashtags_view as select h, count(
*) as total from hashtag_stream group by h;
CREATE CONTINUOUS VIEW
pipeline=# 
```

Figure 7: Continuous view in PipelineDB command line

General schema for the implementation is as follows:

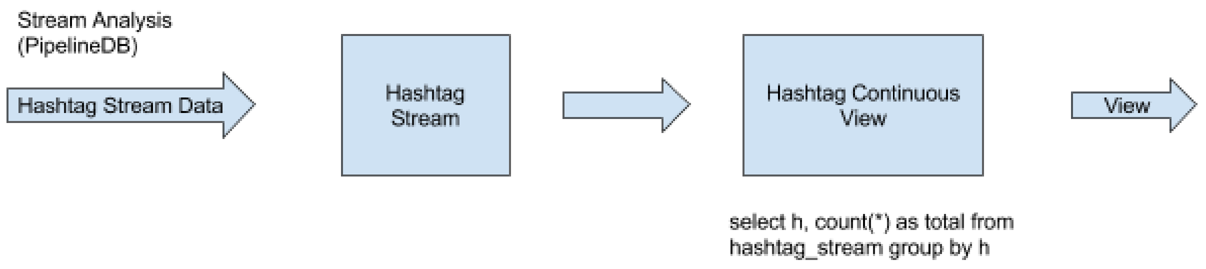
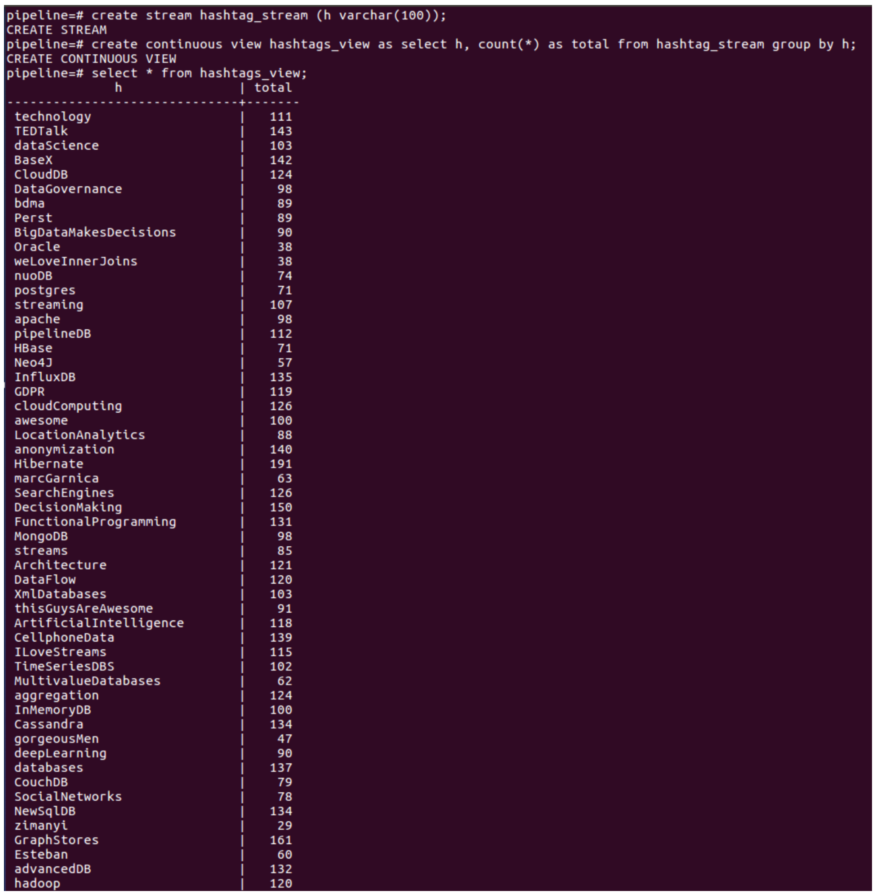

Stream Analysis (PipelineDB)
Hashtag Stream Data
Hashtag Stream
Hashtag Continuous View
View
select h, count(*) as total from hashtag_stream group by h

Figure 8: Hashtag use case data flow

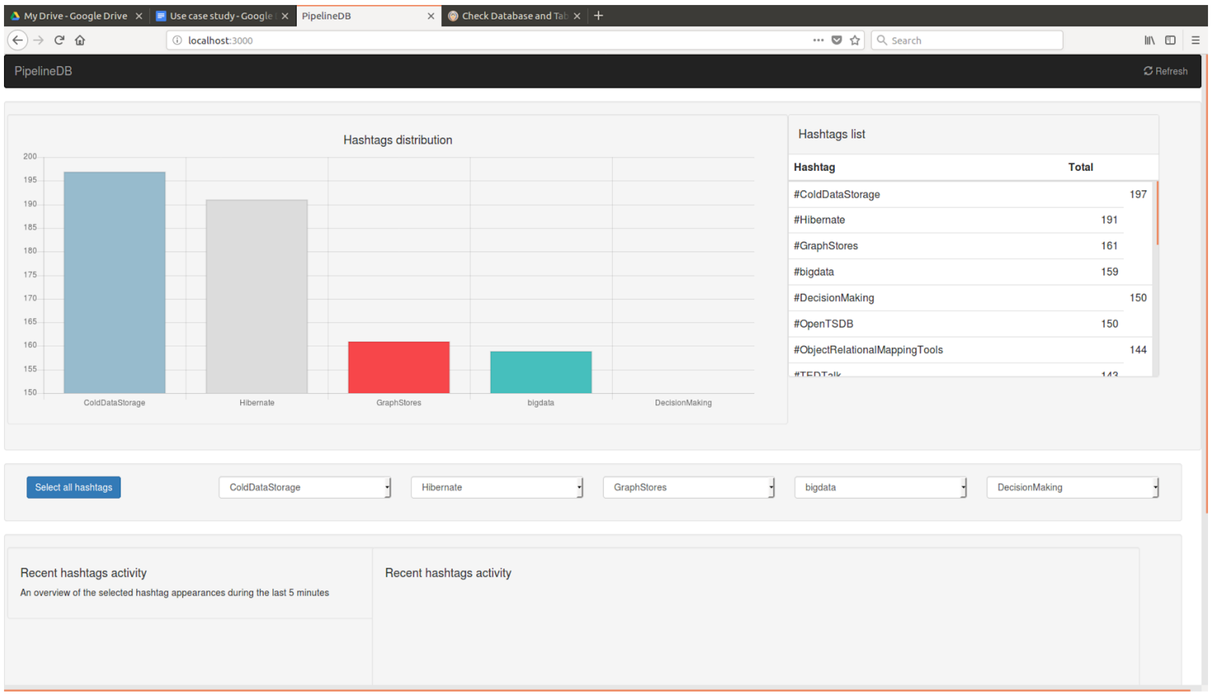
Every time a streaming data is inserted into stream, this continuous view is performing a counting on the hashtag name. After data ingestion this is how the continuous view looks with the data streams consumed:



```
pipeline=# create stream hashtag_stream (h varchar(100));
CREATE STREAM
pipeline=# create continuous view hashtags_view as select h, count(*) as total from hashtag_stream group by h;
CREATE CONTINUOUS VIEW
pipeline=# select * from hashtags_view;
              h               | total
------------------------------+-------
 technology                   |   111
 TEDTalk                      |   143
 dataScience                  |   103
 BaseX                        |   142
 CloudDB                      |   124
 DataGovernance               |    98
 bdma                         |    89
 Perst                        |    89
 BigDataMakesDecisions        |    90
 Oracle                       |    38
 weLoveInnerJoins             |    38
 nuoDB                        |    74
 postgres                     |    71
 streaming                    |   107
 apache                       |    98
 pipelineDB                   |   112
 HBase                        |    71
 Neo4J                        |    57
 InfluxDB                     |   135
 GDPR                         |   119
 cloudComputing               |   126
 awesome                      |   100
 LocationAnalytics            |    88
 anonymization                |   140
 Hibernate                    |   191
 marcGarnica                  |    63
 SearchEngines                |   126
 DecisionMaking               |   150
 FunctionalProgramming        |   131
 MongoDB                      |    98
 streams                      |    85
 Architecture                 |   121
 DataFlow                     |   120
 XmlDatabases                 |   103
 thisGuysAreAwesome           |    91
 ArtificialIntelligence       |   118
 CellphoneData                |   139
 ILoveStreams                 |   115
 TimeSeriesDBS                |   102
 MultivalueDatabases          |    62
 aggregation                  |   124
 InMemoryDB                   |   100
 Cassandra                    |   134
 gorgeousMen                  |    47
 deepLearning                 |    90
 databases                    |   137
 CouchDB                      |    79
 SocialNetworks               |    78
 NewSqlDB                     |   134
 zimanyi                      |    29
 GraphStores                  |   161
 Esteban                      |    60
 advancedDB                   |   132
 hadoop                       |   120
```

Figure 9: Data captured in hashtags_view

The data visualization tools implemented for this case are a dynamic bar chart and a complete list of hashtags with its number of appearance in the social network.



Hashtags list

Hashtag	Total
#ColdDataStorage	197
#Hibernate	191
#GraphStores	161
#bigdata	159
#DecisionMaking	150
#OpenTSDB	150
#ObjectRelationalMappingTools	144

Recent hashtags activity

An overview of the selected hashtag appearances during the last 5 minutes

Figure 10: Visualization of hashtags_view: Histogram and complete list of hashtags

Streaming Analysis with Sliding Windows

Secondly, one main objective of this implementation was also to test the Sliding Windows features of PipelineDB. By means of the where clause, PipelineDB is defining a moving in time window from where all the aggregations are computed. No other data is used apart from the included inside the boundaries of the window.

For this use case, the system is still using the same stream **hashtag_stream** presented before. This is also a proof of concept that with the same stream defined in the system, PipelineDB allows the users to consume the stream as many times and in as many different manners regarding the user visualizations and analytics requirements.

The statement to create the sliding windows view for hashtags is the following:

```
CREATE CONTINUOUS VIEW timing_hashtags WITH(sw = '5 minutes') AS
SELECT h, minute(arrival_timestamp) as minuteOfArrival, COUNT( * ) as quantity
FROM hashtag_stream GROUP BY h, minuteOfArrival;
```

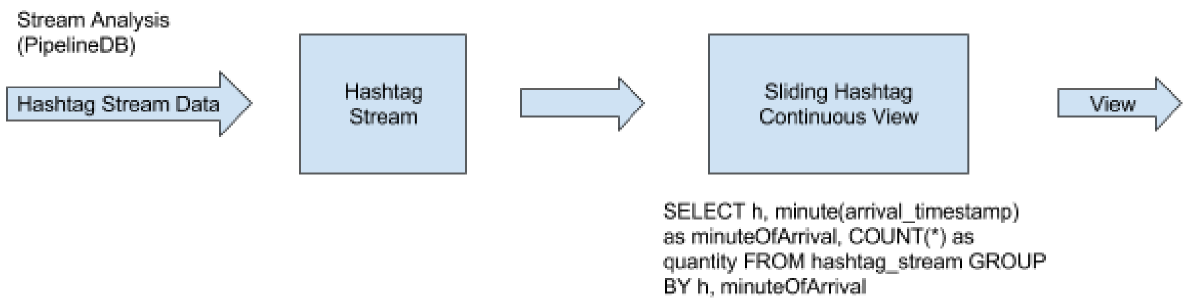
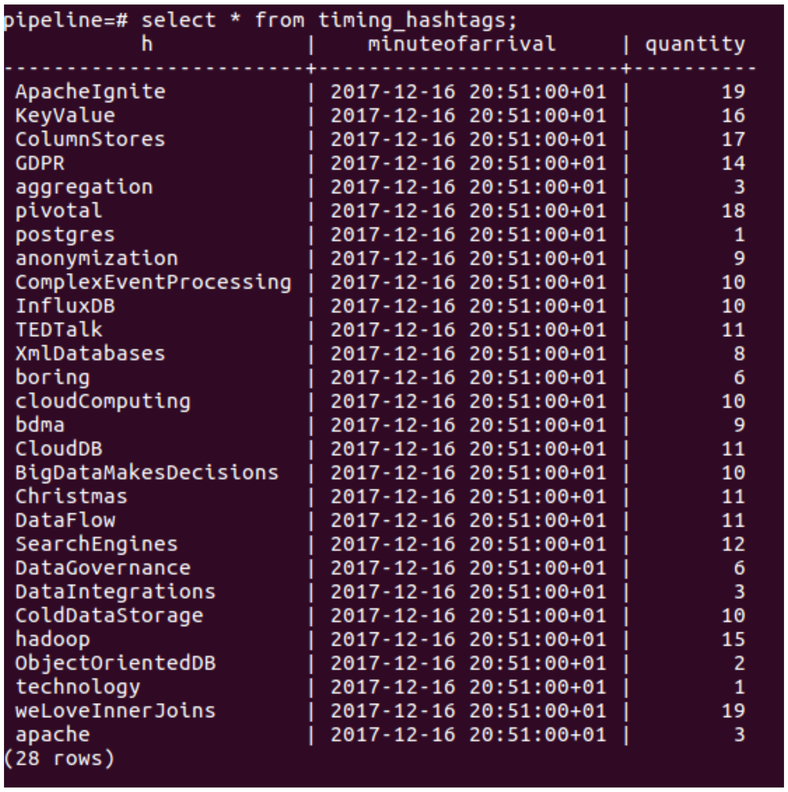

Stream Analysis (PipelineDB)
Hashtag Stream Data
Hashtag Stream
Sliding Hashtag Continuous View
View
SELECT h, minute(arrival_timestamp) as minuteOfArrival, COUNT(*) as quantity FROM hashtag_stream GROUP BY h, minuteOfArrival

Figure 11: Sliding hashtags use case data flow

This view is only looking at the window of time of 5 minutes before and is aggregate the number of hashtags appearance by hashtag name and minute of arrival. With these data, the following time series have been added to the data visualization.



