Horizon 2020

Understanding Europe's Fashion Data Universe

Data Integration Solution

Deliverable number: D2.3

Version 2.0

Funded by the European Union's Horizon 2020 research and innovation programme under Grant Agreement No. 732328

Project Acronym:	FashionBrain
Project Full Title:	Understanding Europe's Fashion Data Universe
Call:	H2020-ICT-2016-1
Topic:	ICT-14-2016-2017, Big Data PPP: Cross-sectorial and cross-lingual data integration and experimentation
Project URL:	https://fashionbrain-project.eu

Deliverable type	Other (O)
Dissemination level	Public (PU)
Contractual Delivery Date	31 December 2018
Resubmission Delivery Date	4 February 2019
Number of pages	40, the last one being no. 34
Authors	Ying Zhang, Pedro E. S. Ferreira, Svetlin Stalinov, Aris Koning, Martin Kersten - MDBS Torsten Kilias, Alexander Löser - BEUTH Roland Vollgraf - Zalando
Peer review	Ines Arous, Mourad Khayati - UNIFR Alan Akbik - Zalando

Change Log

Version	Date	Status	Partner	Remarks
0.1	14/11/2018	Draft	MDBS	
0.2	15/12/2018	Full Draft	MDBS, BEUTH, Zalando	
1.0	20/12/2018	Final	MDBS, BEUTH, Zalando, UNIFR	Rejected 30/01/2019
2.0	04/02/2019	Resumitted Final	MDBS, BEUTH, Zalando, UNIFR	

Deliverable Description

D2.3 Data integration solution (M24). A MonetDB data integration solution for modelling and storing i) the different available datasets from all partners, ii) the taxonomy, iii) the extracted named entities and links. The deliverable will also include the extension of MonetDB with JSON support to include the management of semi-structured data. The proposed solution will be used in WP4, WP5, and WP6.

Abstract

In this deliverable, we report the work we have done under the context of *T2.3 "infrastructures for scalable cross-domain data integration and management"* which has resulted in the design and implementation of the MonetDB-based FashionBrain Integrated Architecture (FaBIAM) for storing, managing and processing heterogeneous fashion data (i.e. structured relational and unstructured text data).

First we present the architecture of FaBIAM. Then we highlight several main components of FaBIAM. Finally, we show FaBIAM can be used to support fashion time series data use case as defined in D1.2 [3].

Table of Contents

List of Figures

List of Tables

List of Acronyms and Abbreviations

ACID	Atomicity, Consistency, Isolation and Durability
BigComp 2019	6th IEEE International Conference on Big Data and Smart Computing
CD	Centroid Decomposition
COLING 2018	27th International Conference on Computational Linguistics
CQE	Continuous Query Engine
CSV	Comma Separated Values
DBMS	Database Management System
FaBIAM	FashionBrain Integrated Architecture
ICDE 2019	35th IEEE International Conference on Data Engineering
IDEL	In-Database Entity Linking
IoT	Internet of Things
JSON	JavaScript Object Notation
MAL	MonetDB Assembly Language
NLP	Natural Language Processing
OLAP	Online Analytical Processing
RDBMS	Relational Database Management System
SQL	Structured Query Language
UDF	User Defined Functions
UDP	User Defined Procedure

1. Introduction

The FashionBrain project targets at consolidating and extending existing European technologies in the area of database management, data mining, machine learning, image processing, information retrieval, and crowd sourcing to strengthen the positions of European (fashion) retailers among their world-wide competitors.

For fashion retailers, the ability to efficiently extract, store, manage and analyse information from heterogeneous data sources, including structured data (e.g. product catalogues and sales information), unstructured data (e.g. twitter, blogs and customer reviews), and binary (multimedia) data (e.g. YouTube and Instagram), is business critical. Therefore, in work package *WP2 "Semantic Data Integration for Fashion Industry"*, one of the main objectives is to

> "Design and deploy novel big data infrastructure to support scalable multi-source data federation, and implement efficient analysis primitives at the core of the data management solution."

This objective is addressed by task *T2.3 "Infrastructures for scalable cross-domain data integration and management"*, which focus on the implementation of cross-domain integration facilities to support advanced (fashion) text and time series analysis.

The work we have done in the context of T2.3 is being reported now in two deliverables both due in M24 (Dec2018). In the sibling deliverable *D2.4 "Time Series Operators for MonetDB"*, we report the new time series analysis techniques and operators we have added into the analytical Relational Database Management System (RDBMS) MonetDB[1].

In this deliverable, we detail the design and implementation of FaBIAM, a MonetDB-based architecture for storing, managing and analysing of both structured and unstructured data, which has a three layer structure:

- At the bottom, the *data ingestion layer* supports ingestion and storage of both structured (i.e. Comma Separated Values (CSV)) and unstructured (i.e. JavaScript Object Notation (JSON)) data.
- In the middle, the *processing layer* supports advanced Structured Query Language (SQL) window functions, in-database machine learning and continuous queries. These features have been added into the MonetDB kernel in the context of T2.3.

[1] https://www.monetdb.org

- At the top, the *analysis layer* has integrated tools provided by FashionBrain partners for advanced time series analysis (with partner UNIFR) and text data processing (with partners BEUTH and Zalando).

1.1. Scope of this Deliverable

The design of FaBIAM has been guided by the business scenarios identified in *D1.2 "Requirement analysis document"*, in particular:

- Scenario 3: Brand Monitoring for Internal Stakeholders
  - Challenge 7: Online Analytical Processing (OLAP) Queries over Text- and Catalogue Data
- Scenario 4: Fashion Trends Analysis
  - Challenge 8: Textual Time Trails
  - Challenge 9: Time Series Analysis

In FaBIAM the following work with partners has been integrated:

- RecovDB [2]: an advanced time series missing value recovery system using MonetDB and centroid decomposition [4]. This work is reported in details in the sibling deliverable *D2.4 "Time Series Operators for MonetDB"* and has been submitted to 35th IEEE International Conference on Data Engineering (ICDE 2019). This work is in collaboration with partner UNIFR.
- In-Database Entity Linking (IDEL) [5]: an entity linking system for both text data and relational records based on neural embeddings. This work was reported in details in deliverables *D4.1 "Report on text joins"* and *D4.2 "Demo on text joins"* and is to appear in 6th IEEE International Conference on Big Data and Smart Computing (BigComp 2019). This work is in collaboration with partner BEUTH.
- FLAIR[1]: a Natural Language Processing (NLP) library[2] based on word and document embeddings. This work was reported in details in deliverable *D6.2 "Entity linkage data model"* and was published in 27th International Conference on Computational Linguistics (COLING 2018). This work is in collaboration with partner Zalando.

For the work reported in this deliverable, we have used all data sets that have been provided by project partners for FashionBrain as described in deliverable *D2.2 "Requirement analysis document WP2"*.

The remainder of this deliverable is as follows. In Chapter 2, we describe the architecture of FaBIAM. We first give an overview of the architecture in Section 2.1. Then, we zoom in into several individual components that have not been covered by other deliverables, including MonetDB's support for JSON data as a native data

[2] https://github.com/zalandoresearch/flair

type (Section 2.2), streaming data and continuous query processing (Section 2.3), and text data analysis with machine learning (Section 2.4). In Chapter 3, we present the implementation of a fashion use case to demonstrate how FaBIAM can be used to process, analyse and store a stream of reviews posted by Zalando customers. Finally, in Chapter 4 we conclude with outlook for future work.

2. FaBIAM

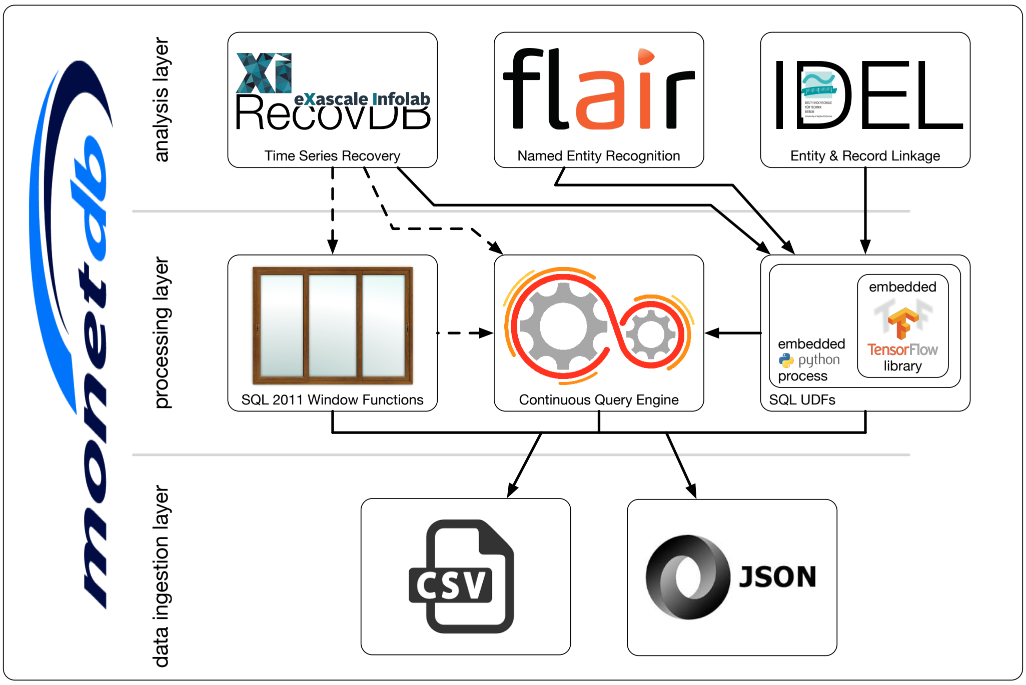

Figure 2.1: Architecture of the FashionBrain integrated architecture (FaBIAM).