```
pipeline=# select * from timing_hashtags;
          h            |      minuteofarrival      | quantity
-----------------------+------------------------+----------
 ApacheIgnite          | 2017-12-16 20:51:00+01 |       19
 KeyValue              | 2017-12-16 20:51:00+01 |       16
 ColumnStores          | 2017-12-16 20:51:00+01 |       17
 GDPR                  | 2017-12-16 20:51:00+01 |       14
 aggregation           | 2017-12-16 20:51:00+01 |        3
 pivotal               | 2017-12-16 20:51:00+01 |       18
 postgres              | 2017-12-16 20:51:00+01 |        1
 anonymization         | 2017-12-16 20:51:00+01 |        9
 ComplexEventProcessing | 2017-12-16 20:51:00+01 |       10
 InfluxDB              | 2017-12-16 20:51:00+01 |       10
 TEDTalk               | 2017-12-16 20:51:00+01 |       11
 XmlDatabases          | 2017-12-16 20:51:00+01 |        8
 boring                | 2017-12-16 20:51:00+01 |        6
 cloudComputing        | 2017-12-16 20:51:00+01 |       10
 bdma                  | 2017-12-16 20:51:00+01 |        9
 CloudDB               | 2017-12-16 20:51:00+01 |       11
 BigDataMakesDecisions | 2017-12-16 20:51:00+01 |       10
 Christmas             | 2017-12-16 20:51:00+01 |       11
 DataFlow              | 2017-12-16 20:51:00+01 |       11
 SearchEngines         | 2017-12-16 20:51:00+01 |       12
 DataGovernance        | 2017-12-16 20:51:00+01 |        6
 DataIntegrations      | 2017-12-16 20:51:00+01 |        3
 ColdDataStorage       | 2017-12-16 20:51:00+01 |       10
 hadoop                | 2017-12-16 20:51:00+01 |       15
 ObjectOrientedDB      | 2017-12-16 20:51:00+01 |        2
 technology            | 2017-12-16 20:51:00+01 |        1
 weLoveInnerJoins      | 2017-12-16 20:51:00+01 |       19
 apache                | 2017-12-16 20:51:00+01 |        3
(28 rows)
```

Figure 12: Data captured in timing_hashtags continuous view

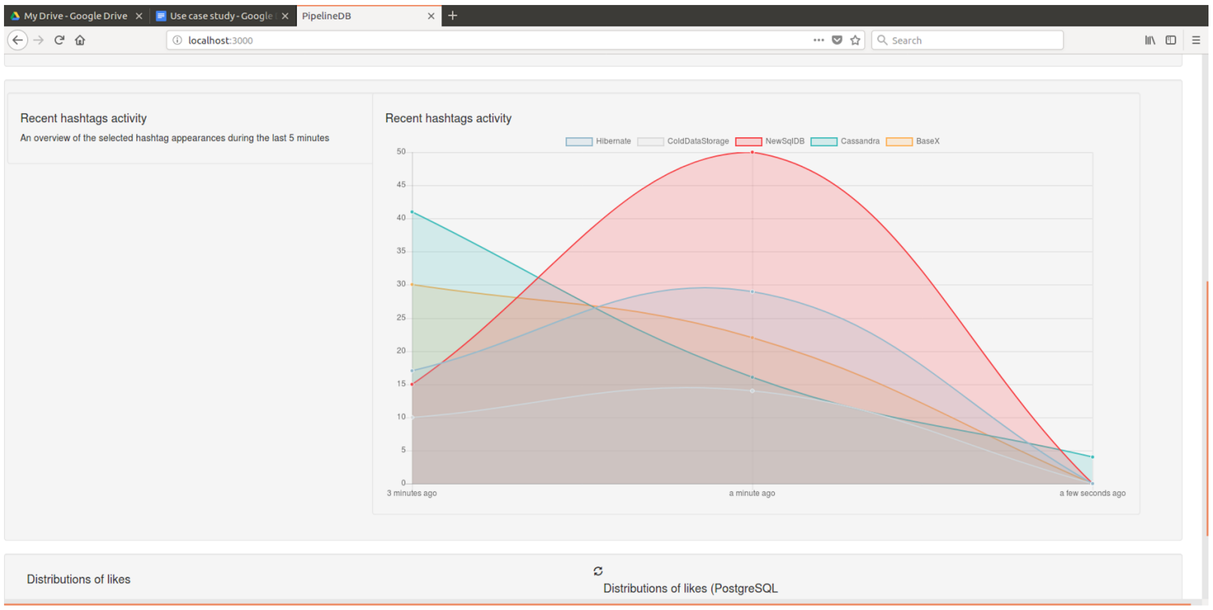
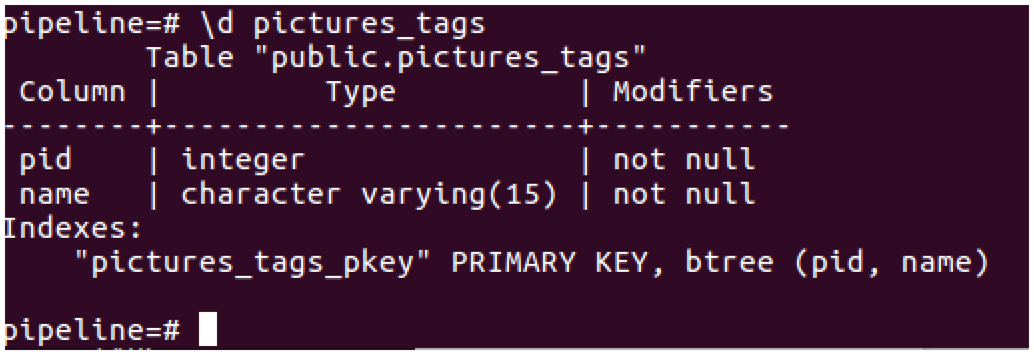


Figure 13: Data visualization for the sliding windows use case

Likes Use Case

In this particular use case, a complete and isolated reference table is already stored in the system with information about pictures tags.

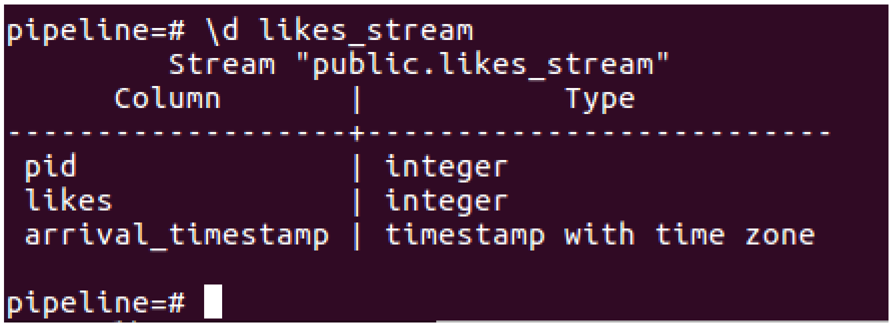


```
pipeline=# \d pictures_tags
        Table "public.pictures_tags"
 Column |         Type          | Modifiers
--------+-----------------------+-----------
 pid    | integer               | not null
 name   | character varying(15) | not null
Indexes:
    "pictures_tags_pkey" PRIMARY KEY, btree (pid, name)

pipeline=# 
```

Figure 14: Static reference table pictures_tags

Basically, this table contains pairs such as <photo_id, person_name> that means that person name is tagged in photo with id photo_id. When the system receives a like, it only receives the photo identifier and the number of likes, but to process this information and be able to account this like to all the people tagged in the photo some operations needs to be done.

There are four schemas that are used in the PipelineDB for this use case. Three of them are special schemas that are belong to PipelineDB; Stream, Continuous Transform and Continuous Table.

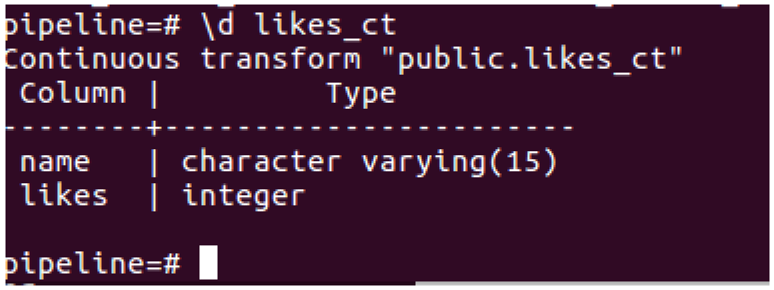


```
pipeline=# \d likes_stream
         Stream "public.likes_stream"
      Column       |           Type
-------------------+--------------------------
 pid               | integer
 likes             | integer
 arrival_timestamp | timestamp with time zone

pipeline=#
```

Figure 15: Likes_stream representing the incoming likes to the system

Likes streams are going to represent the likes incoming into the system. But as explained before, this likes cannot go directly to the visualization because the system does not know who is tagged in the picture at this moment. To know that, the system needs to look up for the photo id on the reference table pictures_tags (Figure 11) and get all the names of the people tagged in this photo. (See figure 15). This join between the incoming data and the reference table is efficiently done by the continuous transform feature of PipelineDB. The syntax goes as follows:

```
CREATE CONTINUOUS TRANSFORM likes_ct AS
SELECT t.name, l.likes
FROM likes_stream l JOIN pictures_tags t ON l.pid = t.pid;
```



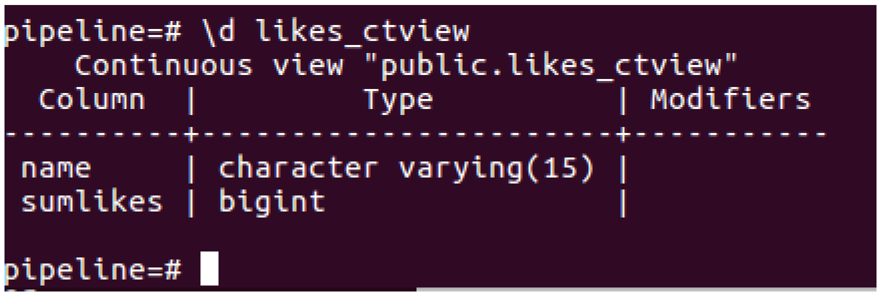
```
pipeline=# \d likes_ct
Continuous transform "public.likes_ct"
 Column |          Type
--------+-----------------------
 name   | character varying(15)
 likes  | integer

pipeline=#
```

Figure 16: Likes_continuous transform operation on the incoming likes

Likes_ct continuous transform is continuously joining the likes streams flowing into the system with the pictures tags table in order to output data streams indicating the likes per name. This output data streams then can be consumed by a continuous view as follows:

```
CREATE CONTINUOUS VIEW likes_ctView AS
SELECT name, sum(likes) as sumLikes
FROM output_of('likes_ct') GROUP BY name;
```

The from statement of this continuous view is directly linked with the streams form the output channel of likes_ct continuous transforms.



```
pipeline=# \d likes_ctview
   Continuous view "public.likes_ctview"
 Column   |          Type          | Modifiers
----------+------------------------+----------
 name     | character varying(15) |
 sumlikes | bigint                 |

pipeline=#
```

Figure 17: Likes continues views

Apart from the Picture Tags table that is stored in PostgreSQL, only the Continuous View is stored in PipelineDB where it stores the results of the desired analysis queries, such as aggregation etc. Continuous View is updated automatically every time when an insertion into Stream occurs and this update is a fast process. Selection of rows from Likes Stream is not possible since it is just an intermediate table that only pushes data to Continuous Transform and selection from Continuous Transform is also not possible, since it only works for joining data that is coming from Likes Stream and passing it into Continuous Views.

The complete data flow is shown in the following figure.

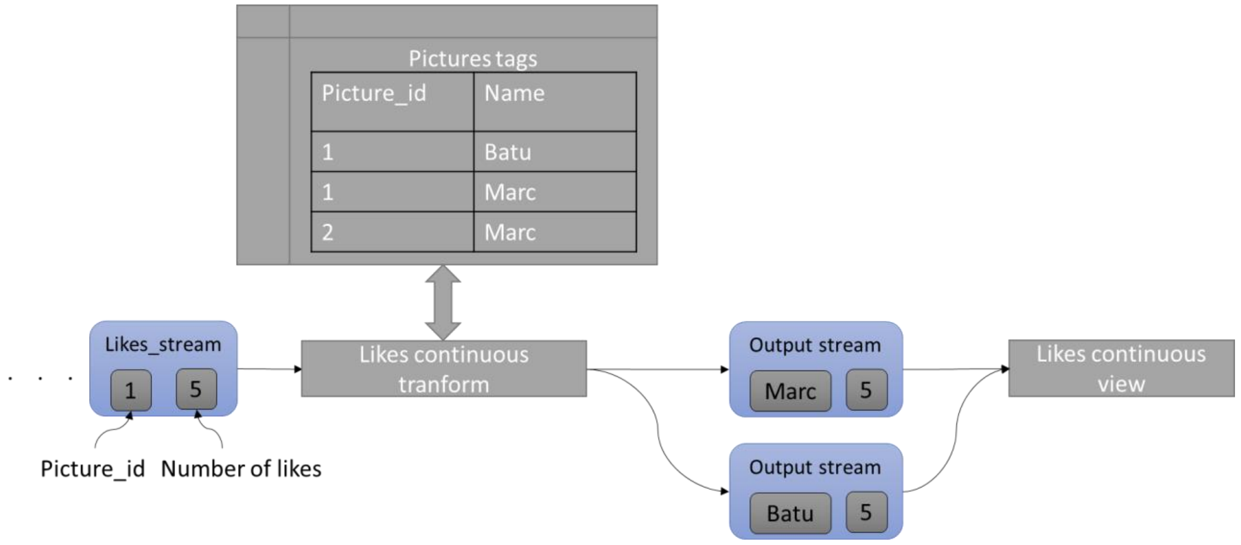


Figure 18: Likes streaming transformation

For this use case, random likes are created with a bash process, in the form of (PhotoID, # of likes), to simulate a streaming data and the created data is inserted into Likes Stream instantly. Picture Tags Table is stored in PostgreSQL, where the information is being kept in the form of (PhotoID, Name of the person who is tagged in the photo). Every time a like data is inserted into Likes Stream, Likes Continuous Transform performs a lookup on Picture Tags table and selects Name with the # of likes that person receives. Continuous View, calculates the total number of likes each person has by aggregating on the name of the person. This is a simplified case analysis to show the capabilities of the PipelineDB, whereas more complex lookups and joins could also be done.

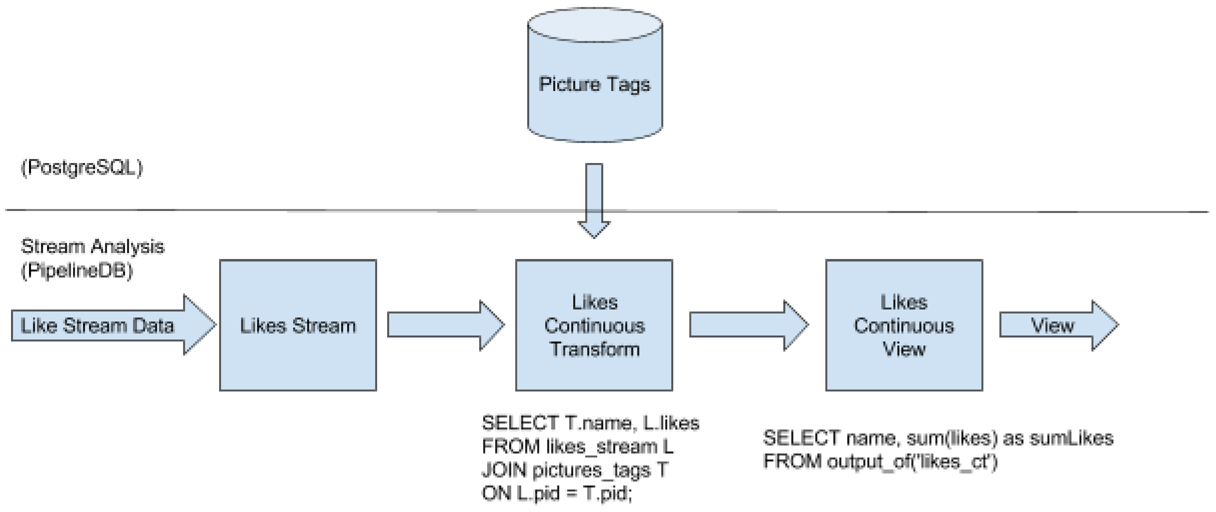
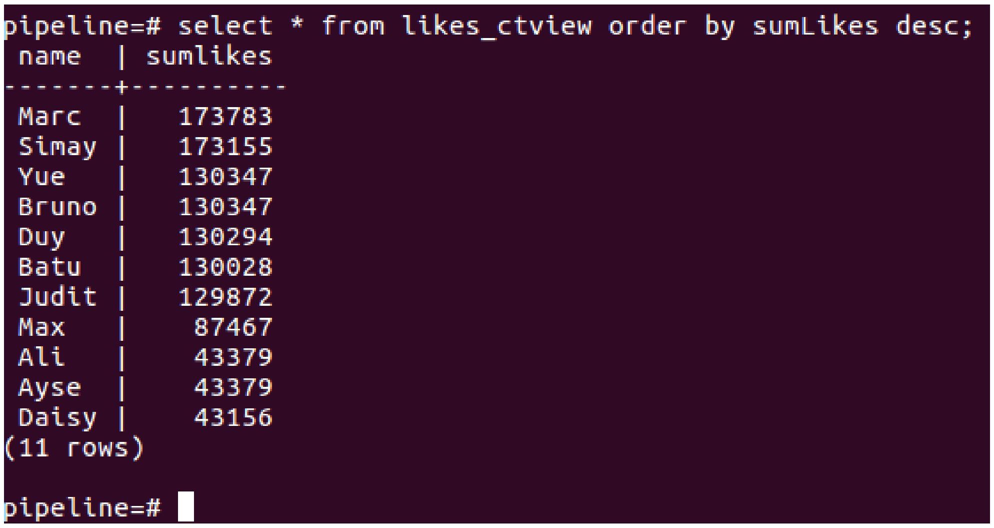

Picture Tags
(PostgreSQL)
Stream Analysis (PipelineDB)
Like Stream Data
Likes Stream
Likes Continuous Transform
Likes Continuous View
View
SELECT T.name, L.likes FROM likes_stream L JOIN pictures_tags T ON L.pid = T.pid;
SELECT name, sum(likes) as sumLikes FROM output_of('likes_ct')

Figure 19: Likes use case data flow

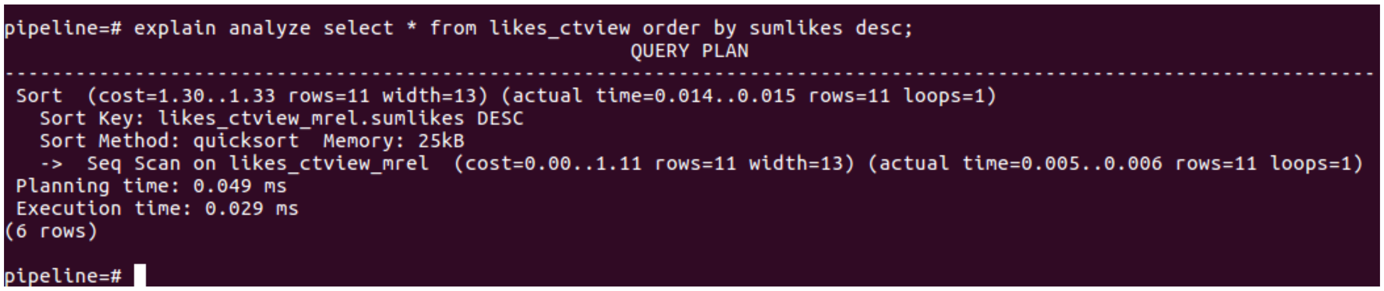
When the Continuous View is updated automatically after an insertion is done into Stream, only a simple select query is needed to view the data that is kept in the Continuous View.