2.1. Architecture Overview

Figure 2.1 shows an overview of FaBIAM. All components are integrated into the kernel of MonetDB. The solid arrows indicate the components that can already work together, while the dashed arrows indicating future integration. From bottom to top, they are divided into three layers:

Data Ingestion Layer This layer at the bottom of the MonetDB kernel provides various features for loading data into MonetDB. In the fashion world, there are three major groups of data: structured (e.g. product catalogues and sales information), unstructured (e.g. fashion blogs, customer reviews, social media posts and news messages) and binary data (e.g. videos and pictures). A prerequisite for the design of FaBIAM is that it must be able to store and process both structured and unstructured data, while binary data can be generally left as is. Therefore, next to CSV (the de facto standard data format for structured data) MonetDB also support JSON (the de facto standard data format for unstructured data) as a native data type. In Section 2.2, we will detail how JSON data can be loaded into MonetDB and queried.

Processing Layer This layer in the middle of the MonetDB kernel provides various features to facilitate query processing. In the context of the FashionBrain project in general and WP2 in particular, we have introduced several major extensions in this layer geared towards streaming and time series (fashion) data processing by means of both traditional SQL queries, as well as using modern machine learning technologies. This include i) major extensions to MonetDB's support for Window Function, which is detailed in the sibling deliverable *D2.4 "Time Series Operators for MonetDB"*; ii) a Continuous Query Engine (CQE) for streaming and Internet of Things (IoT) data, which will be detailed below in Section 2.3; and iii) a tight integration with various machine learning libraries, including the popular TensorFlow library, through SQL Python User Defined Functions (UDF)s, which will be detailed below in Section 2.4.

Analysis Layer In this layer at the top of the MonetDB kernel, we have integrated technologies of FashionBrain partners (under the collaborations of the respective partner) to enrich MonetDB's analytical features for (fashion) text data and time series data:

- *FLAIR* [1] is a python library (provided by Zalando) for named entity recognition. In Section 2.4, we will describe how one can use FLAIR from within MonetDB through SQL Python UDF.
- *IDEL* [5] is also a python library (provided by BEUTH), but for linking of already identified entities between text data and relational records, and for records linkage of already identified entities in relational records. The integration of IDEL in MonetDB was described in details in earlier deliverables *D4.1 "Report on text joins"* and *D4.2 "Demo on text joins"*.
- *RecovDB* [2] is a MonetDB-based RDBMS for the recovery of blocks of missing values in time series stored in MonetDB. The Centroid Decomposition (CD)-based recovery algorithm (provided by UNIFR) is implemented as SQL Python UDFs, but UNIFR and MDBS are working together on porting it to MonetDB native C-UDFs. This work is detailed in the sibling deliverable *D2.4 "Time Series Operators for MonetDB"*.

In summary, the design of the FaBIAM architecture covers the whole stack of data loading, processing and analysis specially for fashion text and time series data. Further in this chapter, we detail one component in each layer in a separate section, i.e. JSON, continuous query processing and FLAIR integration. In Chapter 3, we demonstrate how FaBIAM can be used to process, analyse and store a stream of reviews posted by Zalando customers.

JSON Function	Description
`json.filter(J, Pathexpr)`	Extracts the component from `J` that satisfied the `Pathexpr`.
`json.filter(J, Number)`	Extracts a indexed component from `J`.
`json.text(J, [Sep])`	Glue together the values separated by `Sep` character (default space).
`json.number(J)`	Turn a number, singleton array value, or singleton object tag into a `double`.
`json."integer"(J)`	Turn a number, singleton array value, or singleton object element into an `integer`.
`json.isvalid(StringExpr)`	Checks the string for JSON compliance. Returns `boolean`.
`json.isobject(StringExpr)`	Checks the string for JSON object compliance. Returns `boolean`.
`json.isarray(StringExpr)`	Checks the string for JSON array compliance. Returns `boolean`.
`json.length(J)`	Returns the number of top-level components of `J`.
`json.keyarray(J)`	Returns a list of key tags for the top-level components of `J`.
`json.valuearray(J)`	Returns a list of values for the top-level components of `J`.
`json.filter(J, Pathexpr)`	Extracts the component from `J` that satisfied the `Pathexpr`.

Table 2.1: Supported functions on MonetDB native JSON data type.

JSON path expression			Description
“$”			The root object
“.”	childname		The child step operator
“..”	childname		Recursive child step
“*”			Child name wildcard
“[”	nr	“]”	Array element access
“[”	*	“]”	Any array element access
E1	“,”	E2	Union path expressions

Table 2.2: Supported path expressions on MonetDB native JSON data type.

2.2. JSON Data Processing

JSON[1] is a lightweight data-interchange format. Since its initial introduction in 1999, it has quickly become the de facto standard format for data exchange, not only for text data but also often for numerical data. In FashionBrain, sometimes the whole data set is provided as a JSON file, such as the product examples of Macys and GAP, while in other data sets, the data of some columns is encoded as JSON objects. In this section, we describe two different ways to load JSON data into MonetDB.

JSON as Native Data Type

In MonetDB one can declare an SQL column to be of the type `JSON` and load JSON objects into the column. In the example below, we first create a table `json_example` with a single column `c1`, and then insert three JSON objects into it:

```
$ mclient -d fb -H
Welcome to mclient, the MonetDB/SQL interactive terminal (unreleased)
Database: MonetDB v11.31.12 (unreleased), 'mapi:monetdb://Nyx.local:50000/fb'
...
sql>CREATE TABLE json_example(c1 JSON);
operation successful
sql>INSERT INTO json_example VALUES
more>('{ "category": "reference",
```

[1] http://www.json.org

```
more>    "author": "Nigel Rees",
more>    "title": "Sayings of the Century",
more>    "price": 8.95
more>}'),
more>('{ "category": "fiction",
more>    "author": "Evelyn Waugh",
more>    "title": "Sword of Honour",
more>    "price": 12.99
more>}'),
more>('{ "category": "novel",
more>    "author": "Evelyn Waugh",
more>    "title": "Brideshead Revisited",
more>    "price": 13.60
more>}');
3 affected rows
```

Subsequently, one can query the JSON objects using the built-in JSON functions and path expressions. For example the following query turns the JSON objects into a table with the proper data conversion:

```
sql>SELECT json.text(json.filter(c1, '$.category'), ',') AS category,
more>      json.text(json.filter(c1, '$.author'), ',') AS author,
more>      json.text(json.filter(c1, '$.title'),',') AS book,
more>      json.number(json.filter(c1, '$.price')) AS price
more> FROM json_example ;
+-----------+-------------+------------------------+-------------------------+
| category  | author      | book                   | price                   |
+===========+=============+========================+=========================+
| reference | Nigel Rees  | Sayings of the Century |                    8.95 |
| fiction   | Evelyn Waugh | Sword of Honour       |                   12.99 |
| novel     | Evelyn Waugh | Brideshead Revisited  |                    13.6 |
+-----------+-------------+------------------------+-------------------------+
3 tuples
```

The following query compute the total book sales price per author:

```
sql>SELECT json.text(json.filter(c1, '$.author'), ',') AS author,
more>      SUM(json.number(json.filter(c1, '$.price'))) AS total_price
more> FROM json_example GROUP BY author;
+--------------+--------------------------+
| author       | total_price              |
+==============+==========================+
| Nigel Rees   |                     8.95 |
| Evelyn Waugh |                    26.59 |
+--------------+--------------------------+
2 tuples
```

Table 2.1 and Table 2.2 show the built-in functions and path expressions supported by MonetDB for its native JSON data type.

JSON into SQL Tables

Although JSON was originally introduced to encode unstructured data, in reality, JSON data often has a fairly clear structure, for instance, when used to encode persons, books, fashion items information. Therefore, MonetDB provides a second way to process and load JSON data directly into SQL tables using the MonetDB/Python loader functions[2].

[2] https://www.monetdb.org/blog/monetdbpython-loader-functions

Assume the information of authors and books above is stored in a file “books.json”:

```
$ cat books.json
{ "category": ["reference", "fiction", "novel"],
  "author":   ["Nigel Rees", "Evelyn Waugh", "Evelyn Waugh"],
  "title":    ["Sayings of the Century", "Sword of Honour", "Brideshead Revisited"],
  "price":    [8.95, 12.99, 13.60]
}
```

We can create a simple `json_loader` function to process the JSON file and load its contents into an SQL table:

```
sql>CREATE LOADER json_loader(filename STRING) LANGUAGE python {
more>  import json
more>  f = open(filename)
more>  _emit.emit(json.load(f))
more>  f.close
more>};
operation successful
sql>CREATE TABLE json_books FROM loader json_loader('<path-to>/books.json');
operation successful
sql>SELECT * FROM json_books;
+-----------+-------------+------------------------+--------------------------+
| category  | author      | title                  | price                    |
+===========+=============+========================+==========================+
| reference | Nigel Rees  | Sayings of the Century |                     8.95 |
| fiction   | Evelyn Waugh| Sword of Honour        |                    12.99 |
| novel     | Evelyn Waugh| Brideshead Revisited   |                     13.6 |
+-----------+-------------+------------------------+--------------------------+
3 tuples
```

Now we can freely query the data using any SQL features supported by MonetDB, e.g.:

```
sql>SELECT author, SUM(price) FROM json_books GROUP BY author;
+--------------+--------------------------+
| author       | L3                       |
+==============+==========================+
| Nigel Rees   |                     8.95 |
| Evelyn Waugh |                    26.59 |
+--------------+--------------------------+
2 tuples
```

2.3. Streaming Data and Continuous Query

Time series data are series of values obtained at successive times with regular or irregular intervals between them. Many fashion data, such as customer reviews, fashion blogs, social media messages and click streams, can be regarded as time series (mostly) with irregular time intervals. Taking customer reviews as an example, the raw data can be simply modelled as one long series of `<ts TIMESTAMP, review STRING>` pairs[3]. By analysing this type of data, which we refer to as *fashion time series*, fashion retailers would be able to gain valuable insights of not only trends, moods and opinions of potential customers at a given moment in time, but also the

[3] Of course, each pair needs to be annotated with meta information, such as the identify of the customer and the reviewed product.

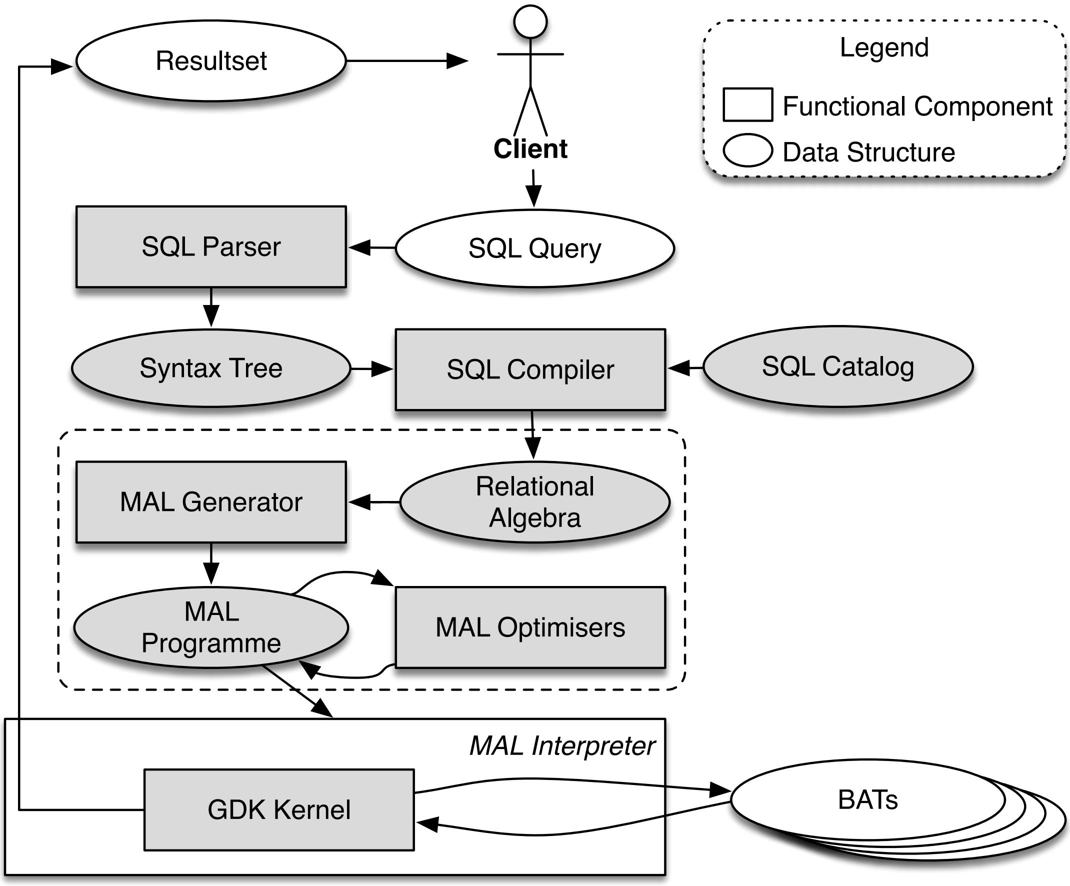

Figure 2.2: MonetDB server software implementation stack: query ⇒ parser ⇒ plan generator ⇒ optimiser ⇒ executor. Boxes with gray background are components modified for streaming data and continuous query processing.

changes in trends, moods and opinions of a period of time [3].

Fashion time series is usually produced as streams of data. Hence, supporting fashion time series does not only require a system to be able to process streaming data but also persistent relational data, because information from customer reviews and fashion blogs typically need to be linked to the product catalogues and sales information which is generally stored in a relational database. Therefore, we have extended MonetDB with the notion of `STREAM TABLE` and a continuous query scheduler, so that we can also benefit from the 25+ years research and engineering work on optimising data processing in MonetDB for fashion time series streaming data processing.

MonetDB Implementation Architecture

Figure 2.2 shows the software implementation stack of a MonetDB database server. The boxes with gray background are components that have been modified or extended to support the storage and management of streaming data, and the processing of continuous queries.

At the SQL level, all components, including the parser, syntax tree, compiler and catalog, have been extended to support the new language syntax for streaming tables and continuous queries.

The MonetDB Assembly Language (MAL) is a MonetDB internal language in which the physical query execution plans are expressed[4]. At the MAL level, all components have been extended to support the new language features, as well as a new MAL optimiser, called *continuous query scheduler*, who is in charge of the administration and invocation of continuous queries.

Finally, at the database execution kernel level (i.e. the *GDK kernel*), the transaction manager has been modified to use a much lighter transaction scheme for streaming tables and continuous queries, because streaming tables only contain transient data to which the strict database Atomicity, Consistency, Isolation and Durability (ACID) properties do not apply.

Streaming Tables

Data delivered or generated in streaming applications often require immediate processing. In most cases, the raw data need not end-up in the persistent store of a database. Instead, it is first refined and aggregated. Moreover, modern message brokers often manage the persistency of the raw data in a distributed and reliable fashion. In case of failures, they already provide methods to go back in time to start reprocessing. Redoing this work as part of a database transaction would be unnecessary and a potential performance drain.

This leaves us with the notion of *streaming tables* which are common in most streaming databases. They are light versions of normal relational tables, often solely kept in memory and not subjected to the transaction management. They are the end-points to deliver the streaming events. The following SQL syntax specifies how a streaming table can be created in MonetDB:

```
CREATE STREAM TABLE tname (... columns ...)
  [SET [WINDOW positive_number] [STRIDE positive_number]];
```

The column definitions follow the regular definition of persistent tables. Primary Keys and Foreign Key constraints are ignored as a reaction to their violation would be ill-defined in case of a streaming table.

[4] https://www.monetdb.org/Documentation/Manuals/MonetDB/Architecture

The `WINDOW` property determines when a continuous query that has been defined on this table should be triggered. When set, the `WINDOW` parameter denotes the minimal number of tuples in the streaming table to trigger a continuous query on it. If not provided (default), then any continuous query using this stream table will be triggered by an interval timer instead.

The `STRIDE` property determines what to do with tuples that have been consumed by a continuous query. When set, the `STRIDE` parameter denotes the number of tuples to be deleted from this stream table at the end of a continuous query invocation. The default action is to remove all tuples seen in the query invocation, otherwise the oldest N tuples are removed. Setting N to zero will keep all tuples until explicitly deletion by a continuous query. The `STRIDE` size cannot be larger than the size of the window to avoid events received but never processed. The parameters can be changed later with the following SQL commands:

```
ALTER STREAM TABLE tname SET WINDOW positive_number;
ALTER STREAM TABLE tname SET STRIDE positive_number;
```

Continuous Queries

The semantics of continuous queries are encapsulated into ordinary SQL UDF and User Defined Procedure (UDP). They only differ in the way they are called, and they only use `STREAM TABLE`s as input/output. Given an existing SQL UDF, it can be registered at the continuous query scheduler using the command:

```
START CONTINUOUS { PROCEDURE | FUNCTION } fname '(' arguments  ')'
  [WITH [HEARTBEAT positive_number] [CLOCK literal] [CYCLES positive_number]] [AS tagname];
```

The scheduler is bases on a Petri-net model[5], which activates the execution of a continuous UDF/UDP when all its input triggers are satisfied.

The `HEARTBEAT` parameter indicates the number of milliseconds between calls to the continuous query. If not set (default), the streaming tables used in the UDF/UDP will be scanned making it a tuple-based continuous query instead. It is not possible to set both `HEARTBEAT` and a `WINDOW` parameters at the same time, i.e. only one of the temporal and spatial conditions may be set. If neither is set, then the continuous query will be triggered in each Petri-net cycle. The `CYCLES` parameter tells the number of times the continuous query will be run before being removed by the Petri-net. If not indicated (default), the continuous query will run forever.

The `CLOCK` parameter specifies the wall-clock time for the continuous query to start, otherwise it will start immediately upon registration.

The `literal` can be a timestamp (e.g. `timestamp '2017-08-29 15:05:40'`) which sets the continuous query to start at that point, a date (e.g. `date '2017-08-29'`) on which the continuous query will start at midnight, a time value (e.g. `time`

[5] https://en.wikipedia.org/wiki/Petri_net

'15:05:40') meaning that the continuous query will start today at that time, or simply a UNIX timestamp integer with millisecond precision.

The `tagname` parameter is used to identify a continuous query. In this way, an SQL UDF/UDP with different arguments can be registered as different continuous queries. If a `tagname` is not provided, then the function/procedure name will be used instead.

After having registered a continuous query, it is possible to pause, resume or stop it. Their syntax is as follows:

```
-- Stop and remove a continuous query from the Petri-net.
STOP CONTINUOUS tagname;

-- Pause a continuous query from the Petri-net but do not remove it.
PAUSE CONTINUOUS tagname;

-- Resume a paused continuous query. If the HEARTBEAT and CYCLES parameters are not provided
--   (default), then the previous registered values will be used.
RESUME CONTINUOUS tagname [WITH [HEARTBEAT positive_number] [CLOCK literal] [CYCLES positive_number]]
```

The following SQL commands apply to all:

```
-- Stop and remove all continuous queries from the Petri-net.
STOP ALL CONTINUOUS;

-- Pause all continuous queries in the Petri-net.
PAUSE ALL CONTINUOUS

-- Resume all continuous queries in the Petri-net with the previous HEARTBEAT.
RESUME ALL CONTINUOUS and CYCLES values.
```

During the first iteration of a continuous function, a streaming table is created under the `cquery` schema to store the outputs of the function during its lifetime in the scheduler. This streaming table will be dropped once the continuous function is deregistered from the scheduler or the MonetDB server restarts.