```
pipeline=# select * from likes_ctview order by sumLikes desc;
 name   | sumlikes
-------+----------
 Marc   |   173783
 Simay |   173155
 Yue    |   130347
 Bruno |   130347
 Duy    |   130294
 Batu   |   130028
 Judit |   129872
 Max    |    87467
 Ali    |    43379
 Ayse   |    43379
 Daisy |    43156
(11 rows)

pipeline=# 
```

Figure 20: Likes continuous view with data

Querying the results in the Continuous View is similar to querying a Materialized View in standard SQL, it is very fast and efficient. To see the query execution plan and execution time of the above query, PostgreSQL's standart "Explain Analyze" expression can be added to the beginning of the query;



```
pipeline=# explain analyze select * from likes_ctview order by sumlikes desc;
                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=1.30..1.33 rows=11 width=13) (actual time=0.014..0.015 rows=11 loops=1)
   Sort Key: likes_ctview_mrel.sumlikes DESC
   Sort Method: quicksort  Memory: 25kB
   ->  Seq Scan on likes_ctview_mrel  (cost=0.00..1.11 rows=11 width=13) (actual time=0.005..0.006 rows=11 loops=1)
 Planning time: 0.049 ms
 Execution time: 0.029 ms
(6 rows)

pipeline=# 
```

Selection from Continuous View is a fast process since the Continuous View is updated already automatically; the execution time is 0.029 ms and the cost is 1.33. Again, it is important to note that, only the Continuous View is kept stored in PipelineDB and it only stores the results of queries known a-priori.

PERFORMANCE ANALYSIS: PIPELINEDB VS POSTGRESQL

Once implemented and analysed the different features of PipelineDB, the objectives of this section is to benchmark its performance with normal PostgreSQL installations. In order to compare both performances and characteristics Likes use case has been selected since it has Join operations with the table that is present in PostgreSQL Database.

A parallel use case has been implemented using the same streaming data source than Likes use case but storing every stream in a Relational Table called Likes_table. After this, this relational table can be exploited to get the same results as the likes continuous view explained in the previous section. To mimic the same results as the continuous views on PipelineDB, two main options have been selected:

- Simply implement a normal SQL view to encapsulate the query and then simply query this view. This view will need to compute the results every time they are needed.
- Implement a materialized view that joins the data from the reference pictures_tags table and the likes_table previously mentioned. The concept is mainly the same but in this case the view is physically stored in the system.

Consequently, Likes Table is very similar to PipelineDB's Stream and Materialized View is very similar to PipelineDB's Continuous View, in terms of their schema.

Table 2: PipelineDB and PostgreSQL declarations for the comparison

PipelineDB declarations	PostgreSQL declarations	

`CREATE STREAM likes_stream`
`(pid integer, likes integer);`
	`CREATE TABLE likes_table(pid integer, likes integer);`	

`CREATE CONTINUOUS TRANSFORM likes_ct`
`AS`
`SELECT t.name, l.likes`
`FROM likes_stream l`
`JOIN pictures_tags t`
`ON l.pid = t.pid;`
	`CREATE VIEW likes_view`
	`AS`
	`SELECT name, sum(likes)`
	`FROM likes_table l`
	`JOIN pictures_tags t`
	`ON I.pid = t.pid`
	`GROUP BY name;`
		`CREATE MATERIALIZED VIEW`
		`likes_mtView`
		`AS`
		`SELECT name, sum(likes)`
		`as sumLikes`
		`FROM likes_table l`
		`JOIN pictures_tags t`
		`ON l.pid = t.pid GROUP BY name;`

`CREATE CONTINUOUS VIEW likes_ctView`
`AS`
`SELECT name, sum(likes) as sumLikes`
`FROM output_of('likes_ct')`
`GROUP BY name;`		

Comparing PipelineDB with Views on PostgreSQL

As introduced before, the main conceptual difference is that in the traditional SQL Analysis the database does not keep the results of the view in the system, but it computes them every time the view is asked. In the Stream Analysis the continuous transform will be operating on the streams flowing into the system and output them towards the continuous view.

On the other side, in the Traditional SQL analysis the streams will represent inserts in a table and then the view will perform the join on every execution.

Views on PostgreSQL are just encapsulations of SQL queries to fix the queries and structure the analysis of the data. The main difference with PipelineDB in this case is that while the Continuous Views are updated on the fly and consequently their time to access is fast and constant, Views need to be computed every time they are asked which add a computation cost that can be clearly noticed in the following graphs.

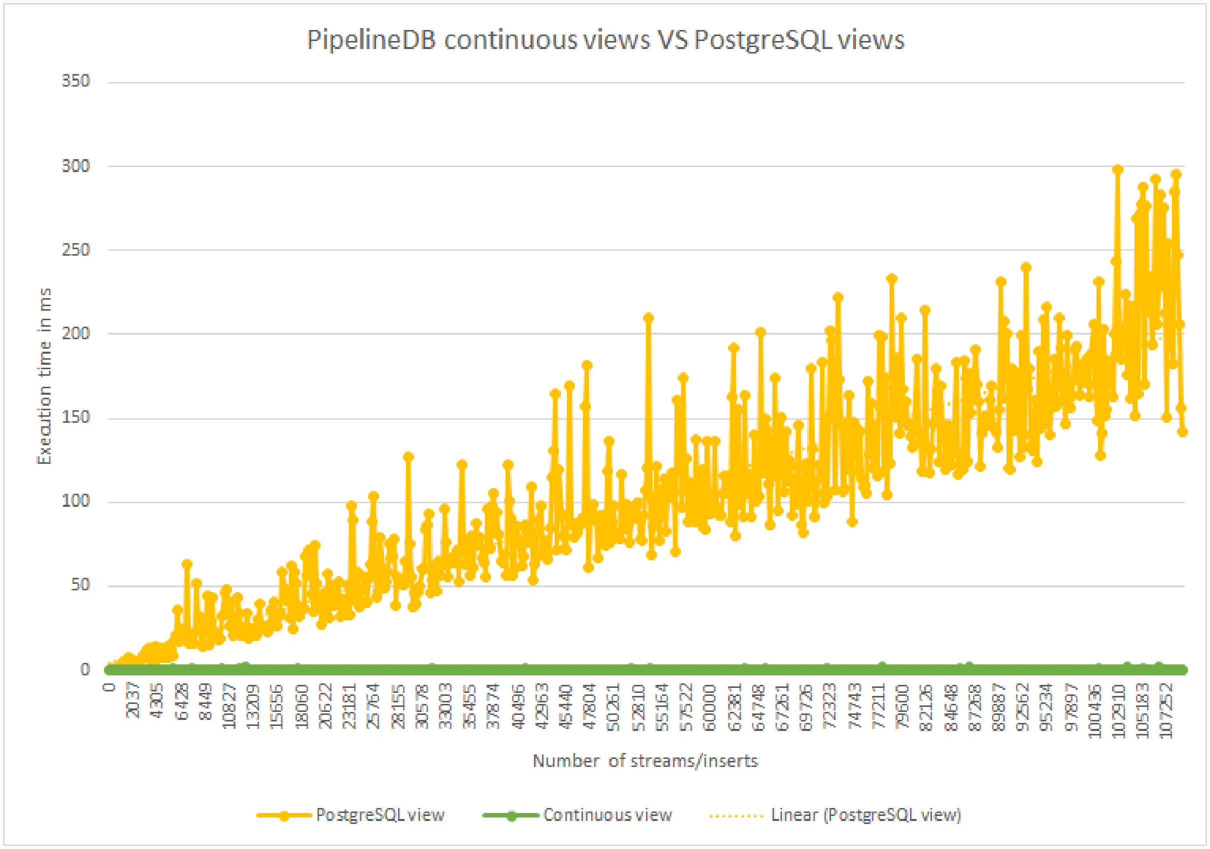


Figure 21: Execution times PostgreSQL views vs PipelineDB continuous views

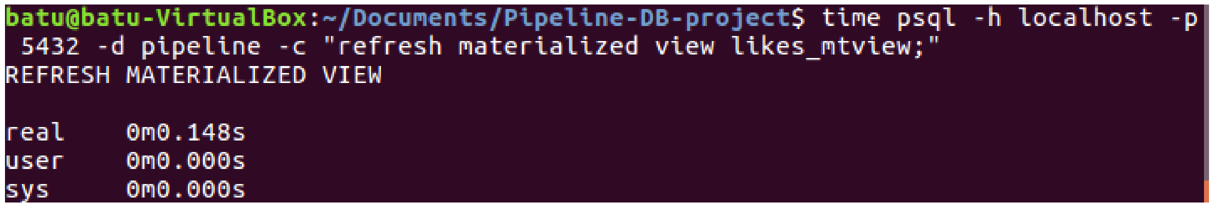
As shown, the time to get the results from a continuous view (green label) is constant and does not depend on the number of events of the system. This is because, the streams are consumed on the fly and they update directly the continuous view, more precisely, is the continuous transform implemented that operates on each stream to perform the join with the reference table. On the other side, yellow label represents the execution time of selecting the views in Traditional PostgreSQL. Clearly, the time increases on the number of likes received, this comes from the fact that every time the view needs to be shown, a join is performed between the tables containing the likes and the table containing the pictures tags.

Comparing PipelineDB vs PostgreSQL with Materialized views

As it is shown on the figure, on the traditional SQL analysis the incoming data is translated into INSERTs in the likes table. The materialized view is the one in charge of performing the join between the likes and pictures tags and show the desired output to the user. The materialized view can solve the problem shown in the previous analysis where the view needed to compute the result at every execution. Materialized views aim to store the result and only refresh its content in specific circumstances.

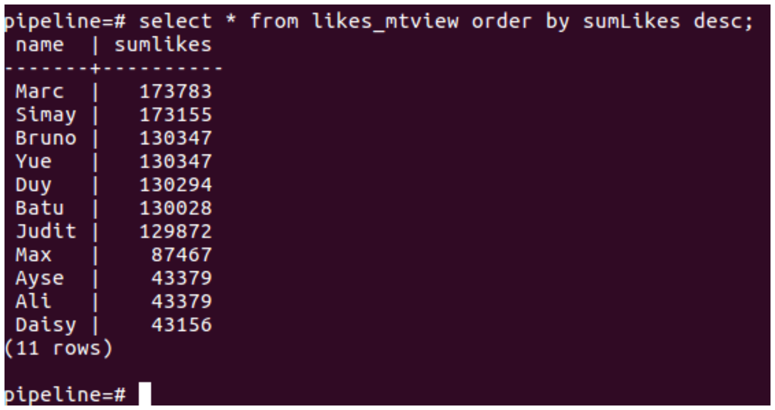
In contrast, and explained in previous sections of the study, the Stream analysis is mainly inserting the incoming data into streams and then consumed by a continuous transform where the join with pictures_tags is performed. The output of this continuous transforms are streams that as well are consumed by the Likes continuous view to show the desired result to the user.

To have the same views in both Analysis, the same query that is applied to Likes Continuous View is going to be applied on Likes Materialized View first, a refresh on the Materialized View is necessary. This refresh will be discussed in the next part as the main downside of doing such analysis on PostgreSQL:



```
batu@batu-VirtualBox:~/Documents/Pipeline-DB-project$ time psql -h localhost -p
 5432 -d pipeline -c "refresh materialized view likes_mtview;"
REFRESH MATERIALIZED VIEW

real	0m0.148s
user	0m0.000s
sys	0m0.000s
```

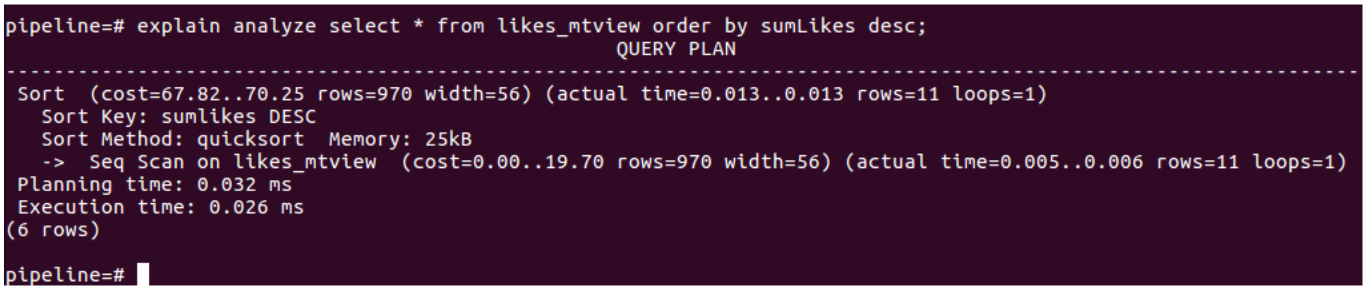
Figure 22: Execution time of refreshing the materialized view