Several implementation choices should be noted:

- All continuous queries are stopped once the MonetDB server shuts down. The user must start the continuous queries manually at restart of the server.
- Streaming tables are volatile for better performance under large workloads. This means that upon restart of the database server their data is lost.
- A streaming table cannot be dropped while there is a continuous query using it. The same condition holds for registered UDFs.
- The SQL catalog properties of a streaming table including columns cannot be altered unlike regular SQL tables. Users must drop the table and recreate it with the desired changes.
- The current scheduler implementation is agnostic of transaction management. This means that if a continuous query was started, paused, resumed or stopped during a rollbacked transaction, the changes are not reverted.

- If an error happens during a single execution, the continuous query gets paused automatically. The error can be checked with a `cquery.status()` or `cquery.log()` call.

Moving Average Example

Moving average[6] is an important calculation in statistics. For example, it is often used in technical analysis of financial data, such as stock prices, returns or trading volumes. It is also used in economics to examine gross domestic product, employment or other macroeconomic time series.

Although the concept of moving average is fairly simply, it is exceptionally difficult to express this calculation in vanilla SQL queries, because e.g. the relational data model does not have the notion of order. So, computing moving average using vanilla SQL often results in complex queries with many expensive self-joins which end up in bad performance.

However, when a Database Management System (DBMS) is extended with features to support streaming data and continuous queries, computing statistic functions such as moving average becomes simple. The queries below show how this can be done in MonetDB.

First, we create a streaming table `inputStream` to temporarily store a stream of integers. We also specify that continuous queries on this table should be triggered whenever at least four tuples have been inserted into the table, i.e. `WINDOW 4`; and the first two tuples should be deleted after each invocation of a continuous query, i.e. In this way, we create a window of size 4 and it will be advanced 2 positions at a time. `STRIDE 2`.

```
sql>CREATE STREAM TABLE inputStream (val INT) SET WINDOW 4 STRIDE 2;
operation successful
```

Then, we create an ordinary SQL UDF to compute the average of all values currently in `inputStream`. And we immediately register this function at the continuous query scheduler for execution in the background (i.e. `START CONTINUOUS FUNCTION`).

```
sql>-- calculate the average value of the window during execution
sql>CREATE FUNCTION calculateAverage() RETURNS REAL BEGIN
more>   RETURN SELECT AVG(val) as calc FROM inputStream;
more>END;
operation successful
sql>START CONTINUOUS FUNCTION calculateAverage() AS calcavg;
operation successful
```

Now we can trigger the work of the continuous query by simply adding the sufficient amount of values into the streaming table `inputStream`:

```
sql>INSERT INTO inputStream VALUES (33), (29), (30), (32);
4 affected rows
```

[6] https://en.wikipedia.org/wiki/Moving_average

Because the continuous query schedule is constantly running in the background, after the `INSERT` query above, the scheduler will automatically trigger the execution of the continuous query `calcavg`. We can check the effect of this both in the streaming table `inputStream` and in the temporary table that holds the output of `calcavg`:

```
sql>-- The first 2 tuples should have been deleted
sql>SELECT * FROM inputStream;
+------+
| val  |
+======+
|   30 |
|   32 |
+------+
2 tuples

sql>SELECT result FROM cquery.calcavg; -- The CQ has been performed once
+-----------------+
| result          |
+=================+
| 31              |
+-----------------+
1 tuple
```

Next, we insert one new tuple into `inputStream`, which should not trigger an execution of `calcavg`. So, `inputStream` contains three tuples, and `cquery.calcavg` still contains the one result from the previous execution:

```
sql>INSERT INTO inputStream VALUES (42);
1 affected row
sql>SELECT * FROM inputStream;
+------+
| val  |
+======+
|   30 |
|   32 |
|   42 |
+------+
3 tuples
sql>SELECT result FROM cquery.calcavg; --The CQ was not performed again yet
+-----------------+
| result          |
+=================+
| 31              |
+-----------------+
1 tuple
```

Adding one more tuple into `inputStream` will make it contain four tuples again, which triggers the second execution of `calcavg`. Examining the contents of `inputStream` again shows two remaining tuples[7], while `cquery.calcavg` now contains two averages:

```
sql>INSERT INTO inputStream VALUES (24);
1 affected row
sql>SELECT * FROM inputStream;
```

[7] Actually, there is a slight delay between a streaming table get sufficient number of tuples inserted and the corresponding continuous query gets executed. So, if one executes the `SELECT` query immediately after the `INSERT` query (e.g. `"INSERT INTO inputStream VALUES (24); SELECT * FROM inputStream;"`, the `SELECT` query might still return four tuples.

```
+-------+
| val   |
+=======+
|    42 |
|    24 |
+-------+
2 tuples
sql>SELECT result FROM cquery.calcavg; --The query was performed again
+------------------+
| result           |
+==================+
| 31               |
| 32               |
+------------------+
2 tuples
```

The query will continue running in the background until it is manually stopped (using `STOP CONTINUOUS calcavg;`) or the server shuts down.

2.4. Text Data Analysis with Machine Learning

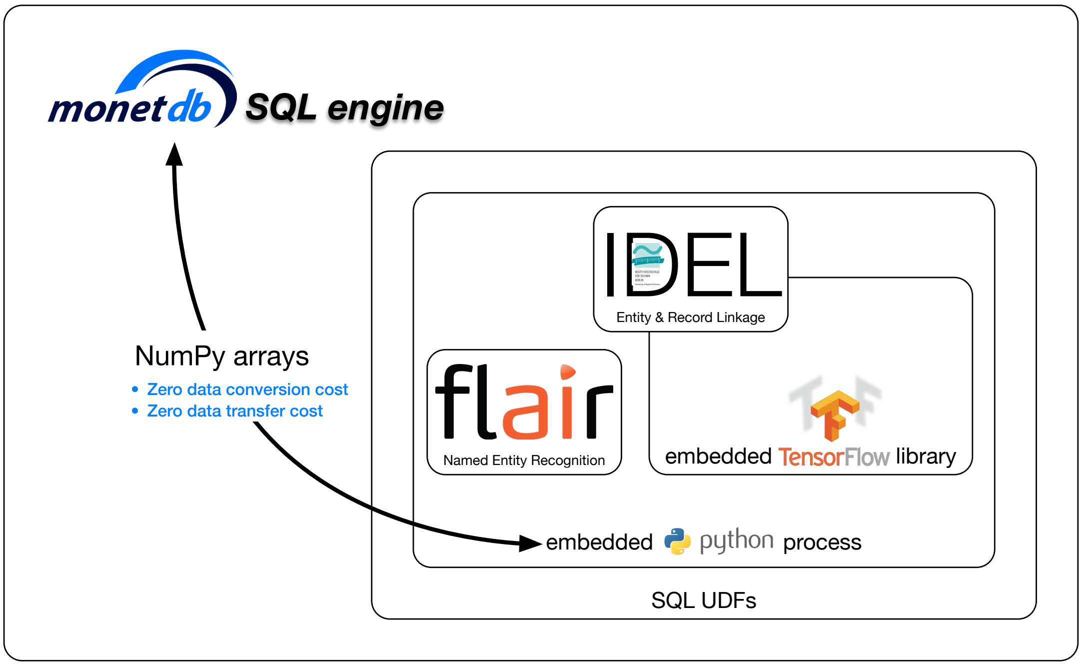

Figure 2.3: Architecture of In-Database Machine Learning in MonetDB.

Figure 2.3 shows the architecture of in-database machine learning in MonetDB.

In the remainder of this section, we describe in details how the integration of MonetDB/FLAIR is done. The integration of MonetDB/TensorFlow is done in a similar way, so the examples are included in Appendix A, which consist of some basic (matrix) operations and the Word2Vec model.

MonetDB/FLAIR Examples

The function `flair()` below shows the most straightforward way to integrate FLAIR into MonetDB. It simply wraps an SQL `CREATE FUNCTION` definition around the Python code from the FLAIR "Example Usage"[8], so that one can apply FLAIR on the input `STRING s` and receive a tagged string as the result:

```
CREATE FUNCTION flair (s STRING) RETURNS STRING LANGUAGE python {
  from flair.data import Sentence
  from flair.models import SequenceTagger

  # Make a sentence object from the input string
  sentence = Sentence(s, use_tokenizer=True)
  # load the NER tagger
  tagger = SequenceTagger.load('ner')
  # run NER over sentence
  tagger.predict(sentence)
  return sentence.to_tagged_string()
};
```

The function `flair_bulk()` shows how we can leverage MonetDB's bulk execution model[9] to apply FLAIR on multiple sentences in one go. This also shows how the use case of "Tagging a List of Sentences"[10] can be translated into MonetDB.

Although the input parameter `s` is still declared as a single `STRING`, at the runtime, MonetDB actually passes a NumPy array of strings to this function. So, unlike the single-string-input function `flair()` above, the implementation of `flair_bulk()` is adjusted in such a way that it can handle an array of strings and also returns an array of tagged strings.

```
CREATE FUNCTION flair_bulk (s STRING) RETURNS STRING
LANGUAGE python
{
  from flair.data import Sentence
  from flair.models import SequenceTagger

  # Make sentence objects from the input strings
  sentences = [Sentence(sent, use_tokenizer=True) for sent in s]
  # load the NER tagger
  tagger = SequenceTagger.load('ner')
  # run NER over sentences
  tagger.predict(sentences)
  return [sent.to_tagged_string() for sent in sentences]
};
```

The function `flair_tbl()` is similar to `flair_bulk()` in the sense that it also operates on an array of strings in one go. However, instead of returning an array of tagged strings, it returns a table with a single column, in which the array of tagged strings are stored. This is a so-called "table returning function".