```
pipeline=# select * from likes_mtview order by sumLikes desc;
 name  | sumlikes
-------+----------
 Marc  |   173783
 Simay |   173155
 Bruno |   130347
 Yue   |   130347
 Duy   |   130294
 Batu  |   130028
 Judit |   129872
 Max   |    87467
 Ayse  |    43379
 Ali   |    43379
 Daisy |    43156
(11 rows)

pipeline=#
```

Figure 23: Materialized view results

Above query can also be analysed:

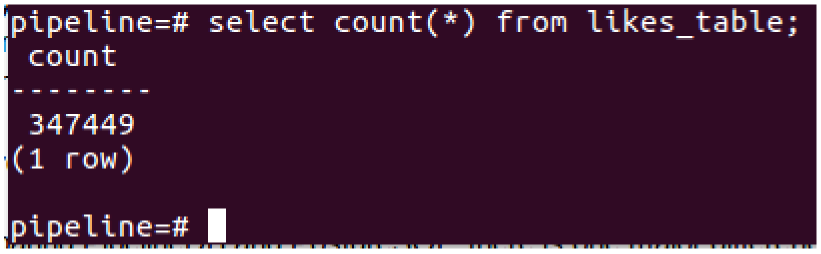


```
pipeline=# explain analyze select * from likes_mtview order by sumLikes desc;
                                                  QUERY PLAN
--------------------------------------------------------------------------------------------------------------------
 Sort  (cost=67.82..70.25 rows=970 width=56) (actual time=0.013..0.013 rows=11 loops=1)
   Sort Key: sumlikes DESC
   Sort Method: quicksort  Memory: 25kB
   ->  Seq Scan on likes_mtview  (cost=0.00..19.70 rows=970 width=56) (actual time=0.005..0.006 rows=11 loops=1)
 Planning time: 0.032 ms
 Execution time: 0.026 ms
(6 rows)

pipeline=#
```

Figure 24: Execution plan and time of querying the materialized view

The execution time of this Traditional SQL Analysis is 0.026 ms and the cost is 70.25, where the previous execution time of Stream Analysis was 0.029 ms with the cost of 1.33. Since above query is running on a Materialized View, it's execution time is very similar to the execution time of the query of Continuous View but the cost much more since we are doing a sort on a table.

When comparing PipelineDB and PostgreSQL, there is one major difference that must be underlined; PipelineDB stores no data compared to PostgreSQL. Since PipelineDB is a stream analysis database, it does not store streaming data and in order to do the same analysis on PostgreSQL, streaming data has to be kept in a table:



```
pipeline=# select count(*) from likes_table;
 count
--------
 347449
(1 row)

pipeline=# 
```

Figure 25: Size in rows of the Relational table ingesting all the likes flowing into the system

Regarding the materialized view analysis, both entities, materialized views and continuous views have a relatively small and constant cost of query. Despite that, Continuous View automatically updates itself every time a stream data is inserted into stream, Materialized View is not capable of updating itself after every insertion. A possible solution to this problem can be creating a trigger on the likes_table to update Materialized View every time there is an insertion but since this update is going to be working when there is already another stream insertion into the table occurs, the trigger is going to refresh the materialized view when a refresh is already in progress. Since this is not a desired option, Materialized View needs to be refreshed before every view and the time that this refresh takes is proportional to the size of the table since the refresh operation completely replaces the contents of a materialized view.

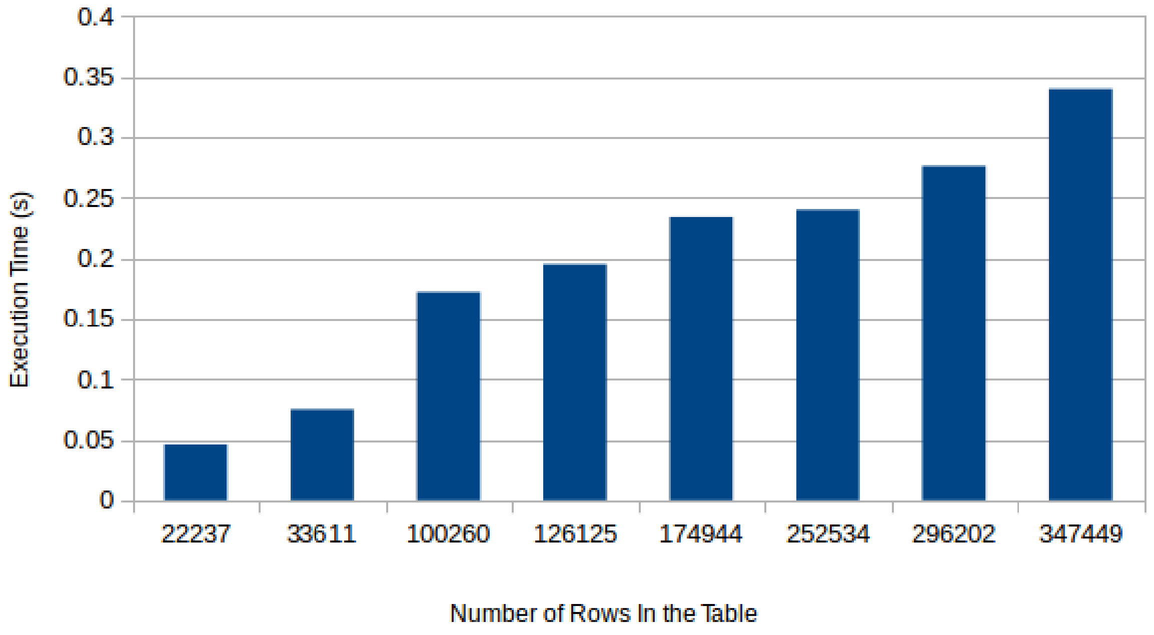


Figure 26: Analysis of the execution time of refreshing the materialized view by number of rows in Likes_table

The figure clearly shows that when table has more entries, the execution time of the update of materialized view increases proportionally. Reason why the execution time does not take too long is because the data in likes_table is very small (kilobytes) since our streaming data generator creates very small data. When the streaming data is bigger, the refresh time of Materialized View is expected to be bigger as well.

To conclude; PipelineDB is not a data warehouse, it is designed to be working with continuous queries where the queries are known a priori. While the output of continuous queries may be explored in an ad-hoc fashion, all of the raw data that has ever passed through PipelineDB are discarded after they've been read. PipelineDB can work on analysis that can be expressed with SQL. To do the such analysis by using traditional PostgreSQL, a data table needs to be kept and a materialized view needs to be refreshed to see the current state of the table before every view.

DATA VISUALIZATION

Most of the analytics systems are mainly used by means of nice and fashion data visualizations. Even is not explicitly included in the scope of this project, it was decided to build an efficient data visualization module to exploit the data and show the features of PipelineDB in a user-friendly manner.

Figure illustrates the schema defined for the data visualization module.

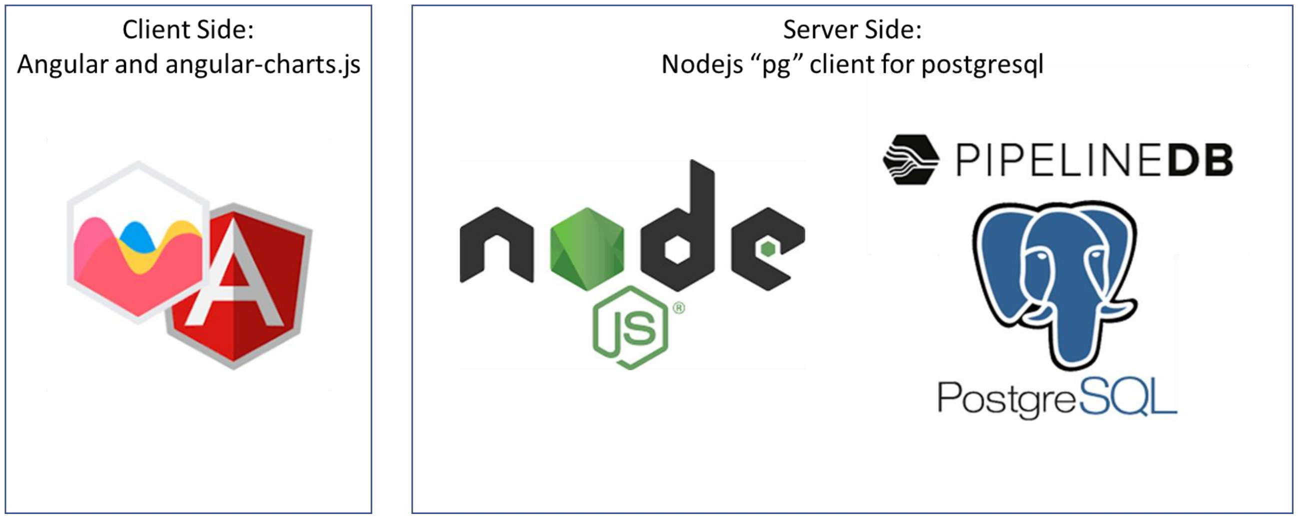

Client Side:
Angular and angular-charts.js
Server Side:
Nodejs "pg" client for postgresql
node JS
PIPELINEDB
PostgreSQL

Figure 27: Main schema of the technologies used by the data visualization

It is important to mention the difference between a streaming database and a real-time interface. As mentioned in previous chapters the database content is updated in real time by means of the streaming processing implemented with PipelineDB. However, this does not imply directly that the interface on the client side is showing the last update of the content of the database. For that, out of topic technologies are needed to connect the data model of the client side with the data stored in the database, an example can be the implementation of continuous pool between Nodejs and PostgreSQL or sockets that keeps both data models synchronized. Further versions of this project should consider these possibilities to propagate the change of the database to the interface in real time.

Conclusion

In this project, Stream Databases are examined via PipelineDB and its features. PipelineDB is a new Streaming Database which is an extension of PostgreSQL and it allows to use SQL for the analysis. PipelineDB offers fast and high throughput data analysis without storing the stream data on any table and its internal view, called continuous view, can update itself regularly, which yields a fast query analysis. Internal features of PipelineDB are examined within two use-cases; Hashtags and Likes, to demonstrate some basic capabilities on today's most popular social media data concepts. These use-cases analyses are then compared with traditional PostgreSQL data analysis, to show how tedious to do a streaming analysis by using tables and views.

The conclusion of this study will be conducted by two main discussions. First of all, a SWOT analysis has been performed on PipelineDB to evaluate its main features, and then a final conclusion has been extracted from Streaming Databases paradigm.

SWOT ANALYSIS ON PIPELINEDB

Table 3: SWOT analysis on PipelineDB

Strengths	Weaknesses
• Streaming processing on SQL. • Easy installation and setup. • Familiar and easy data model. • Postgres full compatibility (PostGis, temporal databases, pg clients..)	• Small community • Young tool (version 0.9.*) • Queries defined a priori.
Opportunities	**Threats**
• PostgreSQL official extension.	• SQL is the limit

STREAMING DATABASES OVERVIEW CONCLUSION

As an important conclusion of this study, it has been shown that the data-in-motion paradigm is representing a high number of data types from the real world: Sports analytics, health applications, social media, sensors... all this data sources are continuous by definition, consequently DBMS needs to adapt to the nature of this data.

In detail, new waves of technologies and paradigms cannot ignore the tools that are currently active and used by the users. People do not like changes, that’s why Streaming processors using SQL environments will still be needed. The trust, and familiarity on SQL by most of the users can speed up the spread of streaming paradigms all over the sectors. PipelineDB has this on his main business strategy and that’s why the tool is going to be released as an official extension of PostgreSQL soon.

Regarding the project objectives presented at the beginning of this document, a general and accurate description of Streaming Databases has been document through the study and a proof of concept has been implemented by means of PipelineDB. Furthermore, PostgreSQL has been used as a comparison to highlight the main advantages of choosing PipelineDB for Stream Analysis purpose.

References

[1] R. Allen, "Internet Marketing Statistics," 6 February 2017. [Online]. Available: https://www.smartinsights.com/internet-marketing-statistics/happens-online-60-seconds/attachment/what-happens-online-in-60-seconds/.

[2] PipelineDB, 2017. [Online]. Available: https://www.pipelinedb.com/.

[3] "PipelineDB Docs," 2017. [Online]. Available: http://docs.pipelinedb.com/.

[4] "What is Stream Processing," Data Artisans, 2017. [Online]. Available: https://data-artisans.com/what-is-stream-processing.

[5] K. Paramasivam, "Stream Processing Hard Problems Part II: Data Access," LinkedIn, 22 August 2016. [Online]. Available: https://engineering.linkedin.com/blog/2016/08/stream-processing-hard-problems-part-ii--data-access .

[6] Fabian Hueske, Shaoxuan Wang, and Xiaowei Jiang, "Continuous Queries on Dynamic Tables," Apache Flink, 4 April 2017. [Online]. Available: https://flink.apache.org/news/2017/04/04/dynamic-tables.html.

[7] "APACHE FLINK: SCALABLE STREAM AND BATCH DATA PROCESSING," Apache Flink, [Online]. Available: https://flink.apache.org/.

[8] H. Freeman, "Streaming Analytics 101: The What, Why, and How," Data Versity, 26 April 2016. [Online]. Available: http://www.dataversity.net/streaming-analytics-101/.

[9] "Heron's Design Goals," Heron, [Online]. Available: https://twitter.github.io/heron/docs/concepts/design-goals/.

[10] "APACHE STORM," Hortonworks, [Online]. Available: https://hortonworks.com/apache/storm/ .

[11] K. Wähner, "Real-Time Stream Processing as Game Changer in a Big Data World with Hadoop and Data Warehouse," 10 September 2014. [Online]. Available: https://www.infoq.com/articles/stream-processing-hadoop .

[12] J. Touffe-Blin, "Angular Chart," [Online]. Available: http://jtblin.github.io/angular-chart.js/.

[13] J. Kreps, "Introducing Kafka Streams: Stream Processing Made Simple," Confluent, 10 March 2016. [Online]. Available: https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/.

[14] "Newest "PipelineDB" Questions," PipelineDB, [Online]. Available: https://stackoverflow.com/questions/tagged/pipelinedb?page=2&sort=newest&pagesize=15.

[15] C. TOZZI, "Big Data 101: Dummy’s Guide to Batch vs. Streaming Data," 25 July 2017. [Online]. Available: http://blog.syncsort.com/2017/07/big-data/big-data-101-batch-stream-processing/.

[16] A. Yemelianov, "PipelineDB: Working with Data Streams," Selectel, 16 May 2017. [Online]. Available: https://blog.selectel.com/pipelinedb-working-data-streams/.