[8] https://github.com/zalandoresearch/flair
[9] https://www.monetdb.org/Documentation/Manuals/MonetDB/Architecture/ExecutionModel
[10] https://github.com/zalandoresearch/flair/blob/master/resources/docs/TUTORIAL_TAGGING.md

```
CREATE FUNCTION flair_tbl (s STRING) RETURNS TABLE(ss STRING)
LANGUAGE python
{
  from flair.data import Sentence
  from flair.models import SequenceTagger

  # Make sentence objects from the input strings
  sentences = [Sentence(sent, use_tokenizer=True) for sent in s]
  # load the NER tagger
  tagger = SequenceTagger.load('ner')
  # run NER over sentences
  tagger.predict(sentences)
  return [sent.to_tagged_string() for sent in sentences]
};
```

Once the SQL Python functions have been created, we can call them to use FLAIR from within MonetDB. In the examples below, the sentences were taken from the FLAIR "Tutorial"[11].

In the following `SELECT` query, the function `flair()` is executed seven times, each time to tag one string.

```
sql>SELECT
more> flair ('I love Berlin.'),
more> flair ('The grass is green .'),
more> flair ('France is the current world cup winner.'),
more> flair ('George Washington went to Washington.'),
more> flair ('George Washington ging nach Washington.'),
more> flair ('George returned to Berlin to return his hat.'),
more> flair ('He had a look at different hats.');
+-------------+-------------+-------------+-------------+-------------+-------------+-------------+
| L2          | L4          | L6          | L10         | L12         | L14         | L16         |
+=============+=============+=============+=============+=============+=============+=============+
| I love Berl | The grass i | France <S-L | George <B-P | George <B-P | George <S-P | He had a lo |
: in <S-LOC>  : s green .   : OC> is the  : ER> Washing : ER> Washing : ER> returne : ok at diffe :
: .           :             : current w...> ton <E-PE...> ton <E-PE...> d to Berl...> rent hats . :
+-------------+-------------+-------------+-------------+-------------+-------------+-------------+
1 tuple !4 fields truncated!
clk: 16.165 sec
```

In the queries below, we first store the sentences in a table in MonetDB, then in the two `SELECT` queries, we first retrieve the sentences from the SQL, before passing them to `flair_bulk()` and `flair_tbl()`, respectively. In those queries, the UDFs `flair_bulk()` and `flair_tbl()` are each executed only once to tag *all* sentences. The effect of these bulk executions is clear to see in their execution times. Compared to the above `SELECT` query with seven calls to `flair()`, the bulk versions are ~5 times faster.

```
sql>CREATE TABLE sentences (s STRING);
CREATE TABLE: name 'sentences' already in use
sql>INSERT INTO sentences VALUES
more>     ('I love Berlin.'),
more>     ('The grass is green .'),
more>     ('France is the current world cup winner.'),
more>     ('George Washington went to Washington.'),
more>     ('George Washington ging nach Washington.'),
more>     ('George returned to Berlin to return his hat.'),
```

[11] https://github.com/zalandoresearch/flair

```
more>    ('He had a look at different hats.');
7 affected rows
clk: 1.856 ms
sql>SELECT flair_bulk(s) FROM sentences;
+-------------------------------------------------------------------+
| L2                                                                |
+===================================================================+
| I love Berlin <S-LOC> .                                           |
| The grass is green .                                              |
| France <S-LOC> is the current world cup winner .                  |
| George <B-PER> Washington <E-PER> went to Washington <S-LOC> .    |
| George <B-PER> Washington <E-PER> ging nach Washington <S-PER> .  |
| George <S-PER> returned to Berlin <S-LOC> to return his hat .     |
| He had a look at different hats .                                 |
+-------------------------------------------------------------------+
7 tuples
clk: 3.004 sec

sql>SELECT * FROM flair_tbl((SELECT s FROM sentences));
+-------------------------------------------------------------------+
| ss                                                                |
+===================================================================+
| I love Berlin <S-LOC> .                                           |
| The grass is green .                                              |
| France <S-LOC> is the current world cup winner .                  |
| George <B-PER> Washington <E-PER> went to Washington <S-LOC> .    |
| George <B-LOC> Washington <E-LOC> ging nach Washington <S-PER> .  |
| George <S-PER> returned to Berlin <S-LOC> to return his hat .     |
| He had a look at different hats .                                 |
+-------------------------------------------------------------------+
7 tuples
clk: 3.018 sec
```

3. FashionBrain Use Cases

In this chapter, we show with a running example how continuous query processing (Section 2.3) and FLAIR (embedded as SQL Python UDFs, see Section 2.4) can be combined to continuously analyse incoming stream of fashion text. This is an implementation of the business scenario 4 “Fashion Time Series Analysis” defined in deliverable *D1.2 “Requirement analysis document”*.

Step 1: create the core UDF for text analysis. The function `flair_entity_tags()` below takes a column of strings (i.e. `s STRING`) together with the corresponding ID of those strings (i.e. `id INT`) as its input. It uses FLAIR’s “ner” model to tag all review strings. The tagging results are extracted into a table with five columns as the function’s return value: `id` is the ID of the original string, `entity` is the identified entity, `tag` is the type assigned to this entity (i.e. what type of entity is it), and `start_pos` and `end_pos` are respectively the start and end positions of this entity in the original string, since an entity can contain multiple words. The value of a `tag` can be `LOC` for location, `PER` for person, `ORG` for organisation and `MISC` for miscellaneous.

```
$ mclient -d fb -H -t clock
Welcome to mclient, the MonetDB/SQL interactive terminal (unreleased)
Database: MonetDB v11.32.0 (hg id: 181dee462c5b), 'mapi:monetdb://dhcp-51.eduroam.cwi.nl:50000/fb'
Type \q to quit, \? for a list of available commands
auto commit mode: on
sql>CREATE FUNCTION flair_entity_tags (id INT, s STRING)
more>RETURNS TABLE(id INT, entity STRING, tag STRING, start_pos INT, end_pos INT)
more>LANGUAGE python
more>{
more>  from flair.data import Sentence
more>  from flair.models import SequenceTagger
more>  import numpy
more>
more>  # Make sentence objects from the input strings
more>  sentences = [Sentence(sent, use_tokenizer=True) for sent in s]
more>  # load the NER tagger
more>  tagger = SequenceTagger.load('ner')
more>  # run NER over sentences
more>  tagger.predict(sentences)
more>
more>  ids = []
more>  entities = []
more>  tags = []
more>  start_poss = []
more>  end_poss = []
more>
more>  for idx,sent in numpy.ndenumerate(sentences):
more>    for e in sent.get_spans('ner'):
more>        ids.append(id[idx[0]])
more>        entities.append(e.text)
more>        tags.append(e.tag)
more>        start_poss.append(e.start_pos)
more>        end_poss.append(e.end_pos)
```

```
more>
more>  return [ids, entities, tags, start_poss, end_poss]
more>};
operation successful
```

Step 2: prepare data. For this use case, we have used the product reviews data set provided by Zalando as one of the M3 data sets. This data set (called "zalando-reviews-FB-release") contains reviews left by customers on the Zalando web page. Users who have purchased a product may leave a textual review, typically describing and/or rating the product. Each record in "zalando-reviews-FB-release" contains five fields:

- `id`: a unique identifier of this record (SQL data type: `INTEGER`)
- `title`: title of this review, as given by the customer writing this review (data type: `STRING`)
- `text`: customer review as plain text (SQL data type: `STRING`)
- `sku`: product SKU, used to identify the product (SQL data type: `STRING`)
- `language_code`: two-letter code identifying the language used in this review (SQL data type: `CHAR(2)`)

Since in this use case we are going to apply FLAIR to recognise the entities mentioned in the customer reviews, we create the following `STREAM TABLE` to temporarily store only the `id` and `text` of a customer review. The `WINDOW` size is set to 4, i.e. whenever at least four records have been inserted into `review_stream`, the continuous query defined on this table will be automatically executed. We do not modify the default value of `STRIDE`, hence the records in `review_stream` will be immediately deleted once they have been consumed by a continuous query.

```
sql>CREATE STREAM TABLE review_stream (id INT, review STRING) SET WINDOW 4;
operation successful
```

Step 3: create and start continuous query. The function `tagreviews()` below is a wrapper of the continuous query we want to execute. It applies `flair_entity_tags()` on all reviews currently in `review_stream` and returns the `id`, `entity` and `tag` information (since we are not interested in the positions of the entities right now). After the execution of the statement `START CONTINUOUS FUNCTION`, `targreviews()` is registered at the continuous query scheduler as a continuous function under the name `tagged_reviews` and waiting for a trigger to be executed.

```
sql>CREATE FUNCTION tagreviews () RETURNS TABLE (id INT, entity STRING, tag STRING)
more>BEGIN
more>    RETURN SELECT id, entity, tag FROM flair_entity_tags(
more>        (SELECT id, review FROM review_stream));
more>END;
operation successful
sql>START CONTINUOUS FUNCTION tagreviews() AS tagged_reviews;
operation successful
```

Step 4: inserting data to trigger the continuous query. Here we use the reviews written in English. By inserting the reviews in small batches one at a time, we mimic the real-world situation in which customer reviews are posted in the streaming fashion.

In the queries below, we first insert two records. Since there are not enough records in `review_stream`, the continuous query `tagged_reviews()` is not triggered. We can check this by looking into `cquery.tagged_reviews` (the temporary table that has been automatically created to store the results of each invocation of `tagged_reviews()`) and `review_stream`. As expected, `cquery.tagged_reviews` is empty, while `review_stream` contains the two newly records.

```
sql>INSERT INTO review_stream VALUES
more>    (1862,'My first order with Zalando and so far everything has gone well. I\'ll be back.
more>        The shoes, met my expectations despite a little tightness at first - which is normal,
more>        I suppose . '),
more>    (1893,'Great quality and the best design by Boxfresh! After being really disappointed
more>        by the quality of Pointers, I have decided that Boxfresh are the shoe brand for me.
more>        Only drawback: they are really hot and sweaty... so really better for the winter
more>        (but not with the white sole!)');
2 affected rows
sql>SELECT * FROM cquery.tagged_reviews;
+----+--------+-----+
| id | entity | tag |
+====+========+=====+
+----+--------+-----+
0 tuples
sql>SELECT * FROM review_stream;
+------+--------------------------------------------------------------------------+
| id   | review                                                                   |
+======+==========================================================================+
| 1862 | My first order with Zalando and so far everything has gone well. I'll be ba |
:      : ck. The shoes, met my expectations despite a little      tightness at first :
:      :  - which is normal, I suppose .                                          :
| 1893 | Great quality and the best design by Boxfresh! After being really disappoin |
:      : ted by the quality of Pointers, I have decided that       Boxfresh are the  :
:      : shoe brand for me. Only drawback: they are really hot and sweaty... so really :
:      :  better for the winter (but not with the white sole!)                     :
+------+--------------------------------------------------------------------------+
2 tuples
```

In the queries below, we insert two more records. Now that there are four records in `review_stream`, the continuous query scheduler will automatically start the execution of `tagged_reviews` after the `INSERT INTO` query. Let us wait for a while for `tagged_reviews` to finish...

```
sql>INSERT INTO review_stream VALUES
more>(1905,'Boxfresh...always great'),
more>(1906,'I am very happy with the shoes and with Zalando\'s service');
2 affected rows
sql>-- wait some time for the CQ tagged_review to finish...
```

If we now check the tables again. The table `cquery.tagged_reviews` now contains the output of the continuous query `tagged_reviews`, while `review_stream` is empty (because the records are automatically deleted after having been consumed by a continuous query):

```
sql>SELECT * FROM cquery.tagged_reviews;
```

```
+------+---------+------+
| id   | entity  | tag  |
+======+=========+======+
| 1862 | Zalando | PER  |
| 1893 | Boxfresh | PER  |
| 1893 | Boxfresh | ORG  |
| 1905 | Boxfresh | PER  |
| 1906 | Zalando | PER  |
+------+---------+------+
5 tuples
sql>SELECT * FROM review_stream;
+------+--------+
| id   | review |
+======+========+
+------+--------+
2 tuples
```

Let us insert another batch of reviews. This time we immediately insert four records, which should trigger an invocation of `tagged_reviews`.

```
sql>INSERT INTO review_stream VALUES
more>    (1919,'A quality Boxfresh casual shoe / trainer. Slightly larger fit. Thin & very
more>        uncomfortable sole. \r\n\r\nThe second day I wore these,  I did a fair bit of walking
more>        and paid for it with some almighty blisters! I now have some inner soles in them and
more>        all is well again. So I would  definately recommend buying some inner soles to fit
more>        these when purchasing.'),
more>    (2591,'These are lovely boots, very comfy (especially because you can adjust the width and
more>        leg fitting using the lace up front) and warm    but not too warm.  I can\'t really
more>        comment on the quality because I\'ve only had them a few days, but they seem well made.
more>        However, they are   huge!  I\'m usually a 7.5 and have worn size 7 Skechers before, but
more>        I ended up with a size 6 in these boots as the 7s were so long I would have been
more>        tripping over them.'),
more>    (2906,'In my opinion Iso-Parcour make the best winter boots money can buy. The neoprene
more>        lining insulates the feet against extreme cold (-   20!!!) You only need to wear thin
more>        stockings or socks - that way you feel as if you are walking barefoot through the snow!
more>        Long walks in snow    are no problem with these boots. Two small criticisms: the price
more>        is really steep and you can really only wear these boots in minus temps. At +  10 these
more>        boots are far too heavy and hot.'),
more>    (3550,'The shoes look just like the pictures on the online shop, which I was very happy
more>        about!! My size fit perfectly and after a few days  of wearing them in they were very
more>        comfortable. The only thing that is a shame is that the shoes are "Made in India",
more>        despite being marketed as  a traditional English brand. For this price, I would expect
more>        better.');
4 affected rows
```

If we again wait for a while before checking the results, we can now see that more results have been added to `cquery.tagged_reviews`, while `review_stream` is emptied again:

```
sql>SELECT * FROM cquery.tagged_reviews;
+------+--------------+------+
| id   | entity       | tag  |
+======+==============+======+
| 1862 | Zalando      | PER  |
| 1893 | Boxfresh     | MISC |
| 1893 | Pointers     | MISC |
| 1893 | Boxfresh     | ORG  |
| 1905 | Boxfresh     | PER  |
| 1906 | Zalando      | PER  |
| 2591 | Skechers     | MISC |
| 2906 | Iso-Parcour  | ORG  |
| 3550 | Made in India | MISC |
| 3550 | English      | MISC |
+------+--------------+------+
```

```
10 tuples
sql>SELECT * FROM review_stream;
+----+--------+
| id | review |
+====+========+
+----+--------+
0 tuples
```

If we continue inserting more review records, the continuous query `tagged_reviews` will be repeated until we explicitly stop it as in the queries below or shut down the MonetDB server. Note that the temporary table `cquery.tagged_reviews` associated with the continuous query will cease to exist, once the continuous query is stopped. So, before stopping the continuous query, we save the contents of the temporary table in a persistent table. This is one of the main advantage of using an RDBMS such as MonetDB as the backbone of a continuous query processing system, i.e. one can freely switch between streaming and persistent data within a single system:

```
sql>CREATE TABLE tagged_reviews_persist AS SELECT * FROM cquery.tagged_reviews;
operation successful
sql>STOP CONTINUOUS tagged_reviews;
operation successful
sql>SELECT * FROM cquery.tagged_reviews;
SELECT: no such table 'tagged_reviews'
```

Alternatively, a user might want to emit the results of the continuous query to some output channel, such as a web page, or trigger some notifications when certain results have been generated. Supporting such features is on our list of future work.

4. Conclusions

In this document, we have described the design and implementation of FaBIAM, our solution for storing, managing and processing heterogeneous fashion data. In particular, we details our implementation in MonetDB to support unstructured data, continuous query processing and entity recognition using embedded machine learning technique. Finally, we demonstrated the usefulness of this architecture using an example in processing fashion review data.

With the introduction of streaming tables and continuous query engine, we have taken a first important step towards streaming data processing using a powerful and highly optimised relational engine. As a next step, we will continue extending the FaBIAM architecture into a complete IoT platform for streaming fashion time series processing. First of all, the new window function implementation needs to be integrated with the continuous query engine to strengthen its power on time series data processing. Secondly, the architecture needs to be extended for streaming data ingestion, continuous query results emission to external channels.

Bibliography

[1] Alan Akbik, Duncan Blythe, and Roland Vollgraf. Contextual string embeddings for sequence labeling. In *COLING 2018, 27th International Conference on Computational Linguistics*, pages 1638–1649, 2018.

[2] Ines Arous, Mourad Khayati, Philippe Cudré-Mauroux, Ying Zhang, Martin Kersten, and Svetlin Stalinlov. RecoveDB: accurate and efficient missing blocks recovery for large time series. In *Proceedings of the 35th IEEE International Conference on Data Engineering (ICDE 2019)*, April 2019. Submitted.

[3] FashionBrain. Deliverable D.1.2, Requirement analysis document, June 2017.

[4] Mourad Khayati, Michael H. Böhlen, and Johann Gamper. Memory-efficient centroid decomposition for long time series. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 100–111, April 2014. doi: 10.1109/ICDE.2014.6816643. URL https://doi.org/10.1109/ICDE.2014.6816643.

[5] Torsten Kilias, Alexander Löser, Felix Gers, Ying Zhang, Richard Koopmanschap, and Martin Kersten. IDEL: In-Database Neural Entity Linking. In *Proceedings of the IEEE International Conference on Big Data and Smart Computing (BigComp)*, Feburary 2019. To appear.

A. Appendix: MonetDB/TensorFlow Examples

In this chapter, we include several examples to show various ways in which MonetDB and TensorFlow can interacts with each other, i.e. how values can be past from the database environment to the Python/TensorFlow environment and vice versa: one at a time or in a bulk (as a NumPy array or as a database column).

A.1. Basic Operations

In this section, we show how the basic operations from the TensorFlow-Examples[1] repository can be implemented as SQL Python UDFs in MonetDB.

```
-- THE ‘‘hello world’’ example TensorFlow
CREATE FUNCTION hw () RETURNS STRING LANGUAGE PYTHON {
    import tensorflow as tf
    hello = tf.constant('Hello, TensorFlow!')
    # Start tf session
    sess = tf.Session()
    # Run the op
    return sess.run(hello)
};
SELECT hw();
```

```
-- Basic constant operation example using TensorFlow library
CREATE FUNCTION bsc_cnst() RETURNS TABLE(add_cnst INT, mul_cnst INT) LANGUAGE PYTHON {
    import tensorflow as tf

    # The value returned by the constructor represents the output
    # of the Constant op.
    a = tf.constant(2)
    b = tf.constant(3)

    # Launch the default graph.
    with tf.Session() as sess:
        res = [[sess.run(a+b)], [sess.run(a*b)]]
    return res
};
SELECT * FROM bsc_cnst();
```

```
-- A little helper table for the functions below
CREATE TABLE inputs (i INT, j INT, f FLOAT);
INSERT INTO inputs VALUES (2, 11, 9.1), (3, 12, 19.1), (4, 13, 29.1);
```

```
-- Basic constant operation example using TensorFlow library, and MonetDB bulk processing
CREATE FUNCTION bsc_cnst_add(i INT, j INT) RETURNS BIGINT LANGUAGE PYTHON {
    import tensorflow as tf

    # The value returned by the constructor represents the output
    # of the Constant op.
```

[1] https://github.com/aymericdamien/TensorFlow-Examples/tree/master/examples/1_Introduction

```
    a = tf.constant(i)
    b = tf.constant(j)

    # Launch the default graph.
    with tf.Session() as sess:
        res = sess.run(a+b)
    return res
};
CREATE FUNCTION bsc_cnst_mul(i INT, j INT) RETURNS BIGINT LANGUAGE PYTHON {
    import tensorflow as tf

    # The value returned by the constructor represents the output
    # of the Constant op.
    a = tf.constant(i)
    b = tf.constant(j)

    # Launch the default graph.
    with tf.Session() as sess:
        res = sess.run(a*b)
    return res
};
SELECT bsc_cnst_add(i, j) AS add_cnst, bsc_cnst_mul(i, j) AS mul_cnst FROM inputs;
```

```
-- Basic variable operation example using TensorFlow library
CREATE FUNCTION bsc_vars() RETURNS TABLE(add_var INT, mul_var INT) LANGUAGE PYTHON {
    import tensorflow as tf

    # The value returned by the constructor represents the output
    # of the Variable op. (define as input when running session)
    a = tf.placeholder(tf.int16)
    b = tf.placeholder(tf.int16)

    # Define some operations
    add = tf.add(a, b)
    mul = tf.multiply(a, b)

    # Launch the default graph.
    with tf.Session() as sess:
        res = [[sess.run(add, feed_dict={a:20, b:30})], [sess.run(mul, feed_dict={a:20, b:30})]]
    return res
};
SELECT * FROM bsc_vars();
```

```
-- Basic variable operation example using TensorFlow library, and MonetDB bulk processing
CREATE FUNCTION bsc_vars_add(i INT, j INT) RETURNS BIGINT LANGUAGE PYTHON {
    import tensorflow as tf

    # The value returned by the constructor represents the output
    # of the Variable op. (define as input when running session)
    a = tf.placeholder(tf.int16)
    b = tf.placeholder(tf.int16)

    # Define some operations
    add = tf.add(a, b)

    # Launch the default graph.
    with tf.Session() as sess:
        res = sess.run(add, feed_dict={a:i, b:j})
    return res
};
CREATE FUNCTION bsc_vars_mul(i INT, j INT) RETURNS BIGINT LANGUAGE PYTHON {
    import tensorflow as tf

    # The value returned by the constructor represents the output
    # of the Variable op. (define as input when running session)
```

```
    a = tf.placeholder(tf.int16)
    b = tf.placeholder(tf.int16)

    # Define some operations
    mul = tf.multiply(a, b)

    # Launch the default graph.
    with tf.Session() as sess:
        res = sess.run(mul, feed_dict={a:i, b:j})
    return res
};
SELECT bsc_vars_add(i, j) AS add_vars, bsc_vars_mul(i, j) AS mul_vars FROM inputs;
```

```
-- Basic matrix operation example using TensorFlow library
CREATE FUNCTION bsc_mtrx_1_1 () RETURNS FLOAT LANGUAGE PYTHON {
    import tensorflow as tf

    # Matrix Multiplication from TensorFlow official tutorial

    # Create a Constant op that produces a 1x2 matrix.  The op is
    # added as a node to the default graph.
    #
    # The value returned by the constructor represents the output
    # of the Constant op.
    matrix1 = tf.constant([[3., 3.]])

    # Create another Constant that produces a 2x1 matrix.
    matrix2 = tf.constant([[2.],[2.]])

    # Create a Matmul op that takes 'matrix1' and 'matrix2' as inputs.
    # The returned value, 'product', represents the result of the matrix
    # multiplication.
    product = tf.matmul(matrix1, matrix2)

    # To run the matmul op we call the session 'run()' method, passing 'product'
    # which represents the output of the matmul op.  This indicates to the call
    # that we want to get the output of the matmul op back.
    #
    # All inputs needed by the op are run automatically by the session.  They
    # typically are run in parallel.
    #
    # The call 'run(product)' thus causes the execution of threes ops in the
    # graph: the two constants and matmul.
    #
    # The output of the op is returned in 'result' as a numpy `ndarray` object.
    with tf.Session() as sess:
        res = sess.run(product)
    return res
};
SELECT bsc_mtrx_1_1();
```

```
-- Basic matrix operation example using TensorFlow library
CREATE FUNCTION bsc_mtrx_1_n () RETURNS TABLE (c1 FLOAT) LANGUAGE PYTHON {
    import tensorflow as tf

    # Matrix Multiplication from TensorFlow official tutorial

    # Create a Constant op that produces a 1x2 matrix.  The op is
    # added as a node to the default graph.
    #
    # The value returned by the constructor represents the output
    # of the Constant op.
    matrix1 = tf.constant([[3.0, 5.0]])

    # Create another Constant that produces a 2xN matrix.
```

```
    matrix2 = tf.constant([[2.1, 2.3, 2.7],[4.1, 4.3, 4.7]])

    # Create a Matmul op that takes 'matrix1' and 'matrix2' as inputs.
    # The returned value, 'product', represents the result of the matrix
    # multiplication.
    product = tf.matmul(matrix1, matrix2)

    # To run the matmul op we call the session 'run()' method, passing 'product'
    # which represents the output of the matmul op.  This indicates to the call
    # that we want to get the output of the matmul op back.
    #
    # All inputs needed by the op are run automatically by the session.  They
    # typically are run in parallel.
    #
    # The call 'run(product)' thus causes the execution of threes ops in the
    # graph: the two constants and matmul.
    #
    # The output of the op is returned in 'result' as a numpy `ndarray` object.
    with tf.Session() as sess:
        res = sess.run(product)
    return res
};
SELECT * FROM bsc_mtrx_1_n();
```

```
-- Basic matrix operation example using TensorFlow library
CREATE FUNCTION bsc_mtrx_1_n_param (f FLOAT) RETURNS TABLE (c1 FLOAT) LANGUAGE PYTHON {
    import tensorflow as tf

    # Matrix Multiplication from TensorFlow official tutorial

    # Create a Constant op that produces a 1x3 matrix.  The op is
    # added as a node to the default graph.
    #
    # The value returned by the constructor represents the output
    # of the Constant op.
    matrix1 = tf.constant([[f]])

    # Create another Constant that produces a 1xN matrix.
    matrix2 = tf.constant([[2.1, 2.3, 2.7, 4.1, 4.3, 4.7]])

    # Create a Matmul op that takes 'matrix1' and 'matrix2' as inputs.
    # The returned value, 'product', represents the result of the matrix
    # multiplication.
    product = tf.matmul(matrix1, matrix2)

    # To run the matmul op we call the session 'run()' method, passing 'product'
    # which represents the output of the matmul op.  This indicates to the call
    # that we want to get the output of the matmul op back.
    #
    # All inputs needed by the op are run automatically by the session.  They
    # typically are run in parallel.
    #
    # The call 'run(product)' thus causes the execution of threes ops in the
    # graph: the two constants and matmul.
    #
    # The output of the op is returned in 'result' as a numpy `ndarray` object.
    with tf.Session() as sess:
        res = sess.run(product)
    return res
};
SELECT * FROM bsc_mtrx_1_n_param(5.3);
```

```
-- Basic matrix operation example using TensorFlow library
CREATE FUNCTION bsc_mtrx_2_n () RETURNS TABLE (c1 FLOAT, c2 FLOAT) LANGUAGE PYTHON {
    import tensorflow as tf
```

```
    # Matrix Multiplication from TensorFlow official tutorial

    # Create a Constant op that produces a 1x3 matrix.  The op is
    # added as a node to the default graph.
    #
    # The value returned by the constructor represents the output
    # of the Constant op.
    matrix1 = tf.constant([[3, 5, 7]])

    # Create another Constant that produces a 2x1 matrix.
    matrix2 = tf.constant([[2],[4]])

    # Create a Matmul op that takes 'matrix2' and 'matrix1' as inputs.
    # The returned value, 'product', represents the result of the matrix
    # multiplication.
    product = tf.matmul(matrix2, matrix1)

    # To run the matmul op we call the session 'run()' method, passing 'product'
    # which represents the output of the matmul op.  This indicates to the call
    # that we want to get the output of the matmul op back.
    #
    # All inputs needed by the op are run automatically by the session.  They
    # typically are run in parallel.
    #
    # The call 'run(product)' thus causes the execution of threes ops in the
    # graph: the two constants and matmul.
    #
    # The output of the op is returned in 'result' as a numpy `ndarray` object.
    with tf.Session() as sess:
        res = sess.run(product)
    return res
};
SELECT * FROM bsc_mtrx_2_n();
```

A.2. Word Embeddings

In this section, we show how the word2vec model can be implemented as SQL Python UDFs in MonetDB. The Python code is based on that used in the “Vector Representations of Words” tutorial of TensorFlow[2], and the data was downloaded from there.

```
---- Step 1: read_data()

-- Contains the raw inputs
CREATE TABLE vocabulary(wrd STRING);

-- Assume text8.zip is uncompressed into /tmp/text8
COPY INTO vocabulary FROM '/tmp/text8' DELIMITERS '','' ' NULL AS 'thisstringdoesnotexist' BEST EFFORT;
SELECT count(*) from vocabulary; -- sanity check
```

```
---- Step 2: build the dictionary and replace rare words with UNK token
----  Basically, this is the SQL implementation of build_dataset()

-- This is the equivalend of the "count" list
CREATE TABLE word_cnt(
    idx INT GENERATED ALWAYS AS IDENTITY
        (START WITH 0 INCREMENT BY 1 MAXVALUE 49999 NO CYCLE),
```

[2] https://www.tensorflow.org/tutorials/representation/word2vec

```
    wrd STRING UNIQUE,
    cnt INT);

INSERT INTO word_cnt(wrd, cnt) VALUES ('UNK', -1);
-- Select 49999 most_common words, plus 'UNK', we have a Top-50000
INSERT INTO word_cnt(wrd, cnt)
  SELECT wrd, COUNT(wrd) AS cnt FROM vocabulary
  GROUP BY wrd ORDER BY cnt DESC LIMIT 49999;
-- => Python "dictionary" == word_cnt(wrd, idx)
-- => Python "reverse_dictionary" == word_cnt(idx, wrd)
-- => Python "count" == word_cnt(wrd, cnt)

-- Compute the index of each word in the list of selected words, or 0 otherwise
ALTER TABLE vocabulary ADD COLUMN idx INT DEFAULT 0;
UPDATE vocabulary SET idx = w.idx FROM word_cnt w WHERE vocabulary.wrd = w.wrd;
-- Sanity check: => 418391
SELECT COUNT(*) FROM vocabulary WHERE idx = 0;

-- Now we know the count of 'UNK' words
UPDATE word_cnt
  SET cnt = (SELECT COUNT(*) FROM vocabulary WHERE idx = 0) WHERE wrd = 'UNK';
-- Sanity check: => 418391
SELECT cnt FROM word_cnt WHERE wrd = 'UNK';
-- => Python "vocabulary" == vocabulary.wrd
-- => Python "data" == vocabulary.idx
```

```
---- Step 3: Function to generate a training batch for the skip-gram model.

DECLARE data_index INT;
SET data_index = 0;
DROP FUNCTION generate_batch;
CREATE FUNCTION generate_batch(data INT)
RETURNS TABLE(batch INT, labels INT) LANGUAGE PYTHON
{
  import numpy as np
  import collections
  import random

  # Use the loopback query feature of MonetDB/Python to implement Python
  #   "global" variable
  data_index  = int(list(_conn.execute("SELECT data_index;").values())[0][0])
  batch_size  = int(list(_conn.execute("SELECT batch_size;").values())[0][0])
  num_skips   = int(list(_conn.execute("SELECT num_skips;").values())[0][0])
  skip_window = int(list(_conn.execute("SELECT skip_window;").values())[0][0])

  assert batch_size % num_skips == 0
  assert num_skips <= 2 * skip_window
  batch = np.ndarray(shape=(batch_size), dtype=np.int32)
  labels = np.ndarray(shape=(batch_size), dtype=np.int32)
  span = 2 * skip_window + 1  # [ skip_window target skip_window ]
  buffer = collections.deque(maxlen=span)
  if data_index + span > len(data):
    data_index = 0
  buffer.extend(data[data_index:data_index + span])
  data_index += span
  for i in range(batch_size // num_skips):
    target = skip_window  # target label at the center of the buffer
    targets_to_avoid = [skip_window]
    for j in range(num_skips):
      while target in targets_to_avoid:
        target = random.randint(0, span - 1)
      targets_to_avoid.append(target)
      batch[i * num_skips + j] = buffer[skip_window]
      labels[i * num_skips + j] = buffer[target]
    if data_index == len(data):
```

```
      for word in data[:span]:
        buffer.append(word)
      data_index = span
    else:
      buffer.append(data[data_index])
      data_index += 1
  # Backtrack a little bit to avoid skipping words in the end of a batch
  data_index = (data_index + len(data) - span) % len(data)
  _conn.execute("SET data_index = {val};".format(val=data_index))

  # NOTE: the original python function returns batch as a horizontal array,
  #   labels as a vertical array. Here we return both as a horizontal array.
  return [batch, labels]
};
DECLARE batch_size INT, num_skips INT, skip_window INT;
SET batch_size = 8;
SET num_skips = 2;   -- How many times to reuse an input to generate a label
SET skip_window = 1; -- How many words to consider left and right.
SELECT * FROM generate_batch((SELECT idx FROM vocabulary)); -- => 169.640ms
```

```
---- Step 4 & 5: Build and train a skip-gram model,  and start training

CREATE FUNCTION compute_embeddings (dict_keys INT, dict_vals STRING)
RETURNS TABLE (embeddings BLOB) LANGUAGE PYTHON
{
  import tensorflow as tf
  import numpy as np
  import math

  # -----------------------------------------
  # Step 4: Build and train a skip-gram model
  embedding_size  = int(list(
        _conn.execute("SELECT embedding_size;").values())[0][0])
  vocabulary_size  = int(list(
        _conn.execute("SELECT vocabulary_size;").values())[0][0])
  batch_size  = int(list(_conn.execute("SELECT batch_size;").values())[0][0])
  num_skips   = int(list(_conn.execute("SELECT num_skips;").values())[0][0])
  skip_window = int(list(_conn.execute("SELECT skip_window;").values())[0][0])

  # FIXME: might want to declare these variables as SQL variables
  # We pick a random validation set to sample nearest neighbors. Here we limit the
  # validation samples to the words that have a low numeric ID, which by
  # construction are also the most frequent.
  valid_size = 16     # Random set of words to evaluate similarity on.
  valid_window = 100  # Only pick dev samples in the head of the distribution.
  valid_examples = np.random.choice(valid_window, valid_size, replace=False)
  num_sampled = 64    # Number of negative examples to sample.

  graph = tf.Graph()

  with graph.as_default():
    # Input data.
    train_inputs = tf.placeholder(tf.int32, shape=[batch_size])
    train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
    valid_dataset = tf.constant(valid_examples, dtype=tf.int32)

    # Ops and variables pinned to the CPU because of missing GPU implementation
    with tf.device('/cpu:0'):
      # Look up embeddings for inputs.
      embeddings = tf.Variable(
          tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0))
      embed = tf.nn.embedding_lookup(embeddings, train_inputs)

      # Construct the variables for the NCE loss
      nce_weights = tf.Variable(
```

```
        tf.truncated_normal([vocabulary_size, embedding_size],
                            stddev=1.0 / math.sqrt(embedding_size)))
    nce_biases = tf.Variable(tf.zeros([vocabulary_size]))

  # Compute the average NCE loss for the batch.
  # tf.nce_loss automatically draws a new sample of the negative labels each
  # time we evaluate the loss.
  loss = tf.reduce_mean(
      tf.nn.nce_loss(weights=nce_weights,
                     biases=nce_biases,
                     labels=train_labels,
                     inputs=embed,
                     num_sampled=num_sampled,
                     num_classes=vocabulary_size))

  # Construct the SGD optimizer using a learning rate of 1.0.
  optimizer = tf.train.GradientDescentOptimizer(1.0).minimize(loss)

  # Compute the cosine similarity between minibatch examples and all embeddings.
  norm = tf.sqrt(tf.reduce_sum(tf.square(embeddings), 1, keep_dims=True))
  normalized_embeddings = embeddings / norm
  valid_embeddings = tf.nn.embedding_lookup(
      normalized_embeddings, valid_dataset)
  similarity = tf.matmul(
      valid_embeddings, normalized_embeddings, transpose_b=True)

  # Add variable initializer.
  init = tf.global_variables_initializer()

# -----------------------------------------
# Step 5: Begin training
num_steps  = int(list(
      _conn.execute("SELECT num_steps;").values())[0][0])

with tf.Session(graph=graph) as session:
  # We must initialize all variables before we use them.
  init.run()

  # NOTE: replaced xrange of Python2 with range of Python3
  average_loss = 0
  for step in range(num_steps):
    # NOTE: the original generate_batch() is now an SQL function
    res = _conn.execute("SELECT * FROM generate_batch((SELECT idx FROM vocabulary));")
    batch_inputs = list(res['batch'])
    batch_labels = np.array([list(res['labels'])]).transpose()
    feed_dict = {train_inputs: batch_inputs, train_labels: batch_labels}

    # We perform one update step by evaluating the optimizer op (including it
    # in the list of returned values for session.run()
    _, loss_val = session.run([optimizer, loss], feed_dict=feed_dict)
    average_loss += loss_val

    if step % 2000 == 0:
      if step > 0:
        average_loss /= 2000
      # The average loss is an estimate of the loss over the last 2000 batches.
      print('Average loss at step ', step, ': ', average_loss)
      average_loss = 0

    # Construct the reverse_dictionary from database data
    reverse_dictionary = dict(zip(dict_keys, dict_vals))
    # Note that this is expensive (~20% slowdown if computed every 500 steps)
    if step % 10000 == 0:
      sim = similarity.eval()
      for i in range(valid_size):
```

```
          valid_word = reverse_dictionary[valid_examples[i]]
          top_k = 8  # number of nearest neighbors
          nearest = (-sim[i, :]).argsort()[1:top_k + 1]
          log_str = 'Nearest to %s:' % valid_word
          for k in range(top_k):
            close_word = reverse_dictionary[nearest[k]]
            log_str = '%s %s,' % (log_str, close_word)
          print(log_str)
    final_embeddings = normalized_embeddings.eval()

  res = list()
  for row in final_embeddings[:]:
    res.append(row.tobytes())
  return res
};
```

```
---- Finally, set up the SQL environment and use the above functions to do the job.

DECLARE embedding_size INT, vocabulary_size INT, num_steps INT;
SET embedding_size = 128; -- Dimension of the embedding vector.
SET vocabulary_size = 50000;
SET num_steps = 100001;
SET batch_size = 128;
SET num_skips = 2;   -- How many times to reuse an input to generate a label
SET skip_window = 1; -- How many words to consider left and right.
CREATE TABLE embeddings (embd BLOB);
INSERT INTO embeddings
  SELECT * FROM compute_embeddings( (SELECT idx, wrd FROM word_cnt) );

-- A simple example of how to return an M x N matrix as a column of BLOBs
CREATE FUNCTION b()
RETURNS TABLE(b BLOB) LANGUAGE PYTHON
{
  import numpy as np
  ary = np.array([[ 0.14792269,  0.27395913, -0.1010012 ],
         [ 0.10359713,  0.03716619, -0.16007   ],
         [ 0.0925912 ,  0.18409029,  0.05768026],
         [ 0.09125206,  0.12109927, -0.06208262],
         [ 0.17739627,  0.10868212, -0.04809754]])
  res = list()
  for row in ary[:]:
    res.append(row.tobytes())
  return res
};
SELECT * FROM b();
```